FORM

4.2.0

Generated by Doxygen 1.9.8

1 Data Structure Index 1
1.1 Data Structures . . . 1

2 File Index 3
2.1 File List . . . 3

3 Data Structure Documentation 5
3.1 AllGlobals Struct Reference . . . 5
3.1.1 Detailed Description . . . 6
3.1.2 Field Documentation . . . 6
3.1.2.1 M . . . 6
3.1.2.2 C . . . 6
3.1.2.3 S . . . 6
3.1.2.4 R . . . 6
3.1.2.5 N . . . 7
3.1.2.6 O . . . 7
3.1.2.7 P . . . 7
3.1.2.8 T . . . 7
3.1.2.9 X . . . 7
3.2 ARGBUFFER Struct Reference . . . 7
3.2.1 Detailed Description . . . 8
3.2.2 Field Documentation . . . 8
3.2.2.1 buffer . . . 8
3.2.2.2 dollar . . . 8
3.2.2.3 size . . . 8
3.2.2.4 type . . . 8
3.2.2.5 dummy . . . 8
3.3 bit_field Struct Reference . . . 9
3.3.1 Detailed Description . . . 9
3.3.2 Field Documentation . . . 9
3.3.2.1 bit_0 . . . 9
3.3.2.2 bit_1 . . . 9
3.3.2.3 bit_2 . . . 9
3.3.2.4 bit_3 . . . 9
3.3.2.5 bit_4 . . . 10
3.3.2.6 bit_5 . . . 10
3.3.2.7 bit_6 . . . 10
3.3.2.8 bit_7 . . . 10
3.4 BrAcKeTiNdEx Struct Reference . . . 10
3.4.1 Detailed Description . . . 11
3.4.2 Field Documentation . . . 11
3.4.2.1 start . . . 11
3.4.2.2 next . . . 11

3.4.2.3 bracket . . . 11
3.4.2.4 termsinbracket . . . 11
3.5 BrAcKeTiNfO Struct Reference . . . 12
3.5.1 Detailed Description . . . 12
3.5.2 Field Documentation . . . 12
3.5.2.1 indexbuffer . . . 12
3.5.2.2 bracketbuffer . . . 13
3.5.2.3 bracketbuffersize . . . 13
3.5.2.4 indexbuffersize . . . 13
3.5.2.5 bracketfill . . . 13
3.5.2.6 indexfill . . . 13
3.5.2.7 SortType . . . 13
3.6 bufIPstruct Struct Reference . . . 14
3.6.1 Detailed Description . . . 14
3.6.2 Field Documentation . . . 14
3.6.2.1 i . . . 14
3.6.2.2 e . . . 15
3.7 C_const Struct Reference . . . 15
3.7.1 Detailed Description . . . 19
3.7.2 Field Documentation . . . 19
3.7.2.1 separators . . . 19
3.7.2.2 StoreFileSize . . . 19
3.7.2.3 dollarnames . . . 19
3.7.2.4 exprnames . . . 20
3.7.2.5 varnames . . . 20
3.7.2.6 ChannelList . . . 20
3.7.2.7 DubiousList . . . 20
3.7.2.8 FunctionList . . . 20
3.7.2.9 ExpressionList . . . 20
3.7.2.10 IndexList . . . 21
3.7.2.11 SetElementList . . . 21
3.7.2.12 SetList . . . 21
3.7.2.13 SymbolList . . . 21
3.7.2.14 VectorList . . . 21
3.7.2.15 PotModDolList . . . 21
3.7.2.16 ModOptDolList . . . 22
3.7.2.17 TableBaseList . . . 22
3.7.2.18 cbufList . . . 22
3.7.2.19 AutoSymbolList . . . 22
3.7.2.20 AutoIndexList . . . 22
3.7.2.21 AutoVectorList . . . 22
3.7.2.22 AutoFunctionList . . . 22

3.7.2.23 autonames . . . 23
3.7.2.24 Symbols . . . 23
3.7.2.25 Indices . . . 23
3.7.2.26 Vectors . . . 23
3.7.2.27 Functions . . . 23
3.7.2.28 activenames . . . 23
3.7.2.29 Streams . . . 23
3.7.2.30 CurrentStream . . . 24
3.7.2.31 termstack . . . 24
3.7.2.32 termsortstack . . . 24
3.7.2.33 cmod . . . 24
3.7.2.34 powmod . . . 24
3.7.2.35 modpowers . . . 24
3.7.2.36 halfmod . . . 25
3.7.2.37 ProtoType . . . 25
3.7.2.38 WildC . . . 25
3.7.2.39 IfHeap . . . 25
3.7.2.40 IfCount . . . 25
3.7.2.41 IfStack . . . 25
3.7.2.42 iBuffer . . . 25
3.7.2.43 iPointer . . . 26
3.7.2.44 iStop . . . 26
3.7.2.45 LabelNames . . . 26
3.7.2.46 FixIndices . . . 26
3.7.2.47 termsumcheck . . . 26
3.7.2.48 WildcardNames . . . 26
3.7.2.49 Labels . . . 27
3.7.2.50 tokens . . . 27
3.7.2.51 toptokens . . . 27
3.7.2.52 endoftokens . . . 27
3.7.2.53 tokenarglevel . . . 27
3.7.2.54 modinverses . . . 27
3.7.2.55 Fortran90Kind . . . 28
3.7.2.56 MultiBracketBuf . . . 28
3.7.2.57 extrasym . . . 28
3.7.2.58 doloopstack . . . 28
3.7.2.59 doloopnest . . . 28
3.7.2.60 CheckpointRunAfter . . . 28
3.7.2.61 CheckpointRunBefore . . . 28
3.7.2.62 IfSumCheck . . . 29
3.7.2.63 CommuteInSet . . . 29
3.7.2.64 TestValue . . . 29

3.7.2.65 argstack . . . 29
3.7.2.66 insidestack . . . 29
3.7.2.67 inexprstack . . . 29
3.7.2.68 iBufferSize . . . 29
3.7.2.69 TransEname . . . 30
3.7.2.70 ProcessBucketSize . . . 30
3.7.2.71 mProcessBucketSize . . . 30
3.7.2.72 CModule . . . 30
3.7.2.73 ThreadBucketSize . . . 30
3.7.2.74 CheckpointStamp . . . 30
3.7.2.75 CheckpointInterval . . . 30
3.7.2.76 cbufnum . . . 31
3.7.2.77 AutoDeclareFlag . . . 31
3.7.2.78 NoShowInput . . . 31
3.7.2.79 ShortStats . . . 31
3.7.2.80 compiletype . . . 31
3.7.2.81 firstconstindex . . . 31
3.7.2.82 insidefirst . . . 31
3.7.2.83 minsidefirst . . . 32
3.7.2.84 wildflag . . . 32
3.7.2.85 NumLabels . . . 32
3.7.2.86 MaxLabels . . . 32
3.7.2.87 lDefDim . . . 32
3.7.2.88 lDefDim4 . . . 32
3.7.2.89 NumWildcardNames . . . 32
3.7.2.90 WildcardBufferSize . . . 32
3.7.2.91 MaxIf . . . 33
3.7.2.92 NumStreams . . . 33
3.7.2.93 MaxNumStreams . . . 33
3.7.2.94 firstctypemessage . . . 33
3.7.2.95 tablecheck . . . 33
3.7.2.96 idoption . . . 33
3.7.2.97 BottomLevel . . . 33
3.7.2.98 CompileLevel . . . 33
3.7.2.99 TokensWriteFlag . . . 34
3.7.2.100 UnsureDollarMode . . . 34
3.7.2.101 outsidefun . . . 34
3.7.2.102 funpowers . . . 34
3.7.2.103 WarnFlag . . . 34
3.7.2.104 StatsFlag . . . 34
3.7.2.105 NamesFlag . . . 34
3.7.2.106 CodesFlag . . . 34

3.7.2.107 SetupFlag . . . 35
3.7.2.108 SortType . . . 35
3.7.2.109 lSortType . . . 35
3.7.2.110 ThreadStats . . . 35
3.7.2.111 FinalStats . . . 35
3.7.2.112 OldParallelStats . . . 35
3.7.2.113 ThreadsFlag . . . 35
3.7.2.114 ThreadBalancing . . . 35
3.7.2.115 ThreadSortFileSynch . . . 36
3.7.2.116 ProcessStats . . . 36
3.7.2.117 BracketNormalize . . . 36
3.7.2.118 maxtermlevel . . . 36
3.7.2.119 dumnumflag . . . 36
3.7.2.120 bracketindexflag . . . 36
3.7.2.121 parallelflag . . . 36
3.7.2.122 mparallelflag . . . 36
3.7.2.123 inparallelflag . . . 37
3.7.2.124 partodoflag . . . 37
3.7.2.125 properorderflag . . . 37
3.7.2.126 vetofilling . . . 37
3.7.2.127 tablefilling . . . 37
3.7.2.128 vetotablebasefill . . . 37
3.7.2.129 exprfillwarning . . . 37
3.7.2.130 lhdollarflag . . . 37
3.7.2.131 NoCompress . . . 38
3.7.2.132 IsFortran90 . . . 38
3.7.2.133 MultiBracketLevels . . . 38
3.7.2.134 topolynomialflag . . . 38
3.7.2.135 ffbufnum . . . 38
3.7.2.136 OldFactArgFlag . . . 38
3.7.2.137 MemDebugFlag . . . 38
3.7.2.138 OldGCDflag . . . 38
3.7.2.139 WTimeStatsFlag . . . 39
3.7.2.140 doloopstacksize . . . 39
3.7.2.141 dolooplevel . . . 39
3.7.2.142 CheckpointFlag . . . 39
3.7.2.143 SizeCommuteInSet . . . 39
3.7.2.144 origin . . . 39
3.7.2.145 vectorlikeLHS . . . 39
3.7.2.146 argsumcheck . . . 40
3.7.2.147 insidesumcheck . . . 40
3.7.2.148 inexprsumcheck . . . 40

3.7.2.149 RepSumCheck . . . 40
3.7.2.150 lUniTrace . . . 40
3.7.2.151 RepLevel . . . 40
3.7.2.152 arglevel . . . 40
3.7.2.153 insidelevel . . . 40
3.7.2.154 inexprlevel . . . 41
3.7.2.155 termlevel . . . 41
3.7.2.156 MustTestTable . . . 41
3.7.2.157 DumNum . . . 41
3.7.2.158 ncmod . . . 41
3.7.2.159 npowmod . . . 41
3.7.2.160 modmode . . . 41
3.7.2.161 nhalfmod . . . 41
3.7.2.162 DirtPow . . . 42
3.7.2.163 lUnitTrace . . . 42
3.7.2.164 NwildC . . . 42
3.7.2.165 ComDefer . . . 42
3.7.2.166 CollectFun . . . 42
3.7.2.167 AltCollectFun . . . 42
3.7.2.168 OutputMode . . . 42
3.7.2.169 Cnumpows . . . 42
3.7.2.170 OutputSpaces . . . 43
3.7.2.171 OutNumberType . . . 43
3.7.2.172 DidClean . . . 43
3.7.2.173 IfLevel . . . 43
3.7.2.174 WhileLevel . . . 43
3.7.2.175 LogHandle . . . 43
3.7.2.176 LineLength . . . 43
3.7.2.177 StoreHandle . . . 43
3.7.2.178 HideLevel . . . 44
3.7.2.179 lPolyFun . . . 44
3.7.2.180 lPolyFunInv . . . 44
3.7.2.181 lPolyFunType . . . 44
3.7.2.182 lPolyFunExp . . . 44
3.7.2.183 lPolyFunVar . . . 44
3.7.2.184 lPolyFunPow . . . 44
3.7.2.185 SymChangeFlag . . . 44
3.7.2.186 CollectPercentage . . . 45
3.7.2.187 ShortStatsMax . . . 45
3.7.2.188 extrasymbols . . . 45
3.7.2.189 PolyRatFunChanged . . . 45
3.7.2.190 ToBeInFactors . . . 45

3.7.2.191 InnerTest . . . 45
3.7.2.192 Commercial . . . 45
3.7.2.193 debugFlags . . . 46
3.8 CbUf Struct Reference . . . 46
3.8.1 Detailed Description . . . 47
3.8.2 Field Documentation . . . 47
3.8.2.1 Buffer . . . 47
3.8.2.2 Top . . . 47
3.8.2.3 Pointer . . . 47
3.8.2.4 lhs . . . 47
3.8.2.5 rhs . . . 48
3.8.2.6 CanCommu . . . 48
3.8.2.7 NumTerms . . . 48
3.8.2.8 numdum . . . 48
3.8.2.9 dimension . . . 48
3.8.2.10 boomlijst . . . 49
3.8.2.11 BufferSize . . . 49
3.8.2.12 numlhs . . . 49
3.8.2.13 numrhs . . . 49
3.8.2.14 maxlhs . . . 49
3.8.2.15 maxrhs . . . 49
3.8.2.16 mnumlhs . . . 49
3.8.2.17 mnumrhs . . . 50
3.8.2.18 numtree . . . 50
3.8.2.19 rootnum . . . 50
3.8.2.20 MaxTreeSize . . . 50
3.9 ChAnNeL Struct Reference . . . 50
3.9.1 Detailed Description . . . 50
3.9.2 Field Documentation . . . 51
3.9.2.1 name . . . 51
3.9.2.2 handle . . . 51
3.10 COST Struct Reference . . . 51
3.10.1 Detailed Description . . . 51
3.10.2 Field Documentation . . . 51
3.10.2.1 add . . . 51
3.10.2.2 mul . . . 52
3.10.2.3 div . . . 52
3.10.2.4 pow . . . 52
3.11 dbase Struct Reference . . . 52
3.11.1 Detailed Description . . . 53
3.11.2 Field Documentation . . . 53
3.11.2.1 info . . . 53

3.11.2.2 mode . . . 53
3.11.2.3 tablenamessize . . . 53
3.11.2.4 topnumber . . . 53
3.11.2.5 tablenamefill . . . 54
3.11.2.6 iblocks . . . 54
3.11.2.7 nblocks . . . 54
3.11.2.8 handle . . . 54
3.11.2.9 name . . . 54
3.11.2.10 fullname . . . 54
3.11.2.11 tablenames . . . 54
3.12 DICTIONARY Struct Reference . . . 55
3.12.1 Detailed Description . . . 55
3.12.2 Field Documentation . . . 55
3.12.2.1 elements . . . 55
3.12.2.2 name . . . 55
3.12.2.3 sizeelements . . . 56
3.12.2.4 numelements . . . 56
3.12.2.5 numbers . . . 56
3.12.2.6 variables . . . 56
3.12.2.7 characters . . . 56
3.12.2.8 funwith . . . 56
3.12.2.9 gnumelements . . . 56
3.12.2.10 ranges . . . 57
3.13 DICTIONARY_ELEMENT Struct Reference . . . 57
3.13.1 Detailed Description . . . 57
3.13.2 Field Documentation . . . 57
3.13.2.1 lhs . . . 57
3.13.2.2 rhs . . . 57
3.13.2.3 type . . . 57
3.13.2.4 size . . . 58
3.14 DiStRiBuTe Struct Reference . . . 58
3.14.1 Detailed Description . . . 58
3.14.2 Field Documentation . . . 58
3.14.2.1 obj1 . . . 58
3.14.2.2 obj2 . . . 58
3.14.2.3 out . . . 59
3.14.2.4 sign . . . 59
3.14.2.5 n1 . . . 59
3.14.2.6 n2 . . . 59
3.14.2.7 n . . . 59
3.14.2.8 cycle . . . 59
3.15 dollar_buf Struct Reference . . . 59

3.15.1 Detailed Description . . . 59
3.16 DoLIArS Struct Reference . . . 60
3.16.1 Detailed Description . . . 60
3.16.2 Field Documentation . . . 60
3.16.2.1 where . . . 60
3.16.2.2 factors . . . 60
3.16.2.3 size . . . 61
3.16.2.4 name . . . 61
3.16.2.5 type . . . 61
3.16.2.6 node . . . 61
3.16.2.7 index . . . 61
3.16.2.8 zero . . . 61
3.16.2.9 numdummies . . . 61
3.16.2.10 nfactors . . . 62
3.17 DoLoOp Struct Reference . . . 62
3.17.1 Detailed Description . . . 63
3.17.2 Field Documentation . . . 63
3.17.2.1 p . . . 63
3.17.2.2 name . . . 63
3.17.2.3 vars . . . 63
3.17.2.4 contents . . . 63
3.17.2.5 dollarname . . . 63
3.17.2.6 startlinenumber . . . 64
3.17.2.7 firstnum . . . 64
3.17.2.8 lastnum . . . 64
3.17.2.9 incnum . . . 64
3.17.2.10 type . . . 64
3.17.2.11 NoShowInput . . . 64
3.17.2.12 errorsinloop . . . 64
3.17.2.13 firstloopcall . . . 64
3.17.2.14 firstdollar . . . 65
3.17.2.15 lastdollar . . . 65
3.17.2.16 incdollar . . . 65
3.17.2.17 NumPreTypes . . . 65
3.17.2.18 PreIfLevel . . . 65
3.17.2.19 PreSwitchLevel . . . 65
3.18 DuBiOuS Struct Reference . . . 65
3.18.1 Detailed Description . . . 66
3.18.2 Field Documentation . . . 66
3.18.2.1 name . . . 66
3.18.2.2 node . . . 66
3.18.2.3 dummy . . . 66

3.19 ExPrEsSiOn Struct Reference . . . 67
3.19.1 Detailed Description . . . 68
3.19.2 Field Documentation . . . 68
3.19.2.1 onfile . . . 68
3.19.2.2 prototype . . . 68
3.19.2.3 size . . . 68
3.19.2.4 renum . . . 68
3.19.2.5 bracketinfo . . . 68
3.19.2.6 newbracketinfo . . . 69
3.19.2.7 renumlists . . . 69
3.19.2.8 inmem . . . 69
3.19.2.9 counter . . . 69
3.19.2.10 name . . . 69
3.19.2.11 hidelevel . . . 69
3.19.2.12 vflags . . . 69
3.19.2.13 printflag . . . 70
3.19.2.14 status . . . 70
3.19.2.15 replace . . . 70
3.19.2.16 node . . . 70
3.19.2.17 whichbuffer . . . 70
3.19.2.18 namesize . . . 70
3.19.2.19 compression . . . 70
3.19.2.20 numdummies . . . 70
3.19.2.21 numfactors . . . 71
3.19.2.22 sizeprototype . . . 71
3.20 FaCdOlLaR Struct Reference . . . 71
3.20.1 Detailed Description . . . 71
3.20.2 Field Documentation . . . 71
3.20.2.1 where . . . 71
3.20.2.2 size . . . 71
3.20.2.3 type . . . 72
3.20.2.4 value . . . 72
3.21 factorized_poly Class Reference . . . 72
3.21.1 Detailed Description . . . 72
3.21.2 Field Documentation . . . 72
3.21.2.1 factor . . . 72
3.21.2.2 power . . . 73
3.22 FiLe Struct Reference . . . 73
3.22.1 Detailed Description . . . 74
3.22.2 Field Documentation . . . 74
3.22.2.1 POposition . . . 74
3.22.2.2 filesize . . . 74

3.22.2.3 PObuffer . . . 74
3.22.2.4 POstop . . . 74
3.22.2.5 POfill . . . 74
3.22.2.6 POfull . . . 74
3.22.2.7 name . . . 75
3.22.2.8 numblocks . . . 75
3.22.2.9 inbuffer . . . 75
3.22.2.10 POsize . . . 75
3.22.2.11 handle . . . 75
3.22.2.12 active . . . 75
3.23 FiLeDaTa Struct Reference . . . 76
3.23.1 Detailed Description . . . 76
3.23.2 Field Documentation . . . 77
3.23.2.1 Index . . . 77
3.23.2.2 Fill . . . 77
3.23.2.3 Position . . . 77
3.23.2.4 Handle . . . 77
3.23.2.5 dirtyflag . . . 77
3.24 FiLeInDeX Struct Reference . . . 78
3.24.1 Detailed Description . . . 78
3.24.2 Field Documentation . . . 79
3.24.2.1 next . . . 79
3.24.2.2 number . . . 79
3.24.2.3 expression . . . 79
3.24.2.4 empty . . . 79
3.25 FixedGlobals Struct Reference . . . 79
3.25.1 Detailed Description . . . 80
3.25.2 Field Documentation . . . 80
3.25.2.1 Operation . . . 80
3.25.2.2 OperaFind . . . 80
3.25.2.3 VarType . . . 80
3.25.2.4 ExprStat . . . 80
3.25.2.5 FunNam . . . 81
3.25.2.6 swmes . . . 81
3.25.2.7 fname . . . 81
3.25.2.8 fname2 . . . 81
3.25.2.9 s_one . . . 81
3.25.2.10 fnamebase . . . 81
3.25.2.11 fname2base . . . 81
3.25.2.12 cTable . . . 82
3.26 fixedset Struct Reference . . . 82
3.26.1 Detailed Description . . . 82

3.26.2 Field Documentation . . . 82
3.26.2.1 name . . . 82
3.26.2.2 description . . . 82
3.26.2.3 type . . . 82
3.26.2.4 dimension . . . 83
3.27 FUN_INFO Struct Reference . . . 83
3.27.1 Detailed Description . . . 83
3.27.2 Field Documentation . . . 83
3.27.2.1 location . . . 83
3.27.2.2 numargs . . . 83
3.27.2.3 numfunnies . . . 84
3.27.2.4 numwildcards . . . 84
3.27.2.5 symmet . . . 84
3.27.2.6 tensor . . . 84
3.27.2.7 commute . . . 84
3.28 FuNcTiOn Struct Reference . . . 85
3.28.1 Detailed Description . . . 85
3.28.2 Field Documentation . . . 86
3.28.2.1 tabl . . . 86
3.28.2.2 symminfo . . . 86
3.28.2.3 name . . . 86
3.28.2.4 commute . . . 86
3.28.2.5 complex . . . 86
3.28.2.6 number . . . 87
3.28.2.7 flags . . . 87
3.28.2.8 spec . . . 87
3.28.2.9 symmetric . . . 87
3.28.2.10 node . . . 87
3.28.2.11 namesize . . . 88
3.28.2.12 dimension . . . 88
3.28.2.13 maxnumargs . . . 88
3.28.2.14 minnumargs . . . 88
3.29 gcd_heuristic_failed Class Reference . . . 88
3.29.1 Detailed Description . . . 88
3.30 HANDLERS Struct Reference . . . 88
3.30.1 Detailed Description . . . 89
3.30.2 Field Documentation . . . 89
3.30.2.1 newlogonly . . . 89
3.30.2.2 newhandle . . . 89
3.30.2.3 oldhandle . . . 89
3.30.2.4 oldlogonly . . . 89
3.30.2.5 oldprinttype . . . 89

3.30.2.6 oldsilent . . . 90
3.31 InDeX Struct Reference . . . 90
3.31.1 Detailed Description . . . 90
3.31.2 Field Documentation . . . 90
3.31.2.1 name . . . 90
3.31.2.2 type . . . 90
3.31.2.3 dimension . . . 90
3.31.2.4 number . . . 91
3.31.2.5 flags . . . 91
3.31.2.6 nmin4 . . . 91
3.31.2.7 node . . . 91
3.31.2.8 namesize . . . 91
3.32 indexblock Struct Reference . . . 91
3.32.1 Detailed Description . . . 92
3.32.2 Field Documentation . . . 92
3.32.2.1 flags . . . 92
3.32.2.2 previousblock . . . 92
3.32.2.3 position . . . 92
3.32.2.4 objects . . . 92
3.33 InDeXeNtRy Struct Reference . . . 93
3.33.1 Detailed Description . . . 93
3.33.2 Field Documentation . . . 94
3.33.2.1 position . . . 94
3.33.2.2 length . . . 94
3.33.2.3 variables . . . 94
3.33.2.4 CompressSize . . . 94
3.33.2.5 nsymbols . . . 94
3.33.2.6 nindices . . . 95
3.33.2.7 nvectors . . . 95
3.33.2.8 nfunctions . . . 95
3.33.2.9 size . . . 95
3.33.2.10 name . . . 95
3.34 iniinfo Struct Reference . . . 96
3.34.1 Detailed Description . . . 96
3.34.2 Field Documentation . . . 96
3.34.2.1 entriesinindex . . . 96
3.34.2.2 numberofindexblocks . . . 96
3.34.2.3 firstindexblock . . . 96
3.34.2.4 lastindexblock . . . 96
3.34.2.5 numberoftables . . . 97
3.34.2.6 numberofnamesblocks . . . 97
3.34.2.7 firstnameblock . . . 97

3.34.2.8 lastnameblock . . . 97
3.35 INSIDEINFO Struct Reference . . . 97
3.35.1 Detailed Description . . . 98
3.35.2 Field Documentation . . . 98
3.35.2.1 buffer . . . 98
3.35.2.2 oldcompiletype . . . 98
3.35.2.3 oldparallelflag . . . 98
3.35.2.4 oldnumpotmoddollars . . . 98
3.35.2.5 size . . . 98
3.35.2.6 numdollars . . . 98
3.35.2.7 oldcbuf . . . 99
3.35.2.8 oldrbuf . . . 99
3.35.2.9 inscbuf . . . 99
3.35.2.10 oldcnumlhs . . . 99
3.36 KEYWORD Struct Reference . . . 99
3.36.1 Detailed Description . . . 99
3.36.2 Field Documentation . . . 100
3.36.2.1 name . . . 100
3.36.2.2 func . . . 100
3.36.2.3 type . . . 100
3.36.2.4 flags . . . 100
3.37 LIST Struct Reference . . . 100
3.37.1 Detailed Description . . . 101
3.37.2 Field Documentation . . . 101
3.37.2.1 lijst . . . 101
3.37.2.2 message . . . 101
3.37.2.3 num . . . 101
3.37.2.4 maxnum . . . 101
3.37.2.5 size . . . 101
3.37.2.6 numglobal . . . 102
3.37.2.7 numtemp . . . 102
3.37.2.8 numclear . . . 102
3.38 longMultiStruct Struct Reference . . . 102
3.38.1 Detailed Description . . . 103
3.38.2 Field Documentation . . . 103
3.38.2.1 buffer . . . 103
3.38.2.2 bufpos . . . 103
3.38.2.3 packpos . . . 103
3.38.2.4 nPacks . . . 103
3.38.2.5 lastLen . . . 103
3.38.2.6 next . . . 103
3.39 M_const Struct Reference . . . 104

3.39.1 Detailed Description . . . 107
3.39.2 Field Documentation . . . 107
3.39.2.1 zeropos . . . 107
3.39.2.2 S0 . . . 108
3.39.2.3 gcmod . . . 108
3.39.2.4 gpowmod . . . 108
3.39.2.5 TempDir . . . 108
3.39.2.6 TempSortDir . . . 108
3.39.2.7 IncDir . . . 108
3.39.2.8 InputFileName . . . 108
3.39.2.9 LogFileName . . . 109
3.39.2.10 OutBuffer . . . 109
3.39.2.11 Path . . . 109
3.39.2.12 SetupDir . . . 109
3.39.2.13 SetupFile . . . 109
3.39.2.14 gFortran90Kind . . . 109
3.39.2.15 gextrasym . . . 109
3.39.2.16 ggextrasym . . . 109
3.39.2.17 oldnumextrasymbols . . . 110
3.39.2.18 SpectatorFiles . . . 110
3.39.2.19 MaxTer . . . 110
3.39.2.20 CompressSize . . . 110
3.39.2.21 ScratSize . . . 110
3.39.2.22 HideSize . . . 110
3.39.2.23 SizeStoreCache . . . 110
3.39.2.24 MaxStreamSize . . . 110
3.39.2.25 SIOsize . . . 111
3.39.2.26 SLargeSize . . . 111
3.39.2.27 SSmallEsize . . . 111
3.39.2.28 SSmallSize . . . 111
3.39.2.29 STermsInSmall . . . 111
3.39.2.30 MaxBracketBufferSize . . . 111
3.39.2.31 hProcessBucketSize . . . 111
3.39.2.32 gProcessBucketSize . . . 111
3.39.2.33 shmWinSize . . . 112
3.39.2.34 OldChildTime . . . 112
3.39.2.35 OldSecTime . . . 112
3.39.2.36 OldMilliTime . . . 112
3.39.2.37 WorkSize . . . 112
3.39.2.38 gThreadBucketSize . . . 112
3.39.2.39 ggThreadBucketSize . . . 112
3.39.2.40 SumTime . . . 112

3.39.2.41 SpectatorSize . . . 113
3.39.2.42 FileOnlyFlag . . . 113
3.39.2.43 Interact . . . 113
3.39.2.44 MaxParLevel . . . 113
3.39.2.45 OutBufSize . . . 113
3.39.2.46 SMaxFpatches . . . 113
3.39.2.47 SMaxPatches . . . 113
3.39.2.48 StdOut . . . 113
3.39.2.49 ginsidefirst . . . 114
3.39.2.50 gDefDim . . . 114
3.39.2.51 gDefDim4 . . . 114
3.39.2.52 NumFixedSets . . . 114
3.39.2.53 NumFixedFunctions . . . 114
3.39.2.54 rbufnum . . . 114
3.39.2.55 dbufnum . . . 114
3.39.2.56 sbufnum . . . 114
3.39.2.57 zbufnum . . . 115
3.39.2.58 SkipClears . . . 115
3.39.2.59 gTokensWriteFlag . . . 115
3.39.2.60 gfunpowers . . . 115
3.39.2.61 gStatsFlag . . . 115
3.39.2.62 gNamesFlag . . . 115
3.39.2.63 gCodesFlag . . . 115
3.39.2.64 gSortType . . . 115
3.39.2.65 gproperorderflag . . . 116
3.39.2.66 hparallelflag . . . 116
3.39.2.67 gparallelflag . . . 116
3.39.2.68 totalnumberofthreads . . . 116
3.39.2.69 gSizeCommuteInSet . . . 116
3.39.2.70 gThreadStats . . . 116
3.39.2.71 ggThreadStats . . . 116
3.39.2.72 gFinalStats . . . 116
3.39.2.73 ggFinalStats . . . 117
3.39.2.74 gThreadsFlag . . . 117
3.39.2.75 ggThreadsFlag . . . 117
3.39.2.76 gThreadBalancing . . . 117
3.39.2.77 ggThreadBalancing . . . 117
3.39.2.78 gThreadSortFileSynch . . . 117
3.39.2.79 ggThreadSortFileSynch . . . 117
3.39.2.80 gProcessStats . . . 117
3.39.2.81 ggProcessStats . . . 118
3.39.2.82 gOldParallelStats . . . 118

3.39.2.83 ggOldParallelStats . . . 118
3.39.2.84 maxFlevels . . . 118
3.39.2.85 resetTimeOnClear . . . 118
3.39.2.86 gcNumDollars . . . 118
3.39.2.87 MultiRun . . . 118
3.39.2.88 gNoSpacesInNumbers . . . 118
3.39.2.89 ggNoSpacesInNumbers . . . 119
3.39.2.90 gIsFortran90 . . . 119
3.39.2.91 PrintTotalSize . . . 119
3.39.2.92 fbuffersize . . . 119
3.39.2.93 gOldFactArgFlag . . . 119
3.39.2.94 ggOldFactArgFlag . . . 119
3.39.2.95 gnumextrasym . . . 119
3.39.2.96 ggnumextrasym . . . 119
3.39.2.97 NumSpectatorFiles . . . 120
3.39.2.98 SizeForSpectatorFiles . . . 120
3.39.2.99 gOldGCDflag . . . 120
3.39.2.100 ggOldGCDflag . . . 120
3.39.2.101 gWTimeStatsFlag . . . 120
3.39.2.102 ggWTimeStatsFlag . . . 120
3.39.2.103 MaxTal . . . 120
3.39.2.104 IndDum . . . 120
3.39.2.105 DumInd . . . 121
3.39.2.106 WilInd . . . 121
3.39.2.107 gncmod . . . 121
3.39.2.108 gnpowmod . . . 121
3.39.2.109 gmodmode . . . 121
3.39.2.110 gUnitTrace . . . 121
3.39.2.111 gOutputMode . . . 121
3.39.2.112 gOutputSpaces . . . 121
3.39.2.113 gOutNumberType . . . 122
3.39.2.114 gCnumpows . . . 122
3.39.2.115 gUniTrace . . . 122
3.39.2.116 MaxWildcards . . . 122
3.39.2.117 mTraceDum . . . 122
3.39.2.118 OffsetIndex . . . 122
3.39.2.119 OffsetVector . . . 122
3.39.2.120 RepMax . . . 122
3.39.2.121 LogType . . . 123
3.39.2.122 ggStatsFlag . . . 123
3.39.2.123 gLineLength . . . 123
3.39.2.124 qError . . . 123

3.39.2.125 FortranCont . . . 123
3.39.2.126 HoldFlag . . . 123
3.39.2.127 Ordering . . . 123
3.39.2.128 silent . . . 123
3.39.2.129 tracebackflag . . . 124
3.39.2.130 expnum . . . 124
3.39.2.131 denomnum . . . 124
3.39.2.132 facnum . . . 124
3.39.2.133 invfacnum . . . 124
3.39.2.134 sumnum . . . 124
3.39.2.135 sumpnum . . . 124
3.39.2.136 OldOrderFlag . . . 124
3.39.2.137 termfunnum . . . 125
3.39.2.138 matchfunnum . . . 125
3.39.2.139 countfunnum . . . 125
3.39.2.140 gPolyFun . . . 125
3.39.2.141 gPolyFunInv . . . 125
3.39.2.142 gPolyFunType . . . 125
3.39.2.143 gPolyFunExp . . . 125
3.39.2.144 gPolyFunVar . . . 125
3.39.2.145 gPolyFunPow . . . 126
3.39.2.146 dollarzero . . . 126
3.39.2.147 atstartup . . . 126
3.39.2.148 exitflag . . . 126
3.39.2.149 NumStoreCaches . . . 126
3.39.2.150 gIndentSpace . . . 126
3.39.2.151 ggIndentSpace . . . 126
3.39.2.152 gShortStatsMax . . . 127
3.39.2.153 ggShortStatsMax . . . 127
3.39.2.154 gextrasymbols . . . 127
3.39.2.155 ggextrasymbols . . . 127
3.39.2.156 zerorhs . . . 127
3.39.2.157 onerhs . . . 127
3.39.2.158 havesortdir . . . 127
3.39.2.159 BracketFactors . . . 128
3.40 MiNmAx Struct Reference . . . 128
3.40.1 Detailed Description . . . 128
3.40.2 Field Documentation . . . 128
3.40.2.1 mini . . . 128
3.40.2.2 maxi . . . 128
3.40.2.3 size . . . 129
3.41 MODNUM Struct Reference . . . 129

3.41.1 Detailed Description . . . 129
3.41.2 Field Documentation . . . 129
3.41.2.1 a . . . 129
3.41.2.2 m . . . 129
3.41.2.3 na . . . 129
3.41.2.4 nm . . . 130
3.42 MoDoPtDoLlArS Struct Reference . . . 130
3.42.1 Detailed Description . . . 130
3.42.2 Field Documentation . . . 130
3.42.2.1 number . . . 130
3.42.2.2 type . . . 130
3.43 monomial_larger Class Reference . . . 130
3.43.1 Detailed Description . . . 131
3.43.2 Constructor & Destructor Documentation . . . 131
3.43.2.1 monomial_larger() . . . 131
3.43.3 Member Function Documentation . . . 131
3.43.3.1 operator()() . . . 131
3.44 N_const Struct Reference . . . 131
3.44.1 Detailed Description . . . 134
3.44.2 Field Documentation . . . 134
3.44.2.1 OldPosIn . . . 134
3.44.2.2 OldPosOut . . . 134
3.44.2.3 theposition . . . 134
3.44.2.4 EndNest . . . 134
3.44.2.5 Frozen . . . 134
3.44.2.6 FullProto . . . 135
3.44.2.7 cTerm . . . 135
3.44.2.8 RepPoint . . . 135
3.44.2.9 WildValue . . . 135
3.44.2.10 WildStop . . . 135
3.44.2.11 argaddress . . . 135
3.44.2.12 RepFunList . . . 135
3.44.2.13 patstop . . . 135
3.44.2.14 terstop . . . 136
3.44.2.15 terstart . . . 136
3.44.2.16 terfirstcomm . . . 136
3.44.2.17 DumFound . . . 136
3.44.2.18 DumPlace . . . 136
3.44.2.19 DumFunPlace . . . 136
3.44.2.20 UsedSymbol . . . 136
3.44.2.21 UsedVector . . . 136
3.44.2.22 UsedIndex . . . 137

3.44.2.23 UsedFunction . . . 137
3.44.2.24 MaskPointer . . . 137
3.44.2.25 ForFindOnly . . . 137
3.44.2.26 findTerm . . . 137
3.44.2.27 findPattern . . . 137
3.44.2.28 dummyrenumlist . . . 137
3.44.2.29 funargs . . . 137
3.44.2.30 funlocs . . . 138
3.44.2.31 funinds . . . 138
3.44.2.32 NoScrat2 . . . 138
3.44.2.33 ReplaceScrat . . . 138
3.44.2.34 tracestack . . . 138
3.44.2.35 selecttermundo . . . 138
3.44.2.36 patternbuffer . . . 138
3.44.2.37 termbuffer . . . 138
3.44.2.38 PoinScratch . . . 139
3.44.2.39 FunScratch . . . 139
3.44.2.40 RenumScratch . . . 139
3.44.2.41 FunInfo . . . 139
3.44.2.42 SplitScratch . . . 139
3.44.2.43 SplitScratch1 . . . 139
3.44.2.44 FunSorts . . . 139
3.44.2.45 SoScratC . . . 139
3.44.2.46 listinprint . . . 140
3.44.2.47 currentTerm . . . 140
3.44.2.48 arglist . . . 140
3.44.2.49 tlistbuf . . . 140
3.44.2.50 compressSpace . . . 140
3.44.2.51 SHcombi . . . 140
3.44.2.52 poly_vars . . . 140
3.44.2.53 cmod . . . 140
3.44.2.54 SHvar . . . 141
3.44.2.55 deferskipped . . . 141
3.44.2.56 InScratch . . . 141
3.44.2.57 SplitScratchSize . . . 141
3.44.2.58 InScratch1 . . . 141
3.44.2.59 SplitScratchSize1 . . . 141
3.44.2.60 ninterms . . . 141
3.44.2.61 SHcombisize . . . 141
3.44.2.62 NumTotWildArgs . . . 142
3.44.2.63 UseFindOnly . . . 142
3.44.2.64 UsedOtherFind . . . 142

3.44.2.65 ErrorInDollar . . . 142
3.44.2.66 numfargs . . . 142
3.44.2.67 numflocs . . . 142
3.44.2.68 nargs . . . 142
3.44.2.69 tohunt . . . 142
3.44.2.70 numoffuns . . . 143
3.44.2.71 funisize . . . 143
3.44.2.72 RSsize . . . 143
3.44.2.73 numtracesctack . . . 143
3.44.2.74 intracestack . . . 143
3.44.2.75 numfuninfo . . . 143
3.44.2.76 NumFunSorts . . . 143
3.44.2.77 MaxFunSorts . . . 143
3.44.2.78 arglistsize . . . 144
3.44.2.79 tlistsize . . . 144
3.44.2.80 filenum . . . 144
3.44.2.81 compressSize . . . 144
3.44.2.82 polysortflag . . . 144
3.44.2.83 nogroundlevel . . . 144
3.44.2.84 subsubveto . . . 144
3.44.2.85 MaxRenumScratch . . . 144
3.44.2.86 oldtype . . . 145
3.44.2.87 oldvalue . . . 145
3.44.2.88 NumWild . . . 145
3.44.2.89 RepFunNum . . . 145
3.44.2.90 DisOrderFlag . . . 145
3.44.2.91 WildDirt . . . 145
3.44.2.92 NumFound . . . 145
3.44.2.93 WildReserve . . . 145
3.44.2.94 TeInFun . . . 146
3.44.2.95 TeSuOut . . . 146
3.44.2.96 WildArgs . . . 146
3.44.2.97 WildEat . . . 146
3.44.2.98 PolyNormFlag . . . 146
3.44.2.99 PolyFunTodo . . . 146
3.44.2.100 sizeselecttermundo . . . 146
3.44.2.101 patternbuffersize . . . 146
3.44.2.102 numlistinprint . . . 147
3.44.2.103 ncmod . . . 147
3.44.2.104 ExpectedSign . . . 147
3.44.2.105 SignCheck . . . 147
3.44.2.106 IndDum . . . 147

3.44.2.107 poly_num_vars . . . 147
3.44.2.108 idfunctionflag . . . 147
3.44.2.109 poly_vars_type . . . 148
3.44.2.110 tryterm . . . 148
3.45 nameblock Struct Reference . . . 148
3.45.1 Detailed Description . . . 148
3.45.2 Field Documentation . . . 148
3.45.2.1 previousblock . . . 148
3.45.2.2 position . . . 148
3.45.2.3 names . . . 149
3.46 NaMeNode Struct Reference . . . 149
3.46.1 Detailed Description . . . 149
3.46.2 Field Documentation . . . 149
3.46.2.1 name . . . 149
3.46.2.2 parent . . . 149
3.46.2.3 left . . . 150
3.46.2.4 right . . . 150
3.46.2.5 balance . . . 150
3.46.2.6 type . . . 150
3.46.2.7 number . . . 150
3.47 NaMeTree Struct Reference . . . 151
3.47.1 Detailed Description . . . 151
3.47.2 Field Documentation . . . 151
3.47.2.1 namenode . . . 151
3.47.2.2 namebuffer . . . 152
3.47.2.3 nodesize . . . 152
3.47.2.4 nodefill . . . 152
3.47.2.5 namesize . . . 152
3.47.2.6 namefill . . . 152
3.47.2.7 oldnamefill . . . 152
3.47.2.8 oldnodefill . . . 153
3.47.2.9 globalnamefill . . . 153
3.47.2.10 globalnodefill . . . 153
3.47.2.11 clearnamefill . . . 153
3.47.2.12 clearnodefill . . . 153
3.47.2.13 headnode . . . 153
3.48 NeStInG Struct Reference . . . 154
3.48.1 Detailed Description . . . 154
3.48.2 Field Documentation . . . 154
3.48.2.1 termsize . . . 154
3.48.2.2 funsize . . . 154
3.48.2.3 argsize . . . 154

3.49 NoDe Struct Reference . . . 155
3.49.1 Detailed Description . . . 155
3.49.2 Field Documentation . . . 155
3.49.2.1 left . . . 155
3.49.2.2 rght . . . 155
3.49.2.3 lloser . . . 155
3.49.2.4 rloser . . . 156
3.49.2.5 lsrc . . . 156
3.49.2.6 rsrc . . . 156
3.50 O_const Struct Reference . . . 156
3.50.1 Detailed Description . . . 158
3.50.2 Field Documentation . . . 158
3.50.2.1 SaveData . . . 158
3.50.2.2 SaveHeader . . . 158
3.50.2.3 OptimizeResult . . . 158
3.50.2.4 OutputLine . . . 159
3.50.2.5 OutStop . . . 159
3.50.2.6 OutFill . . . 159
3.50.2.7 bracket . . . 159
3.50.2.8 termbuf . . . 159
3.50.2.9 tabstring . . . 159
3.50.2.10 wpos . . . 159
3.50.2.11 wpoin . . . 159
3.50.2.12 DollarOutBuffer . . . 160
3.50.2.13 CurBufWrt . . . 160
3.50.2.14 FlipWORD . . . 160
3.50.2.15 FlipLONG . . . 160
3.50.2.16 FlipPOS . . . 160
3.50.2.17 FlipPOINTER . . . 160
3.50.2.18 ResizeData . . . 160
3.50.2.19 ResizeWORD . . . 160
3.50.2.20 ResizeNCWORD . . . 161
3.50.2.21 ResizeLONG . . . 161
3.50.2.22 ResizePOS . . . 161
3.50.2.23 ResizePOINTER . . . 161
3.50.2.24 CheckPower . . . 161
3.50.2.25 RenumberVec . . . 161
3.50.2.26 Dictionaries . . . 161
3.50.2.27 tensorList . . . 161
3.50.2.28 inscheme . . . 162
3.50.2.29 NumInBrack . . . 162
3.50.2.30 wlen . . . 162

3.50.2.31 DollarOutSizeBuffer . . . 162
3.50.2.32 DollarInOutBuffer . . . 162
3.50.2.33 Optimize . . . 162
3.50.2.34 OutInBuffer . . . 162
3.50.2.35 NoSpacesInNumbers . . . 162
3.50.2.36 BlockSpaces . . . 163
3.50.2.37 CurrentDictionary . . . 163
3.50.2.38 SizeDictionaries . . . 163
3.50.2.39 NumDictionaries . . . 163
3.50.2.40 CurDictNumbers . . . 163
3.50.2.41 CurDictVariables . . . 163
3.50.2.42 CurDictSpecials . . . 163
3.50.2.43 CurDictFunWithArgs . . . 163
3.50.2.44 CurDictNumberWarning . . . 164
3.50.2.45 CurDictNotInFunctions . . . 164
3.50.2.46 CurDictInDollars . . . 164
3.50.2.47 gNumDictionaries . . . 164
3.50.2.48 schemenum . . . 164
3.50.2.49 transFlag . . . 164
3.50.2.50 powerFlag . . . 164
3.50.2.51 mpower . . . 164
3.50.2.52 resizeFlag . . . 165
3.50.2.53 bufferedInd . . . 165
3.50.2.54 OutSkip . . . 165
3.50.2.55 IsBracket . . . 165
3.50.2.56 InFbrack . . . 165
3.50.2.57 PrintType . . . 165
3.50.2.58 FortFirst . . . 165
3.50.2.59 DoubleFlag . . . 165
3.50.2.60 IndentSpace . . . 166
3.50.2.61 FactorMode . . . 166
3.50.2.62 FactorNum . . . 166
3.50.2.63 ErrorBlock . . . 166
3.50.2.64 OptimizationLevel . . . 166
3.50.2.65 FortDotChar . . . 166
3.51 objects Struct Reference . . . 167
3.51.1 Detailed Description . . . 167
3.51.2 Field Documentation . . . 167
3.51.2.1 position . . . 167
3.51.2.2 size . . . 167
3.51.2.3 date . . . 167
3.51.2.4 tablenumber . . . 167

3.51.2.5 uncompressed . . . 168
3.51.2.6 spare1 . . . 168
3.51.2.7 spare2 . . . 168
3.51.2.8 spare3 . . . 168
3.51.2.9 element . . . 168
3.52 OPTIMIZE Struct Reference . . . 169
3.52.1 Detailed Description . . . 169
3.52.2 Field Documentation . . . 169
3.52.2.1 fval . . . 169
3.52.2.2 ival . . . 169
3.52.2.3 horner . . . 170
3.52.2.4 hornerdirection . . . 170
3.52.2.5 method . . . 170
3.52.2.6 mctstimelimit . . . 170
3.52.2.7 mctsnumexpand . . . 170
3.52.2.8 mctsnumkeep . . . 170
3.52.2.9 mctsnumrepeat . . . 170
3.52.2.10 greedytimelimit . . . 170
3.52.2.11 greedyminnum . . . 171
3.52.2.12 greedymaxperc . . . 171
3.52.2.13 printstats . . . 171
3.52.2.14 debugflags . . . 171
3.52.2.15 schemeflags . . . 171
3.52.2.16 mctsdecaymode . . . 171
3.52.2.17 salter . . . 171
3.53 OPTIMIZERESULT Struct Reference . . . 172
3.53.1 Detailed Description . . . 172
3.53.2 Field Documentation . . . 172
3.53.2.1 code . . . 172
3.53.2.2 nameofexpr . . . 172
3.53.2.3 codesize . . . 172
3.53.2.4 exprnr . . . 172
3.53.2.5 minvar . . . 173
3.53.2.6 maxvar . . . 173
3.54 P_const Struct Reference . . . 173
3.54.1 Detailed Description . . . 174
3.54.2 Field Documentation . . . 174
3.54.2.1 DollarList . . . 174
3.54.2.2 PreVarList . . . 175
3.54.2.3 LoopList . . . 175
3.54.2.4 ProcList . . . 175
3.54.2.5 inside . . . 175

3.54.2.6 PreSwitchStrings . . . 175
3.54.2.7 preStart . . . 175
3.54.2.8 preStop . . . 175
3.54.2.9 preFill . . . 175
3.54.2.10 procedureExtension . . . 176
3.54.2.11 cprocedureExtension . . . 176
3.54.2.12 PreAssignStack . . . 176
3.54.2.13 PreIfStack . . . 176
3.54.2.14 PreSwitchModes . . . 176
3.54.2.15 PreTypes . . . 176
3.54.2.16 StopWatchZero . . . 176
3.54.2.17 InOutBuf . . . 176
3.54.2.18 pSize . . . 177
3.54.2.19 PreAssignFlag . . . 177
3.54.2.20 PreContinuation . . . 177
3.54.2.21 PreproFlag . . . 177
3.54.2.22 iBufError . . . 177
3.54.2.23 PreOut . . . 177
3.54.2.24 PreSwitchLevel . . . 177
3.54.2.25 NumPreSwitchStrings . . . 177
3.54.2.26 MaxPreTypes . . . 178
3.54.2.27 NumPreTypes . . . 178
3.54.2.28 MaxPreIfLevel . . . 178
3.54.2.29 PreIfLevel . . . 178
3.54.2.30 PreInsideLevel . . . 178
3.54.2.31 DelayPrevar . . . 178
3.54.2.32 AllowDelay . . . 178
3.54.2.33 lhdollarerror . . . 178
3.54.2.34 eat . . . 179
3.54.2.35 gNumPre . . . 179
3.54.2.36 PreDebug . . . 179
3.54.2.37 OpenDictionary . . . 179
3.54.2.38 PreAssignLevel . . . 179
3.54.2.39 MaxPreAssignLevel . . . 179
3.54.2.40 DebugFlag . . . 179
3.54.2.41 preError . . . 179
3.54.2.42 ComChar . . . 180
3.54.2.43 cComChar . . . 180
3.55 ParallelVars Struct Reference . . . 180
3.55.1 Detailed Description . . . 181
3.55.2 Field Documentation . . . 181
3.55.2.1 slavebuf . . . 181

3.55.2.2 sbuf . . . 181
3.55.2.3 rbufs . . . 181
3.55.2.4 me . . . 181
3.55.2.5 numtasks . . . 182
3.55.2.6 parallel . . . 182
3.55.2.7 rhsInParallel . . . 182
3.55.2.8 mkSlaveInfile . . . 182
3.55.2.9 exprbufsize . . . 182
3.55.2.10 exprtodo . . . 182
3.55.2.11 log . . . 182
3.55.2.12 numsbufs . . . 182
3.55.2.13 numrbufs . . . 183
3.56 PaRtI Struct Reference . . . 183
3.56.1 Detailed Description . . . 183
3.56.2 Field Documentation . . . 183
3.56.2.1 psize . . . 183
3.56.2.2 args . . . 183
3.56.2.3 nargs . . . 184
3.56.2.4 nfun . . . 184
3.56.2.5 numargs . . . 184
3.56.2.6 numpart . . . 184
3.56.2.7 where . . . 184
3.57 PeRmUtE Struct Reference . . . 184
3.57.1 Detailed Description . . . 185
3.57.2 Field Documentation . . . 185
3.57.2.1 objects . . . 185
3.57.2.2 sign . . . 185
3.57.2.3 n . . . 185
3.57.2.4 cycle . . . 185
3.58 PeRmUtEp Struct Reference . . . 185
3.58.1 Detailed Description . . . 186
3.58.2 Field Documentation . . . 186
3.58.2.1 objects . . . 186
3.58.2.2 sign . . . 186
3.58.2.3 n . . . 186
3.58.2.4 cycle . . . 186
3.59 PF_BUFFER Struct Reference . . . 186
3.59.1 Detailed Description . . . 187
3.59.2 Field Documentation . . . 187
3.59.2.1 buff . . . 187
3.59.2.2 fill . . . 187
3.59.2.3 full . . . 187

3.59.2.4 stop . . . 187
3.59.2.5 status . . . 188
3.59.2.6 retstat . . . 188
3.59.2.7 request . . . 188
3.59.2.8 type . . . 188
3.59.2.9 index . . . 188
3.59.2.10 tag . . . 188
3.59.2.11 from . . . 188
3.59.2.12 numbufs . . . 188
3.59.2.13 active . . . 189
3.60 poly Class Reference . . . 189
3.60.1 Detailed Description . . . 190
3.60.2 Member Function Documentation . . . 191
3.60.2.1 operator[]() [1/2] . . . 191
3.60.2.2 operator[]() [2/2] . . . 191
3.60.2.3 termscopy() . . . 191
3.60.2.4 check_memory() . . . 191
3.60.3 Field Documentation . . . 191
3.60.3.1 terms . . . 191
3.60.3.2 size_of_terms . . . 191
3.60.3.3 modp . . . 192
3.60.3.4 modn . . . 192
3.61 POLYMOD Struct Reference . . . 192
3.61.1 Detailed Description . . . 192
3.61.2 Field Documentation . . . 192
3.61.2.1 coefs . . . 192
3.61.2.2 numsym . . . 193
3.61.2.3 arraysize . . . 193
3.61.2.4 polysize . . . 193
3.61.2.5 modnum . . . 193
3.62 PoSiTiOn Struct Reference . . . 193
3.62.1 Detailed Description . . . 193
3.62.2 Field Documentation . . . 193
3.62.2.1 p1 . . . 193
3.63 PRELOAD Struct Reference . . . 194
3.63.1 Detailed Description . . . 194
3.63.2 Field Documentation . . . 194
3.63.2.1 buffer . . . 194
3.63.2.2 size . . . 194
3.64 pReVaR Struct Reference . . . 194
3.64.1 Detailed Description . . . 195
3.64.2 Field Documentation . . . 195

3.64.2.1 name . . . 195
3.64.2.2 value . . . 195
3.64.2.3 argnames . . . 195
3.64.2.4 nargs . . . 195
3.64.2.5 wildarg . . . 196
3.65 PROCEDURE Struct Reference . . . 196
3.65.1 Detailed Description . . . 196
3.65.2 Field Documentation . . . 197
3.65.2.1 p . . . 197
3.65.2.2 name . . . 197
3.65.2.3 loadmode . . . 197
3.66 R_const Struct Reference . . . 197
3.66.1 Detailed Description . . . 199
3.66.2 Field Documentation . . . 199
3.66.2.1 StoreData . . . 199
3.66.2.2 Fscr . . . 200
3.66.2.3 FoStage4 . . . 200
3.66.2.4 DefPosition . . . 200
3.66.2.5 infile . . . 200
3.66.2.6 outfile . . . 200
3.66.2.7 hidefile . . . 200
3.66.2.8 CompressBuffer . . . 200
3.66.2.9 ComprTop . . . 200
3.66.2.10 CompressPointer . . . 201
3.66.2.11 CompareRoutine . . . 201
3.66.2.12 wranfia . . . 201
3.66.2.13 OldTime . . . 201
3.66.2.14 InInBuf . . . 201
3.66.2.15 InHiBuf . . . 201
3.66.2.16 pWorkSize . . . 201
3.66.2.17 lWorkSize . . . 201
3.66.2.18 posWorkSize . . . 202
3.66.2.19 wranfseed . . . 202
3.66.2.20 NoCompress . . . 202
3.66.2.21 gzipCompress . . . 202
3.66.2.22 Cnumlhs . . . 202
3.66.2.23 outtohide . . . 202
3.66.2.24 wranfcall . . . 202
3.66.2.25 wranfnpair1 . . . 202
3.66.2.26 wranfnpair2 . . . 203
3.66.2.27 GetFile . . . 203
3.66.2.28 KeptInHold . . . 203

3.66.2.29 BracketOn . . . 203
3.66.2.30 MaxBracket . . . 203
3.66.2.31 CurDum . . . 203
3.66.2.32 DeferFlag . . . 203
3.66.2.33 TePos . . . 203
3.66.2.34 sLevel . . . 204
3.66.2.35 Stage4Name . . . 204
3.66.2.36 GetOneFile . . . 204
3.66.2.37 PolyFun . . . 204
3.66.2.38 PolyFunInv . . . 204
3.66.2.39 PolyFunType . . . 204
3.66.2.40 PolyFunExp . . . 204
3.66.2.41 PolyFunVar . . . 204
3.66.2.42 PolyFunPow . . . 205
3.66.2.43 Eside . . . 205
3.66.2.44 MaxDum . . . 205
3.66.2.45 level . . . 205
3.66.2.46 expchanged . . . 205
3.66.2.47 expflags . . . 205
3.66.2.48 CurExpr . . . 205
3.66.2.49 SortType . . . 205
3.66.2.50 ShortSortCount . . . 206
3.67 ReNuMbEr Struct Reference . . . 206
3.67.1 Detailed Description . . . 207
3.67.2 Field Documentation . . . 207
3.67.2.1 startposition . . . 207
3.67.2.2 symb . . . 207
3.67.2.3 indi . . . 207
3.67.2.4 vect . . . 207
3.67.2.5 func . . . 208
3.67.2.6 symnum . . . 208
3.67.2.7 indnum . . . 208
3.67.2.8 vecnum . . . 208
3.67.2.9 funnum . . . 208
3.68 S_const Struct Reference . . . 209
3.68.1 Detailed Description . . . 209
3.68.2 Field Documentation . . . 209
3.68.2.1 MaxExprSize . . . 209
3.68.2.2 OldOnFile . . . 210
3.68.2.3 OldNumFactors . . . 210
3.68.2.4 Oldvflags . . . 210
3.68.2.5 NumOldOnFile . . . 210

3.68.2.6 NumOldNumFactors . . . 210
3.68.2.7 MultiThreaded . . . 210
3.68.2.8 Balancing . . . 210
3.68.2.9 ExecMode . . . 210
3.68.2.10 CollectOverFlag . . . 211
3.69 SeTs Struct Reference . . . 211
3.69.1 Detailed Description . . . 211
3.69.2 Field Documentation . . . 211
3.69.2.1 name . . . 211
3.69.2.2 type . . . 211
3.69.2.3 first . . . 211
3.69.2.4 last . . . 212
3.69.2.5 node . . . 212
3.69.2.6 namesize . . . 212
3.69.2.7 dimension . . . 212
3.70 SETUPPARAMETERS Struct Reference . . . 212
3.70.1 Detailed Description . . . 212
3.70.2 Field Documentation . . . 213
3.70.2.1 parameter . . . 213
3.70.2.2 type . . . 213
3.70.2.3 flags . . . 213
3.70.2.4 value . . . 213
3.71 SHvariables Struct Reference . . . 213
3.71.1 Detailed Description . . . 214
3.71.2 Field Documentation . . . 214
3.71.2.1 outterm . . . 214
3.71.2.2 outfun . . . 214
3.71.2.3 incoef . . . 214
3.71.2.4 stop1 . . . 214
3.71.2.5 stop2 . . . 214
3.71.2.6 ststop1 . . . 214
3.71.2.7 ststop2 . . . 215
3.71.2.8 finishuf . . . 215
3.71.2.9 do_uffle . . . 215
3.71.2.10 combilast . . . 215
3.71.2.11 nincoef . . . 215
3.71.2.12 level . . . 215
3.71.2.13 thefunction . . . 215
3.71.2.14 option . . . 216
3.72 sOrT Struct Reference . . . 216
3.72.1 Detailed Description . . . 218
3.72.2 Field Documentation . . . 218

3.72.2.1 file . . . 218
3.72.2.2 SizeInFile . . . 218
3.72.2.3 lBuffer . . . 218
3.72.2.4 lTop . . . 218
3.72.2.5 lFill . . . 218
3.72.2.6 used . . . 218
3.72.2.7 sBuffer . . . 219
3.72.2.8 sTop . . . 219
3.72.2.9 sTop2 . . . 219
3.72.2.10 sHalf . . . 219
3.72.2.11 sFill . . . 219
3.72.2.12 sPointer . . . 219
3.72.2.13 PoinFill . . . 219
3.72.2.14 SplitScratch . . . 219
3.72.2.15 cBuffer . . . 220
3.72.2.16 Patches . . . 220
3.72.2.17 pStop . . . 220
3.72.2.18 poina . . . 220
3.72.2.19 poin2a . . . 220
3.72.2.20 ktoi . . . 220
3.72.2.21 tree . . . 220
3.72.2.22 fPatches . . . 220
3.72.2.23 inPatches . . . 221
3.72.2.24 fPatchesStop . . . 221
3.72.2.25 iPatches . . . 221
3.72.2.26 f . . . 221
3.72.2.27 ff . . . 221
3.72.2.28 sTerms . . . 221
3.72.2.29 LargeSize . . . 221
3.72.2.30 SmallSize . . . 221
3.72.2.31 SmallEsize . . . 222
3.72.2.32 TermsInSmall . . . 222
3.72.2.33 Terms2InSmall . . . 222
3.72.2.34 GenTerms . . . 222
3.72.2.35 TermsLeft . . . 222
3.72.2.36 GenSpace . . . 222
3.72.2.37 SpaceLeft . . . 222
3.72.2.38 putinsize . . . 222
3.72.2.39 ninterms . . . 223
3.72.2.40 MaxPatches . . . 223
3.72.2.41 MaxFpatches . . . 223
3.72.2.42 type . . . 223

3.72.2.43 lPatch . . . 223
3.72.2.44 fPatchN1 . . . 223
3.72.2.45 PolyWise . . . 223
3.72.2.46 PolyFlag . . . 223
3.72.2.47 cBufferSize . . . 224
3.72.2.48 maxtermsize . . . 224
3.72.2.49 newmaxtermsize . . . 224
3.72.2.50 outputmode . . . 224
3.72.2.51 stagelevel . . . 224
3.72.2.52 fPatchN . . . 224
3.72.2.53 inNum . . . 224
3.72.2.54 stage4 . . . 225
3.73 SpecTatoR Struct Reference . . . 225
3.73.1 Detailed Description . . . 226
3.73.2 Field Documentation . . . 226
3.73.2.1 position . . . 226
3.73.2.2 readpos . . . 226
3.73.2.3 fh . . . 226
3.73.2.4 name . . . 226
3.73.2.5 exprnumber . . . 226
3.73.2.6 flags . . . 226
3.74 StOrEcAcHe Struct Reference . . . 227
3.74.1 Detailed Description . . . 227
3.74.2 Field Documentation . . . 227
3.74.2.1 position . . . 227
3.74.2.2 toppos . . . 228
3.74.2.3 next . . . 228
3.74.2.4 buffer . . . 228
3.75 STOREHEADER Struct Reference . . . 228
3.75.1 Detailed Description . . . 229
3.75.2 Field Documentation . . . 229
3.75.2.1 headermark . . . 229
3.75.2.2 lenWORD . . . 229
3.75.2.3 lenLONG . . . 229
3.75.2.4 lenPOS . . . 229
3.75.2.5 lenPOINTER . . . 230
3.75.2.6 endianness . . . 230
3.75.2.7 sSym . . . 230
3.75.2.8 sInd . . . 230
3.75.2.9 sVec . . . 230
3.75.2.10 sFun . . . 231
3.75.2.11 maxpower . . . 231

3.75.2.12 wildoffset . . . 231
3.75.2.13 revision . . . 231
3.75.2.14 reserved . . . 231
3.76 StreaM Struct Reference . . . 232
3.76.1 Detailed Description . . . 232
3.76.2 Field Documentation . . . 232
3.76.2.1 fileposition . . . 232
3.76.2.2 linenumber . . . 233
3.76.2.3 prevline . . . 233
3.76.2.4 buffer . . . 233
3.76.2.5 pointer . . . 233
3.76.2.6 top . . . 233
3.76.2.7 FoldName . . . 233
3.76.2.8 name . . . 234
3.76.2.9 pname . . . 234
3.76.2.10 buffersize . . . 234
3.76.2.11 bufferposition . . . 234
3.76.2.12 inbuffer . . . 234
3.76.2.13 previous . . . 234
3.76.2.14 handle . . . 234
3.76.2.15 type . . . 235
3.76.2.16 prevars . . . 235
3.76.2.17 previousNoShowInput . . . 235
3.76.2.18 eqnum . . . 235
3.76.2.19 afterwards . . . 235
3.76.2.20 olddelay . . . 235
3.76.2.21 oldnoshowinput . . . 235
3.76.2.22 isnextchar . . . 235
3.76.2.23 nextchar . . . 236
3.76.2.24 reserved . . . 236
3.77 SuBbUf Struct Reference . . . 236
3.77.1 Detailed Description . . . 236
3.77.2 Field Documentation . . . 236
3.77.2.1 subexpnum . . . 236
3.77.2.2 buffernum . . . 236
3.78 SyMbOl Struct Reference . . . 237
3.78.1 Detailed Description . . . 237
3.78.2 Field Documentation . . . 237
3.78.2.1 name . . . 237
3.78.2.2 minpower . . . 237
3.78.2.3 maxpower . . . 237
3.78.2.4 complex . . . 237

3.78.2.5 number . . . 238
3.78.2.6 flags . . . 238
3.78.2.7 node . . . 238
3.78.2.8 namesize . . . 238
3.78.2.9 dimension . . . 238
3.79 T_const Struct Reference . . . 239
3.79.1 Detailed Description . . . 241
3.79.2 Field Documentation . . . 241
3.79.2.1 S0 . . . 241
3.79.2.2 SS . . . 241
3.79.2.3 Nest . . . 241
3.79.2.4 NestStop . . . 241
3.79.2.5 NestPoin . . . 241
3.79.2.6 BrackBuf . . . 242
3.79.2.7 StoreCache . . . 242
3.79.2.8 StoreCacheAlloc . . . 242
3.79.2.9 pWorkSpace . . . 242
3.79.2.10 lWorkSpace . . . 242
3.79.2.11 posWorkSpace . . . 242
3.79.2.12 WorkSpace . . . 242
3.79.2.13 WorkTop . . . 242
3.79.2.14 WorkPointer . . . 243
3.79.2.15 RepCount . . . 243
3.79.2.16 RepTop . . . 243
3.79.2.17 WildArgTaken . . . 243
3.79.2.18 factorials . . . 243
3.79.2.19 small_power_n . . . 243
3.79.2.20 small_power . . . 243
3.79.2.21 bernoullis . . . 243
3.79.2.22 primelist . . . 244
3.79.2.23 pfac . . . 244
3.79.2.24 pBer . . . 244
3.79.2.25 TMaddr . . . 244
3.79.2.26 WildMask . . . 244
3.79.2.27 previousEfactor . . . 244
3.79.2.28 TermMemHeap . . . 244
3.79.2.29 NumberMemHeap . . . 244
3.79.2.30 CacheNumberMemHeap . . . 245
3.79.2.31 bracketinfo . . . 245
3.79.2.32 ListPoly . . . 245
3.79.2.33 ListSymbols . . . 245
3.79.2.34 NumMem . . . 245

3.79.2.35 partitions . . . 245
3.79.2.36 sBer . . . 245
3.79.2.37 pWorkPointer . . . 245
3.79.2.38 lWorkPointer . . . 246
3.79.2.39 posWorkPointer . . . 246
3.79.2.40 InNumMem . . . 246
3.79.2.41 sfact . . . 246
3.79.2.42 mfac . . . 246
3.79.2.43 ebufnum . . . 246
3.79.2.44 fbufnum . . . 246
3.79.2.45 allbufnum . . . 246
3.79.2.46 aebufnum . . . 247
3.79.2.47 idallflag . . . 247
3.79.2.48 idallnum . . . 247
3.79.2.49 idallmaxnum . . . 247
3.79.2.50 WildcardBufferSize . . . 247
3.79.2.51 TermMemMax . . . 247
3.79.2.52 TermMemTop . . . 247
3.79.2.53 NumberMemMax . . . 247
3.79.2.54 NumberMemTop . . . 248
3.79.2.55 CacheNumberMemMax . . . 248
3.79.2.56 CacheNumberMemTop . . . 248
3.79.2.57 bracketindexflag . . . 248
3.79.2.58 optimtimes . . . 248
3.79.2.59 ListSymbolsSize . . . 248
3.79.2.60 NumListSymbols . . . 248
3.79.2.61 numpoly . . . 248
3.79.2.62 LeaveNegative . . . 249
3.79.2.63 TrimPower . . . 249
3.79.2.64 small_power_maxx . . . 249
3.79.2.65 small_power_maxn . . . 249
3.79.2.66 dummysubexp . . . 249
3.79.2.67 comsym . . . 249
3.79.2.68 comnum . . . 249
3.79.2.69 comfun . . . 249
3.79.2.70 comind . . . 250
3.79.2.71 MinVecArg . . . 250
3.79.2.72 FunArg . . . 250
3.79.2.73 locwildvalue . . . 250
3.79.2.74 mulpat . . . 250
3.79.2.75 proexp . . . 250
3.79.2.76 TMout . . . 250

3.79.2.77 TMbuff . . . 250
3.79.2.78 TMdolfac . . . 251
3.79.2.79 nfac . . . 251
3.79.2.80 nBer . . . 251
3.79.2.81 mBer . . . 251
3.79.2.82 PolyAct . . . 251
3.79.2.83 RecFlag . . . 251
3.79.2.84 inprimelist . . . 251
3.79.2.85 sizeprimelist . . . 251
3.79.2.86 fromindex . . . 252
3.80 TaBlEbAsE Struct Reference . . . 252
3.80.1 Detailed Description . . . 252
3.80.2 Field Documentation . . . 253
3.80.2.1 fillpoint . . . 253
3.80.2.2 current . . . 253
3.80.2.3 name . . . 253
3.80.2.4 tablenumbers . . . 253
3.80.2.5 subindex . . . 253
3.80.2.6 numtables . . . 253
3.81 TaBlEbAsEsUbInDeX Struct Reference . . . 254
3.81.1 Detailed Description . . . 254
3.81.2 Field Documentation . . . 254
3.81.2.1 where . . . 254
3.81.2.2 size . . . 254
3.82 TaBlEs Struct Reference . . . 255
3.82.1 Detailed Description . . . 256
3.82.2 Field Documentation . . . 256
3.82.2.1 tablepointers . . . 256
3.82.2.2 prototype . . . 256
3.82.2.3 pattern . . . 256
3.82.2.4 mm . . . 256
3.82.2.5 flags . . . 257
3.82.2.6 boomlijst . . . 257
3.82.2.7 argtail . . . 257
3.82.2.8 spare . . . 257
3.82.2.9 buffers . . . 257
3.82.2.10 totind . . . 258
3.82.2.11 reserved . . . 258
3.82.2.12 defined . . . 258
3.82.2.13 mdefined . . . 258
3.82.2.14 prototypeSize . . . 258
3.82.2.15 numind . . . 259

3.82.2.16 bounds . . . 259
3.82.2.17 strict . . . 259
3.82.2.18 sparse . . . 259
3.82.2.19 numtree . . . 259
3.82.2.20 rootnum . . . 260
3.82.2.21 MaxTreeSize . . . 260
3.82.2.22 bufnum . . . 260
3.82.2.23 bufferssize . . . 260
3.82.2.24 buffersfill . . . 260
3.82.2.25 tablenum . . . 261
3.82.2.26 mode . . . 261
3.82.2.27 numdummies . . . 261
3.83 TrAcEn Struct Reference . . . 261
3.83.1 Detailed Description . . . 261
3.83.2 Field Documentation . . . 262
3.83.2.1 accu . . . 262
3.83.2.2 accup . . . 262
3.83.2.3 termp . . . 262
3.83.2.4 perm . . . 262
3.83.2.5 inlist . . . 262
3.83.2.6 sgn . . . 262
3.83.2.7 num . . . 262
3.83.2.8 level . . . 263
3.83.2.9 factor . . . 263
3.83.2.10 allsign . . . 263
3.84 TrAcEs Struct Reference . . . 263
3.84.1 Detailed Description . . . 264
3.84.2 Field Documentation . . . 264
3.84.2.1 accu . . . 264
3.84.2.2 accup . . . 264
3.84.2.3 termp . . . 264
3.84.2.4 perm . . . 264
3.84.2.5 inlist . . . 265
3.84.2.6 nt3 . . . 265
3.84.2.7 nt4 . . . 265
3.84.2.8 j3 . . . 265
3.84.2.9 j4 . . . 265
3.84.2.10 e3 . . . 265
3.84.2.11 e4 . . . 265
3.84.2.12 eers . . . 265
3.84.2.13 mepf . . . 266
3.84.2.14 mdel . . . 266

3.84.2.15 pepf . . . 266
3.84.2.16 pdel . . . 266
3.84.2.17 sgn . . . 266
3.84.2.18 stap . . . 266
3.84.2.19 step1 . . . 266
3.84.2.20 kstep . . . 266
3.84.2.21 mdum . . . 267
3.84.2.22 gamm . . . 267
3.84.2.23 ad . . . 267
3.84.2.24 a3 . . . 267
3.84.2.25 a4 . . . 267
3.84.2.26 lc3 . . . 267
3.84.2.27 lc4 . . . 267
3.84.2.28 sign1 . . . 267
3.84.2.29 sign2 . . . 268
3.84.2.30 gamma5 . . . 268
3.84.2.31 num . . . 268
3.84.2.32 level . . . 268
3.84.2.33 factor . . . 268
3.84.2.34 allsign . . . 268
3.84.2.35 finalstep . . . 268
3.85 tree Struct Reference . . . 269
3.85.1 Detailed Description . . . 269
3.85.2 Field Documentation . . . 269
3.85.2.1 parent . . . 269
3.85.2.2 left . . . 269
3.85.2.3 right . . . 269
3.85.2.4 value . . . 270
3.85.2.5 blnce . . . 270
3.85.2.6 usage . . . 270
3.86 VaRrEnUm Struct Reference . . . 270
3.86.1 Detailed Description . . . 270
3.86.2 Field Documentation . . . 271
3.86.2.1 start . . . 271
3.86.2.2 lo . . . 271
3.86.2.3 hi . . . 271
3.87 VeCtOr Struct Reference . . . 271
3.87.1 Detailed Description . . . 272
3.87.2 Field Documentation . . . 272
3.87.2.1 name . . . 272
3.87.2.2 complex . . . 272
3.87.2.3 number . . . 272

3.87.2.4 flags . . . 272
3.87.2.5 node . . . 272
3.87.2.6 namesize . . . 272
3.87.2.7 dimension . . . 273
3.88 X_const Struct Reference . . . 273
3.88.1 Detailed Description . . . 273
3.88.2 Field Documentation . . . 273
3.88.2.1 currentPrompt . . . 273
3.88.2.2 shellname . . . 273
3.88.2.3 stderrname . . . 274
3.88.2.4 timeout . . . 274
3.88.2.5 killSignal . . . 274
3.88.2.6 killWholeGroup . . . 274
3.88.2.7 daemonize . . . 274
3.88.2.8 currentExternalChannel . . . 274

4 File Documentation **275**
4.1 argument.c File Reference . . . 275
4.1.1 Detailed Description . . . 276
4.1.2 Macro Definition Documentation . . . 276
4.1.2.1 NEWORDER . . . 276
4.1.3 Function Documentation . . . 276
4.1.3.1 execarg() . . . 276
4.1.3.2 execterm() . . . 276
4.1.3.3 ArgumentImplode() . . . 276
4.1.3.4 ArgumentExplode() . . . 277
4.1.3.5 ArgFactorize() . . . 277
4.1.3.6 FindArg() . . . 277
4.1.3.7 InsertArg() . . . 277
4.1.3.8 CleanupArgCache() . . . 278
4.1.3.9 ArgSymbolMerge() . . . 278
4.1.3.10 ArgDotproductMerge() . . . 278
4.1.3.11 TakeArgContent() . . . 279
4.1.3.12 MakeInteger() . . . 279
4.1.3.13 MakeMod() . . . 280
4.1.3.14 SortWeights() . . . 280
4.2 argument.c . . . 281
4.3 bugtool.c File Reference . . . 321
4.3.1 Detailed Description . . . 321
4.3.2 Function Documentation . . . 321
4.3.2.1 ExprStatus() . . . 321
4.4 bugtool.c . . . 321

4.5 checkpoint.c File Reference . . . 322
4.5.1 Detailed Description . . . 323
4.5.2 Macro Definition Documentation . . . 324
4.5.2.1 CACHED_SNAPSHOT . . . 324
4.5.2.2 CACHE_SIZE . . . 324
4.5.2.3 R_FREE . . . 324
4.5.2.4 R_FREE_NAMETREE . . . 324
4.5.2.5 R_FREE_STREAM . . . 325
4.5.2.6 R_SET . . . 325
4.5.2.7 R_COPY_B . . . 325
4.5.2.8 S_WRITE_B . . . 325
4.5.2.9 S_FLUSH_B . . . 325
4.5.2.10 R_COPY_S . . . 326
4.5.2.11 S_WRITE_S . . . 326
4.5.2.12 R_COPY_LIST . . . 326
4.5.2.13 S_WRITE_LIST . . . 326
4.5.2.14 R_COPY_NAMETREE . . . 327
4.5.2.15 S_WRITE_NAMETREE . . . 327
4.5.2.16 S_WRITE_DOLLAR . . . 327
4.5.2.17 ANNOUNCE . . . 327
4.5.3 Function Documentation . . . 328
4.5.3.1 CheckRecoveryFile() . . . 328
4.5.3.2 DeleteRecoveryFile() . . . 328
4.5.3.3 RecoveryFilename() . . . 328
4.5.3.4 InitRecovery() . . . 328
4.5.3.5 fwrite_cached() . . . 329
4.5.3.6 flush_cache() . . . 329
4.5.3.7 DoRecovery() . . . 329
4.5.3.8 DoCheckpoint() . . . 330
4.5.4 Variable Documentation . . . 330
4.5.4.1 cache_buffer . . . 330
4.5.4.2 cache_fill . . . 331
4.6 checkpoint.c . . . 331
4.7 comexpr.c File Reference . . . 367
4.7.1 Detailed Description . . . 367
4.7.2 Function Documentation . . . 368
4.7.2.1 CoLocal() . . . 368
4.7.2.2 CoGlobal() . . . 368
4.7.2.3 CoLocalFactorized() . . . 368
4.7.2.4 CoGlobalFactorized() . . . 368
4.7.2.5 DoExpr() . . . 368
4.7.2.6 CoIdOld() . . . 368

4.7.2.7 CoId() . . . 369
4.7.2.8 CoIdNew() . . . 369
4.7.2.9 CoDisorder() . . . 369
4.7.2.10 CoMany() . . . 369
4.7.2.11 CoMulti() . . . 369
4.7.2.12 CoIfMatch() . . . 369
4.7.2.13 CoIfNoMatch() . . . 369
4.7.2.14 CoOnce() . . . 370
4.7.2.15 CoOnly() . . . 370
4.7.2.16 CoSelect() . . . 370
4.7.2.17 CoIdExpression() . . . 370
4.7.2.18 CoMultiply() . . . 370
4.7.2.19 CoFill() . . . 370
4.7.2.20 CoFillExpression() . . . 370
4.7.2.21 CoPrintTable() . . . 371
4.7.2.22 CoAssign() . . . 371
4.7.2.23 CoDeallocateTable() . . . 371
4.8 comexpr.c . . . 371
4.9 compcomm.c File Reference . . . 394
4.9.1 Detailed Description . . . 397
4.9.2 Function Documentation . . . 397
4.9.2.1 CoCollect() . . . 397
4.9.2.2 setonoff() . . . 398
4.9.2.3 CoCompress() . . . 398
4.9.2.4 CoFlags() . . . 398
4.9.2.5 CoOff() . . . 398
4.9.2.6 CoOn() . . . 398
4.9.2.7 CoInsideFirst() . . . 398
4.9.2.8 CoProperCount() . . . 399
4.9.2.9 CoDelete() . . . 399
4.9.2.10 CoFormat() . . . 399
4.9.2.11 CoKeep() . . . 399
4.9.2.12 CoFixIndex() . . . 399
4.9.2.13 CoMetric() . . . 399
4.9.2.14 DoPrint() . . . 399
4.9.2.15 CoPrint() . . . 400
4.9.2.16 CoPrintB() . . . 400
4.9.2.17 CoNPrint() . . . 400
4.9.2.18 CoPushHide() . . . 400
4.9.2.19 CoPopHide() . . . 400
4.9.2.20 SetExprCases() . . . 400
4.9.2.21 SetExpr() . . . 401

4.9.2.22 CoDrop() . . . 401
4.9.2.23 CoNoDrop() . . . 401
4.9.2.24 CoSkip() . . . 401
4.9.2.25 CoNoSkip() . . . 401
4.9.2.26 CoHide() . . . 401
4.9.2.27 CoIntoHide() . . . 401
4.9.2.28 CoNoHide() . . . 402
4.9.2.29 CoUnHide() . . . 402
4.9.2.30 CoNoUnHide() . . . 402
4.9.2.31 AddToCom() . . . 402
4.9.2.32 AddComString() . . . 402
4.9.2.33 Add2ComStrings() . . . 402
4.9.2.34 CoDiscard() . . . 403
4.9.2.35 CoContract() . . . 403
4.9.2.36 CoGoTo() . . . 403
4.9.2.37 CoLabel() . . . 403
4.9.2.38 DoArgument() . . . 403
4.9.2.39 CoArgument() . . . 403
4.9.2.40 CoEndArgument() . . . 403
4.9.2.41 CoInside() . . . 404
4.9.2.42 CoEndInside() . . . 404
4.9.2.43 CoNormalize() . . . 404
4.9.2.44 CoMakeInteger() . . . 404
4.9.2.45 CoSplitArg() . . . 404
4.9.2.46 CoSplitFirstArg() . . . 404
4.9.2.47 CoSplitLastArg() . . . 404
4.9.2.48 CoFactArg() . . . 405
4.9.2.49 DoSymmetrize() . . . 405
4.9.2.50 CoSymmetrize() . . . 405
4.9.2.51 CoAntiSymmetrize() . . . 405
4.9.2.52 CoCycleSymmetrize() . . . 405
4.9.2.53 CoRCycleSymmetrize() . . . 405
4.9.2.54 CoWrite() . . . 405
4.9.2.55 CoNWrite() . . . 406
4.9.2.56 CoRatio() . . . 406
4.9.2.57 CoRedefine() . . . 406
4.9.2.58 CoRenumber() . . . 406
4.9.2.59 CoSum() . . . 406
4.9.2.60 CoToTensor() . . . 406
4.9.2.61 CoToVector() . . . 406
4.9.2.62 CoTrace4() . . . 407
4.9.2.63 CoTraceN() . . . 407

4.9.2.64 CoChisholm() . . . 407
4.9.2.65 DoChain() . . . 407
4.9.2.66 CoChainin() . . . 407
4.9.2.67 CoChainout() . . . 407
4.9.2.68 CoExit() . . . 407
4.9.2.69 CoInParallel() . . . 408
4.9.2.70 CoNotInParallel() . . . 408
4.9.2.71 DoInParallel() . . . 408
4.9.2.72 CoInExpression() . . . 408
4.9.2.73 CoEndInExpression() . . . 408
4.9.2.74 CoSetExitFlag() . . . 408
4.9.2.75 CoTryReplace() . . . 408
4.9.2.76 CoModulus() . . . 409
4.9.2.77 CoRepeat() . . . 409
4.9.2.78 CoEndRepeat() . . . 409
4.9.2.79 DoBrackets() . . . 409
4.9.2.80 CoBracket() . . . 409
4.9.2.81 CoAntiBracket() . . . 409
4.9.2.82 CoMultiBracket() . . . 409
4.9.2.83 CountComp() . . . 410
4.9.2.84 CoIf() . . . 410
4.9.2.85 CoElse() . . . 410
4.9.2.86 CoElseIf() . . . 410
4.9.2.87 CoEndIf() . . . 410
4.9.2.88 CoWhile() . . . 410
4.9.2.89 CoEndWhile() . . . 410
4.9.2.90 DoFindLoop() . . . 411
4.9.2.91 CoFindLoop() . . . 411
4.9.2.92 CoReplaceLoop() . . . 411
4.9.2.93 CoFunPowers() . . . 411
4.9.2.94 CoUnitTrace() . . . 411
4.9.2.95 CoTerm() . . . 411
4.9.2.96 CoEndTerm() . . . 411
4.9.2.97 CoSort() . . . 412
4.9.2.98 CoPolyFun() . . . 412
4.9.2.99 CoPolyRatFun() . . . 412
4.9.2.100 CoMerge() . . . 412
4.9.2.101 CoStuffle() . . . 412
4.9.2.102 CoProcessBucket() . . . 412
4.9.2.103 CoThreadBucket() . . . 412
4.9.2.104 DoArgPlode() . . . 413
4.9.2.105 CoArgExplode() . . . 413

4.9.2.106 CoArgImplode() . . . 413
4.9.2.107 CoClearTable() . . . 413
4.9.2.108 CoDenominators() . . . 413
4.9.2.109 CoDropCoefficient() . . . 413
4.9.2.110 CoDropSymbols() . . . 413
4.9.2.111 CoToPolynomial() . . . 414
4.9.2.112 CoFromPolynomial() . . . 414
4.9.2.113 CoArgToExtraSymbol() . . . 414
4.9.2.114 CoExtraSymbols() . . . 414
4.9.2.115 GetIfDollarFactor() . . . 414
4.9.2.116 GetDoParam() . . . 414
4.9.2.117 CoDo() . . . 415
4.9.2.118 CoEndDo() . . . 415
4.9.2.119 CoFactDollar() . . . 415
4.9.2.120 CoFactorize() . . . 415
4.9.2.121 CoNFactorize() . . . 415
4.9.2.122 CoUnFactorize() . . . 415
4.9.2.123 CoNUnFactorize() . . . 415
4.9.2.124 DoFactorize() . . . 416
4.9.2.125 CoOptimizeOption() . . . 416
4.9.2.126 CoPutInside() . . . 416
4.9.2.127 CoAntiPutInside() . . . 416
4.9.2.128 DoPutInside() . . . 416
4.10 compcomm.c . . . 417
4.11 compiler.c File Reference . . . 497
4.11.1 Detailed Description . . . 498
4.11.2 Macro Definition Documentation . . . 499
4.11.2.1 OPTION0 . . . 499
4.11.2.2 OPTION1 . . . 499
4.11.2.3 OPTION2 . . . 499
4.11.2.4 REDUCESUBEXPBUFFERS . . . 499
4.11.3 Function Documentation . . . 499
4.11.3.1 inictable() . . . 499
4.11.3.2 findcommand() . . . 499
4.11.3.3 ParenthesesTest() . . . 500
4.11.3.4 SkipAName() . . . 500
4.11.3.5 IsRHS() . . . 500
4.11.3.6 IsIdStatement() . . . 500
4.11.3.7 CompileAlgebra() . . . 500
4.11.3.8 CompileStatement() . . . 500
4.11.3.9 TestTables() . . . 501
4.11.3.10 CompileSubExpressions() . . . 501

4.11.3.11 CodeGenerator() . . . 501
4.11.3.12 CompleteTerm() . . . 501
4.11.3.13 CodeFactors() . . . 501
4.11.3.14 GenerateFactors() . . . 501
4.11.4 Variable Documentation . . . 502
4.11.4.1 alfatable1 . . . 502
4.11.4.2 subexpbuffers . . . 502
4.11.4.3 topsubexpbuffers . . . 502
4.11.4.4 insubexpbuffers . . . 502
4.12 compiler.c . . . 502
4.13 compress.c File Reference . . . 528
4.13.1 Detailed Description . . . 528
4.14 compress.c . . . 528
4.15 comtool.c File Reference . . . 537
4.15.1 Detailed Description . . . 537
4.15.2 Function Documentation . . . 538
4.15.2.1 inicbufs() . . . 538
4.15.2.2 finishcbuf() . . . 538
4.15.2.3 clearcbuf() . . . 538
4.15.2.4 DoubleCbuffer() . . . 539
4.15.2.5 AddLHS() . . . 539
4.15.2.6 AddRHS() . . . 539
4.15.2.7 AddNtoL() . . . 540
4.15.2.8 AddNtoC() . . . 541
4.15.2.9 InsTree() . . . 542
4.15.2.10 FindTree() . . . 542
4.15.2.11 RedoTree() . . . 542
4.15.2.12 ClearTree() . . . 542
4.15.2.13 IniFbuffer() . . . 542
4.15.2.14 numcommute() . . . 542
4.16 comtool.c . . . 543
4.17 comtool.h File Reference . . . 550
4.17.1 Detailed Description . . . 550
4.18 comtool.h . . . 551
4.19 declare.h File Reference . . . 552
4.19.1 Detailed Description . . . 573
4.19.2 Macro Definition Documentation . . . 573
4.19.2.1 MaX . . . 573
4.19.2.2 MiN . . . 574
4.19.2.3 ABS . . . 574
4.19.2.4 SGN . . . 574
4.19.2.5 REDLENG . . . 574

4.19.2.6 INCLENG . . . 574
4.19.2.7 GETCOEF . . . 574
4.19.2.8 GETSTOP . . . 575
4.19.2.9 StuffAdd . . . 575
4.19.2.10 TOKENTOLINE . . . 575
4.19.2.11 UngetFromStream . . . 575
4.19.2.12 AddLineFeed . . . 575
4.19.2.13 TryRecover . . . 575
4.19.2.14 UngetChar . . . 576
4.19.2.15 ParseNumber . . . 576
4.19.2.16 ParseSign . . . 576
4.19.2.17 ParseSignedNumber . . . 576
4.19.2.18 NCOPY . . . 576
4.19.2.19 NCOPYI . . . 577
4.19.2.20 NCOPYB . . . 577
4.19.2.21 NCOPYI32 . . . 577
4.19.2.22 WCOPY . . . 577
4.19.2.23 NeedNumber . . . 577
4.19.2.24 SKIPBLANKS . . . 578
4.19.2.25 FLUSHCONSOLE . . . 578
4.19.2.26 SKIPBRA1 . . . 578
4.19.2.27 SKIPBRA2 . . . 578
4.19.2.28 SKIPBRA3 . . . 578
4.19.2.29 SKIPBRA4 . . . 579
4.19.2.30 SKIPBRA5 . . . 579
4.19.2.31 CYCLE1 . . . 579
4.19.2.32 AddToCB . . . 579
4.19.2.33 EXCHINOUT . . . 579
4.19.2.34 BACKINOUT . . . 580
4.19.2.35 CopyArg . . . 580
4.19.2.36 FILLARG . . . 580
4.19.2.37 COPYARG . . . 580
4.19.2.38 ZEROARG . . . 580
4.19.2.39 FILLFUN . . . 580
4.19.2.40 COPYFUN . . . 581
4.19.2.41 COPYFUN3 . . . 581
4.19.2.42 FILLFUN3 . . . 581
4.19.2.43 FILLSUB . . . 581
4.19.2.44 COPYSUB . . . 581
4.19.2.45 FILLEXPR . . . 581
4.19.2.46 NEXTARG . . . 582
4.19.2.47 COPY1ARG . . . 582

4.19.2.48 ZeroFillRange . . . 582
4.19.2.49 TABLESIZE . . . 582
4.19.2.50 WORDDIF . . . 583
4.19.2.51 wsizeof . . . 583
4.19.2.52 VARNAME . . . 583
4.19.2.53 DOLLARNAME . . . 583
4.19.2.54 EXPRNAME . . . 583
4.19.2.55 PREV . . . 583
4.19.2.56 SETERROR . . . 584
4.19.2.57 DUMMYUSE . . . 584
4.19.2.58 ADDPOS . . . 584
4.19.2.59 SETBASELENGTH . . . 584
4.19.2.60 SETBASEPOSITION . . . 584
4.19.2.61 ISEQUALPOSINC . . . 584
4.19.2.62 ISGEPOSINC . . . 585
4.19.2.63 DIVPOS . . . 585
4.19.2.64 MULPOS . . . 585
4.19.2.65 DIFPOS . . . 585
4.19.2.66 DIFBASE . . . 585
4.19.2.67 ADD2POS . . . 585
4.19.2.68 PUTZERO . . . 586
4.19.2.69 BASEPOSITION . . . 586
4.19.2.70 SETSTARTPOS . . . 586
4.19.2.71 NOTSTARTPOS . . . 586
4.19.2.72 ISMINPOS . . . 586
4.19.2.73 ISEQUALPOS . . . 586
4.19.2.74 ISNOTEQUALPOS . . . 586
4.19.2.75 ISLESSPOS . . . 587
4.19.2.76 ISGEPOS . . . 587
4.19.2.77 ISNOTZEROPOS . . . 587
4.19.2.78 ISZEROPOS . . . 587
4.19.2.79 ISPOSPOS . . . 587
4.19.2.80 ISNEGPOS . . . 587
4.19.2.81 TOLONG . . . 587
4.19.2.82 Add2Com . . . 588
4.19.2.83 Add3Com . . . 588
4.19.2.84 Add4Com . . . 588
4.19.2.85 Add5Com . . . 588
4.19.2.86 WantAddPointers . . . 588
4.19.2.87 WantAddLongs . . . 589
4.19.2.88 WantAddPositions . . . 589
4.19.2.89 FORM_INLINE . . . 589

4.19.2.90 MEMORYMACROS . . . 589
4.19.2.91 TermMalloc . . . 589
4.19.2.92 NumberMalloc . . . 589
4.19.2.93 CacheNumberMalloc . . . 590
4.19.2.94 TermFree . . . 590
4.19.2.95 NumberFree . . . 590
4.19.2.96 CacheNumberFree . . . 590
4.19.2.97 NestingChecksum . . . 590
4.19.2.98 MesNesting . . . 590
4.19.2.99 MarkPolyRatFunDirty . . . 591
4.19.2.100 PUSHPREASSIGNLEVEL . . . 591
4.19.2.101 POPPREASSIGNLEVEL . . . 591
4.19.2.102 EXTERNLOCK . . . 591
4.19.2.103 INILOCK . . . 591
4.19.2.104 LOCK . . . 592
4.19.2.105 UNLOCK . . . 592
4.19.2.106 EXTERNRWLOCK . . . 592
4.19.2.107 INIRWLOCK . . . 592
4.19.2.108 RWLOCKR . . . 592
4.19.2.109 RWLOCKW . . . 592
4.19.2.110 UNRWLOCK . . . 592
4.19.2.111 MLOCK . . . 593
4.19.2.112 MUNLOCK . . . 593
4.19.2.113 GETIDENTITY . . . 593
4.19.2.114 GETBIDENTITY . . . 593
4.19.2.115 M_alloc . . . 593
4.19.2.116 CompareTerms . . . 593
4.19.2.117 FiniShuffle . . . 593
4.19.2.118 DoShtuffle . . . 594
4.19.3 Typedef Documentation . . . 594
4.19.3.1 WRITEBUFTOEXTCHANNEL . . . 594
4.19.3.2 GETCFROMEXTCHANNEL . . . 594
4.19.3.3 SETTERMINATORFOREXTERNALCHANNEL . . . 594
4.19.3.4 SETKILLMODEFOREXTERNALCHANNEL . . . 594
4.19.3.5 WRITEFILE . . . 594
4.19.3.6 COMPARE . . . 594
4.19.3.7 GETTERM . . . 595
4.19.3.8 FINISHUFFLE . . . 595
4.19.3.9 DO_UFFLE . . . 595
4.19.4 Function Documentation . . . 595
4.19.4.1 TELLFILE() . . . 595
4.19.4.2 StartVariables() . . . 595

4.19.4.3 CodeToLine() . . . 596
4.19.4.4 AddArrayIndex() . . . 596
4.19.4.5 FindInIndex() . . . 596
4.19.4.6 NextFileIndex() . . . 596
4.19.4.7 PasteTerm() . . . 596
4.19.4.8 StrCopy() . . . 597
4.19.4.9 WrtPower() . . . 597
4.19.4.10 AccumGCD() . . . 597
4.19.4.11 AddArgs() . . . 597
4.19.4.12 AddCoef() . . . 598
4.19.4.13 AddLong() . . . 598
4.19.4.14 AddPLon() . . . 599
4.19.4.15 AddPoly() . . . 599
4.19.4.16 AddRat() . . . 600
4.19.4.17 AddToLine() . . . 600
4.19.4.18 AddWild() . . . 600
4.19.4.19 BigLong() . . . 600
4.19.4.20 BinomGen() . . . 601
4.19.4.21 CheckWild() . . . 601
4.19.4.22 Chisholm() . . . 601
4.19.4.23 CleanExpr() . . . 601
4.19.4.24 CleanUp() . . . 601
4.19.4.25 ClearWild() . . . 602
4.19.4.26 CompareFunctions() . . . 602
4.19.4.27 Commute() . . . 602
4.19.4.28 DetCommu() . . . 602
4.19.4.29 DoesCommu() . . . 602
4.19.4.30 CompArg() . . . 602
4.19.4.31 CompCoef() . . . 603
4.19.4.32 CompGroup() . . . 603
4.19.4.33 Compare1() . . . 603
4.19.4.34 CountDo() . . . 604
4.19.4.35 CountFun() . . . 604
4.19.4.36 DimensionSubterm() . . . 604
4.19.4.37 DimensionTerm() . . . 604
4.19.4.38 DimensionExpression() . . . 605
4.19.4.39 Deferred() . . . 605
4.19.4.40 DeleteStore() . . . 606
4.19.4.41 DetCurDum() . . . 606
4.19.4.42 DetVars() . . . 606
4.19.4.43 Distribute() . . . 606
4.19.4.44 DivLong() . . . 606

4.19.4.45 DivRat() . . . 607
4.19.4.46 Divvy() . . . 607
4.19.4.47 DoDelta() . . . 607
4.19.4.48 DoDelta3() . . . 607
4.19.4.49 TestPartitions() . . . 607
4.19.4.50 DoPartitions() . . . 607
4.19.4.51 DoTableExpansion() . . . 608
4.19.4.52 DoDistrib() . . . 608
4.19.4.53 DoShuffle() . . . 608
4.19.4.54 DoPermutations() . . . 608
4.19.4.55 Shuffle() . . . 608
4.19.4.56 FinishShuffle() . . . 608
4.19.4.57 DoStuffle() . . . 609
4.19.4.58 Stuffle() . . . 609
4.19.4.59 FinishStuffle() . . . 609
4.19.4.60 StuffRootAdd() . . . 609
4.19.4.61 TestUse() . . . 609
4.19.4.62 FindTB() . . . 609
4.19.4.63 CheckTableDeclarations() . . . 610
4.19.4.64 Apply() . . . 610
4.19.4.65 ApplyExec() . . . 610
4.19.4.66 ApplyReset() . . . 610
4.19.4.67 ReWorkT() . . . 610
4.19.4.68 GetIfDollarNum() . . . 610
4.19.4.69 FindVar() . . . 611
4.19.4.70 DoIfStatement() . . . 611
4.19.4.71 DoOnePow() . . . 611
4.19.4.72 DoRevert() . . . 612
4.19.4.73 DoSumF1() . . . 612
4.19.4.74 DoSumF2() . . . 612
4.19.4.75 DoTheta() . . . 613
4.19.4.76 EndSort() . . . 613
4.19.4.77 EntVar() . . . 614
4.19.4.78 EpfCon() . . . 614
4.19.4.79 EpfFind() . . . 615
4.19.4.80 EpfGen() . . . 615
4.19.4.81 EqualArg() . . . 615
4.19.4.82 Factorial() . . . 615
4.19.4.83 Bernoulli() . . . 615
4.19.4.84 FactorIn() . . . 616
4.19.4.85 FactorInExpr() . . . 616
4.19.4.86 FindAll() . . . 616

4.19.4.87 FindMulti() . . . 616
4.19.4.88 FindOnce() . . . 616
4.19.4.89 FindOnly() . . . 616
4.19.4.90 FindRest() . . . 617
4.19.4.91 FindrNumber() . . . 617
4.19.4.92 FiniTerm() . . . 617
4.19.4.93 FlushOut() . . . 617
4.19.4.94 FunLevel() . . . 618
4.19.4.95 AdjustRenumScratch() . . . 618
4.19.4.96 GcdLong() . . . 618
4.19.4.97 LcmLong() . . . 619
4.19.4.98 Generator() . . . 619
4.19.4.99 GetBinom() . . . 620
4.19.4.100 GetFromStore() . . . 620
4.19.4.101 GetLong() . . . 620
4.19.4.102 GetMoreTerms() . . . 621
4.19.4.103 GetMoreFromMem() . . . 621
4.19.4.104 GetOneTerm() . . . 621
4.19.4.105 GetTable() . . . 621
4.19.4.106 GetTerm() . . . 621
4.19.4.107 Glue() . . . 621
4.19.4.108 InFunction() . . . 621
4.19.4.109 IniLine() . . . 622
4.19.4.110 InsertTerm() . . . 623
4.19.4.111 LongToLine() . . . 623
4.19.4.112 MakeDirty() . . . 623
4.19.4.113 MarkDirty() . . . 624
4.19.4.114 PolyFunDirty() . . . 624
4.19.4.115 PolyFunClean() . . . 624
4.19.4.116 MatchE() . . . 624
4.19.4.117 MatchCy() . . . 624
4.19.4.118 FunMatchCy() . . . 624
4.19.4.119 FunMatchSy() . . . 625
4.19.4.120 MatchArgument() . . . 625
4.19.4.121 MatchFunction() . . . 625
4.19.4.122 MergePatches() . . . 625
4.19.4.123 MesCerr() . . . 626
4.19.4.124 MesComp() . . . 626
4.19.4.125 Modulus() . . . 626
4.19.4.126 MoveDummies() . . . 627
4.19.4.127 MulLong() . . . 627
4.19.4.128 MulRat() . . . 627

4.19.4.129 Mully() . . . 627
4.19.4.130 MultDo() . . . 627
4.19.4.131 NewSort() . . . 628
4.19.4.132 ExtraSymbol() . . . 628
4.19.4.133 Normalize() . . . 628
4.19.4.134 BracketNormalize() . . . 628
4.19.4.135 DropCoefficient() . . . 628
4.19.4.136 DropSymbols() . . . 629
4.19.4.137 PutInside() . . . 629
4.19.4.138 Pack() . . . 629
4.19.4.139 PasteFile() . . . 629
4.19.4.140 Permute() . . . 630
4.19.4.141 PermuteP() . . . 630
4.19.4.142 PolyFunMul() . . . 630
4.19.4.143 PrepPoly() . . . 631
4.19.4.144 Product() . . . 631
4.19.4.145 PrtLong() . . . 632
4.19.4.146 PutBracket() . . . 632
4.19.4.147 PutIn() . . . 632
4.19.4.148 PutInStore() . . . 632
4.19.4.149 PutOut() . . . 633
4.19.4.150 Quotient() . . . 633
4.19.4.151 RaisPow() . . . 634
4.19.4.152 RaisPowCached() . . . 634
4.19.5 Description . . . 634
4.19.6 Notes . . . 634
4.19.6.1 RaisPowMod() . . . 634
4.19.6.2 NormalModulus() . . . 634
4.19.6.3 GetModInverses() . . . 635
4.19.6.4 GetLongModInverses() . . . 635
4.19.6.5 RatToLine() . . . 635
4.19.6.6 RatioFind() . . . 635
4.19.6.7 RatioGen() . . . 635
4.19.6.8 ReNumber() . . . 636
4.19.6.9 ReadSnum() . . . 636
4.19.6.10 Remain10() . . . 636
4.19.6.11 Remain4() . . . 636
4.19.6.12 ResolveSet() . . . 636
4.19.6.13 ScanFunctions() . . . 636
4.19.6.14 SeekScratch() . . . 637
4.19.6.15 SetEndScratch() . . . 637
4.19.6.16 SetEndHScratch() . . . 637

4.19.6.17 Sflush() . . . 637
4.19.6.18 Simplify() . . . 637
4.19.6.19 SortWild() . . . 638
4.19.6.20 LocateBase() . . . 638
4.19.6.21 SplitMerge() . . . 638
4.19.6.22 StoreTerm() . . . 639
4.19.6.23 SubPLon() . . . 640
4.19.6.24 Substitute() . . . 640
4.19.6.25 SymFind() . . . 640
4.19.6.26 SymGen() . . . 641
4.19.6.27 Symmetrize() . . . 641
4.19.6.28 FullSymmetrize() . . . 641
4.19.6.29 TakeModulus() . . . 641
4.19.6.30 TakeNormalModulus() . . . 641
4.19.6.31 TalToLine() . . . 642
4.19.6.32 TenVec() . . . 642
4.19.6.33 TenVecFind() . . . 642
4.19.6.34 TermRenumber() . . . 642
4.19.6.35 PutInVflags() . . . 643
4.19.6.36 TestMatch() . . . 643
4.19.6.37 TestSub() . . . 644
4.19.6.38 TimeCPU() . . . 645
4.19.6.39 TimeChildren() . . . 645
4.19.6.40 TimeWallClock() . . . 646
4.19.6.41 ToStorage() . . . 646
4.19.6.42 TokenToLine() . . . 646
4.19.6.43 Trace4() . . . 646
4.19.6.44 Trace4Gen() . . . 646
4.19.6.45 Trace4no() . . . 647
4.19.6.46 TraceFind() . . . 647
4.19.6.47 TraceN() . . . 647
4.19.6.48 TraceNgen() . . . 647
4.19.6.49 TraceNno() . . . 647
4.19.6.50 Traces() . . . 647
4.19.6.51 Trick() . . . 648
4.19.6.52 TryDo() . . . 648
4.19.6.53 UnPack() . . . 648
4.19.6.54 VarStore() . . . 648
4.19.6.55 WildFill() . . . 648
4.19.6.56 WriteOne() . . . 649
4.19.6.57 WriteArgument() . . . 649
4.19.6.58 WriteExpression() . . . 649

4.19.6.59 WriteInnerTerm() . . . 649
4.19.6.60 WriteStats() . . . 649
4.19.6.61 WriteSubTerm() . . . 650
4.19.6.62 WriteTerm() . . . 650
4.19.6.63 execarg() . . . 650
4.19.6.64 execterm() . . . 650
4.19.6.65 SpecialCleanup() . . . 651
4.19.6.66 SetMods() . . . 651
4.19.6.67 UnSetMods() . . . 651
4.19.6.68 DoExecute() . . . 651
4.19.6.69 SetScratch() . . . 651
4.19.6.70 Warning() . . . 651
4.19.6.71 HighWarning() . . . 651
4.19.6.72 SpareTable() . . . 652
4.19.6.73 strDup1() . . . 652
4.19.6.74 Malloc() . . . 652
4.19.6.75 Malloc1() . . . 652
4.19.6.76 DoTail() . . . 652
4.19.6.77 PutPreVar() . . . 652
4.19.6.78 Error0() . . . 653
4.19.6.79 Error1() . . . 653
4.19.6.80 Error2() . . . 653
4.19.6.81 ReadFromStream() . . . 653
4.19.6.82 GetFromStream() . . . 654
4.19.6.83 LookInStream() . . . 654
4.19.6.84 OpenStream() . . . 654
4.19.6.85 LocateFile() . . . 654
4.19.6.86 CloseStream() . . . 654
4.19.6.87 PositionStream() . . . 654
4.19.6.88 ReverseStatements() . . . 655
4.19.6.89 ProcessOption() . . . 655
4.19.6.90 Terminate() . . . 655
4.19.6.91 GetNode() . . . 655
4.19.6.92 AddName() . . . 655
4.19.6.93 GetName() . . . 655
4.19.6.94 GetLastExprName() . . . 656
4.19.6.95 GetAutoName() . . . 656
4.19.6.96 GetVar() . . . 656
4.19.6.97 MakeDubious() . . . 656
4.19.6.98 GetOName() . . . 656
4.19.6.99 DumpTree() . . . 656
4.19.6.100 DumpNode() . . . 657

4.19.6.101 LinkTree() 657
4.19.6.102 CopyTree() 657
4.19.6.103 CompactifyTree() 657
4.19.6.104 FreeNameTree() 657
4.19.6.105 AddExpression() 657
4.19.6.106 AddSymbol() 658
4.19.6.107 AddDollar() 658
4.19.6.108 ReplaceDollar() 658
4.19.6.109 DollarRaiseLow() 658
4.19.6.110 AddVector() 658
4.19.6.111 AddDubious() 659
4.19.6.112 AddIndex() 659
4.19.6.113 DoDimension() 659
4.19.6.114 AddFunction() 659
4.19.6.115 CoCommuteInSet() 659
4.19.6.116 CoFunction() 660
4.19.6.117 TestName() 660
4.19.6.118 AddSet() 660
4.19.6.119 DoElements() 660
4.19.6.120 DoTempSet() 660
4.19.6.121 NameConflict() 660
4.19.6.122 OpenFile() 661
4.19.6.123 OpenAddFile() 661
4.19.6.124 ReOpenFile() 661
4.19.6.125 CreateFile() 661
4.19.6.126 CreateLogFile() 661
4.19.6.127 CloseFile() 661
4.19.6.128 CopyFile() 662
4.19.6.129 ReadFile() 662
4.19.6.130 ReadPosFile() 662
4.19.6.131 WriteFileToFile() 662
4.19.6.132 SeekFile() 662
4.19.6.133 TellFile() 663
4.19.6.134 FlushFile() 663
4.19.6.135 GetPosFile() 663
4.19.6.136 SetPosFile() 663
4.19.6.137 SynchFile() 663
4.19.6.138 TruncateFile() 663
4.19.6.139 GetChannel() 663
4.19.6.140 GetAppendChannel() 664
4.19.6.141 CloseChannel() 664
4.19.6.142 findcommand() 664

4.19.6.143 inicbufs() . . . 664
4.19.6.144 FromList() . . . 664
4.19.6.145 From0List() . . . 665
4.19.6.146 FromVarList() . . . 665
4.19.6.147 DoubleList() . . . 665
4.19.6.148 DoubleLList() . . . 665
4.19.6.149 DoubleBuffer() . . . 665
4.19.6.150 ExpandBuffer() . . . 665
4.19.6.151 iexp() . . . 666
4.19.6.152 IsLikeVector() . . . 666
4.19.6.153 AreArgsEqual() . . . 666
4.19.6.154 CompareArgs() . . . 666
4.19.6.155 SkipField() . . . 666
4.19.6.156 StrCmp() . . . 666
4.19.6.157 StrICmp() . . . 667
4.19.6.158 StrHICmp() . . . 667
4.19.6.159 StrICont() . . . 667
4.19.6.160 ConWord() . . . 667
4.19.6.161 StrLen() . . . 667
4.19.6.162 GetPreVar() . . . 667
4.19.6.163 ToGeneral() . . . 668
4.19.6.164 ToPolyFunGeneral() . . . 668
4.19.6.165 ToFast() . . . 668
4.19.6.166 GetSetupPar() . . . 668
4.19.6.167 AllocSort() . . . 668
4.19.6.168 AllocSortFileName() . . . 668
4.19.6.169 LoadInputFile() . . . 669
4.19.6.170 GetChar() . . . 669
4.19.6.171 CharOut() . . . 669
4.19.6.172 PopPreVars() . . . 669
4.19.6.173 IniModule() . . . 669
4.19.6.174 IniSpecialModule() . . . 669
4.19.6.175 ModuleInstruction() . . . 669
4.19.6.176 LoadInstruction() . . . 670
4.19.6.177 LoadStatement() . . . 670
4.19.6.178 FindKeyWord() . . . 670
4.19.6.179 FindInKeyWord() . . . 670
4.19.6.180 DoDefine() . . . 670
4.19.6.181 DoRedefine() . . . 670
4.19.6.182 TheDefine() . . . 670
4.19.6.183 TheUndefine() . . . 671
4.19.6.184 ClearMacro() . . . 671

4.19.6.185 DoUndefine() . . . 671
4.19.6.186 DoInclude() . . . 672
4.19.6.187 DoReverseInclude() . . . 672
4.19.6.188 Include() . . . 672
4.19.6.189 DoExternal() . . . 672
4.19.6.190 DoToExternal() . . . 672
4.19.6.191 DoFromExternal() . . . 672
4.19.6.192 DoPrompt() . . . 672
4.19.6.193 DoSetExternal() . . . 673
4.19.6.194 DoSetExternalAttr() . . . 673
4.19.6.195 DoRmExternal() . . . 673
4.19.6.196 DoFactDollar() . . . 673
4.19.6.197 GetDollarNumber() . . . 673
4.19.6.198 DoSetRandom() . . . 673
4.19.6.199 DoOptimize() . . . 673
4.19.6.200 DoClearOptimize() . . . 674
4.19.6.201 DoSkipExtraSymbols() . . . 674
4.19.6.202 DoMessage() . . . 674
4.19.6.203 DoPreOut() . . . 674
4.19.6.204 DoPreAppend() . . . 674
4.19.6.205 DoPreCreate() . . . 674
4.19.6.206 DoPreAssign() . . . 674
4.19.6.207 DoPreBreak() . . . 675
4.19.6.208 DoPreDefault() . . . 675
4.19.6.209 DoPreSwitch() . . . 675
4.19.6.210 DoPreEndSwitch() . . . 675
4.19.6.211 DoPreCase() . . . 675
4.19.6.212 DoPreShow() . . . 675
4.19.6.213 DoPreExchange() . . . 675
4.19.6.214 DoSystem() . . . 676
4.19.6.215 DoPipe() . . . 676
4.19.6.216 DoIfdef() . . . 676
4.19.6.217 DoElse() . . . 676
4.19.6.218 DoElseif() . . . 676
4.19.6.219 DoEndif() . . . 676
4.19.6.220 DoTerminate() . . . 676
4.19.6.221 DoIf() . . . 677
4.19.6.222 DoCall() . . . 677
4.19.6.223 DoDebug() . . . 677
4.19.6.224 DoDo() . . . 677
4.19.6.225 DoBreakDo() . . . 677
4.19.6.226 DoEnddo() . . . 677

4.19.6.227 DoEndprocedure() . . . 677
4.19.6.228 DoInside() . . . 678
4.19.6.229 DoEndInside() . . . 678
4.19.6.230 DoProcedure() . . . 678
4.19.6.231 DoPrePrintTimes() . . . 678
4.19.6.232 DoPreWrite() . . . 678
4.19.6.233 DoPreClose() . . . 678
4.19.6.234 DoPreRemove() . . . 678
4.19.6.235 DoCommentChar() . . . 679
4.19.6.236 DoPrcExtension() . . . 679
4.19.6.237 DoPreReset() . . . 679
4.19.6.238 WriteString() . . . 679
4.19.6.239 WriteUnfinString() . . . 679
4.19.6.240 AddToString() . . . 679
4.19.6.241 PreEval() . . . 680
4.19.6.242 NumToStr() . . . 680
4.19.6.243 PreCmp() . . . 680
4.19.6.244 PreEq() . . . 680
4.19.6.245 pParseObject() . . . 680
4.19.6.246 PreIfEval() . . . 681
4.19.6.247 EvalPreIf() . . . 681
4.19.6.248 PreLoad() . . . 681
4.19.6.249 PreSkip() . . . 681
4.19.6.250 EndOfToken() . . . 681
4.19.6.251 SetSpecialMode() . . . 681
4.19.6.252 ExecModule() . . . 682
4.19.6.253 DoPolyfun() . . . 682
4.19.6.254 DoPolyratfun() . . . 682
4.19.6.255 CompileStatement() . . . 682
4.19.6.256 ToToken() . . . 682
4.19.6.257 GetDollar() . . . 682
4.19.6.258 MesCall() . . . 682
4.19.6.259 NumCopy() . . . 683
4.19.6.260 LongCopy() . . . 683
4.19.6.261 LongLongCopy() . . . 683
4.19.6.262 ReserveTempFiles() . . . 683
4.19.6.263 PrintTerm() . . . 683
4.19.6.264 PrintTermC() . . . 683
4.19.6.265 PrintSubTerm() . . . 684
4.19.6.266 PrintWords() . . . 684
4.19.6.267 PrintSeq() . . . 684
4.19.6.268 ExpandTripleDots() . . . 684

4.19.6.269 ComPress() . . . 684
4.19.6.270 StageSort() . . . 685
4.19.6.271 M_free() . . . 685
4.19.6.272 AddWildcardName() . . . 685
4.19.6.273 GetWildcardName() . . . 685
4.19.6.274 Globalize() . . . 685
4.19.6.275 ResetVariables() . . . 686
4.19.6.276 AddToPreTypes() . . . 686
4.19.6.277 MessPreNesting() . . . 686
4.19.6.278 GetStreamPosition() . . . 686
4.19.6.279 DoubleCbuffer() . . . 686
4.19.6.280 AddLHS() . . . 687
4.19.6.281 AddRHS() . . . 687
4.19.6.282 AddNtoL() . . . 688
4.19.6.283 AddNtoC() . . . 688
4.19.6.284 CreateStream() . . . 689
4.19.6.285 setonoff() . . . 689
4.19.6.286 DoPrint() . . . 689
4.19.6.287 SetExpr() . . . 690
4.19.6.288 AddToCom() . . . 690
4.19.6.289 Add2ComStrings() . . . 690
4.19.6.290 DoSymmetrize() . . . 690
4.19.6.291 DoArgument() . . . 690
4.19.6.292 ArgFactorize() . . . 690
4.19.6.293 TakeArgContent() . . . 691
4.19.6.294 MakeInteger() . . . 691
4.19.6.295 MakeMod() . . . 692
4.19.6.296 FindArg() . . . 692
4.19.6.297 InsertArg() . . . 692
4.19.6.298 CleanupArgCache() . . . 693
4.19.6.299 ArgSymbolMerge() . . . 693
4.19.6.300 ArgDotproductMerge() . . . 694
4.19.6.301 SortWeights() . . . 694
4.19.6.302 DoBrackets() . . . 694
4.19.6.303 DoPutInside() . . . 694
4.19.6.304 CountComp() . . . 695
4.19.6.305 CoAntiBracket() . . . 695
4.19.6.306 CoAntiSymmetrize() . . . 695
4.19.6.307 DoArgPlode() . . . 695
4.19.6.308 CoArgExplode() . . . 695
4.19.6.309 CoArgImplode() . . . 695
4.19.6.310 CoArgument() . . . 695

4.19.6.311 CoInside() . . . 696
4.19.6.312 ExecInside() . . . 696
4.19.6.313 CoInExpression() . . . 696
4.19.6.314 CoInParallel() . . . 696
4.19.6.315 CoNotInParallel() . . . 696
4.19.6.316 DoInParallel() . . . 696
4.19.6.317 CoEndInExpression() . . . 696
4.19.6.318 CoBracket() . . . 697
4.19.6.319 CoPutInside() . . . 697
4.19.6.320 CoAntiPutInside() . . . 697
4.19.6.321 CoMultiBracket() . . . 697
4.19.6.322 CoCFunction() . . . 697
4.19.6.323 CoCTensor() . . . 697
4.19.6.324 CoCollect() . . . 697
4.19.6.325 CoCompress() . . . 698
4.19.6.326 CoContract() . . . 698
4.19.6.327 CoCycleSymmetrize() . . . 698
4.19.6.328 CoDelete() . . . 698
4.19.6.329 CoTableBase() . . . 698
4.19.6.330 CoApply() . . . 698
4.19.6.331 CoDenominators() . . . 698
4.19.6.332 CoDimension() . . . 699
4.19.6.333 CoDiscard() . . . 699
4.19.6.334 CoDisorder() . . . 699
4.19.6.335 CoDrop() . . . 699
4.19.6.336 CoDropCoefficient() . . . 699
4.19.6.337 CoDropSymbols() . . . 699
4.19.6.338 CoElse() . . . 699
4.19.6.339 CoElseIf() . . . 700
4.19.6.340 CoEndArgument() . . . 700
4.19.6.341 CoEndInside() . . . 700
4.19.6.342 CoEndIf() . . . 700
4.19.6.343 CoEndRepeat() . . . 700
4.19.6.344 CoEndTerm() . . . 700
4.19.6.345 CoEndWhile() . . . 700
4.19.6.346 CoExit() . . . 701
4.19.6.347 CoFactArg() . . . 701
4.19.6.348 CoFactDollar() . . . 701
4.19.6.349 CoFactorize() . . . 701
4.19.6.350 CoNFactorize() . . . 701
4.19.6.351 CoUnFactorize() . . . 701
4.19.6.352 CoNUnFactorize() . . . 701

4.19.6.353 DoFactorize() . . . 702
4.19.6.354 CoFill() . . . 702
4.19.6.355 CoFillExpression() . . . 702
4.19.6.356 CoFixIndex() . . . 702
4.19.6.357 CoFormat() . . . 702
4.19.6.358 CoGlobal() . . . 702
4.19.6.359 CoGlobalFactorized() . . . 702
4.19.6.360 CoGoTo() . . . 703
4.19.6.361 CoId() . . . 703
4.19.6.362 CoIdNew() . . . 703
4.19.6.363 CoIdOld() . . . 703
4.19.6.364 CoIf() . . . 703
4.19.6.365 CoIfMatch() . . . 703
4.19.6.366 CoIfNoMatch() . . . 703
4.19.6.367 CoIndex() . . . 704
4.19.6.368 CoInsideFirst() . . . 704
4.19.6.369 CoKeep() . . . 704
4.19.6.370 CoLabel() . . . 704
4.19.6.371 CoLoad() . . . 704
4.19.6.372 CoLocal() . . . 704
4.19.6.373 CoLocalFactorized() . . . 704
4.19.6.374 CoMany() . . . 705
4.19.6.375 CoMerge() . . . 705
4.19.6.376 CoStuffle() . . . 705
4.19.6.377 CoMetric() . . . 705
4.19.6.378 CoModOption() . . . 705
4.19.6.379 CoModuleOption() . . . 705
4.19.6.380 CoModulus() . . . 705
4.19.6.381 CoMulti() . . . 706
4.19.6.382 CoMultiply() . . . 706
4.19.6.383 CoNFunction() . . . 706
4.19.6.384 CoNPrint() . . . 706
4.19.6.385 CoNTensor() . . . 706
4.19.6.386 CoNWrite() . . . 706
4.19.6.387 CoNoDrop() . . . 706
4.19.6.388 CoNoSkip() . . . 707
4.19.6.389 CoNormalize() . . . 707
4.19.6.390 CoMakeInteger() . . . 707
4.19.6.391 CoFlags() . . . 707
4.19.6.392 CoOff() . . . 707
4.19.6.393 CoOn() . . . 707
4.19.6.394 CoOnce() . . . 707

4.19.6.395 CoOnly() . . . 708
4.19.6.396 CoOptimizeOption() . . . 708
4.19.6.397 CoPolyFun() . . . 708
4.19.6.398 CoPolyRatFun() . . . 708
4.19.6.399 CoPrint() . . . 708
4.19.6.400 CoPrintB() . . . 708
4.19.6.401 CoProperCount() . . . 708
4.19.6.402 CoUnitTrace() . . . 709
4.19.6.403 CoRCycleSymmetrize() . . . 709
4.19.6.404 CoRatio() . . . 709
4.19.6.405 CoRedefine() . . . 709
4.19.6.406 CoRenumber() . . . 709
4.19.6.407 CoRepeat() . . . 709
4.19.6.408 CoSave() . . . 709
4.19.6.409 CoSelect() . . . 710
4.19.6.410 CoSet() . . . 710
4.19.6.411 CoSetExitFlag() . . . 710
4.19.6.412 CoSkip() . . . 710
4.19.6.413 CoProcessBucket() . . . 710
4.19.6.414 CoPushHide() . . . 710
4.19.6.415 CoPopHide() . . . 710
4.19.6.416 CoHide() . . . 711
4.19.6.417 CoIntoHide() . . . 711
4.19.6.418 CoNoHide() . . . 711
4.19.6.419 CoUnHide() . . . 711
4.19.6.420 CoNoUnHide() . . . 711
4.19.6.421 CoSort() . . . 711
4.19.6.422 CoSplitArg() . . . 711
4.19.6.423 CoSplitFirstArg() . . . 712
4.19.6.424 CoSplitLastArg() . . . 712
4.19.6.425 CoSum() . . . 712
4.19.6.426 CoSymbol() . . . 712
4.19.6.427 CoSymmetrize() . . . 712
4.19.6.428 DoTable() . . . 712
4.19.6.429 CoTable() . . . 712
4.19.6.430 CoTerm() . . . 713
4.19.6.431 CoNTable() . . . 713
4.19.6.432 CoCTable() . . . 713
4.19.6.433 EmptyTable() . . . 713
4.19.6.434 CoToTensor() . . . 713
4.19.6.435 CoToVector() . . . 713
4.19.6.436 CoTrace4() . . . 713

4.19.6.437 CoTraceN() . . . 714
4.19.6.438 CoChisholm() . . . 714
4.19.6.439 CoTransform() . . . 714
4.19.6.440 CoClearTable() . . . 714
4.19.6.441 DoChain() . . . 714
4.19.6.442 CoChainin() . . . 714
4.19.6.443 CoChainout() . . . 714
4.19.6.444 CoTryReplace() . . . 715
4.19.6.445 CoVector() . . . 715
4.19.6.446 CoWhile() . . . 715
4.19.6.447 CoWrite() . . . 715
4.19.6.448 CoAuto() . . . 715
4.19.6.449 CoTBaddto() . . . 715
4.19.6.450 CoTBaudit() . . . 715
4.19.6.451 CoTBcleanup() . . . 716
4.19.6.452 CoTBcreate() . . . 716
4.19.6.453 CoTBenter() . . . 716
4.19.6.454 CoTBhelp() . . . 716
4.19.6.455 CoTBload() . . . 716
4.19.6.456 CoTBoff() . . . 716
4.19.6.457 CoTBon() . . . 716
4.19.6.458 CoTBopen() . . . 717
4.19.6.459 CoTBreplace() . . . 717
4.19.6.460 CoTBuse() . . . 717
4.19.6.461 CoTestUse() . . . 717
4.19.6.462 CoThreadBucket() . . . 717
4.19.6.463 AddComString() . . . 717
4.19.6.464 CompileAlgebra() . . . 718
4.19.6.465 IsIdStatement() . . . 718
4.19.6.466 IsRHS() . . . 718
4.19.6.467 ParenthesesTest() . . . 718
4.19.6.468 tokenize() . . . 718
4.19.6.469 WriteTokens() . . . 718
4.19.6.470 simp1token() . . . 719
4.19.6.471 simpwtoken() . . . 719
4.19.6.472 simp2token() . . . 719
4.19.6.473 simp3atoken() . . . 719
4.19.6.474 simp3btoken() . . . 719
4.19.6.475 simp4token() . . . 719
4.19.6.476 simp5token() . . . 720
4.19.6.477 simp6token() . . . 720
4.19.6.478 SkipAName() . . . 720

4.19.6.479 GetLabel() . . . 720
4.19.6.480 ColdExpression() . . . 720
4.19.6.481 CoAssign() . . . 720
4.19.6.482 DoExpr() . . . 721
4.19.6.483 CompileSubExpressions() . . . 721
4.19.6.484 CodeGenerator() . . . 721
4.19.6.485 CompleteTerm() . . . 721
4.19.6.486 CodeFactors() . . . 721
4.19.6.487 GenerateFactors() . . . 721
4.19.6.488 InsTree() . . . 722
4.19.6.489 FindTree() . . . 722
4.19.6.490 RedoTree() . . . 722
4.19.6.491 ClearTree() . . . 722
4.19.6.492 CatchDollar() . . . 722
4.19.6.493 AssignDollar() . . . 722
4.19.6.494 WriteDollarToBuffer() . . . 723
4.19.6.495 WriteDollarFactorToBuffer() . . . 723
4.19.6.496 AddToDollarBuffer() . . . 723
4.19.6.497 TermAssign() . . . 723
4.19.6.498 WildDollars() . . . 723
4.19.6.499 numcommute() . . . 723
4.19.6.500 FullRenumber() . . . 724
4.19.6.501 Lus() . . . 724
4.19.6.502 FindLus() . . . 724
4.19.6.503 CoReplaceLoop() . . . 724
4.19.6.504 CoFindLoop() . . . 724
4.19.6.505 DoFindLoop() . . . 724
4.19.6.506 CoFunPowers() . . . 725
4.19.6.507 SortTheList() . . . 725
4.19.6.508 MatchIsPossible() . . . 725
4.19.6.509 StudyPattern() . . . 725
4.19.6.510 DolToTensor() . . . 725
4.19.6.511 DolToFunction() . . . 725
4.19.6.512 DolToVector() . . . 725
4.19.6.513 DolToNumber() . . . 726
4.19.6.514 DolToSymbol() . . . 726
4.19.6.515 DolToIndex() . . . 726
4.19.6.516 DolToLong() . . . 726
4.19.6.517 DollarFactorize() . . . 726
4.19.6.518 CoPrintTable() . . . 726
4.19.6.519 CoDeallocateTable() . . . 726
4.19.6.520 CleanDollarFactors() . . . 727

4.19.6.521 TakeDollarContent() . . . 727
4.19.6.522 MakeDollarInteger() . . . 727
4.19.6.523 MakeDollarMod() . . . 728
4.19.6.524 GetDolNum() . . . 728
4.19.6.525 AddPotModdollar() . . . 728
4.19.6.526 TakeLongRoot() . . . 729
4.19.6.527 TakeRatRoot() . . . 729
4.19.6.528 MakeRational() . . . 729
4.19.6.529 MakeLongRational() . . . 729
4.19.6.530 ClearTableTree() . . . 730
4.19.6.531 InsTableTree() . . . 730
4.19.6.532 RedoTableTree() . . . 730
4.19.6.533 FindTableTree() . . . 730
4.19.6.534 finishcbuf() . . . 730
4.19.6.535 clearcbuf() . . . 731
4.19.6.536 CleanUpSort() . . . 731
4.19.6.537 AllocFileHandle() . . . 731
4.19.6.538 DeAllocFileHandle() . . . 731
4.19.6.539 PolyRatFunSpecial() . . . 731
4.19.6.540 InsideDollar() . . . 732
4.19.6.541 DolToTerms() . . . 732
4.19.6.542 EvalDoLoopArg() . . . 732
4.19.6.543 SetExprCases() . . . 733
4.19.6.544 TestSelect() . . . 733
4.19.6.545 SubsInAll() . . . 733
4.19.6.546 TransferBuffer() . . . 733
4.19.6.547 TakeIDfunction() . . . 733
4.19.6.548 ExchangeExpressions() . . . 733
4.19.6.549 ExchangeDollars() . . . 734
4.19.6.550 GetFirstBracket() . . . 734
4.19.6.551 GetFirstTerm() . . . 734
4.19.6.552 GetContent() . . . 734
4.19.6.553 CleanupTerm() . . . 734
4.19.6.554 ContentMerge() . . . 734
4.19.6.555 PreIfDollarEval() . . . 735
4.19.6.556 TermsInDollar() . . . 735
4.19.6.557 TermsInExpression() . . . 735
4.19.6.558 TranslateExpression() . . . 735
4.19.6.559 IsSetMember() . . . 735
4.19.6.560 IsMultipleOf() . . . 735
4.19.6.561 TwoExprCompare() . . . 736
4.19.6.562 FindBracket() . . . 736

4.19.6.563 PutBracketInIndex() . . . 736
4.19.6.564 ClearBracketIndex() . . . 736
4.19.6.565 OpenBracketIndex() . . . 736
4.19.6.566 DoNoParallel() . . . 736
4.19.6.567 DoParallel() . . . 737
4.19.6.568 DoModSum() . . . 737
4.19.6.569 DoModMax() . . . 737
4.19.6.570 DoModMin() . . . 737
4.19.6.571 DoModLocal() . . . 737
4.19.6.572 DoModDollar() . . . 737
4.19.6.573 DoProcessBucket() . . . 737
4.19.6.574 DoinParallel() . . . 738
4.19.6.575 DonotinParallel() . . . 738
4.19.6.576 FlipTable() . . . 738
4.19.6.577 ChainIn() . . . 738
4.19.6.578 ChainOut() . . . 738
4.19.6.579 ArgumentImplode() . . . 738
4.19.6.580 ArgumentExplode() . . . 739
4.19.6.581 DenToFunction() . . . 739
4.19.6.582 HowMany() . . . 739
4.19.6.583 CountTerms1() . . . 739
4.19.6.584 TermsInBracket() . . . 739
4.19.6.585 str_dup() . . . 739
4.19.6.586 convertblock() . . . 740
4.19.6.587 convertnamesblock() . . . 740
4.19.6.588 convertiniinfo() . . . 740
4.19.6.589 ReadIndex() . . . 740
4.19.6.590 WriteIndexBlock() . . . 740
4.19.6.591 WriteNamesBlock() . . . 740
4.19.6.592 WriteIndex() . . . 741
4.19.6.593 WriteIniInfo() . . . 741
4.19.6.594 ReadIniInfo() . . . 741
4.19.6.595 AddToIndex() . . . 741
4.19.6.596 GetDbase() . . . 741
4.19.6.597 OpenDbase() . . . 741
4.19.6.598 ReadObject() . . . 741
4.19.6.599 ReadijObject() . . . 742
4.19.6.600 ExistsObject() . . . 742
4.19.6.601 DeleteObject() . . . 742
4.19.6.602 WriteObject() . . . 742
4.19.6.603 AddObject() . . . 742
4.19.6.604 NewDbase() . . . 743

4.19.6.605 FreeTableBase() . . . 743
4.19.6.606 ComposeTableNames() . . . 743
4.19.6.607 PutTableNames() . . . 743
4.19.6.608 AddTableName() . . . 743
4.19.6.609 GetTableName() . . . 743
4.19.6.610 FindTableNumber() . . . 744
4.19.6.611 CopyExpression() . . . 744
4.19.6.612 set_in() . . . 744
4.19.6.613 set_set() . . . 744
4.19.6.614 set_del() . . . 744
4.19.6.615 set_sub() . . . 744
4.19.6.616 DoPreAddSeparator() . . . 745
4.19.6.617 DoPreRmSeparator() . . . 745
4.19.6.618 openExternalChannel() . . . 745
4.19.6.619 initPresetExternalChannels() . . . 745
4.19.6.620 closeExternalChannel() . . . 745
4.19.6.621 selectExternalChannel() . . . 745
4.19.6.622 defineChannel() . . . 746
4.19.6.623 writeToChannel() . . . 746
4.19.6.624 SymbolNormalize() . . . 746
4.19.6.625 TestFunFlag() . . . 746
4.19.6.626 CompareSymbols() . . . 746
4.19.6.627 CompareHSymbols() . . . 747
4.19.6.628 NextPrime() . . . 747
4.19.6.629 wranf() . . . 747
4.19.6.630 iranf() . . . 747
4.19.6.631 iniwranf() . . . 747
4.19.6.632 PreRandom() . . . 748
4.19.6.633 TreatPolyRatFun() . . . 748
4.19.6.634 ReadSaveIndex() . . . 748
4.19.6.635 ReadSaveExpression() . . . 748
4.19.6.636 ReadSaveTerm32() . . . 749
4.19.6.637 ReadSaveVariables() . . . 750
4.19.6.638 WriteStoreHeader() . . . 751
4.19.6.639 DoRecovery() . . . 751
4.19.6.640 DoCheckpoint() . . . 752
4.19.6.641 NumberMallocAddMemory() . . . 753
4.19.6.642 CacheNumberMallocAddMemory() . . . 753
4.19.6.643 TermMallocAddMemory() . . . 753
4.19.6.644 ExprStatus() . . . 753
4.19.6.645 iniTools() . . . 754
4.19.6.646 TestTerm() . . . 754

4.19.6.647 RunTransform() . . . 754
4.19.6.648 RunEncode() . . . 755
4.19.6.649 RunDecode() . . . 755
4.19.6.650 RunReplace() . . . 755
4.19.6.651 RunImplode() . . . 755
4.19.6.652 RunExplode() . . . 755
4.19.6.653 TestArgNum() . . . 755
4.19.6.654 PutArgInScratch() . . . 756
4.19.6.655 ReadRange() . . . 756
4.19.6.656 FindRange() . . . 756
4.19.6.657 RunPermute() . . . 756
4.19.6.658 RunReverse() . . . 756
4.19.6.659 RunCycle() . . . 756
4.19.6.660 RunAddArg() . . . 757
4.19.6.661 RunMulArg() . . . 757
4.19.6.662 RunIsLyndon() . . . 757
4.19.6.663 RunToLyndon() . . . 757
4.19.6.664 RunDropArg() . . . 757
4.19.6.665 RunSelectArg() . . . 757
4.19.6.666 RunDedup() . . . 758
4.19.6.667 NormPolyTerm() . . . 758
4.19.6.668 ConvertToPoly() . . . 758
4.19.6.669 LocalConvertToPoly() . . . 758
4.19.6.670 ConvertFromPoly() . . . 758
4.19.6.671 FindSubterm() . . . 759
4.19.6.672 FindLocalSubterm() . . . 759
4.19.6.673 PrintSubtermList() . . . 759
4.19.6.674 PrintExtraSymbol() . . . 759
4.19.6.675 FindSubexpression() . . . 759
4.19.6.676 CoToPolynomial() . . . 759
4.19.6.677 CoFromPolynomial() . . . 760
4.19.6.678 CoArgToExtraSymbol() . . . 760
4.19.6.679 CoExtraSymbols() . . . 760
4.19.6.680 GetDoParam() . . . 760
4.19.6.681 GetIfDollarFactor() . . . 760
4.19.6.682 CoDo() . . . 760
4.19.6.683 CoEndDo() . . . 761
4.19.6.684 ExtraSymFun() . . . 761
4.19.6.685 PruneExtraSymbols() . . . 761
4.19.6.686 IniFbuffer() . . . 761
4.19.6.687 GCDfunction() . . . 761
4.19.6.688 GCDfunction3() . . . 761

4.19.6.689 ReadPolyRatFun() . . . 762
4.19.6.690 FromPolyRatFun() . . . 762
4.19.6.691 PolyDiv() . . . 762
4.19.6.692 GCDclean() . . . 762
4.19.6.693 TakeSymbolContent() . . . 762
4.19.6.694 GCDterms() . . . 763
4.19.6.695 PutExtraSymbols() . . . 763
4.19.6.696 TakeExtraSymbols() . . . 763
4.19.6.697 MultiplyWithTerm() . . . 763
4.19.6.698 TakeContent() . . . 763
4.19.6.699 MergeSymbolLists() . . . 764
4.19.6.700 MergeDotproductLists() . . . 764
4.19.6.701 CreateExpression() . . . 764
4.19.6.702 DIVfunction() . . . 764
4.19.6.703 MULfunc() . . . 765
4.19.6.704 ConvertArgument() . . . 765
4.19.6.705 ExpandRat() . . . 765
4.19.6.706 InvPoly() . . . 765
4.19.6.707 TestDoLoop() . . . 765
4.19.6.708 TestEndDoLoop() . . . 765
4.19.6.709 DoPreAdd() . . . 766
4.19.6.710 DoPreUseDictionary() . . . 766
4.19.6.711 DoPreCloseDictionary() . . . 766
4.19.6.712 DoPreOpenDictionary() . . . 766
4.19.6.713 RemoveDictionary() . . . 766
4.19.6.714 UnSetDictionary() . . . 766
4.19.6.715 SetDictionaryOptions() . . . 766
4.19.6.716 AddToDictionary() . . . 767
4.19.6.717 AddDictionary() . . . 767
4.19.6.718 FindDictionary() . . . 767
4.19.6.719 IsExponentSign() . . . 767
4.19.6.720 IsMultiplySign() . . . 767
4.19.6.721 TransformRational() . . . 767
4.19.6.722 WriteDictionary() . . . 768
4.19.6.723 ShrinkDictionary() . . . 768
4.19.6.724 FindSymbol() . . . 768
4.19.6.725 FindVector() . . . 768
4.19.6.726 FindIndex() . . . 768
4.19.6.727 FindFunction() . . . 768
4.19.6.728 FindFunWithArgs() . . . 768
4.19.6.729 FindExtraSymbol() . . . 769
4.19.6.730 DictToBytes() . . . 769

4.19.6.731 DictFromBytes() . . . 769
4.19.6.732 CoCreateSpectator() . . . 769
4.19.6.733 CoToSpectator() . . . 769
4.19.6.734 CoRemoveSpectator() . . . 769
4.19.6.735 CoEmptySpectator() . . . 769
4.19.6.736 CoCopySpectator() . . . 770
4.19.6.737 PutInSpectator() . . . 770
4.19.6.738 ClearSpectators() . . . 770
4.19.6.739 GetFromSpectator() . . . 770
4.19.6.740 FlushSpectators() . . . 770
4.19.6.741 ReadFromScratch() . . . 770
4.19.6.742 AddToScratch() . . . 771
4.19.6.743 DoPreAppendPath() . . . 771
4.19.6.744 DoPrePrependPath() . . . 771
4.20 declare.h . . . 771
4.21 dict.c File Reference . . . 789
4.21.1 Detailed Description . . . 790
4.21.2 Function Documentation . . . 790
4.21.2.1 TransformRational() . . . 790
4.21.2.2 IsMultiplySign() . . . 790
4.21.2.3 IsExponentSign() . . . 791
4.21.2.4 FindSymbol() . . . 791
4.21.2.5 FindVector() . . . 791
4.21.2.6 FindIndex() . . . 791
4.21.2.7 FindFunction() . . . 791
4.21.2.8 FindFunWithArgs() . . . 791
4.21.2.9 FindExtraSymbol() . . . 791
4.21.2.10 FindDictionary() . . . 792
4.21.2.11 AddDictionary() . . . 792
4.21.2.12 AddToDictionary() . . . 792
4.21.2.13 UseDictionary() . . . 792
4.21.2.14 SetDictionaryOptions() . . . 792
4.21.2.15 UnSetDictionary() . . . 792
4.21.2.16 RemoveDictionary() . . . 793
4.21.2.17 ShrinkDictionary() . . . 793
4.21.2.18 DoPreOpenDictionary() . . . 793
4.21.2.19 DoPreCloseDictionary() . . . 793
4.21.2.20 DoPreUseDictionary() . . . 793
4.21.2.21 DoPreAdd() . . . 793
4.21.2.22 DictToBytes() . . . 793
4.21.2.23 DictFromBytes() . . . 794
4.22 dict.c . . . 794

4.23 dollar.c File Reference . . . 807
4.23.1 Detailed Description . . . 808
4.23.2 Macro Definition Documentation . . . 808
4.23.2.1 STEP2 . . . 808
4.23.3 Function Documentation . . . 809
4.23.3.1 CatchDollar() . . . 809
4.23.3.2 AssignDollar() . . . 809
4.23.3.3 WriteDollarToBuffer() . . . 809
4.23.3.4 WriteDollarFactorToBuffer() . . . 809
4.23.3.5 AddToDollarBuffer() . . . 809
4.23.3.6 TermAssign() . . . 809
4.23.3.7 WildDollars() . . . 810
4.23.3.8 DolToTensor() . . . 810
4.23.3.9 DolToFunction() . . . 810
4.23.3.10 DolToVector() . . . 810
4.23.3.11 DolToNumber() . . . 810
4.23.3.12 DolToSymbol() . . . 810
4.23.3.13 DolToIndex() . . . 810
4.23.3.14 DolToTerms() . . . 811
4.23.3.15 DolToLong() . . . 811
4.23.3.16 ExecInside() . . . 811
4.23.3.17 InsideDollar() . . . 811
4.23.3.18 ExchangeDollars() . . . 811
4.23.3.19 TermsInDollar() . . . 811
4.23.3.20 PreIfDollarEval() . . . 812
4.23.3.21 TranslateExpression() . . . 812
4.23.3.22 IsSetMember() . . . 812
4.23.3.23 IsMultipleOf() . . . 812
4.23.3.24 TwoExprCompare() . . . 812
4.23.3.25 DollarRaiseLow() . . . 812
4.23.3.26 EvalDoLoopArg() . . . 813
4.23.3.27 TestDoLoop() . . . 813
4.23.3.28 TestEndDoLoop() . . . 813
4.23.3.29 DollarFactorize() . . . 814
4.23.3.30 CleanDollarFactors() . . . 814
4.23.3.31 TakeDollarContent() . . . 814
4.23.3.32 MakeDollarInteger() . . . 814
4.23.3.33 MakeDollarMod() . . . 815
4.23.3.34 GetDolNum() . . . 816
4.23.3.35 AddPotModdollar() . . . 816
4.24 dollar.c . . . 817
4.25 execute.c File Reference . . . 861

4.25.1 Detailed Description . . . 862
4.25.2 Macro Definition Documentation . . . 862
4.25.2.1 CURRENTBRACKET . . . 862
4.25.2.2 BRACKETCURRENTEXPR . . . 862
4.25.2.3 BRACKETOTHEREXPR . . . 862
4.25.2.4 NOBRACKETACTIVE . . . 862
4.25.3 Function Documentation . . . 862
4.25.3.1 CleanExpr() . . . 862
4.25.3.2 PopVariables() . . . 862
4.25.3.3 MakeGlobal() . . . 863
4.25.3.4 TestDrop() . . . 863
4.25.3.5 PutInVflags() . . . 863
4.25.3.6 DoExecute() . . . 863
4.25.3.7 PutBracket() . . . 863
4.25.3.8 SpecialCleanup() . . . 863
4.25.3.9 SetMods() . . . 863
4.25.3.10 UnSetMods() . . . 864
4.25.3.11 ExchangeExpressions() . . . 864
4.25.3.12 GetFirstBracket() . . . 864
4.25.3.13 GetFirstTerm() . . . 864
4.25.3.14 GetContent() . . . 864
4.25.3.15 CleanupTerm() . . . 864
4.25.3.16 ContentMerge() . . . 865
4.25.3.17 TermsInExpression() . . . 865
4.25.3.18 UpdatePositions() . . . 865
4.25.3.19 CountTerms1() . . . 865
4.25.3.20 TermsInBracket() . . . 865
4.26 execute.c . . . 866
4.27 extcmd.c File Reference . . . 896
4.27.1 Detailed Description . . . 896
4.27.2 Function Documentation . . . 896
4.27.2.1 openExternalChannel() . . . 896
4.27.2.2 initPresetExternalChannels() . . . 896
4.27.2.3 closeExternalChannel() . . . 897
4.27.2.4 selectExternalChannel() . . . 897
4.27.2.5 getCurrentExternalChannel() . . . 897
4.27.2.6 closeAllExternalChannels() . . . 897
4.28 extcmd.c . . . 897
4.29 factor.c File Reference . . . 916
4.29.1 Detailed Description . . . 917
4.29.2 Function Documentation . . . 917
4.29.2.1 FactorIn() . . . 917

4.29.2.2 FactorInExpr() . . . 917
4.30 factor.c . . . 917
4.31 findpat.c File Reference . . . 928
4.31.1 Detailed Description . . . 928
4.31.2 Function Documentation . . . 928
4.31.2.1 FindOnly() . . . 928
4.31.2.2 FindOnce() . . . 928
4.31.2.3 FindMulti() . . . 929
4.31.2.4 FindRest() . . . 929
4.32 findpat.c . . . 929
4.33 form3.h File Reference . . . 945
4.33.1 Detailed Description . . . 947
4.33.2 Macro Definition Documentation . . . 947
4.33.2.1 MAJORVERSION . . . 947
4.33.2.2 MINORVERSION . . . 947
4.33.2.3 PRODUCTIONDATE . . . 948
4.33.2.4 inline . . . 948
4.33.2.5 NDEBUG . . . 948
4.33.2.6 STATIC_ASSERT . . . 948
4.33.2.7 STATIC_ASSERT__1 . . . 948
4.33.2.8 STATIC_ASSERT__2 . . . 948
4.33.2.9 STATIC_ASSERT__3 . . . 948
4.33.2.10 TOPBITONLY . . . 949
4.33.2.11 TOPLONGBITONLY . . . 949
4.33.2.12 SPECMASK . . . 949
4.33.2.13 WILDMASK . . . 949
4.33.2.14 WORDMASK . . . 949
4.33.2.15 AWORDMASK . . . 949
4.33.2.16 FULLMAX . . . 949
4.33.2.17 MAXPOSITIVE . . . 949
4.33.2.18 MAXLONG . . . 950
4.33.2.19 MAXPOSITIVE2 . . . 950
4.33.2.20 MAXPOSITIVE4 . . . 950
4.33.2.21 alignof . . . 950
4.33.2.22 PADDUMMY . . . 950
4.33.2.23 PADPOSITION . . . 950
4.33.2.24 PADPOINTER . . . 951
4.33.2.25 PADLONG . . . 951
4.33.2.26 PADINT . . . 951
4.33.2.27 PADWORD . . . 951
4.33.2.28 WITHSORTBOTS . . . 952
4.33.2.29 FILES . . . 952

4.33.2.30 Uopen . . . 952
4.33.2.31 Uflush . . . 952
4.33.2.32 Uclose . . . 952
4.33.2.33 Uread . . . 952
4.33.2.34 Uwrite . . . 953
4.33.2.35 Usetbuf . . . 953
4.33.2.36 Useek . . . 953
4.33.2.37 Utell . . . 953
4.33.2.38 Ugetpos . . . 953
4.33.2.39 Usetpos . . . 953
4.33.2.40 Usync . . . 954
4.33.2.41 Utruncate . . . 954
4.33.2.42 Ustdout . . . 954
4.33.2.43 MAX_OPEN_FILES . . . 954
4.33.2.44 bzero . . . 954
4.33.2.45 GetPID . . . 954
4.33.3 Typedef Documentation . . . 954
4.33.3.1 VOID . . . 954
4.33.3.2 SBYTE . . . 955
4.33.3.3 UBYTE . . . 955
4.33.3.4 UINT . . . 955
4.33.3.5 RLONG . . . 955
4.33.3.6 MLONG . . . 955
4.34 form3.h . . . 955
4.35 fsizes.h File Reference . . . 961
4.35.1 Detailed Description . . . 963
4.35.2 Macro Definition Documentation . . . 963
4.35.2.1 MAXPRENAMESIZE . . . 963
4.35.2.2 MAXPOWER . . . 963
4.35.2.3 MAXVARIABLES . . . 963
4.35.2.4 MAXDOLLARVARIABLES . . . 963
4.35.2.5 WILDOFFSET . . . 963
4.35.2.6 MAXINNAMETREE . . . 964
4.35.2.7 MAXDUMMIES . . . 964
4.35.2.8 WORKBUFFER . . . 964
4.35.2.9 MAXTER . . . 964
4.35.2.10 HALFMAX . . . 964
4.35.2.11 MAXSUBEXPRESSIONS . . . 964
4.35.2.12 MAXFILESTREAMSIZE . . . 964
4.35.2.13 MAXENAME . . . 964
4.35.2.14 MAXSAVEFUNCTION . . . 965
4.35.2.15 MAXPARLEVEL . . . 965

4.35.2.16 MAXNUMBERSIZE . . . 965
4.35.2.17 MAXREPEAT . . . 965
4.35.2.18 NORMSIZE . . . 965
4.35.2.19 INITNODESIZE . . . 965
4.35.2.20 INITNAMESIZE . . . 965
4.35.2.21 NUMFIXED . . . 965
4.35.2.22 MAXNEST . . . 966
4.35.2.23 MAXMATCH . . . 966
4.35.2.24 MAXIF . . . 966
4.35.2.25 SIZEFACS . . . 966
4.35.2.26 NUMFACS . . . 966
4.35.2.27 MAXLOOPS . . . 966
4.35.2.28 MAXLABELS . . . 966
4.35.2.29 COMMERCIALSIZE . . . 966
4.35.2.30 MAXFLAGS . . . 967
4.35.2.31 COMPRESSBUFFER . . . 967
4.35.2.32 FORTRANCONTINUATIONLINES . . . 967
4.35.2.33 MAXLEVELS . . . 967
4.35.2.34 MAXLHS . . . 967
4.35.2.35 MAXWILDC . . . 967
4.35.2.36 NUMTABLEENTRIES . . . 967
4.35.2.37 COMPILERBUFFER . . . 967
4.35.2.38 SMALLBUFFER . . . 968
4.35.2.39 SMALLOVERFLOW . . . 968
4.35.2.40 TERMSSMALL . . . 968
4.35.2.41 LARGEBUFFER . . . 968
4.35.2.42 MAXPATCHES . . . 968
4.35.2.43 MAXFPATCHES . . . 968
4.35.2.44 SORTIOSIZE . . . 968
4.35.2.45 SSMALLBUFFER . . . 968
4.35.2.46 SSMALLOVERFLOW . . . 969
4.35.2.47 STERMSSMALL . . . 969
4.35.2.48 SLARGEBUFFER . . . 969
4.35.2.49 SMAXPATCHES . . . 969
4.35.2.50 SMAXFPATCHES . . . 969
4.35.2.51 SSORTIOSIZE . . . 969
4.35.2.52 SCRATCHSIZE . . . 969
4.35.2.53 SPECTATORSIZE . . . 969
4.35.2.54 MAXFLEVELS . . . 970
4.35.2.55 COMPINC . . . 970
4.35.2.56 MAXNUMSIZE . . . 970
4.35.2.57 MAXBRACKETBUFFERSIZE . . . 970

4.35.2.58 SFHSIZE . . . 970
4.35.2.59 DEFAULTPROCESSBUCKETSIZE . . . 970
4.35.2.60 SHMWINSIZE . . . 970
4.35.2.61 TABLEEXTENSION . . . 970
4.35.2.62 GZIPDEFAULT . . . 971
4.35.2.63 DEFAULTTHREADS . . . 971
4.35.2.64 DEFAULTTHREADBUCKETSIZE . . . 971
4.35.2.65 DEFAULTTHREADLOADBALANCING . . . 971
4.35.2.66 THREADSCRATCHSIZE . . . 971
4.35.2.67 THREADSCRATCHOUTSIZE . . . 971
4.35.2.68 MAXTABLECOMBUF . . . 971
4.35.2.69 MAXCOMBUFRHS . . . 971
4.35.2.70 NUMSTORECACHES . . . 972
4.35.2.71 SIZESTORECACHE . . . 972
4.35.2.72 INDENTSPACE . . . 972
4.35.2.73 MULTIINDENTSPACE . . . 972
4.35.2.74 MAXMULTIBRACKETLEVELS . . . 972
4.35.2.75 FBUFFERSIZE . . . 972
4.35.2.76 NPAIR1 . . . 972
4.35.2.77 NPAIR2 . . . 972
4.35.2.78 MAXLINELENGTH . . . 973
4.36 fsizes.h . . . 973
4.37 ftypes.h File Reference . . . 975
4.37.1 Detailed Description . . . 986
4.37.2 Macro Definition Documentation . . . 986
4.37.2.1 PHEAD . . . 986
4.37.2.2 PHEAD0 . . . 987
4.37.2.3 BHEAD . . . 987
4.37.2.4 BHEAD0 . . . 987
4.37.2.5 WITHOUTERROR . . . 987
4.37.2.6 WITHERROR . . . 987
4.37.2.7 FILESTREAM . . . 987
4.37.2.8 PREVARSTREAM . . . 987
4.37.2.9 PREREADSTREAM . . . 987
4.37.2.10 PIPESTREAM . . . 988
4.37.2.11 PRECALCSTREAM . . . 988
4.37.2.12 DOLLARSTREAM . . . 988
4.37.2.13 PREREADSTREAM2 . . . 988
4.37.2.14 EXTERNALCHANNELSTREAM . . . 988
4.37.2.15 PREREADSTREAM3 . . . 988
4.37.2.16 REVERSEFILESTREAM . . . 988
4.37.2.17 ENDOFSTREAM . . . 988

4.37.2.18 ENDOFINPUT . . . 989
4.37.2.19 SUBROUTINEFILE . . . 989
4.37.2.20 PROCEDUREFILE . . . 989
4.37.2.21 HEADERFILE . . . 989
4.37.2.22 SETUPFILE . . . 989
4.37.2.23 TABLEBASEFILE . . . 989
4.37.2.24 FIRSTMODULE . . . 989
4.37.2.25 GLOBALMODULE . . . 989
4.37.2.26 SORTMODULE . . . 990
4.37.2.27 STOREMODULE . . . 990
4.37.2.28 CLEARMODULE . . . 990
4.37.2.29 ENDMODULE . . . 990
4.37.2.30 POLYFUN . . . 990
4.37.2.31 NOPARALLEL_DOLLAR . . . 990
4.37.2.32 NOPARALLEL_RHS . . . 990
4.37.2.33 NOPARALLEL_CONVPOLY . . . 990
4.37.2.34 NOPARALLEL_SPECTATOR . . . 991
4.37.2.35 NOPARALLEL_USER . . . 991
4.37.2.36 NOPARALLEL_TBLDOLLAR . . . 991
4.37.2.37 NOPARALLEL_NPROC . . . 991
4.37.2.38 PARALLELFLAG . . . 991
4.37.2.39 PRENOACTION . . . 991
4.37.2.40 PRERAISEAFTER . . . 991
4.37.2.41 PRELOWERAFTER . . . 991
4.37.2.42 WITHSEMICOLON . . . 992
4.37.2.43 WITHOUTSEMICOLON . . . 992
4.37.2.44 MODULEINSTACK . . . 992
4.37.2.45 EXECUTINGIF . . . 992
4.37.2.46 LOOKINGFORELSE . . . 992
4.37.2.47 LOOKINGFORENDIF . . . 992
4.37.2.48 NEWSTATEMENT . . . 992
4.37.2.49 OLDSTATEMENT . . . 992
4.37.2.50 EXECUTINGPRESWITCH . . . 993
4.37.2.51 SEARCHINGPRECASE . . . 993
4.37.2.52 SEARCHINGPREENDSWITCH . . . 993
4.37.2.53 PREPROONLY . . . 993
4.37.2.54 DUMPTOCOMPILER . . . 993
4.37.2.55 DUMPOUTTERMS . . . 993
4.37.2.56 DUMPINTERMS . . . 993
4.37.2.57 DUMPTOSORT . . . 993
4.37.2.58 DUMPTOPARALLEL . . . 994
4.37.2.59 THREADSDEBUG . . . 994

4.37.2.60 ERROROUT . . . 994
4.37.2.61 INPUTOUT . . . 994
4.37.2.62 STATSOUT . . . 994
4.37.2.63 EXPRSOUT . . . 994
4.37.2.64 WRITEOUT . . . 994
4.37.2.65 EXTERNALCHANNELOUT . . . 994
4.37.2.66 NUMERICALLOOP . . . 995
4.37.2.67 LISTEDLOOP . . . 995
4.37.2.68 ONEEXPRESSION . . . 995
4.37.2.69 PRETYPENONE . . . 995
4.37.2.70 PRETYPEIF . . . 995
4.37.2.71 PRETYPEDO . . . 995
4.37.2.72 PRETYPEPROCEDURE . . . 995
4.37.2.73 PRETYPESWITCH . . . 995
4.37.2.74 PRETYPEINSIDE . . . 996
4.37.2.75 DECLARATION . . . 996
4.37.2.76 SPECIFICATION . . . 996
4.37.2.77 DEFINITION . . . 996
4.37.2.78 STATEMENT . . . 996
4.37.2.79 TOOUTPUT . . . 996
4.37.2.80 ATENDOFMODULE . . . 996
4.37.2.81 MIXED . . . 996
4.37.2.82 VOID . . . 997
4.37.2.83 NOAUTO . . . 997
4.37.2.84 PARTEST . . . 997
4.37.2.85 WITHAUTO . . . 997
4.37.2.86 ALLVARIABLES . . . 997
4.37.2.87 SYMBOLONLY . . . 997
4.37.2.88 INDEXONLY . . . 997
4.37.2.89 VECTORONLY . . . 997
4.37.2.90 FUNCTIONONLY . . . 998
4.37.2.91 SETONLY . . . 998
4.37.2.92 EXPRESSIONONLY . . . 998
4.37.2.93 CDELETE . . . 998
4.37.2.94 ANYTYPE . . . 998
4.37.2.95 CSYMBOL . . . 998
4.37.2.96 CINDEX . . . 998
4.37.2.97 CVECTOR . . . 998
4.37.2.98 CFUNCTION . . . 999
4.37.2.99 CSET . . . 999
4.37.2.100 CEXPRESSION . . . 999
4.37.2.101 CDOTPRODUCT . . . 999

4.37.2.102 CNUMBER . . . 999
4.37.2.103 CSUBEXP . . . 999
4.37.2.104 CDELTA . . . 999
4.37.2.105 CDOLLAR . . . 999
4.37.2.106 CDUBIOUS . . . 1000
4.37.2.107 CRANGE . . . 1000
4.37.2.108 CVECTOR1 . . . 1000
4.37.2.109 CDOUBLEDOT . . . 1000
4.37.2.110 TSYMBOL . . . 1000
4.37.2.111 TINDEX . . . 1000
4.37.2.112 TVECTOR . . . 1000
4.37.2.113 TFUNCTION . . . 1000
4.37.2.114 TSET . . . 1001
4.37.2.115 TEXPRESSION . . . 1001
4.37.2.116 TDOTPRODUCT . . . 1001
4.37.2.117 TNUMBER . . . 1001
4.37.2.118 TSUBEXP . . . 1001
4.37.2.119 TDELTA . . . 1001
4.37.2.120 TDOLLAR . . . 1001
4.37.2.121 TDUBIOUS . . . 1001
4.37.2.122 LPARENTHESIS . . . 1002
4.37.2.123 RPARENTHESIS . . . 1002
4.37.2.124 TWILDCARD . . . 1002
4.37.2.125 TWILDARG . . . 1002
4.37.2.126 TDOT . . . 1002
4.37.2.127 LBRACE . . . 1002
4.37.2.128 RBRACE . . . 1002
4.37.2.129 TCOMMA . . . 1002
4.37.2.130 TFUNOPEN . . . 1003
4.37.2.131 TFUNCLOSE . . . 1003
4.37.2.132 TMULTIPLY . . . 1003
4.37.2.133 TDIVIDE . . . 1003
4.37.2.134 TPOWER . . . 1003
4.37.2.135 TPLUS . . . 1003
4.37.2.136 TMINUS . . . 1003
4.37.2.137 TNOT . . . 1003
4.37.2.138 TENDOFIT . . . 1004
4.37.2.139 TSETOPEN . . . 1004
4.37.2.140 TSETCLOSE . . . 1004
4.37.2.141 TGENINDEX . . . 1004
4.37.2.142 TCONJUGATE . . . 1004
4.37.2.143 LRPARENTHESES . . . 1004

4.37.2.144 TNUMBER1 1004
4.37.2.145 TPOWER1 1004
4.37.2.146 TEMPTY 1005
4.37.2.147 TSETNUM 1005
4.37.2.148 TSGAMMA 1005
4.37.2.149 TSETDOL 1005
4.37.2.150 TYPEISFUN 1005
4.37.2.151 TYPEISSUB 1005
4.37.2.152 TYPEISMYSTERY 1005
4.37.2.153 LHSIDEX 1005
4.37.2.154 LHSIDE 1006
4.37.2.155 RHSIDE 1006
4.37.2.156 FORTRANMODE 1006
4.37.2.157 REDUCEMODE 1006
4.37.2.158 MAPLEMODE 1006
4.37.2.159 MATHEMATICAMODE 1006
4.37.2.160 CMODE 1006
4.37.2.161 VORTRANMODE 1006
4.37.2.162 PFORTRANMODE 1007
4.37.2.163 DOUBLEFORTRANMODE 1007
4.37.2.164 DOUBLEPRECISIONFLAG 1007
4.37.2.165 NODOUBLEMASK 1007
4.37.2.166 QUADRUPLEFORTRANMODE 1007
4.37.2.167 QUADRUPLEPRECISIONFLAG 1007
4.37.2.168 NOQUADMASK 1007
4.37.2.169 NORMALFORMAT 1007
4.37.2.170 NOSPACEFORMAT 1008
4.37.2.171 ISNOTFORTRAN90 1008
4.37.2.172 ISFORTRAN90 1008
4.37.2.173 ALSOREVERSE 1008
4.37.2.174 CHISHOLM 1008
4.37.2.175 NOTRICK 1008
4.37.2.176 SORTLOWFIRST 1008
4.37.2.177 SORTHIGHFIRST 1008
4.37.2.178 SORTPOWERFIRST 1009
4.37.2.179 SORTANTIPOWER 1009
4.37.2.180 NMIN4SHIFT 1009
4.37.2.181 SYMBOL 1009
4.37.2.182 DOTPRODUCT 1009
4.37.2.183 VECTOR 1009
4.37.2.184 INDEX 1009
4.37.2.185 EXPRESSION 1009

4.37.2.186 SUBEXPRESSION . . . 1010
4.37.2.187 DOLLAREXPRESSION . . . 1010
4.37.2.188 SETSET . . . 1010
4.37.2.189 ARGWILD . . . 1010
4.37.2.190 MINVECTOR . . . 1010
4.37.2.191 SETEXP . . . 1010
4.37.2.192 DOLLAREXPR2 . . . 1010
4.37.2.193 HAAKJE0 . . . 1010
4.37.2.194 FUNCTION . . . 1011
4.37.2.195 TMPPOLYFUN . . . 1011
4.37.2.196 ARGFIELD . . . 1011
4.37.2.197 SNUMBER . . . 1011
4.37.2.198 LNUMBER . . . 1011
4.37.2.199 HAAKJE . . . 1011
4.37.2.200 DELTA . . . 1011
4.37.2.201 EXPONENT . . . 1011
4.37.2.202 DENOMINATOR . . . 1012
4.37.2.203 SETFUNCTION . . . 1012
4.37.2.204 GAMMA . . . 1012
4.37.2.205 GAMMAI . . . 1012
4.37.2.206 GAMMAFIVE . . . 1012
4.37.2.207 GAMMASIX . . . 1012
4.37.2.208 GAMMASEVEN . . . 1012
4.37.2.209 SUMF1 . . . 1012
4.37.2.210 SUMF2 . . . 1013
4.37.2.211 DUMFUN . . . 1013
4.37.2.212 REPLACEMENT . . . 1013
4.37.2.213 REVERSEFUNCTION . . . 1013
4.37.2.214 DISTRIBUTION . . . 1013
4.37.2.215 DELTA3 . . . 1013
4.37.2.216 DUMMYFUN . . . 1013
4.37.2.217 DUMMYTEN . . . 1013
4.37.2.218 LEVICIVITA . . . 1014
4.37.2.219 FACTORIAL . . . 1014
4.37.2.220 INVERSEFACTORIAL . . . 1014
4.37.2.221 BINOMIAL . . . 1014
4.37.2.222 NUMARGSFUN . . . 1014
4.37.2.223 SIGNFUN . . . 1014
4.37.2.224 MODFUNCTION . . . 1014
4.37.2.225 MOD2FUNCTION . . . 1014
4.37.2.226 MINFUNCTION . . . 1015
4.37.2.227 MAXFUNCTION . . . 1015

4.37.2.228 ABSFUNCTION 1015
4.37.2.229 SIGFUNCTION 1015
4.37.2.230 INTFUNCTION 1015
4.37.2.231 THETA 1015
4.37.2.232 THETA2 1015
4.37.2.233 DELTA2 1015
4.37.2.234 DELTAP 1016
4.37.2.235 BERNOULLIFUNCTION 1016
4.37.2.236 COUNTFUNCTION 1016
4.37.2.237 MATCHFUNCTION 1016
4.37.2.238 PATTERNFUNCTION 1016
4.37.2.239 TERMFUNCTION 1016
4.37.2.240 CONJUGATION 1016
4.37.2.241 ROOTFUNCTION 1016
4.37.2.242 TABLEFUNCTION 1017
4.37.2.243 FIRSTBRACKET 1017
4.37.2.244 TERMSINEXPR 1017
4.37.2.245 NUMTERMSFUN 1017
4.37.2.246 GCDFUNCTION 1017
4.37.2.247 DIVFUNCTION 1017
4.37.2.248 REMFUNCTION 1017
4.37.2.249 MAXPOWEROF 1017
4.37.2.250 MINPOWEROF 1018
4.37.2.251 TABLESTUB 1018
4.37.2.252 FACTORIN 1018
4.37.2.253 TERMSINBRACKET 1018
4.37.2.254 WILDARGFUN 1018
4.37.2.255 SQRTFUNCTION 1018
4.37.2.256 LNFUNCTION 1018
4.37.2.257 SINFUNCTION 1018
4.37.2.258 COSFUNCTION 1019
4.37.2.259 TANFUNCTION 1019
4.37.2.260 ASINFUNCTION 1019
4.37.2.261 ACOSFUNCTION 1019
4.37.2.262 ATANFUNCTION 1019
4.37.2.263 ATAN2FUNCTION 1019
4.37.2.264 SINHFUNCTION 1019
4.37.2.265 COSHFUNCTION 1019
4.37.2.266 TANHFUNCTION 1020
4.37.2.267 ASINHFUNCTION 1020
4.37.2.268 ACOSHFUNCTION 1020
4.37.2.269 ATANHFUNCTION 1020

4.37.2.270 LI2FUNCTION 1020
4.37.2.271 LINFUNCTION 1020
4.37.2.272 EXTRASYMFUN 1020
4.37.2.273 RANDOMFUNCTION 1020
4.37.2.274 RANPERM 1021
4.37.2.275 NUMFACTORS 1021
4.37.2.276 FIRSTTERM 1021
4.37.2.277 CONTENTTERM 1021
4.37.2.278 PRIMENUMBER 1021
4.37.2.279 EXTEUCLIDEAN 1021
4.37.2.280 MAKERATIONAL 1021
4.37.2.281 INVERSEFUNCTION 1021
4.37.2.282 IDFUNCTION 1022
4.37.2.283 PUTFIRST 1022
4.37.2.284 PERMUTATIONS 1022
4.37.2.285 PARTITIONS 1022
4.37.2.286 MULFUNCTION 1022
4.37.2.287 MAXBUILTINFUNCTION 1022
4.37.2.288 FIRSTUSERFUNCTION 1022
4.37.2.289 ISYMBOL 1022
4.37.2.290 PISYMBOL 1023
4.37.2.291 COEFFSYMBOL 1023
4.37.2.292 NUMERATORSYMBOL 1023
4.37.2.293 DENOMINATORSYMBOL 1023
4.37.2.294 WILDARGSYMBOL 1023
4.37.2.295 DIMENSIONSYMBOL 1023
4.37.2.296 FACTORSYMBOL 1023
4.37.2.297 SEPARATESYMBOL 1023
4.37.2.298 BUILTINSYMBOLS 1024
4.37.2.299 FIRSTUSERSYMBOL 1024
4.37.2.300 BUILTININDICES 1024
4.37.2.301 BUILTINVECTORS 1024
4.37.2.302 BUILTINDOLLARS 1024
4.37.2.303 WILDARGVECTOR 1024
4.37.2.304 WILDARGINDEX 1024
4.37.2.305 TYPEEXPRESSION 1024
4.37.2.306 TYPEIDNEW 1025
4.37.2.307 TYPEIDOLD 1025
4.37.2.308 TYPEOPERATION 1025
4.37.2.309 TYPEREPEAT 1025
4.37.2.310 TYPEENDREPEAT 1025
4.37.2.311 TYPECOUNT 1025

4.37.2.312 TYPEMULT . . . 1025
4.37.2.313 TYPEGOTO . . . 1025
4.37.2.314 TYPEDISCARD . . . 1026
4.37.2.315 TYPEIF . . . 1026
4.37.2.316 TYPEELSE . . . 1026
4.37.2.317 TYPEELIF . . . 1026
4.37.2.318 TYPEENDIF . . . 1026
4.37.2.319 TYPESUM . . . 1026
4.37.2.320 TYPECHISHOLM . . . 1026
4.37.2.321 TYPEREVERSE . . . 1026
4.37.2.322 TYPEARG . . . 1027
4.37.2.323 TYPENORM . . . 1027
4.37.2.324 TYPENORM2 . . . 1027
4.37.2.325 TYPENORM3 . . . 1027
4.37.2.326 TYPEEXIT . . . 1027
4.37.2.327 TYPESETEXIT . . . 1027
4.37.2.328 TYPEPRINT . . . 1027
4.37.2.329 TYPEFPRINT . . . 1027
4.37.2.330 TYPEREDEFPRE . . . 1028
4.37.2.331 TYPESPLITARG . . . 1028
4.37.2.332 TYPESPLITARG2 . . . 1028
4.37.2.333 TYPEFACTARG . . . 1028
4.37.2.334 TYPEFACTARG2 . . . 1028
4.37.2.335 TYPETRY . . . 1028
4.37.2.336 TYPEASSIGN . . . 1028
4.37.2.337 TYPERENUMBER . . . 1028
4.37.2.338 TYPESUMFIX . . . 1029
4.37.2.339 TYPEFINDLOOP . . . 1029
4.37.2.340 TYPEUNRAVEL . . . 1029
4.37.2.341 TYPEADJUSTBOUNDS . . . 1029
4.37.2.342 TYPEINSIDE . . . 1029
4.37.2.343 TYPETERM . . . 1029
4.37.2.344 TYPESORT . . . 1029
4.37.2.345 TYPEDETCURDUM . . . 1029
4.37.2.346 TYPEINEXPRESSION . . . 1030
4.37.2.347 TYPESPLITFIRSTARG . . . 1030
4.37.2.348 TYPESPLITLASTARG . . . 1030
4.37.2.349 TYPEMERGE . . . 1030
4.37.2.350 TYPETESTUSE . . . 1030
4.37.2.351 TYPEAPPLY . . . 1030
4.37.2.352 TYPEAPPLYRESET . . . 1030
4.37.2.353 TYPECHAININ . . . 1030

4.37.2.354 TYPECHAINOUT . . . 1031
4.37.2.355 TYPENORM4 . . . 1031
4.37.2.356 TYPEFACTOR . . . 1031
4.37.2.357 TYPEARGIMPLODE . . . 1031
4.37.2.358 TYPEARGEXPLODE . . . 1031
4.37.2.359 TYPEDENOMINATORS . . . 1031
4.37.2.360 TYPESTUFFLE . . . 1031
4.37.2.361 TYPEDROPCOEFFICIENT . . . 1031
4.37.2.362 TYPETRANSFORM . . . 1032
4.37.2.363 TYPETOPOLYNOMIAL . . . 1032
4.37.2.364 TYPEFROMPOLYNOMIAL . . . 1032
4.37.2.365 TYPEDOLOOP . . . 1032
4.37.2.366 TYPEENDDOLOOP . . . 1032
4.37.2.367 TYPEDROPSYMBOLS . . . 1032
4.37.2.368 TYPEPUTINSIDE . . . 1032
4.37.2.369 TYPETOSPECTATOR . . . 1032
4.37.2.370 TYPEARGTOEXTRASYMBOL . . . 1033
4.37.2.371 TAKETRACE . . . 1033
4.37.2.372 CONTRACT . . . 1033
4.37.2.373 RATIO . . . 1033
4.37.2.374 SYMMETRIZE . . . 1033
4.37.2.375 TENVEC . . . 1033
4.37.2.376 SUMNUM1 . . . 1033
4.37.2.377 SUMNUM2 . . . 1033
4.37.2.378 WILDDUMMY . . . 1034
4.37.2.379 SYMTONUM . . . 1034
4.37.2.380 SYMTOSYM . . . 1034
4.37.2.381 SYMTOSUB . . . 1034
4.37.2.382 VECTOMIN . . . 1034
4.37.2.383 VECTOVEC . . . 1034
4.37.2.384 VECTOSUB . . . 1034
4.37.2.385 INDTOIND . . . 1034
4.37.2.386 INDTOSUB . . . 1035
4.37.2.387 FUNTOFUN . . . 1035
4.37.2.388 ARGTOARG . . . 1035
4.37.2.389 ARLTOARL . . . 1035
4.37.2.390 EXPTOEXP . . . 1035
4.37.2.391 FROMBRAC . . . 1035
4.37.2.392 FROMSET . . . 1035
4.37.2.393 SETTONUM . . . 1035
4.37.2.394 WILDCARDS . . . 1036
4.37.2.395 SETNUMBER . . . 1036

4.37.2.396 LOADDOLLAR . . . 1036
4.37.2.397 NUMTONUM . . . 1036
4.37.2.398 NUMTOSYM . . . 1036
4.37.2.399 NUMTOIND . . . 1036
4.37.2.400 NUMTOSUB . . . 1036
4.37.2.401 CLEANFLAG . . . 1036
4.37.2.402 DIRTYFLAG . . . 1037
4.37.2.403 DIRTYSYMFLAG . . . 1037
4.37.2.404 MUSTCLEANPRF . . . 1037
4.37.2.405 ALLDIRTY . . . 1037
4.37.2.406 ARGHEAD . . . 1037
4.37.2.407 FUNHEAD . . . 1037
4.37.2.408 SUBEXPSIZE . . . 1037
4.37.2.409 EXPRHEAD . . . 1037
4.37.2.410 TYPEARGHEADSIZE . . . 1038
4.37.2.411 SKIP . . . 1038
4.37.2.412 DROP . . . 1038
4.37.2.413 HIDE . . . 1038
4.37.2.414 UNHIDE . . . 1038
4.37.2.415 INTOHIDE . . . 1038
4.37.2.416 LOCALEXPRESSION . . . 1038
4.37.2.417 SKIPLEXPRESSION . . . 1038
4.37.2.418 DROPLEXPRESSION . . . 1039
4.37.2.419 DROPPEDEXPRESSION . . . 1039
4.37.2.420 GLOBALEXPRESSION . . . 1039
4.37.2.421 SKIPGEXPRESSION . . . 1039
4.37.2.422 DROPGEXPRESSION . . . 1039
4.37.2.423 STOREDEXPRESSION . . . 1039
4.37.2.424 HIDDENLEXPRESSION . . . 1039
4.37.2.425 HIDDENGEXPRESSION . . . 1039
4.37.2.426 INCEXPRESSION . . . 1040
4.37.2.427 HIDELEXPRESSION . . . 1040
4.37.2.428 HIDEGEXPRESSION . . . 1040
4.37.2.429 DROPHLEXPRESSION . . . 1040
4.37.2.430 DROPHGEXPRESSION . . . 1040
4.37.2.431 UNHIDELEXPRESSION . . . 1040
4.37.2.432 UNHIDEGEXPRESSION . . . 1040
4.37.2.433 INTOHIDELEXPRESSION . . . 1040
4.37.2.434 INTOHIDEGEXPRESSION . . . 1041
4.37.2.435 SPECTATOREXPRESSION . . . 1041
4.37.2.436 DROPSPECTATOREXPRESSION . . . 1041
4.37.2.437 SKIPUNHIDELEXPRESSION . . . 1041

4.37.2.438 SKIPUNHIDEGEXPRESSION . . . 1041
4.37.2.439 PRINTOFF . . . 1041
4.37.2.440 PRINTON . . . 1041
4.37.2.441 PRINTCONTENTS . . . 1041
4.37.2.442 PRINTCONTENT . . . 1042
4.37.2.443 PRINTLFILE . . . 1042
4.37.2.444 PRINTONETERM . . . 1042
4.37.2.445 PRINTONEFUNCTION . . . 1042
4.37.2.446 PRINTALL . . . 1042
4.37.2.447 REGULAREXPRESSION . . . 1042
4.37.2.448 REDEFINEDEXPRESSION . . . 1042
4.37.2.449 NEWLYDEFINEDEXPRESSION . . . 1042
4.37.2.450 GENERALFUNCTION . . . 1043
4.37.2.451 FASTFUNCTION . . . 1043
4.37.2.452 TENSORFUNCTION . . . 1043
4.37.2.453 GAMMAFUNCTION . . . 1043
4.37.2.454 POS_ . . . 1043
4.37.2.455 POS0_ . . . 1043
4.37.2.456 NEG_ . . . 1043
4.37.2.457 NEG0_ . . . 1043
4.37.2.458 EVEN_ . . . 1044
4.37.2.459 ODD_ . . . 1044
4.37.2.460 Z_ . . . 1044
4.37.2.461 SYMBOL_ . . . 1044
4.37.2.462 FIXED_ . . . 1044
4.37.2.463 INDEX_ . . . 1044
4.37.2.464 Q_ . . . 1044
4.37.2.465 DUMMYINDEX_ . . . 1044
4.37.2.466 VECTOR_ . . . 1045
4.37.2.467 GAMMA1 . . . 1045
4.37.2.468 GAMMA5 . . . 1045
4.37.2.469 GAMMA6 . . . 1045
4.37.2.470 GAMMA7 . . . 1045
4.37.2.471 FUNNYVEC . . . 1045
4.37.2.472 FUNNYWILD . . . 1045
4.37.2.473 SUMMEDIND . . . 1045
4.37.2.474 NOINDEX . . . 1046
4.37.2.475 FUNNYDOLLAR . . . 1046
4.37.2.476 EMPTYINDEX . . . 1046
4.37.2.477 MINSPEC . . . 1046
4.37.2.478 USEDFLAG . . . 1046
4.37.2.479 DUMMYFLAG . . . 1046

4.37.2.480 MAINSORT . . . 1046
4.37.2.481 FUNCTIONSORT . . . 1046
4.37.2.482 SUBSORT . . . 1047
4.37.2.483 FLOATMODE . . . 1047
4.37.2.484 RATIONALMODE . . . 1047
4.37.2.485 NUMSPECSETS . . . 1047
4.37.2.486 EATTENSOR . . . 1047
4.37.2.487 ISZERO . . . 1047
4.37.2.488 ISUNMODIFIED . . . 1047
4.37.2.489 ISCOMPRESSED . . . 1047
4.37.2.490 ISINRHS . . . 1048
4.37.2.491 ISFACTORIZED . . . 1048
4.37.2.492 TOBEFACTORED . . . 1048
4.37.2.493 TOBEUNFACTORED . . . 1048
4.37.2.494 KEEPZERO . . . 1048
4.37.2.495 VARTYPENONE . . . 1048
4.37.2.496 VARTYPECOMPLEX . . . 1048
4.37.2.497 VARTYPEIMAGINARY . . . 1048
4.37.2.498 VARTYPEROOTOFUNITY . . . 1049
4.37.2.499 VARTYPEMINUS . . . 1049
4.37.2.500 CYCLESYMMETRIC . . . 1049
4.37.2.501 RCYCLESYMMETRIC . . . 1049
4.37.2.502 SYMMETRIC . . . 1049
4.37.2.503 ANTISYMMETRIC . . . 1049
4.37.2.504 REVERSEORDER . . . 1049
4.37.2.505 SUBMULTI . . . 1049
4.37.2.506 SUBONCE . . . 1050
4.37.2.507 SUBONLY . . . 1050
4.37.2.508 SUBMANY . . . 1050
4.37.2.509 SUBVECTOR . . . 1050
4.37.2.510 SUBSELECT . . . 1050
4.37.2.511 SUBALL . . . 1050
4.37.2.512 SUBMASK . . . 1050
4.37.2.513 SUBDISORDER . . . 1050
4.37.2.514 SUBAFTER . . . 1051
4.37.2.515 SUBAFTERNOT . . . 1051
4.37.2.516 IDHEAD . . . 1051
4.37.2.517 DOLLARFLAG . . . 1051
4.37.2.518 NORMALIZEFLAG . . . 1051
4.37.2.519 GIDENT . . . 1051
4.37.2.520 GFIVE . . . 1051
4.37.2.521 GPLUS . . . 1051

4.37.2.522 GMINUS . . . 1052
4.37.2.523 LONGNUMBER . . . 1052
4.37.2.524 MATCH . . . 1052
4.37.2.525 COEFFI . . . 1052
4.37.2.526 SUBEXPR . . . 1052
4.37.2.527 MULTIPLEOF . . . 1052
4.37.2.528 IFDOLLAR . . . 1052
4.37.2.529 IFEXPRESSION . . . 1052
4.37.2.530 IFDOLLAREXTRA . . . 1053
4.37.2.531 IFISFACTORIZED . . . 1053
4.37.2.532 IFOCCURS . . . 1053
4.37.2.533 GREATER . . . 1053
4.37.2.534 GREATEREQUAL . . . 1053
4.37.2.535 LESS . . . 1053
4.37.2.536 LESSEQUAL . . . 1053
4.37.2.537 EQUAL . . . 1053
4.37.2.538 NOTEQUAL . . . 1054
4.37.2.539 ORCOND . . . 1054
4.37.2.540 ANDCOND . . . 1054
4.37.2.541 DUMMY . . . 1054
4.37.2.542 SORT . . . 1054
4.37.2.543 STORE . . . 1054
4.37.2.544 END . . . 1054
4.37.2.545 GLOBAL . . . 1054
4.37.2.546 CLEAR . . . 1055
4.37.2.547 VECTBIT . . . 1055
4.37.2.548 DOTPBIT . . . 1055
4.37.2.549 FUNBIT . . . 1055
4.37.2.550 SETBIT . . . 1055
4.37.2.551 EXTRAPARAMETER . . . 1055
4.37.2.552 GENCOMMUTE . . . 1055
4.37.2.553 GENNONCOMMUTE . . . 1055
4.37.2.554 NAMENOTFOUND . . . 1056
4.37.2.555 DOLUNDEFINED . . . 1056
4.37.2.556 DOLNUMBER . . . 1056
4.37.2.557 DOLARGUMENT . . . 1056
4.37.2.558 DOLSUBTERM . . . 1056
4.37.2.559 DOLTERMS . . . 1056
4.37.2.560 DOLWILDARGS . . . 1056
4.37.2.561 DOLINDEX . . . 1056
4.37.2.562 DOLZERO . . . 1057
4.37.2.563 FINDLOOP . . . 1057

4.37.2.564 REPLACELOOP . . . 1057
4.37.2.565 NOFUNPOWERS . . . 1057
4.37.2.566 COMFUNPOWERS . . . 1057
4.37.2.567 ALLFUNPOWERS . . . 1057
4.37.2.568 PROPERORDERFLAG . . . 1057
4.37.2.569 REGULAR . . . 1057
4.37.2.570 FINISH . . . 1058
4.37.2.571 POLYADD . . . 1058
4.37.2.572 POLYSUB . . . 1058
4.37.2.573 POLYMUL . . . 1058
4.37.2.574 POLYDIV . . . 1058
4.37.2.575 POLYREM . . . 1058
4.37.2.576 POLYGCD . . . 1058
4.37.2.577 POLYINTFAC . . . 1058
4.37.2.578 POLYNORM . . . 1059
4.37.2.579 MODNONE . . . 1059
4.37.2.580 MODSUM . . . 1059
4.37.2.581 MODMAX . . . 1059
4.37.2.582 MODMIN . . . 1059
4.37.2.583 MODLOCAL . . . 1059
4.37.2.584 ELEMENTUSED . . . 1059
4.37.2.585 ELEMENTLOADED . . . 1059
4.37.2.586 POSNEG . . . 1060
4.37.2.587 INVERSETABLE . . . 1060
4.37.2.588 COEFFICIENTSONLY . . . 1060
4.37.2.589 ALSOPOWERS . . . 1060
4.37.2.590 ALSOFUNARGS . . . 1060
4.37.2.591 ALSODOLLARS . . . 1060
4.37.2.592 NOINVERSES . . . 1060
4.37.2.593 POSITIVEONLY . . . 1060
4.37.2.594 UNPACK . . . 1061
4.37.2.595 NOUNPACK . . . 1061
4.37.2.596 FROMFUNCTION . . . 1061
4.37.2.597 VARNAMES . . . 1061
4.37.2.598 AUTONAMES . . . 1061
4.37.2.599 EXPRNAMES . . . 1061
4.37.2.600 DOLLARNAMES . . . 1061
4.37.2.601 DUMMYBUFFER . . . 1061
4.37.2.602 ALLARGS . . . 1062
4.37.2.603 NUMARG . . . 1062
4.37.2.604 ARGRANGE . . . 1062
4.37.2.605 MAKEARGS . . . 1062

4.37.2.606 MAXRANGEINDICATOR . . . 1062
4.37.2.607 REPLACEARG . . . 1062
4.37.2.608 ENCODEARG . . . 1062
4.37.2.609 DECODEARG . . . 1062
4.37.2.610 IMPLODEARG . . . 1063
4.37.2.611 EXPLODEARG . . . 1063
4.37.2.612 PERMUTEARG . . . 1063
4.37.2.613 REVERSEARG . . . 1063
4.37.2.614 CYCLEARG . . . 1063
4.37.2.615 ISLYNDON . . . 1063
4.37.2.616 ISLYNDONR . . . 1063
4.37.2.617 TOLYNDON . . . 1063
4.37.2.618 TOLYNDONR . . . 1064
4.37.2.619 ADDARG . . . 1064
4.37.2.620 MULTIPLYARG . . . 1064
4.37.2.621 DROPARG . . . 1064
4.37.2.622 SELECTARG . . . 1064
4.37.2.623 DEDUPARG . . . 1064
4.37.2.624 BASECODE . . . 1064
4.37.2.625 YESLYNDON . . . 1064
4.37.2.626 NOLYNDON . . . 1065
4.37.2.627 TOPOLYNOMIALFLAG . . . 1065
4.37.2.628 FACTARGFLAG . . . 1065
4.37.2.629 OLDFACTARG . . . 1065
4.37.2.630 NEWFACTARG . . . 1065
4.37.2.631 FROMMODULEOPTION . . . 1065
4.37.2.632 FROMPOINTINSTRUCTION . . . 1065
4.37.2.633 EXTRASYMBOL . . . 1065
4.37.2.634 REGULARSYMBOL . . . 1066
4.37.2.635 EXPRESSIONNUMBER . . . 1066
4.37.2.636 O_NONE . . . 1066
4.37.2.637 O_CSE . . . 1066
4.37.2.638 O_CSEGREEDY . . . 1066
4.37.2.639 O_GREEDY . . . 1066
4.37.2.640 O_OCCURRENCE . . . 1066
4.37.2.641 O_MCTS . . . 1066
4.37.2.642 O_SIMULATED_ANNEALING . . . 1067
4.37.2.643 O_FORWARD . . . 1067
4.37.2.644 O_BACKWARD . . . 1067
4.37.2.645 O_FORWARDORBACKWARD . . . 1067
4.37.2.646 O_FORWARDANDBACKWARD . . . 1067
4.37.2.647 OPTHEAD . . . 1067

4.37.2.648 DOALL . . . 1067
4.37.2.649 ONLYFUNCTIONS . . . 1067
4.37.2.650 INUSE . . . 1068
4.37.2.651 COULDCOMMUTE . . . 1068
4.37.2.652 DOESNOTCOMMUTE . . . 1068
4.37.2.653 DICT_NONUMBERS . . . 1068
4.37.2.654 DICT_INTEGERONLY . . . 1068
4.37.2.655 DICT_RATIONALONLY . . . 1068
4.37.2.656 DICT_ALLNUMBERS . . . 1068
4.37.2.657 DICT_NOVARIABLES . . . 1068
4.37.2.658 DICT_DOVARIABLES . . . 1069
4.37.2.659 DICT_NOSPECIALS . . . 1069
4.37.2.660 DICT_DOSPECIALS . . . 1069
4.37.2.661 DICT_NOFUNWITHARGS . . . 1069
4.37.2.662 DICT_DOFUNWITHARGS . . . 1069
4.37.2.663 DICT_NOTINDOLLARS . . . 1069
4.37.2.664 DICT_INDOLLARS . . . 1069
4.37.2.665 DICT_INTEGERNUMBER . . . 1069
4.37.2.666 DICT_RATIONALNUMBER . . . 1070
4.37.2.667 DICT_SYMBOL . . . 1070
4.37.2.668 DICT_VECTOR . . . 1070
4.37.2.669 DICT_INDEX . . . 1070
4.37.2.670 DICT_FUNCTION . . . 1070
4.37.2.671 DICT_FUNCTION_WITH_ARGUMENTS . . . 1070
4.37.2.672 DICT_SPECIALCHARACTER . . . 1070
4.37.2.673 DICT_RANGE . . . 1070
4.37.2.674 READSPECTATORFLAG . . . 1071
4.37.2.675 GLOBALSPECTATORFLAG . . . 1071
4.37.3 Typedef Documentation . . . 1071
4.37.3.1 PVFUNWP . . . 1071
4.37.3.2 PVFUNV . . . 1071
4.37.3.3 CFUN . . . 1071
4.37.3.4 TFUN . . . 1071
4.37.3.5 TFUN1 . . . 1071
4.38 ftypes.h . . . 1072
4.39 function.c File Reference . . . 1083
4.39.1 Detailed Description . . . 1084
4.39.2 Function Documentation . . . 1084
4.39.2.1 MakeDirty() . . . 1084
4.39.2.2 MarkDirty() . . . 1084
4.39.2.3 PolyFunDirty() . . . 1084
4.39.2.4 PolyFunClean() . . . 1085

4.39.2.5 Symmetrize() . . . 1085
4.39.2.6 CompGroup() . . . 1085
4.39.2.7 FullSymmetrize() . . . 1085
4.39.2.8 SymGen() . . . 1085
4.39.2.9 SymFind() . . . 1086
4.39.2.10 ChainIn() . . . 1086
4.39.2.11 ChainOut() . . . 1086
4.39.2.12 MatchFunction() . . . 1086
4.39.2.13 ScanFunctions() . . . 1086
4.40 function.c . . . 1087
4.41 fwin.h File Reference . . . 1109
4.41.1 Detailed Description . . . 1109
4.41.2 Macro Definition Documentation . . . 1109
4.41.2.1 LINEFEED . . . 1109
4.41.2.2 CARRIAGERETURN . . . 1110
4.41.2.3 WITHRETURN . . . 1110
4.41.2.4 WITHSYSTEM . . . 1110
4.41.2.5 P_term . . . 1110
4.41.2.6 SEPARATOR . . . 1110
4.41.2.7 ALTSEPARATOR . . . 1110
4.41.2.8 PATHSEPARATOR . . . 1110
4.41.2.9 WITH_ENV . . . 1111
4.42 fwin.h . . . 1111
4.43 if.c File Reference . . . 1111
4.43.1 Detailed Description . . . 1112
4.43.2 Function Documentation . . . 1112
4.43.2.1 GetIfDollarNum() . . . 1112
4.43.2.2 FindVar() . . . 1112
4.43.2.3 DoIfStatement() . . . 1112
4.43.2.4 HowMany() . . . 1112
4.43.2.5 DoubleIfBuffers() . . . 1113
4.44 if.c . . . 1113
4.45 index.c File Reference . . . 1125
4.45.1 Detailed Description . . . 1125
4.45.2 Function Documentation . . . 1125
4.45.2.1 FindBracket() . . . 1125
4.45.2.2 PutBracketInIndex() . . . 1126
4.45.2.3 ClearBracketIndex() . . . 1126
4.45.2.4 OpenBracketIndex() . . . 1126
4.45.2.5 PutInside() . . . 1126
4.46 index.c . . . 1126
4.47 inivar.h File Reference . . . 1135

4.47.1 Detailed Description . . . 1135
4.47.2 Variable Documentation . . . 1135
4.47.2.1 FG . . . 1135
4.47.2.2 A . . . 1135
4.47.2.3 fixedsets . . . 1136
4.47.2.4 BufferForOutput . . . 1136
4.47.2.5 setupfilename . . . 1136
4.48 inivar.h . . . 1136
4.49 lus.c File Reference . . . 1139
4.49.1 Detailed Description . . . 1140
4.49.2 Function Documentation . . . 1140
4.49.2.1 Lus() . . . 1140
4.49.2.2 FindLus() . . . 1140
4.49.2.3 SortTheList() . . . 1140
4.50 lus.c . . . 1141
4.51 mallocprotect.h . . . 1148
4.52 message.c File Reference . . . 1153
4.52.1 Detailed Description . . . 1153
4.52.2 Function Documentation . . . 1153
4.52.2.1 Error0() . . . 1153
4.52.2.2 Error1() . . . 1154
4.52.2.3 Error2() . . . 1154
4.52.2.4 MesWork() . . . 1154
4.52.2.5 MesPrint() . . . 1154
4.52.2.6 Warning() . . . 1154
4.52.2.7 HighWarning() . . . 1154
4.52.2.8 MesCall() . . . 1154
4.52.2.9 MesCerr() . . . 1155
4.52.2.10 MesComp() . . . 1155
4.52.2.11 PrintTerm() . . . 1155
4.52.2.12 PrintTermC() . . . 1155
4.52.2.13 PrintSubTerm() . . . 1155
4.52.2.14 PrintWords() . . . 1155
4.52.2.15 PrintSeq() . . . 1156
4.53 message.c . . . 1156
4.54 minos.c File Reference . . . 1167
4.54.1 Detailed Description . . . 1168
4.54.2 Macro Definition Documentation . . . 1168
4.54.2.1 CFD . . . 1168
4.54.2.2 CTD . . . 1169
4.54.3 Function Documentation . . . 1169
4.54.3.1 minosread() . . . 1169

4.54.3.2 minoswrite() . . . 1169
4.54.3.3 str_dup() . . . 1169
4.54.3.4 convertblock() . . . 1169
4.54.3.5 convertnamesblock() . . . 1170
4.54.3.6 convertiniinfo() . . . 1170
4.54.3.7 LocateBase() . . . 1170
4.54.3.8 ReadIndex() . . . 1170
4.54.3.9 WriteIndexBlock() . . . 1170
4.54.3.10 WriteNamesBlock() . . . 1170
4.54.3.11 WriteIndex() . . . 1171
4.54.3.12 WriteIniInfo() . . . 1171
4.54.3.13 ReadIniInfo() . . . 1171
4.54.3.14 GetDbase() . . . 1171
4.54.3.15 NewDbase() . . . 1171
4.54.3.16 FreeTableBase() . . . 1171
4.54.3.17 ComposeTableNames() . . . 1171
4.54.3.18 OpenDbase() . . . 1172
4.54.3.19 AddTableName() . . . 1172
4.54.3.20 GetTableName() . . . 1172
4.54.3.21 PutTableNames() . . . 1172
4.54.3.22 AddToIndex() . . . 1172
4.54.3.23 AddObject() . . . 1172
4.54.3.24 FindTableNumber() . . . 1173
4.54.3.25 WriteObject() . . . 1173
4.54.3.26 ReadObject() . . . 1173
4.54.3.27 ReadijObject() . . . 1173
4.54.3.28 ExistsObject() . . . 1173
4.54.3.29 DeleteObject() . . . 1174
4.54.4 Variable Documentation . . . 1174
4.54.4.1 withoutflush . . . 1174
4.55 minos.c . . . 1174
4.56 minos.h File Reference . . . 1191
4.56.1 Detailed Description . . . 1192
4.56.2 Macro Definition Documentation . . . 1192
4.56.2.1 MAXBASES . . . 1192
4.56.2.2 NUMOBJECTS . . . 1193
4.56.2.3 MAXINDEXSIZE . . . 1193
4.56.2.4 NAMETABLESIZE . . . 1193
4.56.2.5 ELEMENTSIZE . . . 1193
4.56.2.6 TODISK . . . 1193
4.56.2.7 FROMDISK . . . 1193
4.56.2.8 MCLEANFLAG . . . 1193

4.56.2.9 DIRTYFLAG . . . 1193
4.56.2.10 INANDOUT . . . 1194
4.56.2.11 INPUTONLY . . . 1194
4.56.2.12 OUTPUTONLY . . . 1194
4.56.2.13 NOCOMPRESS . . . 1194
4.56.3 Function Documentation . . . 1194
4.56.3.1 minosread() . . . 1194
4.56.3.2 minoswrite() . . . 1194
4.56.4 Variable Documentation . . . 1195
4.56.4.1 withoutflush . . . 1195
4.57 minos.h . . . 1195
4.58 module.c File Reference . . . 1197
4.58.1 Detailed Description . . . 1197
4.58.2 Function Documentation . . . 1198
4.58.2.1 ModuleInstruction() . . . 1198
4.58.2.2 CoModuleOption() . . . 1198
4.58.2.3 CoModOption() . . . 1198
4.58.2.4 SetSpecialMode() . . . 1198
4.58.2.5 ExecModule() . . . 1198
4.58.2.6 ExecStore() . . . 1198
4.58.2.7 FullCleanUp() . . . 1199
4.58.2.8 DoPolyfun() . . . 1199
4.58.2.9 DoPolyratfun() . . . 1199
4.58.2.10 DoNoParallel() . . . 1199
4.58.2.11 DoParallel() . . . 1199
4.58.2.12 DoModSum() . . . 1199
4.58.2.13 DoModMax() . . . 1199
4.58.2.14 DoModMin() . . . 1200
4.58.2.15 DoModLocal() . . . 1200
4.58.2.16 DoProcessBucket() . . . 1200
4.58.2.17 DoModDollar() . . . 1200
4.58.2.18 DoinParallel() . . . 1200
4.58.2.19 DonotinParallel() . . . 1200
4.58.2.20 DoExecStatement() . . . 1200
4.58.2.21 DoPipeStatement() . . . 1201
4.59 module.c . . . 1201
4.60 mpi.c File Reference . . . 1210
4.60.1 Detailed Description . . . 1211
4.60.2 Macro Definition Documentation . . . 1211
4.60.2.1 PF_PACKSIZE . . . 1211
4.60.2.2 MPI_ERRCODE_CHECK . . . 1212
4.60.3 Function Documentation . . . 1212

4.60.3.1 PF_RealTime() . . . 1212
4.60.3.2 PF_LibInit() . . . 1212
4.60.3.3 PF_LibTerminate() . . . 1213
4.60.3.4 PF_Probe() . . . 1213
4.60.3.5 PF_ISendSbuf() . . . 1214
4.60.3.6 PF_RecvWbuf() . . . 1214
4.60.3.7 PF_IRecvRbuf() . . . 1214
4.60.3.8 PF_WaitRbuf() . . . 1215
4.60.3.9 PF_Bcast() . . . 1215
4.60.3.10 PF_RawSend() . . . 1216
4.60.3.11 PF_RawRecv() . . . 1216
4.60.3.12 PF_RawProbe() . . . 1217
4.60.3.13 PF_PrintPackBuf() . . . 1217
4.60.3.14 PF_PreparePack() . . . 1218
4.60.3.15 PF_Pack() . . . 1218
4.60.3.16 PF_Unpack() . . . 1218
4.60.3.17 PF_PackString() . . . 1219
4.60.3.18 PF_UnpackString() . . . 1219
4.60.3.19 PF_Send() . . . 1220
4.60.3.20 PF_Receive() . . . 1220
4.60.3.21 PF_Broadcast() . . . 1221
4.60.3.22 PF_PrepareLongSinglePack() . . . 1221
4.60.3.23 PF_LongSinglePack() . . . 1221
4.60.3.24 PF_LongSingleUnpack() . . . 1222
4.60.3.25 PF_LongSingleSend() . . . 1222
4.60.3.26 PF_LongSingleReceive() . . . 1223
4.60.3.27 PF_PrepareLongMultiPack() . . . 1223
4.60.3.28 PF_LongMultiPackImpl() . . . 1224
4.60.3.29 PF_LongMultiUnpackImpl() . . . 1224
4.60.3.30 PF_LongMultiBroadcast() . . . 1225
4.60.4 Variable Documentation . . . 1226
4.60.4.1 PF_maxDollarChunkSize . . . 1226
4.61 mpi.c . . . 1226
4.62 mpidbg.h File Reference . . . 1244
4.62.1 Detailed Description . . . 1245
4.62.2 Macro Definition Documentation . . . 1245
4.62.2.1 MPIDBG_RANK . . . 1245
4.62.2.2 MPIDBG_Out . . . 1245
4.62.2.3 MPIDBG_EXTARG . . . 1245
4.62.2.4 MPI_Send . . . 1245
4.62.2.5 MPI_Recv . . . 1245
4.62.2.6 MPI_Bsend . . . 1245

4.62.2.7 MPI_Ssend . . . 1246
4.62.2.8 MPI_Rsend . . . 1246
4.62.2.9 MPI_Isend . . . 1246
4.62.2.10 MPI_Ibsend . . . 1246
4.62.2.11 MPI_Issend . . . 1246
4.62.2.12 MPI_Irsend . . . 1246
4.62.2.13 MPI_Irecv . . . 1246
4.62.2.14 MPI_Wait . . . 1247
4.62.2.15 MPI_Test . . . 1247
4.62.2.16 MPI_Waitany . . . 1247
4.62.2.17 MPI_Testany . . . 1247
4.62.2.18 MPI_Waitall . . . 1247
4.62.2.19 MPI_Testall . . . 1247
4.62.2.20 MPI_Waitsome . . . 1247
4.62.2.21 MPI_Testsome . . . 1248
4.62.2.22 MPI_Iprobe . . . 1248
4.62.2.23 MPI_Probe . . . 1248
4.62.2.24 MPI_Cancel . . . 1248
4.62.2.25 MPI_Test_cancelled . . . 1248
4.62.2.26 MPI_Barrier . . . 1248
4.62.2.27 MPI_Bcast . . . 1248
4.63 mpidbg.h . . . 1249
4.64 mytime.h . . . 1257
4.65 names.c File Reference . . . 1257
4.65.1 Detailed Description . . . 1258
4.65.2 Function Documentation . . . 1259
4.65.2.1 GetNode() . . . 1259
4.65.2.2 AddName() . . . 1259
4.65.2.3 GetName() . . . 1259
4.65.2.4 GetLastExprName() . . . 1259
4.65.2.5 GetOName() . . . 1259
4.65.2.6 GetAutoName() . . . 1260
4.65.2.7 GetVar() . . . 1260
4.65.2.8 EntVar() . . . 1260
4.65.2.9 GetDollar() . . . 1260
4.65.2.10 DumpTree() . . . 1260
4.65.2.11 DumpNode() . . . 1260
4.65.2.12 CompactifyTree() . . . 1261
4.65.2.13 CopyTree() . . . 1261
4.65.2.14 LinkTree() . . . 1261
4.65.2.15 MakeNameTree() . . . 1261
4.65.2.16 FreeNameTree() . . . 1261

4.65.2.17 ClearWildcardNames() . . . 1261
4.65.2.18 AddWildcardName() . . . 1262
4.65.2.19 GetWildcardName() . . . 1262
4.65.2.20 AddSymbol() . . . 1262
4.65.2.21 CoSymbol() . . . 1262
4.65.2.22 AddIndex() . . . 1262
4.65.2.23 CoIndex() . . . 1262
4.65.2.24 DoDimension() . . . 1263
4.65.2.25 CoDimension() . . . 1263
4.65.2.26 AddVector() . . . 1263
4.65.2.27 CoVector() . . . 1263
4.65.2.28 AddFunction() . . . 1263
4.65.2.29 CoCommuteInSet() . . . 1263
4.65.2.30 CoFunction() . . . 1264
4.65.2.31 CoNFunction() . . . 1264
4.65.2.32 CoCFunction() . . . 1264
4.65.2.33 CoNTensor() . . . 1264
4.65.2.34 CoCTensor() . . . 1264
4.65.2.35 DoTable() . . . 1264
4.65.2.36 CoTable() . . . 1265
4.65.2.37 CoNTable() . . . 1265
4.65.2.38 CoCTable() . . . 1265
4.65.2.39 EmptyTable() . . . 1265
4.65.2.40 AddSet() . . . 1265
4.65.2.41 DoElements() . . . 1265
4.65.2.42 CoSet() . . . 1266
4.65.2.43 DoTempSet() . . . 1266
4.65.2.44 CoAuto() . . . 1266
4.65.2.45 AddDollar() . . . 1266
4.65.2.46 ReplaceDollar() . . . 1266
4.65.2.47 AddDubious() . . . 1266
4.65.2.48 MakeDubious() . . . 1267
4.65.2.49 NameConflict() . . . 1267
4.65.2.50 AddExpression() . . . 1267
4.65.2.51 GetLabel() . . . 1267
4.65.2.52 ResetVariables() . . . 1267
4.65.2.53 RemoveDollars() . . . 1267
4.65.2.54 Globalize() . . . 1268
4.65.2.55 TestName() . . . 1268
4.66 names.c . . . 1268
4.67 normal.c File Reference . . . 1304
4.67.1 Detailed Description . . . 1304

4.67.2 Macro Definition Documentation . . . 1305
4.67.2.1 MAXNUMBEROFNONCOMTERMS . . . 1305
4.67.2.2 EXCHN . . . 1305
4.67.2.3 EXCH . . . 1305
4.67.3 Function Documentation . . . 1305
4.67.3.1 CompareFunctions() . . . 1305
4.67.3.2 Commute() . . . 1305
4.67.3.3 Normalize() . . . 1305
4.67.3.4 ExtraSymbol() . . . 1306
4.67.3.5 DoTheta() . . . 1306
4.67.3.6 DoDelta() . . . 1306
4.67.3.7 DoRevert() . . . 1306
4.67.3.8 DetCommu() . . . 1306
4.67.3.9 DoesCommu() . . . 1306
4.67.3.10 TreatPolyRatFun() . . . 1307
4.67.3.11 DropCoefficient() . . . 1307
4.67.3.12 DropSymbols() . . . 1307
4.67.3.13 SymbolNormalize() . . . 1307
4.67.3.14 TestFunFlag() . . . 1307
4.67.3.15 BracketNormalize() . . . 1307
4.68 normal.c . . . 1308
4.69 notation.c File Reference . . . 1368
4.69.1 Detailed Description . . . 1369
4.69.2 Function Documentation . . . 1369
4.69.2.1 NormPolyTerm() . . . 1369
4.69.2.2 ConvertToPoly() . . . 1369
4.69.2.3 LocalConvertToPoly() . . . 1369
4.69.2.4 ConvertFromPoly() . . . 1370
4.69.2.5 FindSubterm() . . . 1370
4.69.2.6 FindLocalSubterm() . . . 1370
4.69.2.7 PrintSubtermList() . . . 1370
4.69.2.8 PrintExtraSymbol() . . . 1370
4.69.2.9 FindSubexpression() . . . 1370
4.69.2.10 ExtraSymFun() . . . 1371
4.69.2.11 PruneExtraSymbols() . . . 1371
4.70 notation.c . . . 1371
4.71 opera.c File Reference . . . 1385
4.71.1 Detailed Description . . . 1385
4.71.2 Function Documentation . . . 1385
4.71.2.1 EpfFind() . . . 1385
4.71.2.2 EpfCon() . . . 1386
4.71.2.3 EpfGen() . . . 1386

4.71.2.4 Trick() . . . 1386
4.71.2.5 Trace4no() . . . 1386
4.71.2.6 Trace4() . . . 1386
4.71.2.7 Trace4Gen() . . . 1387
4.71.2.8 TraceNno() . . . 1387
4.71.2.9 TraceN() . . . 1387
4.71.2.10 TraceNgen() . . . 1387
4.71.2.11 Traces() . . . 1387
4.71.2.12 TraceFind() . . . 1387
4.71.2.13 Chisholm() . . . 1388
4.71.2.14 TenVecFind() . . . 1388
4.71.2.15 TenVec() . . . 1388
4.72 opera.c . . . 1388
4.73 parallel.c File Reference . . . 1417
4.73.1 Detailed Description . . . 1419
4.73.2 Macro Definition Documentation . . . 1419
4.73.2.1 PRINTFBUF . . . 1419
4.73.2.2 SWAP . . . 1419
4.73.2.3 PACK_LONG . . . 1419
4.73.2.4 UNPACK_LONG . . . 1420
4.73.2.5 CHECK . . . 1420
4.73.2.6 _CHECK . . . 1420
4.73.2.7 __CHECK . . . 1420
4.73.2.8 DBGOUT . . . 1421
4.73.2.9 DBGOUT_NINTERMS . . . 1421
4.73.2.10 PF_STATS_SIZE . . . 1421
4.73.2.11 PF_SNDFILEBUFSIZE . . . 1421
4.73.2.12 recvBuffer . . . 1421
4.73.3 Typedef Documentation . . . 1421
4.73.3.1 NODE . . . 1421
4.73.4 Function Documentation . . . 1421
4.73.4.1 PF_RealTime() . . . 1421
4.73.4.2 PF_LibInit() . . . 1422
4.73.4.3 PF_LibTerminate() . . . 1422
4.73.4.4 PF_Probe() . . . 1423
4.73.4.5 PF_RecvWbuf() . . . 1423
4.73.4.6 PF_IRecvRbuf() . . . 1423
4.73.4.7 PF_WaitRbuf() . . . 1424
4.73.4.8 PF_RawSend() . . . 1424
4.73.4.9 PF_RawRecv() . . . 1425
4.73.4.10 PF_RawProbe() . . . 1425
4.73.4.11 PF_EndSort() . . . 1426

4.73.4.12 PF_Deferred() . . . 1426
4.73.4.13 PF_Processor() . . . 1427
4.73.4.14 PF_Init() . . . 1429
4.73.4.15 PF_Terminate() . . . 1429
4.73.4.16 PF_GetSlaveTimes() . . . 1430
4.73.4.17 PF_BroadcastNumber() . . . 1430
4.73.4.18 PF_BroadcastBuffer() . . . 1431
4.73.4.19 PF_BroadcastString() . . . 1432
4.73.4.20 PF_BroadcastPreDollar() . . . 1433
4.73.4.21 PF_CollectModifiedDollars() . . . 1434
4.73.4.22 PF_BroadcastModifiedDollars() . . . 1435
4.73.4.23 PF_BroadcastRedefinedPreVars() . . . 1435
4.73.4.24 PF_StoreInsideInfo() . . . 1436
4.73.4.25 PF_RestoreInsideInfo() . . . 1436
4.73.4.26 PF_BroadcastCBuf() . . . 1436
4.73.4.27 PF_BroadcastExpFlags() . . . 1437
4.73.4.28 PF_BroadcastExpr() . . . 1438
4.73.4.29 PF_BroadcastRHS() . . . 1438
4.73.4.30 PF_InParallelProcessor() . . . 1439
4.73.4.31 PF_SendFile() . . . 1439
4.73.4.32 PF_RecvFile() . . . 1440
4.73.4.33 PF_MLock() . . . 1440
4.73.4.34 PF_MUnlock() . . . 1441
4.73.4.35 PF_WriteFileToFile() . . . 1441
4.73.4.36 PF_FlushStdOutBuffer() . . . 1442
4.73.4.37 PF_FreeErrorMessageBuffers() . . . 1442
4.73.5 Variable Documentation . . . 1442
4.73.5.1 PF . . . 1442
4.74 parallel.c . . . 1442
4.75 parallel.h File Reference . . . 1491
4.75.1 Detailed Description . . . 1493
4.75.2 Macro Definition Documentation . . . 1493
4.75.2.1 MASTER . . . 1493
4.75.2.2 PF_RESET . . . 1493
4.75.2.3 PF_TIME . . . 1493
4.75.2.4 PF_TERM_MSGTAG . . . 1493
4.75.2.5 PF_ENDSORT_MSGTAG . . . 1493
4.75.2.6 PF_DOLLAR_MSGTAG . . . 1494
4.75.2.7 PF_BUFFER_MSGTAG . . . 1494
4.75.2.8 PF_ENDBUFFER_MSGTAG . . . 1494
4.75.2.9 PF_READY_MSGTAG . . . 1494
4.75.2.10 PF_DATA_MSGTAG . . . 1494

4.75.2.11 PF_EMPTY_MSGTAG . . . 1494
4.75.2.12 PF_STDOUT_MSGTAG . . . 1494
4.75.2.13 PF_LOG_MSGTAG . . . 1495
4.75.2.14 PF_OPT_MCTS_MSGTAG . . . 1495
4.75.2.15 PF_OPT_HORNER_MSGTAG . . . 1495
4.75.2.16 PF_OPT_COLLECT_MSGTAG . . . 1495
4.75.2.17 PF_MISC_MSGTAG . . . 1495
4.75.2.18 GNUC_PREREQ . . . 1495
4.75.2.19 indices . . . 1495
4.75.2.20 PF_ANY_SOURCE . . . 1496
4.75.2.21 PF_ANY_MSGTAG . . . 1496
4.75.2.22 PF_COMM . . . 1496
4.75.2.23 PF_BYTE . . . 1496
4.75.2.24 PF_INT . . . 1496
4.75.2.25 PF_LongMultiPack . . . 1496
4.75.2.26 PF_LongMultiUnpack . . . 1496
4.75.3 Function Documentation . . . 1496
4.75.3.1 PF_ISendSbuf() . . . 1496
4.75.3.2 PF_Bcast() . . . 1497
4.75.3.3 PF_RawSend() . . . 1497
4.75.3.4 PF_RawRecv() . . . 1498
4.75.3.5 PF_PreparePack() . . . 1498
4.75.3.6 PF_Pack() . . . 1498
4.75.3.7 PF_Unpack() . . . 1499
4.75.3.8 PF_PackString() . . . 1499
4.75.3.9 PF_UnpackString() . . . 1500
4.75.3.10 PF_Send() . . . 1500
4.75.3.11 PF_Receive() . . . 1501
4.75.3.12 PF_Broadcast() . . . 1501
4.75.3.13 PF_PrepareLongSinglePack() . . . 1502
4.75.3.14 PF_LongSinglePack() . . . 1502
4.75.3.15 PF_LongSingleUnpack() . . . 1503
4.75.3.16 PF_LongSingleSend() . . . 1503
4.75.3.17 PF_LongSingleReceive() . . . 1504
4.75.3.18 PF_PrepareLongMultiPack() . . . 1504
4.75.3.19 PF_LongMultiPackImpl() . . . 1504
4.75.3.20 PF_LongMultiUnpackImpl() . . . 1505
4.75.3.21 PF_LongMultiBroadcast() . . . 1506
4.75.3.22 PF_EndSort() . . . 1506
4.75.3.23 PF_Deferred() . . . 1507
4.75.3.24 PF_Processor() . . . 1508
4.75.3.25 PF_Init() . . . 1509

4.75.3.26 PF_Terminate() . . . 1510
4.75.3.27 PF_GetSlaveTimes() . . . 1511
4.75.3.28 PF_BroadcastNumber() . . . 1511
4.75.3.29 PF_BroadcastBuffer() . . . 1512
4.75.3.30 PF_BroadcastString() . . . 1513
4.75.3.31 PF_BroadcastPreDollar() . . . 1513
4.75.3.32 PF_CollectModifiedDollars() . . . 1514
4.75.3.33 PF_BroadcastModifiedDollars() . . . 1515
4.75.3.34 PF_BroadcastRedefinedPreVars() . . . 1516
4.75.3.35 PF_BroadcastCBuf() . . . 1517
4.75.3.36 PF_BroadcastExpFlags() . . . 1518
4.75.3.37 PF_StoreInsideInfo() . . . 1519
4.75.3.38 PF_RestoreInsideInfo() . . . 1519
4.75.3.39 PF_BroadcastExpr() . . . 1519
4.75.3.40 PF_BroadcastRHS() . . . 1519
4.75.3.41 PF_InParallelProcessor() . . . 1520
4.75.3.42 PF_SendFile() . . . 1520
4.75.3.43 PF_RecvFile() . . . 1521
4.75.3.44 PF_MLock() . . . 1522
4.75.3.45 PF_MUnlock() . . . 1522
4.75.3.46 PF_WriteFileToFile() . . . 1522
4.75.3.47 PF_FlushStdOutBuffer() . . . 1523
4.75.4 Variable Documentation . . . 1523
4.75.4.1 PF . . . 1523
4.75.4.2 PF_maxDollarChunkSize . . . 1523
4.76 parallel.h . . . 1523
4.77 pattern.c File Reference . . . 1527
4.77.1 Detailed Description . . . 1527
4.77.2 Macro Definition Documentation . . . 1528
4.77.2.1 PutInBuffers . . . 1528
4.77.3 Function Documentation . . . 1528
4.77.3.1 TestMatch() . . . 1528
4.77.3.2 Substitute() . . . 1529
4.77.3.3 FindAll() . . . 1529
4.77.3.4 TestSelect() . . . 1529
4.77.3.5 SubsInAll() . . . 1530
4.77.3.6 TransferBuffer() . . . 1530
4.77.3.7 TakeIDfunction() . . . 1530
4.78 pattern.c . . . 1530
4.79 poly.h . . . 1557
4.80 polyfact.h . . . 1560
4.81 polygcd.h . . . 1561

4.82 portsignals.h File Reference . . . 1562
4.82.1 Detailed Description . . . 1563
4.82.2 Macro Definition Documentation . . . 1563
4.82.2.1 FATAL_SIG_ERROR . . . 1563
4.82.2.2 NSIG . . . 1563
4.82.2.3 SIGSEGV . . . 1563
4.82.2.4 SIGFPE . . . 1564
4.82.2.5 SIGILL . . . 1564
4.82.2.6 SIGEMT . . . 1564
4.82.2.7 SIGSYS . . . 1564
4.82.2.8 SIGPIPE . . . 1564
4.82.2.9 SIGLOST . . . 1564
4.82.2.10 SIGXCPU . . . 1564
4.82.2.11 SIGXFSZ . . . 1564
4.82.2.12 SIGTERM . . . 1565
4.82.2.13 SIGINT . . . 1565
4.82.2.14 SIGQUIT . . . 1565
4.82.2.15 SIGHUP . . . 1565
4.82.2.16 SIGALRM . . . 1565
4.82.2.17 SIGVTALRM . . . 1565
4.82.2.18 SIGPROF . . . 1565
4.83 portsignals.h . . . 1566
4.84 pre.c File Reference . . . 1567
4.84.1 Detailed Description . . . 1569
4.84.2 Macro Definition Documentation . . . 1569
4.84.2.1 SKIPBUFSIZE . . . 1569
4.84.2.2 KILL . . . 1569
4.84.2.3 KILLALL . . . 1570
4.84.2.4 DAEMON . . . 1570
4.84.2.5 SHELL . . . 1570
4.84.2.6 STDERR . . . 1570
4.84.2.7 TRUE_EXPR . . . 1570
4.84.2.8 FALSE_EXPR . . . 1570
4.84.2.9 NOSHELL . . . 1570
4.84.2.10 TERMINAL . . . 1570
4.84.3 Function Documentation . . . 1571
4.84.3.1 GetInput() . . . 1571
4.84.3.2 ClearPushback() . . . 1571
4.84.3.3 GetChar() . . . 1571
4.84.3.4 CharOut() . . . 1571
4.84.3.5 UnsetAllowDelay() . . . 1571
4.84.3.6 GetPreVar() . . . 1571

4.84.3.7 PutPreVar() . . . 1571
4.84.3.8 PopPreVars() . . . 1572
4.84.3.9 IniModule() . . . 1572
4.84.3.10 IniSpecialModule() . . . 1572
4.84.3.11 PreProcessor() . . . 1572
4.84.3.12 PreProInstruction() . . . 1573
4.84.3.13 LoadInstruction() . . . 1573
4.84.3.14 LoadStatement() . . . 1573
4.84.3.15 ExpandTripleDots() . . . 1573
4.84.3.16 FindKeyWord() . . . 1573
4.84.3.17 FindInKeyWord() . . . 1573
4.84.3.18 TheDefine() . . . 1573
4.84.3.19 DoCommentChar() . . . 1574
4.84.3.20 DoPreAssign() . . . 1574
4.84.3.21 DoDefine() . . . 1574
4.84.3.22 DoRedefine() . . . 1575
4.84.3.23 ClearMacro() . . . 1575
4.84.3.24 TheUndefine() . . . 1575
4.84.3.25 DoUndefine() . . . 1575
4.84.3.26 DoInclude() . . . 1575
4.84.3.27 DoReverseInclude() . . . 1575
4.84.3.28 Include() . . . 1575
4.84.3.29 DoPreExchange() . . . 1576
4.84.3.30 DoCall() . . . 1576
4.84.3.31 DoDebug() . . . 1576
4.84.3.32 DoTerminate() . . . 1576
4.84.3.33 DoDo() . . . 1576
4.84.3.34 DoBreakDo() . . . 1576
4.84.3.35 DoElse() . . . 1576
4.84.3.36 DoElseif() . . . 1577
4.84.3.37 DoEnddo() . . . 1577
4.84.3.38 DoEndif() . . . 1577
4.84.3.39 DoEndprocedure() . . . 1577
4.84.3.40 DoIf() . . . 1577
4.84.3.41 DoIfdef() . . . 1577
4.84.3.42 DoInside() . . . 1577
4.84.3.43 DoEndInside() . . . 1578
4.84.3.44 DoMessage() . . . 1578
4.84.3.45 DoPipe() . . . 1578
4.84.3.46 DoPrcExtension() . . . 1578
4.84.3.47 DoPreOut() . . . 1578
4.84.3.48 DoPrePrintTimes() . . . 1578

4.84.3.49 DoPreAppend() . . . 1578
4.84.3.50 DoPreCreate() . . . 1579
4.84.3.51 DoPreRemove() . . . 1579
4.84.3.52 DoPreClose() . . . 1579
4.84.3.53 DoPreWrite() . . . 1579
4.84.3.54 DoProcedure() . . . 1579
4.84.3.55 DoPreBreak() . . . 1579
4.84.3.56 DoPreCase() . . . 1579
4.84.3.57 DoPreDefault() . . . 1580
4.84.3.58 DoPreEndSwitch() . . . 1580
4.84.3.59 DoPreSwitch() . . . 1580
4.84.3.60 DoPreShow() . . . 1580
4.84.3.61 DoSystem() . . . 1580
4.84.3.62 PreLoad() . . . 1580
4.84.3.63 PreSkip() . . . 1581
4.84.3.64 StartPrepro() . . . 1581
4.84.3.65 EvalPreIf() . . . 1581
4.84.3.66 PreIfEval() . . . 1581
4.84.3.67 PreCmp() . . . 1581
4.84.3.68 PreEq() . . . 1581
4.84.3.69 pParseObject() . . . 1582
4.84.3.70 PreCalc() . . . 1582
4.84.3.71 PreEval() . . . 1582
4.84.3.72 AddToPreTypes() . . . 1582
4.84.3.73 MessPreNesting() . . . 1582
4.84.3.74 DoPreAddSeparator() . . . 1582
4.84.3.75 DoPreRmSeparator() . . . 1582
4.84.3.76 DoExternal() . . . 1583
4.84.3.77 DoPrompt() . . . 1583
4.84.3.78 DoSetExternal() . . . 1583
4.84.3.79 DoSetExternalAttr() . . . 1583
4.84.3.80 DoRmExternal() . . . 1583
4.84.3.81 DoFromExternal() . . . 1583
4.84.3.82 DoToExternal() . . . 1583
4.84.3.83 defineChannel() . . . 1584
4.84.3.84 writeToChannel() . . . 1584
4.84.3.85 DoFactDollar() . . . 1584
4.84.3.86 GetDollarNumber() . . . 1584
4.84.3.87 DoSetRandom() . . . 1584
4.84.3.88 DoOptimize() . . . 1584
4.84.3.89 DoClearOptimize() . . . 1585
4.84.3.90 DoSkipExtraSymbols() . . . 1585

4.84.3.91 DoPreReset() . . . 1585
4.84.3.92 DoPreAppendPath() . . . 1585
4.84.3.93 DoPrePrependPath() . . . 1585
4.85 pre.c . . . 1586
4.86 proces.c File Reference . . . 1665
4.86.1 Detailed Description . . . 1665
4.86.2 Function Documentation . . . 1666
4.86.2.1 Processor() . . . 1666
4.86.2.2 TestSub() . . . 1667
4.86.2.3 InFunction() . . . 1668
4.86.2.4 InsertTerm() . . . 1669
4.86.2.5 PasteFile() . . . 1670
4.86.2.6 PasteTerm() . . . 1671
4.86.2.7 FiniTerm() . . . 1671
4.86.2.8 Generator() . . . 1672
4.86.2.9 DoOnePow() . . . 1673
4.86.2.10 Deferred() . . . 1674
4.86.2.11 PrepPoly() . . . 1675
4.86.2.12 PolyFunMul() . . . 1675
4.86.3 Variable Documentation . . . 1676
4.86.3.1 printscratch . . . 1676
4.87 proces.c . . . 1677
4.88 ratio.c File Reference . . . 1738
4.88.1 Detailed Description . . . 1739
4.88.2 Function Documentation . . . 1739
4.88.2.1 RatioFind() . . . 1739
4.88.2.2 RatioGen() . . . 1739
4.88.2.3 BinomGen() . . . 1739
4.88.2.4 DoSumF1() . . . 1740
4.88.2.5 Glue() . . . 1740
4.88.2.6 DoSumF2() . . . 1740
4.88.2.7 GCDfunction() . . . 1740
4.88.2.8 GCDfunction3() . . . 1740
4.88.2.9 PutExtraSymbols() . . . 1741
4.88.2.10 TakeExtraSymbols() . . . 1741
4.88.2.11 MultiplyWithTerm() . . . 1741
4.88.2.12 TakeContent() . . . 1741
4.88.2.13 MergeSymbolLists() . . . 1742
4.88.2.14 MergeDotproductLists() . . . 1742
4.88.2.15 CreateExpression() . . . 1742
4.88.2.16 GCDterms() . . . 1742
4.88.2.17 ReadPolyRatFun() . . . 1742

4.88.2.18 FromPolyRatFun() . . . 1742
4.88.2.19 TakeSymbolContent() . . . 1743
4.88.2.20 GCDclean() . . . 1743
4.88.2.21 PolyDiv() . . . 1743
4.88.2.22 DIVfunction() . . . 1743
4.88.2.23 MULfunc() . . . 1744
4.88.2.24 ConvertArgument() . . . 1744
4.88.2.25 ExpandRat() . . . 1744
4.88.2.26 InvPoly() . . . 1744
4.88.3 Variable Documentation . . . 1744
4.88.3.1 divrem . . . 1744
4.88.3.2 TheErrorMessage . . . 1744
4.89 ratio.c . . . 1745
4.90 reken.c File Reference . . . 1787
4.90.1 Detailed Description . . . 1788
4.90.2 Macro Definition Documentation . . . 1788
4.90.2.1 GCDMAX . . . 1788
4.90.2.2 NEWTRICK . . . 1789
4.90.2.3 COPYLONG . . . 1789
4.90.2.4 WARMUP . . . 1789
4.90.3 Function Documentation . . . 1789
4.90.3.1 Pack() . . . 1789
4.90.3.2 UnPack() . . . 1789
4.90.3.3 Mully() . . . 1790
4.90.3.4 Divvy() . . . 1790
4.90.3.5 AddRat() . . . 1790
4.90.3.6 MulRat() . . . 1790
4.90.3.7 DivRat() . . . 1790
4.90.3.8 Simplify() . . . 1791
4.90.3.9 AccumGCD() . . . 1791
4.90.3.10 TakeRatRoot() . . . 1791
4.90.3.11 AddLong() . . . 1791
4.90.3.12 AddPLon() . . . 1791
4.90.3.13 SubPLon() . . . 1792
4.90.3.14 MulLong() . . . 1792
4.90.3.15 BigLong() . . . 1792
4.90.3.16 DivLong() . . . 1792
4.90.3.17 RaisPow() . . . 1793
4.90.3.18 RaisPowCached() . . . 1793
4.90.4 Description . . . 1793
4.90.5 Notes . . . 1793
4.90.5.1 RaisPowMod() . . . 1793

4.90.5.2 NormalModulus() . . . 1793
4.90.5.3 MakeInverses() . . . 1794
4.90.5.4 GetModInverses() . . . 1794
4.90.5.5 GetLongModInverses() . . . 1794
4.90.5.6 Product() . . . 1795
4.90.5.7 Quotient() . . . 1795
4.90.5.8 Remain10() . . . 1795
4.90.5.9 Remain4() . . . 1795
4.90.5.10 PrtLong() . . . 1795
4.90.5.11 GetLong() . . . 1795
4.90.5.12 GcdLong() . . . 1796
4.90.5.13 GetBinom() . . . 1796
4.90.5.14 LcmLong() . . . 1796
4.90.5.15 TakeLongRoot() . . . 1796
4.90.5.16 MakeRational() . . . 1796
4.90.5.17 MakeLongRational() . . . 1797
4.90.5.18 CompCoef() . . . 1797
4.90.5.19 Modulus() . . . 1797
4.90.5.20 TakeModulus() . . . 1797
4.90.5.21 TakeNormalModulus() . . . 1798
4.90.5.22 MakeModTable() . . . 1798
4.90.5.23 Factorial() . . . 1798
4.90.5.24 Bernoulli() . . . 1798
4.90.5.25 NextPrime() . . . 1798
4.90.5.26 iniwranf() . . . 1799
4.90.5.27 wranf() . . . 1799
4.90.5.28 iranf() . . . 1799
4.90.5.29 PreRandom() . . . 1799
4.91 reken.c . . . 1799
4.92 reshuf.c File Reference . . . 1844
4.92.1 Detailed Description . . . 1844
4.92.2 Macro Definition Documentation . . . 1845
4.92.2.1 NEWCODE . . . 1845
4.92.3 Function Documentation . . . 1845
4.92.3.1 ReNumber() . . . 1845
4.92.3.2 FunLevel() . . . 1845
4.92.3.3 DetCurDum() . . . 1845
4.92.3.4 FullRenumber() . . . 1845
4.92.3.5 MoveDummies() . . . 1845
4.92.3.6 AdjustRenumScratch() . . . 1846
4.92.3.7 CountDo() . . . 1846
4.92.3.8 CountFun() . . . 1846

4.92.3.9 DimensionSubterm() . . . 1846
4.92.3.10 DimensionTerm() . . . 1846
4.92.3.11 DimensionExpression() . . . 1846
4.92.3.12 MultDo() . . . 1847
4.92.3.13 TryDo() . . . 1847
4.92.3.14 DoDistrib() . . . 1847
4.92.3.15 EqualArg() . . . 1847
4.92.3.16 DoDelta3() . . . 1847
4.92.3.17 TestPartitions() . . . 1847
4.92.3.18 DoPartitions() . . . 1848
4.92.3.19 DoPermutations() . . . 1848
4.92.3.20 DoShuffle() . . . 1848
4.92.3.21 Shuffle() . . . 1848
4.92.3.22 FinishShuffle() . . . 1848
4.92.3.23 DoStuffle() . . . 1848
4.92.3.24 Stuffle() . . . 1849
4.92.3.25 FinishStuffle() . . . 1849
4.92.3.26 StuffRootAdd() . . . 1849
4.93 reshuf.c . . . 1849
4.94 sch.c File Reference . . . 1885
4.94.1 Detailed Description . . . 1886
4.94.2 Macro Definition Documentation . . . 1886
4.94.2.1 va_dcl . . . 1886
4.94.2.2 va_start . . . 1886
4.94.2.3 va_end . . . 1887
4.94.2.4 va_arg . . . 1887
4.94.3 Typedef Documentation . . . 1887
4.94.3.1 va_list . . . 1887
4.94.4 Function Documentation . . . 1887
4.94.4.1 StrCopy() . . . 1887
4.94.4.2 AddToLine() . . . 1887
4.94.4.3 FiniLine() . . . 1887
4.94.4.4 IniLine() . . . 1888
4.94.4.5 LongToLine() . . . 1888
4.94.4.6 RatToLine() . . . 1888
4.94.4.7 TalToLine() . . . 1888
4.94.4.8 TokenToLine() . . . 1888
4.94.4.9 CodeToLine() . . . 1888
4.94.4.10 MultiplyToLine() . . . 1889
4.94.4.11 AddArrayIndex() . . . 1889
4.94.4.12 PrtTerms() . . . 1889
4.94.4.13 WrtPower() . . . 1889

4.94.4.14 PrintTime() . . . 1889
4.94.4.15 WriteLists() . . . 1889
4.94.4.16 WriteDictionary() . . . 1889
4.94.4.17 WriteArgument() . . . 1890
4.94.4.18 WriteSubTerm() . . . 1890
4.94.4.19 WriteInnerTerm() . . . 1890
4.94.4.20 WriteTerm() . . . 1890
4.94.4.21 WriteExpression() . . . 1890
4.94.4.22 WriteAll() . . . 1890
4.94.4.23 WriteOne() . . . 1891
4.95 sch.c . . . 1891
4.96 setfile.c File Reference . . . 1923
4.96.1 Detailed Description . . . 1924
4.96.2 Macro Definition Documentation . . . 1924
4.96.2.1 NUMERICALVALUE . . . 1924
4.96.2.2 STRINGVALUE . . . 1924
4.96.2.3 PATHVALUE . . . 1924
4.96.2.4 ONOFFVALUE . . . 1924
4.96.2.5 DEFINEVALUE . . . 1924
4.96.2.6 SETBUFSIZE . . . 1925
4.96.3 Function Documentation . . . 1925
4.96.3.1 DoSetups() . . . 1925
4.96.3.2 ProcessOption() . . . 1925
4.96.3.3 GetSetupPar() . . . 1925
4.96.3.4 RecalcSetups() . . . 1925
4.96.3.5 AllocSetups() . . . 1925
4.96.3.6 WriteSetup() . . . 1925
4.96.3.7 AllocSort() . . . 1926
4.96.3.8 AllocSortFileName() . . . 1926
4.96.3.9 AllocFileHandle() . . . 1926
4.96.3.10 DeAllocFileHandle() . . . 1926
4.96.3.11 MakeSetupAllocs() . . . 1926
4.96.3.12 TryFileSetups() . . . 1926
4.96.3.13 TryEnvironment() . . . 1927
4.96.4 Variable Documentation . . . 1927
4.96.4.1 curdirp . . . 1927
4.96.4.2 cursortdirp . . . 1927
4.96.4.3 commentchar . . . 1927
4.96.4.4 dotchar . . . 1927
4.96.4.5 highfirst . . . 1927
4.96.4.6 lowfirst . . . 1927
4.96.4.7 procedureextension . . . 1928

4.96.4.8 setupparameters . . . 1928
4.97 setfile.c . . . 1928
4.98 smart.c File Reference . . . 1942
4.98.1 Detailed Description . . . 1942
4.98.2 Function Documentation . . . 1942
4.98.2.1 StudyPattern() . . . 1942
4.98.2.2 MatchIsPossible() . . . 1943
4.99 smart.c . . . 1943
4.100 sort.c File Reference . . . 1948
4.100.1 Detailed Description . . . 1949
4.100.2 Macro Definition Documentation . . . 1949
4.100.2.1 NEWSPLITMERGE . . . 1949
4.100.2.2 INSLENGTH . . . 1949
4.100.3 Function Documentation . . . 1949
4.100.3.1 WriteStats() . . . 1949
4.100.3.2 NewSort() . . . 1950
4.100.3.3 EndSort() . . . 1950
4.100.3.4 PutIn() . . . 1951
4.100.3.5 Sflush() . . . 1952
4.100.3.6 PutOut() . . . 1952
4.100.3.7 FlushOut() . . . 1953
4.100.3.8 AddCoef() . . . 1954
4.100.3.9 AddPoly() . . . 1954
4.100.3.10 AddArgs() . . . 1955
4.100.3.11 Compare1() . . . 1956
4.100.3.12 CompareSymbols() . . . 1956
4.100.3.13 CompareHSymbols() . . . 1957
4.100.3.14 ComPress() . . . 1957
4.100.3.15 SplitMerge() . . . 1958
4.100.3.16 GarbHand() . . . 1958
4.100.3.17 MergePatches() . . . 1959
4.100.3.18 StoreTerm() . . . 1960
4.100.3.19 StageSort() . . . 1960
4.100.3.20 SortWild() . . . 1961
4.100.3.21 CleanUpSort() . . . 1961
4.100.3.22 LowerSortLevel() . . . 1961
4.100.3.23 PolyRatFunSpecial() . . . 1961
4.100.4 Variable Documentation . . . 1962
4.100.4.1 toterms . . . 1962
4.101 sort.c . . . 1962
4.102 spectator.c File Reference . . . 2013
4.102.1 Detailed Description . . . 2013

4.102.2 Function Documentation . . . 2013
4.102.2.1 CoCreateSpectator() . . . 2013
4.102.2.2 CoToSpectator() . . . 2013
4.102.2.3 CoRemoveSpectator() . . . 2014
4.102.2.4 CoEmptySpectator() . . . 2014
4.102.2.5 PutInSpectator() . . . 2014
4.102.2.6 FlushSpectators() . . . 2014
4.102.2.7 CoCopySpectator() . . . 2014
4.102.2.8 GetFromSpectator() . . . 2014
4.102.2.9 ClearSpectators() . . . 2014
4.103 spectator.c . . . 2015
4.104 startup.c File Reference . . . 2023
4.104.1 Detailed Description . . . 2024
4.104.2 Macro Definition Documentation . . . 2024
4.104.2.1 STRINGIFY . . . 2024
4.104.2.2 STRINGIFY__ . . . 2024
4.104.2.3 FORMNAME . . . 2024
4.104.2.4 VERSIONSTR__ . . . 2024
4.104.2.5 VERSIONSTR . . . 2024
4.104.2.6 TAKEPATH . . . 2025
4.104.3 Function Documentation . . . 2025
4.104.3.1 DoTail() . . . 2025
4.104.3.2 OpenInput() . . . 2025
4.104.3.3 ReserveTempFiles() . . . 2025
4.104.3.4 StartVariables() . . . 2025
4.104.3.5 StartMore() . . . 2026
4.104.3.6 IniVars() . . . 2026
4.104.3.7 main() . . . 2026
4.104.3.8 CleanUp() . . . 2026
4.104.3.9 Terminate() . . . 2027
4.104.3.10 PrintRunningTime() . . . 2027
4.104.3.11 GetRunningTime() . . . 2027
4.104.4 Variable Documentation . . . 2027
4.104.4.1 emptystring . . . 2027
4.104.4.2 defaulttempfilename . . . 2027
4.105 startup.c . . . 2028
4.106 store.c File Reference . . . 2049
4.106.1 Detailed Description . . . 2050
4.106.2 Macro Definition Documentation . . . 2050
4.106.2.1 SAVEREVISION . . . 2050
4.106.3 Function Documentation . . . 2050
4.106.3.1 OpenTemp() . . . 2050

4.106.3.2 SeekScratch() . . . 2051
4.106.3.3 SetEndScratch() . . . 2051
4.106.3.4 SetEndHScratch() . . . 2051
4.106.3.5 SetScratch() . . . 2051
4.106.3.6 RevertScratch() . . . 2051
4.106.3.7 ResetScratch() . . . 2051
4.106.3.8 ReadFromScratch() . . . 2052
4.106.3.9 AddToScratch() . . . 2052
4.106.3.10 CoSave() . . . 2052
4.106.3.11 CoLoad() . . . 2052
4.106.3.12 DeleteStore() . . . 2052
4.106.3.13 PutInStore() . . . 2052
4.106.3.14 GetTerm() . . . 2053
4.106.3.15 GetOneTerm() . . . 2053
4.106.3.16 GetMoreTerms() . . . 2053
4.106.3.17 GetMoreFromMem() . . . 2053
4.106.3.18 GetFromStore() . . . 2053
4.106.3.19 DetVars() . . . 2053
4.106.3.20 ToStorage() . . . 2054
4.106.3.21 NextFileIndex() . . . 2054
4.106.3.22 SetFileIndex() . . . 2054
4.106.3.23 VarStore() . . . 2054
4.106.3.24 TermRenumber() . . . 2055
4.106.3.25 FindrNumber() . . . 2055
4.106.3.26 FindInIndex() . . . 2055
4.106.3.27 GetTable() . . . 2056
4.106.3.28 CopyExpression() . . . 2056
4.106.3.29 WriteStoreHeader() . . . 2056
4.106.3.30 ReadSaveHeader() . . . 2056
4.106.3.31 ReadSaveIndex() . . . 2057
4.106.3.32 ReadSaveVariables() . . . 2057
4.106.3.33 ReadSaveTerm32() . . . 2058
4.106.3.34 ReadSaveExpression() . . . 2059
4.107 store.c . . . 2060
4.108 structs.h File Reference . . . 2117
4.108.1 Detailed Description . . . 2120
4.108.2 Macro Definition Documentation . . . 2120
4.108.2.1 INFILEINDEX . . . 2120
4.108.2.2 EMPTYININDEX . . . 2120
4.108.3 Typedef Documentation . . . 2121
4.108.3.1 INDEXENTRY . . . 2121
4.108.3.2 FILEINDEX . . . 2121

4.108.3.3 VARRENUM . . . 2121
4.108.3.4 RENUMBER . . . 2121
4.108.3.5 NAMENODE . . . 2121
4.108.3.6 NAMETREE . . . 2121
4.108.3.7 COMPTREE . . . 2122
4.108.3.8 TABLES . . . 2122
4.108.3.9 FUNCTIONS . . . 2122
4.108.3.10 FILEHANDLE . . . 2122
4.108.3.11 STREAM . . . 2122
4.108.3.12 TRACES . . . 2122
4.108.3.13 TRACEN . . . 2123
4.108.3.14 PREVAR . . . 2123
4.108.3.15 DOLOOP . . . 2123
4.108.3.16 set_of_char . . . 2123
4.108.3.17 one_byte . . . 2123
4.108.3.18 CBUF . . . 2123
4.108.3.19 CHANNEL . . . 2124
4.108.3.20 NESTING . . . 2124
4.108.3.21 STORECACHE . . . 2124
4.108.3.22 PERM . . . 2124
4.108.3.23 PERMP . . . 2124
4.108.3.24 DISTRIBUTE . . . 2124
4.108.3.25 PARTI . . . 2124
4.108.3.26 SORTING . . . 2125
4.108.3.27 ALLGLOBALS . . . 2125
4.108.3.28 WCN . . . 2125
4.108.3.29 WCN2 . . . 2125
4.108.3.30 FIXEDGLOBALS . . . 2125
4.109 structs.h . . . 2126
4.110 symmetr.c File Reference . . . 2149
4.110.1 Detailed Description . . . 2149
4.110.2 Function Documentation . . . 2150
4.110.2.1 MatchE() . . . 2150
4.110.2.2 Permute() . . . 2150
4.110.2.3 PermuteP() . . . 2150
4.110.2.4 Distribute() . . . 2150
4.110.2.5 MatchCy() . . . 2150
4.110.2.6 FunMatchCy() . . . 2151
4.110.2.7 FunMatchSy() . . . 2151
4.110.2.8 MatchArgument() . . . 2151
4.111 symmetr.c . . . 2151
4.112 tables.c File Reference . . . 2178

4.112.1 Detailed Description . . . 2179
4.112.2 Function Documentation . . . 2179
4.112.2.1 ClearTableTree() . . . 2179
4.112.2.2 InsTableTree() . . . 2180
4.112.2.3 RedoTableTree() . . . 2180
4.112.2.4 FindTableTree() . . . 2180
4.112.2.5 DoTableExpansion() . . . 2180
4.112.2.6 CoTableBase() . . . 2180
4.112.2.7 FlipTable() . . . 2180
4.112.2.8 SpareTable() . . . 2181
4.112.2.9 FindTB() . . . 2181
4.112.2.10 CoTBcreate() . . . 2181
4.112.2.11 CoTBopen() . . . 2181
4.112.2.12 CoTBaddto() . . . 2181
4.112.2.13 CoTBenter() . . . 2181
4.112.2.14 CoTestUse() . . . 2181
4.112.2.15 CheckTableDeclarations() . . . 2182
4.112.2.16 CoTBload() . . . 2182
4.112.2.17 TestUse() . . . 2182
4.112.2.18 CoTBaudit() . . . 2182
4.112.2.19 CoTBon() . . . 2182
4.112.2.20 CoTBoff() . . . 2182
4.112.2.21 CoTBcleanup() . . . 2182
4.112.2.22 CoTBreplace() . . . 2183
4.112.2.23 CoTBuse() . . . 2183
4.112.2.24 CoApply() . . . 2183
4.112.2.25 CoTBhelp() . . . 2183
4.112.2.26 ReWorkT() . . . 2183
4.112.2.27 Apply() . . . 2183
4.112.2.28 ApplyExec() . . . 2184
4.112.2.29 ApplyReset() . . . 2184
4.112.2.30 TableReset() . . . 2184
4.112.2.31 ReleaseTB() . . . 2184
4.112.3 Variable Documentation . . . 2184
4.112.3.1 helptb . . . 2184
4.113 tables.c . . . 2185
4.114 threads.c File Reference . . . 2211
4.114.1 Detailed Description . . . 2211
4.115 threads.c . . . 2211
4.116 token.c File Reference . . . 2262
4.116.1 Detailed Description . . . 2263
4.116.2 Macro Definition Documentation . . . 2263

4.116.2.1 CHECKPOLY . . . 2263
4.116.3 Function Documentation . . . 2263
4.116.3.1 tokenize() . . . 2263
4.116.3.2 WriteTokens() . . . 2263
4.116.3.3 simp1token() . . . 2264
4.116.3.4 simpwtoken() . . . 2264
4.116.3.5 simp2token() . . . 2264
4.116.3.6 simp3atoken() . . . 2264
4.116.3.7 simp3btoken() . . . 2264
4.116.3.8 simp4token() . . . 2264
4.116.3.9 simp5token() . . . 2265
4.116.3.10 simp6token() . . . 2265
4.116.4 Variable Documentation . . . 2265
4.116.4.1 ttypes . . . 2265
4.117 token.c . . . 2265
4.118 tools.c File Reference . . . 2289
4.118.1 Detailed Description . . . 2291
4.118.2 Macro Definition Documentation . . . 2291
4.118.2.1 FILLVALUE . . . 2291
4.118.2.2 TERMMEMSTARTNUM . . . 2291
4.118.2.3 TERMEXTRAWORDS . . . 2292
4.118.2.4 NUMBERMEMSTARTNUM . . . 2292
4.118.2.5 NUMBEREXTRAWORDS . . . 2292
4.118.2.6 DODOUBLE . . . 2292
4.118.2.7 DOEXPAND . . . 2292
4.118.3 Function Documentation . . . 2293
4.118.3.1 LoadInputFile() . . . 2293
4.118.3.2 ReadFromStream() . . . 2293
4.118.3.3 GetFromStream() . . . 2293
4.118.3.4 LookInStream() . . . 2293
4.118.3.5 OpenStream() . . . 2293
4.118.3.6 LocateFile() . . . 2293
4.118.3.7 CloseStream() . . . 2294
4.118.3.8 CreateStream() . . . 2294
4.118.3.9 GetStreamPosition() . . . 2294
4.118.3.10 PositionStream() . . . 2294
4.118.3.11 ReverseStatements() . . . 2294
4.118.3.12 StartFiles() . . . 2294
4.118.3.13 OpenFile() . . . 2294
4.118.3.14 OpenAddFile() . . . 2295
4.118.3.15 ReOpenFile() . . . 2295
4.118.3.16 CreateFile() . . . 2295

4.118.3.17 CreateLogFile() . . . 2295
4.118.3.18 CloseFile() . . . 2295
4.118.3.19 CopyFile() . . . 2295
4.118.3.20 CreateHandle() . . . 2296
4.118.3.21 ReadFile() . . . 2296
4.118.3.22 ReadPosFile() . . . 2296
4.118.3.23 WriteFileToFile() . . . 2296
4.118.3.24 SeekFile() . . . 2296
4.118.3.25 TellFile() . . . 2296
4.118.3.26 TELLFILE() . . . 2297
4.118.3.27 FlushFile() . . . 2297
4.118.3.28 GetPosFile() . . . 2297
4.118.3.29 SetPosFile() . . . 2297
4.118.3.30 SynchFile() . . . 2297
4.118.3.31 TruncateFile() . . . 2297
4.118.3.32 GetChannel() . . . 2298
4.118.3.33 GetAppendChannel() . . . 2298
4.118.3.34 CloseChannel() . . . 2298
4.118.3.35 UpdateMaxSize() . . . 2298
4.118.3.36 StrCmp() . . . 2298
4.118.3.37 StrICmp() . . . 2298
4.118.3.38 StrHICmp() . . . 2298
4.118.3.39 StrICont() . . . 2299
4.118.3.40 ConWord() . . . 2299
4.118.3.41 StrLen() . . . 2299
4.118.3.42 NumToStr() . . . 2299
4.118.3.43 WriteString() . . . 2299
4.118.3.44 WriteUnfinString() . . . 2299
4.118.3.45 AddToString() . . . 2300
4.118.3.46 strDup1() . . . 2300
4.118.3.47 EndOfToken() . . . 2300
4.118.3.48 ToToken() . . . 2300
4.118.3.49 SkipField() . . . 2300
4.118.3.50 ReadSnum() . . . 2300
4.118.3.51 NumCopy() . . . 2301
4.118.3.52 LongCopy() . . . 2301
4.118.3.53 LongLongCopy() . . . 2301
4.118.3.54 MakeDate() . . . 2301
4.118.3.55 set_in() . . . 2301
4.118.3.56 set_set() . . . 2301
4.118.3.57 set_del() . . . 2302
4.118.3.58 set_sub() . . . 2302

4.118.3.59 iniTools() . . . 2302
4.118.3.60 Malloc() . . . 2302
4.118.3.61 Malloc1() . . . 2302
4.118.3.62 M_free() . . . 2302
4.118.3.63 M_check1() . . . 2303
4.118.3.64 M_print() . . . 2303
4.118.3.65 TermMallocAddMemory() . . . 2303
4.118.3.66 NumberMallocAddMemory() . . . 2303
4.118.3.67 CacheNumberMallocAddMemory() . . . 2303
4.118.3.68 FromList() . . . 2303
4.118.3.69 From0List() . . . 2303
4.118.3.70 FromVarList() . . . 2304
4.118.3.71 DoubleList() . . . 2304
4.118.3.72 DoubleLList() . . . 2304
4.118.3.73 DoubleBuffer() . . . 2304
4.118.3.74 ExpandBuffer() . . . 2304
4.118.3.75 iexp() . . . 2305
4.118.3.76 ToGeneral() . . . 2305
4.118.3.77 ToFast() . . . 2305
4.118.3.78 ToPolyFunGeneral() . . . 2305
4.118.3.79 IsLikeVector() . . . 2305
4.118.3.80 AreArgsEqual() . . . 2305
4.118.3.81 CompareArgs() . . . 2306
4.118.3.82 CompArg() . . . 2306
4.118.3.83 TimeWallClock() . . . 2306
4.118.3.84 TimeChildren() . . . 2306
4.118.3.85 TimeCPU() . . . 2306
4.118.3.86 Crash() . . . 2307
4.118.3.87 TestTerm() . . . 2307
4.118.4 Variable Documentation . . . 2308
4.118.4.1 filelist . . . 2308
4.118.4.2 numinfilelist . . . 2308
4.118.4.3 filelistsize . . . 2308
4.118.4.4 WriteFile . . . 2308
4.119 tools.c . . . 2308
4.120 transform.c File Reference . . . 2353
4.120.1 Detailed Description . . . 2353
4.120.2 Function Documentation . . . 2353
4.120.2.1 CoTransform() . . . 2353
4.120.2.2 RunTransform() . . . 2354
4.120.2.3 RunEncode() . . . 2354
4.120.2.4 RunDecode() . . . 2354

4.120.2.5 RunReplace() . . . 2354
4.120.2.6 RunImplode() . . . 2354
4.120.2.7 RunExplode() . . . 2354
4.120.2.8 RunPermute() . . . 2355
4.120.2.9 RunReverse() . . . 2355
4.120.2.10 RunDedup() . . . 2355
4.120.2.11 RunCycle() . . . 2355
4.120.2.12 RunAddArg() . . . 2355
4.120.2.13 RunMulArg() . . . 2355
4.120.2.14 RunIsLyndon() . . . 2356
4.120.2.15 RunToLyndon() . . . 2356
4.120.2.16 RunDropArg() . . . 2356
4.120.2.17 RunSelectArg() . . . 2356
4.120.2.18 TestArgNum() . . . 2356
4.120.2.19 PutArgInScratch() . . . 2356
4.120.2.20 ReadRange() . . . 2357
4.120.2.21 FindRange() . . . 2357
4.121 transform.c . . . 2357
4.122 unix.h File Reference . . . 2396
4.122.1 Detailed Description . . . 2397
4.122.2 Macro Definition Documentation . . . 2397
4.122.2.1 LINEFEED . . . 2397
4.122.2.2 CARRIAGERETURN . . . 2397
4.122.2.3 WITHPIPE . . . 2397
4.122.2.4 WITHSYSTEM . . . 2397
4.122.2.5 WITHEXTERNALCHANNEL . . . 2397
4.122.2.6 TRAPSIGNALS . . . 2397
4.122.2.7 P_term . . . 2398
4.122.2.8 SEPARATOR . . . 2398
4.122.2.9 ALTSEPARATOR . . . 2398
4.122.2.10 PATHSEPARATOR . . . 2398
4.122.2.11 WITH_ENV . . . 2398
4.123 unix.h . . . 2398
4.124 unixfile.c File Reference . . . 2399
4.124.1 Detailed Description . . . 2399
4.125 unixfile.c . . . 2399
4.126 variable.h File Reference . . . 2402
4.126.1 Detailed Description . . . 2404
4.126.2 Macro Definition Documentation . . . 2404
4.126.2.1 chartype . . . 2404
4.126.2.2 Procedures . . . 2404
4.126.2.3 NumProcedures . . . 2404

4.126.2.4 MaxProcedures . . . 2404
4.126.2.5 DoLoops . . . 2405
4.126.2.6 NumDoLoops . . . 2405
4.126.2.7 MaxDoLoops . . . 2405
4.126.2.8 PreVar . . . 2405
4.126.2.9 NumPre . . . 2405
4.126.2.10 MaxNumPre . . . 2405
4.126.2.11 SetElements . . . 2405
4.126.2.12 Sets . . . 2405
4.126.2.13 functions . . . 2406
4.126.2.14 indices . . . 2406
4.126.2.15 symbols . . . 2406
4.126.2.16 vectors . . . 2406
4.126.2.17 tablebases . . . 2406
4.126.2.18 NumFunctions . . . 2406
4.126.2.19 NumIndices . . . 2406
4.126.2.20 NumSymbols . . . 2406
4.126.2.21 NumVectors . . . 2407
4.126.2.22 NumSets . . . 2407
4.126.2.23 NumSetElements . . . 2407
4.126.2.24 NumTableBases . . . 2407
4.126.2.25 GlobalFunctions . . . 2407
4.126.2.26 GlobalIndices . . . 2407
4.126.2.27 GlobalSymbols . . . 2407
4.126.2.28 GlobalVectors . . . 2407
4.126.2.29 GlobalSets . . . 2408
4.126.2.30 GlobalSetElements . . . 2408
4.126.2.31 cbuf . . . 2408
4.126.2.32 channels . . . 2408
4.126.2.33 NumOutputChannels . . . 2408
4.126.2.34 Dollars . . . 2408
4.126.2.35 NumDollars . . . 2408
4.126.2.36 Dubious . . . 2408
4.126.2.37 NumDubious . . . 2409
4.126.2.38 Expressions . . . 2409
4.126.2.39 NumExpressions . . . 2409
4.126.2.40 autofunctions . . . 2409
4.126.2.41 autoindices . . . 2409
4.126.2.42 autosymbols . . . 2409
4.126.2.43 autovectors . . . 2409
4.126.2.44 xsymbol . . . 2409
4.126.2.45 numxsymbol . . . 2410

4.126.2.46 PotModdollars . . . 2410
4.126.2.47 NumPotModdollars . . . 2410
4.126.2.48 ModOptdollars . . . 2410
4.126.2.49 NumModOptdollars . . . 2410
4.126.2.50 BUG . . . 2410
4.126.2.51 AC . . . 2410
4.126.2.52 AM . . . 2410
4.126.2.53 AN . . . 2411
4.126.2.54 AO . . . 2411
4.126.2.55 AP . . . 2411
4.126.2.56 AR . . . 2411
4.126.2.57 AS . . . 2411
4.126.2.58 AT . . . 2411
4.126.2.59 AX . . . 2411
4.126.3 Variable Documentation . . . 2412
4.126.3.1 WriteFile . . . 2412
4.126.3.2 A . . . 2412
4.126.3.3 FG . . . 2412
4.126.3.4 fixedsets . . . 2412
4.126.3.5 setupfilename . . . 2412
4.127 variable.h . . . 2413
4.128 vector.h File Reference . . . 2414
4.128.1 Detailed Description . . . 2416
4.128.2 Macro Definition Documentation . . . 2416
4.128.2.1 VectorStruct . . . 2416
4.128.2.2 Vector . . . 2416
4.128.2.3 DeclareVector . . . 2417
4.128.2.4 VectorInit . . . 2417
4.128.2.5 VectorFree . . . 2417
4.128.2.6 VectorPtr . . . 2418
4.128.2.7 VectorFront . . . 2418
4.128.2.8 VectorBack . . . 2418
4.128.2.9 VectorSize . . . 2419
4.128.2.10 VectorCapacity . . . 2419
4.128.2.11 VectorEmpty . . . 2419
4.128.2.12 VectorClear . . . 2420
4.128.2.13 VectorReserve . . . 2420
4.128.2.14 VectorPushBack . . . 2421
4.128.2.15 VectorPushBacks . . . 2421
4.128.2.16 VectorPopBack . . . 2421
4.128.2.17 VectorInsert . . . 2422
4.128.2.18 VectorInserts . . . 2422

4.128.2.19 VectorErase . . . 2423
4.128.2.20 VectorErases . . . 2423
4.129 vector.h . . . 2423
4.130 wildcard.c File Reference . . . 2427
4.130.1 Detailed Description . . . 2427
4.130.2 Macro Definition Documentation . . . 2427
4.130.2.1 DEBUG . . . 2427
4.130.3 Function Documentation . . . 2428
4.130.3.1 WildFill() . . . 2428
4.130.3.2 ResolveSet() . . . 2428
4.130.3.3 ClearWild() . . . 2428
4.130.3.4 AddWild() . . . 2428
4.130.3.5 CheckWild() . . . 2428
4.130.3.6 DenToFunction() . . . 2429
4.131 wildcard.c . . . 2429

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

AllGlobals 5
ARGBUFFER 7
bit_field 9
BrAcKeTiNdEx 10
BrAcKeTiNfO 12
bufIPstruct 14
C_const 15
CbUf 46
ChAnNeL 50
COST 51
dbase 52
DICTIONARY 55
DICTIONARY_ELEMENT 57
DiStRiBuTe 58
dollar_buf 59
DoLIArS 60
DoLoOp 62
DuBiOuS 65
ExPrEsSiOn 67
FaCdOlLaR 71
factorized_poly 72
FiLe 73
FiLeDaTa 76
FiLeInDeX 78
FixedGlobals 79
fixedset 82
FUN_INFO 83
FuNcTiOn 85
gcd_heuristic_failed 88
HANDLERS 88
InDeX 90
indexblock 91
InDeXeNtRy 93
iniinfo 96
INSIDEINFO 97

KEYWORD . . . 99
LIST . . . 100
longMultiStruct . . . 102
M_const . . . 104
MiNmAx . . . 128
MODNUM . . . 129
MoDoPtDoLlArS . . . 130
monomial_larger . . . 130
N_const . . . 131
nameblock . . . 148
NaMeNode . . . 149
NaMeTree . . . 151
NeStInG . . . 154
NoDe . . . 155
O_const . . . 156
objects . . . 167
OPTIMIZE . . . 169
OPTIMIZERESULT . . . 172
P_const . . . 173
ParallelVars . . . 180
PaRtI . . . 183
PeRmUtE . . . 184
PeRmUtEp . . . 185
PF_BUFFER . . . 186
poly . . . 189
POLYMOD . . . 192
PoSiTiOn . . . 193
PRELOAD . . . 194
pReVaR . . . 194
PROCEDURE . . . 196
R_const . . . 197
ReNuMbEr . . . 206
S_const . . . 209
SeTs . . . 211
SETUPPARAMETERS . . . 212
SHvariables . . . 213
sOrT . . . 216
SpecTatoR . . . 225
StOrEcAcHe . . . 227
STOREHEADER . . . 228
StreaM . . . 232
SuBbUf . . . 236
SyMbOl . . . 237
T_const . . . 239
TaBlEbAsE . . . 252
TaBlEbAsEsUbInDeX . . . 254
TaBlEs . . . 255
TrAcEn . . . 261
TrAcEs . . . 263
tree . . . 269
VaRrEnUm . . . 270
VeCtOr . . . 271
X_const . . . 273

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

argument.c . . . 275
bugtool.c . . . 321
checkpoint.c . . . 322
comexpr.c . . . 367
compcomm.c . . . 394
compiler.c . . . 497
compress.c . . . 528
comtool.c . . . 537
comtool.h . . . 550
declare.h . . . 552
dict.c . . . 789
dollar.c . . . 807
execute.c . . . 861
extcmd.c . . . 896
factor.c . . . 916
findpat.c . . . 928
form3.h . . . 945
fsizes.h . . . 961
ftypes.h . . . 975
function.c . . . 1083
fwin.h . . . 1109
if.c . . . 1111
index.c . . . 1125
inivar.h . . . 1135
lus.c . . . 1139
mallocprotect.h . . . 1148
message.c . . . 1153
minos.c . . . 1167
minos.h . . . 1191
module.c . . . 1197
mpi.c . . . 1210
mpidbg.h . . . 1244
mytime.h . . . 1257
names.c . . . 1257
normal.c . . . 1304

notation.c . . . 1368
opera.c . . . 1385
parallel.c . . . 1417
parallel.h . . . 1491
pattern.c . . . 1527
poly.h . . . 1557
polyfact.h . . . 1560
polygcd.h . . . 1561
portsignals.h . . . 1562
pre.c . . . 1567
proces.c . . . 1665
ratio.c . . . 1738
reken.c . . . 1787
reshuf.c . . . 1844
sch.c . . . 1885
setfile.c . . . 1923
smart.c . . . 1942
sort.c . . . 1948
spectator.c . . . 2013
startup.c . . . 2023
store.c . . . 2049
structs.h . . . 2117
symmetr.c . . . 2149
tables.c . . . 2178
threads.c . . . 2211
token.c . . . 2262
tools.c . . . 2289
transform.c . . . 2353
unix.h . . . 2396
unixfile.c . . . 2399
variable.h . . . 2402
vector.h . . . 2414
wildcard.c . . . 2427

Chapter 3

Data Structure Documentation

3.1 AllGlobals Struct Reference

`#include <structs.h>`

Collaboration diagram for AllGlobals:

Public Member Functions

- **PADPOSITION** (0, 0, 0, 0, sizeof(struct P_const)+sizeof(struct T_const)+sizeof(struct X_const))

Data Fields

- struct M_const M
- struct C_const C
- struct S_const S
- struct R_const R
- struct N_const N
- struct O_const O
- struct P_const P
- struct T_const T
- struct X_const X

3.1.1 Detailed Description

Without pthreads (FORM) the ALLGLOBALS struct has all the global variables

Definition at line 2398 of file structs.h.

3.1.2 Field Documentation

3.1.2.1 M

```
struct M_const AllGlobals::M
```

Definition at line 2399 of file structs.h.

3.1.2.2 C

```
struct C_const AllGlobals::C
```

Definition at line 2400 of file structs.h.

3.1.2.3 S

```
struct S_const AllGlobals::S
```

Definition at line 2401 of file structs.h.

3.1.2.4 R

```
struct R_const AllGlobals::R
```

Definition at line 2402 of file structs.h.

3.1.2.5 N

```
struct N_const AllGlobals::N
```

Definition at line 2403 of file structs.h.

3.1.2.6 O

```
struct O_const AllGlobals::O
```

Definition at line 2404 of file structs.h.

3.1.2.7 P

```
struct P_const AllGlobals::P
```

Definition at line 2405 of file structs.h.

3.1.2.8 T

```
struct T_const AllGlobals::T
```

Definition at line 2406 of file structs.h.

3.1.2.9 X

```
struct X_const AllGlobals::X
```

Definition at line 2407 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.2 ARGBUFFER Struct Reference

Collaboration diagram for ARGBUFFER:

Data Fields

- WORD * buffer
- DOLLARS dollar
- LONG size
- int type
- int dummy

3.2.1 Detailed Description

Definition at line 601 of file ratio.c.

3.2.2 Field Documentation

3.2.2.1 buffer

```
WORD* ARGBUFFER::buffer
```

Definition at line 602 of file ratio.c.

3.2.2.2 dollar

```
DOLLARS ARGBUFFER::dollar
```

Definition at line 603 of file ratio.c.

3.2.2.3 size

```
LONG ARGBUFFER::size
```

Definition at line 604 of file ratio.c.

3.2.2.4 type

```
int ARGBUFFER::type
```

Definition at line 605 of file ratio.c.

3.2.2.5 dummy

```
int ARGBUFFER::dummy
```

Definition at line 606 of file ratio.c.

The documentation for this struct was generated from the following file:

- ratio.c

3.3 bit_field Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE bit_0: 1
- UBYTE bit_1: 1
- UBYTE bit_2: 1
- UBYTE bit_3: 1
- UBYTE bit_4: 1
- UBYTE bit_5: 1
- UBYTE bit_6: 1
- UBYTE bit_7: 1

3.3.1 Detailed Description

The struct bit_field is used by set_in, set_set, set_del and set_sub. They in turn are used in pre.c to toggle bits that indicate whether a character can be used as a separator of function arguments. This facility is used in the communication with external channels.

Definition at line 865 of file structs.h.

3.3.2 Field Documentation

3.3.2.1 bit_0

```
UBYTE bit_field::bit_0
```

Definition at line 866 of file structs.h.

3.3.2.2 bit_1

```
UBYTE bit_field::bit_1
```

Definition at line 867 of file structs.h.

3.3.2.3 bit_2

```
UBYTE bit_field::bit_2
```

Definition at line 868 of file structs.h.

3.3.2.4 bit_3

```
UBYTE bit_field::bit_3
```

Definition at line 869 of file structs.h.

3.3.2.5 bit_4

```
UBYTE bit_field::bit_4
```

Definition at line 870 of file structs.h.

3.3.2.6 bit_5

```
UBYTE bit_field::bit_5
```

Definition at line 871 of file structs.h.

3.3.2.7 bit_6

```
UBYTE bit_field::bit_6
```

Definition at line 872 of file structs.h.

3.3.2.8 bit_7

```
UBYTE bit_field::bit_7
```

Definition at line 873 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.4 BrAcKeTiNdEx Struct Reference

Collaboration diagram for BrAcKeTiNdEx:

Public Member Functions

- **PADPOSITION** (0, 2, 0, 0, 0)

Data Fields

- POSITION start
- POSITION next
- LONG bracket
- LONG termsinbracket

3.4.1 Detailed Description

Definition at line 304 of file structs.h.

3.4.2 Field Documentation

3.4.2.1 start

```
POSITION BrAcKeTiNdEx::start
```

Definition at line 305 of file structs.h.

3.4.2.2 next

```
POSITION BrAcKeTiNdEx::next
```

Definition at line 306 of file structs.h.

3.4.2.3 bracket

```
LONG BrAcKeTiNdEx::bracket
```

Definition at line 307 of file structs.h.

3.4.2.4 termsinbracket

```
LONG BrAcKeTiNdEx::termsinbracket
```

Definition at line 308 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.5 BrAcKeTiNfO Struct Reference

Collaboration diagram for BrAcKeTiNfO:

Data Fields

- BRACKETINDEX * indexbuffer
- WORD * bracketbuffer
- LONG bracketbuffersize
- LONG indexbuffersize
- LONG bracketfill
- LONG indexfill
- WORD SortType

3.5.1 Detailed Description

Definition at line 316 of file structs.h.

3.5.2 Field Documentation

3.5.2.1 indexbuffer

```
BRACKETINDEX* BrAcKeTiNfO::indexbuffer
```

[D]

Definition at line 317 of file structs.h.

Referenced by DoRecovery(), and FlushOut().

3.5.2.2 bracketbuffer

```
WORD* BrAcKeTiNfO::bracketbuffer
```

[D]

Definition at line 318 of file structs.h.

Referenced by DoRecovery().

3.5.2.3 bracketbuffersize

```
LONG BrAcKeTiNfO::bracketbuffersize
```

Definition at line 319 of file structs.h.

3.5.2.4 indexbuffersize

```
LONG BrAcKeTiNfO::indexbuffersize
```

Definition at line 320 of file structs.h.

3.5.2.5 bracketfill

```
LONG BrAcKeTiNfO::bracketfill
```

Definition at line 321 of file structs.h.

3.5.2.6 indexfill

```
LONG BrAcKeTiNfO::indexfill
```

Definition at line 322 of file structs.h.

3.5.2.7 SortType

```
WORD BrAcKeTiNfO::SortType
```

The sorting criterium used (like POWERFIRST etc)

Definition at line 323 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.6 bufIPstruct Struct Reference

Collaboration diagram for bufIPstruct:

Data Fields

- LONG i
- struct ExPrEsSiOn e

3.6.1 Detailed Description

Definition at line 3908 of file parallel.c.

3.6.2 Field Documentation

3.6.2.1 i

```
LONG bufIPstruct::i
```

Definition at line 3909 of file parallel.c.

3.6.2.2 e

```
struct ExPrEsSiOn bufIPstruct::e
```

Definition at line 3910 of file parallel.c.

The documentation for this struct was generated from the following file:

- parallel.c

3.7 C_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for C_const:

Public Member Functions

- **PADPOSITION** (45, 8+3 ∗MAXNEST, 70, 45+3 ∗MAXNEST+MAXREPEAT, COMMERCIALSIZE+MAXFLAGS+4+sizeof(LIST) ∗17)

Data Fields

- set_of_char separators
- POSITION StoreFileSize
- NAMETREE ∗ dollarnames
- NAMETREE ∗ exprnames
- NAMETREE ∗ varnames
- LIST ChannelList
- LIST DubiousList
- LIST FunctionList
- LIST ExpressionList

- LIST IndexList
- LIST SetElementList
- LIST SetList
- LIST SymbolList
- LIST VectorList
- LIST PotModDolList
- LIST ModOptDolList
- LIST TableBaseList
- LIST cbufList
- LIST AutoSymbolList
- LIST AutoIndexList
- LIST AutoVectorList
- LIST AutoFunctionList
- NAMETREE * autonames
- LIST * Symbols
- LIST * Indices
- LIST * Vectors
- LIST * Functions
- NAMETREE ** activenames
- STREAM * Streams
- STREAM * CurrentStream
- LONG * termstack
- LONG * termsortstack
- UWORD * cmod
- UWORD * powmod
- UWORD * modpowers
- UWORD * halfmod
- WORD * ProtoType
- WORD * WildC
- LONG * IfHeap
- LONG * IfCount
- LONG * IfStack
- UBYTE * iBuffer
- UBYTE * iPointer
- UBYTE * iStop
- UBYTE ** LabelNames
- WORD * FixIndices
- WORD * termsumcheck
- UBYTE * WildcardNames
- int * Labels
- SBYTE * tokens
- SBYTE * toptokens
- SBYTE * endoftokens
- WORD * tokenarglevel
- UWORD * modinverses
- UBYTE * Fortran90Kind
- WORD ** MultiBracketBuf
- UBYTE * extrasym
- WORD * doloopstack
- WORD * doloopnest
- char * CheckpointRunAfter
- char * CheckpointRunBefore
- WORD * IfSumCheck
- WORD * CommuteInSet
- UBYTE * TestValue

- LONG argstack [MAXNEST]
- LONG insidestack [MAXNEST]
- LONG inexprstack [MAXNEST]
- LONG iBufferSize
- LONG TransEname
- LONG ProcessBucketSize
- LONG mProcessBucketSize
- LONG CModule
- LONG ThreadBucketSize
- LONG CheckpointStamp
- LONG CheckpointInterval
- int cbufnum
- int AutoDeclareFlag
- int NoShowInput
- int ShortStats
- int compiletype
- int firstconstindex
- int insidefirst
- int minsidefirst
- int wildflag
- int NumLabels
- int MaxLabels
- int lDefDim
- int lDefDim4
- int NumWildcardNames
- int WildcardBufferSize
- int MaxIf
- int NumStreams
- int MaxNumStreams
- int firstctypemessage
- int tablecheck
- int idoption
- int BottomLevel
- int CompileLevel
- int TokensWriteFlag
- int UnsureDollarMode
- int outsidefun
- int funpowers
- int WarnFlag
- int StatsFlag
- int NamesFlag
- int CodesFlag
- int SetupFlag
- int SortType
- int lSortType
- int ThreadStats
- int FinalStats
- int OldParallelStats
- int ThreadsFlag
- int ThreadBalancing
- int ThreadSortFileSynch
- int ProcessStats
- int BracketNormalize
- int maxtermlevel
- int dumnumflag

- int bracketindexflag
- int parallelflag
- int mparallelflag
- int inparallelflag
- int partodoflag
- int properorderflag
- int vetofilling
- int tablefilling
- int vetotablebasefill
- int exprfillwarning
- int lhdollarflag
- int NoCompress
- int IsFortran90
- int MultiBracketLevels
- int topolynomialflag
- int ffbufnum
- int OldFactArgFlag
- int MemDebugFlag
- int OldGCDflag
- int WTimeStatsFlag
- int doloopstacksize
- int doloopplevel
- int CheckpointFlag
- int SizeCommuteInSet
- int origin
- int vectorlikeLHS
- WORD argsumcheck [MAXNEST]
- WORD insidesumcheck [MAXNEST]
- WORD inexprsumcheck [MAXNEST]
- WORD RepSumCheck [MAXREPEAT]
- WORD lUniTrace [4]
- WORD RepLevel
- WORD arglevel
- WORD insidelevel
- WORD inexprlevel
- WORD termlevel
- WORD MustTestTable
- WORD DumNum
- WORD ncmod
- WORD npowmod
- WORD modmode
- WORD nhalfmod
- WORD DirtPow
- WORD lUnitTrace
- WORD NwildC
- WORD ComDefer
- WORD CollectFun
- WORD AltCollectFun
- WORD OutputMode
- WORD Cnumpows
- WORD OutputSpaces
- WORD OutNumberType
- WORD DidClean
- WORD IfLevel
- WORD WhileLevel

- WORD LogHandle
- WORD LineLength
- WORD StoreHandle
- WORD HideLevel
- WORD lPolyFun
- WORD lPolyFunInv
- WORD lPolyFunType
- WORD lPolyFunExp
- WORD lPolyFunVar
- WORD lPolyFunPow
- WORD SymChangeFlag
- WORD CollectPercentage
- WORD ShortStatsMax
- WORD extrasymbols
- WORD PolyRatFunChanged
- WORD ToBeInFactors
- WORD InnerTest
- UBYTE Commercial [COMMERCIALSIZE+2]
- UBYTE debugFlags [MAXFLAGS+2]

3.7.1 Detailed Description

The C_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name #C. We see it used with the macro AC as in AC.exprnames. It contains variables that involve the compiler and objects set during compilation.

Definition at line 1565 of file structs.h.

3.7.2 Field Documentation

3.7.2.1 separators

```
set_of_char C_const::separators
```

Separators in #call and #do

Definition at line 1566 of file structs.h.

3.7.2.2 StoreFileSize

```
POSITION C_const::StoreFileSize
```

Definition at line 1567 of file structs.h.

3.7.2.3 dollarnames

```
NAMETREE* C_const::dollarnames
```

[D] Names of dollar variables

Definition at line 1568 of file structs.h.

3.7.2.4 exprnames

```
NAMETREE* C_const::exprnames
```

[D] Names of expressions

Definition at line 1569 of file structs.h.

3.7.2.5 varnames

```
NAMETREE* C_const::varnames
```

[D] Names of regular variables

Definition at line 1570 of file structs.h.

3.7.2.6 ChannelList

```
LIST C_const::ChannelList
```

Used for the #write statement. Contains CHANNEL

Definition at line 1571 of file structs.h.

3.7.2.7 DubiousList

```
LIST C_const::DubiousList
```

List of dubious variables. Contains DUBIOUSV. If not empty -> no execution

Definition at line 1573 of file structs.h.

3.7.2.8 FunctionList

```
LIST C_const::FunctionList
```

List of functions and properties. Contains FUNCTIONS

Definition at line 1575 of file structs.h.

3.7.2.9 ExpressionList

```
LIST C_const::ExpressionList
```

List of expressions, locations etc.

Definition at line 1576 of file structs.h.

3.7.2.10 IndexList

```
LIST C_const::IndexList
```

List of indices

Definition at line 1577 of file structs.h.

3.7.2.11 SetElementList

```
LIST C_const::SetElementList
```

List of all elements of all sets

Definition at line 1578 of file structs.h.

3.7.2.12 SetList

```
LIST C_const::SetList
```

List of the sets

Definition at line 1579 of file structs.h.

3.7.2.13 SymbolList

```
LIST C_const::SymbolList
```

List of the symbols and their properties

Definition at line 1580 of file structs.h.

3.7.2.14 VectorList

```
LIST C_const::VectorList
```

List of the vectors

Definition at line 1581 of file structs.h.

3.7.2.15 PotModDolList

```
LIST C_const::PotModDolList
```

Potentially changed dollars

Definition at line 1582 of file structs.h.

3.7.2.16 ModOptDolList

```
LIST C_const::ModOptDolList
```

Module Option Dollars list

Definition at line 1583 of file structs.h.

3.7.2.17 TableBaseList

```
LIST C_const::TableBaseList
```

TableBase list

Definition at line 1584 of file structs.h.

3.7.2.18 cbufList

```
LIST C_const::cbufList
```

List of compiler buffers

Definition at line 1588 of file structs.h.

3.7.2.19 AutoSymbolList

```
LIST C_const::AutoSymbolList
```

Definition at line 1592 of file structs.h.

3.7.2.20 AutoIndexList

```
LIST C_const::AutoIndexList
```

Definition at line 1593 of file structs.h.

3.7.2.21 AutoVectorList

```
LIST C_const::AutoVectorList
```

Definition at line 1594 of file structs.h.

3.7.2.22 AutoFunctionList

```
LIST C_const::AutoFunctionList
```

Definition at line 1595 of file structs.h.

3.7.2.23 autonames

```
NAMETREE* C_const::autonames
```

[D] Names in autodeclare

Definition at line 1596 of file structs.h.

3.7.2.24 Symbols

```
LIST* C_const::Symbols
```

Definition at line 1598 of file structs.h.

3.7.2.25 Indices

```
LIST* C_const::Indices
```

Definition at line 1600 of file structs.h.

3.7.2.26 Vectors

```
LIST* C_const::Vectors
```

Definition at line 1601 of file structs.h.

3.7.2.27 Functions

```
LIST* C_const::Functions
```

Definition at line 1602 of file structs.h.

3.7.2.28 activenames

```
NAMETREE** C_const::activenames
```

Definition at line 1603 of file structs.h.

3.7.2.29 Streams

```
STREAM* C_const::Streams
```

(C) Pointer for AutoDeclare statement. Points either to varnames or autonames. [D] The input streams.

Definition at line 1605 of file structs.h.

3.7.2.30 CurrentStream

```
STREAM* C_const::CurrentStream
```

(C) The current input stream. Streams are: do loop, file, prevariable. points into Streams memory.

Definition at line 1606 of file structs.h.

3.7.2.31 termstack

```
LONG* C_const::termstack
```

[D] Last term statement {offset}

Definition at line 1608 of file structs.h.

3.7.2.32 termsortstack

```
LONG* C_const::termsortstack
```

[D] Last sort statement {offset}

Definition at line 1609 of file structs.h.

3.7.2.33 cmod

```
UWORD* C_const::cmod
```

[D] Local setting of modulus. Pointer to value.

Definition at line 1610 of file structs.h.

3.7.2.34 powmod

```
UWORD* C_const::powmod
```

Local setting printing as powers. Points into cmod memory

Definition at line 1611 of file structs.h.

3.7.2.35 modpowers

```
UWORD* C_const::modpowers
```

[D] The conversion table for mod-> powers.

Definition at line 1612 of file structs.h.

3.7.2.36 halfmod

```
UWORD* C_const::halfmod
```

Definition at line 1613 of file structs.h.

3.7.2.37 ProtoType

```
WORD* C_const::ProtoType
```

Definition at line 1614 of file structs.h.

3.7.2.38 WildC

```
WORD* C_const::WildC
```

Definition at line 1615 of file structs.h.

3.7.2.39 IfHeap

```
LONG* C_const::IfHeap
```

[D] Keeps track of where to go in if

Definition at line 1616 of file structs.h.

3.7.2.40 IfCount

```
LONG* C_const::IfCount
```

[D] Keeps track of where to go in if

Definition at line 1617 of file structs.h.

3.7.2.41 IfStack

```
LONG* C_const::IfStack
```

Keeps track of where to go in if. Points into IfHeap-memory

Definition at line 1618 of file structs.h.

3.7.2.42 iBuffer

```
UBYTE* C_const::iBuffer
```

[D] Compiler input buffer

Definition at line 1619 of file structs.h.

3.7.2.43 iPointer

```
UBYTE* C_const::iPointer
```

Running pointer in the compiler input buffer

Definition at line 1620 of file structs.h.

3.7.2.44 iStop

```
UBYTE* C_const::iStop
```

Top of iBuffer

Definition at line 1621 of file structs.h.

3.7.2.45 LabelNames

```
UBYTE** C_const::LabelNames
```

[D] List of names in label statements

Definition at line 1622 of file structs.h.

3.7.2.46 FixIndices

```
WORD* C_const::FixIndices
```

[D] Buffer of fixed indices

Definition at line 1623 of file structs.h.

3.7.2.47 termsumcheck

```
WORD* C_const::termsumcheck
```

[D] Checking of nesting

Definition at line 1624 of file structs.h.

3.7.2.48 WildcardNames

```
UBYTE* C_const::WildcardNames
```

[D] Names of ?a variables

Definition at line 1625 of file structs.h.

3.7.2.49 Labels

```
int* C_const::Labels
```

Label information for during run. Pointer into LabelNames memory.

Definition at line 1626 of file structs.h.

3.7.2.50 tokens

```
SBYTE* C_const::tokens
```

[D] Array with tokens for tokenizer

Definition at line 1627 of file structs.h.

3.7.2.51 toptokens

```
SBYTE* C_const::toptokens
```

Top of tokens

Definition at line 1628 of file structs.h.

3.7.2.52 endoftokens

```
SBYTE* C_const::endoftokens
```

End of the actual tokens

Definition at line 1629 of file structs.h.

3.7.2.53 tokenarglevel

```
WORD* C_const::tokenarglevel
```

[D] Keeps track of function arguments

Definition at line 1630 of file structs.h.

3.7.2.54 modinverses

```
UWORD* C_const::modinverses
```

Definition at line 1631 of file structs.h.

3.7.2.55 Fortran90Kind

```
UBYTE* C_const::Fortran90Kind
```

Definition at line 1632 of file structs.h.

3.7.2.56 MultiBracketBuf

```
WORD** C_const::MultiBracketBuf
```

Definition at line 1633 of file structs.h.

3.7.2.57 extrasym

```
UBYTE* C_const::extrasym
```

Definition at line 1634 of file structs.h.

3.7.2.58 doloopstack

```
WORD* C_const::doloopstack
```

Definition at line 1635 of file structs.h.

3.7.2.59 doloopnest

```
WORD* C_const::doloopnest
```

Definition at line 1636 of file structs.h.

3.7.2.60 CheckpointRunAfter

```
char* C_const::CheckpointRunAfter
```

[D] Filename of script to be executed *before* creating the snapshot. =0 if no script shall be executed.

Definition at line 1637 of file structs.h.

3.7.2.61 CheckpointRunBefore

```
char* C_const::CheckpointRunBefore
```

[D] Filename of script to be executed *after* having created the snapshot. =0 if no script shall be executed.

Definition at line 1639 of file structs.h.

3.7.2.62 IfSumCheck

```
WORD* C_const::IfSumCheck
```

[D] Keeps track of if-nesting

Definition at line 1641 of file structs.h.

3.7.2.63 CommuteInSet

```
WORD* C_const::CommuteInSet
```

Definition at line 1642 of file structs.h.

3.7.2.64 TestValue

```
UBYTE* C_const::TestValue
```

Definition at line 1643 of file structs.h.

3.7.2.65 argstack

```
LONG C_const::argstack[MAXNEST]
```

Definition at line 1651 of file structs.h.

3.7.2.66 insidestack

```
LONG C_const::insidestack[MAXNEST]
```

Definition at line 1652 of file structs.h.

3.7.2.67 inexprstack

```
LONG C_const::inexprstack[MAXNEST]
```

Definition at line 1653 of file structs.h.

3.7.2.68 iBufferSize

```
LONG C_const::iBufferSize
```

Definition at line 1654 of file structs.h.

3.7.2.69 TransEname

```
LONG C_const::TransEname
```

Definition at line 1655 of file structs.h.

3.7.2.70 ProcessBucketSize

```
LONG C_const::ProcessBucketSize
```

Definition at line 1657 of file structs.h.

3.7.2.71 mProcessBucketSize

```
LONG C_const::mProcessBucketSize
```

Definition at line 1658 of file structs.h.

3.7.2.72 CModule

```
LONG C_const::CModule
```

Definition at line 1659 of file structs.h.

3.7.2.73 ThreadBucketSize

```
LONG C_const::ThreadBucketSize
```

Definition at line 1660 of file structs.h.

3.7.2.74 CheckpointStamp

```
LONG C_const::CheckpointStamp
```

Timestamp of the last created snapshot (set to Timer(0)).

Definition at line 1661 of file structs.h.

3.7.2.75 CheckpointInterval

```
LONG C_const::CheckpointInterval
```

Time interval in milliseconds for snapshots. =0 if snapshots shall be created at the end of *every* module.

Definition at line 1662 of file structs.h.

3.7.2.76 cbufnum

```
int C_const::cbufnum
```

Current compiler buffer

Definition at line 1664 of file structs.h.

3.7.2.77 AutoDeclareFlag

```
int C_const::AutoDeclareFlag
```

Definition at line 1665 of file structs.h.

3.7.2.78 NoShowInput

```
int C_const::NoShowInput
```

(C) Mode of looking for names. Set to NOAUTO (=0) or WITHAUTO (=2), cf. AutoDeclare statement

Definition at line 1667 of file structs.h.

3.7.2.79 ShortStats

```
int C_const::ShortStats
```

Definition at line 1668 of file structs.h.

3.7.2.80 compiletype

```
int C_const::compiletype
```

Definition at line 1669 of file structs.h.

3.7.2.81 firstconstindex

```
int C_const::firstconstindex
```

Definition at line 1670 of file structs.h.

3.7.2.82 insidefirst

```
int C_const::insidefirst
```

Definition at line 1671 of file structs.h.

3.7.2.83 minsidefirst

```
int C_const::minsidefirst
```

Definition at line 1672 of file structs.h.

3.7.2.84 wildflag

```
int C_const::wildflag
```

Definition at line 1673 of file structs.h.

3.7.2.85 NumLabels

```
int C_const::NumLabels
```

Definition at line 1674 of file structs.h.

3.7.2.86 MaxLabels

```
int C_const::MaxLabels
```

Definition at line 1675 of file structs.h.

3.7.2.87 lDefDim

```
int C_const::lDefDim
```

Definition at line 1676 of file structs.h.

3.7.2.88 lDefDim4

```
int C_const::lDefDim4
```

Definition at line 1677 of file structs.h.

3.7.2.89 NumWildcardNames

```
int C_const::NumWildcardNames
```

Definition at line 1678 of file structs.h.

3.7.2.90 WildcardBufferSize

```
int C_const::WildcardBufferSize
```

Definition at line 1679 of file structs.h.

3.7.2.91 MaxIf

```
int C_const::MaxIf
```

Definition at line 1680 of file structs.h.

3.7.2.92 NumStreams

```
int C_const::NumStreams
```

Definition at line 1681 of file structs.h.

3.7.2.93 MaxNumStreams

```
int C_const::MaxNumStreams
```

Definition at line 1682 of file structs.h.

3.7.2.94 firstctypemessage

```
int C_const::firstctypemessage
```

Definition at line 1683 of file structs.h.

3.7.2.95 tablecheck

```
int C_const::tablecheck
```

Definition at line 1684 of file structs.h.

3.7.2.96 idoption

```
int C_const::idoption
```

Definition at line 1685 of file structs.h.

3.7.2.97 BottomLevel

```
int C_const::BottomLevel
```

Definition at line 1686 of file structs.h.

3.7.2.98 CompileLevel

```
int C_const::CompileLevel
```

Definition at line 1687 of file structs.h.

3.7.2.99 TokensWriteFlag

```
int C_const::TokensWriteFlag
```

Definition at line 1688 of file structs.h.

3.7.2.100 UnsureDollarMode

```
int C_const::UnsureDollarMode
```

Definition at line 1689 of file structs.h.

3.7.2.101 outsidefun

```
int C_const::outsidefun
```

Definition at line 1690 of file structs.h.

3.7.2.102 funpowers

```
int C_const::funpowers
```

Definition at line 1691 of file structs.h.

3.7.2.103 WarnFlag

```
int C_const::WarnFlag
```

Definition at line 1692 of file structs.h.

3.7.2.104 StatsFlag

```
int C_const::StatsFlag
```

Definition at line 1693 of file structs.h.

3.7.2.105 NamesFlag

```
int C_const::NamesFlag
```

Definition at line 1694 of file structs.h.

3.7.2.106 CodesFlag

```
int C_const::CodesFlag
```

Definition at line 1695 of file structs.h.

3.7.2.107 SetupFlag

```
int C_const::SetupFlag
```

Definition at line 1696 of file structs.h.

3.7.2.108 SortType

```
int C_const::SortType
```

Definition at line 1697 of file structs.h.

3.7.2.109 lSortType

```
int C_const::lSortType
```

Definition at line 1698 of file structs.h.

3.7.2.110 ThreadStats

```
int C_const::ThreadStats
```

Definition at line 1699 of file structs.h.

3.7.2.111 FinalStats

```
int C_const::FinalStats
```

Definition at line 1700 of file structs.h.

3.7.2.112 OldParallelStats

```
int C_const::OldParallelStats
```

Definition at line 1701 of file structs.h.

3.7.2.113 ThreadsFlag

```
int C_const::ThreadsFlag
```

Definition at line 1702 of file structs.h.

3.7.2.114 ThreadBalancing

```
int C_const::ThreadBalancing
```

Definition at line 1703 of file structs.h.

3.7.2.115 ThreadSortFileSynch

```
int C_const::ThreadSortFileSynch
```

Definition at line 1704 of file structs.h.

3.7.2.116 ProcessStats

```
int C_const::ProcessStats
```

Definition at line 1705 of file structs.h.

3.7.2.117 BracketNormalize

```
int C_const::BracketNormalize
```

Definition at line 1706 of file structs.h.

3.7.2.118 maxtermlevel

```
int C_const::maxtermlevel
```

Definition at line 1707 of file structs.h.

3.7.2.119 dumnumflag

```
int C_const::dumnumflag
```

Definition at line 1708 of file structs.h.

3.7.2.120 bracketindexflag

```
int C_const::bracketindexflag
```

Definition at line 1709 of file structs.h.

3.7.2.121 parallelflag

```
int C_const::parallelflag
```

Definition at line 1710 of file structs.h.

3.7.2.122 mparallelflag

```
int C_const::mparallelflag
```

Definition at line 1711 of file structs.h.

3.7.2.123 inparallelflag

```
int C_const::inparallelflag
```

Definition at line 1712 of file structs.h.

3.7.2.124 partodoflag

```
int C_const::partodoflag
```

Definition at line 1713 of file structs.h.

3.7.2.125 properorderflag

```
int C_const::properorderflag
```

Definition at line 1714 of file structs.h.

3.7.2.126 vetofilling

```
int C_const::vetofilling
```

Definition at line 1715 of file structs.h.

3.7.2.127 tablefilling

```
int C_const::tablefilling
```

Definition at line 1716 of file structs.h.

3.7.2.128 vetotablebasefill

```
int C_const::vetotablebasefill
```

Definition at line 1717 of file structs.h.

3.7.2.129 exprfillwarning

```
int C_const::exprfillwarning
```

Definition at line 1718 of file structs.h.

3.7.2.130 lhdollarflag

```
int C_const::lhdollarflag
```

Definition at line 1719 of file structs.h.

3.7.2.131 NoCompress

```
int C_const::NoCompress
```

Definition at line 1720 of file structs.h.

3.7.2.132 IsFortran90

```
int C_const::IsFortran90
```

Definition at line 1721 of file structs.h.

3.7.2.133 MultiBracketLevels

```
int C_const::MultiBracketLevels
```

Definition at line 1722 of file structs.h.

3.7.2.134 topolynomialflag

```
int C_const::topolynomialflag
```

Definition at line 1723 of file structs.h.

3.7.2.135 ffbufnum

```
int C_const::ffbufnum
```

Definition at line 1724 of file structs.h.

3.7.2.136 OldFactArgFlag

```
int C_const::OldFactArgFlag
```

Definition at line 1725 of file structs.h.

3.7.2.137 MemDebugFlag

```
int C_const::MemDebugFlag
```

Definition at line 1726 of file structs.h.

3.7.2.138 OldGCDflag

```
int C_const::OldGCDflag
```

Definition at line 1727 of file structs.h.

3.7.2.139 WTimeStatsFlag

```
int C_const::WTimeStatsFlag
```

Definition at line 1728 of file structs.h.

3.7.2.140 doloopstacksize

```
int C_const::doloopstacksize
```

Definition at line 1729 of file structs.h.

3.7.2.141 dolooplevel

```
int C_const::dolooplevel
```

Definition at line 1730 of file structs.h.

3.7.2.142 CheckpointFlag

```
int C_const::CheckpointFlag
```

Tells preprocessor whether checkpoint code must executed. -1 : do recovery from snapshot, set by command line option; 0 : do nothing; 1 : create snapshots, set by On checkpoint statement

Definition at line 1731 of file structs.h.

3.7.2.143 SizeCommuteInSet

```
int C_const::SizeCommuteInSet
```

Definition at line 1735 of file structs.h.

3.7.2.144 origin

```
int C_const::origin
```

Definition at line 1740 of file structs.h.

3.7.2.145 vectorlikeLHS

```
int C_const::vectorlikeLHS
```

Definition at line 1741 of file structs.h.

3.7.2.146 argsumcheck

```
WORD C_const::argsumcheck[MAXNEST]
```

Definition at line 1742 of file structs.h.

3.7.2.147 insidesumcheck

```
WORD C_const::insidesumcheck[MAXNEST]
```

Definition at line 1743 of file structs.h.

3.7.2.148 inexprsumcheck

```
WORD C_const::inexprsumcheck[MAXNEST]
```

Definition at line 1744 of file structs.h.

3.7.2.149 RepSumCheck

```
WORD C_const::RepSumCheck[MAXREPEAT]
```

Definition at line 1745 of file structs.h.

3.7.2.150 lUniTrace

```
WORD C_const::lUniTrace[4]
```

Definition at line 1746 of file structs.h.

3.7.2.151 RepLevel

```
WORD C_const::RepLevel
```

Definition at line 1747 of file structs.h.

3.7.2.152 arglevel

```
WORD C_const::arglevel
```

Definition at line 1748 of file structs.h.

3.7.2.153 insidelevel

```
WORD C_const::insidelevel
```

Definition at line 1749 of file structs.h.

3.7.2.154 inexprlevel

```
WORD C_const::inexprlevel
```

Definition at line 1750 of file structs.h.

3.7.2.155 termlevel

```
WORD C_const::termlevel
```

Definition at line 1751 of file structs.h.

3.7.2.156 MustTestTable

```
WORD C_const::MustTestTable
```

Definition at line 1752 of file structs.h.

3.7.2.157 DumNum

```
WORD C_const::DumNum
```

Definition at line 1753 of file structs.h.

3.7.2.158 ncmod

```
WORD C_const::ncmod
```

Definition at line 1754 of file structs.h.

3.7.2.159 npowmod

```
WORD C_const::npowmod
```

Definition at line 1755 of file structs.h.

3.7.2.160 modmode

```
WORD C_const::modmode
```

Definition at line 1756 of file structs.h.

3.7.2.161 nhalfmod

```
WORD C_const::nhalfmod
```

Definition at line 1757 of file structs.h.

3.7.2.162 DirtPow

```
WORD C_const::DirtPow
```

Definition at line 1758 of file structs.h.

3.7.2.163 lUnitTrace

```
WORD C_const::lUnitTrace
```

Definition at line 1759 of file structs.h.

3.7.2.164 NwildC

```
WORD C_const::NwildC
```

Definition at line 1760 of file structs.h.

3.7.2.165 ComDefer

```
WORD C_const::ComDefer
```

Definition at line 1761 of file structs.h.

3.7.2.166 CollectFun

```
WORD C_const::CollectFun
```

Definition at line 1762 of file structs.h.

3.7.2.167 AltCollectFun

```
WORD C_const::AltCollectFun
```

Definition at line 1763 of file structs.h.

3.7.2.168 OutputMode

```
WORD C_const::OutputMode
```

Definition at line 1764 of file structs.h.

3.7.2.169 Cnumpows

```
WORD C_const::Cnumpows
```

Definition at line 1765 of file structs.h.

3.7.2.170 OutputSpaces

```
WORD C_const::OutputSpaces
```

Definition at line 1766 of file structs.h.

3.7.2.171 OutNumberType

```
WORD C_const::OutNumberType
```

Definition at line 1767 of file structs.h.

3.7.2.172 DidClean

```
WORD C_const::DidClean
```

Definition at line 1768 of file structs.h.

3.7.2.173 IfLevel

```
WORD C_const::IfLevel
```

Definition at line 1769 of file structs.h.

3.7.2.174 WhileLevel

```
WORD C_const::WhileLevel
```

Definition at line 1770 of file structs.h.

3.7.2.175 LogHandle

```
WORD C_const::LogHandle
```

Definition at line 1771 of file structs.h.

3.7.2.176 LineLength

```
WORD C_const::LineLength
```

Definition at line 1772 of file structs.h.

3.7.2.177 StoreHandle

```
WORD C_const::StoreHandle
```

Definition at line 1773 of file structs.h.

3.7.2.178 HideLevel

```
WORD C_const::HideLevel
```

Definition at line 1774 of file structs.h.

3.7.2.179 lPolyFun

```
WORD C_const::lPolyFun
```

Definition at line 1775 of file structs.h.

3.7.2.180 lPolyFunInv

```
WORD C_const::lPolyFunInv
```

Definition at line 1776 of file structs.h.

3.7.2.181 lPolyFunType

```
WORD C_const::lPolyFunType
```

Definition at line 1777 of file structs.h.

3.7.2.182 lPolyFunExp

```
WORD C_const::lPolyFunExp
```

Definition at line 1778 of file structs.h.

3.7.2.183 lPolyFunVar

```
WORD C_const::lPolyFunVar
```

Definition at line 1779 of file structs.h.

3.7.2.184 lPolyFunPow

```
WORD C_const::lPolyFunPow
```

Definition at line 1780 of file structs.h.

3.7.2.185 SymChangeFlag

```
WORD C_const::SymChangeFlag
```

Definition at line 1781 of file structs.h.

3.7.2.186 CollectPercentage

```
WORD C_const::CollectPercentage
```

Definition at line 1782 of file structs.h.

3.7.2.187 ShortStatsMax

```
WORD C_const::ShortStatsMax
```

Definition at line 1783 of file structs.h.

3.7.2.188 extrasymbols

```
WORD C_const::extrasymbols
```

Definition at line 1784 of file structs.h.

3.7.2.189 PolyRatFunChanged

```
WORD C_const::PolyRatFunChanged
```

Definition at line 1785 of file structs.h.

3.7.2.190 ToBeInFactors

```
WORD C_const::ToBeInFactors
```

Definition at line 1786 of file structs.h.

3.7.2.191 InnerTest

```
WORD C_const::InnerTest
```

Definition at line 1787 of file structs.h.

3.7.2.192 Commercial

```
UBYTE C_const::Commercial[COMMERCIALSIZE+2]
```

Definition at line 1791 of file structs.h.

3.7.2.193 debugFlags

```
UBYTE C_const::debugFlags[MAXFLAGS+2]
```

Definition at line 1792 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.8 CbUf Struct Reference

```
#include <structs.h>
```

Collaboration diagram for CbUf:

Data Fields

- WORD * Buffer
- WORD * Top
- WORD * Pointer
- WORD ** lhs
- WORD ** rhs
- LONG * CanCommu
- LONG * NumTerms
- WORD * numdum
- WORD * dimension
- COMPTREE * boomlijst
- LONG BufferSize
- int numlhs
- int numrhs
- int maxlhs
- int maxrhs
- int mnumlhs
- int mnumrhs
- int numtree
- int rootnum
- int MaxTreeSize

3.8.1 Detailed Description

The CBUF struct is used by the compiler. It is a compiler buffer of which since version 3.0 there can be many.

Definition at line 921 of file structs.h.

3.8.2 Field Documentation

3.8.2.1 Buffer

```
WORD* CbUf::Buffer
```

[D] Size in BufferSize

Definition at line 922 of file structs.h.

Referenced by CleanupArgCache(), clearcbuf(), DoubleCbuffer(), finishcbuf(), inicbufs(), PF_BroadcastCBuf(), and Processor().

3.8.2.2 Top

```
WORD* CbUf::Top
```

pointer to the end of the Buffer memory

Definition at line 923 of file structs.h.

Referenced by AddNtoC(), AddNtoL(), DoubleCbuffer(), finishcbuf(), inicbufs(), PF_BroadcastCBuf(), and TestSub().

3.8.2.3 Pointer

```
WORD* CbUf::Pointer
```

pointer into the Buffer memory

Definition at line 924 of file structs.h.

Referenced by AddLHS(), AddNtoC(), AddNtoL(), AddRHS(), CleanupArgCache(), clearcbuf(), DoubleCbuffer(), finishcbuf(), Generator(), inicbufs(), PF_BroadcastCBuf(), Processor(), and TestSub().

3.8.2.4 lhs

```
WORD** CbUf::lhs
```

[D] Size in maxlhs. list of pointers into Buffer.

Definition at line 925 of file structs.h.

Referenced by AddLHS(), DoubleCbuffer(), finishcbuf(), Generator(), inicbufs(), PF_BroadcastCBuf(), Processor(), and TestMatch().

3.8.2.5 rhs

```
WORD** CbUf::rhs
```

[D] Size in maxrhs. list of pointers into Buffer.

Definition at line 926 of file structs.h.

Referenced by AddRHS(), CleanupArgCache(), clearcbuf(), DoubleCbuffer(), finishcbuf(), Generator(), inicbufs(), IniFbuffer(), PF_BroadcastCBuf(), Processor(), and TestSub().

3.8.2.6 CanCommu

```
LONG* CbUf::CanCommu
```

points into rhs memory behind WORD∗ area.

Definition at line 927 of file structs.h.

Referenced by AddRHS(), finishcbuf(), inicbufs(), IniFbuffer(), PF_BroadcastCBuf(), and TestSub().

3.8.2.7 NumTerms

```
LONG* CbUf::NumTerms
```

points into rhs memory behind CanCommu area

Definition at line 928 of file structs.h.

Referenced by AddRHS(), finishcbuf(), inicbufs(), IniFbuffer(), and PF_BroadcastCBuf().

3.8.2.8 numdum

```
WORD* CbUf::numdum
```

points into rhs memory behind NumTerms

Definition at line 929 of file structs.h.

Referenced by AddRHS(), inicbufs(), IniFbuffer(), and PF_BroadcastCBuf().

3.8.2.9 dimension

```
WORD* CbUf::dimension
```

points into rhs memory behind numdum

Definition at line 930 of file structs.h.

Referenced by AddRHS(), inicbufs(), IniFbuffer(), and PF_BroadcastCBuf().

3.8.2.10 boomlijst

```
COMPTREE* CbUf::boomlijst
```

[D] Number elements in MaxTreeSize

Definition at line 931 of file structs.h.

Referenced by CleanupArgCache(), clearcbuf(), finishcbuf(), inicbufs(), IniFbuffer(), and PF_BroadcastCBuf().

3.8.2.11 BufferSize

```
LONG CbUf::BufferSize
```

Number of allocated WORD's in Buffer

Definition at line 932 of file structs.h.

Referenced by DoubleCbuffer(), finishcbuf(), inicbufs(), and PF_BroadcastCBuf().

3.8.2.12 numlhs

```
int CbUf::numlhs
```

Definition at line 933 of file structs.h.

3.8.2.13 numrhs

```
int CbUf::numrhs
```

Definition at line 934 of file structs.h.

3.8.2.14 maxlhs

```
int CbUf::maxlhs
```

Definition at line 935 of file structs.h.

3.8.2.15 maxrhs

```
int CbUf::maxrhs
```

Definition at line 936 of file structs.h.

3.8.2.16 mnumlhs

```
int CbUf::mnumlhs
```

Definition at line 937 of file structs.h.

3.8.2.17 mnumrhs

```
int CbUf::mnumrhs
```

Definition at line 938 of file structs.h.

3.8.2.18 numtree

```
int CbUf::numtree
```

Definition at line 939 of file structs.h.

3.8.2.19 rootnum

```
int CbUf::rootnum
```

Definition at line 940 of file structs.h.

3.8.2.20 MaxTreeSize

```
int CbUf::MaxTreeSize
```

Definition at line 941 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.9 ChAnNeL Struct Reference

```
#include <structs.h>
```

Data Fields

- char * name
- int handle

3.9.1 Detailed Description

When we read input from text files we have to remember not only their handle but also their name. This is needed for error messages. Hence we call such a file a channel and reserve a struct of type CHANNEL to allow to lay this link.

Definition at line 952 of file structs.h.

3.9.2 Field Documentation

3.9.2.1 name

```
char* ChAnNeL::name
```

[D] Name of the channel

Definition at line 953 of file structs.h.

3.9.2.2 handle

```
int ChAnNeL::handle
```

File handle

Definition at line 954 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.10 COST Struct Reference

Data Fields

- LONG add
- LONG mul
- LONG div
- LONG pow

3.10.1 Detailed Description

Definition at line 1224 of file structs.h.

3.10.2 Field Documentation

3.10.2.1 add

```
LONG COST::add
```

Definition at line 1225 of file structs.h.

3.10.2.2 mul

```
LONG COST::mul
```

Definition at line 1226 of file structs.h.

3.10.2.3 div

```
LONG COST::div
```

Definition at line 1227 of file structs.h.

3.10.2.4 pow

```
LONG COST::pow
```

Definition at line 1228 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.11 dbase Struct Reference

Collaboration diagram for dbase:

Data Fields

- INIINFO info
- MLONG mode
- MLONG tablenamessize
- MLONG topnumber
- MLONG tablenamefill
- INDEXBLOCK ** iblocks
- NAMESBLOCK ** nblocks
- FILE * handle
- char * name
- char * fullname
- char * tablenames

3.11.1 Detailed Description

Definition at line 120 of file minos.h.

3.11.2 Field Documentation

3.11.2.1 info

```
INIINFO dbase::info
```

Definition at line 121 of file minos.h.

3.11.2.2 mode

```
MLONG dbase::mode
```

Definition at line 122 of file minos.h.

3.11.2.3 tablenamessize

```
MLONG dbase::tablenamessize
```

Definition at line 123 of file minos.h.

3.11.2.4 topnumber

```
MLONG dbase::topnumber
```

Definition at line 124 of file minos.h.

3.11.2.5 tablenamefill

```
MLONG dbase::tablenamefill
```

Definition at line 125 of file minos.h.

3.11.2.6 iblocks

```
INDEXBLOCK** dbase::iblocks
```

Definition at line 126 of file minos.h.

3.11.2.7 nblocks

```
NAMESBLOCK** dbase::nblocks
```

Definition at line 127 of file minos.h.

3.11.2.8 handle

```
FILE* dbase::handle
```

Definition at line 128 of file minos.h.

3.11.2.9 name

```
char* dbase::name
```

Definition at line 129 of file minos.h.

3.11.2.10 fullname

```
char* dbase::fullname
```

Definition at line 130 of file minos.h.

3.11.2.11 tablenames

```
char* dbase::tablenames
```

Definition at line 131 of file minos.h.

The documentation for this struct was generated from the following file:

- minos.h

3.12 DICTIONARY Struct Reference

Collaboration diagram for DICTIONARY:

Data Fields

- DICTIONARY_ELEMENT ** elements
- UBYTE * name
- int sizeelements
- int numelements
- int numbers
- int variables
- int characters
- int funwith
- int gnumelements
- int ranges

3.12.1 Detailed Description

Definition at line 1290 of file structs.h.

3.12.2 Field Documentation

3.12.2.1 elements

```
DICTIONARY_ELEMENT** DICTIONARY::elements
```

Definition at line 1291 of file structs.h.

3.12.2.2 name

```
UBYTE* DICTIONARY::name
```

Definition at line 1292 of file structs.h.

3.12.2.3 sizeelements

```
int DICTIONARY::sizeelements
```

Definition at line 1293 of file structs.h.

3.12.2.4 numelements

```
int DICTIONARY::numelements
```

Definition at line 1294 of file structs.h.

3.12.2.5 numbers

```
int DICTIONARY::numbers
```

Definition at line 1295 of file structs.h.

3.12.2.6 variables

```
int DICTIONARY::variables
```

Definition at line 1296 of file structs.h.

3.12.2.7 characters

```
int DICTIONARY::characters
```

Definition at line 1297 of file structs.h.

3.12.2.8 funwith

```
int DICTIONARY::funwith
```

Definition at line 1298 of file structs.h.

3.12.2.9 gnumelements

```
int DICTIONARY::gnumelements
```

Definition at line 1299 of file structs.h.

3.12.2.10 ranges

```
int DICTIONARY::ranges
```

Definition at line 1300 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.13 DICTIONARY_ELEMENT Struct Reference

Data Fields

- WORD * lhs
- WORD * rhs
- int type
- int size

3.13.1 Detailed Description

Definition at line 1283 of file structs.h.

3.13.2 Field Documentation

3.13.2.1 lhs

```
WORD* DICTIONARY_ELEMENT::lhs
```

Definition at line 1284 of file structs.h.

3.13.2.2 rhs

```
WORD* DICTIONARY_ELEMENT::rhs
```

Definition at line 1285 of file structs.h.

3.13.2.3 type

```
int DICTIONARY_ELEMENT::type
```

Definition at line 1286 of file structs.h.

3.13.2.4 size

```
int DICTIONARY_ELEMENT::size
```

Definition at line 1287 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.14 DiStRiBuTe Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * obj1
- WORD * obj2
- WORD * out
- WORD sign
- WORD n1
- WORD n2
- WORD n
- WORD cycle [MAXMATCH]

3.14.1 Detailed Description

The struct DISTRIBUTE is used to help the pattern matcher when matching antisymmetric tensors.

Definition at line 1032 of file structs.h.

3.14.2 Field Documentation

3.14.2.1 obj1

```
WORD* DiStRiBuTe::obj1
```

Definition at line 1033 of file structs.h.

3.14.2.2 obj2

```
WORD* DiStRiBuTe::obj2
```

Definition at line 1034 of file structs.h.

3.14.2.3 out

```
WORD* DiStRiBuTe::out
```

Definition at line 1035 of file structs.h.

3.14.2.4 sign

```
WORD DiStRiBuTe::sign
```

Definition at line 1036 of file structs.h.

3.14.2.5 n1

```
WORD DiStRiBuTe::n1
```

Definition at line 1037 of file structs.h.

3.14.2.6 n2

```
WORD DiStRiBuTe::n2
```

Definition at line 1038 of file structs.h.

3.14.2.7 n

```
WORD DiStRiBuTe::n
```

Definition at line 1039 of file structs.h.

3.14.2.8 cycle

```
WORD DiStRiBuTe::cycle[MAXMATCH]
```

Definition at line 1040 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.15 dollar_buf Struct Reference

3.15.1 Detailed Description

Definition at line 2468 of file parallel.c.

The documentation for this struct was generated from the following file:

- parallel.c

3.16 DoLlArS Struct Reference

Collaboration diagram for DoLlArS:

Data Fields

- WORD * where
- FACDOLLAR * factors
- LONG size
- LONG name
- WORD type
- WORD node
- WORD index
- WORD zero
- WORD numdummies
- WORD nfactors

3.16.1 Detailed Description

Definition at line 515 of file structs.h.

3.16.2 Field Documentation

3.16.2.1 where

```
WORD* DoLlArS::where
```

Definition at line 516 of file structs.h.

3.16.2.2 factors

```
FACDOLLAR* DoLlArS::factors
```

Definition at line 517 of file structs.h.

3.16.2.3 size

```
LONG DoLlArS::size
```

Definition at line 522 of file structs.h.

3.16.2.4 name

```
LONG DoLlArS::name
```

Definition at line 523 of file structs.h.

3.16.2.5 type

```
WORD DoLlArS::type
```

Definition at line 524 of file structs.h.

3.16.2.6 node

```
WORD DoLlArS::node
```

Definition at line 525 of file structs.h.

3.16.2.7 index

```
WORD DoLlArS::index
```

Definition at line 526 of file structs.h.

3.16.2.8 zero

```
WORD DoLlArS::zero
```

Definition at line 527 of file structs.h.

3.16.2.9 numdummies

```
WORD DoLlArS::numdummies
```

Definition at line 528 of file structs.h.

3.16.2.10 nfactors

```
WORD DoLlArS::nfactors
```

Definition at line 529 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.17 DoLoOp Struct Reference

```
#include <structs.h>
```

Collaboration diagram for DoLoOp:

Data Fields

- PRELOAD p
- UBYTE * name
- UBYTE * vars
- UBYTE * contents
- UBYTE * dollarname
- LONG startlinenumber
- LONG firstnum
- LONG lastnum
- LONG incnum
- int type
- int NoShowInput
- int errorsinloop
- int firstloopcall
- WORD firstdollar
- WORD lastdollar
- WORD incdollar
- WORD NumPreTypes
- WORD PreIfLevel
- WORD PreSwitchLevel

3.17.1 Detailed Description

Each preprocessor do loop has a struct of type DOLOOP to keep track of all relevant parameters like where the beginning of the loop is, what the boundaries, increment and value of the loop parameter are, etc. Also we keep the whole loop inside a buffer of type PRELOAD

Definition at line 835 of file structs.h.

3.17.2 Field Documentation

3.17.2.1 p

```
PRELOAD DoLoOp::p
```

size, name and buffer

Definition at line 836 of file structs.h.

3.17.2.2 name

```
UBYTE* DoLoOp::name
```

pointer into PRELOAD buffer

Definition at line 837 of file structs.h.

3.17.2.3 vars

```
UBYTE* DoLoOp::vars
```

Definition at line 838 of file structs.h.

3.17.2.4 contents

```
UBYTE* DoLoOp::contents
```

Definition at line 839 of file structs.h.

3.17.2.5 dollarname

```
UBYTE* DoLoOp::dollarname
```

For loop over terms in expression. Allocated with Malloc1()

Definition at line 840 of file structs.h.

3.17.2.6 startlinenumber

```
LONG DoLoOp::startlinenumber
```

Definition at line 841 of file structs.h.

3.17.2.7 firstnum

```
LONG DoLoOp::firstnum
```

Definition at line 842 of file structs.h.

3.17.2.8 lastnum

```
LONG DoLoOp::lastnum
```

Definition at line 843 of file structs.h.

3.17.2.9 incnum

```
LONG DoLoOp::incnum
```

Definition at line 844 of file structs.h.

3.17.2.10 type

```
int DoLoOp::type
```

Definition at line 845 of file structs.h.

3.17.2.11 NoShowInput

```
int DoLoOp::NoShowInput
```

Definition at line 846 of file structs.h.

3.17.2.12 errorsinloop

```
int DoLoOp::errorsinloop
```

Definition at line 847 of file structs.h.

3.17.2.13 firstloopcall

```
int DoLoOp::firstloopcall
```

Definition at line 848 of file structs.h.

3.17.2.14 firstdollar

```
WORD DoLoOp::firstdollar
```

Definition at line 849 of file structs.h.

3.17.2.15 lastdollar

```
WORD DoLoOp::lastdollar
```

Definition at line 850 of file structs.h.

3.17.2.16 incdollar

```
WORD DoLoOp::incdollar
```

Definition at line 851 of file structs.h.

3.17.2.17 NumPreTypes

```
WORD DoLoOp::NumPreTypes
```

Definition at line 852 of file structs.h.

3.17.2.18 PreIfLevel

```
WORD DoLoOp::PreIfLevel
```

Definition at line 853 of file structs.h.

3.17.2.19 PreSwitchLevel

```
WORD DoLoOp::PreSwitchLevel
```

Definition at line 854 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.18 DuBiOuS Struct Reference

Data Fields

- LONG name
- WORD node
- WORD dummy

3.18.1 Detailed Description

Definition at line 500 of file structs.h.

3.18.2 Field Documentation

3.18.2.1 name

```
LONG DuBiOuS::name
```

Definition at line 501 of file structs.h.

3.18.2.2 node

```
WORD DuBiOuS::node
```

Definition at line 502 of file structs.h.

3.18.2.3 dummy

```
WORD DuBiOuS::dummy
```

Definition at line 503 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.19 ExPrEsSiOn Struct Reference

Collaboration diagram for ExPrEsSiOn:

Public Member Functions

- **PADPOSITION** (5, 2, 0, 12, 0)

Data Fields

- POSITION onfile
- POSITION prototype
- POSITION size
- RENUMBER renum
- BRACKETINFO * bracketinfo
- BRACKETINFO * newbracketinfo
- WORD * renumlists
- WORD * inmem
- LONG counter
- LONG name
- WORD hidelevel
- WORD vflags
- WORD printflag

- WORD status
- WORD replace
- WORD node
- WORD whichbuffer
- WORD namesize
- WORD compression
- WORD numdummies
- WORD numfactors
- WORD sizeprototype

3.19.1 Detailed Description

Definition at line 378 of file structs.h.

3.19.2 Field Documentation

3.19.2.1 onfile

```
POSITION ExPrEsSiOn::onfile
```

Definition at line 379 of file structs.h.

3.19.2.2 prototype

```
POSITION ExPrEsSiOn::prototype
```

Definition at line 380 of file structs.h.

3.19.2.3 size

```
POSITION ExPrEsSiOn::size
```

Definition at line 381 of file structs.h.

3.19.2.4 renum

```
RENUMBER ExPrEsSiOn::renum
```

Definition at line 382 of file structs.h.

3.19.2.5 bracketinfo

```
BRACKETINFO* ExPrEsSiOn::bracketinfo
```

Definition at line 383 of file structs.h.

3.19.2.6 newbracketinfo

```
BRACKETINFO* ExPrEsSiOn::newbracketinfo
```

Definition at line 384 of file structs.h.

3.19.2.7 renumlists

```
WORD* ExPrEsSiOn::renumlists
```

Allocated only for threaded version if variables exist, else points to AN.dummyrenumlist

Definition at line 385 of file structs.h.

Referenced by DoRecovery().

3.19.2.8 inmem

```
WORD* ExPrEsSiOn::inmem
```

Definition at line 387 of file structs.h.

3.19.2.9 counter

```
LONG ExPrEsSiOn::counter
```

Definition at line 388 of file structs.h.

3.19.2.10 name

```
LONG ExPrEsSiOn::name
```

Definition at line 389 of file structs.h.

3.19.2.11 hidelevel

```
WORD ExPrEsSiOn::hidelevel
```

Definition at line 390 of file structs.h.

3.19.2.12 vflags

```
WORD ExPrEsSiOn::vflags
```

Definition at line 391 of file structs.h.

3.19.2.13 printflag

```
WORD ExPrEsSiOn::printflag
```

Definition at line 392 of file structs.h.

3.19.2.14 status

```
WORD ExPrEsSiOn::status
```

Definition at line 393 of file structs.h.

3.19.2.15 replace

```
WORD ExPrEsSiOn::replace
```

Definition at line 394 of file structs.h.

3.19.2.16 node

```
WORD ExPrEsSiOn::node
```

Definition at line 395 of file structs.h.

3.19.2.17 whichbuffer

```
WORD ExPrEsSiOn::whichbuffer
```

Definition at line 396 of file structs.h.

3.19.2.18 namesize

```
WORD ExPrEsSiOn::namesize
```

Definition at line 397 of file structs.h.

3.19.2.19 compression

```
WORD ExPrEsSiOn::compression
```

Definition at line 398 of file structs.h.

3.19.2.20 numdummies

```
WORD ExPrEsSiOn::numdummies
```

Definition at line 399 of file structs.h.

3.19.2.21 numfactors

```
WORD ExPrEsSiOn::numfactors
```

Definition at line 400 of file structs.h.

3.19.2.22 sizeprototype

```
WORD ExPrEsSiOn::sizeprototype
```

Definition at line 401 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.20 FaCdOlLaR Struct Reference

Data Fields

- WORD * where
- LONG size
- WORD type
- WORD value

3.20.1 Detailed Description

Definition at line 507 of file structs.h.

3.20.2 Field Documentation

3.20.2.1 where

```
WORD* FaCdOlLaR::where
```

Definition at line 508 of file structs.h.

3.20.2.2 size

```
LONG FaCdOlLaR::size
```

Definition at line 509 of file structs.h.

3.20.2.3 type

```
WORD FaCdOlLaR::type
```

Definition at line 510 of file structs.h.

3.20.2.4 value

```
WORD FaCdOlLaR::value
```

Definition at line 511 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.21 factorized_poly Class Reference

Public Member Functions

- void **add_factor** (const poly &f, int p=1)
- const std::string **tostring** () const

Data Fields

- std::vector< poly > factor
- std::vector< int > power

Friends

- std::ostream & **operator**<< (std::ostream &out, const poly &p)

3.21.1 Detailed Description

Definition at line 54 of file polyfact.h.

3.21.2 Field Documentation

3.21.2.1 factor

```
std::vector<poly> factorized_poly::factor
```

Definition at line 59 of file polyfact.h.

3.21.2.2 power

```
std::vector<int> factorized_poly::power
```

Definition at line 60 of file polyfact.h.

The documentation for this class was generated from the following file:

- polyfact.h

3.22 FiLe Struct Reference

```
#include <structs.h>
```

Collaboration diagram for FiLe:

Public Member Functions

- **PADPOSITION** (5, 3, 2, 0, 0)

Data Fields

- POSITION POposition
- POSITION filesize
- WORD * PObuffer
- WORD * POstop
- WORD * POfill
- WORD * POfull
- char * name
- ULONG numblocks
- ULONG inbuffer
- LONG POsize
- int handle
- int active

3.22.1 Detailed Description

The type FILEHANDLE is the struct that controls all relevant information of a file, whether it is open or not. The file may even not yet exist. There is a system of caches (PObuffer) and as long as the information to be written still fits inside the cache the file may never be created. There are variables that can store information about different types of files, like scratch files or sort files. Depending on what is available in the system we may also have information about gzip compression (currently sort file only) or locks (TFORM).

Definition at line 620 of file structs.h.

3.22.2 Field Documentation

3.22.2.1 POposition

```
POSITION FiLe::POposition
```

Definition at line 621 of file structs.h.

3.22.2.2 filesize

```
POSITION FiLe::filesize
```

Definition at line 622 of file structs.h.

3.22.2.3 PObuffer

```
WORD* FiLe::PObuffer
```

Definition at line 623 of file structs.h.

3.22.2.4 POstop

```
WORD* FiLe::POstop
```

Definition at line 624 of file structs.h.

3.22.2.5 POfill

```
WORD* FiLe::POfill
```

Definition at line 625 of file structs.h.

3.22.2.6 POfull

```
WORD* FiLe::POfull
```

Definition at line 626 of file structs.h.

3.22.2.7 name

```
char* FiLe::name
```

Definition at line 633 of file structs.h.

3.22.2.8 numblocks

```
ULONG FiLe::numblocks
```

Definition at line 638 of file structs.h.

3.22.2.9 inbuffer

```
ULONG FiLe::inbuffer
```

Definition at line 639 of file structs.h.

3.22.2.10 POsize

```
LONG FiLe::POsize
```

Definition at line 640 of file structs.h.

3.22.2.11 handle

```
int FiLe::handle
```

Our own handle. Equal -1 if no file exists.

Definition at line 648 of file structs.h.

Referenced by DoOnePow(), FlushOut(), MergePatches(), PutIn(), PutOut(), Sflush(), and StageSort().

3.22.2.12 active

```
int FiLe::active
```

Definition at line 649 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.23 FiLeDaTa Struct Reference

Collaboration diagram for FiLeDaTa:

Public Member Functions

- **PADPOSITION** (0, 0, 0, 2, 0)

Data Fields

- FILEINDEX Index
- POSITION Fill
- POSITION Position
- WORD Handle
- WORD dirtyflag

3.23.1 Detailed Description

Definition at line 150 of file structs.h.

3.23.2 Field Documentation

3.23.2.1 Index

```
FILEINDEX FiLeDaTa::Index
```

Definition at line 151 of file structs.h.

3.23.2.2 Fill

```
POSITION FiLeDaTa::Fill
```

Definition at line 152 of file structs.h.

3.23.2.3 Position

```
POSITION FiLeDaTa::Position
```

Definition at line 153 of file structs.h.

3.23.2.4 Handle

```
WORD FiLeDaTa::Handle
```

Definition at line 154 of file structs.h.

3.23.2.5 dirtyflag

```
WORD FiLeDaTa::dirtyflag
```

Definition at line 155 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.24 FiLeInDeX Struct Reference

```
#include <structs.h>
```

Collaboration diagram for FiLeInDeX:

Data Fields

- POSITION next
- POSITION number
- INDEXENTRY expression [INFILEINDEX]
- SBYTE empty [EMPTYININDEX]

3.24.1 Detailed Description

Defines the structure of a file index in store-files and save-files.

It contains several entries (see struct InDeXeNtRy) up to a maximum of INFILEINDEX.

The variable number has been made of type POSITION to avoid padding problems with some types of computers/↩ OS and keep system independence of the .sav files.

This struct is always 512 bytes long.

Definition at line 137 of file structs.h.

3.24.2 Field Documentation

3.24.2.1 next

```
POSITION FiLeInDeX::next
```

Position of next FILEINDEX if any

Definition at line 138 of file structs.h.

Referenced by ReadSaveIndex().

3.24.2.2 number

```
POSITION FiLeInDeX::number
```

Number of used entries in this index

Definition at line 139 of file structs.h.

Referenced by ReadSaveIndex().

3.24.2.3 expression

```
INDEXENTRY FiLeInDeX::expression[INFILEINDEX]
```

File index entries

Definition at line 140 of file structs.h.

3.24.2.4 empty

```
SBYTE FiLeInDeX::empty[EMPTYININDEX]
```

Padding to 512 bytes

Definition at line 141 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.25 FixedGlobals Struct Reference

```
#include <structs.h>
```

Data Fields

- WCN Operation [8]
- WCN2 OperaFind [6]
- char * VarType [10]
- char * ExprStat [21]
- char * FunNam [2]
- char * swmes [3]
- char * fname
- char * fname2
- UBYTE * s_one
- WORD fnamebase
- WORD fname2base
- UINT cTable [256]

3.25.1 Detailed Description

The FIXEDGLOBALS struct is an anachronism. It started as the struct with global variables that needed initialization. It contains the elements Operation and OperaFind which define a very early way of automatically jumping to the proper operation. We find the results of it in parts of the file opera.c Later operations were treated differently in a more transparent way. We never changed the existing code. The most important part is currently the cTable which is used intensively in the compiler.

Definition at line 2438 of file structs.h.

3.25.2 Field Documentation

3.25.2.1 Operation

```
WCN FixedGlobals::Operation[8]
```

Definition at line 2439 of file structs.h.

3.25.2.2 OperaFind

```
WCN2 FixedGlobals::OperaFind[6]
```

Definition at line 2440 of file structs.h.

3.25.2.3 VarType

```
char* FixedGlobals::VarType[10]
```

Definition at line 2441 of file structs.h.

3.25.2.4 ExprStat

```
char* FixedGlobals::ExprStat[21]
```

Definition at line 2442 of file structs.h.

3.25.2.5 FunNam

```
char* FixedGlobals::FunNam[2]
```

Definition at line 2443 of file structs.h.

3.25.2.6 swmes

```
char* FixedGlobals::swmes[3]
```

Definition at line 2444 of file structs.h.

3.25.2.7 fname

```
char* FixedGlobals::fname
```

Definition at line 2445 of file structs.h.

3.25.2.8 fname2

```
char* FixedGlobals::fname2
```

Definition at line 2446 of file structs.h.

3.25.2.9 s_one

```
UBYTE* FixedGlobals::s_one
```

Definition at line 2447 of file structs.h.

3.25.2.10 fnamebase

```
WORD FixedGlobals::fnamebase
```

Definition at line 2448 of file structs.h.

3.25.2.11 fname2base

```
WORD FixedGlobals::fname2base
```

Definition at line 2449 of file structs.h.

3.25.2.12 cTable

```
UINT FixedGlobals::cTable[256]
```

Definition at line 2450 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.26 fixedset Struct Reference

Data Fields

- char * name
- char * description
- int type
- int dimension

3.26.1 Detailed Description

Definition at line 556 of file structs.h.

3.26.2 Field Documentation

3.26.2.1 name

```
char* fixedset::name
```

Definition at line 557 of file structs.h.

3.26.2.2 description

```
char* fixedset::description
```

Definition at line 558 of file structs.h.

3.26.2.3 type

```
int fixedset::type
```

Definition at line 559 of file structs.h.

3.26.2.4 dimension

```
int fixedset::dimension
```

Definition at line 560 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.27 FUN_INFO Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * location
- int numargs
- int numfunnies
- int numwildcards
- int symmet
- int tensor
- int commute

3.27.1 Detailed Description

The struct FUN_INFO is used for information about functions in the file smart.c which is supposed to intelligently look for patterns in complicated wildcard situations involving symmetric functions.

Definition at line 593 of file structs.h.

3.27.2 Field Documentation

3.27.2.1 location

```
WORD* FUN_INFO::location
```

Definition at line 594 of file structs.h.

3.27.2.2 numargs

```
int FUN_INFO::numargs
```

Definition at line 595 of file structs.h.

3.27.2.3 numfunnies

```
int FUN_INFO::numfunnies
```

Definition at line 596 of file structs.h.

3.27.2.4 numwildcards

```
int FUN_INFO::numwildcards
```

Definition at line 597 of file structs.h.

3.27.2.5 symmet

```
int FUN_INFO::symmet
```

Definition at line 598 of file structs.h.

3.27.2.6 tensor

```
int FUN_INFO::tensor
```

Definition at line 599 of file structs.h.

3.27.2.7 commute

```
int FUN_INFO::commute
```

Definition at line 600 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.28 FuNcTiOn Struct Reference

```
#include <structs.h>
```

Collaboration diagram for FuNcTiOn:

Data Fields

- TABLES tabl
- LONG symminfo
- LONG name
- WORD commute
- WORD complex
- WORD number
- WORD flags
- WORD spec
- WORD symmetric
- WORD node
- WORD namesize
- WORD dimension
- WORD maxnumargs
- WORD minnumargs

3.28.1 Detailed Description

Contains all information about a function. Also used for tables. It is used in the LIST elements of AC.

Definition at line 463 of file structs.h.

3.28.2 Field Documentation

3.28.2.1 tabl

`TABLES FuNcTiOn::tabl`

Used if redefined as table. != 0 if function is a table

Definition at line 464 of file structs.h.

3.28.2.2 symminfo

`LONG FuNcTiOn::symminfo`

Info regarding symm properties offset in buffer

Definition at line 465 of file structs.h.

3.28.2.3 name

`LONG FuNcTiOn::name`

Location in namebuffer of NAMETREE

Definition at line 466 of file structs.h.

Referenced by StartVariables().

3.28.2.4 commute

`WORD FuNcTiOn::commute`

Commutation properties

Definition at line 467 of file structs.h.

3.28.2.5 complex

`WORD FuNcTiOn::complex`

Properties under complex conjugation

Definition at line 468 of file structs.h.

3.28.2.6 number

```
WORD FuNcTiOn::number
```

Number when stored in file

Definition at line 469 of file structs.h.

Referenced by ReadSaveIndex().

3.28.2.7 flags

```
WORD FuNcTiOn::flags
```

Used to indicate usage when storing

Definition at line 470 of file structs.h.

3.28.2.8 spec

```
WORD FuNcTiOn::spec
```

Regular, Tensor, etc. See FunSpecs.

Definition at line 471 of file structs.h.

Referenced by ReadSaveVariables().

3.28.2.9 symmetric

```
WORD FuNcTiOn::symmetric
```

0 if symmetric properties

Definition at line 472 of file structs.h.

Referenced by StartVariables().

3.28.2.10 node

```
WORD FuNcTiOn::node
```

Location in namenode of NAMETREE

Definition at line 473 of file structs.h.

3.28.2.11 namesize

```
WORD FuNcTiOn::namesize
```

Length of the name

Definition at line 474 of file structs.h.

3.28.2.12 dimension

```
WORD FuNcTiOn::dimension
```

Definition at line 475 of file structs.h.

3.28.2.13 maxnumargs

```
WORD FuNcTiOn::maxnumargs
```

Definition at line 476 of file structs.h.

3.28.2.14 minnumargs

```
WORD FuNcTiOn::minnumargs
```

Definition at line 477 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.29 gcd_heuristic_failed Class Reference

3.29.1 Detailed Description

Definition at line 33 of file polygcd.h.

The documentation for this class was generated from the following file:

- polygcd.h

3.30 HANDLERS Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD newlogonly
- WORD newhandle
- WORD oldhandle
- WORD oldlogonly
- WORD oldprinttype
- WORD oldsilent

3.30.1 Detailed Description

The struct HANDLERS is used in the communication with external channels.

Definition at line 902 of file structs.h.

3.30.2 Field Documentation

3.30.2.1 newlogonly

```
WORD HANDLERS::newlogonly
```

Definition at line 903 of file structs.h.

3.30.2.2 newhandle

```
WORD HANDLERS::newhandle
```

Definition at line 904 of file structs.h.

3.30.2.3 oldhandle

```
WORD HANDLERS::oldhandle
```

Definition at line 905 of file structs.h.

3.30.2.4 oldlogonly

```
WORD HANDLERS::oldlogonly
```

Definition at line 906 of file structs.h.

3.30.2.5 oldprinttype

```
WORD HANDLERS::oldprinttype
```

Definition at line 907 of file structs.h.

3.30.2.6 oldsilent

```
WORD HANDLERS::oldsilent
```

Definition at line 908 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.31 InDeX Struct Reference

Data Fields

- LONG name
- WORD type
- WORD dimension
- WORD number
- WORD flags
- WORD nmin4
- WORD node
- WORD namesize

3.31.1 Detailed Description

Definition at line 431 of file structs.h.

3.31.2 Field Documentation

3.31.2.1 name

```
LONG InDeX::name
```

Definition at line 432 of file structs.h.

3.31.2.2 type

```
WORD InDeX::type
```

Definition at line 433 of file structs.h.

3.31.2.3 dimension

```
WORD InDeX::dimension
```

Definition at line 434 of file structs.h.

3.31.2.4 number

```
WORD InDeX::number
```

Definition at line 435 of file structs.h.

3.31.2.5 flags

```
WORD InDeX::flags
```

Definition at line 436 of file structs.h.

3.31.2.6 nmin4

```
WORD InDeX::nmin4
```

Definition at line 437 of file structs.h.

3.31.2.7 node

```
WORD InDeX::node
```

Definition at line 438 of file structs.h.

3.31.2.8 namesize

```
WORD InDeX::namesize
```

Definition at line 439 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.32 indexblock Struct Reference

Collaboration diagram for indexblock:

Data Fields

- MLONG flags
- MLONG previousblock
- MLONG position
- OBJECTS objects [NUMOBJECTS]

3.32.1 Detailed Description

Definition at line 107 of file minos.h.

3.32.2 Field Documentation

3.32.2.1 flags

```
MLONG indexblock::flags
```

Definition at line 108 of file minos.h.

3.32.2.2 previousblock

```
MLONG indexblock::previousblock
```

Definition at line 109 of file minos.h.

3.32.2.3 position

```
MLONG indexblock::position
```

Definition at line 110 of file minos.h.

3.32.2.4 objects

```
OBJECTS indexblock::objects[NUMOBJECTS]
```

Definition at line 111 of file minos.h.

The documentation for this struct was generated from the following file:

- minos.h

3.33 InDeXeNtRy Struct Reference

```
#include <structs.h>
```

Collaboration diagram for InDeXeNtRy:

Public Member Functions

- **PADPOSITION** (0, 1, 0, 5, MAXENAME+1)

Data Fields

- POSITION position
- POSITION length
- POSITION variables
- LONG CompressSize
- WORD nsymbols
- WORD nindices
- WORD nvectors
- WORD nfunctions
- WORD size
- SBYTE name [MAXENAME+1]

3.33.1 Detailed Description

Defines the structure of an entry in a file index (see struct FiLeInDeX).

It represents one expression in the file.

Definition at line 99 of file structs.h.

3.33.2 Field Documentation

3.33.2.1 position

```
POSITION InDeXeNtRy::position
```

Position of the expression itself

Definition at line 100 of file structs.h.

3.33.2.2 length

```
POSITION InDeXeNtRy::length
```

Length of the expression itself

Definition at line 101 of file structs.h.

3.33.2.3 variables

```
POSITION InDeXeNtRy::variables
```

Position of the list with variables

Definition at line 102 of file structs.h.

3.33.2.4 CompressSize

```
LONG InDeXeNtRy::CompressSize
```

Size of buffer before compress

Definition at line 103 of file structs.h.

3.33.2.5 nsymbols

```
WORD InDeXeNtRy::nsymbols
```

Number of symbols in the list

Definition at line 104 of file structs.h.

Referenced by ReadSaveVariables().

3.33.2.6 nindices

```
WORD InDeXeNtRy::nindices
```

Number of indices in the list

Definition at line 105 of file structs.h.

Referenced by ReadSaveVariables().

3.33.2.7 nvectors

```
WORD InDeXeNtRy::nvectors
```

Number of vectors in the list

Definition at line 106 of file structs.h.

Referenced by ReadSaveVariables().

3.33.2.8 nfunctions

```
WORD InDeXeNtRy::nfunctions
```

Number of functions in the list

Definition at line 107 of file structs.h.

Referenced by ReadSaveVariables().

3.33.2.9 size

```
WORD InDeXeNtRy::size
```

Size of variables field

Definition at line 108 of file structs.h.

3.33.2.10 name

```
SBYTE InDeXeNtRy::name[MAXENAME+1]
```

Name of expression

Definition at line 109 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.34 iniinfo Struct Reference

Data Fields

- MLONG entriesinindex
- MLONG numberofindexblocks
- MLONG firstindexblock
- MLONG lastindexblock
- MLONG numberoftables
- MLONG numberofnamesblocks
- MLONG firstnameblock
- MLONG lastnameblock

3.34.1 Detailed Description

Definition at line 82 of file minos.h.

3.34.2 Field Documentation

3.34.2.1 entriesinindex

```
MLONG iniinfo::entriesinindex
```

Definition at line 84 of file minos.h.

3.34.2.2 numberofindexblocks

```
MLONG iniinfo::numberofindexblocks
```

Definition at line 85 of file minos.h.

3.34.2.3 firstindexblock

```
MLONG iniinfo::firstindexblock
```

Definition at line 86 of file minos.h.

3.34.2.4 lastindexblock

```
MLONG iniinfo::lastindexblock
```

Definition at line 87 of file minos.h.

3.34.2.5 numberoftables

```
MLONG iniinfo::numberoftables
```

Definition at line 88 of file minos.h.

3.34.2.6 numberofnamesblocks

```
MLONG iniinfo::numberofnamesblocks
```

Definition at line 89 of file minos.h.

3.34.2.7 firstnameblock

```
MLONG iniinfo::firstnameblock
```

Definition at line 90 of file minos.h.

3.34.2.8 lastnameblock

```
MLONG iniinfo::lastnameblock
```

Definition at line 91 of file minos.h.

The documentation for this struct was generated from the following file:

- minos.h

3.35 INSIDEINFO Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * buffer
- int oldcompiletype
- int oldparallelflag
- int oldnumpotmoddollars
- WORD size
- WORD numdollars
- WORD oldcbuf
- WORD oldrbuf
- WORD inscbuf
- WORD oldcnumlhs

3.35.1 Detailed Description

Used for #inside

Definition at line 792 of file structs.h.

3.35.2 Field Documentation

3.35.2.1 buffer

```
WORD* INSIDEINFO::buffer
```

Definition at line 793 of file structs.h.

3.35.2.2 oldcompiletype

```
int INSIDEINFO::oldcompiletype
```

Definition at line 794 of file structs.h.

3.35.2.3 oldparallelflag

```
int INSIDEINFO::oldparallelflag
```

Definition at line 795 of file structs.h.

3.35.2.4 oldnumpotmoddollars

```
int INSIDEINFO::oldnumpotmoddollars
```

Definition at line 796 of file structs.h.

3.35.2.5 size

```
WORD INSIDEINFO::size
```

Definition at line 797 of file structs.h.

3.35.2.6 numdollars

```
WORD INSIDEINFO::numdollars
```

Definition at line 798 of file structs.h.

3.35.2.7 oldcbuf

```
WORD INSIDEINFO::oldcbuf
```

Definition at line 799 of file structs.h.

3.35.2.8 oldrbuf

```
WORD INSIDEINFO::oldrbuf
```

Definition at line 800 of file structs.h.

3.35.2.9 inscbuf

```
WORD INSIDEINFO::inscbuf
```

Definition at line 801 of file structs.h.

3.35.2.10 oldcnumlhs

```
WORD INSIDEINFO::oldcnumlhs
```

Definition at line 802 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.36 KEYWORD Struct Reference

```
#include <structs.h>
```

Data Fields

- char * name
- TFUN func
- int type
- int flags

3.36.1 Detailed Description

The KEYWORD struct defines names of commands/statements and the routine to be called when they are encountered by the compiler or preprocessor.

Definition at line 221 of file structs.h.

3.36.2 Field Documentation

3.36.2.1 name

```
char* KEYWORD::name
```

Definition at line 222 of file structs.h.

3.36.2.2 func

```
TFUN KEYWORD::func
```

Definition at line 223 of file structs.h.

3.36.2.3 type

```
int KEYWORD::type
```

Definition at line 224 of file structs.h.

3.36.2.4 flags

```
int KEYWORD::flags
```

Definition at line 225 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.37 LIST Struct Reference

```
#include <structs.h>
```

Data Fields

- void * lijst
- char * message
- int num
- int maxnum
- int size
- int numglobal
- int numtemp
- int numclear

3.37.1 Detailed Description

Much information is stored in arrays of which we can double the size if the array proves to be too small. Such arrays are controled by a variable of type LIST. The routines that expand the lists are in the file tools.c

Definition at line 204 of file structs.h.

3.37.2 Field Documentation

3.37.2.1 lijst

```
void* LIST::lijst
```

[D] Holds space for "maxnum" elements of size "size" each

Definition at line 205 of file structs.h.

3.37.2.2 message

```
char* LIST::message
```

Text for Malloc1 when allocating lijst. Set to constant string.

Definition at line 206 of file structs.h.

3.37.2.3 num

```
int LIST::num
```

Number of elements in lijst.

Definition at line 207 of file structs.h.

3.37.2.4 maxnum

```
int LIST::maxnum
```

Maximum number of elements in lijst.

Definition at line 208 of file structs.h.

3.37.2.5 size

```
int LIST::size
```

Size of one element in lijst.

Definition at line 209 of file structs.h.

3.37.2.6 numglobal

```
int LIST::numglobal
```

Marker for position when .global is executed.

Definition at line 210 of file structs.h.

3.37.2.7 numtemp

```
int LIST::numtemp
```

At the moment only needed for sets and setstore.

Definition at line 211 of file structs.h.

3.37.2.8 numclear

```
int LIST::numclear
```

Only for the clear instruction.

Definition at line 212 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.38 longMultiStruct Struct Reference

Collaboration diagram for longMultiStruct:

Data Fields

- UBYTE * buffer
- int bufpos
- int packpos
- int nPacks
- int lastLen
- struct longMultiStruct * next

3.38.1 Detailed Description

Definition at line 1013 of file mpi.c.

3.38.2 Field Documentation

3.38.2.1 buffer

```
UBYTE* longMultiStruct::buffer
```

Definition at line 1014 of file mpi.c.

3.38.2.2 bufpos

```
int longMultiStruct::bufpos
```

Definition at line 1015 of file mpi.c.

3.38.2.3 packpos

```
int longMultiStruct::packpos
```

Definition at line 1016 of file mpi.c.

3.38.2.4 nPacks

```
int longMultiStruct::nPacks
```

Definition at line 1017 of file mpi.c.

3.38.2.5 lastLen

```
int longMultiStruct::lastLen
```

Definition at line 1018 of file mpi.c.

3.38.2.6 next

```
struct longMultiStruct* longMultiStruct::next
```

Definition at line 1021 of file mpi.c.

The documentation for this struct was generated from the following file:

- mpi.c

3.39 M_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for M_const:

Public Member Functions

- **PADPOSITION** (17, 23, 61, 81, 0)

Data Fields

- POSITION zeropos
- SORTING * S0
- UWORD * gcmod
- UWORD * gpowmod
- UBYTE * TempDir
- UBYTE * TempSortDir
- UBYTE * IncDir
- UBYTE * InputFileName
- UBYTE * LogFileName
- UBYTE * OutBuffer

- UBYTE * Path
- UBYTE * SetupDir
- UBYTE * SetupFile
- UBYTE * gFortran90Kind
- UBYTE * gextrasym
- UBYTE * ggextrasym
- UBYTE * oldnumextrasymbols
- SPECTATOR * SpectatorFiles
- LONG MaxTer
- LONG CompressSize
- LONG ScratSize
- LONG HideSize
- LONG SizeStoreCache
- LONG MaxStreamSize
- LONG SIOsize
- LONG SLargeSize
- LONG SSmallEsize
- LONG SSmallSize
- LONG STermsInSmall
- LONG MaxBracketBufferSize
- LONG hProcessBucketSize
- LONG gProcessBucketSize
- LONG shmWinSize
- LONG OldChildTime
- LONG OldSecTime
- LONG OldMilliTime
- LONG WorkSize
- LONG gThreadBucketSize
- LONG ggThreadBucketSize
- LONG SumTime
- LONG SpectatorSize
- int FileOnlyFlag
- int Interact
- int MaxParLevel
- int OutBufSize
- int SMaxFpatches
- int SMaxPatches
- int StdOut
- int ginsidefirst
- int gDefDim
- int gDefDim4
- int NumFixedSets
- int NumFixedFunctions
- int rbufnum
- int dbufnum
- int sbufnum
- int zbufnum
- int SkipClears
- int gTokensWriteFlag
- int gfunpowers
- int gStatsFlag
- int gNamesFlag
- int gCodesFlag
- int gSortType
- int gproperorderflag

- int hparallelflag
- int gparallelflag
- int totalnumberofthreads
- int gSizeCommuteInSet
- int gThreadStats
- int ggThreadStats
- int gFinalStats
- int ggFinalStats
- int gThreadsFlag
- int ggThreadsFlag
- int gThreadBalancing
- int ggThreadBalancing
- int gThreadSortFileSynch
- int ggThreadSortFileSynch
- int gProcessStats
- int ggProcessStats
- int gOldParallelStats
- int ggOldParallelStats
- int maxFlevels
- int resetTimeOnClear
- int gcNumDollars
- int MultiRun
- int gNoSpacesInNumbers
- int ggNoSpacesInNumbers
- int gIsFortran90
- int PrintTotalSize
- int fbuffersize
- int gOldFactArgFlag
- int ggOldFactArgFlag
- int gnumextrasym
- int ggnumextrasym
- int NumSpectatorFiles
- int SizeForSpectatorFiles
- int gOldGCDflag
- int ggOldGCDflag
- int gWTimeStatsFlag
- int ggWTimeStatsFlag
- WORD MaxTal
- WORD IndDum
- WORD DumInd
- WORD WilInd
- WORD gncmod
- WORD gnpowmod
- WORD gmodmode
- WORD gUnitTrace
- WORD gOutputMode
- WORD gOutputSpaces
- WORD gOutNumberType
- WORD gCnumpows
- WORD gUniTrace [4]
- WORD MaxWildcards
- WORD mTraceDum
- WORD OffsetIndex
- WORD OffsetVector
- WORD RepMax

- WORD LogType
- WORD ggStatsFlag
- WORD gLineLength
- WORD qError
- WORD FortranCont
- WORD HoldFlag
- WORD Ordering [15]
- WORD silent
- WORD tracebackflag
- WORD expnum
- WORD denomnum
- WORD facnum
- WORD invfacnum
- WORD sumnum
- WORD sumpnum
- WORD OldOrderFlag
- WORD termfunnum
- WORD matchfunnum
- WORD countfunnum
- WORD gPolyFun
- WORD gPolyFunInv
- WORD gPolyFunType
- WORD gPolyFunExp
- WORD gPolyFunVar
- WORD gPolyFunPow
- WORD dollarzero
- WORD atstartup
- WORD exitflag
- WORD NumStoreCaches
- WORD gIndentSpace
- WORD ggIndentSpace
- WORD gShortStatsMax
- WORD ggShortStatsMax
- WORD gextrasymbols
- WORD ggextrasymbols
- WORD zerorhs
- WORD onerhs
- WORD havesortdir
- WORD BracketFactors [8]

3.39.1 Detailed Description

The M_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name #M. We see it used with the macro AM as in AM.S0. It contains global settings at startup or .clear.

Definition at line 1316 of file structs.h.

3.39.2 Field Documentation

3.39.2.1 zeropos

```
POSITION M_const::zeropos
```

Definition at line 1317 of file structs.h.

3.39.2.2 S0

```
SORTING* M_const::S0
```

[D] The main sort buffer

Definition at line 1318 of file structs.h.

3.39.2.3 gcmod

```
UWORD* M_const::gcmod
```

Global setting of modulus. Uses AC.cmod's memory

Definition at line 1319 of file structs.h.

3.39.2.4 gpowmod

```
UWORD* M_const::gpowmod
```

Global setting printing as powers. Uses AC.cmod's memory

Definition at line 1320 of file structs.h.

3.39.2.5 TempDir

```
UBYTE* M_const::TempDir
```

Definition at line 1321 of file structs.h.

3.39.2.6 TempSortDir

```
UBYTE* M_const::TempSortDir
```

Definition at line 1322 of file structs.h.

3.39.2.7 IncDir

```
UBYTE* M_const::IncDir
```

Definition at line 1323 of file structs.h.

3.39.2.8 InputFileName

```
UBYTE* M_const::InputFileName
```

Definition at line 1324 of file structs.h.

3.39.2.9 LogFileName

```
UBYTE* M_const::LogFileName
```

Definition at line 1325 of file structs.h.

3.39.2.10 OutBuffer

```
UBYTE* M_const::OutBuffer
```

Definition at line 1326 of file structs.h.

3.39.2.11 Path

```
UBYTE* M_const::Path
```

Definition at line 1327 of file structs.h.

3.39.2.12 SetupDir

```
UBYTE* M_const::SetupDir
```

Definition at line 1328 of file structs.h.

3.39.2.13 SetupFile

```
UBYTE* M_const::SetupFile
```

Definition at line 1329 of file structs.h.

3.39.2.14 gFortran90Kind

```
UBYTE* M_const::gFortran90Kind
```

Definition at line 1330 of file structs.h.

3.39.2.15 gextrasym

```
UBYTE* M_const::gextrasym
```

Definition at line 1331 of file structs.h.

3.39.2.16 ggextrasym

```
UBYTE* M_const::ggextrasym
```

Definition at line 1332 of file structs.h.

3.39.2.17 oldnumextrasymbols

```
UBYTE* M_const::oldnumextrasymbols
```

Definition at line 1333 of file structs.h.

3.39.2.18 SpectatorFiles

```
SPECTATOR* M_const::SpectatorFiles
```

Definition at line 1334 of file structs.h.

3.39.2.19 MaxTer

```
LONG M_const::MaxTer
```

Definition at line 1342 of file structs.h.

3.39.2.20 CompressSize

```
LONG M_const::CompressSize
```

Definition at line 1343 of file structs.h.

3.39.2.21 ScratSize

```
LONG M_const::ScratSize
```

Definition at line 1344 of file structs.h.

3.39.2.22 HideSize

```
LONG M_const::HideSize
```

Definition at line 1345 of file structs.h.

3.39.2.23 SizeStoreCache

```
LONG M_const::SizeStoreCache
```

Definition at line 1346 of file structs.h.

3.39.2.24 MaxStreamSize

```
LONG M_const::MaxStreamSize
```

Definition at line 1347 of file structs.h.

3.39.2.25 SIOsize

```
LONG M_const::SIOsize
```

Definition at line 1348 of file structs.h.

3.39.2.26 SLargeSize

```
LONG M_const::SLargeSize
```

Definition at line 1349 of file structs.h.

3.39.2.27 SSmallEsize

```
LONG M_const::SSmallEsize
```

Definition at line 1350 of file structs.h.

3.39.2.28 SSmallSize

```
LONG M_const::SSmallSize
```

Definition at line 1351 of file structs.h.

3.39.2.29 STermsInSmall

```
LONG M_const::STermsInSmall
```

Definition at line 1352 of file structs.h.

3.39.2.30 MaxBracketBufferSize

```
LONG M_const::MaxBracketBufferSize
```

Definition at line 1353 of file structs.h.

3.39.2.31 hProcessBucketSize

```
LONG M_const::hProcessBucketSize
```

Definition at line 1354 of file structs.h.

3.39.2.32 gProcessBucketSize

```
LONG M_const::gProcessBucketSize
```

Definition at line 1355 of file structs.h.

3.39.2.33 shmWinSize

```
LONG M_const::shmWinSize
```

Definition at line 1356 of file structs.h.

3.39.2.34 OldChildTime

```
LONG M_const::OldChildTime
```

Definition at line 1357 of file structs.h.

3.39.2.35 OldSecTime

```
LONG M_const::OldSecTime
```

Definition at line 1358 of file structs.h.

3.39.2.36 OldMilliTime

```
LONG M_const::OldMilliTime
```

Definition at line 1359 of file structs.h.

3.39.2.37 WorkSize

```
LONG M_const::WorkSize
```

Definition at line 1360 of file structs.h.

3.39.2.38 gThreadBucketSize

```
LONG M_const::gThreadBucketSize
```

Definition at line 1361 of file structs.h.

3.39.2.39 ggThreadBucketSize

```
LONG M_const::ggThreadBucketSize
```

Definition at line 1362 of file structs.h.

3.39.2.40 SumTime

```
LONG M_const::SumTime
```

Definition at line 1363 of file structs.h.

3.39.2.41 SpectatorSize

```
LONG M_const::SpectatorSize
```

Definition at line 1364 of file structs.h.

3.39.2.42 FileOnlyFlag

```
int M_const::FileOnlyFlag
```

Definition at line 1365 of file structs.h.

3.39.2.43 Interact

```
int M_const::Interact
```

Definition at line 1366 of file structs.h.

3.39.2.44 MaxParLevel

```
int M_const::MaxParLevel
```

Definition at line 1367 of file structs.h.

3.39.2.45 OutBufSize

```
int M_const::OutBufSize
```

Definition at line 1368 of file structs.h.

3.39.2.46 SMaxFpatches

```
int M_const::SMaxFpatches
```

Definition at line 1369 of file structs.h.

3.39.2.47 SMaxPatches

```
int M_const::SMaxPatches
```

Definition at line 1370 of file structs.h.

3.39.2.48 StdOut

```
int M_const::StdOut
```

Definition at line 1371 of file structs.h.

3.39.2.49 ginsidefirst

```
int M_const::ginsidefirst
```

Definition at line 1372 of file structs.h.

3.39.2.50 gDefDim

```
int M_const::gDefDim
```

Definition at line 1373 of file structs.h.

3.39.2.51 gDefDim4

```
int M_const::gDefDim4
```

Definition at line 1374 of file structs.h.

3.39.2.52 NumFixedSets

```
int M_const::NumFixedSets
```

Definition at line 1375 of file structs.h.

3.39.2.53 NumFixedFunctions

```
int M_const::NumFixedFunctions
```

Definition at line 1376 of file structs.h.

3.39.2.54 rbufnum

```
int M_const::rbufnum
```

Definition at line 1377 of file structs.h.

3.39.2.55 dbufnum

```
int M_const::dbufnum
```

Definition at line 1378 of file structs.h.

3.39.2.56 sbufnum

```
int M_const::sbufnum
```

Definition at line 1379 of file structs.h.

3.39.2.57 zbufnum

```
int M_const::zbufnum
```

Definition at line 1380 of file structs.h.

3.39.2.58 SkipClears

```
int M_const::SkipClears
```

Definition at line 1381 of file structs.h.

3.39.2.59 gTokensWriteFlag

```
int M_const::gTokensWriteFlag
```

Definition at line 1382 of file structs.h.

3.39.2.60 gfunpowers

```
int M_const::gfunpowers
```

Definition at line 1383 of file structs.h.

3.39.2.61 gStatsFlag

```
int M_const::gStatsFlag
```

Definition at line 1384 of file structs.h.

3.39.2.62 gNamesFlag

```
int M_const::gNamesFlag
```

Definition at line 1385 of file structs.h.

3.39.2.63 gCodesFlag

```
int M_const::gCodesFlag
```

Definition at line 1386 of file structs.h.

3.39.2.64 gSortType

```
int M_const::gSortType
```

Definition at line 1387 of file structs.h.

3.39.2.65 gproperorderflag

```
int M_const::gproperorderflag
```

Definition at line 1388 of file structs.h.

3.39.2.66 hparallelflag

```
int M_const::hparallelflag
```

Definition at line 1389 of file structs.h.

3.39.2.67 gparallelflag

```
int M_const::gparallelflag
```

Definition at line 1390 of file structs.h.

3.39.2.68 totalnumberofthreads

```
int M_const::totalnumberofthreads
```

Definition at line 1391 of file structs.h.

3.39.2.69 gSizeCommuteInSet

```
int M_const::gSizeCommuteInSet
```

Definition at line 1392 of file structs.h.

3.39.2.70 gThreadStats

```
int M_const::gThreadStats
```

Definition at line 1393 of file structs.h.

3.39.2.71 ggThreadStats

```
int M_const::ggThreadStats
```

Definition at line 1394 of file structs.h.

3.39.2.72 gFinalStats

```
int M_const::gFinalStats
```

Definition at line 1395 of file structs.h.

3.39.2.73 ggFinalStats

```
int M_const::ggFinalStats
```

Definition at line 1396 of file structs.h.

3.39.2.74 gThreadsFlag

```
int M_const::gThreadsFlag
```

Definition at line 1397 of file structs.h.

3.39.2.75 ggThreadsFlag

```
int M_const::ggThreadsFlag
```

Definition at line 1398 of file structs.h.

3.39.2.76 gThreadBalancing

```
int M_const::gThreadBalancing
```

Definition at line 1399 of file structs.h.

3.39.2.77 ggThreadBalancing

```
int M_const::ggThreadBalancing
```

Definition at line 1400 of file structs.h.

3.39.2.78 gThreadSortFileSynch

```
int M_const::gThreadSortFileSynch
```

Definition at line 1401 of file structs.h.

3.39.2.79 ggThreadSortFileSynch

```
int M_const::ggThreadSortFileSynch
```

Definition at line 1402 of file structs.h.

3.39.2.80 gProcessStats

```
int M_const::gProcessStats
```

Definition at line 1403 of file structs.h.

3.39.2.81 ggProcessStats

```
int M_const::ggProcessStats
```

Definition at line 1404 of file structs.h.

3.39.2.82 gOldParallelStats

```
int M_const::gOldParallelStats
```

Definition at line 1405 of file structs.h.

3.39.2.83 ggOldParallelStats

```
int M_const::ggOldParallelStats
```

Definition at line 1406 of file structs.h.

3.39.2.84 maxFlevels

```
int M_const::maxFlevels
```

Definition at line 1407 of file structs.h.

3.39.2.85 resetTimeOnClear

```
int M_const::resetTimeOnClear
```

Definition at line 1408 of file structs.h.

3.39.2.86 gcNumDollars

```
int M_const::gcNumDollars
```

Definition at line 1409 of file structs.h.

3.39.2.87 MultiRun

```
int M_const::MultiRun
```

Definition at line 1410 of file structs.h.

3.39.2.88 gNoSpacesInNumbers

```
int M_const::gNoSpacesInNumbers
```

Definition at line 1411 of file structs.h.

3.39.2.89 ggNoSpacesInNumbers

```
int M_const::ggNoSpacesInNumbers
```

Definition at line 1412 of file structs.h.

3.39.2.90 gIsFortran90

```
int M_const::gIsFortran90
```

Definition at line 1413 of file structs.h.

3.39.2.91 PrintTotalSize

```
int M_const::PrintTotalSize
```

Definition at line 1414 of file structs.h.

3.39.2.92 fbuffersize

```
int M_const::fbuffersize
```

Definition at line 1415 of file structs.h.

3.39.2.93 gOldFactArgFlag

```
int M_const::gOldFactArgFlag
```

Definition at line 1416 of file structs.h.

3.39.2.94 ggOldFactArgFlag

```
int M_const::ggOldFactArgFlag
```

Definition at line 1417 of file structs.h.

3.39.2.95 gnumextrasym

```
int M_const::gnumextrasym
```

Definition at line 1418 of file structs.h.

3.39.2.96 ggnumextrasym

```
int M_const::ggnumextrasym
```

Definition at line 1419 of file structs.h.

3.39.2.97 NumSpectatorFiles

```
int M_const::NumSpectatorFiles
```

Definition at line 1420 of file structs.h.

3.39.2.98 SizeForSpectatorFiles

```
int M_const::SizeForSpectatorFiles
```

Definition at line 1421 of file structs.h.

3.39.2.99 gOldGCDflag

```
int M_const::gOldGCDflag
```

Definition at line 1422 of file structs.h.

3.39.2.100 ggOldGCDflag

```
int M_const::ggOldGCDflag
```

Definition at line 1423 of file structs.h.

3.39.2.101 gWTimeStatsFlag

```
int M_const::gWTimeStatsFlag
```

Definition at line 1424 of file structs.h.

3.39.2.102 ggWTimeStatsFlag

```
int M_const::ggWTimeStatsFlag
```

Definition at line 1425 of file structs.h.

3.39.2.103 MaxTal

```
WORD M_const::MaxTal
```

Definition at line 1426 of file structs.h.

3.39.2.104 IndDum

```
WORD M_const::IndDum
```

Definition at line 1427 of file structs.h.

3.39.2.105 DumInd

```
WORD M_const::DumInd
```

Definition at line 1428 of file structs.h.

3.39.2.106 WilInd

```
WORD M_const::WilInd
```

Definition at line 1429 of file structs.h.

3.39.2.107 gncmod

```
WORD M_const::gncmod
```

Definition at line 1430 of file structs.h.

3.39.2.108 gnpowmod

```
WORD M_const::gnpowmod
```

Definition at line 1431 of file structs.h.

3.39.2.109 gmodmode

```
WORD M_const::gmodmode
```

Definition at line 1432 of file structs.h.

3.39.2.110 gUnitTrace

```
WORD M_const::gUnitTrace
```

Definition at line 1433 of file structs.h.

3.39.2.111 gOutputMode

```
WORD M_const::gOutputMode
```

Definition at line 1434 of file structs.h.

3.39.2.112 gOutputSpaces

```
WORD M_const::gOutputSpaces
```

Definition at line 1435 of file structs.h.

3.39.2.113 gOutNumberType

```
WORD M_const::gOutNumberType
```

Definition at line 1436 of file structs.h.

3.39.2.114 gCnumpows

```
WORD M_const::gCnumpows
```

Definition at line 1437 of file structs.h.

3.39.2.115 gUniTrace

```
WORD M_const::gUniTrace[4]
```

Definition at line 1438 of file structs.h.

3.39.2.116 MaxWildcards

```
WORD M_const::MaxWildcards
```

Definition at line 1439 of file structs.h.

3.39.2.117 mTraceDum

```
WORD M_const::mTraceDum
```

Definition at line 1440 of file structs.h.

3.39.2.118 OffsetIndex

```
WORD M_const::OffsetIndex
```

Definition at line 1441 of file structs.h.

3.39.2.119 OffsetVector

```
WORD M_const::OffsetVector
```

Definition at line 1442 of file structs.h.

3.39.2.120 RepMax

```
WORD M_const::RepMax
```

Definition at line 1443 of file structs.h.

3.39.2.121 LogType

```
WORD M_const::LogType
```

Definition at line 1444 of file structs.h.

3.39.2.122 ggStatsFlag

```
WORD M_const::ggStatsFlag
```

Definition at line 1445 of file structs.h.

3.39.2.123 gLineLength

```
WORD M_const::gLineLength
```

Definition at line 1446 of file structs.h.

3.39.2.124 qError

```
WORD M_const::qError
```

Definition at line 1447 of file structs.h.

3.39.2.125 FortranCont

```
WORD M_const::FortranCont
```

Definition at line 1448 of file structs.h.

3.39.2.126 HoldFlag

```
WORD M_const::HoldFlag
```

Definition at line 1449 of file structs.h.

3.39.2.127 Ordering

```
WORD M_const::Ordering[15]
```

Definition at line 1450 of file structs.h.

3.39.2.128 silent

```
WORD M_const::silent
```

Definition at line 1451 of file structs.h.

3.39.2.129 tracebackflag

```
WORD M_const::tracebackflag
```

Definition at line 1452 of file structs.h.

3.39.2.130 expnum

```
WORD M_const::expnum
```

Definition at line 1453 of file structs.h.

3.39.2.131 denomnum

```
WORD M_const::denomnum
```

Definition at line 1454 of file structs.h.

3.39.2.132 facnum

```
WORD M_const::facnum
```

Definition at line 1455 of file structs.h.

3.39.2.133 invfacnum

```
WORD M_const::invfacnum
```

Definition at line 1456 of file structs.h.

3.39.2.134 sumnum

```
WORD M_const::sumnum
```

Definition at line 1457 of file structs.h.

3.39.2.135 sumpnum

```
WORD M_const::sumpnum
```

Definition at line 1458 of file structs.h.

3.39.2.136 OldOrderFlag

```
WORD M_const::OldOrderFlag
```

Definition at line 1459 of file structs.h.

3.39.2.137 termfunnum

```
WORD M_const::termfunnum
```

Definition at line 1460 of file structs.h.

3.39.2.138 matchfunnum

```
WORD M_const::matchfunnum
```

Definition at line 1461 of file structs.h.

3.39.2.139 countfunnum

```
WORD M_const::countfunnum
```

Definition at line 1462 of file structs.h.

3.39.2.140 gPolyFun

```
WORD M_const::gPolyFun
```

Definition at line 1463 of file structs.h.

3.39.2.141 gPolyFunInv

```
WORD M_const::gPolyFunInv
```

Definition at line 1464 of file structs.h.

3.39.2.142 gPolyFunType

```
WORD M_const::gPolyFunType
```

Definition at line 1465 of file structs.h.

3.39.2.143 gPolyFunExp

```
WORD M_const::gPolyFunExp
```

Definition at line 1466 of file structs.h.

3.39.2.144 gPolyFunVar

```
WORD M_const::gPolyFunVar
```

Definition at line 1467 of file structs.h.

3.39.2.145 gPolyFunPow

```
WORD M_const::gPolyFunPow
```

Definition at line 1468 of file structs.h.

3.39.2.146 dollarzero

```
WORD M_const::dollarzero
```

Definition at line 1469 of file structs.h.

3.39.2.147 atstartup

```
WORD M_const::atstartup
```

Definition at line 1470 of file structs.h.

3.39.2.148 exitflag

```
WORD M_const::exitflag
```

Definition at line 1471 of file structs.h.

3.39.2.149 NumStoreCaches

```
WORD M_const::NumStoreCaches
```

Definition at line 1472 of file structs.h.

3.39.2.150 gIndentSpace

```
WORD M_const::gIndentSpace
```

Definition at line 1473 of file structs.h.

3.39.2.151 ggIndentSpace

```
WORD M_const::ggIndentSpace
```

Definition at line 1474 of file structs.h.

3.39.2.152 gShortStatsMax

```
WORD M_const::gShortStatsMax
```

For On FewerStatistics 10;

Definition at line 1475 of file structs.h.

3.39.2.153 ggShortStatsMax

```
WORD M_const::ggShortStatsMax
```

For On FewerStatistics 10;

Definition at line 1476 of file structs.h.

3.39.2.154 gextrasymbols

```
WORD M_const::gextrasymbols
```

Definition at line 1477 of file structs.h.

3.39.2.155 ggextrasymbols

```
WORD M_const::ggextrasymbols
```

Definition at line 1478 of file structs.h.

3.39.2.156 zerorhs

```
WORD M_const::zerorhs
```

Definition at line 1479 of file structs.h.

3.39.2.157 onerhs

```
WORD M_const::onerhs
```

Definition at line 1480 of file structs.h.

3.39.2.158 havesortdir

```
WORD M_const::havesortdir
```

Definition at line 1481 of file structs.h.

3.39.2.159 BracketFactors

```
WORD M_const::BracketFactors[8]
```

Definition at line 1482 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.40 MiNmAx Struct Reference

Data Fields

- WORD mini
- WORD maxi
- WORD size

3.40.1 Detailed Description

Definition at line 294 of file structs.h.

3.40.2 Field Documentation

3.40.2.1 mini

```
WORD MiNmAx::mini
```

Minimum value

Definition at line 295 of file structs.h.

Referenced by TestSub().

3.40.2.2 maxi

```
WORD MiNmAx::maxi
```

Maximum value

Definition at line 296 of file structs.h.

Referenced by TestSub().

3.40.2.3 size

```
WORD MiNmAx::size
```

Value of one unit in this position.

Definition at line 297 of file structs.h.

Referenced by TestSub().

The documentation for this struct was generated from the following file:

- structs.h

3.41 MODNUM Struct Reference

Data Fields

- UWORD * a
- UWORD * m
- WORD na
- WORD nm

3.41.1 Detailed Description

Definition at line 1231 of file structs.h.

3.41.2 Field Documentation

3.41.2.1 a

```
UWORD* MODNUM::a
```

Definition at line 1232 of file structs.h.

3.41.2.2 m

```
UWORD* MODNUM::m
```

Definition at line 1233 of file structs.h.

3.41.2.3 na

```
WORD MODNUM::na
```

Definition at line 1234 of file structs.h.

3.41.2.4 nm

```
WORD MODNUM::nm
```

Definition at line 1235 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.42 MoDoPtDoLlArS Struct Reference

Data Fields

- WORD number
- WORD type

3.42.1 Detailed Description

Definition at line 541 of file structs.h.

3.42.2 Field Documentation

3.42.2.1 number

```
WORD MoDoPtDoLlArS::number
```

Definition at line 545 of file structs.h.

3.42.2.2 type

```
WORD MoDoPtDoLlArS::type
```

Definition at line 546 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.43 monomial_larger Class Reference

Public Member Functions

- bool operator() (const WORD *a, const WORD *b)

3.43.1 Detailed Description

Definition at line 169 of file poly.h.

3.43.2 Constructor & Destructor Documentation

3.43.2.1 monomial_larger()

```
monomial_larger::monomial_larger ( ) [inline]
```

Definition at line 174 of file poly.h.

3.43.3 Member Function Documentation

3.43.3.1 operator()()

```
bool monomial_larger::operator() (
            const WORD * a,
            const WORD * b ) [inline]
```

Definition at line 180 of file poly.h.

The documentation for this class was generated from the following file:

- poly.h

3.44 N_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for N_const:

Public Member Functions

- **PADPOSITION** (50, 7, 23, 26, sizeof(SHvariables))

Data Fields

- POSITION OldPosIn
- POSITION OldPosOut
- POSITION theposition
- WORD * EndNest
- WORD * Frozen
- WORD * FullProto
- WORD * cTerm
- int * RepPoint
- WORD * WildValue
- WORD * WildStop
- WORD * argaddress
- WORD * RepFunList
- WORD * patstop
- WORD * terstop
- WORD * terstart
- WORD * terfirstcomm
- WORD * DumFound
- WORD ** DumPlace
- WORD ** DumFunPlace
- WORD * UsedSymbol
- WORD * UsedVector
- WORD * UsedIndex
- WORD * UsedFunction
- WORD * MaskPointer
- WORD * ForFindOnly
- WORD * findTerm
- WORD * findPattern
- WORD * dummyrenumlist
- int * funargs
- WORD ** funlocs
- int * funinds
- UWORD * NoScrat2
- WORD * ReplaceScrat
- TRACES * tracestack
- WORD * selecttermundo
- WORD * patternbuffer
- WORD * termbuffer
- WORD ** PoinScratch
- WORD ** FunScratch
- WORD * RenumScratch
- FUN_INFO * FunInfo
- WORD ** SplitScratch
- WORD ** SplitScratch1
- SORTING ** FunSorts
- UWORD * SoScratC
- WORD * listinprint
- WORD * currentTerm
- WORD ** arglist

- int * tlistbuf
- WORD * compressSpace
- UWORD * SHcombi
- WORD * poly_vars
- UWORD * cmod
- SHvariables SHvar
- LONG deferskipped
- LONG InScratch
- LONG SplitScratchSize
- LONG InScratch1
- LONG SplitScratchSize1
- LONG ninterms
- LONG SHcombisize
- int NumTotWildArgs
- int UseFindOnly
- int UsedOtherFind
- int ErrorInDollar
- int numfargs
- int numflocs
- int nargs
- int tohunt
- int numoffuns
- int funisize
- int RSsize
- int numtracesctack
- int intracestack
- int numfuninfo
- int NumFunSorts
- int MaxFunSorts
- int arglistsize
- int tlistsize
- int filenum
- int compressSize
- int polysortflag
- int nogroundlevel
- int subsubveto
- WORD MaxRenumScratch
- WORD oldtype
- WORD oldvalue
- WORD NumWild
- WORD RepFunNum
- WORD DisOrderFlag
- WORD WildDirt
- WORD NumFound
- WORD WildReserve
- WORD TeInFun
- WORD TeSuOut
- WORD WildArgs
- WORD WildEat
- WORD PolyNormFlag
- WORD PolyFunTodo
- WORD sizeselecttermundo
- WORD patternbuffersize
- WORD numlistinprint
- WORD ncmod

- WORD ExpectedSign
- WORD SignCheck
- WORD IndDum
- WORD poly_num_vars
- WORD idfunctionflag
- WORD poly_vars_type
- WORD tryterm

3.44.1 Detailed Description

The N_const struct is part of the global data and resides either in the ALLGLOBALS struct A, or the ALLPRIVATES struct B (TFORM) under the name N We see it used with the macro AN as in AN.RepFunNum It has variables that are private to each thread and are used as temporary storage during the expansion of the terms tree.

Definition at line 2074 of file structs.h.

3.44.2 Field Documentation

3.44.2.1 OldPosIn

```
POSITION N_const::OldPosIn
```

Definition at line 2075 of file structs.h.

3.44.2.2 OldPosOut

```
POSITION N_const::OldPosOut
```

Definition at line 2076 of file structs.h.

3.44.2.3 theposition

```
POSITION N_const::theposition
```

Definition at line 2077 of file structs.h.

3.44.2.4 EndNest

```
WORD* N_const::EndNest
```

Definition at line 2078 of file structs.h.

3.44.2.5 Frozen

```
WORD* N_const::Frozen
```

Definition at line 2079 of file structs.h.

3.44.2.6 FullProto

```
WORD* N_const::FullProto
```

Definition at line 2080 of file structs.h.

3.44.2.7 cTerm

```
WORD* N_const::cTerm
```

Definition at line 2081 of file structs.h.

3.44.2.8 RepPoint

```
int* N_const::RepPoint
```

Definition at line 2082 of file structs.h.

3.44.2.9 WildValue

```
WORD* N_const::WildValue
```

Definition at line 2083 of file structs.h.

3.44.2.10 WildStop

```
WORD* N_const::WildStop
```

Definition at line 2084 of file structs.h.

3.44.2.11 argaddress

```
WORD* N_const::argaddress
```

Definition at line 2085 of file structs.h.

3.44.2.12 RepFunList

```
WORD* N_const::RepFunList
```

Definition at line 2086 of file structs.h.

3.44.2.13 patstop

```
WORD* N_const::patstop
```

Definition at line 2087 of file structs.h.

3.44.2.14 terstop

```
WORD* N_const::terstop
```

Definition at line 2088 of file structs.h.

3.44.2.15 terstart

```
WORD* N_const::terstart
```

Definition at line 2089 of file structs.h.

3.44.2.16 terfirstcomm

```
WORD* N_const::terfirstcomm
```

Definition at line 2090 of file structs.h.

3.44.2.17 DumFound

```
WORD* N_const::DumFound
```

Definition at line 2091 of file structs.h.

3.44.2.18 DumPlace

```
WORD** N_const::DumPlace
```

Definition at line 2092 of file structs.h.

3.44.2.19 DumFunPlace

```
WORD** N_const::DumFunPlace
```

Definition at line 2093 of file structs.h.

3.44.2.20 UsedSymbol

```
WORD* N_const::UsedSymbol
```

Definition at line 2094 of file structs.h.

3.44.2.21 UsedVector

```
WORD* N_const::UsedVector
```

Definition at line 2095 of file structs.h.

3.44.2.22 UsedIndex

```
WORD* N_const::UsedIndex
```

Definition at line 2096 of file structs.h.

3.44.2.23 UsedFunction

```
WORD* N_const::UsedFunction
```

Definition at line 2097 of file structs.h.

3.44.2.24 MaskPointer

```
WORD* N_const::MaskPointer
```

Definition at line 2098 of file structs.h.

3.44.2.25 ForFindOnly

```
WORD* N_const::ForFindOnly
```

Definition at line 2099 of file structs.h.

3.44.2.26 findTerm

```
WORD* N_const::findTerm
```

Definition at line 2100 of file structs.h.

3.44.2.27 findPattern

```
WORD* N_const::findPattern
```

Definition at line 2101 of file structs.h.

3.44.2.28 dummyrenumlist

```
WORD* N_const::dummyrenumlist
```

Definition at line 2106 of file structs.h.

3.44.2.29 funargs

```
int* N_const::funargs
```

Definition at line 2107 of file structs.h.

3.44.2.30 funlocs

```
WORD** N_const::funlocs
```

Definition at line 2108 of file structs.h.

3.44.2.31 funinds

```
int* N_const::funinds
```

Definition at line 2109 of file structs.h.

3.44.2.32 NoScrat2

```
UWORD* N_const::NoScrat2
```

Definition at line 2110 of file structs.h.

3.44.2.33 ReplaceScrat

```
WORD* N_const::ReplaceScrat
```

Definition at line 2111 of file structs.h.

3.44.2.34 tracestack

```
TRACES* N_const::tracestack
```

Definition at line 2112 of file structs.h.

3.44.2.35 selecttermundo

```
WORD* N_const::selecttermundo
```

Definition at line 2113 of file structs.h.

3.44.2.36 patternbuffer

```
WORD* N_const::patternbuffer
```

Definition at line 2114 of file structs.h.

3.44.2.37 termbuffer

```
WORD* N_const::termbuffer
```

Definition at line 2115 of file structs.h.

3.44.2.38 PoinScratch

```
WORD** N_const::PoinScratch
```

Definition at line 2116 of file structs.h.

3.44.2.39 FunScratch

```
WORD** N_const::FunScratch
```

Definition at line 2117 of file structs.h.

3.44.2.40 RenumScratch

```
WORD* N_const::RenumScratch
```

Definition at line 2118 of file structs.h.

3.44.2.41 FunInfo

```
FUN_INFO* N_const::FunInfo
```

Definition at line 2119 of file structs.h.

3.44.2.42 SplitScratch

```
WORD** N_const::SplitScratch
```

Definition at line 2120 of file structs.h.

3.44.2.43 SplitScratch1

```
WORD** N_const::SplitScratch1
```

Definition at line 2121 of file structs.h.

3.44.2.44 FunSorts

```
SORTING** N_const::FunSorts
```

Definition at line 2122 of file structs.h.

3.44.2.45 SoScratC

```
UWORD* N_const::SoScratC
```

Definition at line 2123 of file structs.h.

3.44.2.46 listinprint

```
WORD* N_const::listinprint
```

Definition at line 2124 of file structs.h.

3.44.2.47 currentTerm

```
WORD* N_const::currentTerm
```

Definition at line 2125 of file structs.h.

3.44.2.48 arglist

```
WORD** N_const::arglist
```

Definition at line 2126 of file structs.h.

3.44.2.49 tlistbuf

```
int* N_const::tlistbuf
```

Definition at line 2127 of file structs.h.

3.44.2.50 compressSpace

```
WORD* N_const::compressSpace
```

Definition at line 2131 of file structs.h.

3.44.2.51 SHcombi

```
UWORD* N_const::SHcombi
```

Definition at line 2136 of file structs.h.

3.44.2.52 poly_vars

```
WORD* N_const::poly_vars
```

Definition at line 2137 of file structs.h.

3.44.2.53 cmod

```
UWORD* N_const::cmod
```

Definition at line 2138 of file structs.h.

3.44.2.54 SHvar

```
SHvariables N_const::SHvar
```

Definition at line 2139 of file structs.h.

3.44.2.55 deferskipped

```
LONG N_const::deferskipped
```

Definition at line 2140 of file structs.h.

3.44.2.56 InScratch

```
LONG N_const::InScratch
```

Definition at line 2141 of file structs.h.

3.44.2.57 SplitScratchSize

```
LONG N_const::SplitScratchSize
```

Definition at line 2142 of file structs.h.

3.44.2.58 InScratch1

```
LONG N_const::InScratch1
```

Definition at line 2143 of file structs.h.

3.44.2.59 SplitScratchSize1

```
LONG N_const::SplitScratchSize1
```

Definition at line 2144 of file structs.h.

3.44.2.60 ninterms

```
LONG N_const::ninterms
```

Definition at line 2145 of file structs.h.

3.44.2.61 SHcombisize

```
LONG N_const::SHcombisize
```

Definition at line 2154 of file structs.h.

3.44.2.62 NumTotWildArgs

```
int N_const::NumTotWildArgs
```

Definition at line 2155 of file structs.h.

3.44.2.63 UseFindOnly

```
int N_const::UseFindOnly
```

Definition at line 2156 of file structs.h.

3.44.2.64 UsedOtherFind

```
int N_const::UsedOtherFind
```

Definition at line 2157 of file structs.h.

3.44.2.65 ErrorInDollar

```
int N_const::ErrorInDollar
```

Definition at line 2158 of file structs.h.

3.44.2.66 numfargs

```
int N_const::numfargs
```

Definition at line 2159 of file structs.h.

3.44.2.67 numflocs

```
int N_const::numflocs
```

Definition at line 2160 of file structs.h.

3.44.2.68 nargs

```
int N_const::nargs
```

Definition at line 2161 of file structs.h.

3.44.2.69 tohunt

```
int N_const::tohunt
```

Definition at line 2162 of file structs.h.

3.44.2.70 numoffuns

```
int N_const::numoffuns
```

Definition at line 2163 of file structs.h.

3.44.2.71 funisize

```
int N_const::funisize
```

Definition at line 2164 of file structs.h.

3.44.2.72 RSsize

```
int N_const::RSsize
```

Definition at line 2165 of file structs.h.

3.44.2.73 numtracesctack

```
int N_const::numtracesctack
```

Definition at line 2166 of file structs.h.

3.44.2.74 intracestack

```
int N_const::intracestack
```

Definition at line 2167 of file structs.h.

3.44.2.75 numfuninfo

```
int N_const::numfuninfo
```

Definition at line 2168 of file structs.h.

3.44.2.76 NumFunSorts

```
int N_const::NumFunSorts
```

Definition at line 2169 of file structs.h.

3.44.2.77 MaxFunSorts

```
int N_const::MaxFunSorts
```

Definition at line 2170 of file structs.h.

3.44.2.78 arglistsize

```
int N_const::arglistsize
```

Definition at line 2171 of file structs.h.

3.44.2.79 tlistsize

```
int N_const::tlistsize
```

Definition at line 2172 of file structs.h.

3.44.2.80 filenum

```
int N_const::filenum
```

Definition at line 2173 of file structs.h.

3.44.2.81 compressSize

```
int N_const::compressSize
```

Definition at line 2174 of file structs.h.

3.44.2.82 polysortflag

```
int N_const::polysortflag
```

Definition at line 2175 of file structs.h.

3.44.2.83 nogroundlevel

```
int N_const::nogroundlevel
```

Definition at line 2176 of file structs.h.

3.44.2.84 subsubveto

```
int N_const::subsubveto
```

Definition at line 2177 of file structs.h.

3.44.2.85 MaxRenumScratch

```
WORD N_const::MaxRenumScratch
```

Definition at line 2178 of file structs.h.

3.44.2.86 oldtype

```
WORD N_const::oldtype
```

Definition at line 2179 of file structs.h.

3.44.2.87 oldvalue

```
WORD N_const::oldvalue
```

Definition at line 2180 of file structs.h.

3.44.2.88 NumWild

```
WORD N_const::NumWild
```

Definition at line 2181 of file structs.h.

3.44.2.89 RepFunNum

```
WORD N_const::RepFunNum
```

Definition at line 2182 of file structs.h.

3.44.2.90 DisOrderFlag

```
WORD N_const::DisOrderFlag
```

Definition at line 2183 of file structs.h.

3.44.2.91 WildDirt

```
WORD N_const::WildDirt
```

Definition at line 2184 of file structs.h.

3.44.2.92 NumFound

```
WORD N_const::NumFound
```

Definition at line 2185 of file structs.h.

3.44.2.93 WildReserve

```
WORD N_const::WildReserve
```

Definition at line 2186 of file structs.h.

3.44.2.94 TeInFun

```
WORD N_const::TeInFun
```

Definition at line 2187 of file structs.h.

3.44.2.95 TeSuOut

```
WORD N_const::TeSuOut
```

Definition at line 2188 of file structs.h.

3.44.2.96 WildArgs

```
WORD N_const::WildArgs
```

Definition at line 2189 of file structs.h.

3.44.2.97 WildEat

```
WORD N_const::WildEat
```

Definition at line 2190 of file structs.h.

3.44.2.98 PolyNormFlag

```
WORD N_const::PolyNormFlag
```

Definition at line 2191 of file structs.h.

3.44.2.99 PolyFunTodo

```
WORD N_const::PolyFunTodo
```

Definition at line 2192 of file structs.h.

3.44.2.100 sizeselecttermundo

```
WORD N_const::sizeselecttermundo
```

Definition at line 2193 of file structs.h.

3.44.2.101 patternbuffersize

```
WORD N_const::patternbuffersize
```

Definition at line 2194 of file structs.h.

3.44.2.102 numlistinprint

```
WORD N_const::numlistinprint
```

Definition at line 2195 of file structs.h.

3.44.2.103 ncmod

```
WORD N_const::ncmod
```

Definition at line 2196 of file structs.h.

3.44.2.104 ExpectedSign

```
WORD N_const::ExpectedSign
```

Definition at line 2197 of file structs.h.

3.44.2.105 SignCheck

```
WORD N_const::SignCheck
```

Used in pattern matching of antisymmetric functions

Definition at line 2198 of file structs.h.

3.44.2.106 IndDum

```
WORD N_const::IndDum
```

Used in pattern matching of antisymmetric functions

Definition at line 2199 of file structs.h.

3.44.2.107 poly_num_vars

```
WORD N_const::poly_num_vars
```

Definition at line 2200 of file structs.h.

3.44.2.108 idfunctionflag

```
WORD N_const::idfunctionflag
```

Definition at line 2201 of file structs.h.

3.44.2.109 poly_vars_type

```
WORD N_const::poly_vars_type
```

Definition at line 2202 of file structs.h.

3.44.2.110 tryterm

```
WORD N_const::tryterm
```

Definition at line 2203 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.45 nameblock Struct Reference

Data Fields

- MLONG previousblock
- MLONG position
- char names [NAMETABLESIZE]

3.45.1 Detailed Description

Definition at line 114 of file minos.h.

3.45.2 Field Documentation

3.45.2.1 previousblock

```
MLONG nameblock::previousblock
```

Definition at line 115 of file minos.h.

3.45.2.2 position

```
MLONG nameblock::position
```

Definition at line 116 of file minos.h.

3.45.2.3 names

```
char nameblock::names[NAMETABLESIZE]
```

Definition at line 117 of file minos.h.

The documentation for this struct was generated from the following file:

- minos.h

3.46 NaMeNode Struct Reference

```
#include <structs.h>
```

Data Fields

- LONG name
- WORD parent
- WORD left
- WORD right
- WORD balance
- WORD type
- WORD number

3.46.1 Detailed Description

The names of variables are kept in an array. Elements of type NAMENODE define a tree (that is kept balanced) that make it easy and fast to look for variables. See also NAMETREE.

Definition at line 234 of file structs.h.

3.46.2 Field Documentation

3.46.2.1 name

```
LONG NaMeNode::name
```

Offset into NAMETREE::namebuffer.

Definition at line 235 of file structs.h.

3.46.2.2 parent

```
WORD NaMeNode::parent
```

=-1 if no parent.

Definition at line 236 of file structs.h.

3.46.2.3 left

```
WORD NaMeNode::left
```

=-1 if no child.

Definition at line 237 of file structs.h.

3.46.2.4 right

```
WORD NaMeNode::right
```

=-1 if no child.

Definition at line 238 of file structs.h.

3.46.2.5 balance

```
WORD NaMeNode::balance
```

Used for the balancing of the tree.

Definition at line 239 of file structs.h.

3.46.2.6 type

```
WORD NaMeNode::type
```

Type associated with the name. See compiler types.

Definition at line 240 of file structs.h.

3.46.2.7 number

```
WORD NaMeNode::number
```

Number of variable in LIST's like for example C_const::SymbolList.

Definition at line 241 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.47 NaMeTree Struct Reference

```
#include <structs.h>
```

Collaboration diagram for NaMeTree:

Data Fields

- NAMENODE * namenode
- UBYTE * namebuffer
- LONG nodesize
- LONG nodefill
- LONG namesize
- LONG namefill
- LONG oldnamefill
- LONG oldnodefill
- LONG globalnamefill
- LONG globalnodefill
- LONG clearnamefill
- LONG clearnodefill
- WORD headnode

3.47.1 Detailed Description

A struct of type NAMETREE controls a complete (balanced) tree of names for the compiler. The compiler maintains several of such trees and the system has been set up in such a way that one could define more of them if we ever want to work with local name spaces.

Definition at line 252 of file structs.h.

3.47.2 Field Documentation

3.47.2.1 namenode

```
NAMENODE* NaMeTree::namenode
```

[D] Vector of NAMENODE's. Number of elements is nodesize. =0 if no memory has been allocated.

Definition at line 253 of file structs.h.

3.47.2.2 namebuffer

```
UBYTE* NaMeTree::namebuffer
```

[D] Buffer that holds all the name strings refered to by the NAMENODE's. Allocation size is namesize. =0 if no memory has been allocated.

Definition at line 255 of file structs.h.

3.47.2.3 nodesize

```
LONG NaMeTree::nodesize
```

Maximum number of elements in namenode.

Definition at line 258 of file structs.h.

3.47.2.4 nodefill

```
LONG NaMeTree::nodefill
```

Number of currently used nodes in namenode.

Definition at line 259 of file structs.h.

3.47.2.5 namesize

```
LONG NaMeTree::namesize
```

Allocation size of namebuffer in bytes.

Definition at line 260 of file structs.h.

3.47.2.6 namefill

```
LONG NaMeTree::namefill
```

Number of bytes occupied.

Definition at line 261 of file structs.h.

3.47.2.7 oldnamefill

```
LONG NaMeTree::oldnamefill
```

UNUSED

Definition at line 262 of file structs.h.

3.47.2.8 oldnodefill

```
LONG NaMeTree::oldnodefill
```

UNUSED

Definition at line 263 of file structs.h.

3.47.2.9 globalnamefill

```
LONG NaMeTree::globalnamefill
```

Set by .global statement to the value of namefill. When a .store command is processed, this value will be used to reset namefill.

Definition at line 264 of file structs.h.

3.47.2.10 globalnodefill

```
LONG NaMeTree::globalnodefill
```

Same usage as globalnamefill, but for nodefill.

Definition at line 266 of file structs.h.

3.47.2.11 clearnamefill

```
LONG NaMeTree::clearnamefill
```

Marks the reset point used by the .clear statement.

Definition at line 267 of file structs.h.

3.47.2.12 clearnodefill

```
LONG NaMeTree::clearnodefill
```

Marks the reset point used by the .clear statement.

Definition at line 268 of file structs.h.

3.47.2.13 headnode

```
WORD NaMeTree::headnode
```

Offset in namenode of head node. =-1 if tree is empty.

Definition at line 269 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.48 NeStInG Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * termsize
- WORD * funsize
- WORD * argsize

3.48.1 Detailed Description

The NESTING struct is used when we enter the argument of functions and there is the possibility that we have to change something there. Because functions can be nested we have to keep track of all levels of functions in case we have to move the outer layers to make room for a larger function argument.

Definition at line 982 of file structs.h.

3.48.2 Field Documentation

3.48.2.1 termsize

```
WORD* NeStInG::termsize
```

Definition at line 983 of file structs.h.

3.48.2.2 funsize

```
WORD* NeStInG::funsize
```

Definition at line 984 of file structs.h.

3.48.2.3 argsize

```
WORD* NeStInG::argsize
```

Definition at line 985 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.49 NoDe Struct Reference

Collaboration diagram for NoDe:

Data Fields

- struct NoDe * left
- struct NoDe * rght
- int lloser
- int rloser
- int lsrc
- int rsrc

3.49.1 Detailed Description

A node for the tree of losers in the final sorting on the master.

Definition at line 265 of file parallel.c.

3.49.2 Field Documentation

3.49.2.1 left

```
struct NoDe* NoDe::left
```

Definition at line 266 of file parallel.c.

3.49.2.2 rght

```
struct NoDe* NoDe::rght
```

Definition at line 267 of file parallel.c.

3.49.2.3 lloser

```
int NoDe::lloser
```

Definition at line 268 of file parallel.c.

3.49.2.4 rloser

```
int NoDe::rloser
```

Definition at line 269 of file parallel.c.

3.49.2.5 lsrc

```
int NoDe::lsrc
```

Definition at line 270 of file parallel.c.

3.49.2.6 rsrc

```
int NoDe::rsrc
```

Definition at line 271 of file parallel.c.

The documentation for this struct was generated from the following file:

- parallel.c

3.50 O_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for O_const:

Public Member Functions

- **PADPOSITION** (25, 4, 35, 17, 1)

Data Fields

- FILEDATA SaveData
- STOREHEADER SaveHeader
- OPTIMIZERESULT OptimizeResult
- UBYTE * OutputLine
- UBYTE * OutStop
- UBYTE * OutFill
- WORD * bracket
- WORD * termbuf
- WORD * tabstring
- UBYTE * wpos
- UBYTE * wpoin
- UBYTE * DollarOutBuffer
- UBYTE * CurBufWrt
- VOID(* FlipWORD)(UBYTE *)
- VOID(* FlipLONG)(UBYTE *)
- VOID(* FlipPOS)(UBYTE *)
- VOID(* FlipPOINTER)(UBYTE *)
- VOID(* ResizeData)(UBYTE *, int, UBYTE *, int)
- VOID(* ResizeWORD)(UBYTE *, UBYTE *)
- VOID(* ResizeNCWORD)(UBYTE *, UBYTE *)
- VOID(* ResizeLONG)(UBYTE *, UBYTE *)
- VOID(* ResizePOS)(UBYTE *, UBYTE *)
- VOID(* ResizePOINTER)(UBYTE *, UBYTE *)
- VOID(* CheckPower)(UBYTE *)
- VOID(* RenumberVec)(UBYTE *)
- DICTIONARY ** Dictionaries
- UBYTE * tensorList
- WORD * inscheme
- LONG NumInBrack
- LONG wlen
- LONG DollarOutSizeBuffer
- LONG DollarInOutBuffer
- OPTIMIZE Optimize
- int OutInBuffer
- int NoSpacesInNumbers
- int BlockSpaces
- int CurrentDictionary
- int SizeDictionaries
- int NumDictionaries
- int CurDictNumbers
- int CurDictVariables
- int CurDictSpecials
- int CurDictFunWithArgs
- int CurDictNumberWarning
- int CurDictNotInFunctions
- int CurDictInDollars
- int gNumDictionaries
- WORD schemenum

- WORD transFlag
- WORD powerFlag
- WORD mpower
- WORD resizeFlag
- WORD bufferedInd
- WORD OutSkip
- WORD IsBracket
- WORD InFbrack
- WORD PrintType
- WORD FortFirst
- WORD DoubleFlag
- WORD IndentSpace
- WORD FactorMode
- WORD FactorNum
- WORD ErrorBlock
- WORD OptimizationLevel
- UBYTE FortDotChar

3.50.1 Detailed Description

The O_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name O We see it used with the macro AO as in AO.OutputLine It contains variables that involve the writing of text output and save/store files.

Definition at line 2251 of file structs.h.

3.50.2 Field Documentation

3.50.2.1 SaveData

```
FILEDATA O_const::SaveData
```

Definition at line 2252 of file structs.h.

3.50.2.2 SaveHeader

```
STOREHEADER O_const::SaveHeader
```

Definition at line 2253 of file structs.h.

3.50.2.3 OptimizeResult

```
OPTIMIZERESULT O_const::OptimizeResult
```

Definition at line 2254 of file structs.h.

3.50.2.4 OutputLine

```
UBYTE* O_const::OutputLine
```

Definition at line 2255 of file structs.h.

3.50.2.5 OutStop

```
UBYTE* O_const::OutStop
```

Definition at line 2256 of file structs.h.

3.50.2.6 OutFill

```
UBYTE* O_const::OutFill
```

Definition at line 2257 of file structs.h.

3.50.2.7 bracket

```
WORD* O_const::bracket
```

Definition at line 2258 of file structs.h.

3.50.2.8 termbuf

```
WORD* O_const::termbuf
```

Definition at line 2259 of file structs.h.

3.50.2.9 tabstring

```
WORD* O_const::tabstring
```

Definition at line 2260 of file structs.h.

3.50.2.10 wpos

```
UBYTE* O_const::wpos
```

Definition at line 2261 of file structs.h.

3.50.2.11 wpoin

```
UBYTE* O_const::wpoin
```

Definition at line 2262 of file structs.h.

3.50.2.12 DollarOutBuffer

```
UBYTE* O_const::DollarOutBuffer
```

Definition at line 2263 of file structs.h.

3.50.2.13 CurBufWrt

```
UBYTE* O_const::CurBufWrt
```

Definition at line 2264 of file structs.h.

3.50.2.14 FlipWORD

```
VOID(* O_const::FlipWORD) (UBYTE *)
```

Definition at line 2265 of file structs.h.

3.50.2.15 FlipLONG

```
VOID(* O_const::FlipLONG) (UBYTE *)
```

Definition at line 2266 of file structs.h.

3.50.2.16 FlipPOS

```
VOID(* O_const::FlipPOS) (UBYTE *)
```

Definition at line 2267 of file structs.h.

3.50.2.17 FlipPOINTER

```
VOID(* O_const::FlipPOINTER) (UBYTE *)
```

Definition at line 2268 of file structs.h.

3.50.2.18 ResizeData

```
VOID(* O_const::ResizeData) (UBYTE *, int, UBYTE *, int)
```

Definition at line 2269 of file structs.h.

3.50.2.19 ResizeWORD

```
VOID(* O_const::ResizeWORD) (UBYTE *, UBYTE *)
```

Definition at line 2270 of file structs.h.

3.50.2.20 ResizeNCWORD

```
VOID(* O_const::ResizeNCWORD) (UBYTE *, UBYTE *)
```

Definition at line 2271 of file structs.h.

3.50.2.21 ResizeLONG

```
VOID(* O_const::ResizeLONG) (UBYTE *, UBYTE *)
```

Definition at line 2272 of file structs.h.

3.50.2.22 ResizePOS

```
VOID(* O_const::ResizePOS) (UBYTE *, UBYTE *)
```

Definition at line 2273 of file structs.h.

3.50.2.23 ResizePOINTER

```
VOID(* O_const::ResizePOINTER) (UBYTE *, UBYTE *)
```

Definition at line 2274 of file structs.h.

3.50.2.24 CheckPower

```
VOID(* O_const::CheckPower) (UBYTE *)
```

Definition at line 2275 of file structs.h.

3.50.2.25 RenumberVec

```
VOID(* O_const::RenumberVec) (UBYTE *)
```

Definition at line 2276 of file structs.h.

3.50.2.26 Dictionaries

```
DICTIONARY** O_const::Dictionaries
```

Definition at line 2277 of file structs.h.

3.50.2.27 tensorList

```
UBYTE* O_const::tensorList
```

Definition at line 2278 of file structs.h.

3.50.2.28 inscheme

```
WORD* O_const::inscheme
```

Definition at line 2279 of file structs.h.

3.50.2.29 NumInBrack

```
LONG O_const::NumInBrack
```

Definition at line 2281 of file structs.h.

3.50.2.30 wlen

```
LONG O_const::wlen
```

Definition at line 2282 of file structs.h.

3.50.2.31 DollarOutSizeBuffer

```
LONG O_const::DollarOutSizeBuffer
```

Definition at line 2283 of file structs.h.

3.50.2.32 DollarInOutBuffer

```
LONG O_const::DollarInOutBuffer
```

Definition at line 2284 of file structs.h.

3.50.2.33 Optimize

```
OPTIMIZE O_const::Optimize
```

Definition at line 2289 of file structs.h.

3.50.2.34 OutInBuffer

```
int O_const::OutInBuffer
```

Definition at line 2290 of file structs.h.

3.50.2.35 NoSpacesInNumbers

```
int O_const::NoSpacesInNumbers
```

Definition at line 2291 of file structs.h.

3.50.2.36 BlockSpaces

```
int O_const::BlockSpaces
```

Definition at line 2292 of file structs.h.

3.50.2.37 CurrentDictionary

```
int O_const::CurrentDictionary
```

Definition at line 2293 of file structs.h.

3.50.2.38 SizeDictionaries

```
int O_const::SizeDictionaries
```

Definition at line 2294 of file structs.h.

3.50.2.39 NumDictionaries

```
int O_const::NumDictionaries
```

Definition at line 2295 of file structs.h.

3.50.2.40 CurDictNumbers

```
int O_const::CurDictNumbers
```

Definition at line 2296 of file structs.h.

3.50.2.41 CurDictVariables

```
int O_const::CurDictVariables
```

Definition at line 2297 of file structs.h.

3.50.2.42 CurDictSpecials

```
int O_const::CurDictSpecials
```

Definition at line 2298 of file structs.h.

3.50.2.43 CurDictFunWithArgs

```
int O_const::CurDictFunWithArgs
```

Definition at line 2299 of file structs.h.

3.50.2.44 CurDictNumberWarning

```
int O_const::CurDictNumberWarning
```

Definition at line 2300 of file structs.h.

3.50.2.45 CurDictNotInFunctions

```
int O_const::CurDictNotInFunctions
```

Definition at line 2301 of file structs.h.

3.50.2.46 CurDictInDollars

```
int O_const::CurDictInDollars
```

Definition at line 2302 of file structs.h.

3.50.2.47 gNumDictionaries

```
int O_const::gNumDictionaries
```

Definition at line 2303 of file structs.h.

3.50.2.48 schemenum

```
WORD O_const::schemenum
```

Definition at line 2304 of file structs.h.

3.50.2.49 transFlag

```
WORD O_const::transFlag
```

Definition at line 2305 of file structs.h.

3.50.2.50 powerFlag

```
WORD O_const::powerFlag
```

Definition at line 2306 of file structs.h.

3.50.2.51 mpower

```
WORD O_const::mpower
```

Definition at line 2307 of file structs.h.

3.50.2.52 resizeFlag

```
WORD O_const::resizeFlag
```

Definition at line 2308 of file structs.h.

3.50.2.53 bufferedInd

```
WORD O_const::bufferedInd
```

Definition at line 2309 of file structs.h.

3.50.2.54 OutSkip

```
WORD O_const::OutSkip
```

Definition at line 2310 of file structs.h.

3.50.2.55 IsBracket

```
WORD O_const::IsBracket
```

Definition at line 2311 of file structs.h.

3.50.2.56 InFbrack

```
WORD O_const::InFbrack
```

Definition at line 2312 of file structs.h.

3.50.2.57 PrintType

```
WORD O_const::PrintType
```

Definition at line 2313 of file structs.h.

3.50.2.58 FortFirst

```
WORD O_const::FortFirst
```

Definition at line 2314 of file structs.h.

3.50.2.59 DoubleFlag

```
WORD O_const::DoubleFlag
```

Definition at line 2315 of file structs.h.

3.50.2.60 IndentSpace

```
WORD O_const::IndentSpace
```

Definition at line 2316 of file structs.h.

3.50.2.61 FactorMode

```
WORD O_const::FactorMode
```

Definition at line 2317 of file structs.h.

3.50.2.62 FactorNum

```
WORD O_const::FactorNum
```

Definition at line 2318 of file structs.h.

3.50.2.63 ErrorBlock

```
WORD O_const::ErrorBlock
```

Definition at line 2319 of file structs.h.

3.50.2.64 OptimizationLevel

```
WORD O_const::OptimizationLevel
```

Definition at line 2320 of file structs.h.

3.50.2.65 FortDotChar

```
UBYTE O_const::FortDotChar
```

Definition at line 2321 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.51 objects Struct Reference

Data Fields

- MLONG position
- MLONG size
- MLONG date
- MLONG tablenumber
- MLONG uncompressed
- MLONG spare1
- MLONG spare2
- MLONG spare3
- char element [ELEMENTSIZE]

3.51.1 Detailed Description

Definition at line 94 of file minos.h.

3.51.2 Field Documentation

3.51.2.1 position

```
MLONG objects::position
```

Definition at line 96 of file minos.h.

3.51.2.2 size

```
MLONG objects::size
```

Definition at line 97 of file minos.h.

3.51.2.3 date

```
MLONG objects::date
```

Definition at line 98 of file minos.h.

3.51.2.4 tablenumber

```
MLONG objects::tablenumber
```

Definition at line 99 of file minos.h.

3.51.2.5 uncompressed

```
MLONG objects::uncompressed
```

Definition at line 100 of file minos.h.

3.51.2.6 spare1

```
MLONG objects::spare1
```

Definition at line 101 of file minos.h.

3.51.2.7 spare2

```
MLONG objects::spare2
```

Definition at line 102 of file minos.h.

3.51.2.8 spare3

```
MLONG objects::spare3
```

Definition at line 103 of file minos.h.

3.51.2.9 element

```
char objects::element[ELEMENTSIZE]
```

Definition at line 104 of file minos.h.

The documentation for this struct was generated from the following file:

- minos.h

3.52 OPTIMIZE Struct Reference

Data Fields

- union {
  float fval
  int ival [2]
  } **mctsconstant**

- int horner
- int hornerdirection
- int method
- int mctstimelimit
- int mctsnumexpand
- int mctsnumkeep
- int mctsnumrepeat
- int greedytimelimit
- int greedyminnum
- int greedymaxperc
- int printstats
- int debugflags
- int schemeflags
- int mctsdecaymode
- int salter
- union {
  float fval
  int ival [2]
  } **saMaxT**

- union {
  float fval
  int ival [2]
  } **saMinT**

3.52.1 Detailed Description

Definition at line 1242 of file structs.h.

3.52.2 Field Documentation

3.52.2.1 fval

```
float OPTIMIZE::fval
```

Definition at line 1244 of file structs.h.

3.52.2.2 ival

```
int OPTIMIZE::ival[2]
```

Definition at line 1245 of file structs.h.

3.52.2.3 horner

```
int OPTIMIZE::horner
```

Definition at line 1247 of file structs.h.

3.52.2.4 hornerdirection

```
int OPTIMIZE::hornerdirection
```

Definition at line 1248 of file structs.h.

3.52.2.5 method

```
int OPTIMIZE::method
```

Definition at line 1249 of file structs.h.

3.52.2.6 mctstimelimit

```
int OPTIMIZE::mctstimelimit
```

Definition at line 1250 of file structs.h.

3.52.2.7 mctsnumexpand

```
int OPTIMIZE::mctsnumexpand
```

Definition at line 1251 of file structs.h.

3.52.2.8 mctsnumkeep

```
int OPTIMIZE::mctsnumkeep
```

Definition at line 1252 of file structs.h.

3.52.2.9 mctsnumrepeat

```
int OPTIMIZE::mctsnumrepeat
```

Definition at line 1253 of file structs.h.

3.52.2.10 greedytimelimit

```
int OPTIMIZE::greedytimelimit
```

Definition at line 1254 of file structs.h.

3.52.2.11 greedyminnum

```
int OPTIMIZE::greedyminnum
```

Definition at line 1255 of file structs.h.

3.52.2.12 greedymaxperc

```
int OPTIMIZE::greedymaxperc
```

Definition at line 1256 of file structs.h.

3.52.2.13 printstats

```
int OPTIMIZE::printstats
```

Definition at line 1257 of file structs.h.

3.52.2.14 debugflags

```
int OPTIMIZE::debugflags
```

Definition at line 1258 of file structs.h.

3.52.2.15 schemeflags

```
int OPTIMIZE::schemeflags
```

Definition at line 1259 of file structs.h.

3.52.2.16 mctsdecaymode

```
int OPTIMIZE::mctsdecaymode
```

Definition at line 1260 of file structs.h.

3.52.2.17 salter

```
int OPTIMIZE::saIter
```

Definition at line 1261 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.53 OPTIMIZERESULT Struct Reference

Public Member Functions

- **PADPOSITION** (2, 1, 0, 3, 0)

Data Fields

- WORD * code
- UBYTE * nameofexpr
- LONG codesize
- WORD exprnr
- WORD minvar
- WORD maxvar

3.53.1 Detailed Description

Definition at line 1272 of file structs.h.

3.53.2 Field Documentation

3.53.2.1 code

```
WORD* OPTIMIZERESULT::code
```

Definition at line 1273 of file structs.h.

3.53.2.2 nameofexpr

```
UBYTE* OPTIMIZERESULT::nameofexpr
```

Definition at line 1274 of file structs.h.

3.53.2.3 codesize

```
LONG OPTIMIZERESULT::codesize
```

Definition at line 1275 of file structs.h.

3.53.2.4 exprnr

```
WORD OPTIMIZERESULT::exprnr
```

Definition at line 1276 of file structs.h.

3.53.2.5 minvar

```
WORD OPTIMIZERESULT::minvar
```

Definition at line 1277 of file structs.h.

3.53.2.6 maxvar

```
WORD OPTIMIZERESULT::maxvar
```

Definition at line 1278 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.54 P_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for P_const:

Data Fields

- LIST DollarList
- LIST PreVarList
- LIST LoopList
- LIST ProcList
- INSIDEINFO inside
- UBYTE ** PreSwitchStrings
- UBYTE * preStart
- UBYTE * preStop
- UBYTE * preFill

- UBYTE * procedureExtension
- UBYTE * cprocedureExtension
- LONG * PreAssignStack
- int * PreIfStack
- int * PreSwitchModes
- int * PreTypes
- LONG StopWatchZero
- LONG InOutBuf
- LONG pSize
- int PreAssignFlag
- int PreContinuation
- int PreproFlag
- int iBufError
- int PreOut
- int PreSwitchLevel
- int NumPreSwitchStrings
- int MaxPreTypes
- int NumPreTypes
- int MaxPreIfLevel
- int PreIfLevel
- int PreInsideLevel
- int DelayPrevar
- int AllowDelay
- int lhdollarerror
- int eat
- int gNumPre
- int PreDebug
- int OpenDictionary
- int PreAssignLevel
- int MaxPreAssignLevel
- WORD DebugFlag
- WORD preError
- UBYTE ComChar
- UBYTE cComChar

3.54.1 Detailed Description

The P_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name P We see it used with the macro AP as in AP.InOutBuf It contains objects that have dealings with the preprocessor.

Definition at line 1500 of file structs.h.

3.54.2 Field Documentation

3.54.2.1 DollarList

```
LIST P_const::DollarList
```

Definition at line 1501 of file structs.h.

3.54.2.2 PreVarList

```
LIST P_const::PreVarList
```

Definition at line 1503 of file structs.h.

3.54.2.3 LoopList

```
LIST P_const::LoopList
```

Definition at line 1505 of file structs.h.

3.54.2.4 ProcList

```
LIST P_const::ProcList
```

Definition at line 1506 of file structs.h.

3.54.2.5 inside

```
INSIDEINFO P_const::inside
```

Definition at line 1507 of file structs.h.

3.54.2.6 PreSwitchStrings

```
UBYTE** P_const::PreSwitchStrings
```

Definition at line 1508 of file structs.h.

3.54.2.7 preStart

```
UBYTE* P_const::preStart
```

Definition at line 1509 of file structs.h.

3.54.2.8 preStop

```
UBYTE* P_const::preStop
```

Definition at line 1510 of file structs.h.

3.54.2.9 preFill

```
UBYTE* P_const::preFill
```

Definition at line 1511 of file structs.h.

3.54.2.10 procedureExtension

```
UBYTE* P_const::procedureExtension
```

Definition at line 1512 of file structs.h.

3.54.2.11 cprocedureExtension

```
UBYTE* P_const::cprocedureExtension
```

Definition at line 1513 of file structs.h.

3.54.2.12 PreAssignStack

```
LONG* P_const::PreAssignStack
```

Definition at line 1514 of file structs.h.

3.54.2.13 PreIfStack

```
int* P_const::PreIfStack
```

Definition at line 1515 of file structs.h.

3.54.2.14 PreSwitchModes

```
int* P_const::PreSwitchModes
```

Definition at line 1516 of file structs.h.

3.54.2.15 PreTypes

```
int* P_const::PreTypes
```

Definition at line 1517 of file structs.h.

3.54.2.16 StopWatchZero

```
LONG P_const::StopWatchZero
```

Definition at line 1521 of file structs.h.

3.54.2.17 InOutBuf

```
LONG P_const::InOutBuf
```

Definition at line 1522 of file structs.h.

3.54.2.18 pSize

```
LONG P_const::pSize
```

Definition at line 1523 of file structs.h.

3.54.2.19 PreAssignFlag

```
int P_const::PreAssignFlag
```

Definition at line 1524 of file structs.h.

3.54.2.20 PreContinuation

```
int P_const::PreContinuation
```

Definition at line 1525 of file structs.h.

3.54.2.21 PreproFlag

```
int P_const::PreproFlag
```

Definition at line 1526 of file structs.h.

3.54.2.22 iBufError

```
int P_const::iBufError
```

Definition at line 1527 of file structs.h.

3.54.2.23 PreOut

```
int P_const::PreOut
```

Definition at line 1528 of file structs.h.

3.54.2.24 PreSwitchLevel

```
int P_const::PreSwitchLevel
```

Definition at line 1529 of file structs.h.

3.54.2.25 NumPreSwitchStrings

```
int P_const::NumPreSwitchStrings
```

Definition at line 1530 of file structs.h.

3.54.2.26 MaxPreTypes

```
int P_const::MaxPreTypes
```

Definition at line 1531 of file structs.h.

3.54.2.27 NumPreTypes

```
int P_const::NumPreTypes
```

Definition at line 1532 of file structs.h.

3.54.2.28 MaxPreIfLevel

```
int P_const::MaxPreIfLevel
```

Definition at line 1533 of file structs.h.

3.54.2.29 PreIfLevel

```
int P_const::PreIfLevel
```

Definition at line 1534 of file structs.h.

3.54.2.30 PreInsideLevel

```
int P_const::PreInsideLevel
```

Definition at line 1535 of file structs.h.

3.54.2.31 DelayPrevar

```
int P_const::DelayPrevar
```

Definition at line 1536 of file structs.h.

3.54.2.32 AllowDelay

```
int P_const::AllowDelay
```

Definition at line 1537 of file structs.h.

3.54.2.33 lhdollarerror

```
int P_const::lhdollarerror
```

Definition at line 1538 of file structs.h.

3.54.2.34 eat

```
int P_const::eat
```

Definition at line 1539 of file structs.h.

3.54.2.35 gNumPre

```
int P_const::gNumPre
```

Definition at line 1540 of file structs.h.

3.54.2.36 PreDebug

```
int P_const::PreDebug
```

Definition at line 1541 of file structs.h.

3.54.2.37 OpenDictionary

```
int P_const::OpenDictionary
```

Definition at line 1542 of file structs.h.

3.54.2.38 PreAssignLevel

```
int P_const::PreAssignLevel
```

Definition at line 1543 of file structs.h.

3.54.2.39 MaxPreAssignLevel

```
int P_const::MaxPreAssignLevel
```

Definition at line 1544 of file structs.h.

3.54.2.40 DebugFlag

```
WORD P_const::DebugFlag
```

Definition at line 1545 of file structs.h.

3.54.2.41 preError

```
WORD P_const::preError
```

Definition at line 1546 of file structs.h.

3.54.2.42 ComChar

```
UBYTE P_const::ComChar
```

Definition at line 1547 of file structs.h.

3.54.2.43 cComChar

```
UBYTE P_const::cComChar
```

Definition at line 1548 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.55 ParallelVars Struct Reference

Collaboration diagram for ParallelVars:

Public Member Functions

- **PADPOSITION** (2, 0, 8, 2, 0)

Data Fields

- FILEHANDLE slavebuf
- PF_BUFFER * sbuf
- PF_BUFFER ** rbufs
- int me
- int numtasks
- int parallel
- int rhsInParallel
- int mkSlaveInfile
- int exprbufsize
- int exprtodo
- int log
- WORD numsbufs
- WORD numrbufs

3.55.1 Detailed Description

Definition at line 168 of file parallel.h.

3.55.2 Field Documentation

3.55.2.1 slavebuf

```
FILEHANDLE ParallelVars::slavebuf
```

Definition at line 169 of file parallel.h.

3.55.2.2 sbuf

```
PF_BUFFER* ParallelVars::sbuf
```

Definition at line 171 of file parallel.h.

3.55.2.3 rbufs

```
PF_BUFFER** ParallelVars::rbufs
```

Definition at line 172 of file parallel.h.

3.55.2.4 me

```
int ParallelVars::me
```

Definition at line 173 of file parallel.h.

3.55.2.5 numtasks

```
int ParallelVars::numtasks
```

Definition at line 174 of file parallel.h.

3.55.2.6 parallel

```
int ParallelVars::parallel
```

Definition at line 175 of file parallel.h.

3.55.2.7 rhsInParallel

```
int ParallelVars::rhsInParallel
```

Definition at line 177 of file parallel.h.

3.55.2.8 mkSlaveInfile

```
int ParallelVars::mkSlaveInfile
```

Definition at line 178 of file parallel.h.

3.55.2.9 exprbufsize

```
int ParallelVars::exprbufsize
```

Definition at line 179 of file parallel.h.

3.55.2.10 exprtodo

```
int ParallelVars::exprtodo
```

Definition at line 180 of file parallel.h.

3.55.2.11 log

```
int ParallelVars::log
```

Definition at line 181 of file parallel.h.

3.55.2.12 numsbufs

```
WORD ParallelVars::numsbufs
```

Definition at line 182 of file parallel.h.

3.55.2.13 numrbufs

```
WORD ParallelVars::numrbufs
```

Definition at line 183 of file parallel.h.

The documentation for this struct was generated from the following file:

- parallel.h

3.56 PaRtI Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * psize
- WORD * args
- WORD * nargs
- WORD * nfun
- WORD numargs
- WORD numpart
- WORD where

3.56.1 Detailed Description

The struct PARTI is used to help determining whether a partition_ function can be replaced.

Definition at line 1049 of file structs.h.

3.56.2 Field Documentation

3.56.2.1 psize

```
WORD* PaRtI::psize
```

Definition at line 1050 of file structs.h.

3.56.2.2 args

```
WORD* PaRtI::args
```

Definition at line 1051 of file structs.h.

3.56.2.3 nargs

```
WORD* PaRtI::nargs
```

Definition at line 1052 of file structs.h.

3.56.2.4 nfun

```
WORD* PaRtI::nfun
```

Definition at line 1053 of file structs.h.

3.56.2.5 numargs

```
WORD PaRtI::numargs
```

Definition at line 1054 of file structs.h.

3.56.2.6 numpart

```
WORD PaRtI::numpart
```

Definition at line 1055 of file structs.h.

3.56.2.7 where

```
WORD PaRtI::where
```

Definition at line 1056 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.57 PeRmUtE Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * objects
- WORD sign
- WORD n
- WORD cycle [MAXMATCH]

3.57.1 Detailed Description

The struct PERM is used to generate all permutations when the pattern matcher has to try to match (anti)symmetric functions.

Definition at line 1007 of file structs.h.

3.57.2 Field Documentation

3.57.2.1 objects

```
WORD* PeRmUtE::objects
```

Definition at line 1008 of file structs.h.

3.57.2.2 sign

```
WORD PeRmUtE::sign
```

Definition at line 1009 of file structs.h.

3.57.2.3 n

```
WORD PeRmUtE::n
```

Definition at line 1010 of file structs.h.

3.57.2.4 cycle

```
WORD PeRmUtE::cycle[MAXMATCH]
```

Definition at line 1011 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.58 PeRmUtEp Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD ** objects
- WORD sign
- WORD n
- WORD cycle [MAXMATCH]

3.58.1 Detailed Description

Like struct PERM but works with pointers.

Definition at line 1019 of file structs.h.

3.58.2 Field Documentation

3.58.2.1 objects

```
WORD** PeRmUtEp::objects
```

Definition at line 1020 of file structs.h.

3.58.2.2 sign

```
WORD PeRmUtEp::sign
```

Definition at line 1021 of file structs.h.

3.58.2.3 n

```
WORD PeRmUtEp::n
```

Definition at line 1022 of file structs.h.

3.58.2.4 cycle

```
WORD PeRmUtEp::cycle[MAXMATCH]
```

Definition at line 1023 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.59 PF_BUFFER Struct Reference

```
#include <parallel.h>
```

Data Fields

- WORD ** buff
- WORD ** fill
- WORD ** full
- WORD ** stop
- MPI_Status * status
- MPI_Status * retstat
- MPI_Request * request
- MPI_Datatype * type
- int * index
- int * tag
- int * from
- int numbufs
- int active

3.59.1 Detailed Description

A struct for nonblocking, unbuffered send of the sorted terms in the PObuffers back to the master using several "rotating" PObuffers.

Definition at line 146 of file parallel.h.

3.59.2 Field Documentation

3.59.2.1 buff

```
WORD** PF_BUFFER::buff
```

Definition at line 147 of file parallel.h.

3.59.2.2 fill

```
WORD** PF_BUFFER::fill
```

Definition at line 148 of file parallel.h.

3.59.2.3 full

```
WORD** PF_BUFFER::full
```

Definition at line 149 of file parallel.h.

3.59.2.4 stop

```
WORD** PF_BUFFER::stop
```

Definition at line 150 of file parallel.h.

3.59.2.5 status

```
MPI_Status* PF_BUFFER::status
```

Definition at line 151 of file parallel.h.

3.59.2.6 retstat

```
MPI_Status* PF_BUFFER::retstat
```

Definition at line 152 of file parallel.h.

3.59.2.7 request

```
MPI_Request* PF_BUFFER::request
```

Definition at line 153 of file parallel.h.

3.59.2.8 type

```
MPI_Datatype* PF_BUFFER::type
```

Definition at line 154 of file parallel.h.

3.59.2.9 index

```
int* PF_BUFFER::index
```

Definition at line 155 of file parallel.h.

3.59.2.10 tag

```
int* PF_BUFFER::tag
```

Definition at line 156 of file parallel.h.

3.59.2.11 from

```
int* PF_BUFFER::from
```

Definition at line 157 of file parallel.h.

3.59.2.12 numbufs

```
int PF_BUFFER::numbufs
```

Definition at line 158 of file parallel.h.

3.59.2.13 active

```
int PF_BUFFER::active
```

Definition at line 159 of file parallel.h.

The documentation for this struct was generated from the following file:

- parallel.h

3.60 poly Class Reference

Public Member Functions

- **poly** (PHEAD int, WORD=-1, WORD=1)
- **poly** (PHEAD const UWORD ∗, WORD, WORD=-1, WORD=1)
- **poly** (const poly &, WORD=-1, WORD=1)
- poly & **operator+=** (const poly &)
- poly & **operator-=** (const poly &)
- poly & **operator∗=** (const poly &)
- poly & **operator/=** (const poly &)
- poly & **operator%=** (const poly &)
- const poly **operator+** (const poly &) const
- const poly **operator-** (const poly &) const
- const poly **operator∗** (const poly &) const
- const poly **operator/** (const poly &) const
- const poly **operator%** (const poly &) const
- bool **operator==** (const poly &) const
- bool **operator!=** (const poly &) const
- poly & **operator=** (const poly &)
- WORD & operator[ ] (int)
- const WORD & operator[ ] (int) const
- void termscopy (const WORD ∗, int, int)
- void check_memory (int)
- void **expand_memory** (int)
- bool **is_zero** () const
- bool **is_one** () const
- bool **is_integer** () const
- bool **is_monomial** () const
- int **is_dense_univariate** () const
- int **sign** () const
- int **degree** (int) const
- int **total_degree** () const
- int **first_variable** () const
- int **number_of_terms** () const
- const std::vector< int > **all_variables** () const
- const poly **integer_lcoeff** () const
- const poly **lcoeff_univar** (int) const
- const poly **lcoeff_multivar** (int) const
- const poly **coefficient** (int, int) const
- const poly **derivative** (int) const
- void **setmod** (WORD, WORD=1)

- void **coefficients_modulo** (UWORD ∗, WORD, bool)
- int **size_of_form_notation** () const
- int **size_of_form_notation_with_den** (WORD) const
- const poly & **normalize** ()
- const std::string **to_string** () const
- WORD **last_monomial_index** () const
- WORD ∗ **last_monomial** () const
- int **compare_degree_vector** (const poly &) const
- std::vector< int > **degree_vector** () const
- int **compare_degree_vector** (const std::vector< int > &) const

Static Public Member Functions

- static const poly **simple_poly** (PHEAD int, int=0, int=1, int=0, int=1)
- static const poly **simple_poly** (PHEAD int, const poly &, int=1, int=0, int=1)
- static void **get_variables** (PHEAD std::vector< WORD ∗ >, bool, bool)
- static const poly **argument_to_poly** (PHEAD WORD ∗, bool, bool, poly ∗den=NULL)
- static void **poly_to_argument** (const poly &, WORD ∗, bool)
- static void **poly_to_argument_with_den** (const poly &, WORD, const UWORD ∗, WORD ∗, bool)
- static const poly **from_coefficient_list** (PHEAD const std::vector< WORD > &, int, WORD)
- static const std::vector< WORD > **to_coefficient_list** (const poly &)
- static const std::vector< WORD > **coefficient_list_divmod** (const std::vector< WORD > &, const std↩ ::vector< WORD > &, WORD, int)
- static int **monomial_compare** (PHEAD const WORD ∗, const WORD ∗)
- static void **add** (const poly &, const poly &, poly &)
- static void **sub** (const poly &, const poly &, poly &)
- static void **mul** (const poly &, const poly &, poly &)
- static void **div** (const poly &, const poly &, poly &)
- static void **mod** (const poly &, const poly &, poly &)
- static void **divmod** (const poly &, const poly &, poly &, poly &, bool only_divides)
- static bool **divides** (const poly &, const poly &)
- static void **mul_one_term** (const poly &, const poly &, poly &)
- static void **mul_univar** (const poly &, const poly &, poly &, int)
- static void **mul_heap** (const poly &, const poly &, poly &)
- static void **divmod_one_term** (const poly &, const poly &, poly &, poly &, bool)
- static void **divmod_univar** (const poly &, const poly &, poly &, poly &, int, bool)
- static void **divmod_heap** (const poly &, const poly &, poly &, poly &, bool)
- static void **push_heap** (PHEAD WORD ∗∗, int)
- static void **pop_heap** (PHEAD WORD ∗∗, int)

Data Fields

- WORD ∗ terms
- LONG size_of_terms
- WORD modp
- WORD modn

3.60.1 Detailed Description

Definition at line 49 of file poly.h.

3.60.2 Member Function Documentation

3.60.2.1 operator[]() [1/2]

```
WORD & poly::operator[] (
            int i )  [inline]
```

Definition at line 201 of file poly.h.

3.60.2.2 operator[]() [2/2]

```
const WORD & poly::operator[] (
            int i ) const  [inline]
```

Definition at line 205 of file poly.h.

3.60.2.3 termscopy()

```
void poly::termscopy (
            const WORD * source,
            int dest,
            int num )  [inline]
```

Definition at line 212 of file poly.h.

3.60.2.4 check_memory()

```
void poly::check_memory (
            int i )  [inline]
```

Definition at line 194 of file poly.h.

3.60.3 Field Documentation

3.60.3.1 terms

```
WORD* poly::terms
```

Definition at line 58 of file poly.h.

3.60.3.2 size_of_terms

```
LONG poly::size_of_terms
```

Definition at line 59 of file poly.h.

3.60.3.3 modp

```
WORD poly::modp
```

Definition at line 60 of file poly.h.

3.60.3.4 modn

```
WORD poly::modn
```

Definition at line 60 of file poly.h.

The documentation for this class was generated from the following file:

- poly.h

3.61 POLYMOD Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * coefs
- WORD numsym
- WORD arraysize
- WORD polysize
- WORD modnum

3.61.1 Detailed Description

The POLYMOD struct controls one univariate polynomial of which the coefficients have been taken modulus a (prime) number that fits inside a variable of type WORD. The polynomial is stored as an array of coefficients of size WORD.

Definition at line 1198 of file structs.h.

3.61.2 Field Documentation

3.61.2.1 coefs

```
WORD* POLYMOD::coefs
```

Definition at line 1199 of file structs.h.

3.61.2.2 numsym

```
WORD POLYMOD::numsym
```

Definition at line 1200 of file structs.h.

3.61.2.3 arraysize

```
WORD POLYMOD::arraysize
```

Definition at line 1201 of file structs.h.

3.61.2.4 polysize

```
WORD POLYMOD::polysize
```

Definition at line 1202 of file structs.h.

3.61.2.5 modnum

```
WORD POLYMOD::modnum
```

Definition at line 1203 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.62 PoSiTiOn Struct Reference

Data Fields

- off_t p1

3.62.1 Detailed Description

Definition at line 58 of file structs.h.

3.62.2 Field Documentation

3.62.2.1 p1

```
off_t PoSiTiOn::p1
```

Definition at line 59 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.63 PRELOAD Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE * buffer
- LONG size

3.63.1 Detailed Description

Used by the preprocessor to load the contents of a doloop or a procedure. The struct PRELOAD is used both in the DOLOOP and PROCEDURE structs.

Definition at line 811 of file structs.h.

3.63.2 Field Documentation

3.63.2.1 buffer

```
UBYTE* PRELOAD::buffer
```

Definition at line 812 of file structs.h.

3.63.2.2 size

```
LONG PRELOAD::size
```

Definition at line 813 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.64 pReVaR Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE * name
- UBYTE * value
- UBYTE * argnames
- int nargs
- int wildarg

3.64.1 Detailed Description

An element of the type PREVAR is needed for each preprocessor variable.

Definition at line 779 of file structs.h.

3.64.2 Field Documentation

3.64.2.1 name

```
UBYTE* pReVaR::name
```

allocated

Definition at line 780 of file structs.h.

Referenced by PutPreVar().

3.64.2.2 value

```
UBYTE* pReVaR::value
```

points into memory of name

Definition at line 781 of file structs.h.

Referenced by PutPreVar().

3.64.2.3 argnames

```
UBYTE* pReVaR::argnames
```

names of arguments, zero separated. points into memory of name

Definition at line 782 of file structs.h.

Referenced by PutPreVar().

3.64.2.4 nargs

```
int pReVaR::nargs
```

0 = regular, >= 1: number of macro arguments. total number

Definition at line 783 of file structs.h.

Referenced by PutPreVar().

3.64.2.5 wildarg

```
int pReVaR::wildarg
```

The number of a potential ?var. If none: 0. wildarg<nargs

Definition at line 784 of file structs.h.

Referenced by PutPreVar().

The documentation for this struct was generated from the following file:

- structs.h

3.65 PROCEDURE Struct Reference

```
#include <structs.h>
```

Collaboration diagram for PROCEDURE:

Data Fields

- PRELOAD p
- UBYTE * name
- int loadmode

3.65.1 Detailed Description

An element of the type PROCEDURE is needed for each procedure in the system.

Definition at line 821 of file structs.h.

3.65.2 Field Documentation

3.65.2.1 p

```
PRELOAD PROCEDURE::p
```

Definition at line 822 of file structs.h.

3.65.2.2 name

```
UBYTE* PROCEDURE::name
```

Definition at line 823 of file structs.h.

3.65.2.3 loadmode

```
int PROCEDURE::loadmode
```

Definition at line 824 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.66 R_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for R_const:

Public Member Functions

- **PADPOSITION** (8, 7, 7, 24, 0)

Data Fields

- FILEDATA StoreData
- FILEHANDLE Fscr [3]
- FILEHANDLE FoStage4 [2]
- POSITION DefPosition
- FILEHANDLE * infile
- FILEHANDLE * outfile
- FILEHANDLE * hidefile
- WORD * CompressBuffer
- WORD * ComprTop
- WORD * CompressPointer
- VOID * CompareRoutine
- ULONG * wranfia
- LONG OldTime
- LONG InInBuf

- LONG InHiBuf
- LONG pWorkSize
- LONG lWorkSize
- LONG posWorkSize
- ULONG wranfseed
- int NoCompress
- int gzipCompress
- int Cnumlhs
- int outtohide
- int wranfcall
- int wranfnpair1
- int wranfnpair2
- WORD GetFile
- WORD KeptInHold
- WORD BracketOn
- WORD MaxBracket
- WORD CurDum
- WORD DeferFlag
- WORD TePos
- WORD sLevel
- WORD Stage4Name
- WORD GetOneFile
- WORD PolyFun
- WORD PolyFunInv
- WORD PolyFunType
- WORD PolyFunExp
- WORD PolyFunVar
- WORD PolyFunPow
- WORD Eside
- WORD MaxDum
- WORD level
- WORD expchanged
- WORD expflags
- WORD CurExpr
- WORD SortType
- WORD ShortSortCount

3.66.1 Detailed Description

The R_const struct is part of the global data and resides either in the ALLGLOBALS struct A, or the ALLPRIVATES struct B (TFORM) under the name R We see it used with the macro AR as in AR.infile It has the variables that define the running environment and that should be transferred with a term in a multithreaded run.

Definition at line 1861 of file structs.h.

3.66.2 Field Documentation

3.66.2.1 StoreData

```
FILEDATA R_const::StoreData
```

Definition at line 1862 of file structs.h.

3.66.2.2 Fscr

```
FILEHANDLE R_const::Fscr[3]
```

Definition at line 1863 of file structs.h.

3.66.2.3 FoStage4

```
FILEHANDLE R_const::FoStage4[2]
```

Definition at line 1864 of file structs.h.

3.66.2.4 DefPosition

```
POSITION R_const::DefPosition
```

Definition at line 1865 of file structs.h.

3.66.2.5 infile

```
FILEHANDLE* R_const::infile
```

Definition at line 1866 of file structs.h.

3.66.2.6 outfile

```
FILEHANDLE* R_const::outfile
```

Definition at line 1867 of file structs.h.

3.66.2.7 hidefile

```
FILEHANDLE* R_const::hidefile
```

Definition at line 1868 of file structs.h.

3.66.2.8 CompressBuffer

```
WORD* R_const::CompressBuffer
```

Definition at line 1870 of file structs.h.

3.66.2.9 ComprTop

```
WORD* R_const::ComprTop
```

Definition at line 1871 of file structs.h.

3.66.2.10 CompressPointer

```
WORD* R_const::CompressPointer
```

Definition at line 1872 of file structs.h.

3.66.2.11 CompareRoutine

```
VOID* R_const::CompareRoutine
```

Definition at line 1873 of file structs.h.

3.66.2.12 wranfia

```
ULONG* R_const::wranfia
```

Definition at line 1874 of file structs.h.

3.66.2.13 OldTime

```
LONG R_const::OldTime
```

Definition at line 1876 of file structs.h.

3.66.2.14 InInBuf

```
LONG R_const::InInBuf
```

Definition at line 1877 of file structs.h.

3.66.2.15 InHiBuf

```
LONG R_const::InHiBuf
```

Definition at line 1878 of file structs.h.

3.66.2.16 pWorkSize

```
LONG R_const::pWorkSize
```

Definition at line 1879 of file structs.h.

3.66.2.17 lWorkSize

```
LONG R_const::lWorkSize
```

Definition at line 1880 of file structs.h.

3.66.2.18 posWorkSize

```
LONG R_const::posWorkSize
```

Definition at line 1881 of file structs.h.

3.66.2.19 wranfseed

```
ULONG R_const::wranfseed
```

Definition at line 1882 of file structs.h.

3.66.2.20 NoCompress

```
int R_const::NoCompress
```

Definition at line 1883 of file structs.h.

3.66.2.21 gzipCompress

```
int R_const::gzipCompress
```

Definition at line 1884 of file structs.h.

3.66.2.22 Cnumlhs

```
int R_const::Cnumlhs
```

Definition at line 1885 of file structs.h.

3.66.2.23 outtohide

```
int R_const::outtohide
```

Definition at line 1886 of file structs.h.

3.66.2.24 wranfcall

```
int R_const::wranfcall
```

Definition at line 1890 of file structs.h.

3.66.2.25 wranfnpair1

```
int R_const::wranfnpair1
```

Definition at line 1891 of file structs.h.

3.66.2.26 wranfnpair2

```
int R_const::wranfnpair2
```

Definition at line 1892 of file structs.h.

3.66.2.27 GetFile

```
WORD R_const::GetFile
```

Definition at line 1898 of file structs.h.

3.66.2.28 KeptInHold

```
WORD R_const::KeptInHold
```

Definition at line 1899 of file structs.h.

3.66.2.29 BracketOn

```
WORD R_const::BracketOn
```

Definition at line 1900 of file structs.h.

3.66.2.30 MaxBracket

```
WORD R_const::MaxBracket
```

Definition at line 1901 of file structs.h.

3.66.2.31 CurDum

```
WORD R_const::CurDum
```

Definition at line 1902 of file structs.h.

3.66.2.32 DeferFlag

```
WORD R_const::DeferFlag
```

Definition at line 1903 of file structs.h.

3.66.2.33 TePos

```
WORD R_const::TePos
```

Definition at line 1904 of file structs.h.

3.66.2.34 sLevel

```
WORD R_const::sLevel
```

Definition at line 1905 of file structs.h.

3.66.2.35 Stage4Name

```
WORD R_const::Stage4Name
```

Definition at line 1906 of file structs.h.

3.66.2.36 GetOneFile

```
WORD R_const::GetOneFile
```

Definition at line 1907 of file structs.h.

3.66.2.37 PolyFun

```
WORD R_const::PolyFun
```

Definition at line 1908 of file structs.h.

3.66.2.38 PolyFunInv

```
WORD R_const::PolyFunInv
```

Definition at line 1909 of file structs.h.

3.66.2.39 PolyFunType

```
WORD R_const::PolyFunType
```

Definition at line 1910 of file structs.h.

3.66.2.40 PolyFunExp

```
WORD R_const::PolyFunExp
```

Definition at line 1911 of file structs.h.

3.66.2.41 PolyFunVar

```
WORD R_const::PolyFunVar
```

Definition at line 1912 of file structs.h.

3.66.2.42 PolyFunPow

```
WORD R_const::PolyFunPow
```

Definition at line 1913 of file structs.h.

3.66.2.43 Eside

```
WORD R_const::Eside
```

Definition at line 1914 of file structs.h.

3.66.2.44 MaxDum

```
WORD R_const::MaxDum
```

Definition at line 1915 of file structs.h.

3.66.2.45 level

```
WORD R_const::level
```

Definition at line 1916 of file structs.h.

3.66.2.46 expchanged

```
WORD R_const::expchanged
```

Definition at line 1917 of file structs.h.

3.66.2.47 expflags

```
WORD R_const::expflags
```

Definition at line 1918 of file structs.h.

3.66.2.48 CurExpr

```
WORD R_const::CurExpr
```

Definition at line 1919 of file structs.h.

3.66.2.49 SortType

```
WORD R_const::SortType
```

Definition at line 1920 of file structs.h.

3.66.2.50 ShortSortCount

```
WORD R_const::ShortSortCount
```

Definition at line 1921 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.67 ReNuMbEr Struct Reference

```
#include <structs.h>
```

Collaboration diagram for ReNuMbEr:

Public Member Functions

- **PADPOSITION** (4, 0, 0, 0, sizeof(VARRENUM) ∗4)

Data Fields

- POSITION startposition
- VARRENUM symb
- VARRENUM indi
- VARRENUM vect
- VARRENUM func
- WORD ∗ symnum
- WORD ∗ indnum
- WORD ∗ vecnum
- WORD ∗ funnum

3.67.1 Detailed Description

Only symb.lo gets dynamically allocated. All other pointers points into this memory.

Definition at line 177 of file structs.h.

3.67.2 Field Documentation

3.67.2.1 startposition

```
POSITION ReNuMbEr::startposition
```

Definition at line 178 of file structs.h.

3.67.2.2 symb

```
VARRENUM ReNuMbEr::symb
```

Symbols

Definition at line 180 of file structs.h.

Referenced by DoRecovery(), Generator(), and TermRenumber().

3.67.2.3 indi

```
VARRENUM ReNuMbEr::indi
```

Indices

Definition at line 181 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.67.2.4 vect

```
VARRENUM ReNuMbEr::vect
```

Vectors

Definition at line 182 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.67.2.5 func

```
VARRENUM ReNuMbEr::func
```

Functions

Definition at line 183 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.67.2.6 symnum

```
WORD* ReNuMbEr::symnum
```

Renumbered symbols

Definition at line 185 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.67.2.7 indnum

```
WORD* ReNuMbEr::indnum
```

Renumbered indices

Definition at line 186 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.67.2.8 vecnum

```
WORD* ReNuMbEr::vecnum
```

Renumbered vectors

Definition at line 187 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

3.67.2.9 funnum

```
WORD* ReNuMbEr::funnum
```

Renumbered functions

Definition at line 188 of file structs.h.

Referenced by DoRecovery(), and TermRenumber().

The documentation for this struct was generated from the following file:

- structs.h

3.68 S_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for S_const:

Public Member Functions

- **PADPOSITION** (3, 0, 4, 2, 0)

Data Fields

- POSITION MaxExprSize
- POSITION ∗ OldOnFile
- WORD ∗ OldNumFactors
- WORD ∗ Oldvflags
- int NumOldOnFile
- int NumOldNumFactors
- int MultiThreaded
- int Balancing
- WORD ExecMode
- WORD CollectOverFlag

3.68.1 Detailed Description

The S_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name S We see it used with the macro AS as in AS.ExecMode It has some variables used by the master in multithreaded runs

Definition at line 1813 of file structs.h.

3.68.2 Field Documentation

3.68.2.1 MaxExprSize

```
POSITION S_const::MaxExprSize
```

Definition at line 1814 of file structs.h.

3.68.2.2 OldOnFile

```
POSITION* S_const::OldOnFile
```

Definition at line 1820 of file structs.h.

3.68.2.3 OldNumFactors

```
WORD* S_const::OldNumFactors
```

Definition at line 1821 of file structs.h.

3.68.2.4 Oldvflags

```
WORD* S_const::Oldvflags
```

Definition at line 1822 of file structs.h.

3.68.2.5 NumOldOnFile

```
int S_const::NumOldOnFile
```

Definition at line 1823 of file structs.h.

3.68.2.6 NumOldNumFactors

```
int S_const::NumOldNumFactors
```

Definition at line 1824 of file structs.h.

3.68.2.7 MultiThreaded

```
int S_const::MultiThreaded
```

Definition at line 1825 of file structs.h.

3.68.2.8 Balancing

```
int S_const::Balancing
```

Definition at line 1832 of file structs.h.

3.68.2.9 ExecMode

```
WORD S_const::ExecMode
```

Definition at line 1833 of file structs.h.

3.68.2.10 CollectOverFlag

```
WORD S_const::CollectOverFlag
```

Definition at line 1835 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.69 SeTs Struct Reference

Data Fields

- LONG name
- WORD type
- WORD first
- WORD last
- WORD node
- WORD namesize
- WORD dimension

3.69.1 Detailed Description

Definition at line 485 of file structs.h.

3.69.2 Field Documentation

3.69.2.1 name

```
LONG SeTs::name
```

Definition at line 486 of file structs.h.

3.69.2.2 type

```
WORD SeTs::type
```

Definition at line 487 of file structs.h.

3.69.2.3 first

```
WORD SeTs::first
```

Definition at line 488 of file structs.h.

3.69.2.4 last

```
WORD SeTs::last
```

Definition at line 489 of file structs.h.

3.69.2.5 node

```
WORD SeTs::node
```

Definition at line 490 of file structs.h.

3.69.2.6 namesize

```
WORD SeTs::namesize
```

Definition at line 491 of file structs.h.

3.69.2.7 dimension

```
WORD SeTs::dimension
```

Definition at line 492 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.70 SETUPPARAMETERS Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE * parameter
- int type
- int flags
- LONG value

3.70.1 Detailed Description

Each setup parameter has one element of the struct SETUPPARAMETERS assigned to it. By binary search in the array of them we can then locate the proper element by name. We have to assume that two ints make a long and either one or two longs make a pointer. The long before the ints and padding gives a problem in the initialization.

Definition at line 967 of file structs.h.

3.70.2 Field Documentation

3.70.2.1 parameter

```
UBYTE* SETUPPARAMETERS::parameter
```

Definition at line 968 of file structs.h.

3.70.2.2 type

```
int SETUPPARAMETERS::type
```

Definition at line 969 of file structs.h.

3.70.2.3 flags

```
int SETUPPARAMETERS::flags
```

Definition at line 970 of file structs.h.

3.70.2.4 value

```
LONG SETUPPARAMETERS::value
```

Definition at line 971 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.71 SHvariables Struct Reference

Data Fields

- WORD * outterm
- WORD * outfun
- WORD * incoef
- WORD * stop1
- WORD * stop2
- WORD * ststop1
- WORD * ststop2
- void * finishuf
- void * do_uffle
- LONG combilast
- WORD nincoef
- WORD level
- WORD thefunction
- WORD option

3.71.1 Detailed Description

Definition at line 1206 of file structs.h.

3.71.2 Field Documentation

3.71.2.1 outterm

```
WORD* SHvariables::outterm
```

Definition at line 1207 of file structs.h.

3.71.2.2 outfun

```
WORD* SHvariables::outfun
```

Definition at line 1208 of file structs.h.

3.71.2.3 incoef

```
WORD* SHvariables::incoef
```

Definition at line 1209 of file structs.h.

3.71.2.4 stop1

```
WORD* SHvariables::stop1
```

Definition at line 1210 of file structs.h.

3.71.2.5 stop2

```
WORD* SHvariables::stop2
```

Definition at line 1211 of file structs.h.

3.71.2.6 ststop1

```
WORD* SHvariables::ststop1
```

Definition at line 1212 of file structs.h.

3.71.2.7 ststop2

```
WORD* SHvariables::ststop2
```

Definition at line 1213 of file structs.h.

3.71.2.8 finishuf

```
void* SHvariables::finishuf
```

Definition at line 1214 of file structs.h.

3.71.2.9 do_uffle

```
void* SHvariables::do_uffle
```

Definition at line 1215 of file structs.h.

3.71.2.10 combilast

```
LONG SHvariables::combilast
```

Definition at line 1216 of file structs.h.

3.71.2.11 nincoef

```
WORD SHvariables::nincoef
```

Definition at line 1217 of file structs.h.

3.71.2.12 level

```
WORD SHvariables::level
```

Definition at line 1218 of file structs.h.

3.71.2.13 thefunction

```
WORD SHvariables::thefunction
```

Definition at line 1219 of file structs.h.

3.71.2.14 option

```
WORD SHvariables::option
```

Definition at line 1220 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.72 sOrT Struct Reference

```
#include <structs.h>
```

Collaboration diagram for sOrT:

Public Member Functions

- **PADPOSITION** (25, 12, 12, 3, 0)

Data Fields

- FILEHANDLE file
- POSITION SizeInFile [3]
- WORD * lBuffer
- WORD * lTop
- WORD * lFill
- WORD * used
- WORD * sBuffer
- WORD * sTop
- WORD * sTop2
- WORD * sHalf
- WORD * sFill
- WORD ** sPointer
- WORD ** PoinFill
- WORD ** SplitScratch
- WORD * cBuffer
- WORD ** Patches
- WORD ** pStop
- WORD ** poina
- WORD ** poin2a
- WORD * ktoi
- WORD * tree
- POSITION * fPatches
- POSITION * inPatches
- POSITION * fPatchesStop
- POSITION * iPatches
- FILEHANDLE * f
- FILEHANDLE ** ff
- LONG sTerms
- LONG LargeSize
- LONG SmallSize
- LONG SmallEsize
- LONG TermsInSmall
- LONG Terms2InSmall
- LONG GenTerms
- LONG TermsLeft
- LONG GenSpace
- LONG SpaceLeft
- LONG putinsize
- LONG ninterms
- int MaxPatches
- int MaxFpatches
- int type
- int lPatch
- int fPatchN1
- int PolyWise
- int PolyFlag
- int cBufferSize
- int maxtermsize
- int newmaxtermsize
- int outputmode
- int stagelevel
- WORD fPatchN
- WORD inNum
- WORD stage4

3.72.1 Detailed Description

The struct SORTING is used to control a sort operation. It includes a small and a large buffer and arrays for keeping track of various stages of the (merge) sorts. Each sort level has its own struct and different levels can have different sizes for its arrays. Also different threads have their own set of SORTING structs.

Definition at line 1069 of file structs.h.

3.72.2 Field Documentation

3.72.2.1 file

```
FILEHANDLE sOrT::file
```

Definition at line 1070 of file structs.h.

3.72.2.2 SizeInFile

```
POSITION sOrT::SizeInFile[3]
```

Definition at line 1071 of file structs.h.

3.72.2.3 lBuffer

```
WORD* sOrT::lBuffer
```

Definition at line 1072 of file structs.h.

3.72.2.4 lTop

```
WORD* sOrT::lTop
```

Definition at line 1073 of file structs.h.

3.72.2.5 lFill

```
WORD* sOrT::lFill
```

Definition at line 1074 of file structs.h.

3.72.2.6 used

```
WORD* sOrT::used
```

Definition at line 1075 of file structs.h.

3.72.2.7 sBuffer

```
WORD* sOrT::sBuffer
```

Definition at line 1076 of file structs.h.

3.72.2.8 sTop

```
WORD* sOrT::sTop
```

Definition at line 1077 of file structs.h.

3.72.2.9 sTop2

```
WORD* sOrT::sTop2
```

Definition at line 1078 of file structs.h.

3.72.2.10 sHalf

```
WORD* sOrT::sHalf
```

Definition at line 1079 of file structs.h.

3.72.2.11 sFill

```
WORD* sOrT::sFill
```

Definition at line 1080 of file structs.h.

3.72.2.12 sPointer

```
WORD** sOrT::sPointer
```

Definition at line 1081 of file structs.h.

3.72.2.13 PoinFill

```
WORD** sOrT::PoinFill
```

Definition at line 1082 of file structs.h.

3.72.2.14 SplitScratch

```
WORD** sOrT::SplitScratch
```

Definition at line 1083 of file structs.h.

3.72.2.15 cBuffer

```
WORD* sOrT::cBuffer
```

Definition at line 1084 of file structs.h.

3.72.2.16 Patches

```
WORD** sOrT::Patches
```

Definition at line 1085 of file structs.h.

3.72.2.17 pStop

```
WORD** sOrT::pStop
```

Definition at line 1086 of file structs.h.

3.72.2.18 poina

```
WORD** sOrT::poina
```

Definition at line 1087 of file structs.h.

3.72.2.19 poin2a

```
WORD** sOrT::poin2a
```

Definition at line 1088 of file structs.h.

3.72.2.20 ktoi

```
WORD* sOrT::ktoi
```

Definition at line 1089 of file structs.h.

3.72.2.21 tree

```
WORD* sOrT::tree
```

Definition at line 1090 of file structs.h.

3.72.2.22 fPatches

```
POSITION* sOrT::fPatches
```

Definition at line 1096 of file structs.h.

3.72.2.23 inPatches

```
POSITION* sOrT::inPatches
```

Definition at line 1097 of file structs.h.

3.72.2.24 fPatchesStop

```
POSITION* sOrT::fPatchesStop
```

Definition at line 1098 of file structs.h.

3.72.2.25 iPatches

```
POSITION* sOrT::iPatches
```

Definition at line 1099 of file structs.h.

3.72.2.26 f

```
FILEHANDLE* sOrT::f
```

Definition at line 1100 of file structs.h.

3.72.2.27 ff

```
FILEHANDLE** sOrT::ff
```

Definition at line 1101 of file structs.h.

3.72.2.28 sTerms

```
LONG sOrT::sTerms
```

Definition at line 1102 of file structs.h.

3.72.2.29 LargeSize

```
LONG sOrT::LargeSize
```

Definition at line 1103 of file structs.h.

3.72.2.30 SmallSize

```
LONG sOrT::SmallSize
```

Definition at line 1104 of file structs.h.

3.72.2.31 SmallEsize

```
LONG sOrT::SmallEsize
```

Definition at line 1105 of file structs.h.

3.72.2.32 TermsInSmall

```
LONG sOrT::TermsInSmall
```

Definition at line 1106 of file structs.h.

3.72.2.33 Terms2InSmall

```
LONG sOrT::Terms2InSmall
```

Definition at line 1107 of file structs.h.

3.72.2.34 GenTerms

```
LONG sOrT::GenTerms
```

Definition at line 1108 of file structs.h.

3.72.2.35 TermsLeft

```
LONG sOrT::TermsLeft
```

Definition at line 1109 of file structs.h.

3.72.2.36 GenSpace

```
LONG sOrT::GenSpace
```

Definition at line 1110 of file structs.h.

3.72.2.37 SpaceLeft

```
LONG sOrT::SpaceLeft
```

Definition at line 1111 of file structs.h.

3.72.2.38 putinsize

```
LONG sOrT::putinsize
```

Definition at line 1112 of file structs.h.

3.72.2.39 ninterms

```
LONG sOrT::ninterms
```

Definition at line 1113 of file structs.h.

3.72.2.40 MaxPatches

```
int sOrT::MaxPatches
```

Definition at line 1114 of file structs.h.

3.72.2.41 MaxFpatches

```
int sOrT::MaxFpatches
```

Definition at line 1115 of file structs.h.

3.72.2.42 type

```
int sOrT::type
```

Definition at line 1116 of file structs.h.

3.72.2.43 lPatch

```
int sOrT::lPatch
```

Definition at line 1117 of file structs.h.

3.72.2.44 fPatchN1

```
int sOrT::fPatchN1
```

Definition at line 1118 of file structs.h.

3.72.2.45 PolyWise

```
int sOrT::PolyWise
```

Definition at line 1119 of file structs.h.

3.72.2.46 PolyFlag

```
int sOrT::PolyFlag
```

Definition at line 1120 of file structs.h.

3.72.2.47 cBufferSize

```
int sOrT::cBufferSize
```

Definition at line 1121 of file structs.h.

3.72.2.48 maxtermsize

```
int sOrT::maxtermsize
```

Definition at line 1122 of file structs.h.

3.72.2.49 newmaxtermsize

```
int sOrT::newmaxtermsize
```

Definition at line 1123 of file structs.h.

3.72.2.50 outputmode

```
int sOrT::outputmode
```

Definition at line 1124 of file structs.h.

3.72.2.51 stagelevel

```
int sOrT::stagelevel
```

Definition at line 1125 of file structs.h.

3.72.2.52 fPatchN

```
WORD sOrT::fPatchN
```

Definition at line 1126 of file structs.h.

3.72.2.53 inNum

```
WORD sOrT::inNum
```

Definition at line 1127 of file structs.h.

3.72.2.54 stage4

```
WORD sOrT::stage4
```

Definition at line 1128 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.73 SpecTatoR Struct Reference

Collaboration diagram for SpecTatoR:

Public Member Functions

- **PADPOSITION** (2, 0, 0, 2, 0)

Data Fields

- POSITION position
- POSITION readpos
- FILEHANDLE * fh
- char * name
- WORD exprnumber
- WORD flags

3.73.1 Detailed Description

Definition at line 703 of file structs.h.

3.73.2 Field Documentation

3.73.2.1 position

```
POSITION SpecTatoR::position
```

Definition at line 704 of file structs.h.

3.73.2.2 readpos

```
POSITION SpecTatoR::readpos
```

Definition at line 705 of file structs.h.

3.73.2.3 fh

```
FILEHANDLE* SpecTatoR::fh
```

Definition at line 706 of file structs.h.

3.73.2.4 name

```
char* SpecTatoR::name
```

Definition at line 707 of file structs.h.

3.73.2.5 exprnumber

```
WORD SpecTatoR::exprnumber
```

Definition at line 708 of file structs.h.

3.73.2.6 flags

```
WORD SpecTatoR::flags
```

Definition at line 709 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.74 StOrEcAcHe Struct Reference

```
#include <structs.h>
```

Collaboration diagram for StOrEcAcHe:

Public Member Functions

- **PADPOSITION** (1, 0, 0, 2, 0)

Data Fields

- POSITION position
- POSITION toppos
- struct StOrEcAcHe * next
- WORD buffer [2]

3.74.1 Detailed Description

The struct of type STORECACHE is used by a caching system for reading terms from stored expressions. Each thread should have its own system of caches.

Definition at line 994 of file structs.h.

3.74.2 Field Documentation

3.74.2.1 position

```
POSITION StOrEcAcHe::position
```

Definition at line 995 of file structs.h.

3.74.2.2 toppos

```
POSITION StOrEcAcHe::toppos
```

Definition at line 996 of file structs.h.

3.74.2.3 next

```
struct StOrEcAcHe* StOrEcAcHe::next
```

Definition at line 997 of file structs.h.

3.74.2.4 buffer

```
WORD StOrEcAcHe::buffer[2]
```

Definition at line 998 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.75 STOREHEADER Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE headermark [8]
- UBYTE lenWORD
- UBYTE lenLONG
- UBYTE lenPOS
- UBYTE lenPOINTER
- UBYTE endianness [16]
- UBYTE sSym
- UBYTE sInd
- UBYTE sVec
- UBYTE sFun
- UBYTE maxpower [16]
- UBYTE wildoffset [16]
- UBYTE revision
- UBYTE reserved [512-8-4-16-4-16-16-1]

3.75.1 Detailed Description

Defines the structure of the file header for store-files and save-files.

The first 8 bytes serve as a unique mark to identity save-files that contain such a header. Older versions of FORM don't have this header and will write the POSITION of the next file index (struct FiLeInDeX) here, which is always different from this pattern.

It is always 512 bytes long.

Definition at line 74 of file structs.h.

3.75.2 Field Documentation

3.75.2.1 headermark

```
UBYTE STOREHEADER::headermark[8]
```

Pattern for header identification. Old versions of FORM have a maximum sizeof(POSITION) of 8

Definition at line 75 of file structs.h.

3.75.2.2 lenWORD

```
UBYTE STOREHEADER::lenWORD
```

Number of bytes for WORD

Definition at line 77 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.3 lenLONG

```
UBYTE STOREHEADER::lenLONG
```

Number of bytes for LONG

Definition at line 78 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.4 lenPOS

```
UBYTE STOREHEADER::lenPOS
```

Number of bytes for POSITION

Definition at line 79 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.5 lenPOINTER

```
UBYTE STOREHEADER::lenPOINTER
```

Number of bytes for void *

Definition at line 80 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.6 endianness

```
UBYTE STOREHEADER::endianness[16]
```

Used to determine endianness, sizeof(int) should be <= 16

Definition at line 81 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.7 sSym

```
UBYTE STOREHEADER::sSym
```

sizeof(struct SyMbOl)

Definition at line 82 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.8 sInd

```
UBYTE STOREHEADER::sInd
```

sizeof(struct InDeX)

Definition at line 83 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.9 sVec

```
UBYTE STOREHEADER::sVec
```

sizeof(struct VeCtOr)

Definition at line 84 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.10 sFun

```
UBYTE STOREHEADER::sFun
```

sizeof(struct FuNcTiOn)

Definition at line 85 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.11 maxpower

```
UBYTE STOREHEADER::maxpower[16]
```

Maximum power, see MAXPOWER

Definition at line 86 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.12 wildoffset

```
UBYTE STOREHEADER::wildoffset[16]
```

WILDOFFSET macro

Definition at line 87 of file structs.h.

Referenced by WriteStoreHeader().

3.75.2.13 revision

```
UBYTE STOREHEADER::revision
```

Revision number of save-file system

Definition at line 88 of file structs.h.

3.75.2.14 reserved

```
UBYTE STOREHEADER::reserved[512-8-4-16-4-16-16-1]
```

Padding to 512 bytes

Definition at line 89 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.76 StreaM Struct Reference

```
#include <structs.h>
```

Public Member Functions

- **PADPOSITION** (6, 3, 9, 0, 4)

Data Fields

- off_t fileposition
- off_t linenumber
- off_t prevline
- UBYTE * buffer
- UBYTE * pointer
- UBYTE * top
- UBYTE * FoldName
- UBYTE * name
- UBYTE * pname
- LONG buffersize
- LONG bufferposition
- LONG inbuffer
- int previous
- int handle
- int type
- int prevars
- int previousNoShowInput
- int eqnum
- int afterwards
- int olddelay
- int oldnoshowinput
- UBYTE isnextchar
- UBYTE nextchar [2]
- UBYTE reserved

3.76.1 Detailed Description

Input is read from 'streams' which are represented by objects of type STREAM. A stream can be a file, a do-loop, a procedure, the string value of a preprocessor variable ..... When a new stream is opened we have to keep information about where to fall back in the parent stream to allow this to happen even in the middle of reading names etc as would be the case with a`i'b

Definition at line 674 of file structs.h.

3.76.2 Field Documentation

3.76.2.1 fileposition

```
off_t StreaM::fileposition
```

Definition at line 675 of file structs.h.

3.76.2.2 linenumber

```
off_t StreaM::linenumber
```

Definition at line 676 of file structs.h.

3.76.2.3 prevline

```
off_t StreaM::prevline
```

Definition at line 677 of file structs.h.

3.76.2.4 buffer

```
UBYTE* StreaM::buffer
```

[D] Size in buffersize

Definition at line 678 of file structs.h.

3.76.2.5 pointer

```
UBYTE* StreaM::pointer
```

pointer into buffer memory

Definition at line 679 of file structs.h.

3.76.2.6 top

```
UBYTE* StreaM::top
```

pointer into buffer memory

Definition at line 680 of file structs.h.

3.76.2.7 FoldName

```
UBYTE* StreaM::FoldName
```

[D]

Definition at line 681 of file structs.h.

3.76.2.8 name

```
UBYTE* StreaM::name
```

[D]

Definition at line 682 of file structs.h.

3.76.2.9 pname

```
UBYTE* StreaM::pname
```

for DOLLARSTREAM and PREVARSTREAM it points always to name, else it is undefined

Definition at line 683 of file structs.h.

3.76.2.10 buffersize

```
LONG StreaM::buffersize
```

Definition at line 685 of file structs.h.

3.76.2.11 bufferposition

```
LONG StreaM::bufferposition
```

Definition at line 686 of file structs.h.

3.76.2.12 inbuffer

```
LONG StreaM::inbuffer
```

Definition at line 687 of file structs.h.

3.76.2.13 previous

```
int StreaM::previous
```

Definition at line 688 of file structs.h.

3.76.2.14 handle

```
int StreaM::handle
```

Definition at line 689 of file structs.h.

3.76.2.15 type

```
int StreaM::type
```

Definition at line 690 of file structs.h.

3.76.2.16 prevars

```
int StreaM::prevars
```

Definition at line 691 of file structs.h.

3.76.2.17 previousNoShowInput

```
int StreaM::previousNoShowInput
```

Definition at line 692 of file structs.h.

3.76.2.18 eqnum

```
int StreaM::eqnum
```

Definition at line 693 of file structs.h.

3.76.2.19 afterwards

```
int StreaM::afterwards
```

Definition at line 694 of file structs.h.

3.76.2.20 olddelay

```
int StreaM::olddelay
```

Definition at line 695 of file structs.h.

3.76.2.21 oldnoshowinput

```
int StreaM::oldnoshowinput
```

Definition at line 696 of file structs.h.

3.76.2.22 isnextchar

```
UBYTE StreaM::isnextchar
```

Definition at line 697 of file structs.h.

3.76.2.23 nextchar

```
UBYTE StreaM::nextchar[2]
```

Definition at line 698 of file structs.h.

3.76.2.24 reserved

```
UBYTE StreaM::reserved
```

Definition at line 699 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.77 SuBbUf Struct Reference

Data Fields

- WORD subexpnum
- WORD buffernum

3.77.1 Detailed Description

Definition at line 233 of file compiler.c.

3.77.2 Field Documentation

3.77.2.1 subexpnum

```
WORD SuBbUf::subexpnum
```

Definition at line 234 of file compiler.c.

3.77.2.2 buffernum

```
WORD SuBbUf::buffernum
```

Definition at line 235 of file compiler.c.

The documentation for this struct was generated from the following file:

- compiler.c

3.78 SyMbOl Struct Reference

Data Fields

- LONG name
- WORD minpower
- WORD maxpower
- WORD complex
- WORD number
- WORD flags
- WORD node
- WORD namesize
- WORD dimension

3.78.1 Detailed Description

Definition at line 414 of file structs.h.

3.78.2 Field Documentation

3.78.2.1 name

```
LONG SyMbOl::name
```

Definition at line 415 of file structs.h.

3.78.2.2 minpower

```
WORD SyMbOl::minpower
```

Definition at line 416 of file structs.h.

3.78.2.3 maxpower

```
WORD SyMbOl::maxpower
```

Definition at line 417 of file structs.h.

3.78.2.4 complex

```
WORD SyMbOl::complex
```

Definition at line 418 of file structs.h.

3.78.2.5 number

```
WORD SyMbOl::number
```

Definition at line 419 of file structs.h.

3.78.2.6 flags

```
WORD SyMbOl::flags
```

Definition at line 420 of file structs.h.

3.78.2.7 node

```
WORD SyMbOl::node
```

Definition at line 421 of file structs.h.

3.78.2.8 namesize

```
WORD SyMbOl::namesize
```

Definition at line 422 of file structs.h.

3.78.2.9 dimension

```
WORD SyMbOl::dimension
```

Definition at line 423 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.79 T_const Struct Reference

```
#include <structs.h>
```

Collaboration diagram for T_const:

Data Fields

- SORTING * S0
- SORTING * SS
- NESTING Nest
- NESTING NestStop
- NESTING NestPoin
- WORD * BrackBuf
- STORECACHE StoreCache
- STORECACHE StoreCacheAlloc
- WORD ** pWorkSpace
- LONG * lWorkSpace
- POSITION * posWorkSpace
- WORD * WorkSpace
- WORD * WorkTop
- WORD * WorkPointer
- int * RepCount
- int * RepTop
- WORD * WildArgTaken
- UWORD * factorials
- WORD * small_power_n
- UWORD ** small_power
- UWORD * bernoullis
- WORD * primelist
- LONG * pfac
- LONG * pBer
- WORD * TMaddr

- WORD * WildMask
- WORD * previousEfactor
- WORD ** TermMemHeap
- UWORD ** NumberMemHeap
- UWORD ** CacheNumberMemHeap
- BRACKETINFO * bracketinfo
- WORD ** ListPoly
- WORD * ListSymbols
- UWORD * NumMem
- PARTI partitions
- LONG sBer
- LONG pWorkPointer
- LONG lWorkPointer
- LONG posWorkPointer
- LONG lnNumMem
- int sfact
- int mfac
- int ebufnum
- int fbufnum
- int allbufnum
- int aebufnum
- int idallflag
- int idallnum
- int idallmaxnum
- int WildcardBufferSize
- int TermMemMax
- int TermMemTop
- int NumberMemMax
- int NumberMemTop
- int CacheNumberMemMax
- int CacheNumberMemTop
- int bracketindexflag
- int optimtimes
- int ListSymbolsSize
- int NumListSymbols
- int numpoly
- int LeaveNegative
- int TrimPower
- WORD small_power_maxx
- WORD small_power_maxn
- WORD dummysubexp [SUBEXPSIZE+4]
- WORD comsym [8]
- WORD comnum [4]
- WORD comfun [FUNHEAD+4]
- WORD comind [7]
- WORD MinVecArg [7+ARGHEAD]
- WORD FunArg [4+ARGHEAD+FUNHEAD]
- WORD locwildvalue [SUBEXPSIZE]
- WORD mulpat [SUBEXPSIZE+5]
- WORD proexp [SUBEXPSIZE+5]
- WORD TMout [40]
- WORD TMbuff
- WORD TMdolfac
- WORD nfac
- WORD nBer

- WORD mBer
- WORD PolyAct
- WORD RecFlag
- WORD inprimelist
- WORD sizeprimelist
- WORD fromindex

3.79.1 Detailed Description

The T_const struct is part of the global data and resides either in the ALLGLOBALS struct A, or the ALLPRIVATES struct B (TFORM) under the name T We see it used with the macro AT as in AT.WorkPointer It has variables that are private to each thread, most of which have to be defined at startup.

Definition at line 1949 of file structs.h.

3.79.2 Field Documentation

3.79.2.1 S0

```
SORTING* T_const::S0
```

Definition at line 1953 of file structs.h.

3.79.2.2 SS

```
SORTING* T_const::SS
```

Definition at line 1954 of file structs.h.

3.79.2.3 Nest

```
NESTING T_const::Nest
```

Definition at line 1955 of file structs.h.

3.79.2.4 NestStop

```
NESTING T_const::NestStop
```

Definition at line 1956 of file structs.h.

3.79.2.5 NestPoin

```
NESTING T_const::NestPoin
```

Definition at line 1957 of file structs.h.

3.79.2.6 BrackBuf

```
WORD* T_const::BrackBuf
```

Definition at line 1958 of file structs.h.

3.79.2.7 StoreCache

```
STORECACHE T_const::StoreCache
```

Definition at line 1959 of file structs.h.

3.79.2.8 StoreCacheAlloc

```
STORECACHE T_const::StoreCacheAlloc
```

Definition at line 1960 of file structs.h.

3.79.2.9 pWorkSpace

```
WORD** T_const::pWorkSpace
```

Definition at line 1961 of file structs.h.

3.79.2.10 lWorkSpace

```
LONG* T_const::lWorkSpace
```

Definition at line 1962 of file structs.h.

3.79.2.11 posWorkSpace

```
POSITION* T_const::posWorkSpace
```

Definition at line 1963 of file structs.h.

3.79.2.12 WorkSpace

```
WORD* T_const::WorkSpace
```

Definition at line 1964 of file structs.h.

3.79.2.13 WorkTop

```
WORD* T_const::WorkTop
```

Definition at line 1965 of file structs.h.

3.79.2.14 WorkPointer

```
WORD* T_const::WorkPointer
```

Definition at line 1966 of file structs.h.

3.79.2.15 RepCount

```
int* T_const::RepCount
```

Definition at line 1967 of file structs.h.

3.79.2.16 RepTop

```
int* T_const::RepTop
```

Definition at line 1968 of file structs.h.

3.79.2.17 WildArgTaken

```
WORD* T_const::WildArgTaken
```

Definition at line 1969 of file structs.h.

3.79.2.18 factorials

```
UWORD* T_const::factorials
```

Definition at line 1970 of file structs.h.

3.79.2.19 small_power_n

```
WORD* T_const::small_power_n
```

Definition at line 1971 of file structs.h.

3.79.2.20 small_power

```
UWORD** T_const::small_power
```

Definition at line 1972 of file structs.h.

3.79.2.21 bernoullis

```
UWORD* T_const::bernoullis
```

Definition at line 1973 of file structs.h.

3.79.2.22 primelist

```
WORD* T_const::primelist
```

Definition at line 1974 of file structs.h.

3.79.2.23 pfac

```
LONG* T_const::pfac
```

Definition at line 1975 of file structs.h.

3.79.2.24 pBer

```
LONG* T_const::pBer
```

Definition at line 1976 of file structs.h.

3.79.2.25 TMaddr

```
WORD* T_const::TMaddr
```

Definition at line 1977 of file structs.h.

3.79.2.26 WildMask

```
WORD* T_const::WildMask
```

Definition at line 1978 of file structs.h.

3.79.2.27 previousEfactor

```
WORD* T_const::previousEfactor
```

Definition at line 1979 of file structs.h.

3.79.2.28 TermMemHeap

```
WORD** T_const::TermMemHeap
```

Definition at line 1980 of file structs.h.

3.79.2.29 NumberMemHeap

```
UWORD** T_const::NumberMemHeap
```

Definition at line 1981 of file structs.h.

3.79.2.30 CacheNumberMemHeap

```
UWORD** T_const::CacheNumberMemHeap
```

Definition at line 1982 of file structs.h.

3.79.2.31 bracketinfo

```
BRACKETINFO* T_const::bracketinfo
```

Definition at line 1983 of file structs.h.

3.79.2.32 ListPoly

```
WORD** T_const::ListPoly
```

Definition at line 1984 of file structs.h.

3.79.2.33 ListSymbols

```
WORD* T_const::ListSymbols
```

Definition at line 1985 of file structs.h.

3.79.2.34 NumMem

```
UWORD* T_const::NumMem
```

Definition at line 1986 of file structs.h.

3.79.2.35 partitions

```
PARTI T_const::partitions
```

Definition at line 1987 of file structs.h.

3.79.2.36 sBer

```
LONG T_const::sBer
```

Definition at line 1988 of file structs.h.

3.79.2.37 pWorkPointer

```
LONG T_const::pWorkPointer
```

Definition at line 1989 of file structs.h.

3.79.2.38 lWorkPointer

```
LONG T_const::lWorkPointer
```

Definition at line 1990 of file structs.h.

3.79.2.39 posWorkPointer

```
LONG T_const::posWorkPointer
```

Definition at line 1991 of file structs.h.

3.79.2.40 lnNumMem

```
LONG T_const::lnNumMem
```

Definition at line 1992 of file structs.h.

3.79.2.41 sfact

```
int T_const::sfact
```

Definition at line 1993 of file structs.h.

3.79.2.42 mfac

```
int T_const::mfac
```

Definition at line 1994 of file structs.h.

3.79.2.43 ebufnum

```
int T_const::ebufnum
```

Definition at line 1995 of file structs.h.

3.79.2.44 fbufnum

```
int T_const::fbufnum
```

Definition at line 1996 of file structs.h.

3.79.2.45 allbufnum

```
int T_const::allbufnum
```

Definition at line 1997 of file structs.h.

3.79.2.46 aebufnum

```
int T_const::aebufnum
```

Definition at line 1998 of file structs.h.

3.79.2.47 idallflag

```
int T_const::idallflag
```

Definition at line 1999 of file structs.h.

3.79.2.48 idallnum

```
int T_const::idallnum
```

Definition at line 2000 of file structs.h.

3.79.2.49 idallmaxnum

```
int T_const::idallmaxnum
```

Definition at line 2001 of file structs.h.

3.79.2.50 WildcardBufferSize

```
int T_const::WildcardBufferSize
```

Definition at line 2002 of file structs.h.

3.79.2.51 TermMemMax

```
int T_const::TermMemMax
```

Definition at line 2011 of file structs.h.

3.79.2.52 TermMemTop

```
int T_const::TermMemTop
```

Definition at line 2012 of file structs.h.

3.79.2.53 NumberMemMax

```
int T_const::NumberMemMax
```

Definition at line 2013 of file structs.h.

3.79.2.54 NumberMemTop

```
int T_const::NumberMemTop
```

Definition at line 2014 of file structs.h.

3.79.2.55 CacheNumberMemMax

```
int T_const::CacheNumberMemMax
```

Definition at line 2015 of file structs.h.

3.79.2.56 CacheNumberMemTop

```
int T_const::CacheNumberMemTop
```

Definition at line 2016 of file structs.h.

3.79.2.57 bracketindexflag

```
int T_const::bracketindexflag
```

Definition at line 2017 of file structs.h.

3.79.2.58 optimtimes

```
int T_const::optimtimes
```

Definition at line 2018 of file structs.h.

3.79.2.59 ListSymbolsSize

```
int T_const::ListSymbolsSize
```

Definition at line 2019 of file structs.h.

3.79.2.60 NumListSymbols

```
int T_const::NumListSymbols
```

Definition at line 2020 of file structs.h.

3.79.2.61 numpoly

```
int T_const::numpoly
```

Definition at line 2021 of file structs.h.

3.79.2.62 LeaveNegative

```
int T_const::LeaveNegative
```

Definition at line 2022 of file structs.h.

3.79.2.63 TrimPower

```
int T_const::TrimPower
```

Definition at line 2023 of file structs.h.

3.79.2.64 small_power_maxx

```
WORD T_const::small_power_maxx
```

Definition at line 2024 of file structs.h.

3.79.2.65 small_power_maxn

```
WORD T_const::small_power_maxn
```

Definition at line 2025 of file structs.h.

3.79.2.66 dummysubexp

```
WORD T_const::dummysubexp[SUBEXPSIZE+4]
```

Definition at line 2026 of file structs.h.

3.79.2.67 comsym

```
WORD T_const::comsym[8]
```

Definition at line 2027 of file structs.h.

3.79.2.68 comnum

```
WORD T_const::comnum[4]
```

Definition at line 2028 of file structs.h.

3.79.2.69 comfun

```
WORD T_const::comfun[FUNHEAD+4]
```

Definition at line 2029 of file structs.h.

3.79.2.70 comind

```
WORD T_const::comind[7]
```

Definition at line 2031 of file structs.h.

3.79.2.71 MinVecArg

```
WORD T_const::MinVecArg[7+ARGHEAD]
```

Definition at line 2032 of file structs.h.

3.79.2.72 FunArg

```
WORD T_const::FunArg[4+ARGHEAD+FUNHEAD]
```

Definition at line 2033 of file structs.h.

3.79.2.73 locwildvalue

```
WORD T_const::locwildvalue[SUBEXPSIZE]
```

Definition at line 2034 of file structs.h.

3.79.2.74 mulpat

```
WORD T_const::mulpat[SUBEXPSIZE+5]
```

Definition at line 2035 of file structs.h.

3.79.2.75 proexp

```
WORD T_const::proexp[SUBEXPSIZE+5]
```

Definition at line 2037 of file structs.h.

3.79.2.76 TMout

```
WORD T_const::TMout[40]
```

Definition at line 2038 of file structs.h.

3.79.2.77 TMbuff

```
WORD T_const::TMbuff
```

Definition at line 2039 of file structs.h.

3.79.2.78 TMdolfac

```
WORD T_const::TMdolfac
```

Definition at line 2040 of file structs.h.

3.79.2.79 nfac

```
WORD T_const::nfac
```

Definition at line 2041 of file structs.h.

3.79.2.80 nBer

```
WORD T_const::nBer
```

Definition at line 2042 of file structs.h.

3.79.2.81 mBer

```
WORD T_const::mBer
```

Definition at line 2043 of file structs.h.

3.79.2.82 PolyAct

```
WORD T_const::PolyAct
```

Definition at line 2044 of file structs.h.

3.79.2.83 RecFlag

```
WORD T_const::RecFlag
```

Definition at line 2045 of file structs.h.

3.79.2.84 inprimelist

```
WORD T_const::inprimelist
```

Definition at line 2046 of file structs.h.

3.79.2.85 sizeprimelist

```
WORD T_const::sizeprimelist
```

Definition at line 2047 of file structs.h.

3.79.2.86 fromindex

```
WORD T_const::fromindex
```

Definition at line 2048 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.80 TaBlEbAsE Struct Reference

Collaboration diagram for TaBlEbAsE:

Public Member Functions

- **PADPOSITION** (3, 0, 1, 0, 0)

Data Fields

- POSITION fillpoint
- POSITION current
- UBYTE * name
- int * tablenumbers
- TABLEBASESUBINDEX * subindex
- int numtables

3.80.1 Detailed Description

Definition at line 577 of file structs.h.

3.80.2 Field Documentation

3.80.2.1 fillpoint

```
POSITION TaBlEbAsE::fillpoint
```

Definition at line 578 of file structs.h.

3.80.2.2 current

```
POSITION TaBlEbAsE::current
```

Definition at line 579 of file structs.h.

3.80.2.3 name

```
UBYTE* TaBlEbAsE::name
```

Definition at line 580 of file structs.h.

3.80.2.4 tablenumbers

```
int* TaBlEbAsE::tablenumbers
```

Definition at line 581 of file structs.h.

3.80.2.5 subindex

```
TABLEBASESUBINDEX* TaBlEbAsE::subindex
```

Definition at line 582 of file structs.h.

3.80.2.6 numtables

```
int TaBlEbAsE::numtables
```

Definition at line 583 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.81 TaBlEbAsEsUbInDeX Struct Reference

Collaboration diagram for TaBlEbAsEsUbInDeX:

Public Member Functions

- **PADPOSITION** (0, 1, 0, 0, 0)

Data Fields

- POSITION where
- LONG size

3.81.1 Detailed Description

Definition at line 567 of file structs.h.

3.81.2 Field Documentation

3.81.2.1 where

```
POSITION TaBlEbAsEsUbInDeX::where
```

Definition at line 568 of file structs.h.

3.81.2.2 size

```
LONG TaBlEbAsEsUbInDeX::size
```

Definition at line 569 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.82 TaBlEs Struct Reference

```
#include <structs.h>
```

Collaboration diagram for TaBlEs:

Data Fields

- WORD * tablepointers
- WORD * prototype
- WORD * pattern
- MINMAX * mm
- WORD * flags
- COMPTREE * boomlijst
- UBYTE * argtail
- struct TaBlEs * spare
- WORD * buffers
- LONG totind
- LONG reserved
- LONG defined
- LONG mdefined
- int prototypeSize
- int numind
- int bounds
- int strict
- int sparse
- int numtree
- int rootnum
- int MaxTreeSize
- WORD bufnum
- WORD bufferssize
- WORD buffersfill
- WORD tablenum
- WORD mode
- WORD numdummies

3.82.1 Detailed Description

buffers, mm, flags, and prototype are always dynamically allocated, tablepointers only if needed (=0 if unallocated), boomlijst and argtail only for sparse tables.

Allocation is done for both the normal and the stub instance (spare), except for prototype and argtail which share memory.

Definition at line 337 of file structs.h.

3.82.2 Field Documentation

3.82.2.1 tablepointers

```
WORD* TaBlEs::tablepointers
```

[D] Start in tablepointers table.

Definition at line 338 of file structs.h.

Referenced by DoRecovery(), and TestSub().

3.82.2.2 prototype

```
WORD* TaBlEs::prototype
```

[D] The wildcard prototyping for arguments

Definition at line 343 of file structs.h.

Referenced by DoRecovery(), and TestSub().

3.82.2.3 pattern

```
WORD* TaBlEs::pattern
```

The pattern with which to match the arguments

Definition at line 344 of file structs.h.

Referenced by DoRecovery(), and TestSub().

3.82.2.4 mm

```
MINMAX* TaBlEs::mm
```

[D] Array bounds, dimension by dimension. # elements = numind.

Definition at line 346 of file structs.h.

Referenced by DoRecovery(), and TestSub().

3.82.2.5 flags

```
WORD* TaBlEs::flags
```

[D] Is element in use ? etc. # elements = numind.

Definition at line 347 of file structs.h.

Referenced by DoRecovery().

3.82.2.6 boomlijst

```
COMPTREE* TaBlEs::boomlijst
```

[D] Tree for searching in sparse tables

Definition at line 348 of file structs.h.

Referenced by DoRecovery().

3.82.2.7 argtail

```
UBYTE* TaBlEs::argtail
```

[D] The arguments in characters. Starts for tablebase with parenthesis to indicate tail

Definition at line 349 of file structs.h.

Referenced by DoRecovery().

3.82.2.8 spare

```
struct TaBlEs* TaBlEs::spare
```

[D] For tablebase. Alternatingly stubs and real

Definition at line 351 of file structs.h.

Referenced by DoRecovery().

3.82.2.9 buffers

```
WORD* TaBlEs::buffers
```

[D] When we use more than one compiler buffer.

Definition at line 352 of file structs.h.

Referenced by AddRHS(), and DoRecovery().

3.82.2.10 totind

```
LONG TaBlEs::totind
```

Total number requested

Definition at line 353 of file structs.h.

Referenced by DoRecovery().

3.82.2.11 reserved

```
LONG TaBlEs::reserved
```

Total reservation in tablepointers for sparse

Definition at line 354 of file structs.h.

Referenced by DoRecovery().

3.82.2.12 defined

```
LONG TaBlEs::defined
```

Number of table elements that are defined

Definition at line 355 of file structs.h.

3.82.2.13 mdefined

```
LONG TaBlEs::mdefined
```

Same as defined but after .global

Definition at line 356 of file structs.h.

3.82.2.14 prototypeSize

```
int TaBlEs::prototypeSize
```

Size of allocated memory for prototype in bytes.

Definition at line 357 of file structs.h.

Referenced by DoRecovery().

3.82.2.15 numind

```
int TaBlEs::numind
```

Number of array indices

Definition at line 358 of file structs.h.

Referenced by DoRecovery(), and TestSub().

3.82.2.16 bounds

```
int TaBlEs::bounds
```

Array bounds check on/off.

Definition at line 359 of file structs.h.

Referenced by TestSub().

3.82.2.17 strict

```
int TaBlEs::strict
```

>0: all must be defined. <0: undefined not substitute

Definition at line 360 of file structs.h.

Referenced by TestSub().

3.82.2.18 sparse

```
int TaBlEs::sparse
```

0 --> sparse table

Definition at line 361 of file structs.h.

Referenced by DoRecovery(), and TestSub().

3.82.2.19 numtree

```
int TaBlEs::numtree
```

For the tree for sparse tables

Definition at line 362 of file structs.h.

3.82.2.20 rootnum

```
int TaBlEs::rootnum
```

For the tree for sparse tables

Definition at line 363 of file structs.h.

3.82.2.21 MaxTreeSize

```
int TaBlEs::MaxTreeSize
```

For the tree for sparse tables

Definition at line 364 of file structs.h.

Referenced by DoRecovery().

3.82.2.22 bufnum

```
WORD TaBlEs::bufnum
```

Each table potentially its own buffer

Definition at line 365 of file structs.h.

Referenced by AddRHS(), and TestSub().

3.82.2.23 bufferssize

```
WORD TaBlEs::bufferssize
```

When we use more than one compiler buffer

Definition at line 366 of file structs.h.

Referenced by AddRHS(), and DoRecovery().

3.82.2.24 buffersfill

```
WORD TaBlEs::buffersfill
```

When we use more than one compiler buffer

Definition at line 367 of file structs.h.

Referenced by AddRHS().

3.82.2.25 tablenum

```
WORD TaBlEs::tablenum
```

For testing of tableuse

Definition at line 368 of file structs.h.

3.82.2.26 mode

```
WORD TaBlEs::mode
```

0: normal, 1: stub

Definition at line 369 of file structs.h.

3.82.2.27 numdummies

```
WORD TaBlEs::numdummies
```

Definition at line 370 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.83 TrAcEn Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * accu
- WORD * accup
- WORD * termp
- WORD * perm
- WORD * inlist
- WORD sgn
- WORD num
- WORD level
- WORD factor
- WORD allsign

3.83.1 Detailed Description

The struct TRACEN keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue.

Definition at line 760 of file structs.h.

3.83.2 Field Documentation

3.83.2.1 accu

```
WORD* TrAcEn::accu
```

Definition at line 761 of file structs.h.

3.83.2.2 accup

```
WORD* TrAcEn::accup
```

Definition at line 762 of file structs.h.

3.83.2.3 termp

```
WORD* TrAcEn::termp
```

Definition at line 763 of file structs.h.

3.83.2.4 perm

```
WORD* TrAcEn::perm
```

Definition at line 764 of file structs.h.

3.83.2.5 inlist

```
WORD* TrAcEn::inlist
```

Definition at line 765 of file structs.h.

3.83.2.6 sgn

```
WORD TrAcEn::sgn
```

Definition at line 766 of file structs.h.

3.83.2.7 num

```
WORD TrAcEn::num
```

Definition at line 766 of file structs.h.

3.83.2.8 level

```
WORD TrAcEn::level
```

Definition at line 766 of file structs.h.

3.83.2.9 factor

```
WORD TrAcEn::factor
```

Definition at line 766 of file structs.h.

3.83.2.10 allsign

```
WORD TrAcEn::allsign
```

Definition at line 766 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.84 TrAcEs Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * accu
- WORD * accup
- WORD * termp
- WORD * perm
- WORD * inlist
- WORD * nt3
- WORD * nt4
- WORD * j3
- WORD * j4
- WORD * e3
- WORD * e4
- WORD * eers
- WORD * mepf
- WORD * mdel
- WORD * pepf
- WORD * pdel
- WORD sgn
- WORD stap
- WORD step1
- WORD kstep
- WORD mdum

- WORD gamm
- WORD ad
- WORD a3
- WORD a4
- WORD lc3
- WORD lc4
- WORD sign1
- WORD sign2
- WORD gamma5
- WORD num
- WORD level
- WORD factor
- WORD allsign
- WORD finalstep

3.84.1 Detailed Description

The struct TRACES keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue. The 4-dimensional traces are more complicated than the n-dimensional traces (see TRACEN) because of the extra tricks that can be used. They are responsible for the shorter final expressions.

Definition at line 727 of file structs.h.

3.84.2 Field Documentation

3.84.2.1 accu

```
WORD* TrAcEs::accu
```

Definition at line 728 of file structs.h.

3.84.2.2 accup

```
WORD* TrAcEs::accup
```

Definition at line 729 of file structs.h.

3.84.2.3 termp

```
WORD* TrAcEs::termp
```

Definition at line 730 of file structs.h.

3.84.2.4 perm

```
WORD* TrAcEs::perm
```

Definition at line 731 of file structs.h.

3.84.2.5 inlist

```
WORD* TrAcEs::inlist
```

Definition at line 732 of file structs.h.

3.84.2.6 nt3

```
WORD* TrAcEs::nt3
```

Definition at line 733 of file structs.h.

3.84.2.7 nt4

```
WORD* TrAcEs::nt4
```

Definition at line 734 of file structs.h.

3.84.2.8 j3

```
WORD* TrAcEs::j3
```

Definition at line 735 of file structs.h.

3.84.2.9 j4

```
WORD* TrAcEs::j4
```

Definition at line 736 of file structs.h.

3.84.2.10 e3

```
WORD* TrAcEs::e3
```

Definition at line 737 of file structs.h.

3.84.2.11 e4

```
WORD* TrAcEs::e4
```

Definition at line 738 of file structs.h.

3.84.2.12 eers

```
WORD* TrAcEs::eers
```

Definition at line 739 of file structs.h.

3.84.2.13 mepf

```
WORD* TrAcEs::mepf
```

Definition at line 740 of file structs.h.

3.84.2.14 mdel

```
WORD* TrAcEs::mdel
```

Definition at line 741 of file structs.h.

3.84.2.15 pepf

```
WORD* TrAcEs::pepf
```

Definition at line 742 of file structs.h.

3.84.2.16 pdel

```
WORD* TrAcEs::pdel
```

Definition at line 743 of file structs.h.

3.84.2.17 sgn

```
WORD TrAcEs::sgn
```

Definition at line 744 of file structs.h.

3.84.2.18 stap

```
WORD TrAcEs::stap
```

Definition at line 745 of file structs.h.

3.84.2.19 step1

```
WORD TrAcEs::step1
```

Definition at line 746 of file structs.h.

3.84.2.20 kstep

```
WORD TrAcEs::kstep
```

Definition at line 746 of file structs.h.

3.84.2.21 mdum

```
WORD TrAcEs::mdum
```

Definition at line 746 of file structs.h.

3.84.2.22 gamm

```
WORD TrAcEs::gamm
```

Definition at line 747 of file structs.h.

3.84.2.23 ad

```
WORD TrAcEs::ad
```

Definition at line 747 of file structs.h.

3.84.2.24 a3

```
WORD TrAcEs::a3
```

Definition at line 747 of file structs.h.

3.84.2.25 a4

```
WORD TrAcEs::a4
```

Definition at line 747 of file structs.h.

3.84.2.26 lc3

```
WORD TrAcEs::lc3
```

Definition at line 747 of file structs.h.

3.84.2.27 lc4

```
WORD TrAcEs::lc4
```

Definition at line 747 of file structs.h.

3.84.2.28 sign1

```
WORD TrAcEs::sign1
```

Definition at line 748 of file structs.h.

3.84.2.29 sign2

```
WORD TrAcEs::sign2
```

Definition at line 748 of file structs.h.

3.84.2.30 gamma5

```
WORD TrAcEs::gamma5
```

Definition at line 748 of file structs.h.

3.84.2.31 num

```
WORD TrAcEs::num
```

Definition at line 748 of file structs.h.

3.84.2.32 level

```
WORD TrAcEs::level
```

Definition at line 748 of file structs.h.

3.84.2.33 factor

```
WORD TrAcEs::factor
```

Definition at line 748 of file structs.h.

3.84.2.34 allsign

```
WORD TrAcEs::allsign
```

Definition at line 748 of file structs.h.

3.84.2.35 finalstep

```
WORD TrAcEs::finalstep
```

Definition at line 749 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.85 tree Struct Reference

```
#include <structs.h>
```

Data Fields

- int parent
- int left
- int right
- int value
- int blnce
- int usage

3.85.1 Detailed Description

The subexpressions in the compiler are kept track of in a (balanced) tree to reduce the need for subexpressions and hence save much space in large rhs expressions (like when we have xxxxxxx occurrences of objects like f(x+1,x+1) in which each x+1 becomes a subexpression. The struct that controls this tree is COMPTREE.

Definition at line 281 of file structs.h.

3.85.2 Field Documentation

3.85.2.1 parent

```
int tree::parent
```

Index of parent

Definition at line 282 of file structs.h.

Referenced by IniFbuffer().

3.85.2.2 left

```
int tree::left
```

Left child (if not -1)

Definition at line 283 of file structs.h.

Referenced by IniFbuffer().

3.85.2.3 right

```
int tree::right
```

Right child (if not -1)

Definition at line 284 of file structs.h.

Referenced by IniFbuffer().

3.85.2.4 value

```
int tree::value
```

The object to be sorted and searched

Definition at line 285 of file structs.h.

Referenced by IniFbuffer().

3.85.2.5 blnce

```
int tree::blnce
```

Balance factor

Definition at line 286 of file structs.h.

Referenced by IniFbuffer().

3.85.2.6 usage

```
int tree::usage
```

Number of uses in some types of trees

Definition at line 287 of file structs.h.

Referenced by CleanupArgCache(), and IniFbuffer().

The documentation for this struct was generated from the following file:

- structs.h

3.86 VaRrEnUm Struct Reference

```
#include <structs.h>
```

Data Fields

- WORD * start
- WORD * lo
- WORD * hi

3.86.1 Detailed Description

Contains the pointers to an array in which a binary search will be performed.

Definition at line 165 of file structs.h.

3.86.2 Field Documentation

3.86.2.1 start

```
WORD* VaRrEnUm::start
```

Start point for search. Points inbetween lo and hi

Definition at line 166 of file structs.h.

Referenced by DoRecovery().

3.86.2.2 lo

```
WORD* VaRrEnUm::lo
```

Start of memory area

Definition at line 167 of file structs.h.

Referenced by DoRecovery(), and Generator().

3.86.2.3 hi

```
WORD* VaRrEnUm::hi
```

End of memory area

Definition at line 168 of file structs.h.

Referenced by DoRecovery().

The documentation for this struct was generated from the following file:

- structs.h

3.87 VeCtOr Struct Reference

Data Fields

- LONG name
- WORD complex
- WORD number
- WORD flags
- WORD node
- WORD namesize
- WORD dimension

3.87.1 Detailed Description

Definition at line 447 of file structs.h.

3.87.2 Field Documentation

3.87.2.1 name

```
LONG VeCtOr::name
```

Definition at line 448 of file structs.h.

3.87.2.2 complex

```
WORD VeCtOr::complex
```

Definition at line 449 of file structs.h.

3.87.2.3 number

```
WORD VeCtOr::number
```

Definition at line 450 of file structs.h.

3.87.2.4 flags

```
WORD VeCtOr::flags
```

Definition at line 451 of file structs.h.

3.87.2.5 node

```
WORD VeCtOr::node
```

Definition at line 452 of file structs.h.

3.87.2.6 namesize

```
WORD VeCtOr::namesize
```

Definition at line 453 of file structs.h.

3.87.2.7 dimension

```
WORD VeCtOr::dimension
```

Definition at line 454 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

3.88 X_const Struct Reference

```
#include <structs.h>
```

Data Fields

- UBYTE * currentPrompt
- UBYTE * shellname
- UBYTE * stderrname
- int timeout
- int killSignal
- int killWholeGroup
- int daemonize
- int currentExternalChannel

3.88.1 Detailed Description

The X_const struct is part of the global data and resides in the ALLGLOBALS struct A under the name X We see it used with the macro AX as in AX.timeout It contains variables that involve communication with external programs

Definition at line 2342 of file structs.h.

3.88.2 Field Documentation

3.88.2.1 currentPrompt

```
UBYTE* X_const::currentPrompt
```

Definition at line 2343 of file structs.h.

3.88.2.2 shellname

```
UBYTE* X_const::shellname
```

Definition at line 2344 of file structs.h.

3.88.2.3 stderrname

```
UBYTE* X_const::stderrname
```

Definition at line 2346 of file structs.h.

3.88.2.4 timeout

```
int X_const::timeout
```

Definition at line 2348 of file structs.h.

3.88.2.5 killSignal

```
int X_const::killSignal
```

Definition at line 2351 of file structs.h.

3.88.2.6 killWholeGroup

```
int X_const::killWholeGroup
```

Definition at line 2352 of file structs.h.

3.88.2.7 daemonize

```
int X_const::daemonize
```

Definition at line 2354 of file structs.h.

3.88.2.8 currentExternalChannel

```
int X_const::currentExternalChannel
```

Definition at line 2355 of file structs.h.

The documentation for this struct was generated from the following file:

- structs.h

Chapter 4

File Documentation

4.1 argument.c File Reference

`#include "form3.h"`
Include dependency graph for argument.c:

Macros

- #define NEWORDER

Functions

- WORD execarg (PHEAD WORD ∗term, WORD level)
- WORD execterm (PHEAD WORD ∗term, WORD level)
- int ArgumentImplode (PHEAD WORD ∗term, WORD ∗thelist)
- int ArgumentExplode (PHEAD WORD ∗term, WORD ∗thelist)
- int ArgFactorize (PHEAD WORD ∗argin, WORD ∗argout)
- WORD FindArg (PHEAD WORD ∗a)
- WORD InsertArg (PHEAD WORD ∗argin, WORD ∗argout, int par)
- int CleanupArgCache (PHEAD WORD bufnum)
- int ArgSymbolMerge (WORD ∗t1, WORD ∗t2)
- int ArgDotproductMerge (WORD ∗t1, WORD ∗t2)
- WORD ∗ TakeArgContent (PHEAD WORD ∗argin, WORD ∗argout)
- WORD ∗ MakeInteger (PHEAD WORD ∗argin, WORD ∗argout, WORD ∗argfree)
- WORD ∗ MakeMod (PHEAD WORD ∗argin, WORD ∗argout, WORD ∗argfree)
- void SortWeights (LONG ∗weights, LONG ∗extraspace, WORD number)

4.1.1 Detailed Description

Contains the routines that deal with the execution phase of the argument and related statements (like term)

Definition in file argument.c.

4.1.2 Macro Definition Documentation

4.1.2.1 NEWORDER

```
#define NEWORDER
```

Factorizes an argument in general notation (meaning that the first word of the argument is a positive size indicator) Input (argin): pointer to the complete argument Output (argout): Pointer to where the output should be written. This is in the WorkSpace Return value should be negative if anything goes wrong.

The notation of the output should be a string of arguments terminated by the number zero.

Originally we sorted in a way that the constants came last. This gave conflicts with the dollar and expression factorizations (in the expressions we wanted the zero first and then followed by the constants).

Definition at line 2013 of file argument.c.

4.1.3 Function Documentation

4.1.3.1 execarg()

```
WORD execarg (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 56 of file argument.c.

4.1.3.2 execterm()

```
WORD execterm (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1724 of file argument.c.

4.1.3.3 ArgumentImplode()

```
int ArgumentImplode (
            PHEAD WORD * term,
            WORD * thelist )
```

Definition at line 1800 of file argument.c.

4.1.3.4 ArgumentExplode()

```
int ArgumentExplode (
            PHEAD WORD * term,
            WORD * thelist )
```

Definition at line 1904 of file argument.c.

4.1.3.5 ArgFactorize()

```
int ArgFactorize (
            PHEAD WORD * argin,
            WORD * argout )
```

Definition at line 2015 of file argument.c.

4.1.3.6 FindArg()

```
WORD FindArg (
            PHEAD WORD * a )
```

Looks the argument up in the (workers) table. If it is found the number in the table is returned (plus one to make it positive). If it is not found we look in the compiler provided table. If it is found - the number in the table is returned (minus one to make it negative). If in neither table we return zero.

Definition at line 2463 of file argument.c.

4.1.3.7 InsertArg()

```
WORD InsertArg (
            PHEAD WORD * argin,
            WORD * argout,
            int par )
```

Inserts the argument into the (workers) table. If the table is too full we eliminate half of it. The eliminated elements are the ones that have not been used most recently, weighted by their total use and age(?). If par == 0 it inserts in the regular factorization cache If par == 1 it inserts in the cache defined with the FactorCache statement

Definition at line 2487 of file argument.c.

References AddNtoC(), AddRHS(), and CleanupArgCache().

Here is the call graph for this function:

4.1.3.8 CleanupArgCache()

```
int CleanupArgCache (
            PHEAD WORD bufnum )
```

Cleans up the argument factorization cache. We throw half the elements. For a weight of what we want to keep we use the product of usage and the number in the buffer.

Definition at line 2522 of file argument.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::Pointer, CbUf::rhs, SortWeights(), and tree::usage.

Referenced by InsertArg().

Here is the call graph for this function:

4.1.3.9 ArgSymbolMerge()

```
int ArgSymbolMerge (
            WORD * t1,
            WORD * t2 )
```

Definition at line 2593 of file argument.c.

4.1.3.10 ArgDotproductMerge()

```
int ArgDotproductMerge (
            WORD * t1,
            WORD * t2 )
```

Definition at line 2648 of file argument.c.

4.1.3.11 TakeArgContent()

```
WORD * TakeArgContent (
            PHEAD WORD * argin,
            WORD * argout )
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. The common pieces are put in argout as a sequence of arguments. The part with the multiple terms that are now relative prime is put in argfree which is allocated via TermMalloc and is given as the return value. The difference with the old code is that negative powers are always removed. Hence it is as in MakeInteger in which only numerators will be left: now only zero or positive powers will be remaining.

Definition at line 2716 of file argument.c.

References EndSort(), MakeInteger(), MakeMod(), NewSort(), and StoreTerm().

Here is the call graph for this function:

4.1.3.12 MakeInteger()

```
WORD * MakeInteger (
            PHEAD WORD * argin,
            WORD * argout,
            WORD * argfree )
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in argin. The number that comes out should go to argout. The new pointer in the argout buffer is the return value. The normalized argument is in argfree.

Definition at line 3262 of file argument.c.

Referenced by TakeArgContent().

4.1.3.13 MakeMod()

```
WORD * MakeMod (
            PHEAD WORD * argin,
            WORD * argout,
            WORD * argfree )
```

Similar to MakeInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3433 of file argument.c.

References GetModInverses().

Referenced by TakeArgContent().

Here is the call graph for this function:

4.1.3.14 SortWeights()

```
void SortWeights (
            LONG * weights,
            LONG * extraspace,
            WORD number )
```

Sorts an array of LONGS in the same way SplitMerge (in sort.c) works We use gradual division in two.

Definition at line 3478 of file argument.c.

References SortWeights().

Referenced by CleanupArgCache(), and SortWeights().

Here is the call graph for this function:

4.2 argument.c

Go to the documentation of this file.

```c

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

/*
    #[ include : argument.c
*/

#include "form3.h"

/*
    #] include :
    #[ execarg :

    Executes the subset of statements in an argument environment.
    The calling routine should be of the type
    if ( C->lhs[level][0] == TYPEARG ) {
        if ( execarg(term,level) ) goto GenCall;
        level = C->lhs[level][2];
        goto SkipCount;
    }
    Note that there will be cases in which extra space is needed.
    In addition the compare with C->numlhs isn't very fine, because we
    need to insert a different value (C->lhs[level][2]).
*/

WORD execarg(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    WORD *t, *r, *m, *v;
    WORD *start, *stop, *rstop, *r1, *r2 = 0, *r3 = 0, *r4, *r5, *r6, *r7, *r8, *r9;
    WORD *mm, *mstop, *rnext, *rr, *factor, type, ngcd, nq;
    CBUF *C = cbuf+AM.rbufnum, *CC = cbuf+AT.ebufnum;
    WORD i, j, k, oldnumlhs = AR.Cnumlhs, count, action = 0, olddefer = AR.DeferFlag;
    WORD oldnumrhs = CC->numrhs, size, pow, jj;
    LONG oldcpointer = CC->Pointer - CC->Buffer, oldppointer = AT.pWorkPointer, lp;
    WORD *oldwork = AT.WorkPointer, *oldwork2, scale, renorm;
    WORD kLCM = 0, kGCD = 0, kGCD2, kkLCM = 0, jLCM = 0, jGCD, sign = 1;
    int ii;
    UWORD *EAscrat, *GCDbuffer = 0, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
    AT.WorkPointer += *term;
    start = C->lhs[level];
    AR.Cnumlhs = start[2];
    stop = start + start[1];
    type = *start;
    scale = start[4];
    renorm = start[5];
    start += TYPEARGHEADSIZE;
/*
    #[ Dollars :
*/
    if ( renorm && start[1] != 0 ) {/* We have to evaluate $ symbols inside () */
        t = start+1; factor = oldwork2 = v = AT.WorkPointer;
        i = *t; t++;
        *v++ = i+3; i--; NCOPY(v,t,i);
        *v++ = 1; *v++ = 1; *v++ = 3;
        AT.WorkPointer = v;
        start = t; AR.Eside = LHSIDEX;
```

```
        NewSort(BHEAD0);
        if ( Generator(BHEAD factor,AR.Cnumlhs) ) {
            LowerSortLevel();
            AT.WorkPointer = oldwork;
            return(-1);
        }
        AT.WorkPointer = v;
        if ( EndSort(BHEAD factor,0) < 0 ) {}
        if ( *factor && *(factor+*factor) != 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("&$ in () does not evaluate into a single term");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        AR.Eside = RHSIDE;
        if ( *factor > 0 ) {
            v = factor+*factor;
            v -= ABS(v[-1]);
            *factor = v-factor;
        }
        AT.WorkPointer = v;
    }
    else {
        if ( *start < 0 ) {
            factor = start + 1;
            start += -*start;
        }
        else factor = 0;
    }
/*
    #] Dollars :
*/
    t = term;
    r = t + *t;
    rstop = r - ABS(r[-1]);
    t++;
/*
    #[ Argument detection : + argument statement
*/
    while ( t < rstop ) {
        if ( *t >= FUNCTION && functions[*t-FUNCTION].spec == 0 ) {
/*
            We have a function. First count the number of arguments.
            Tensors are excluded.
*/
            count = 0;
            v = t;
            m = t + FUNHEAD;
            r = t + t[1];
            while ( m < r ) {
                count++;
                NEXTARG(m)
            }
            if ( count <= 0 ) { t += t[1]; continue; }
/*
            Now we take the arguments one by one and test for a match
*/
            for ( i = 1; i <= count; i++ ) {
                m = start;
                while ( m < stop ) {
                    r = m + m[1];
                    j = *r++;
                    if ( j > 1 ) {
                        while ( --j > 0 ) {
                            if ( *r == i ) goto RightNum;
                            r++;
                        }
                        m = r;
                        continue;
                    }
RightNum:
                    if ( m[1] == 2 ) {
                        m += 2;
                        m += *m;
                        goto HaveTodo;
                    }
                    else {
                        r = m + m[1];
                        m += 2;
                        while ( m < r ) {
                            if ( *m == CSET ) {
                                r1 = SetElements + Sets[m[1]].first;
                                r2 = SetElements + Sets[m[1]].last;
                                while ( r1 < r2 ) {
                                    if ( *r1++ == *t ) goto HaveTodo;
                                }
                            }
```

```
                            else if ( m[1] == *t ) goto HaveTodo;
                            m += 2;
                        }
                    }
                    m += *m;
                }
                continue;
HaveTodo:
/*
                If we come here we have to do the argument i (first is 1).
*/
                sign = 1;
                action = 1;
                v[2] |= DIRTYFLAG;
                r = t + FUNHEAD;
                j = i;
                while ( --j > 0 ) { NEXTARG(r) }
                if ( ( type == TYPESPLITARG ) || ( type == TYPESPLITFIRSTARG )
                 || ( type == TYPESPLITLASTARG ) ) {
                    if ( *t > FUNCTION && *r > 0 ) {
                        WantAddPointers(2);
                        AT.pWorkSpace[AT.pWorkPointer++] = t;
                        AT.pWorkSpace[AT.pWorkPointer++] = r;
                    }
                    continue;
                }
                else if ( type == TYPESPLITARG2 ) {
                    if ( *t > FUNCTION && *r > 0 ) {
                        WantAddPointers(2);
                        AT.pWorkSpace[AT.pWorkPointer++] = t;
                        AT.pWorkSpace[AT.pWorkPointer++] = r;
                    }
                    continue;
                }
                else if ( type == TYPEFACTARG || type == TYPEFACTARG2 ) {
                    if ( *t > FUNCTION || *t == DENOMINATOR ) {
                        if ( *r > 0 ) {
                        mm = r + ARGHEAD; mstop = r + *r;
                        if ( mm + *mm < mstop ) {
                            WantAddPointers(2);
                            AT.pWorkSpace[AT.pWorkPointer++] = t;
                            AT.pWorkSpace[AT.pWorkPointer++] = r;
                            continue;
                        }
                        if ( *mm == 1+ABS(mstop[-1]) ) continue;
                        if ( mstop[-3] != 1 || mstop[-2] != 1
                            || mstop[-1] != 3 ) {
                            WantAddPointers(2);
                            AT.pWorkSpace[AT.pWorkPointer++] = t;
                            AT.pWorkSpace[AT.pWorkPointer++] = r;
                            continue;
                        }
                        GETSTOP(mm,mstop); mm++;
                        if ( mm + mm[1] < mstop ) {
                            WantAddPointers(2);
                            AT.pWorkSpace[AT.pWorkPointer++] = t;
                            AT.pWorkSpace[AT.pWorkPointer++] = r;
                            continue;
                        }
                        if ( *mm == SYMBOL && ( mm[1] > 4 ||
                            ( mm[3] != 1 && mm[3] != -1 ) ) ) {
                            WantAddPointers(2);
                            AT.pWorkSpace[AT.pWorkPointer++] = t;
                            AT.pWorkSpace[AT.pWorkPointer++] = r;
                            continue;
                        }
                        else if ( *mm == DOTPRODUCT && ( mm[1] > 5 ||
                            ( mm[4] != 1 && mm[4] != -1 ) ) ) {
                            WantAddPointers(2);
                            AT.pWorkSpace[AT.pWorkPointer++] = t;
                            AT.pWorkSpace[AT.pWorkPointer++] = r;
                            continue;
                        }
                        else if ( ( *mm == DELTA || *mm == VECTOR )
                             && mm[1] > 4 ) {
                            WantAddPointers(2);
                            AT.pWorkSpace[AT.pWorkPointer++] = t;
                            AT.pWorkSpace[AT.pWorkPointer++] = r;
                            continue;
                        }
                        }
                        else if ( factor && *factor == 4 && factor[2] == 1 ) {
                            WantAddPointers(2);
                            AT.pWorkSpace[AT.pWorkPointer++] = t;
                            AT.pWorkSpace[AT.pWorkPointer++] = r;
                            continue;
                        }
```

```
                        else if ( factor && *factor == 0
                        && ( *r == -SNUMBER && r[1] != 1 ) ) {
                            WantAddPointers(2);
                            AT.pWorkSpace[AT.pWorkPointer++] = t;
                            AT.pWorkSpace[AT.pWorkPointer++] = r;
                            continue;
                        }
                        else if ( *r == -MINVECTOR ) {
                            WantAddPointers(2);
                            AT.pWorkSpace[AT.pWorkPointer++] = t;
                            AT.pWorkSpace[AT.pWorkPointer++] = r;
                            continue;
                        }
                    }
                    continue;
                }
                else if ( type == TYPENORM || type == TYPENORM2 || type == TYPENORM3 || type == TYPENORM4 ) {
                    if ( *r < 0 ) {
                        WORD rone;
                        if ( *r == -MINVECTOR ) { rone = -1; *r = -INDEX; }
                        else if ( *r != -SNUMBER || r[1] == 1 || r[1] == 0 ) continue;
                        else { rone = r[1]; r[1] = 1; }
/*
                        Now we must multiply the general coefficient by r[1]
*/
                        if ( scale && ( factor == 0 || *factor ) ) {
                            action = 1;
                            v[2] |= DIRTYFLAG;
                            if ( rone < 0 ) {
                                if ( type == TYPENORM3 ) k = 1;
                                else k = -1;
                                rone = -rone;
                            }
                            else k = 1;
                            r1 = term + *term;
                            size = r1[-1];
                            size = REDLENG(size);
                            if ( scale > 0 ) {
                                for ( jj = 0; jj < scale; jj++ ) {
                                    if ( Mully(BHEAD (UWORD *)rstop,&size,(UWORD *)(&rone),k) )
                                        goto execargerr;
                                }
                            }
                            else {
                                for ( jj = 0; jj > scale; jj-- ) {
                                    if ( Divvy(BHEAD (UWORD *)rstop,&size,(UWORD *)(&rone),k) )
                                        goto execargerr;
                                }
                            }
                            size = INCLENG(size);
                            k = size < 0 ? -size: size;
                            rstop[k-1] = size;
                            *term = (WORD)(rstop - term) + k;
                        }
                        continue;
                    }
/*
                    Now we have to find a reference term.
                    If factor is defined and *factor != 0 we have to
                    look for the first term that matches the pattern exactly
                    Otherwise the first term plays this role
                    If its coefficient is not one,
                    we must set up a division of the whole argument by
                    this coefficient, and a multiplication of the term
                    when the type is not equal to TYPENORM2.
                    We first multiply the coefficient of the term.
                    Then we set up the division.

                    First find the magic term
*/
                    if ( type == TYPENORM4 ) {
/*
                        For normalizing everything to integers we have to
                        determine for all elements of this argument the LCM of
                        the denominators and the GCD of the numerators.
*/
                        GCDbuffer = NumberMalloc("execarg");
                        GCDbuffer2 = NumberMalloc("execarg");
                        LCMbuffer = NumberMalloc("execarg");
                        LCMb = NumberMalloc("execarg"); LCMc = NumberMalloc("execarg");
                        r4 = r + *r;
                        r1 = r + ARGHEAD;
/*
                        First take the first term to load up the LCM and the GCD
*/
                        r2 = r1 + *r1;
```

```
                        j = r2[-1];
                        if ( j < 0 ) sign = -1;
                        r3 = r2 - ABS(j);
                        k = REDLENG(j);
                        if ( k < 0 ) k = -k;
                        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
                        for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
                        k = REDLENG(j);
                        if ( k < 0 ) k = -k;
                        r3 += k;
                        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
                        for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
                        r1 = r2;
/*
                        Now go through the rest of the terms in this argument.
*/
                        while ( r1 < r4 ) {
                            r2 = r1 + *r1;
                            j = r2[-1];
                            r3 = r2 - ABS(j);
                            k = REDLENG(j);
                            if ( k < 0 ) k = -k;
                            while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
                            if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
/*
                                GCD is already 1
*/
                            }
                            else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
                                if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
                                    NumberFree(GCDbuffer,"execarg");
                                    NumberFree(GCDbuffer2,"execarg");
                                    NumberFree(LCMbuffer,"execarg");
                                    NumberFree(LCMb,"execarg"); NumberFree(LCMc,"execarg");
                                    goto execargerr;
                                }
                                kGCD = kGCD2;
                                for ( ii = 0; ii < kGCD; ii++ ) GCDbuffer[ii] = GCDbuffer2[ii];
                            }
                            else {
                                kGCD = 1; GCDbuffer[0] = 1;
                            }
                            k = REDLENG(j);
                            if ( k < 0 ) k = -k;
                            r3 += k;
                            while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
                            if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
                                for ( kLCM = 0; kLCM < k; kLCM++ )
                                    LCMbuffer[kLCM] = r3[kLCM];
                            }
                            else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
                                if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
                                    NumberFree(GCDbuffer,"execarg"); NumberFree(GCDbuffer2,"execarg");
                                    NumberFree(LCMbuffer,"execarg"); NumberFree(LCMb,"execarg");
NumberFree(LCMc,"execarg");
                                    goto execargerr;
                                }
                                DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
                                MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
                                for ( kLCM = 0; kLCM < jLCM; kLCM++ )
                                    LCMbuffer[kLCM] = LCMc[kLCM];
                            }
                            else {} /* LCM doesn't change */
                            r1 = r2;
                        }
/*
                        Now put the factor together: GCD/LCM
*/
                        r3 = (WORD *)(GCDbuffer);
                        if ( kGCD == kLCM ) {
                            for ( jGCD = 0; jGCD < kGCD; jGCD++ )
                                r3[jGCD+kGCD] = LCMbuffer[jGCD];
                            k = kGCD;
                        }
                        else if ( kGCD > kLCM ) {
                            for ( jGCD = 0; jGCD < kLCM; jGCD++ )
                                r3[jGCD+kGCD] = LCMbuffer[jGCD];
                            for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
                                r3[jGCD+kGCD] = 0;
                            k = kGCD;
                        }
                        else {
                            for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
                                r3[jGCD] = 0;
                            for ( jGCD = 0; jGCD < kLCM; jGCD++ )
                                r3[jGCD+kLCM] = LCMbuffer[jGCD];
                            k = kLCM;
```

```
                        }
/*                      NumberFree(GCDbuffer,"execarg"); GCDbuffer = 0; */
                        NumberFree(GCDbuffer2,"execarg");
                        NumberFree(LCMbuffer,"execarg");
                        NumberFree(LCMb,"execarg"); NumberFree(LCMc,"execarg");
                        j = 2*k+1;
/*
                        Now we have to correct the overal factor

                        We have a little problem here.
                        r3 is in GCDbuffer and we returned that.
                        At the same time we still use it.
                        This works as long as each worker has its own TermMalloc
*/
                        if ( scale && ( factor == 0 || *factor > 0 ) )
                            goto ScaledVariety;
/*
                        The if was added 28-nov-2012 to give MakeInteger also
                        the (0) option.
*/
                        if ( scale && ( factor == 0 || *factor ) ) {
                            size = term[*term-1];
                            size = REDLENG(size);
                            if ( MulRat(BHEAD (UWORD *)rstop,size,(UWORD *)r3,k,
                                    (UWORD *)rstop,&size) ) goto execargerr;
                            size = INCLENG(size);
                            k = size < 0 ? -size: size;
                            rstop[k-1] = size*sign;
                            *term = (WORD)(rstop - term) + k;
                        }
                    }
                    else {
                        if ( factor && *factor >= 1 ) {
                            r4 = r + *r;
                            r1 = r + ARGHEAD;
                            while ( r1 < r4 ) {
                                r2 = r1 + *r1;
                                r3 = r2 - ABS(r2[-1]);
                                j = r3 - r1;
                                r5 = factor;
                                if ( j != *r5 ) { r1 = r2; continue; }
                                r5++; r6 = r1+1;
                                while ( --j > 0 ) {
                                    if ( *r5 != *r6 ) break;
                                    r5++; r6++;
                                }
                                if ( j > 0 ) { r1 = r2; continue; }
                                break;
                            }
                            if ( r1 >= r4 ) continue;
                        }
                        else {
                            r1 = r + ARGHEAD;
                            r2 = r1 + *r1;
                            r3 = r2 - ABS(r2[-1]);
                        }
                        if ( *r3 == 1 && r3[1] == 1 ) {
                            if ( r2[-1] == 3 ) continue;
                            if ( r2[-1] == -3 && type == TYPENORM3 ) continue;
                        }
                        action = 1;
                        v[2] |= DIRTYFLAG;
                        j = r2[-1];
                        k = REDLENG(j);
                        if ( j < 0 ) j = -j;
                        if ( type == TYPENORM && scale && ( factor == 0 || *factor ) ) {
/*
                            Now we correct the overal factor
*/
ScaledVariety:;
                            size = term[*term-1];
                            size = REDLENG(size);
                            if ( scale > 0 ) {
                                for ( jj = 0; jj < scale; jj++ ) {
                                    if ( MulRat(BHEAD (UWORD *)rstop,size,(UWORD *)r3,k,
                                        (UWORD *)rstop,&size) ) goto execargerr;
                                }
                            }
                            else {
                                for ( jj = 0; jj > scale; jj-- ) {
                                    if ( DivRat(BHEAD (UWORD *)rstop,size,(UWORD *)r3,k,
                                        (UWORD *)rstop,&size) ) goto execargerr;
                                }
                            }
                            size = INCLENG(size);
                            k = size < 0 ? -size: size;
                            rstop[k-1] = size*sign;
```

```
                        *term = (WORD)(rstop - term) + k;
                    }
                }
/*
                We generate a statement for adapting all terms in the
                argument sucessively
*/
                r4 = AddRHS(AT.ebufnum,1);
                while ( (r4+j+12) > CC->Top ) r4 = DoubleCbuffer(AT.ebufnum,r4,3);
                *r4++ = j+1;
                i = (j-1)»1;
                for ( k = 0; k < i; k++ ) *r4++ = r3[i+k];
                for ( k = 0; k < i; k++ ) *r4++ = r3[k];
                if ( ( type == TYPENORM3 ) || ( type == TYPENORM4 ) ) *r4++ = j*sign;
                else *r4++ = r3[j-1];
                *r4++ = 0;
                CC->rhs[CC->numrhs+1] = r4;
                CC->Pointer = r4;
                AT.mulpat[5] = CC->numrhs;
                AT.mulpat[7] = AT.ebufnum;
            }
            else if ( type == TYPEARGTOEXTRASYMBOL ) {
                WORD n;
                if ( r[0] < 0 ) {
                    /* The argument is in the fast notation. */
                    WORD tmp[MaX(9,FUNHEAD+5)];
                    switch ( r[0] ) {
                        case -SNUMBER:
                            if ( r[1] == 0 ) {
                                tmp[0] = 0;
                            }
                            else {
                                tmp[0] = 4;
                                tmp[1] = ABS(r[1]);
                                tmp[2] = 1;
                                tmp[3] = r[1] > 0 ? 3 : -3;
                                tmp[4] = 0;
                            }
                            break;
                        case -SYMBOL:
                            tmp[0] = 8;
                            tmp[1] = SYMBOL;
                            tmp[2] = 4;
                            tmp[3] = r[1];
                            tmp[4] = 1;
                            tmp[5] = 1;
                            tmp[6] = 1;
                            tmp[7] = 3;
                            tmp[8] = 0;
                            break;
                        case -INDEX:
                        case -VECTOR:
                        case -MINVECTOR:
                            tmp[0] = 7;
                            tmp[1] = INDEX;
                            tmp[2] = 3;
                            tmp[3] = r[1];
                            tmp[4] = 1;
                            tmp[5] = 1;
                            tmp[6] = r[0] != -MINVECTOR ? 3 : -3;
                            tmp[7] = 0;
                            break;
                        default:
                            if ( r[0] <= -FUNCTION ) {
                                tmp[0] = FUNHEAD+4;
                                tmp[1] = -r[0];
                                tmp[2] = FUNHEAD;
                                ZeroFillRange(tmp,3,1+FUNHEAD);
                                tmp[FUNHEAD+1] = 1;
                                tmp[FUNHEAD+2] = 1;
                                tmp[FUNHEAD+3] = 3;
                                tmp[FUNHEAD+4] = 0;
                                break;
                            }
                            else {
                                MLOCK(ErrorMessageLock);
                                MesPrint("Unknown fast notation found (TYPEARGTOEXTRASYMBOL)");
                                MUNLOCK(ErrorMessageLock);
                                return(-1);
                            }
                    }
                    n = FindSubexpression(tmp);
                }
                else {
                    /*
                     * NOTE: writing to r[r[0]] is legal. As long as we work
                     * in a part of the term, at least the coefficient of
```

```
                         * the term must follow.
                         */
                        WORD old_rr0 = r[r[0]];
                        r[r[0]] = 0;  /* zero-terminated */
                        n = FindSubexpression(r+ARGHEAD);
                        r[r[0]] = old_rr0;
                    }
                    /* Put the new argument in the work space. */
                    if ( AT.WorkPointer+2 > AT.WorkTop ) {
                        MLOCK(ErrorMessageLock);
                        MesWork();
                        MUNLOCK(ErrorMessageLock);
                        return(-1);
                    }
                    r1 = AT.WorkPointer;
                    if ( scale ) {  /* means "tonumber" */
                        r1[0] = -SNUMBER;
                        r1[1] = n;
                    }
                    else {
                        r1[0] = -SYMBOL;
                        r1[1] = MAXVARIABLES-n;
                    }
                    /* We need r2, r3, m and k to shift the data. */
                    r2 = r + (r[0] > 0 ? r[0] : r[0] <= -FUNCTION ? 1 : 2);
                    r3 = r;
                    m = r1+ARGHEAD+2;
                    k = 2;
                    goto do_shift;
                }
                r3 = r;
                AR.DeferFlag = 0;
                if ( *r > 0 ) {
                    NewSort(BHEAD0);
                    action = 1;
                    r2 = r + *r;
                    r += ARGHEAD;
                    while ( r < r2 ) {  /* Sum over the terms */
                        m = AT.WorkPointer;
                        j = *r;
                        while ( --j >= 0 ) *m++ = *r++;
                        r1 = AT.WorkPointer;
                        AT.WorkPointer = m;
/*
                        What to do with dummy indices?
*/
                        if ( type == TYPENORM || type == TYPENORM2 || type == TYPENORM3 || type == TYPENORM4 ) {
                            if ( MultDo(BHEAD r1,AT.mulpat) ) goto execargerr;
                            AT.WorkPointer = r1 + *r1;
                        }
                        if ( Generator(BHEAD r1,level) ) goto execargerr;
                        AT.WorkPointer = r1;
                    }
                }
                else {
                    r2 = r + (( *r <= -FUNCTION ) ? 1:2);
                    r1 = AT.WorkPointer;
                    ToGeneral(r,r1,0);
                    m = r1 + ARGHEAD;
                    AT.WorkPointer = r1 + *r1;
                    NewSort(BHEAD0);
                    action = 1;
/*
                    What to do with dummy indices?
*/
                    if ( type == TYPENORM || type == TYPENORM2 || type == TYPENORM3 || type == TYPENORM4 ) {
                        if ( MultDo(BHEAD m,AT.mulpat) ) goto execargerr;
                        AT.WorkPointer = m + *m;
                    }
                    if ( (*m != 0 ) && Generator(BHEAD m,level) ) goto execargerr;
                    AT.WorkPointer = r1;
                }
                if ( EndSort(BHEAD AT.WorkPointer+ARGHEAD,1) < 0 ) goto execargerr;
                AR.DeferFlag = olddefer;
/*
                Now shift the sorted entity over the old argument.
*/
                m = AT.WorkPointer+ARGHEAD;
                while ( *m ) m += *m;
                k = WORDDIF(m,AT.WorkPointer);
                *AT.WorkPointer = k;
                AT.WorkPointer[1] = 0;
                if ( ToFast(AT.WorkPointer,AT.WorkPointer) ) {
                    if ( *AT.WorkPointer <= -FUNCTION ) k = 1;
                    else k = 2;
```

```
                }
do_shift:
                if ( *r3 > 0 ) j = k - *r3;
                else if ( *r3 <= -FUNCTION ) j = k - 1;
                else j = k - 2;

                t[1] += j;
                action = 1;
                v[2] |= DIRTYFLAG;
                if ( j > 0 ) {
                    r = m + j;
                    while ( m > AT.WorkPointer ) *--r = *--m;
                    AT.WorkPointer = r;
                    m = term + *term;
                    r = m + j;
                    while ( m > r2 ) *--r = *--m;
                }
                else if ( j < 0 ) {
                    r = r2 + j;
                    r1 = term + *term;
                    while ( r2 < r1 ) *r++ = *r2++;
                }
                r = r3;
                m = AT.WorkPointer;
                NCOPY(r,m,k);
                *term += j;
                rstop += j;
                CC->numrhs = oldnumrhs;
                CC->Pointer = CC->Buffer + oldcpointer;
            }
        }
        t += t[1];
    }
/*
    #] Argument detection :
    #[ SplitArg : + varieties
*/
    if ( ( type == TYPESPLITARG || type == TYPESPLITARG2
     || type == TYPESPLITFIRSTARG || type == TYPESPLITLASTARG ) &&
    AT.pWorkPointer > oldppointer ) {
        t = term+1;
        r1 = AT.WorkPointer + 1;
        lp = oldppointer;
        while ( t < rstop ) {
            if ( lp < AT.pWorkPointer && t == AT.pWorkSpace[lp] ) {
                v = t;
                m = t + FUNHEAD;
                r = t + t[1];
                r2 = r1; while ( t < m ) *r1++ = *t++;
                while ( m < r ) {
                    t = m;
                    NEXTARG(m)
                    if ( lp >= AT.pWorkPointer || t != AT.pWorkSpace[lp+1] ) {
                        if ( *t > 0 ) t[1] = 0;
                        while ( t < m ) *r1++ = *t++;
                        continue;
                    }
/*
                    Now we have a nontrivial argument that should be done.
*/
                    lp += 2;
                    action = 1;
                    v[2] |= DIRTYFLAG;
                    r3 = t + *t;
                    t += ARGHEAD;
                    if ( type == TYPESPLITFIRSTARG ) {
                        r4 = r1; r5 = t; r7 = oldwork;
                        *r1++ = *t + ARGHEAD;
                        for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
                        j = 0;
                        while ( t < r3 ) {
                            i = *t;
                            if ( j == 0 ) {
                                NCOPY(r7,t,i)
                                j++;
                            }
                            else {
                                NCOPY(r1,t,i)
                            }
                        }
                        *r4 = r1 - r4;
                        if ( j ) {
                            if ( ToFast(r4,r4) ) {
                                r1 = r4;
                                if ( *r1 > -FUNCTION ) r1++;
                                r1++;
                            }
```

```
                        r7 = oldwork;
                        while ( --j >= 0 ) {
                            r4 = r1; i = *r7;
                            *r1++ = i+ARGHEAD; *r1++ = 0;
                            FILLARG(r1);
                            NCOPY(r1,r7,i)
                            if ( ToFast(r4,r4) ) {
                                r1 = r4;
                                if ( *r1 > -FUNCTION ) r1++;
                                r1++;
                            }
                        }
                    }
                    t = r3;
                }
                else if ( type == TYPESPLITLASTARG ) {
                    r4 = r1; r5 = t; r7 = oldwork;
                    *r1++ = *t + ARGHEAD;
                    for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
                    j = 0;
                    while ( t < r3 ) {
                        i = *t;
                        if ( t+i >= r3 ) {
                            NCOPY(r7,t,i)
                            j++;
                        }
                        else {
                            NCOPY(r1,t,i)
                        }
                    }
                    *r4 = r1 - r4;
                    if ( j ) {
                        if ( ToFast(r4,r4) ) {
                            r1 = r4;
                            if ( *r1 > -FUNCTION ) r1++;
                            r1++;
                        }
                        r7 = oldwork;
                        while ( --j >= 0 ) {
                            r4 = r1; i = *r7;
                            *r1++ = i+ARGHEAD; *r1++ = 0;
                            FILLARG(r1);
                            NCOPY(r1,r7,i)
                            if ( ToFast(r4,r4) ) {
                                r1 = r4;
                                if ( *r1 > -FUNCTION ) r1++;
                                r1++;
                            }
                        }
                    }
                    t = r3;
                }
                else if ( factor == 0 || ( type == TYPESPLITARG2 && *factor == 0 ) ) {
                    while ( t < r3 ) {
                        r4 = r1;
                        *r1++ = *t + ARGHEAD;
                        for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
                        i = *t;
                        while ( --i >= 0 ) *r1++ = *t++;
                        if ( ToFast(r4,r4) ) {
                            r1 = r4;
                            if ( *r1 > -FUNCTION ) r1++;
                            r1++;
                        }
                    }
                }
                else if ( type == TYPESPLITARG2 ) {
/*
                    Here we better put the pattern matcher at work?
                    Remember: there are no wildcards.
*/
                    WORD *oRepFunList = AN.RepFunList;
                    WORD *oWildMask = AT.WildMask, *oWildValue = AN.WildValue;
                    AN.WildValue = AT.locwildvalue; AT.WildMask = AT.locwildvalue+2;
                    AN.NumWild = 0;
                    r4 = r1; r5 = t; r7 = oldwork;
                    *r1++ = *t + ARGHEAD;
                    for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
                    j = 0;
                    while ( t < r3 ) {
                        AN.UseFindOnly = 0; oldwork2 = AT.WorkPointer;
                        AN.RepFunList = r1;
                        AT.WorkPointer = r1+AN.RepFunNum+2;
                        i = *t;
                        if ( FindRest(BHEAD t,factor) &&
                         ( AN.UsedOtherFind || FindOnce(BHEAD t,factor) ) ) {
                            NCOPY(r7,t,i)
```

```
                                j++;
                            }
                            else if ( factor[0] == FUNHEAD+1 && factor[1] >= FUNCTION ) {
                                WORD *rr1 = t+1, *rr2 = t+i;
                                rr2 -= ABS(rr2[-1]);
                                while ( rr1 < rr2 ) {
                                    if ( *rr1 == factor[1] ) break;
                                    rr1 += rr1[1];
                                }
                                if ( rr1 < rr2 ) {
                                    NCOPY(r7,t,i)
                                    j++;
                                }
                                else {
                                    NCOPY(r1,t,i)
                                }
                            }
                            else {
                                NCOPY(r1,t,i)
                            }
                            AT.WorkPointer = oldwork2;
                        }
                        AN.RepFunList = oRepFunList;
                        *r4 = r1 - r4;
                        if ( j ) {
                            if ( ToFast(r4,r4) ) {
                                r1 = r4;
                                if ( *r1 > -FUNCTION ) r1++;
                                r1++;
                            }
                            r7 = oldwork;
                            while ( --j >= 0 ) {
                                r4 = r1; i = *r7;
                                *r1++ = i+ARGHEAD; *r1++ = 0;
                                FILLARG(r1);
                                NCOPY(r1,r7,i)
                                if ( ToFast(r4,r4) ) {
                                    r1 = r4;
                                    if ( *r1 > -FUNCTION ) r1++;
                                    r1++;
                                }
                            }
                        }
                        t = r3;
                        AT.WildMask = oWildMask; AN.WildValue = oWildValue;
                    }
                    else {
/*
                        This code deals with splitting off a single term
*/
                        r4 = r1; r5 = t;
                        *r1++ = *t + ARGHEAD;
                        for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
                        j = 0;
                        while ( t < r3 ) {
                            r6 = t + *t; r6 -= ABS(r6[-1]);
                            if ( (r6 - t) == *factor ) {
                                k = *factor - 1;
                                for ( ; k > 0; k-- ) {
                                    if ( t[k] != factor[k] ) break;
                                }
                                if ( k <= 0 ) {
                                    j = r3 - t; t += *t; continue;
                                }
                            }
                            else if ( (r6 - t) == 1 && *factor == 0 ) {
                                j = r3 - t; t += *t; continue;
                            }
                            i = *t;
                            NCOPY(r1,t,i)
                        }
                        *r4 = r1 - r4;
                        if ( j ) {
                            if ( ToFast(r4,r4) ) {
                                r1 = r4;
                                if ( *r1 > -FUNCTION ) r1++;
                                r1++;
                            }
                            t = r3 - j;
                            r4 = r1;
                            *r1++ = *t + ARGHEAD;
                            for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
                            i = *t;
                            while ( --i >= 0 ) *r1++ = *t++;
                            if ( ToFast(r4,r4) ) {
                                r1 = r4;
                                if ( *r1 > -FUNCTION ) r1++;
```

```
                                r1++;
                            }
                        }
                        t = r3;
                    }
                }
                r2[1] = r1 - r2;
            }
            else {
                r = t + t[1];
                while ( t < r ) *r1++ = *t++;
            }
        }
        r = term + *term;
        while ( t < r ) *r1++ = *t++;
        m = AT.WorkPointer;
        i = m[0] = r1 - m;
        t = term;
        while ( --i >= 0 ) *t++ = *m++;
        if ( AT.WorkPointer < m ) AT.WorkPointer = m;
    }
/*
    #] SplitArg :
    #[ FACTARG :
*/
    if ( ( type == TYPEFACTARG || type == TYPEFACTARG2 ) &&
    AT.pWorkPointer > oldppointer ) {
        t = term+1;
        r1 = AT.WorkPointer + 1;
        lp = oldppointer;
        while ( t < rstop ) {
            if ( lp < AT.pWorkPointer && AT.pWorkSpace[lp] == t ) {
                v = t;
                m = t + FUNHEAD;
                r = t + t[1];
                r2 = r1; while ( t < m ) *r1++ = *t++;
                while ( m < r ) {
                    rr = t = m;
                    NEXTARG(m)
                    if ( lp >= AT.pWorkPointer || AT.pWorkSpace[lp+1] != t ) {
                        if ( *t > 0 ) t[1] = 0;
                        while ( t < m ) *r1++ = *t++;
                        continue;
                    }
/*
                    Now we have a nontrivial argument that should be studied.
                    Try to find common factors.
*/
                    lp += 2;
                    if ( *t < 0 ) {
                        if ( factor && ( *factor == 0 && *t == -SNUMBER ) ) {
                            *r1++ = *t++;
                            if ( *t == 0 ) *r1++ = *t++;
                            else { *r1++ = 1; t++; }
                            continue;
                        }
                        else if ( factor && *factor == 4 && factor[2] == 1 ) {
                            if ( *t == -SNUMBER ) {
                                if ( factor[3] < 0 || t[1] >= 0 ) {
                                    while ( t < m ) *r1++ = *t++;
                                }
                                else {
                                    *r1++ = -SNUMBER; *r1++ = -1;
                                    *r1++ = *t++; *r1++ = -*t++;
                                }
                            }
                            else {
                                while ( t < m ) *r1++ = *t++;
                                *r1++ = -SNUMBER; *r1++ = 1;
                            }
                            continue;
                        }
                        else if ( *t == -MINVECTOR ) {
                            *r1++ = -VECTOR; t++; *r1++ = *t++;
                            *r1++ = -SNUMBER; *r1++ = -1;
                            *r1++ = -SNUMBER; *r1++ = 1;
                            continue;
                        }
                    }
/*
                    Now we have a nontrivial argument
*/
                    r3 = t + *t;
                    t += ARGHEAD;  r5 = t; /* Store starting point */
                    /* We have terms from r5 to r3 */
                    if ( r5+*r5 == r3 && factor ) { /* One term only */
                        if ( *factor == 0 ) {
```

```
                        GETSTOP(t,r6);
                        r9 = r1; *r1++ = 0; *r1++ = 1;
                        FILLARG(r1);
                        *r1++ = (r6-t)+3; t++;
                        while ( t < r6 ) *r1++ = *t++;
                        *r1++ = 1; *r1++ = 1; *r1++ = 3;
                        *r9 = r1-r9;
                        if ( ToFast(r9,r9) ) {
                            if ( *r9 <= -FUNCTION ) r1 = r9+1;
                            else r1 = r9+2;
                        }
                        t = r3; continue;
                    }
                    if ( factor[0] == 4 && factor[2] == 1 ) {
                        GETSTOP(t,r6);
                        r7 = r1; *r1++ = (r6-t)+3+ARGHEAD; *r1++ = 0;
                        FILLARG(r1);
                        *r1++ = (r6-t)+3; t++;
                        while ( t < r6 ) *r1++ = *t++;
                        *r1++ = 1; *r1++ = 1; *r1++ = 3;
                        if ( ToFast(r7,r7) ) {
                            if ( *r7 <= -FUNCTION ) r1 = r7+1;
                            else r1 = r7+2;
                        }
                        if ( r3[-1] < 0 && factor[3] > 0 ) {
                            *r1++ = -SNUMBER; *r1++ = -1;
                            if ( r3[-1] == -3 && r3[-2] == 1
                            && ( r3[-3] & MAXPOSITIVE ) == r3[-3] ) {
                                *r1++ = -SNUMBER; *r1++ = r3[-3];
                            }
                            else {
                                *r1++ = (r3-r6)+1+ARGHEAD;
                                *r1++ = 0;
                                FILLARG(r1);
                                *r1++ = (r3-r6+1);
                                while ( t < r3 ) *r1++ = *t++;
                                r1[-1] = -r1[-1];
                            }
                        }
                        else {
                            if ( ( r3[-1] == -3 || r3[-1] == 3 )
                            && r3[-2] == 1
                            && ( r3[-3] & MAXPOSITIVE ) == r3[-3] ) {
                                *r1++ = -SNUMBER; *r1++ = r3[-3];
                                if ( r3[-1] < 0 ) r1[-1] = - r1[-1];
                            }
                            else {
                                *r1++ = (r3-r6)+1+ARGHEAD;
                                *r1++ = 0;
                                FILLARG(r1);
                                *r1++ = (r3-r6+1);
                                while ( t < r3 ) *r1++ = *t++;
                            }
                        }
                        t = r3; continue;
                    }
                }
/*
                Now we take the first term and look for its pieces
                inside the other terms.

                It is at this point that a more general factorization
                routine could take over (allowing for writing the output
                properly of course).
*/
                if ( AC.OldFactArgFlag == NEWFACTARG ) {
                if ( factor == 0 ) {
                    WORD *oldworkpointer2 = AT.WorkPointer;
                    AT.WorkPointer = r1 + AM.MaxTer+FUNHEAD;
                    if ( ArgFactorize(BHEAD t-ARGHEAD,r1) < 0 ) {
                        MesCall("ExecArg");
                        return(-1);
                    }
                    AT.WorkPointer = oldworkpointer2;
                    t = r3;
                    while ( *r1 ) { NEXTARG(r1) }
                }
                else {
                    rnext = t + *t;
                    GETSTOP(t,r6);
                    t++;
                    t = r5; pow = 1;
                    while ( t < r3 ) {
                        t += *t; if ( t[-1] > 0 ) { pow = 0; break; }
                    }
/*
                    We have to add here the code for computing the GCD
```

```
                        and to divide it out.

            #[ Numerical factor :
*/
                        t = r5;
                        EAscrat = (UWORD *)(TermMalloc("execarg"));
                        if ( t + *t == r3 ) goto onetermnew;
                        GETSTOP(t,r6);
                        ngcd = t[t[0]-1];
                        i = abs(ngcd)-1;
                        while ( --i >= 0 ) EAscrat[i] = r6[i];
                        t += *t;
                        while ( t < r3 ) {
                            GETSTOP(t,r6);
                            i = t[t[0]-1];
                            if ( AccumGCD(BHEAD EAscrat,&ngcd,(UWORD *)r6,i) ) goto execargerr;
                            if ( ngcd == 3 && EAscrat[0] == 1 && EAscrat[1] == 1 ) break;
                            t += *t;
                        }
                        if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
                            if ( pow ) ngcd = -ngcd;
                            t = r5; r9 = r1; *r1++ = t[-ARGHEAD]; *r1++ = 1;
                            FILLARG(r1); ngcd = REDLENG(ngcd);
                            while ( t < r3 ) {
                                GETSTOP(t,r6);
                                r7 = t; r8 = r1;
                                while ( r7 < r6) *r1++ = *r7++;
                                t += *t;
                                i = REDLENG(t[-1]);
                                if ( DivRat(BHEAD (UWORD *)r6,i,EAscrat,ngcd,(UWORD *)r1,&nq) ) goto execargerr;
                                nq = INCLENG(nq);
                                i = ABS(nq)-1;
                                r1 += i; *r1++ = nq; *r8 = r1-r8;
                            }
                            *r9 = r1-r9;
                            ngcd = INCLENG(ngcd);
                            i = ABS(ngcd)-1;
                            if ( factor && *factor == 0 ) {}
                            else if ( ( factor && factor[0] == 4 && factor[2] == 1
                            && factor[3] == -3 ) || pow == 0 ) {
                                r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
                                FILLARG(r1); *r1++ = i+2;
                                for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
                                *r1++ = ngcd;
                                if ( ToFast(r9,r9) ) r1 = r9+2;
                            }
                            else if ( factor && factor[0] == 4 && factor[2] == 1
                            && factor[3] > 0 && pow ) {
                                if ( ngcd < 0 ) ngcd = -ngcd;
                                *r1++ = -SNUMBER; *r1++ = -1;
                                r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
                                FILLARG(r1); *r1++ = i+2;
                                for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
                                *r1++ = ngcd;
                                if ( ToFast(r9,r9) ) r1 = r9+2;
                            }
                            else {
                                if ( ngcd < 0 ) ngcd = -ngcd;
                                if ( pow ) { *r1++ = -SNUMBER; *r1++ = -1; }
                                if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
                                    r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
                                    FILLARG(r1); *r1++ = i+2;
                                    for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
                                    *r1++ = ngcd;
                                    if ( ToFast(r9,r9) ) r1 = r9+2;
                                }
                            }
                        }
/*
            #] Numerical factor :
*/
                        else {
onetermnew:;
                            if ( factor == 0 || *factor > 2 ) {
                            if ( pow > 0 ) {
                                *r1++ = -SNUMBER; *r1++ = -1;
                                t = r5;
                                while ( t < r3 ) {
                                    t += *t; t[-1] = -t[-1];
                                }
                            }
                            t = rr; *r1++ = *t++; *r1++ = 1; t++;
                            COPYARG(r1,t);
                            while ( t < m ) *r1++ = *t++;
                            }
                        }
```

```
                        TermFree(EAscrat,"execarg");
                    }
                    }
                    else {  /* AC.OldFactArgFlag is ON */
                    {
                    WORD *mnext, ncom;
                    rnext = t + *t;
                    GETSTOP(t,r6);
                    t++;
                    if ( factor == 0 ) {
                      while ( t < r6 ) {
/*
            #[ SYMBOL :
*/
                        if ( *t == SYMBOL ) {
                            r7 = t; r8 = t + t[1]; t += 2;
                            while ( t < r8 ) {
                                pow = t[1];
                                mm = rnext;
                                while ( mm < r3 ) {
                                    mnext = mm + *mm;
                                    GETSTOP(mm,mstop); mm++;
                                    while ( mm < mstop ) {
                                        if ( *mm != SYMBOL ) mm += mm[1];
                                        else break;
                                    }
                                    if ( *mm == SYMBOL ) {
                                        mstop = mm + mm[1]; mm += 2;
                                        while ( *mm != *t && mm < mstop ) mm += 2;
                                        if ( mm >= mstop ) pow = 0;
                                        else if ( pow > 0 && mm[1] > 0 ) {
                                            if ( mm[1] < pow ) pow = mm[1];
                                        }
                                        else if ( pow < 0 && mm[1] < 0 ) {
                                            if ( mm[1] > pow ) pow = mm[1];
                                        }
                                        else pow = 0;
                                    }
                                    else pow = 0;
                                    if ( pow == 0 ) break;
                                    mm = mnext;
                                }
                                if ( pow == 0 ) { t += 2; continue; }
/*
                                We have a factor
*/
                                action = 1; i = pow;
                                if ( i > 0 ) {
                                    while ( --i >= 0 ) {
                                        *r1++ = -SYMBOL;
                                        *r1++ = *t;
                                    }
                                }
                                else {
                                    while ( i++ < 0 ) {
                                        *r1++ = 8 + ARGHEAD;
                                        for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
                                        *r1++ = 8; *r1++ = SYMBOL;
                                        *r1++ = 4; *r1++ = *t; *r1++ = -1;
                                        *r1++ = 1; *r1++ = 1; *r1++ = 3;
                                    }
                                }
/*
                                Now we have to remove the symbols
*/
                                t[1] -= pow;
                                mm = rnext;
                                while ( mm < r3 ) {
                                    mnext = mm + *mm;
                                    GETSTOP(mm,mstop); mm++;
                                    while ( mm < mstop ) {
                                        if ( *mm != SYMBOL ) mm += mm[1];
                                        else break;
                                    }
                                    mstop = mm + mm[1]; mm += 2;
                                    while ( mm < mstop && *mm != *t ) mm += 2;
                                    mm[1] -= pow;
                                    mm = mnext;
                                }
                                t += 2;
                            }
                        }
/*
            #] SYMBOL :
            #[ DOTPRODUCT :
*/
                        else if ( *t == DOTPRODUCT ) {
```

```
                        r7 = t; r8 = t + t[1]; t += 2;
                        while ( t < r8 ) {
                            pow = t[2];
                            mm = rnext;
                            while ( mm < r3 ) {
                                mnext = mm + *mm;
                                GETSTOP(mm,mstop); mm++;
                                while ( mm < mstop ) {
                                    if ( *mm != DOTPRODUCT ) mm += mm[1];
                                    else break;
                                }
                                if ( *mm == DOTPRODUCT ) {
                                    mstop = mm + mm[1]; mm += 2;
                                    while ( ( *mm != *t || mm[1] != t[1] )
                                         && mm < mstop ) mm += 3;
                                    if ( mm >= mstop ) pow = 0;
                                    else if ( pow > 0 && mm[2] > 0 ) {
                                        if ( mm[2] < pow ) pow = mm[2];
                                    }
                                    else if ( pow < 0 && mm[2] < 0 ) {
                                        if ( mm[2] > pow ) pow = mm[2];
                                    }
                                    else pow = 0;
                                }
                                else pow = 0;
                                if ( pow == 0 ) break;
                                mm = mnext;
                            }
                            if ( pow == 0 ) { t += 3; continue; }
/*
                            We have a factor

*/
                            action = 1; i = pow;
                            if ( i > 0 ) {
                                while ( --i >= 0 ) {
                                    *r1++ = 9 + ARGHEAD;
                                    for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
                                    *r1++ = 9; *r1++ = DOTPRODUCT;
                                    *r1++ = 5; *r1++ = *t; *r1++ = t[1]; *r1++ = 1;
                                    *r1++ = 1; *r1++ = 1; *r1++ = 3;
                                }
                            }
                            else {
                                while ( i++ < 0 ) {
                                    *r1++ = 9 + ARGHEAD;
                                    for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
                                    *r1++ = 9; *r1++ = DOTPRODUCT;
                                    *r1++ = 5; *r1++ = *t; *r1++ = t[1]; *r1++ = -1;
                                    *r1++ = 1; *r1++ = 1; *r1++ = 3;
                                }
                            }
/*
                            Now we have to remove the dotproducts

*/
                            t[2] -= pow;
                            mm = rnext;
                            while ( mm < r3 ) {
                                mnext = mm + *mm;
                                GETSTOP(mm,mstop); mm++;
                                while ( mm < mstop ) {
                                    if ( *mm != DOTPRODUCT ) mm += mm[1];
                                    else break;
                                }
                                mstop = mm + mm[1]; mm += 2;
                                while ( mm < mstop && ( *mm != *t
                                    || mm[1] != t[1] ) ) mm += 3;
                                mm[2] -= pow;
                                mm = mnext;
                            }
                            t += 3;
                        }
                    }
/*
            #] DOTPRODUCT :
            #[ DELTA/VECTOR :
*/
                    else if ( *t == DELTA || *t == VECTOR ) {
                        r7 = t; r8 = t + t[1]; t += 2;
                        while ( t < r8 ) {
                            mm = rnext;
                            pow = 1;
                            while ( mm < r3 ) {
                                mnext = mm + *mm;
                                GETSTOP(mm,mstop); mm++;
                                while ( mm < mstop ) {
                                    if ( *mm != *r7 ) mm += mm[1];
                                    else break;
```

```
                                    }
                                    if ( *mm == *r7 ) {
                                        mstop = mm + mm[1]; mm += 2;
                                        while ( ( *mm != *t || mm[1] != t[1] )
                                            && mm < mstop ) mm += 2;
                                        if ( mm >= mstop ) pow = 0;
                                    }
                                    else pow = 0;
                                    if ( pow == 0 ) break;
                                    mm = mnext;
                                }
                                if ( pow == 0 ) { t += 2; continue; }
/*
                                We have a factor
*/
                                action = 1;
                                *r1++ = 8 + ARGHEAD;
                                for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
                                *r1++ = 8; *r1++ = *r7;
                                *r1++ = 4; *r1++ = *t; *r1++ = t[1];
                                *r1++ = 1; *r1++ = 1; *r1++ = 3;
/*
                                Now we have to remove the delta's/vectors
*/
                                mm = rnext;
                                while ( mm < r3 ) {
                                    mnext = mm + *mm;
                                    GETSTOP(mm,mstop); mm++;
                                    while ( mm < mstop ) {
                                        if ( *mm != *r7 ) mm += mm[1];
                                        else break;
                                    }
                                    mstop = mm + mm[1]; mm += 2;
                                    while ( mm < mstop && (
                                     *mm != *t || mm[1] != t[1] ) ) mm += 2;
                                    *mm = mm[1] = NOINDEX;
                                    mm = mnext;
                                }
                                *t = t[1] = NOINDEX;
                                t += 2;
                            }
                        }
/*
            #] DELTA/VECTOR :
            #[ INDEX :
*/
                        else if ( *t == INDEX ) {
                            r7 = t; r8 = t + t[1]; t += 2;
                            while ( t < r8 ) {
                                mm = rnext;
                                pow = 1;
                                while ( mm < r3 ) {
                                    mnext = mm + *mm;
                                    GETSTOP(mm,mstop); mm++;
                                    while ( mm < mstop ) {
                                        if ( *mm != *r7 ) mm += mm[1];
                                        else break;
                                    }
                                    if ( *mm == *r7 ) {
                                        mstop = mm + mm[1]; mm += 2;
                                        while ( *mm != *t
                                            && mm < mstop ) mm++;
                                        if ( mm >= mstop ) pow = 0;
                                    }
                                    else pow = 0;
                                    if ( pow == 0 ) break;
                                    mm = mnext;
                                }
                                if ( pow == 0 ) { t++; continue; }
/*
                                We have a factor
*/
                                action = 1;
/*
                                The next looks like an error.
                                We should have here a VECTOR or INDEX like object

                                *r1++ = 7 + ARGHEAD;
                                for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
                                *r1++ = 7; *r1++ = *r7;
                                *r1++ = 3; *r1++ = *t;
                                *r1++ = 1; *r1++ = 1; *r1++ = 3;

                                Replace this by:  (11-apr-2007)
*/
                                if ( *t < 0 ) { *r1++ = -VECTOR; }
                                else { *r1++ = -INDEX; }
```

```
                                *r1++ = *t;
/*
                                Now we have to remove the index
*/
                                *t = NOINDEX;
                                mm = rnext;
                                while ( mm < r3 ) {
                                    mnext = mm + *mm;
                                    GETSTOP(mm,mstop); mm++;
                                    while ( mm < mstop ) {
                                        if ( *mm != *r7 ) mm += mm[1];
                                        else break;
                                    }
                                    mstop = mm + mm[1]; mm += 2;
                                    while ( mm < mstop &&
                                     *mm != *t ) mm += 1;
                                    *mm = NOINDEX;
                                    mm = mnext;
                                }
                                t += 1;
                            }
                        }
/*
            #] INDEX :
            #[ FUNCTION :
*/
                        else if ( *t >= FUNCTION ) {
/*
                            In the next code we should actually look inside
                            the DENOMINATOR or EXPONENT for noncommuting objects
*/
                            if ( *t >= FUNCTION &&
                                functions[*t-FUNCTION].commute == 0 ) ncom = 0;
                            else ncom = 1;
                            if ( ncom ) {
                                mm = r5 + 1;
                                while ( mm < t && ( *mm == DUMMYFUN
                                || *mm == DUMMYTEN ) ) mm += mm[1];
                                if ( mm < t ) { t += t[1]; continue; }
                            }
                            mm = rnext; pow = 1;
                            while ( mm < r3 ) {
                                mnext = mm + *mm;
                                GETSTOP(mm,mstop); mm++;
                                while ( mm < mstop ) {
                                    if ( *mm == *t && mm[1] == t[1] ) {
                                        for ( i = 2; i < t[1]; i++ ) {
                                            if ( mm[i] != t[i] ) break;
                                        }
                                        if ( i >= t[1] )
                                            { mm += mm[1]; goto nextmterm; }
                                    }
                                    if ( ncom && *mm != DUMMYFUN && *mm != DUMMYTEN )
                                        { pow = 0; break; }
                                    mm += mm[1];
                                }
                                if ( mm >= mstop ) pow = 0;
                                if ( pow == 0 ) break;
nextmterm:                      mm = mnext;
                            }
                            if ( pow == 0 ) { t += t[1]; continue; }
/*
                            Copy the function
*/
                            action = 1;
                            *r1++ = t[1] + 4 + ARGHEAD;
                            for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
                            *r1++ = t[1] + 4;
                            for ( i = 0; i < t[1]; i++ ) *r1++ = t[i];
                            *r1++ = 1; *r1++ = 1; *r1++ = 3;
/*
                            Now we have to take out the functions
*/
                            mm = rnext;
                            while ( mm < r3 ) {
                                mnext = mm + *mm;
                                GETSTOP(mm,mstop); mm++;
                                while ( mm < mstop ) {
                                    if ( *mm == *t && mm[1] == t[1] ) {
                                        for ( i = 2; i < t[1]; i++ ) {
                                            if ( mm[i] != t[i] ) break;
                                        }
                                        if ( i >= t[1] ) {
                                            if ( functions[*t-FUNCTION].spec > 0 )
                                                *mm = DUMMYTEN;
                                            else
                                                *mm = DUMMYFUN;
```

```
                                        mm += mm[1];
                                        goto nextterm;
                                    }
                                }
                                mm += mm[1];
                            }
nextterm:                       mm = mnext;
                        }
                        if ( functions[*t-FUNCTION].spec > 0 )
                                *t = DUMMYTEN;
                        else
                                *t = DUMMYFUN;
                        action = 1;
                        v[2] = DIRTYFLAG;
                        t += t[1];
                    }
/*
            #] FUNCTION :
*/
                    else {
                        t += t[1];
                    }
                  }
                }
                t = r5; pow = 1;
                while ( t < r3 ) {
                    t += *t; if ( t[-1] > 0 ) { pow = 0; break; }
                }
/*
                We have to add here the code for computing the GCD
                and to divide it out.
*/
/*
            #[ Numerical factor :
*/
                t = r5;
                EAscrat = (UWORD *)(TermMalloc("execarg"));
                if ( t + *t == r3 ) goto oneterm;
                GETSTOP(t,r6);
                ngcd = t[t[0]-1];
                i = abs(ngcd)-1;
                while ( --i >= 0 ) EAscrat[i] = r6[i];
                t += *t;
                while ( t < r3 ) {
                    GETSTOP(t,r6);
                    i = t[t[0]-1];
                    if ( AccumGCD(BHEAD EAscrat,&ngcd,(UWORD *)r6,i) ) goto execargerr;
                    if ( ngcd == 3 && EAscrat[0] == 1 && EAscrat[1] == 1 ) break;
                    t += *t;
                }
                if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
                    if ( pow ) ngcd = -ngcd;
                    t = r5; r9 = r1; *r1++ = t[-ARGHEAD]; *r1++ = 1;
                    FILLARG(r1); ngcd = REDLENG(ngcd);
                    while ( t < r3 ) {
                        GETSTOP(t,r6);
                        r7 = t; r8 = r1;
                        while ( r7 < r6) *r1++ = *r7++;
                        t += *t;
                        i = REDLENG(t[-1]);
                        if ( DivRat(BHEAD (UWORD *)r6,i,EAscrat,ngcd,(UWORD *)r1,&nq) ) goto
execargerr;
                        nq = INCLENG(nq);
                        i = ABS(nq)-1;
                        r1 += i; *r1++ = nq; *r8 = r1-r8;
                    }
                    *r9 = r1-r9;
                    ngcd = INCLENG(ngcd);
                    i = ABS(ngcd)-1;
                    if ( factor && *factor == 0 ) {}
                    else if ( ( factor && factor[0] == 4 && factor[2] == 1
                    && factor[3] == -3 ) || pow == 0 ) {
                        r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
                        FILLARG(r1); *r1++ = i+2;
                        for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
                        *r1++ = ngcd;
                        if ( ToFast(r9,r9) ) r1 = r9+2;
                    }
                    else if ( factor && factor[0] == 4 && factor[2] == 1
                    && factor[3] > 0 && pow ) {
                        if ( ngcd < 0 ) ngcd = -ngcd;
                        *r1++ = -SNUMBER; *r1++ = -1;
                        r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
                        FILLARG(r1); *r1++ = i+2;
                        for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
                        *r1++ = ngcd;
                        if ( ToFast(r9,r9) ) r1 = r9+2;
```

```
                        }
                        else {
                            if ( ngcd < 0 ) ngcd = -ngcd;
                            if ( pow ) { *r1++ = -SNUMBER; *r1++ = -1; }
                            if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
                                r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
                                FILLARG(r1); *r1++ = i+2;
                                for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
                                *r1++ = ngcd;
                                if ( ToFast(r9,r9) ) r1 = r9+2;
                            }
                        }
                    }
/*
            #] Numerical factor :
*/
                    else {
oneterm:;
                        if ( factor == 0 || *factor > 2 ) {
                        if ( pow > 0 ) {
                            *r1++ = -SNUMBER; *r1++ = -1;
                            t = r5;
                            while ( t < r3 ) {
                                t += *t; t[-1] = -t[-1];
                            }
                        }
                        t = rr; *r1++ = *t++; *r1++ = 1; t++;
                        COPYARG(r1,t);
                        while ( t < m ) *r1++ = *t++;
                        }
                    }
                    TermFree(EAscrat,"execarg");
                    }
                    } /* AC.OldFactArgFlag */
                }
                r2[1] = r1 - r2;
                action = 1;
                v[2] = DIRTYFLAG;
            }
            else {
                r = t + t[1];
                while ( t < r ) *r1++ = *t++;
            }
        }
        r = term + *term;
        while ( t < r ) *r1++ = *t++;
        m = AT.WorkPointer;
        i = m[0] = r1 - m;
        t = term;
        while ( --i >= 0 ) *t++ = *m++;
        if ( AT.WorkPointer < t ) AT.WorkPointer = t;
    }
/*
    #] FACTARG :
*/
    AR.Cnumlhs = oldnumlhs;
    if ( action && Normalize(BHEAD term) ) goto execargerr;
    AT.WorkPointer = oldwork;
    if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
    AT.pWorkPointer = oldppointer;
    if ( GCDbuffer ) NumberFree(GCDbuffer,"execarg");
    return(action);
execargerr:
    AT.WorkPointer = oldwork;
    AT.pWorkPointer = oldppointer;
    MLOCK(ErrorMessageLock);
    MesCall("execarg");
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
    #] execarg :
    #[ execterm :
*/

WORD execterm(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    CBUF *C = cbuf+AM.rbufnum;
    WORD oldnumlhs = AR.Cnumlhs;
    WORD maxisat = C->lhs[level][2];
    WORD *buffer1 = 0;
    WORD *oldworkpointer = AT.WorkPointer;
    WORD *t1, i;
    WORD olddeferflag = AR.DeferFlag, tryterm = 0;
    AR.DeferFlag = 0;
```

```
    do {
        AR.Cnumlhs = C->lhs[level][3];
        NewSort(BHEAD0);
/*
        Normally for function arguments we do not use PolyFun/PolyRatFun.
        Hence NewSort sets the corresponding variables to zero.
        Here we overwrite that.
*/
        AN.FunSorts[AR.sLevel]->PolyFlag = ( AR.PolyFun != 0 ) ? AR.PolyFunType: 0;
        if ( AR.PolyFun == 0 ) { AN.FunSorts[AR.sLevel]->PolyFlag = 0; }
        else if ( AR.PolyFunType == 1 ) { AN.FunSorts[AR.sLevel]->PolyFlag = 1; }
        else if ( AR.PolyFunType == 2 ) {
            if ( AR.PolyFunExp == 2 ) AN.FunSorts[AR.sLevel]->PolyFlag = 1;
            else                      AN.FunSorts[AR.sLevel]->PolyFlag = 2;
        }
        if ( buffer1 ) {
            term = buffer1;
            while ( *term ) {
                t1 = oldworkpointer;
                i = *term; while ( --i >= 0 ) *t1++ = *term++;
                AT.WorkPointer = t1;
                if ( Generator(BHEAD oldworkpointer,level) ) goto exectermerr;
            }
        }
        else {
            if ( Generator(BHEAD term,level) ) goto exectermerr;
        }
        if ( buffer1 ) {
            if ( tryterm ) { TermFree(buffer1,"buffer in sort statement"); tryterm = 0; }
            else { M_free((void *)buffer1,"buffer in sort statement"); }
            buffer1 = 0;
        }
        AN.tryterm = 1;
        if ( EndSort(BHEAD (WORD *)((VOID *)(&buffer1)),2) < 0 ) goto exectermerr;
        tryterm = AN.tryterm; AN.tryterm = 0;
        level = AR.Cnumlhs;
    } while ( AR.Cnumlhs < maxisat );
    AR.Cnumlhs = oldnumlhs;
    AR.DeferFlag = olddeferflag;
    term = buffer1;
    while ( *term ) {
        t1 = oldworkpointer;
        i = *term; while ( --i >= 0 ) *t1++ = *term++;
        AT.WorkPointer = t1;
        if ( Generator(BHEAD oldworkpointer,level) ) goto exectermerr;
    }
    if ( tryterm ) { TermFree(buffer1,"buffer in term statement"); tryterm = 0; }
    else { M_free(buffer1,"buffer in term statement"); }
    buffer1 = 0;
    AT.WorkPointer = oldworkpointer;
    return(0);
exectermerr:
    AT.WorkPointer = oldworkpointer;
    AR.DeferFlag = olddeferflag;
    MLOCK(ErrorMessageLock);
    MesCall("execterm");
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
    #] execterm :
    #[ ArgumentImplode :
*/

int ArgumentImplode(PHEAD WORD *term, WORD *thelist)
{
    GETBIDENTITY
    WORD *liststart, *listtstop, *inlist;
    WORD *w, *t, *tend, *tstop, *tt, *ttstop, *ttt, ncount, i;
    int action = 0;
    listtstop = thelist + thelist[1];
    liststart = thelist + 2;
    t = term;
    tend = t + *t;
    tstop = tend - ABS(tend[-1]);
    t++;
    while ( t < tstop ) {
        if ( *t >= FUNCTION ) {
            inlist = liststart;
            while ( inlist < listtstop && *inlist != *t ) inlist += inlist[1];
            if ( inlist < listtstop ) {
                tt = t; ttstop = t + t[1]; w = AT.WorkPointer;
                for ( i = 0; i < FUNHEAD; i++ ) *w++ = *tt++;
                while ( tt < ttstop ) {
                    ncount = 0;
                    if ( *tt == -SNUMBER && tt[1] == 0 ) {
```

```
                        ncount = 1; ttt = tt; tt += 2;
                        while ( tt < ttstop && *tt == -SNUMBER && tt[1] == 0 ) {
                            ncount++; tt += 2;
                        }
                    }
                    if ( ncount > 0 ) {
                        if ( tt < ttstop && *tt == -SNUMBER && ( tt[1] == 1 || tt[1] == -1 ) ) {
                            *w++ = -SNUMBER;
                            *w++ = (ncount+1) * tt[1];
                            tt += 2;
                            action = 1;
                        }
                        else if ( ( tt[0] == tt[ARGHEAD] + ARGHEAD )
                        && ( ABS(tt[tt[0]-1]) == 3 )
                        && ( tt[tt[0]-2] == 1 )
                        && ( tt[tt[0]-3] == 1 ) ) { /* Single term with coef +/- 1 */
                            i = *tt; NCOPY(w,tt,i)
                            w[-3] = ncount+1;
                            action = 1;
                        }
                        else if ( *tt == -SYMBOL ) {
                            *w++ = ARGHEAD+8;
                            *w++ = 0;
                            FILLARG(w)
                            *w++ = 8;
                            *w++ = SYMBOL;
                            *w++ = tt[1];
                            *w++ = 1;
                            *w++ = ncount+1; *w++ = 1; *w++ = 3;
                            tt += 2;
                            action = 1;
                        }
                        else if ( *tt <= -FUNCTION ) {
                            *w++ = ARGHEAD+FUNHEAD+4;
                            *w++ = 0;
                            FILLARG(w)
                            *w++ = -*tt++;
                            *w++ = FUNHEAD+4;
                            FILLFUN(w)
                            *w++ = ncount+1; *w++ = 1; *w++ = 3;
                            action = 1;
                        }
                        else {
                            while ( ttt < tt ) *w++ = *ttt++;
                            if ( tt < ttstop && *tt == -SNUMBER ) {
                                *w++ = *tt++; *w++ = *tt++;
                            }
                        }
                    }
                    else if ( *tt <= -FUNCTION ) {
                        *w++ = *tt++;
                    }
                    else if ( *tt < 0 ) {
                        *w++ = *tt++;
                        *w++ = *tt++;
                    }
                    else {
                        i = *tt; NCOPY(w,tt,i)
                    }
                }
                AT.WorkPointer[1] = w - AT.WorkPointer;
                while ( tt < tend ) *w++ = *tt++;
                ttt = AT.WorkPointer; tt = t;
                while ( ttt < w ) *tt++ = *ttt++;
                term[0] = tt - term;
                AT.WorkPointer = tt;
                tend = tt; tstop = tt - ABS(tt[-1]);
            }
        }
        t += t[1];
    }
    if ( action ) {
        if ( Normalize(BHEAD term) ) return(-1);
    }
    return(0);
}

/*
    #] ArgumentImplode :
    #[ ArgumentExplode :
*/

int ArgumentExplode(PHEAD WORD *term, WORD *thelist)
{
    GETBIDENTITY
    WORD *liststart, *listtop, *inlist, *old;
    WORD *w, *t, *tend, *tstop, *tt, *ttstop, *ttt, ncount, i;
```

```
    int action = 0;
    LONG x;
    liststop = thelist + thelist[1];
    liststart = thelist + 2;
    t = term;
    tend = t + *t;
    tstop = tend - ABS(tend[-1]);
    t++;
    while ( t < tstop ) {
        if ( *t >= FUNCTION ) {
            inlist = liststart;
            while ( inlist < liststop && *inlist != *t ) inlist += inlist[1];
            if ( inlist < liststop ) {
                tt = t; ttstop = t + t[1]; w = AT.WorkPointer;
                for ( i = 0; i < FUNHEAD; i++ ) *w++ = *tt++;
                while ( tt < ttstop ) {
                    if ( *tt == -SNUMBER && tt[1] != 0 ) {
                        if ( tt[1] < AM.MaxTer/((WORD)sizeof(WORD)*4)
                            && tt[1] > -(AM.MaxTer/((WORD)sizeof(WORD)*4))
                            && ( tt[1] > 1 || tt[1] < -1 ) ) {
                            ncount = ABS(tt[1]);
                            while ( ncount > 1 ) {
                                *w++ = -SNUMBER; *w++ = 0; ncount--;
                            }
                            *w++ = -SNUMBER;
                            if ( tt[1] < 0 ) *w++ = -1;
                            else             *w++ =  1;
                            tt += 2;
                            action = 1;
                        }
                        else {
                            *w++ = *tt++; *w++ = *tt++;
                        }
                    }
                    else if ( *tt <= -FUNCTION ) {
                        *w++ = *tt++;
                    }
                    else if ( *tt < 0 ) {
                        *w++ = *tt++;
                        *w++ = *tt++;
                    }
                    else if ( tt[0] == tt[ARGHEAD]+ARGHEAD ) {
                        ttt = tt + tt[0] - 1;
                        i = (ABS(ttt[0])-1)/2;
                        if ( i > 1 ) {
TooMany:                    old = AN.currentTerm;
                            AN.currentTerm = term;
                            MesPrint("Too many arguments in output of ArgExplode");
                            MesPrint("Term = %t");
                            AN.currentTerm = old;
                            return(-1);
                        }
                        if ( ttt[-1] != 1 ) goto NoExplode;
                        x = ttt[-2];
                        if ( 2*x > (AT.WorkTop-w)-*term ) goto TooMany;
                        ncount = x - 1;
                        while ( ncount > 0 ) {
                            *w++ = -SNUMBER; *w++ = 0; ncount--;
                        }
                        ttt[-2] = 1;
                        i = *tt; NCOPY(w,tt,i)
                        action = 1;
                    }
                    else {
NoExplode:
                        i = *tt; NCOPY(w,tt,i)
                    }
                }
                AT.WorkPointer[1] = w - AT.WorkPointer;
                while ( tt < tend ) *w++ = *tt++;
                ttt = AT.WorkPointer; tt = t;
                while ( ttt < w ) *tt++ = *ttt++;
                term[0] = tt - term;
                AT.WorkPointer = tt;
                tend = tt; tstop = tt - ABS(tt[-1]);
            }
        }
        t += t[1];
    }
    if ( action ) {
        if ( Normalize(BHEAD term) ) return(-1);
    }
    return(0);
}

/*
    #] ArgumentExplode :
```

```
    #[ ArgFactorize :
*/
#define NEWORDER

int ArgFactorize(PHEAD WORD *argin, WORD *argout)
{
/*
    #[ step 0 : Declarations and initializations
*/
    WORD *argfree, *argextra, *argcopy, *t, *tstop, *a, *a1, *a2;
#ifdef NEWORDER
    WORD *tt;
#endif
    WORD startebuf = cbuf[AT.ebufnum].numrhs,oldword;
    WORD oldsorttype = AR.SortType, numargs;
    int error = 0, action = 0, i, ii, number, sign = 1;

    *argout = 0;
/*
    #] step 0 :
    #[ step 1 : Take care of ordering
*/
    AR.SortType = SORTHIGHFIRST;
    if ( oldsorttype != AR.SortType ) {
        NewSort(BHEAD0);
        oldword = argin[*argin]; argin[*argin] = 0;
        t = argin+ARGHEAD;
        while ( *t ) {
            tstop = t + *t;
            if ( AN.ncmod != 0 ) {
                if ( AN.ncmod != 1 || ( (WORD)AN.cmod[0] < 0 ) ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Factorization modulus a number, greater than a WORD not implemented.");
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                if ( Modulus(t) ) {
                    MLOCK(ErrorMessageLock);
                    MesCall("ArgFactorize");
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                if ( !*t) { t = tstop; continue; }
            }
            StoreTerm(BHEAD t);
            t = tstop;
        }
        EndSort(BHEAD argin+ARGHEAD,0);
        argin[*argin] = oldword;
    }
/*
    #] step 1 :
    #[ step 2 : take out the 'content'.
*/
    argfree = TakeArgContent(BHEAD argin,argout);
    {
        a1 = argout;
        while ( *a1 ) {
            if ( a1[0] == -SNUMBER && ( a1[1] == 1 || a1[1] == -1 ) ) {
                if ( a1[1] == -1 ) { sign = -sign; a1[1] = 1; }
                if ( a1[2] ) {
                    a = t = a1+2; while ( *t ) NEXTARG(t);
                    i = t - a1-2;
                    t = a1; NCOPY(t,a,i);
                    *t = 0;
                    continue;
                }
                else {
                    a1[0] = 0;
                }
                break;
            }
            else if ( a1[0] == FUNHEAD+ARGHEAD+4 && a1[ARGHEAD] == FUNHEAD+4
            && a1[*a1-1] == 3 && a1[*a1-2] == 1 && a1[*a1-3] == 1
            && a1[ARGHEAD+1] >= FUNCTION ) {
                a = t = a1+*a1; while ( *t ) NEXTARG(t);
                i = t - a;
                *a1 = -a1[ARGHEAD+1]; t = a1+1; NCOPY(t,a,i);
                *t = 0;
            }
            NEXTARG(a1);
        }
    }
    if ( argfree == 0 ) {
        argfree = argin;
    }
    else if ( argfree[0] == ( argfree[ARGHEAD]+ARGHEAD ) ) {
```

```
        Normalize(BHEAD argfree+ARGHEAD);
        argfree[0] = argfree[ARGHEAD]+ARGHEAD;
        argfree[1] = 0;
        if ( ( argfree[0] == ARGHEAD+4 ) && ( argfree[ARGHEAD+3] == 3 )
            && ( argfree[ARGHEAD+1] == 1 ) && ( argfree[ARGHEAD+2] == 1 ) ) {
            goto return0;
        }
    }
    else {
/*
        The way we took out objects is rather brutish. We have to
        normalize
*/
        NewSort(BHEAD0);
        t = argfree+ARGHEAD;
        while ( *t ) {
            tstop = t + *t;
            Normalize(BHEAD t);
            StoreTerm(BHEAD t);
            t = tstop;
        }
        EndSort(BHEAD argfree+ARGHEAD,0);
        t = argfree+ARGHEAD;
        while ( *t ) t += *t;
        *argfree = t - argfree;
    }
/*
    #] step 2 :
    #[ step 3 : look whether we have done this one already.
*/
    if ( ( number = FindArg(BHEAD argfree) ) != 0 ) {
        if ( number > 0 ) t = cbuf[AT.fbufnum].rhs[number-1];
        else              t = cbuf[AC.ffbufnum].rhs[-number-1];
/*
        Now position on the result. Remember we have in the cache:
                    inputarg,0,outputargs,0
        t is currently at inputarg. *inputarg is always positive.
        in principle this holds also for the arguments in the output
        but we take no risks here (in case of future developments).
*/
        t += *t; t++;
        tstop = t;
        ii = 0;
        while ( *tstop ) {
            if ( *tstop == -SNUMBER && tstop[1] == -1 ) {
                sign = -sign; ii += 2;
            }
            NEXTARG(tstop);
        }
        a = argout; while ( *a ) NEXTARG(a);
#ifndef NEWORDER
        if ( sign == -1 ) { *a++ = -SNUMBER; *a++ = -1; *a = 0; sign = 1; }
#endif
        i = tstop - t - ii;
        ii = a - argout;
        a2 = a; a1 = a + i;
        *a1 = 0;
        while ( ii > 0 ) { *--a1 = *--a2; ii--; }
        a = argout;
        while ( *t ) {
            if ( *t == -SNUMBER && t[1] == -1 ) { t += 2; }
            else { COPY1ARG(a,t) }
        }
        goto return0;
    }
/*
    #] step 3 :
    #[ step 4 : invoke ConvertToPoly

                We make a copy first in case there are no factors
*/
    argcopy = TermMalloc("argcopy");
    for ( i = 0; i <= *argfree; i++ ) argcopy[i] = argfree[i];

    tstop = argfree + *argfree;
    {
        WORD sumcommu = 0;
        t = argfree + ARGHEAD;
        while ( t < tstop ) {
            sumcommu += DoesCommu(t);
            t += *t;
        }
        if ( sumcommu > 1 ) {
            MesPrint("ERROR: Cannot factorize an argument with more than one noncommuting object");
            Terminate(-1);
        }
    }
```

```
    t = argfree + ARGHEAD;

    while ( t < tstop ) {
        if ( ( t[1] != SYMBOL ) && ( *t != (ABS(t[*t-1])+1) ) ) {
            action = 1; break;
        }
        t += *t;
    }
    if ( action ) {
        t = argfree + ARGHEAD;
        argextra = AT.WorkPointer;
        NewSort(BHEAD0);
        while ( t < tstop ) {
            if ( LocalConvertToPoly(BHEAD t,argextra,startebuf,0) < 0 ) {
                error = -1;
getout:
                AR.SortType = oldsorttype;
                TermFree(argcopy,"argcopy");
                if ( argfree != argin ) TermFree(argfree,"argfree");
                MesCall("ArgFactorize");
                Terminate(-1);
                return(-1);
            }
            StoreTerm(BHEAD argextra);
            t += *t; argextra += *argextra;
        }
        if ( EndSort(BHEAD argfree+ARGHEAD,0) ) { error = -2; goto getout; }
        t = argfree + ARGHEAD;
        while ( *t > 0 ) t += *t;
        *argfree = t - argfree;
    }
/*
    #] step 4 :
    #[ step 5 : If not in the tables, we have to do this by hard work.
*/

    a = argout;
    while ( *a ) NEXTARG(a);
    if ( poly_factorize_argument(BHEAD argfree,a) < 0 ) {
        MesCall("ArgFactorize");
        error = -1;
    }
/*
    #] step 5 :
    #[ step 6 : use now ConvertFromPoly

                Be careful: there should be more than one argument now.
*/
    if ( error == 0 && action ) {
      a1 = a; NEXTARG(a1);
      if ( *a1 != 0 ) {
        CBUF *C = cbuf+AC.cbufnum;
        CBUF *CC = cbuf+AT.ebufnum;
        WORD *oldworkpointer = AT.WorkPointer;
        WORD *argcopy2 = TermMalloc("argcopy2"), *a1, *a2;
        a1 = a; a2 = argcopy2;
        while ( *a1 ) {
            if ( *a1 < 0 ) {
                if ( *a1 > -FUNCTION ) *a2++ = *a1++;
                *a2++ = *a1++; *a2 = 0;
                continue;
            }
            t = a1 + ARGHEAD;
            tstop = a1 + *a1;
            argextra = AT.WorkPointer;
            NewSort(BHEAD0);
            while ( t < tstop ) {
                if ( ConvertFromPoly(BHEAD t,argextra,numxsymbol,CC->numrhs-startebuf+numxsymbol
                ,startebuf-numxsymbol,1) <= 0 ) {
                    TermFree(argcopy2,"argcopy2");
                    LowerSortLevel();
                    error = -3;
                    goto getout;
                }
                t += *t;
                AT.WorkPointer = argextra + *argextra;
/*
                ConvertFromPoly leaves terms with subexpressions. Hence:
*/
                if ( Generator(BHEAD argextra,C->numlhs) ) {
                    TermFree(argcopy2,"argcopy2");
                    LowerSortLevel();
                    error = -4;
                    goto getout;
                }
            }
            AT.WorkPointer = oldworkpointer;
```

```
            if ( EndSort(BHEAD a2+ARGHEAD,0) ) { error = -5; goto getout; }
            t = a2+ARGHEAD; while ( *t ) t += *t;
            *a2 = t - a2; a2[1] = 0; ZEROARG(a2);
            ToFast(a2,a2); NEXTARG(a2);
            a1 = tstop;
        }
        i = a2 - argcopy2;
        a2 = argcopy2; a1 = a;
        NCOPY(a1,a2,i);
        *a1 = 0;
        TermFree(argcopy2,"argcopy2");
/*
        Erase the entries we made temporarily in cbuf[AT.ebufnum]
*/
        CC->numrhs = startebuf;
      }
      else {    /* no factorization. recover the argument from before step 3. */
        for ( i = 0; i <= *argcopy; i++ ) a[i] = argcopy[i];
      }
    }
/*
    #] step 6 :
    #[ step 7 : Add this one to the tables.

                Possibly drop some elements in the tables
                when they become too full.
*/
    if ( error == 0 && AN.ncmod == 0 ) {
        if ( InsertArg(BHEAD argcopy,a,0) < 0 ) { error = -1; }
    }
/*
    #] step 7 :
    #[ step 8 : Clean up and return.

        Change the order of the arguments in argout and a.
        Use argcopy as spare space.
*/
    ii = a - argout;
    for ( i = 0; i < ii; i++ ) argcopy[i] = argout[i];
    a1 = a;
    while ( *a1 ) {
        if ( *a1 == -SNUMBER && a1[1] < 0 ) {
            sign = -sign; a1[1] = -a1[1];
            if ( a1[1] == 1 ) {
                a2 = a1+2; while ( *a2 ) NEXTARG(a2);
                i = a2-a1-2; a2 = a1+2;
                NCOPY(a1,a2,i);
                *a1 = 0;
            }
            while ( *a1 ) NEXTARG(a1);
            break;
        }
        else {
            if ( *a1 > 0 && *a1 == a1[ARGHEAD]+ARGHEAD && a1[*a1-1] < 0 ) {
                a1[*a1-1] = -a1[*a1-1]; sign = -sign;
            }
            if ( *a1 == ARGHEAD+4 && a1[ARGHEAD+1] == 1 && a1[ARGHEAD+2] == 1 ) {
                a2 = a1+ARGHEAD+4; while ( *a2 ) NEXTARG(a2);
                i = a2-a1-ARGHEAD-4; a2 = a1+ARGHEAD+4;
                NCOPY(a1,a2,i);
                *a1 = 0;
                break;
            }
            while ( *a1 ) NEXTARG(a1);
            break;
        }
        NEXTARG(a1);
    }
    i = a1 - a;
    a2 = argout;
    NCOPY(a2,a,i);
    for ( i = 0; i < ii; i++ ) *a2++ = argcopy[i];
#ifndef NEWORDER
    if ( sign == -1 ) { *a2++ = -SNUMBER; *a2++ = -1; sign = 1; }
#endif
    *a2 = 0;
    TermFree(argcopy,"argcopy");
return0:
    if ( argfree != argin ) TermFree(argfree,"argfree");
    if ( oldsorttype != AR.SortType ) {
        AR.SortType = oldsorttype;
        a = argout;
        while ( *a ) {
            if ( *a > 0 ) {
                NewSort(BHEAD0);
                oldword = a[*a]; a[*a] = 0;
                t = a+ARGHEAD;
```

```
                while ( *t ) {
                    tstop = t + *t;
                    StoreTerm(BHEAD t);
                    t = tstop;
                }
                EndSort(BHEAD a+ARGHEAD,0);
                a[*a] = oldword;
                a += *a;
            }
            else { NEXTARG(a); }
        }
    }
#ifdef NEWORDER
    t = argout; numargs = 0;
    while ( *t ) {
        tt = t;
        NEXTARG(t);
        if ( *tt == ABS(t[-1])+1+ARGHEAD && sign == -1 ) { t[-1] = -t[-1]; sign = 1; }
        else if ( *tt == -SNUMBER && sign == -1 ) { tt[1] = -tt[1]; sign = 1; }
        numargs++;
    }
    if ( sign == -1 ) {
        *t++ = -SNUMBER; *t++ = -1; *t = 0; sign = 1; numargs++;
    }
#else
/*
    Now we have to sort the arguments
    First have the number of 'nontrivial/nonnumerical' arguments
    Then make a piece of code like in FullSymmetrize with that number
    of arguments to be symmetrized.
    Put a function in front
    Call the Symmetrize routine
*/
    t = argout; numargs = 0;
    while ( *t && *t != -SNUMBER && ( *t < 0 || ( ABS(t[*t-1]) != *t-1 ) ) ) {
        NEXTARG(t);
        numargs++;
    }
#endif
    if ( numargs > 1 ) {
        WORD *Lijst;
        WORD x[3];
        x[0] = argout[-FUNHEAD];
        x[1] = argout[-FUNHEAD+1];
        x[2] = argout[-FUNHEAD+2];
        while ( *t ) { NEXTARG(t); }
        argout[-FUNHEAD] = SQRTFUNCTION;
        argout[-FUNHEAD+1] = t-argout+FUNHEAD;
        argout[-FUNHEAD+2] = 0;
        AT.WorkPointer = t+1;
        Lijst = AT.WorkPointer;
        for ( i = 0; i < numargs; i++ ) Lijst[i] = i;
        AT.WorkPointer += numargs;
        error = Symmetrize(BHEAD argout-FUNHEAD,Lijst,numargs,1,SYMMETRIC);
        AT.WorkPointer = Lijst;
        argout[-FUNHEAD] = x[0];
        argout[-FUNHEAD+1] = x[1];
        argout[-FUNHEAD+2] = x[2];
#ifdef NEWORDER
/*
        Now we have to get a potential numerical argument to the first position
*/
        tstop = argout; while ( *tstop ) { NEXTARG(tstop); }
        t = argout; number = 0;
        while ( *t ) {
            tt = t; NEXTARG(t);
            if ( *tt == -SNUMBER ) {
                if ( number == 0 ) break;
                x[0] = tt[1];
                while ( tt > argout ) { *--t = *--tt; }
                argout[0] = -SNUMBER; argout[1] = x[0];
                break;
            }
            else if ( *tt == ABS(t[-1])+1+ARGHEAD ) {
                if ( number == 0 ) break;
                ii = t - tt;
                for ( i = 0; i < ii; i++ ) tstop[i] = tt[i];
                while ( tt > argout ) { *--t = *--tt; }
                for ( i = 0; i < ii; i++ ) argout[i] = tstop[i];
                *tstop = 0;
                break;
            }
            number++;
        }
#endif
    }
/*
```

```
    #] step 8 :
*/
    return(error);
}

/*
    #] ArgFactorize :
    #[ FindArg :
*/
WORD FindArg(PHEAD WORD *a)
{
    int number;
    if ( AN.ncmod != 0 ) return(0); /* no room for mod stuff */
    number = FindTree(AT.fbufnum,a);
    if ( number >= 0 ) return(number+1);
    number = FindTree(AC.ffbufnum,a);
    if ( number >= 0 ) return(-number-1);
    return(0);
}

/*
    #] FindArg :
    #[ InsertArg :
*/
WORD InsertArg(PHEAD WORD *argin, WORD *argout,int par)
{
    CBUF *C;
    WORD *a, i, bufnum;
    if ( par == 0 ) {
        bufnum = AT.fbufnum;
        C = cbuf+bufnum;
        if ( C->numrhs >= (C->maxrhs-2) ) CleanupArgCache(BHEAD AT.fbufnum);
    }
    else if ( par == 1 ) {
        bufnum = AC.ffbufnum;
        C = cbuf+bufnum;
    }
    else { return(-1); }
    AddRHS(bufnum,1);
    AddNtoC(bufnum,*argin,argin,1);
    AddToCB(C,0)
    a = argout; while ( *a ) NEXTARG(a);
    i = a - argout;
    AddNtoC(bufnum,i,argout,2);
    AddToCB(C,0)
    return(InsTree(bufnum,C->numrhs));
}

/*
    #] InsertArg :
    #[ CleanupArgCache :
*/
int CleanupArgCache(PHEAD WORD bufnum)
{
    CBUF *C = cbuf + bufnum;
    COMPTREE *boomlijst = C->boomlijst;
    LONG *weights = (LONG *)Malloc1(2*(C->numrhs+1)*sizeof(LONG),"CleanupArgCache");
    LONG w, whalf, *extraweights;
    WORD *a, *to, *from;
    int i,j,k;
    for ( i = 1; i <= C->numrhs; i++ ) {
        weights[i] = ((LONG)i) * (boomlijst[i].usage);
    }
/*
        Now sort the weights and determine the halfway weight
*/
    extraweights = weights+C->numrhs+1;
    SortWeights(weights+1,extraweights,C->numrhs);
    whalf = weights[C->numrhs/2+1];
/*
        We should drop everybody with a weight < whalf.
*/
    to = C->Buffer;
    k = 1;
    for ( i = 1; i <= C->numrhs; i++ ) {
        from = C->rhs[i]; w = ((LONG)i) * (boomlijst[i].usage);
        if ( w >= whalf ) {
            if ( i < C->numrhs-1 ) {
                if ( to == from ) {
                    to = C->rhs[i+1];
                }
                else {
                    j = C->rhs[i+1] - from;
                    C->rhs[k] = to;
                    NCOPY(to,from,j)
                }
            }
```

```
            else if ( to == from ) {
                to += *to + 1; while ( *to ) NEXTARG(to); to++;
            }
            else {
                a = from; a += *a+1; while ( *a ) NEXTARG(a); a++;
                j = a - from;
                C->rhs[k] = to;
                NCOPY(to,from,j)
            }
            weights[k++] = boomlijst[i].usage;
        }
    }
    C->numrhs = --k;
    C->Pointer = to;
/*
        Next we need to rebuild the tree.
        Note that this can probably be done much faster by using the
        remains of the old tree !!!!!!!!!!!!!!!
*/
    ClearTree(AT.fbufnum);
    for ( i = 1; i <= k; i++ ) {
        InsTree(AT.fbufnum,i);
        boomlijst[i].usage = weights[i];
    }
/*
        And cleanup
*/
    M_free(weights,"CleanupArgCache");
    return(0);
}

/*
    #] CleanupArgCache :
    #[ ArgSymbolMerge :
*/

int ArgSymbolMerge(WORD *t1, WORD *t2)
{
    WORD *t1e = t1+t1[1];
    WORD *t2e = t2+t2[1];
    WORD *t1a = t1+2;
    WORD *t2a = t2+2;
    WORD *t3;
    while ( t1a < t1e && t2a < t2e ) {
        if ( *t1a < *t2a ) {
            if ( t1a[1] >= 0 ) {
                t3 = t1a+2;
                while ( t3 < t1e ) { t3[-2] = *t3; t3[-1] = t3[1]; t3 += 2; }
                t1e -= 2;
            }
            else t1a += 2;
        }
        else if ( *t1a > *t2a ) {
            if ( t2a[1] >= 0 ) t2a += 2;
            else {
                t3 = t1e;
                while ( t3 > t1a ) { *t3 = t3[-2]; t3[1] = t3[-1]; t3 -= 2; }
                *t1a++ = *t2a++;
                *t1a++ = *t2a++;
                t1e += 2;
            }
        }
        else {
            if ( t2a[1] < t1a[1] ) t1a[1] = t2a[1];
            t1a += 2; t2a += 2;
        }
    }
    while ( t2a < t2e ) {
        if ( t2a[1] < 0 ) {
            *t1a++ = *t2a++;
            *t1a++ = *t2a++;
        }
        else t2a += 2;
    }
    while ( t1a < t1e ) {
        if ( t1a[1] >= 0 ) {
            t3 = t1a+2;
            while ( t3 < t1e ) { t3[-2] = *t3; t3[-1] = t3[1]; t3 += 2; }
            t1e -= 2;
        }
        else t1a += 2;
    }
    t1[1] = t1a - t1;
    return(0);
}

/*
```

```
    #] ArgSymbolMerge :
    #[ ArgDotproductMerge :
*/

int ArgDotproductMerge(WORD *t1, WORD *t2)
{
    WORD *t1e = t1+t1[1];
    WORD *t2e = t2+t2[1];
    WORD *t1a = t1+2;
    WORD *t2a = t2+2;
    WORD *t3;
    while ( t1a < t1e && t2a < t2e ) {
        if ( *t1a < *t2a || ( *t1a == *t2a && t1a[1] < t2a[1] ) ) {
            if ( t1a[2] >= 0 ) {
                t3 = t1a+3;
                while ( t3 < t1e ) { t3[-3] = *t3; t3[-2] = t3[1]; t3[-1] = t3[2]; t3 += 3; }
                t1e -= 3;
            }
            else t1a += 3;
        }
        else if ( *t1a > *t2a || ( *t1a == *t2a && t1a[1] > t2a[1] ) ) {
            if ( t2a[2] >= 0 ) t2a += 3;
            else {
                t3 = t1e;
                while ( t3 > t1a ) { *t3 = t3[-3]; t3[1] = t3[-2]; t3[2] = t3[-1]; t3 -= 3; }
                *t1a++ = *t2a++;
                *t1a++ = *t2a++;
                *t1a++ = *t2a++;
                t1e += 3;
            }
        }
        else {
            if ( t2a[2] < t1a[2] ) t1a[2] = t2a[2];
            t1a += 3; t2a += 3;
        }
    }
    while ( t2a < t2e ) {
        if ( t2a[2] < 0 ) {
            *t1a++ = *t2a++;
            *t1a++ = *t2a++;
            *t1a++ = *t2a++;
        }
        else t2a += 3;
    }
    while ( t1a < t1e ) {
        if ( t1a[2] >= 0 ) {
            t3 = t1a+3;
            while ( t3 < t1e ) { t3[-3] = *t3; t3[-2] = t3[1]; t3[-1] = t3[2]; t3 += 3; }
            t1e -= 3;
        }
        else t1a += 2;
    }
    t1[1] = t1a - t1;
    return(0);
}

/*
    #] ArgDotproductMerge :
    #[ TakeArgContent :
*/
WORD *TakeArgContent(PHEAD WORD *argin, WORD *argout)
{
    GETBIDENTITY
    WORD *t, *rnext, *r1, *r2, *r3, *r5, *r6, *r7, *r8, *r9;
    WORD pow, *mm, *mnext, *mstop, *argin2 = argin, *argin3 = argin, *argfree;
    WORD ncom;
    int j, i, act;
    r5 = t = argin + ARGHEAD;
    r3 = argin + *argin;
    rnext = t + *t;
    GETSTOP(t,r6);
    r1 = argout;
    t++;
/*
        First pass: arrange everything but the symbols and dotproducts.
        They need separate treatment because we have to take out negative
        powers.
*/
    while ( t < r6 ) {
/*
            #[ DELTA/VECTOR :
*/
        if ( *t == DELTA || *t == VECTOR ) {
            r7 = t; r8 = t + t[1]; t += 2;
            while ( t < r8 ) {
                mm = rnext;
                pow = 1;
```

```
                while ( mm < r3 ) {
                    mnext = mm + *mm;
                    GETSTOP(mm,mstop); mm++;
                    while ( mm < mstop ) {
                        if ( *mm != *r7 ) mm += mm[1];
                        else break;
                    }
                    if ( *mm == *r7 ) {
                        mstop = mm + mm[1]; mm += 2;
                        while ( ( *mm != *t || mm[1] != t[1] )
                            && mm < mstop ) mm += 2;
                        if ( mm >= mstop ) pow = 0;
                    }
                    else pow = 0;
                    if ( pow == 0 ) break;
                    mm = mnext;
                }
                if ( pow == 0 ) { t += 2; continue; }
/*
                We have a factor
*/
                *r1++ = 8 + ARGHEAD;
                for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
                *r1++ = 8; *r1++ = *r7;
                *r1++ = 4; *r1++ = *t; *r1++ = t[1];
                *r1++ = 1; *r1++ = 1; *r1++ = 3;
                argout = r1;
/*
                Now we have to remove the delta's/vectors
*/
                mm = rnext;
                while ( mm < r3 ) {
                    mnext = mm + *mm;
                    GETSTOP(mm,mstop); mm++;
                    while ( mm < mstop ) {
                        if ( *mm != *r7 ) mm += mm[1];
                        else break;
                    }
                    mstop = mm + mm[1]; mm += 2;
                    while ( mm < mstop && (
                     *mm != *t || mm[1] != t[1] ) ) mm += 2;
                    *mm = mm[1] = NOINDEX;
                    mm = mnext;
                }
                *t = t[1] = NOINDEX;
                t += 2;
            }
        }
/*
            #] DELTA/VECTOR :
            #[ INDEX :
*/
        else if ( *t == INDEX ) {
            r7 = t; r8 = t + t[1]; t += 2;
            while ( t < r8 ) {
                mm = rnext;
                pow = 1;
                while ( mm < r3 ) {
                    mnext = mm + *mm;
                    GETSTOP(mm,mstop); mm++;
                    while ( mm < mstop ) {
                        if ( *mm != *r7 ) mm += mm[1];
                        else break;
                    }
                    if ( *mm == *r7 ) {
                        mstop = mm + mm[1]; mm += 2;
                        while ( *mm != *t
                            && mm < mstop ) mm++;
                        if ( mm >= mstop ) pow = 0;
                    }
                    else pow = 0;
                    if ( pow == 0 ) break;
                    mm = mnext;
                }
                if ( pow == 0 ) { t++; continue; }
/*
                We have a factor
*/
                if ( *t < 0 ) { *r1++ = -VECTOR; }
                else          { *r1++ = -INDEX; }
                *r1++ = *t;
                argout = r1;
/*
                Now we have to remove the index
*/
                *t = NOINDEX;
                mm = rnext;
```

```
                while ( mm < r3 ) {
                    mnext = mm + *mm;
                    GETSTOP(mm,mstop); mm++;
                    while ( mm < mstop ) {
                        if ( *mm != *r7 ) mm += mm[1];
                        else break;
                    }
                    mstop = mm + mm[1]; mm += 2;
                    while ( mm < mstop &&
                     *mm != *t ) mm += 1;
                    *mm = NOINDEX;
                    mm = mnext;
                }
                t += 1;
            }
        }
/*
            #] INDEX :
            #[ FUNCTION :
*/
        else if ( *t >= FUNCTION ) {
/*
            In the next code we should actually look inside
            the DENOMINATOR or EXPONENT for noncommuting objects
*/
            if ( *t >= FUNCTION &&
                functions[*t-FUNCTION].commute == 0 ) ncom = 0;
            else ncom = 1;
            if ( ncom ) {
                mm = r5 + 1;
                while ( mm < t && ( *mm == DUMMYFUN
                || *mm == DUMMYTEN ) ) mm += mm[1];
                if ( mm < t ) { t += t[1]; continue; }
            }
            mm = rnext; pow = 1;
            while ( mm < r3 ) {
                mnext = mm + *mm;
                GETSTOP(mm,mstop); mm++;
                while ( mm < mstop ) {
                    if ( *mm == *t && mm[1] == t[1] ) {
                        for ( i = 2; i < t[1]; i++ ) {
                            if ( mm[i] != t[i] ) break;
                        }
                        if ( i >= t[1] )
                            { mm += mm[1]; goto nextmterm; }
                    }
                    if ( ncom && *mm != DUMMYFUN && *mm != DUMMYTEN )
                        { pow = 0; break; }
                    mm += mm[1];
                }
                if ( mm >= mstop ) pow = 0;
                if ( pow == 0 ) break;
nextmterm:      mm = mnext;
            }
            if ( pow == 0 ) { t += t[1]; continue; }
/*
            Copy the function
*/
            *r1++ = t[1] + 4 + ARGHEAD;
            for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
            *r1++ = t[1] + 4;
            for ( i = 0; i < t[1]; i++ ) *r1++ = t[i];
            *r1++ = 1; *r1++ = 1; *r1++ = 3;
            argout = r1;
/*
            Now we have to take out the functions
*/
            mm = rnext;
            while ( mm < r3 ) {
                mnext = mm + *mm;
                GETSTOP(mm,mstop); mm++;
                while ( mm < mstop ) {
                    if ( *mm == *t && mm[1] == t[1] ) {
                        for ( i = 2; i < t[1]; i++ ) {
                            if ( mm[i] != t[i] ) break;
                        }
                        if ( i >= t[1] ) {
                            if ( functions[*t-FUNCTION].spec > 0 )
                                *mm = DUMMYTEN;
                            else
                                *mm = DUMMYFUN;
                            mm += mm[1];
                            goto nextterm;
                        }
                    }
                    mm += mm[1];
                }
```

```
nextterm:                       mm = mnext;
            }
            if ( functions[*t-FUNCTION].spec > 0 )
                    *t = DUMMYTEN;
            else
                    *t = DUMMYFUN;
            t += t[1];
        }
/*
            #] FUNCTION :
*/
        else {
            t += t[1];
        }
    }
/*
            #[ SYMBOL :

        Now collect all symbols. We can use the space after r1 as storage
*/
    t = argin+ARGHEAD;
    rnext = t + *t;
    r2 = r1;
    while ( t < r3 ) {
        GETSTOP(t,r6);
        t++;
        act = 0;
        while ( t < r6 ) {
            if ( *t == SYMBOL ) {
                act = 1;
                i = t[1];
                NCOPY(r2,t,i)
            }
            else { t += t[1]; }
        }
        if ( act == 0 ) {
            *r2++ = SYMBOL; *r2++ = 2;
        }
        t = rnext; rnext = rnext + *rnext;
    }
    *r2 = 0;
    argin2 = argin;
/*
        Now we have a list of all symbols as a sequence of SYMBOL subterms.
        Any symbol that is absent in a subterm has power zero.
        We now need a list of all minimum powers.
        This can be done by subsequent merges.
*/
    r7 = r1;          /* The first object into which we merge. */
    r8 = r7 + r7[1];  /* The object that gets merged into r7.  */
    while ( *r8 ) {
        r2 = r8 + r8[1]; /* Next object */
        ArgSymbolMerge(r7,r8);
        r8 = r2;
    }
/*
        Now we have to divide by the object in r7 and take it apart as factors.
        The division can be simple if there are no negative powers.
*/
    if ( r7[1] > 2 ) {
        r8 = r7+2;
        r2 = r7 + r7[1];
        act = 0;
        pow = 0;
        while ( r8 < r2 ) {
            if ( r8[1] < 0 ) { act = 1; pow += -r8[1]*(ARGHEAD+8); }
            else { pow += 2*r8[1]; }
            r8 += 2;
        }
/*
        The amount of space we need to move r7 is given in pow
*/
        if ( act == 0 ) {   /* this can be done 'in situ' */
            t = argin + ARGHEAD;
            while ( t < r3 ) {
                rnext = t + *t;
                GETSTOP(t,r6);
                t++;
                while ( t < r6 ) {
                    if ( *t != SYMBOL ) { t += t[1]; continue; }
                    r8 = r7+2; r9 = t + t[1]; t += 2;
                    while ( ( t < r9 ) && ( r8 < r2 ) ) {
                        if ( *t == *r8 ) {
                            t[1] -= r8[1]; t += 2; r8 += 2;
                        }
                        else { /* *t must be < than *r8 !!! */
                            t += 2;
```

```
                        }
                    }
                    t = r9;
                }
                t = rnext;
            }
/*
            And now the factors that go to argout.
            First we have to move r7 out of the way.
*/
            r8 = r7+pow; i = r7[1];
            while ( --i >= 0 ) r8[i] = r7[i];
            r2 += pow;
            r8 += 2;
            while ( r8 < r2 ) {
                for ( i = 0; i < r8[1]; i++ ) { *r1++ = -SYMBOL; *r1++ = *r8; }
                r8 += 2;
            }
        }
        else {  /* this needs a new location */
            argin2 = TermMalloc("TakeArgContent2");
/*
            We have to multiply the inverse of r7 into argin
            The answer should go to argin2.
*/
            r5 = argin2; *r5++ = 0; *r5++ = 0; FILLARG(r5);
            t = argin+ARGHEAD;
            while ( t < r3 ) {
                rnext = t + *t;
                GETSTOP(t,r6);
                r9 = r5;
                *r5++ = *t++ + r7[1];
                while ( t < r6 ) *r5++ = *t++;
                i = r7[1] - 2; r8 = r7+2;
                *r5++ = r7[0]; *r5++ = r7[1];
                while ( i > 0 ) { *r5++ = *r8++; *r5++ = -*r8++; i -= 2; }
                while ( t < rnext ) *r5++ = *t++;
                Normalize(BHEAD r9);
                r5 = r9 + *r9;
            }
            *r5 = 0;
            *argin2 = r5-argin2;
/*
            We may have to sort the terms in argin2.
*/
            NewSort(BHEAD0);
            t = argin2+ARGHEAD;
            while ( *t ) {
                StoreTerm(BHEAD t);
                t += *t;
            }
            t = argin2+ARGHEAD;
            if ( EndSort(BHEAD t,0) < 0 ) goto Irreg;
            while ( *t ) t += *t;
            *argin2 = t - argin2;
            r3 = t;
/*
            And now the factors that go to argout.
            First we have to move r7 out of the way.
*/
            r8 = r7+pow; i = r7[1];
            while ( --i >= 0 ) r8[i] = r7[i];
            r2 += pow;
            r8 += 2;
            while ( r8 < r2 ) {
                if ( r8[1] >= 0 ) {
                    for ( i = 0; i < r8[1]; i++ ) { *r1++ = -SYMBOL; *r1++ = *r8; }
                }
                else {
                    for ( i = 0; i < -r8[1]; i++ ) {
                        *r1++ = ARGHEAD+8; *r1++ = 0;
                        FILLARG(r1);
                        *r1++ = 8; *r1++ = SYMBOL; *r1++ = 4; *r1++ = *r8;
                        *r1++ = -1; *r1++ = 1; *r1++ = 1; *r1++ = 3;
                    }
                }
                r8 += 2;
            }
            argout = r1;
        }
    }
/*
            #] SYMBOL :
            #[ DOTPRODUCT :

        Now collect all dotproducts. We can use the space after r1 as storage
*/
```

```
      t = argin2+ARGHEAD;
      rnext = t + *t;
      r2 = r1;
      while ( t < r3 ) {
        GETSTOP(t,r6);
        t++;
        act = 0;
        while ( t < r6 ) {
            if ( *t == DOTPRODUCT ) {
                act = 1;
                i = t[1];
                NCOPY(r2,t,i)
            }
            else { t += t[1]; }
        }
        if ( act == 0 ) {
            *r2++ = DOTPRODUCT; *r2++ = 2;
        }
        t = rnext; rnext = rnext + *rnext;
      }
      *r2 = 0;
      argin3 = argin2;
/*
        Now we have a list of all dotproducts as a sequence of DOTPRODUCT
        subterms. Any dotproduct that is absent in a subterm has power zero.
        We now need a list of all minimum powers.
        This can be done by subsequent merges.
*/
      r7 = r1;          /* The first object into which we merge. */
      r8 = r7 + r7[1];  /* The object that gets merged into r7.  */
      while ( *r8 ) {
        r2 = r8 + r8[1]; /* Next object */
        ArgDotproductMerge(r7,r8);
        r8 = r2;
      }
/*
        Now we have to divide by the object in r7 and take it apart as factors.
        The division can be simple if there are no negative powers.
*/
      if ( r7[1] > 2 ) {
        r8 = r7+2;
        r2 = r7 + r7[1];
        act = 0;
        pow = 0;
        while ( r8 < r2 ) {
            if ( r8[2] < 0 ) { pow += -r8[2]*(ARGHEAD+9); }
            else             { pow +=  r8[2]*(ARGHEAD+9); }
            r8 += 3;
        }
/*
        The amount of space we need to move r7 is given in pow
        For dotproducts we always need a new location
*/
        {
            argin3 = TermMalloc("TakeArgContent3");
/*
            We have to multiply the inverse of r7 into argin
            The answer should go to argin2.
*/
            r5 = argin3; *r5++ = 0; *r5++ = 0; FILLARG(r5);
            t = argin2+ARGHEAD;
            while ( t < r3 ) {
                rnext = t + *t;
                GETSTOP(t,r6);
                r9 = r5;
                *r5++ = *t++ + r7[1];
                while ( t < r6 ) *r5++ = *t++;
                i = r7[1] - 2; r8 = r7+2;
                *r5++ = r7[0]; *r5++ = r7[1];
                while ( i > 0 ) { *r5++ = *r8++; *r5++ = *r8++; *r5++ = -*r8++; i -= 3; }
                while ( t < rnext ) *r5++ = *t++;
                Normalize(BHEAD r9);
                r5 = r9 + *r9;
            }
            *r5 = 0;
            *argin3 = r5-argin3;
/*
            We may have to sort the terms in argin3.
*/
            NewSort(BHEAD0);
            t = argin3+ARGHEAD;
            while ( *t ) {
                StoreTerm(BHEAD t);
                t += *t;
            }
            t = argin3+ARGHEAD;
            if ( EndSort(BHEAD t,0) < 0 ) goto Irreg;
```

```
            while ( *t ) t += *t;
            *argin3 = t - argin3;
            r3 = t;
/*
            And now the factors that go to argout.
            First we have to move r7 out of the way.
*/
            r8 = r7+pow; i = r7[1];
            while ( --i >= 0 ) r8[i] = r7[i];
            r2 += pow;
            r8 += 2;
            while ( r8 < r2 ) {
                for ( i = ABS(r8[2]); i > 0; i-- ) {
                    *r1++ = ARGHEAD+9; *r1++ = 0; FILLARG(r1);
                    *r1++ = 9; *r1++ = DOTPRODUCT; *r1++ = 5; *r1++ = *r8;
                    *r1++ = r8[1]; *r1++ = r8[2] < 0 ? -1: 1;
                    *r1++ = 1; *r1++ = 1; *r1++ = 3;
                }
                r8 += 3;
            }
            argout = r1;
        }
      }
/*
            #] DOTPRODUCT :

    We have now in argin3 the argument stripped of negative powers and
    common factors. The only thing left to deal with is to make the
    coefficients integer. For that we have to find the LCM of the denominators
    and the GCD of the numerators. And to start with, the sign.
    We force the sign of the first term to be positive.
*/
    t = argin3 + ARGHEAD; pow = 1;
    t += *t;
    if ( t[-1] < 0 ) {
        pow = 0;
        t[-1] = -t[-1];
        while ( t < r3 ) {
            t += *t; t[-1] = -t[-1];
        }
    }
/*
    Now the GCD of the numerators and the LCM of the denominators:
*/
    argfree = TermMalloc("TakeArgContent1");
    if ( AN.cmod != 0 ) {
        r1 = MakeMod(BHEAD argin3,r1,argfree);
    }
    else {
        r1 = MakeInteger(BHEAD argin3,r1,argfree);
    }
    if ( pow == 0 ) {
        *r1++ = -SNUMBER;
        *r1++ = -1;
    }
    *r1 = 0;
/*
    Cleanup
*/
    if ( argin3 != argin2 ) TermFree(argin3,"TakeArgContent3");
    if ( argin2 != argin  ) TermFree(argin2,"TakeArgContent2");
    return(argfree);
Irreg:
    MesPrint("Irregularity while sorting argument in TakeArgContent");
    if ( argin3 != argin2 ) TermFree(argin3,"TakeArgContent3");
    if ( argin2 != argin  ) TermFree(argin2,"TakeArgContent2");
    Terminate(-1);
    return(0);
}

/*
    #] TakeArgContent :
    #[ MakeInteger :
*/
WORD *MakeInteger(PHEAD WORD *argin,WORD *argout,WORD *argfree)
{
    UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
    WORD *r, *r1, *r2, *r3, *r4, *r5, *rnext, i, k, j;
    WORD kGCD, kLCM, kGCD2, kkLCM, jLCM, jGCD;
    GCDbuffer = NumberMalloc("MakeInteger");
    GCDbuffer2 = NumberMalloc("MakeInteger");
    LCMbuffer = NumberMalloc("MakeInteger");
    LCMb = NumberMalloc("MakeInteger");
    LCMc = NumberMalloc("MakeInteger");
    r4 = argin + *argin;
    r = argin + ARGHEAD;
/*
```

```
    First take the first term to load up the LCM and the GCD
*/
    r2 = r + *r;
    j = r2[-1];
    r3 = r2 - ABS(j);
    k = REDLENG(j);
    if ( k < 0 ) k = -k;
    while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
    for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
    k = REDLENG(j);
    if ( k < 0 ) k = -k;
    r3 += k;
    while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
    for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
    r1 = r2;
/*
    Now go through the rest of the terms in this argument.
*/
    while ( r1 < r4 ) {
        r2 = r1 + *r1;
        j = r2[-1];
        r3 = r2 - ABS(j);
        k = REDLENG(j);
        if ( k < 0 ) k = -k;
        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
        if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
/*
            GCD is already 1
*/
        }
        else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
            if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
                NumberFree(GCDbuffer,"MakeInteger");
                NumberFree(GCDbuffer2,"MakeInteger");
                NumberFree(LCMbuffer,"MakeInteger");
                NumberFree(LCMb,"MakeInteger"); NumberFree(LCMc,"MakeInteger");
                goto MakeIntegerErr;
            }
            kGCD = kGCD2;
            for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
        }
        else {
            kGCD = 1; GCDbuffer[0] = 1;
        }
        k = REDLENG(j);
        if ( k < 0 ) k = -k;
        r3 += k;
        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
        if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
            for ( kLCM = 0; kLCM < k; kLCM++ )
                LCMbuffer[kLCM] = r3[kLCM];
        }
        else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
            if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
                NumberFree(GCDbuffer,"MakeInteger"); NumberFree(GCDbuffer2,"MakeInteger");
                NumberFree(LCMbuffer,"MakeInteger"); NumberFree(LCMb,"MakeInteger");
NumberFree(LCMc,"MakeInteger");
                goto MakeIntegerErr;
            }
            DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
            MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
            for ( kLCM = 0; kLCM < jLCM; kLCM++ )
                LCMbuffer[kLCM] = LCMc[kLCM];
        }
        else {} /* LCM doesn't change */
        r1 = r2;
    }
/*
    Now put the factor together: GCD/LCM
*/
    r3 = (WORD *)(GCDbuffer);
    if ( kGCD == kLCM ) {
        for ( jGCD = 0; jGCD < kGCD; jGCD++ )
            r3[jGCD+kGCD] = LCMbuffer[jGCD];
        k = kGCD;
    }
    else if ( kGCD > kLCM ) {
        for ( jGCD = 0; jGCD < kLCM; jGCD++ )
            r3[jGCD+kGCD] = LCMbuffer[jGCD];
        for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
            r3[jGCD+kGCD] = 0;
        k = kGCD;
    }
    else {
        for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
            r3[jGCD] = 0;
        for ( jGCD = 0; jGCD < kLCM; jGCD++ )
```

```
            r3[jGCD+kLCM] = LCMbuffer[jGCD];
        k = kLCM;
    }
    j = 2*k+1;
/*
    Now we have to write this to argout
*/
    if ( ( j == 3 ) && ( r3[1] == 1 ) && ( (WORD)(r3[0]) > 0 ) ) {
        *argout = -SNUMBER;
        argout[1] = r3[0];
        r1 = argout+2;
    }
    else {
        r1 = argout;
        *r1++ = j+1+ARGHEAD; *r1++ = 0; FILLARG(r1);
        *r1++ = j+1; r2 = r3;
        for ( i = 0; i < k; i++ ) { *r1++ = *r2++; *r1++ = *r2++; }
        *r1++ = j;
    }
/*
    Next we have to take the factor out from the argument.
    This cannot be done in location, because the denominator stuff can make
    coefficients longer.
*/
    r2 = argfree + 2; FILLARG(r2)
    while ( r < r4 ) {
        rnext = r + *r;
        j = ABS(rnext[-1]);
        r5 = rnext - j;
        r3 = r2;
        while ( r < r5 ) *r2++ = *r++;
        j = (j-1)/2;    /* reduced length. Remember, k is the other red length */
        if ( DivRat(BHEAD (UWORD *)r5,j,GCDbuffer,k,(UWORD *)r2,&i) ) {
            goto MakeIntegerErr;
        }
        i = 2*i+1;
        r2 = r2 + i;
        if ( rnext[-1] < 0 ) r2[-1] = -i;
        else                 r2[-1] =  i;
        *r3 = r2-r3;
        r = rnext;
    }
    *r2 = 0;
    argfree[0] = r2-argfree;
    argfree[1] = 0;
/*
    Cleanup
*/
    NumberFree(LCMc,"MakeInteger");
    NumberFree(LCMb,"MakeInteger");
    NumberFree(LCMbuffer,"MakeInteger");
    NumberFree(GCDbuffer2,"MakeInteger");
    NumberFree(GCDbuffer,"MakeInteger");
    return(r1);

MakeIntegerErr:
    MesCall("MakeInteger");
    Terminate(-1);
    return(0);
}

/*
    #] MakeInteger :
    #[ MakeMod :
*/
WORD *MakeMod(PHEAD WORD *argin,WORD *argout,WORD *argfree)
{
    WORD *r, *instop, *r1, *m, x, xx, ix, ip;
    int i;
    r = argin; instop = r + *r; r += ARGHEAD;
    x = r[*r-3];
    if ( r[*r-1] < 0 ) x += AN.cmod[0];
    if ( GetModInverses(x,(WORD)(AN.cmod[0]),&ix,&ip) ) {
        Terminate(-1);
    }
    argout[0] = -SNUMBER;
    argout[1] = x;
    argout[2] = 0;
    r1 = argout+2;
/*
    Now we have to multiply all coefficients by ix.
    This does not make things longer, but we should keep to the conventions
    of MakeInteger.
*/
    m = argfree + ARGHEAD;
    while ( r < instop ) {
        xx = r[*r-3]; if ( r[*r-1] < 0 ) xx += AN.cmod[0];
```

```
        xx = (WORD)((((LONG)xx)*ix) % AN.cmod[0]);
        if ( xx != 0 ) {
            i = *r; NCOPY(m,r,i);
            m[-3] = xx; m[-1] = 3;
        }
        else { r += *r; }
    }
    *m = 0;
    *argfree = m - argfree;
    argfree[1] = 0;
    argfree += 2; FILLARG(argfree);
    return(r1);
}

/*
    #] MakeMod :
    #[ SortWeights :
*/
void SortWeights(LONG *weights,LONG *extraspace,WORD number)
{
    LONG w, *fill, *from1, *from2;
    int n1,n2,i;
    if ( number >= 4 ) {
        n1 = number/2; n2 = number - n1;
        SortWeights(weights,extraspace,n1);
        SortWeights(weights+n1,extraspace,n2);
/*
        We copy the first patch to the extra space. Then we merge
        Note that a potential remaining n2 objects are already in place.
*/
        for ( i = 0; i < n1; i++ ) extraspace[i] = weights[i];
        fill = weights; from1 = extraspace; from2 = weights+n1;
        while ( n1 > 0 && n2 > 0 ) {
            if ( *from1 <= *from2 ) { *fill++ = *from1++; n1--; }
            else                    { *fill++ = *from2++; n2--; }
        }
        while ( n1 > 0 ) { *fill++ = *from1++; n1--; }
    }
/*
    Special cases
*/
    else if ( number == 3 ) { /* 6 permutations of which one is trivial */
        if ( weights[0] > weights[1] ) {
            if ( weights[1] > weights[2] ) {
                w = weights[0]; weights[0] = weights[2]; weights[2] = w;
            }
            else if ( weights[0] > weights[2] ) {
                w = weights[0]; weights[0] = weights[1];
                weights[1] = weights[2]; weights[2] = w;
            }
            else {
                w = weights[0]; weights[0] = weights[1]; weights[1] = w;
            }
        }
        else if ( weights[0] > weights[2] ) {
            w = weights[0]; weights[0] = weights[2];
            weights[2] = weights[1]; weights[1] = w;
        }
        else if ( weights[1] > weights[2] ) {
            w = weights[1]; weights[1] = weights[2]; weights[2] = w;
        }
    }
    else if ( number == 2 ) {
        if ( weights[0] > weights[1] ) {
            w = weights[0]; weights[0] = weights[1]; weights[1] = w;
        }
    }
    return;
}

/*
    #] SortWeights :
*/
```

4.3 bugtool.c File Reference

`#include "form3.h"`
Include dependency graph for bugtool.c:

Functions

- void ExprStatus (EXPRESSIONS e)

4.3.1 Detailed Description

Low level routines for debugging

Definition in file bugtool.c.

4.3.2 Function Documentation

4.3.2.1 ExprStatus()

```
void ExprStatus (
            EXPRESSIONS e )
```

Definition at line 66 of file bugtool.c.

4.4 bugtool.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
```

```
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

/*
    #[ Includes :
*/

#include "form3.h"

/*
    #] Includes :
    #[ ExprStatus :
*/

static UBYTE *statusexpr[] = {
     (UBYTE *)"LOCALEXPRESSION"
    ,(UBYTE *)"SKIPLEXPRESSION"
    ,(UBYTE *)"DROPLEXPRESSION"
    ,(UBYTE *)"DROPPEDEXPRESSION"
    ,(UBYTE *)"GLOBALEXPRESSION"
    ,(UBYTE *)"SKIPGEXPRESSION"
    ,(UBYTE *)"DROPGEXPRESSION"
    ,(UBYTE *)"UNKNOWN"
    ,(UBYTE *)"STOREDEXPRESSION"
    ,(UBYTE *)"HIDDENLEXPRESSION"
    ,(UBYTE *)"HIDELEXPRESSION"
    ,(UBYTE *)"DROPHLEXPRESSION"
    ,(UBYTE *)"UNHIDELEXPRESSION"
    ,(UBYTE *)"HIDDENGEXPRESSION"
    ,(UBYTE *)"HIDEGEXPRESSION"
    ,(UBYTE *)"DROPHGEXPRESSION"
    ,(UBYTE *)"UNHIDEGEXPRESSION"
    ,(UBYTE *)"INTOHIDELEXPRESSION"
    ,(UBYTE *)"INTOHIDEGEXPRESSION"
};

void ExprStatus(EXPRESSIONS e)
{
    MesPrint("Expression %s(%d) has status %s(%d,%d). Buffer: %d, Position: %15p",
        AC.exprnames->namebuffer+e->name,(WORD)(e-Expressions),
        statusexpr[e->status],e->status,e->hidelevel,
        e->whichbuffer,&(e->onfile));
}

/*
    #] ExprStatus :
*/
```

4.5 checkpoint.c File Reference

```
#include "form3.h"
#include <errno.h>
```

Include dependency graph for checkpoint.c:

Macros

- #define CACHED_SNAPSHOT
- #define CACHE_SIZE 4096
- #define R_FREE(ARG) if (ARG) M_free(ARG, #ARG);
- #define R_FREE_NAMETREE(ARG)
- #define R_FREE_STREAM(ARG)
- #define R_SET(VAR, TYPE) VAR = ∗((TYPE∗)p); p = (unsigned char∗)p + sizeof(TYPE);
- #define R_COPY_B(VAR, SIZE, CAST)
- #define S_WRITE_B(BUF, LEN) if (fwrite_cached(BUF, 1, LEN, fd) != (size_t)(LEN)) return(__LINE__);
- #define S_FLUSH_B if (flush_cache(fd) != 1) return(__LINE__);
- #define R_COPY_S(VAR, CAST)
- #define S_WRITE_S(STR)
- #define R_COPY_LIST(ARG)
- #define S_WRITE_LIST(LST)
- #define R_COPY_NAMETREE(ARG)
- #define S_WRITE_NAMETREE(ARG)
- #define S_WRITE_DOLLAR(ARG)
- #define ANNOUNCE(str)

Functions

- int CheckRecoveryFile ()
- void DeleteRecoveryFile ()
- char ∗ RecoveryFilename ()
- void InitRecovery ()
- size_t fwrite_cached (const void ∗ptr, size_t size, size_t nmemb, FILE ∗fd)
- size_t flush_cache (FILE ∗fd)
- int DoRecovery (int ∗moduletype)
- void DoCheckpoint (int moduletype)

Variables

- unsigned char cache_buffer [CACHE_SIZE]
- size_t cache_fill = 0

4.5.1 Detailed Description

Contains all functions that deal with the recovery mechanism controlled and activated by the On Checkpoint switch.

The main function are DoCheckpoint, DoRecovery, and DoSnapshot. If the checkpoints are activated DoCheckpoint is called every time a module is finished executing. If the conditions for the creation of a recovery snapshot are met DoCheckpoint calls DoSnapshot. DoRecovery is called once when FORM starts up with the command line argument -R. Most of the other code contains debugging facilities that are only compiled if the macro PRINTDEBUG is defined.

The recovery mechanism is atomic, i.e. only if everything went well, the final recovery file is created (and the older one overwritten) in a single step (copying). If some errors occur, a warning is issued and the program continues without having created a new recovery file. The only situation in which the creation of the recovery data leads to a termination of the running program is if not enough disk or memory space is left.

For ParFORM each slave creates its own recovery file, sends it to the master and then it deletes the recovery file. The master stores all the recovery files and on recovery it feeds these files to the slaves. It is nearly impossible to recover after some MPI fault so ParFORM terminates on any recovery failure.

DoRecovery and DoSnapshot do the loading and saving of the recovery data, respectively. Every change in one functions needs to be accompanied by the appropriate change in the other function. The structure of both functions is quite similar. They handle the relevant global structs one after the other and then care about the copying of the hide and scratch files.

The names of the recovery, scratch and hide files are hard-coded in the variables in fold "filenames and system commands".

If the global structs AM,AP,AC,AR are changed, DoRecovery and DoSnapshot usually also have to be changed. Some structs are read/written as a whole (AP,AC), some are read/written only partly as a selection of their individual elements (AM,AR). If AM or AR have been changed by adding or removing an element that is important for the runtime status, then the reading/writing statements have to be added to or removed from DoRecovery and Do↩Snapshot. If AP or AC are changed, then for non-pointer variables (in the case of a struct it also means that none of its elements is a pointer) nothing has to be changed in the functions here. If pointers are involved, extra code has to be added (or removed). See the comments of DoRecovery and DoSnapshot.

Definition in file checkpoint.c.

4.5.2 Macro Definition Documentation

4.5.2.1 CACHED_SNAPSHOT

```
#define CACHED_SNAPSHOT
```

Definition at line 1211 of file checkpoint.c.

4.5.2.2 CACHE_SIZE

```
#define CACHE_SIZE 4096
```

Definition at line 1213 of file checkpoint.c.

4.5.2.3 R_FREE

```
#define R_FREE(
            ARG )  if ( ARG ) M_free(ARG, #ARG);
```

Definition at line 1278 of file checkpoint.c.

4.5.2.4 R_FREE_NAMETREE

```
#define R_FREE_NAMETREE(
            ARG )
```

Value:

```
R_FREE(ARG->namenode); \
R_FREE(ARG->namebuffer); \
R_FREE(ARG);
```

Definition at line 1281 of file checkpoint.c.

4.5.2.5 R_FREE_STREAM

```
#define R_FREE_STREAM(
              ARG )
```

Value:

```
R_FREE(ARG.buffer); \
R_FREE(ARG.FoldName); \
R_FREE(ARG.name);
```

Definition at line 1286 of file checkpoint.c.

4.5.2.6 R_SET

```
#define R_SET(
              VAR,
              TYPE )  VAR = *((TYPE*)p); p = (unsigned char*)p + sizeof(TYPE);
```

Definition at line 1293 of file checkpoint.c.

4.5.2.7 R_COPY_B

```
#define R_COPY_B(
              VAR,
              SIZE,
              CAST )
```

Value:

```
VAR = (CAST)Malloc1(SIZE,#VAR); \
memcpy(VAR, p, SIZE); p = (unsigned char*)p + SIZE;
```

Definition at line 1298 of file checkpoint.c.

4.5.2.8 S_WRITE_B

```
#define S_WRITE_B(
              BUF,
              LEN )  if ( fwrite_cached(BUF, 1, LEN, fd) != (size_t)(LEN) ) return(__LINE__);
```

Definition at line 1302 of file checkpoint.c.

4.5.2.9 S_FLUSH_B

```
#define S_FLUSH_B  if ( flush_cache(fd) != 1 ) return(__LINE__);
```

Definition at line 1305 of file checkpoint.c.

4.5.2.10 R_COPY_S

```
#define R_COPY_S(
                VAR,
                CAST )
```

Value:

```
    if ( VAR ) { \
        VAR = (CAST)Malloc1(strlen(p)+1,"R_COPY_S"); \
        strcpy((char*)VAR, p); p = (unsigned char*)p + strlen(p) + 1; \
    }
```

Definition at line 1310 of file checkpoint.c.

4.5.2.11 S_WRITE_S

```
#define S_WRITE_S(
                STR )
```

Value:

```
    if ( STR ) { \
        l = strlen((char*)STR) + 1; \
        if ( fwrite_cached(STR, 1, l, fd) != (size_t)l ) return(__LINE__); \
    }
```

Definition at line 1316 of file checkpoint.c.

4.5.2.12 R_COPY_LIST

```
#define R_COPY_LIST(
                ARG )
```

Value:

```
    if ( ARG.maxnum ) { \
        R_COPY_B(ARG.lijst, ARG.size*ARG.maxnum, void*) \
    }
```

Definition at line 1324 of file checkpoint.c.

4.5.2.13 S_WRITE_LIST

```
#define S_WRITE_LIST(
                LST )
```

Value:

```
    if ( LST.maxnum ) { \
        S_WRITE_B((char*)LST.lijst, LST.maxnum*LST.size) \
    }
```

Definition at line 1329 of file checkpoint.c.

4.5.2.14 R_COPY_NAMETREE

```
#define R_COPY_NAMETREE(
            ARG )
```

Value:

```
R_COPY_B(ARG, sizeof(NAMETREE), NAMETREE*); \
if ( ARG->namenode ) { \
    R_COPY_B(ARG->namenode, ARG->nodesize*sizeof(NAMENODE), NAMENODE*); \
} \
if ( ARG->namebuffer ) { \
    R_COPY_B(ARG->namebuffer, ARG->namesize, UBYTE*); \
}
```

Definition at line 1336 of file checkpoint.c.

4.5.2.15 S_WRITE_NAMETREE

```
#define S_WRITE_NAMETREE(
            ARG )
```

Value:

```
S_WRITE_B(ARG, sizeof(NAMETREE)); \
if ( ARG->namenode ) { \
    S_WRITE_B(ARG->namenode, ARG->nodesize*sizeof(struct NaMeNode)); \
} \
if ( ARG->namebuffer ) { \
    S_WRITE_B(ARG->namebuffer, ARG->namesize); \
}
```

Definition at line 1345 of file checkpoint.c.

4.5.2.16 S_WRITE_DOLLAR

```
#define S_WRITE_DOLLAR(
            ARG )
```

Value:

```
if ( ARG.size && ARG.where && ARG.where != &(AM.dollarzero) ) { \
    S_WRITE_B(ARG.where, ARG.size*sizeof(WORD)) \
}
```

Definition at line 1356 of file checkpoint.c.

4.5.2.17 ANNOUNCE

```
#define ANNOUNCE(
            str )
```

Definition at line 1367 of file checkpoint.c.

4.5.3 Function Documentation

4.5.3.1 CheckRecoveryFile()

```
int CheckRecoveryFile ( )
```

Checks whether a snapshot/recovery file exists. Returns 1 if it exists, 0 otherwise.

Definition at line 278 of file checkpoint.c.

References RecoveryFilename().

Here is the call graph for this function:

4.5.3.2 DeleteRecoveryFile()

```
void DeleteRecoveryFile ( )
```

Deletes the recovery files. It is called by CleanUp() in the case of a successful completion.

Definition at line 333 of file checkpoint.c.

4.5.3.3 RecoveryFilename()

```
char * RecoveryFilename ( )
```

Returns pointer to recovery filename.

Definition at line 364 of file checkpoint.c.

Referenced by CheckRecoveryFile(), and DoRecovery().

4.5.3.4 InitRecovery()

```
void InitRecovery ( )
```

Sets up the strings for the filenames of the recovery files. This functions should only be called once to avoid memory leaks and after AM.TempDir has been initialized.

Definition at line 399 of file checkpoint.c.

4.5.3.5 fwrite_cached()

```
size_t fwrite_cached (
            const void * ptr,
            size_t size,
            size_t nmemb,
            FILE * fd )
```

Definition at line 1219 of file checkpoint.c.

4.5.3.6 flush_cache()

```
size_t flush_cache (
            FILE * fd )
```

Definition at line 1244 of file checkpoint.c.

4.5.3.7 DoRecovery()

```
int DoRecovery (
            int * moduletype )
```

Reads from the recovery file and restores all necessary variables and states in FORM, so that the execution can recommence in preprocessor() as if no restart of FORM had occurred.

The recovery file is read into memory as a whole. The pointer p then points into this memory at the next non-processed data. The macros by which variables are restored, like R_SET, automatically increase p appropriately.

If something goes wrong, the function returns with a non-zero value.

Allocated memory that would be lost when overwriting the global structs with data from the file is freed first. A major part of the code deals with the restoration of pointers. The idiom we use is to memorize the original pointer value (org), allocate new memory and copy the data from the file into this memory, calculate the offset between the old pointer value and the new allocated memory position (ofs), and then correct all affected pointers (+=ofs).

We rely on the fact that several variables (especially in AM) are already assigned the correct values by the startup functions. That means, in principle, that a change in the setup files between snapshot creation and recovery will be noticed.

Definition at line 1399 of file checkpoint.c.

References TaBlEs::argtail, TaBlEs::boomlijst, BrAcKeTiNfO::bracketbuffer, TaBlEs::buffers, TaBlEs::bufferssize, CopyFile(), TaBlEs::flags, ReNuMbEr::func, ReNuMbEr::funnum, VaRrEnUm::hi, BrAcKeTiNfO::indexbuffer, ReNuMbEr::indi, ReNuMbEr::indnum, VaRrEnUm::lo, TaBlEs::MaxTreeSize, TaBlEs::mm, TaBlEs::numind, TaBlEs::pattern, TaBlEs::prototype, TaBlEs::prototypeSize, RecoveryFilename(), ExPrEsSiOn::renumlists, TaBlEs::reserved, TaBlEs::spare, TaBlEs::sparse, VaRrEnUm::start, ReNuMbEr::symb, ReNuMbEr::symnum, TaBlEs::tablepointers, TimeWallClock(), TaBlEs::totind, ReNuMbEr::vecnum, and ReNuMbEr::vect.

Here is the call graph for this function:

4.5.3.8 DoCheckpoint()

```
void DoCheckpoint (
            int moduletype )
```

Checks whether a snapshot should be done. Calls DoSnapshot() to create the snapshot.

Definition at line 3102 of file checkpoint.c.

References PF_BroadcastNumber(), PF_RecvFile(), PF_SendFile(), and TimeWallClock().

Here is the call graph for this function:

4.5.4 Variable Documentation

4.5.4.1 cache_buffer

```
unsigned char cache_buffer[CACHE_SIZE]
```

Definition at line 1216 of file checkpoint.c.

4.5.4.2 cache_fill

```
size_t cache_fill = 0
```

Definition at line 1217 of file checkpoint.c.

4.6 checkpoint.c

Go to the documentation of this file.

```
00001 /*
00002     #[ Explanations :
00003 */
00052 /*
00053     #] Explanations :
00054     #[ License :
00055  *
00056  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00057  *   When using this file you are requested to refer to the publication
00058  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00059  *   This is considered a matter of courtesy as the development was paid
00060  *   for by FOM the Dutch physics granting agency and we would like to
00061  *   be able to track its scientific use to convince FOM of its value
00062  *   for the community.
00063  *
00064  *   This file is part of FORM.
00065  *
00066  *   FORM is free software: you can redistribute it and/or modify it under the
00067  *   terms of the GNU General Public License as published by the Free Software
00068  *   Foundation, either version 3 of the License, or (at your option) any later
00069  *   version.
00070  *
00071  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00072  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00073  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00074  *   details.
00075  *
00076  *   You should have received a copy of the GNU General Public License along
00077  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00078  */
00079 /*
00080     #] License :
00081     #[ Includes :
00082 */
00083
00084 #include "form3.h"
00085
00086 #include <errno.h>
00087
00088 /*
00089 #define PRINTDEBUG
00090 */
00091
00092 /*
00093 #define PRINTTIMEMARKS
00094 */
00095
00096 /*
00097     #] Includes :
00098     #[ filenames and system commands :
00099 */
00100
00104 #ifdef WITHMPI
00105 #define BASENAME_FMT "%c%04dFORMrecv"
00111 static char BaseName[] = BASENAME_FMT;
00112 #else
00113 static char *BaseName = "FORMrecv";
00114 #endif
00118 static char *recoveryfile = 0;
00124 static char *intermedfile = 0;
00128 static char *sortfile = 0;
00132 static char *hidefile = 0;
00136 static char *storefile = 0;
00137
00142 static int done_snapshot = 0;
00143
00144 #ifdef WITHMPI
00149 static int PF_fmt_pos;
00150
```

```
static const char *PF_recoveryfile(char prefix, int id, int intermed)
{
    /*
     * Assume that InitRecovery() has been already called, namely
     * recoveryfile, intermedfile and PF_fmt_pos are already initialized.
     */
    static char *tmp_recovery = NULL;
    static char *tmp_intermed  = NULL;
    char *tmp, c;
    if ( tmp_recovery == NULL ) {
        if ( PF.numtasks > 9999 ) {  /* see BASENAME_FMT */
            MesPrint("Checkpoint: too many number of processors.");
            Terminate(-1);
        }
        tmp_recovery = (char *)Malloc1(strlen(recoveryfile) + strlen(intermedfile) + 2, "PF_recoveryfile");
        tmp_intermed = tmp_recovery + strlen(recoveryfile) + 1;
        strcpy(tmp_recovery, recoveryfile);
        strcpy(tmp_intermed, intermedfile);
    }
    tmp = intermed ? tmp_intermed : tmp_recovery;
    c = tmp[PF_fmt_pos + 13];  /* The magic number 13 comes from BASENAME_FMT. */
    sprintf(tmp + PF_fmt_pos, BASENAME_FMT, prefix, id);
    tmp[PF_fmt_pos + 13] = c;
    return tmp;
}
#endif

/*
    #] filenames and system commands :
    #[ CheckRecoveryFile :
*/

#ifdef WITHMPI

static int PF_CheckRecoveryFile()
{
    int i,ret=0;
    FILE *fd;
    /* Check if the recovery file for the master exists. */
    if ( PF.me == MASTER ) {
        if ( (fd = fopen(recoveryfile, "r")) ) {
            fclose(fd);
            PF_BroadcastNumber(1);
        }
        else {
            PF_BroadcastNumber(0);
            return 0;
        }
    }
    else {
        if ( !PF_BroadcastNumber(0) )
            return 0;
    }
    /* Now the main part. */
    if (PF.me == MASTER){
        /*We have to have recovery files for the master and all the slaves:*/
        for(i=1; i<PF.numtasks;i++){
            const char *tmpnam = PF_recoveryfile('m', i, 0);
            if ( (fd = fopen(tmpnam, "r")) )
                fclose(fd);
            else
                break;
        }/*for(i=0; i<PF.numtasks;i++)*/
        if(i!=PF.numtasks){/*some files are absent*/
            int j;
            /*Send all slaves failure*/
            for(j=1; j<PF.numtasks;j++){
                ret=PF_SendFile(j, NULL);
                if(ret<0)
                    return(-1);
            }
            if(i==0)
                return(0);/*Ok, no recovery files at all.*/
            /*The master recovery file exists but some slave files are absent*/
            MesPrint("The file %s exists but some of the slave recovery files are absent.",
                             RecoveryFilename());
            return(-1);
        }/*if(i!=PF.numtasks)*/
        /*All the files are here.*/
        /*Send all slaves success and files:*/
        for(i=1; i<PF.numtasks;i++){
            const char *tmpnam = PF_recoveryfile('m', i, 0);
            fd = fopen(tmpnam, "r");
            ret=PF_SendFile(i, fd);/*if fd==NULL, PF_SendFile seds to a slave the failure tag*/
            if(fd == NULL)
                return(-1);
```

```
            else
                fclose(fd);
            if(ret<0)
                return(-1);
        }/*for(i=0; i<PF.numtasks;i++)*/
        return(1);
    }/*if (PF.me == MASTER)*/
    /*Slave:*/
    /*Get the answer from the master:*/
    fd=fopen(recoveryfile,"w");
    if(fd == NULL) {
        MesPrint("Failed to open %s in write mode in process %w", recoveryfile);
        return(-1);
    }
    ret=PF_RecvFile(MASTER,fd);
    if(ret<0)
        return(-1);
    fclose(fd);
    if(ret==0){
        /*Nothing is found by the master*/
        remove(recoveryfile);
        return(0);
    }
    /*Recovery file is successfully transferred*/
    return(1);
}
#endif

int CheckRecoveryFile()
{
    int ret = 0;
#ifdef WITHMPI
    ret = PF_CheckRecoveryFile();
#else
    FILE *fd;
    if ( (fd = fopen(recoveryfile, "r")) ) {
        fclose(fd);
        ret = 1;
    }
#endif
    if ( ret < 0 ){/*In ParFORM CheckRecoveryFile() may return a fatal error.*/
        MesPrint("Fail checking recovery file");
        Terminate(-1);
    }
    else if  ( ret > 0 ) {
        if ( AC.CheckpointFlag != -1 ) {
            /* recovery file exists but recovery option is not given */
#ifdef WITHMPI
            if ( PF.me == MASTER ) {
#endif
            MesPrint("The recovery file %s exists, but the recovery option -R has not been given!",
    RecoveryFilename());
            MesPrint("FORM will be terminated to avoid unintentional loss of data.");
            MesPrint("Delete the recovery file manually, if you want to start FORM without
    recovery.");
#ifdef WITHMPI
            }
            if(PF.me != MASTER)
                remove(RecoveryFilename());
#endif
            Terminate(-1);
        }
    }
    else {
        if ( AC.CheckpointFlag == -1 ) {
            /* recovery option given but recovery file does not exist */
#ifdef WITHMPI
            if ( PF.me == MASTER )
#endif
            MesPrint("Option -R for recovery has been given, but the recovery file %s does not
    exist!", RecoveryFilename());
            Terminate(-1);
        }
    }
    return(ret);
}

/*
    #] CheckRecoveryFile :
    #[ DeleteRecoveryFile :
*/

void DeleteRecoveryFile()
{
    if ( done_snapshot ) {
        remove(recoveryfile);
#ifdef WITHMPI
```

```
        if( PF.me == MASTER){
            int i;
            for(i=1; i<PF.numtasks;i++){
                const char *tmpnam = PF_recoveryfile('m', i, 0);
                remove(tmpnam);
            }/*for(i=1; i<PF.numtasks;i++)*/
            remove(storefile);
            remove(sortfile);
            remove(hidefile);
        }/*if( PF.me == MASTER)*/
#else
        remove(storefile);
        remove(sortfile);
        remove(hidefile);
#endif
    }
}

/*
    #] DeleteRecoveryFile :
    #[ RecoveryFilename :
*/

char *RecoveryFilename()
{
    return(recoveryfile);
}

/*
    #] RecoveryFilename :
    #[ InitRecovery :
*/

static char *InitName(char *str, char *ext)
{
    char *s, *d = str;
    if ( AM.TempDir ) {
        s = (char*)AM.TempDir;
        while ( *s ) { *d++ = *s++; }
        *d++ = SEPARATOR;
    }
    s = BaseName;
    while ( *s ) { *d++ = *s++; }
    *d++ = '.';
    s = ext;
    while ( *s ) { *d++ = *s++; }
    *d++ = 0;
    return d;
}

void InitRecovery()
{
    int lenpath = AM.TempDir ? strlen((char*)AM.TempDir)+1 : 0;
#ifdef WITHMPI
    sprintf(BaseName,BASENAME_FMT,(PF.me == MASTER)?'m':'s',PF.me);
    /*Now BaseName has a form ?XXXXFORMrecv where ? == 'm' for master and 's' for slave,
        XXXX is a zero - padded PF.me*/
    PF_fmt_pos = lenpath;
#endif
    recoveryfile = (char*)Malloc1(5*(lenpath+strlen(BaseName)+4+1),"InitRecovery");
    intermedfile = InitName(recoveryfile, "tmp");
    sortfile     = InitName(intermedfile, "XXX");
    hidefile     = InitName(sortfile, "out");
    storefile    = InitName(hidefile, "hid");
                   InitName(storefile, "str");
}

/*
    #] InitRecovery :
    #[ Debugging :
*/

#ifdef PRINTDEBUG

void print_BYTE(void *p)
{
    UBYTE h = (UBYTE)(*((UBYTE*)p) » 4);
    UBYTE l = (UBYTE)(*((UBYTE*)p) & 0x0F);
    if ( h > 9 ) h += 55; else h += 48;
    if ( l > 9 ) l += 55; else l += 48;
    printf("%c%c ", h, l);
}

static void print_STR(UBYTE *p)
{
    if ( p ) {
        MesPrint("%s", (char*)p);
```

```
    }
    else {
        MesPrint("NULL");
    }
}

static void print_WORDB(WORD *buf, WORD *top)
{
    LONG size = top-buf;
    int i;
    while ( size > 0 ) {
        if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
        else i = size;
        size -= i;
        MesPrint("%a",i,buf);
        buf += i;
    }
}

static void print_VOIDP(void *p, size_t size)
{
    int i;
    if ( p ) {
        while ( size > 0 ) {
            if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
            else i = size;
            size -= i;
            MesPrint("%b",i,(UBYTE *)p);
            p = ((UBYTE *)p)+i;
        }
    }
    else {
        MesPrint("NULL");
    }
}

static void print_CHARS(UBYTE *p, size_t size)
{
    int i;
    while ( size > 0 ) {
        if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
        else i = size;
        size -= i;
        MesPrint("%C",i,(char *)p);
        p += i;
    }
}

static void print_WORDV(WORD *p, size_t size)
{
    int i;
    if ( p ) {
        while ( size > 0 ) {
            if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
            else i = size;
            size -= i;
            MesPrint("%a",i,p);
            p += i;
        }
    }
    else {
        MesPrint("NULL");
    }
}

static void print_INTV(int *p, size_t size)
{
    int iarray[8];
    WORD i = 0;
    if ( p ) {
        while ( size > 0 ) {
            if ( i >= 8 ) {
                MesPrint("%I",i,iarray);
                i = 0;
            }
            iarray[i++] = *p++;
            size--;
        }
        if ( i > 0 ) MesPrint("%I",i,iarray);
    }
    else {
        MesPrint("NULL");
    }
}

static void print_LONGV(LONG *p, size_t size)
{
```

```
    LONG larray[8];
    WORD i = 0;
    if ( p ) {
        while ( size > 0 ) {
            if ( i >= 8 ) {
                MesPrint("%I",i,larray);
                i = 0;
            }
            larray[i++] = *p++;
            size--;
        }
        if ( i > 0 ) MesPrint("%I",i,larray);
    }
    else {
        MesPrint("NULL");
    }
}

static void print_PRELOAD(PRELOAD *l)
{
    if ( l->size ) {
        print_CHARS(l->buffer, l->size);
    }
    MesPrint("%ld", l->size);
}

static void print_PREVAR(PREVAR *l)
{
    MesPrint("%s", l->name);
    print_STR(l->value);
    if ( l->nargs ) print_STR(l->argnames);
    MesPrint("%d", l->nargs);
    MesPrint("%d", l->wildarg);
}

static void print_DOLLARS(DOLLARS l)
{
    print_VOIDP(l->where, l->size);
    MesPrint("%ld", l->size);
    MesPrint("%ld", l->name);
    MesPrint("%s", AC.dollarnames->namebuffer+l->name);
    MesPrint("%d", l->type);
    MesPrint("%d", l->node);
    MesPrint("%d", l->index);
    MesPrint("%d", l->zero);
    MesPrint("%d", l->numdummies);
    MesPrint("%d", l->nfactors);
}

static void print_LIST(LIST *l)
{
    print_VOIDP(l->lijst, l->size);
    MesPrint("%s", l->message);
    MesPrint("%d", l->num);
    MesPrint("%d", l->maxnum);
    MesPrint("%d", l->size);
    MesPrint("%d", l->numglobal);
    MesPrint("%d", l->numtemp);
    MesPrint("%d", l->numclear);
}

static void print_DOLOOP(DOLOOP *l)
{
    print_PRELOAD(&(l->p));
    print_STR(l->name);
    if ( l->type != NUMERICALLOOP ) {
        print_STR(l->vars);
    }
    print_STR(l->contents);
    if ( l->type != LISTEDLOOP && l->type != NUMERICALLOOP ) {
        print_STR(l->dollarname);
    }
    MesPrint("%l", l->startlinenumber);
    MesPrint("%l", l->firstnum);
    MesPrint("%l", l->lastnum);
    MesPrint("%l", l->incnum);
    MesPrint("%d", l->type);
    MesPrint("%d", l->NoShowInput);
    MesPrint("%d", l->errorsinloop);
    MesPrint("%d", l->firstloopcall);
}

static void print_PROCEDURE(PROCEDURE *l)
{
    if ( l->loadmode != 1 ) {
        print_PRELOAD(&(l->p));
    }
```

```
    print_STR(l->name);
    MesPrint("%d", l->loadmode);
}

static void print_NAMETREE(NAMETREE *t)
{
    int i;
    for ( i=0; i<t->nodefill; ++i ) {
        MesPrint("%l %d %d %d %d %d %d\n", t->namenode[i].name,
             t->namenode[i].parent, t->namenode[i].left, t->namenode[i].right,
             t->namenode[i].balance, t->namenode[i].type, t->namenode[i].number );
    }
    print_CHARS(t->namebuffer, t->namefill);
    MesPrint("%l", t->namesize);
    MesPrint("%l", t->namefill);
    MesPrint("%l", t->nodesize);
    MesPrint("%l", t->nodefill);
    MesPrint("%l", t->oldnamefill);
    MesPrint("%l", t->oldnodefill);
    MesPrint("%l", t->globalnamefill);
    MesPrint("%l", t->globalnodefill);
    MesPrint("%l", t->clearnamefill);
    MesPrint("%l", t->clearnodefill);
    MesPrint("%d", t->headnode);
}

void print_CBUF(CBUF *c)
{
    int i;
    print_WORDV(c->Buffer, c->BufferSize);
    /*
    MesPrint("%x", c->Buffer);
    MesPrint("%x", c->lhs);
    MesPrint("%x", c->rhs);
    */
    for ( i=0; i<c->numlhs; ++i ) {
        if ( c->lhs[i]) MesPrint("%d", *(c->lhs[i]));
    }
    for ( i=0; i<c->numrhs; ++i ) {
        if ( c->rhs[i]) MesPrint("%d", *(c->rhs[i]));
    }
    MesPrint("%l", *c->CanCommu);
    MesPrint("%l", *c->NumTerms);
    MesPrint("%d", *c->numdum);
    for ( i=0; i<c->MaxTreeSize; ++i ) {
        MesPrint("%d %d %d %d %d", c->boomlijst[i].parent, c->boomlijst[i].left,
c->boomlijst[i].right,
                c->boomlijst[i].value, c->boomlijst[i].blnce);
    }
}

static void print_STREAM(STREAM *t)
{
    print_CHARS(t->buffer, t->inbuffer);
    MesPrint("%l", (LONG)(t->pointer-t->buffer));
    print_STR(t->FoldName);
    print_STR(t->name);
    if ( t->type == PREVARSTREAM || t->type == DOLLARSTREAM ) {
        print_STR(t->pname);
    }
    MesPrint("%l", (LONG)t->fileposition);
    MesPrint("%l", (LONG)t->linenumber);
    MesPrint("%l", (LONG)t->prevline);
    MesPrint("%l", t->buffersize);
    MesPrint("%l", t->bufferposition);
    MesPrint("%l", t->inbuffer);
    MesPrint("%d", t->previous);
    MesPrint("%d", t->handle);
    switch ( t->type ) {
        case FILESTREAM: MesPrint("%d == FILESTREAM", t->type); break;
        case PREVARSTREAM: MesPrint("%d == PREVARSTREAM", t->type); break;
        case PREREADSTREAM: MesPrint("%d == PREREADSTREAM", t->type); break;
        case PIPESTREAM: MesPrint("%d == PIPESTREAM", t->type); break;
        case PRECALCSTREAM: MesPrint("%d == PRECALCSTREAM", t->type); break;
        case DOLLARSTREAM: MesPrint("%d == DOLLARSTREAM", t->type); break;
        case PREREADSTREAM2: MesPrint("%d == PREREADSTREAM2", t->type); break;
        case EXTERNALCHANNELSTREAM: MesPrint("%d == EXTERNALCHANNELSTREAM", t->type); break;
        case PREREADSTREAM3: MesPrint("%d == PREREADSTREAM3", t->type); break;
        default: MesPrint("%d == UNKNOWN", t->type);
    }
}

static void print_M()
{
    MesPrint("%%%% M_const");
    MesPrint("%d", *AM.gcmod);
    MesPrint("%d", *AM.gpowmod);
```

```
    print_STR(AM.TempDir);
    print_STR(AM.TempSortDir);
    print_STR(AM.IncDir);
    print_STR(AM.InputFileName);
    print_STR(AM.LogFileName);
    print_STR(AM.OutBuffer);
    print_STR(AM.Path);
    print_STR(AM.SetupDir);
    print_STR(AM.SetupFile);
    MesPrint("--MARK  1");
    MesPrint("%l", (LONG)BASEPOSITION(AM.zeropos));
#ifdef WITHPTHREADS
    MesPrint("%l", AM.ThreadScratSize);
    MesPrint("%l", AM.ThreadScratOutSize);
#endif
    MesPrint("%l", AM.MaxTer);
    MesPrint("%l", AM.CompressSize);
    MesPrint("%l", AM.ScratSize);
    MesPrint("%l", AM.SizeStoreCache);
    MesPrint("%l", AM.MaxStreamSize);
    MesPrint("%l", AM.SIOsize);
    MesPrint("%l", AM.SLargeSize);
    MesPrint("%l", AM.SSmallEsize);
    MesPrint("%l", AM.SSmallSize);
    MesPrint("--MARK  2");
    MesPrint("%l", AM.STermsInSmall);
    MesPrint("%l", AM.MaxBracketBufferSize);
    MesPrint("%l", AM.hProcessBucketSize);
    MesPrint("%l", AM.gProcessBucketSize);
    MesPrint("%l", AM.shmWinSize);
    MesPrint("%l", AM.OldChildTime);
    MesPrint("%l", AM.OldSecTime);
    MesPrint("%l", AM.OldMilliTime);
    MesPrint("%l", AM.WorkSize);
    MesPrint("%l", AM.gThreadBucketSize);
    MesPrint("--MARK  3");
    MesPrint("%l", AM.ggThreadBucketSize);
    MesPrint("%d", AM.FileOnlyFlag);
    MesPrint("%d", AM.Interact);
    MesPrint("%d", AM.MaxParLevel);
    MesPrint("%d", AM.OutBufSize);
    MesPrint("%d", AM.SMaxFpatches);
    MesPrint("%d", AM.SMaxPatches);
    MesPrint("%d", AM.StdOut);
    MesPrint("%d", AM.ginsidefirst);
    MesPrint("%d", AM.gDefDim);
    MesPrint("%d", AM.gDefDim4);
    MesPrint("--MARK  4");
    MesPrint("%d", AM.NumFixedSets);
    MesPrint("%d", AM.NumFixedFunctions);
    MesPrint("%d", AM.rbufnum);
    MesPrint("%d", AM.dbufnum);
    MesPrint("%d", AM.SkipClears);
    MesPrint("%d", AM.gfunpowers);
    MesPrint("%d", AM.gStatsFlag);
    MesPrint("%d", AM.gNamesFlag);
    MesPrint("%d", AM.gCodesFlag);
    MesPrint("%d", AM.gTokensWriteFlag);
    MesPrint("%d", AM.gSortType);
    MesPrint("%d", AM.gproperorderflag);
    MesPrint("--MARK  5");
    MesPrint("%d", AM.hparallelflag);
    MesPrint("%d", AM.gparallelflag);
    MesPrint("%d", AM.totalnumberofthreads);
    MesPrint("%d", AM.gSizeCommuteInSet);
    MesPrint("%d", AM.gThreadStats);
    MesPrint("%d", AM.ggThreadStats);
    MesPrint("%d", AM.gFinalStats);
    MesPrint("%d", AM.ggFinalStats);
    MesPrint("%d", AM.gThreadsFlag);
    MesPrint("%d", AM.ggThreadsFlag);
    MesPrint("%d", AM.gThreadBalancing);
    MesPrint("%d", AM.ggThreadBalancing);
    MesPrint("%d", AM.gThreadSortFileSynch);
    MesPrint("%d", AM.ggThreadSortFileSynch);
    MesPrint("%d", AM.gProcessStats);
    MesPrint("%d", AM.ggProcessStats);
    MesPrint("%d", AM.gOldParallelStats);
    MesPrint("%d", AM.ggOldParallelStats);
    MesPrint("%d", AM.gWTimeStatsFlag);
    MesPrint("%d", AM.ggWTimeStatsFlag);
    MesPrint("%d", AM.maxFlevels);
    MesPrint("--MARK  6");
    MesPrint("%d", AM.resetTimeOnClear);
    MesPrint("%d", AM.gcNumDollars);
    MesPrint("%d", AM.MultiRun);
    MesPrint("%d", AM.gNoSpacesInNumbers);
```

```
    MesPrint("%d", AM.ggNoSpacesInNumbers);
    MesPrint("%d", AM.MaxTal);
    MesPrint("%d", AM.IndDum);
    MesPrint("%d", AM.DumInd);
    MesPrint("%d", AM.WilInd);
    MesPrint("%d", AM.gncmod);
    MesPrint("%d", AM.gnpowmod);
    MesPrint("%d", AM.gmodmode);
    MesPrint("--MARK  7");
    MesPrint("%d", AM.gUnitTrace);
    MesPrint("%d", AM.gOutputMode);
    MesPrint("%d", AM.gCnumpows);
    MesPrint("%d", AM.gOutputSpaces);
    MesPrint("%d", AM.gOutNumberType);
    MesPrint("%d %d %d %d", AM.gUniTrace[0], AM.gUniTrace[1], AM.gUniTrace[2], AM.gUniTrace[3]);
    MesPrint("%d", AM.MaxWildcards);
    MesPrint("%d", AM.mTraceDum);
    MesPrint("%d", AM.OffsetIndex);
    MesPrint("%d", AM.OffsetVector);
    MesPrint("%d", AM.RepMax);
    MesPrint("%d", AM.LogType);
    MesPrint("%d", AM.ggStatsFlag);
    MesPrint("%d", AM.gLineLength);
    MesPrint("%d", AM.qError);
    MesPrint("--MARK  8");
    MesPrint("%d", AM.FortranCont);
    MesPrint("%d", AM.HoldFlag);
    MesPrint("%d %d %d %d %d", AM.Ordering[0], AM.Ordering[1], AM.Ordering[2], AM.Ordering[3],
AM.Ordering[4]);
    MesPrint("%d %d %d %d %d", AM.Ordering[5], AM.Ordering[6], AM.Ordering[7], AM.Ordering[8],
AM.Ordering[9]);
    MesPrint("%d %d %d %d %d", AM.Ordering[10], AM.Ordering[11], AM.Ordering[12], AM.Ordering[13],
AM.Ordering[14]);
    MesPrint("%d", AM.silent);
    MesPrint("%d", AM.tracebackflag);
    MesPrint("%d", AM.expnum);
    MesPrint("%d", AM.denomnum);
    MesPrint("%d", AM.facnum);
    MesPrint("%d", AM.invfacnum);
    MesPrint("%d", AM.sumnum);
    MesPrint("%d", AM.sumpnum);
    MesPrint("--MARK  9");
    MesPrint("%d", AM.OldOrderFlag);
    MesPrint("%d", AM.termfunnum);
    MesPrint("%d", AM.matchfunnum);
    MesPrint("%d", AM.countfunnum);
    MesPrint("%d", AM.gPolyFun);
    MesPrint("%d", AM.gPolyFunInv);
    MesPrint("%d", AM.gPolyFunType);
    MesPrint("%d", AM.gPolyFunExp);
    MesPrint("%d", AM.gPolyFunVar);
    MesPrint("%d", AM.gPolyFunPow);
    MesPrint("--MARK 10");
    MesPrint("%d", AM.dollarzero);
    MesPrint("%d", AM.atstartup);
    MesPrint("%d", AM.exitflag);
    MesPrint("%d", AM.NumStoreCaches);
    MesPrint("%d", AM.gIndentSpace);
    MesPrint("%d", AM.ggIndentSpace);
    MesPrint("%d", AM.gShortStatsMax);
    MesPrint("%d", AM.ggShortStatsMax);
    MesPrint("%%%% END M_const");
/*  fflush(0); */
}

static void print_P()
{
    int i;
    MesPrint("%%%% P_const");
    print_LIST(&AP.DollarList);
    for ( i=0; i<AP.DollarList.num; ++i ) {
        print_DOLLARS(&(Dollars[i]));
    }
    MesPrint("--MARK  1");
    print_LIST(&AP.PreVarList);
    for ( i=0; i<AP.PreVarList.num; ++i ) {
        print_PREVAR(&(PreVar[i]));
    }
    MesPrint("--MARK  2");
    print_LIST(&AP.LoopList);
    for ( i=0; i<AP.LoopList.num; ++i ) {
        print_DOLOOP(&(DoLoops[i]));
    }
    MesPrint("--MARK  3");
    print_LIST(&AP.ProcList);
    for ( i=0; i<AP.ProcList.num; ++i ) {
        print_PROCEDURE(&(Procedures[i]));
```

```
    }
    MesPrint("--MARK  4");
    for ( i=0; i<=AP.PreSwitchLevel; ++i ) {
        print_STR(AP.PreSwitchStrings[i]);
    }
    MesPrint("%l", AP.preStop-AP.preStart);
    if ( AP.preFill ) MesPrint("%l", AP.preFill-AP.preStart);
    print_CHARS(AP.preStart, AP.pSize);
    MesPrint("%s", AP.procedureExtension);
    MesPrint("%s", AP.cprocedureExtension);
    print_INTV(AP.PreIfStack, AP.MaxPreIfLevel);
    print_INTV(AP.PreSwitchModes, AP.NumPreSwitchStrings+1);
    print_INTV(AP.PreTypes, AP.NumPreTypes+1);
    MesPrint("%d", AP.PreAssignFlag);
    MesPrint("--MARK  5");
    MesPrint("%d", AP.PreContinuation);
    MesPrint("%l", AP.InOutBuf);
    MesPrint("%l", AP.pSize);
    MesPrint("%d", AP.PreproFlag);
    MesPrint("%d", AP.iBufError);
    MesPrint("%d", AP.PreOut);
    MesPrint("%d", AP.PreSwitchLevel);
    MesPrint("%d", AP.NumPreSwitchStrings);
    MesPrint("%d", AP.MaxPreTypes);
    MesPrint("--MARK  6");
    MesPrint("%d", AP.NumPreTypes);
    MesPrint("%d", AP.DelayPrevar);
    MesPrint("%d", AP.AllowDelay);
    MesPrint("%d", AP.lhdollarerror);
    MesPrint("%d", AP.eat);
    MesPrint("%d", AP.gNumPre);
    MesPrint("%d", AP.PreDebug);
    MesPrint("--MARK  7");
    MesPrint("%d", AP.DebugFlag);
    MesPrint("%d", AP.preError);
    MesPrint("%C", 1, &(AP.ComChar));
    MesPrint("%C", 1, &(AP.cComChar));
    MesPrint("%%%% END P_const");
/*  fflush(0); */
}

static void print_C()
{
    int i;
    UBYTE buf[40], *t;
    MesPrint("%%%% C_const");
    for ( i=0; i<32; ++i ) {
        t = buf;
        t = NumCopy((WORD)(AC.separators[i].bit_7),t);
        t = NumCopy((WORD)(AC.separators[i].bit_6),t);
        t = NumCopy((WORD)(AC.separators[i].bit_5),t);
        t = NumCopy((WORD)(AC.separators[i].bit_4),t);
        t = NumCopy((WORD)(AC.separators[i].bit_3),t);
        t = NumCopy((WORD)(AC.separators[i].bit_2),t);
        t = NumCopy((WORD)(AC.separators[i].bit_1),t);
        t = NumCopy((WORD)(AC.separators[i].bit_0),t);
        MesPrint("%s ",buf);
    }
    print_NAMETREE(AC.dollarnames);
    print_NAMETREE(AC.exprnames);
    print_NAMETREE(AC.varnames);
    MesPrint("--MARK  1");
    print_LIST(&AC.ChannelList);
    for ( i=0; i<AC.ChannelList.num; ++i ) {
        MesPrint("%s %d", channels[i].name, channels[i].handle);
    }
    MesPrint("--MARK  2");
    print_LIST(&AC.DubiousList);
    MesPrint("--MARK  3");
    print_LIST(&AC.FunctionList);
    for ( i=0; i<AC.FunctionList.num; ++i ) {
        if ( functions[i].tabl ) {

        }
        MesPrint("%l", functions[i].symminfo);
        MesPrint("%l", functions[i].name);
        MesPrint("%d", functions[i].namesize);
    }
    MesPrint("--MARK  4");
    print_LIST(&AC.ExpressionList);
    print_LIST(&AC.IndexList);
    print_LIST(&AC.SetElementList);
    print_LIST(&AC.SetList);
    MesPrint("--MARK  5");
    print_LIST(&AC.SymbolList);
    print_LIST(&AC.VectorList);
    print_LIST(&AC.PotModDolList);
```

```
    print_LIST(&AC.ModOptDolList);
    print_LIST(&AC.TableBaseList);

/*
    print_LIST(&AC.cbufList);
    for ( i=0; i<AC.cbufList.num; ++i ) {
        MesPrint("cbufList.num == %d", i);
        print_CBUF(cbuf+i);
    }
    MesPrint("%d", AC.cbufnum);
*/
    MesPrint("--MARK  6");

    print_LIST(&AC.AutoSymbolList);
    print_LIST(&AC.AutoIndexList);
    print_LIST(&AC.AutoVectorList);
    print_LIST(&AC.AutoFunctionList);

    print_NAMETREE(AC.autonames);
    MesPrint("--MARK  7");

    print_LIST(AC.Symbols);
    print_LIST(AC.Indices);
    print_LIST(AC.Vectors);
    print_LIST(AC.Functions);
    MesPrint("--MARK  8");

    print_NAMETREE(*AC.activenames);

    MesPrint("--MARK  9");

    MesPrint("%d", AC.AutoDeclareFlag);

    for ( i=0; i<AC.NumStreams; ++i ) {
        MesPrint("Stream %d\n", i);
        print_STREAM(AC.Streams+i);
    }
    print_STREAM(AC.CurrentStream);
    MesPrint("--MARK 10");

    print_LONGV(AC.termstack, AC.maxtermlevel);
    print_LONGV(AC.termsortstack, AC.maxtermlevel);
    print_VOIDP(AC.cmod, AM.MaxTal*4*sizeof(UWORD));
    print_WORDV((WORD *)(AC.cmod), 1);
    print_WORDV((WORD *)(AC.powmod), 1);
    print_WORDV((WORD*)AC.modpowers, 1);
    print_WORDV((WORD*)AC.halfmod, 1);
    MesPrint("--MARK 10-2");
    /*
    print_WORDV(AC.ProtoType, AC.ProtoType[1]);
    print_WORDV(AC.WildC, 1);
    */

    MesPrint("--MARK 11");
    /* IfHeap ... Labels */

    print_CHARS((UBYTE*)AC.tokens, AC.toptokens-AC.tokens);
    MesPrint("%l", AC.endoftokens-AC.tokens);
    print_WORDV(AC.tokenarglevel, AM.MaxParLevel);
    print_WORDV((WORD*)AC.modinverses, ABS(AC.ncmod));
#ifdef WITHPTHREADS
    print_LONGV(AC.inputnumbers, AC.sizepfirstnum+AC.sizepfirstnum*sizeof(WORD)/sizeof(LONG));
    print_WORDV(AC.pfirstnum, 1);
#endif
    MesPrint("--MARK 12");
    print_LONGV(AC.argstack, MAXNEST);
    print_LONGV(AC.insidestack, MAXNEST);
    print_LONGV(AC.inexprstack, MAXNEST);
    MesPrint("%l", AC.iBufferSize);
    MesPrint("%l", AC.TransEname);
    MesPrint("%l", AC.ProcessBucketSize);
    MesPrint("%l", AC.mProcessBucketSize);
    MesPrint("%l", AC.CModule);
    MesPrint("%l", AC.ThreadBucketSize);
    MesPrint("%d", AC.NoShowInput);
    MesPrint("%d", AC.ShortStats);
    MesPrint("%d", AC.compiletype);
    MesPrint("%d", AC.firstconstindex);
    MesPrint("%d", AC.insidefirst);
    MesPrint("%d", AC.minsidefirst);
    MesPrint("%d", AC.wildflag);
    MesPrint("%d", AC.NumLabels);
    MesPrint("%d", AC.MaxLabels);
    MesPrint("--MARK 13");
    MesPrint("%d", AC.lDefDim);
    MesPrint("%d", AC.lDefDim4);
    MesPrint("%d", AC.NumWildcardNames);
```

```
    MesPrint("%d", AC.WildcardBufferSize);
    MesPrint("%d", AC.MaxIf);
    MesPrint("%d", AC.NumStreams);
    MesPrint("%d", AC.MaxNumStreams);
    MesPrint("%d", AC.firstctypemessage);
    MesPrint("%d", AC.tablecheck);
    MesPrint("%d", AC.idoption);
    MesPrint("%d", AC.BottomLevel);
    MesPrint("%d", AC.CompileLevel);
    MesPrint("%d", AC.TokensWriteFlag);
    MesPrint("%d", AC.UnsureDollarMode);
    MesPrint("%d", AC.outsidefun);
    MesPrint("%d", AC.funpowers);
    MesPrint("--MARK 14");
    MesPrint("%d", AC.WarnFlag);
    MesPrint("%d", AC.StatsFlag);
    MesPrint("%d", AC.NamesFlag);
    MesPrint("%d", AC.CodesFlag);
    MesPrint("%d", AC.TokensWriteFlag);
    MesPrint("%d", AC.SetupFlag);
    MesPrint("%d", AC.SortType);
    MesPrint("%d", AC.lSortType);
    MesPrint("%d", AC.ThreadStats);
    MesPrint("%d", AC.FinalStats);
    MesPrint("%d", AC.ThreadsFlag);
    MesPrint("%d", AC.ThreadBalancing);
    MesPrint("%d", AC.ThreadSortFileSynch);
    MesPrint("%d", AC.ProcessStats);
    MesPrint("%d", AC.OldParallelStats);
    MesPrint("%d", AC.WTimeStatsFlag);
    MesPrint("%d", AC.BracketNormalize);
    MesPrint("%d", AC.maxtermlevel);
    MesPrint("%d", AC.dumnumflag);
    MesPrint("--MARK 15");
    MesPrint("%d", AC.bracketindexflag);
    MesPrint("%d", AC.parallelflag);
    MesPrint("%d", AC.mparallelflag);
    MesPrint("%d", AC.properorderflag);
    MesPrint("%d", AC.vetofilling);
    MesPrint("%d", AC.tablefilling);
    MesPrint("%d", AC.vetotablebasefill);
    MesPrint("%d", AC.exprfillwarning);
    MesPrint("%d", AC.lhdollarflag);
    MesPrint("%d", AC.NoCompress);
#ifdef WITHPTHREADS
    MesPrint("%d", AC.numpfirstnum);
    MesPrint("%d", AC.sizepfirstnum);
#endif
    MesPrint("%d", AC.RepLevel);
    MesPrint("%d", AC.arglevel);
    MesPrint("%d", AC.insidelevel);
    MesPrint("%d", AC.inexprlevel);
    MesPrint("%d", AC.termlevel);
    MesPrint("--MARK 16");
    print_WORDV(AC.argsumcheck, MAXNEST);
    print_WORDV(AC.insidesumcheck, MAXNEST);
    print_WORDV(AC.inexprsumcheck, MAXNEST);
    MesPrint("%d", AC.MustTestTable);
    MesPrint("%d", AC.DumNum);
    MesPrint("%d", AC.ncmod);
    MesPrint("%d", AC.npowmod);
    MesPrint("%d", AC.modmode);
    MesPrint("%d", AC.nhalfmod);
    MesPrint("%d", AC.DirtPow);
    MesPrint("%d", AC.lUnitTrace);
    MesPrint("%d", AC.NwildC);
    MesPrint("%d", AC.ComDefer);
    MesPrint("%d", AC.CollectFun);
    MesPrint("%d", AC.AltCollectFun);
    MesPrint("--MARK 17");
    MesPrint("%d", AC.OutputMode);
    MesPrint("%d", AC.Cnumpows);
    MesPrint("%d", AC.OutputSpaces);
    MesPrint("%d", AC.OutNumberType);
    print_WORDV(AC.lUniTrace, 4);
    print_WORDV(AC.RepSumCheck, MAXREPEAT);
    MesPrint("%d", AC.DidClean);
    MesPrint("%d", AC.IfLevel);
    MesPrint("%d", AC.WhileLevel);
    print_WORDV(AC.IfSumCheck, (AC.MaxIf+1));
    MesPrint("%d", AC.LogHandle);
    MesPrint("%d", AC.LineLength);
    MesPrint("%d", AC.StoreHandle);
    MesPrint("%d", AC.HideLevel);
    MesPrint("%d", AC.lPolyFun);
    MesPrint("%d", AC.lPolyFunInv);
    MesPrint("%d", AC.lPolyFunType);
```

```
    MesPrint("%d", AC.lPolyFunExp);
    MesPrint("%d", AC.lPolyFunVar);
    MesPrint("%d", AC.lPolyFunPow);
    MesPrint("%d", AC.SymChangeFlag);
    MesPrint("%d", AC.CollectPercentage);
    MesPrint("%d", AC.ShortStatsMax);
    MesPrint("--MARK 18");

    print_CHARS(AC.Commercial, COMMERCIALSIZE+2);

    MesPrint("%", AC.CheckpointFlag);
    MesPrint("%l", AC.CheckpointStamp);
    print_STR((unsigned char*)AC.CheckpointRunAfter);
    print_STR((unsigned char*)AC.CheckpointRunBefore);
    MesPrint("%l", AC.CheckpointInterval);

    MesPrint("%%%% END C_const");
/*  fflush(0); */
}

static void print_R()
{
    GETIDENTITY
    int i;
    MesPrint("%%%% R_const");
    MesPrint("%l", (LONG)(AR.infile-AR.Fscr));
    MesPrint("%s", AR.infile->name);
    MesPrint("%l", (LONG)(AR.outfile-AR.Fscr));
    MesPrint("%s", AR.outfile->name);
    MesPrint("%l", AR.hidefile-AR.Fscr);
    MesPrint("%s", AR.hidefile->name);
    for ( i=0; i<3; ++i ) {
        MesPrint("FSCR %d", i);
        print_WORDB(AR.Fscr[i].PObuffer, AR.Fscr[i].POfull);
    }
    /* ... */
    MesPrint("%l", AR.OldTime);
    MesPrint("%l", AR.InInBuf);
    MesPrint("%l", AR.InHiBuf);
    MesPrint("%l", AR.pWorkSize);
    MesPrint("%l", AR.lWorkSize);
    MesPrint("%l", AR.posWorkSize);
    MesPrint("%d", AR.NoCompress);
    MesPrint("%d", AR.gzipCompress);
    MesPrint("%d", AR.Cnumlhs);
#ifdef WITHPTHREADS
    MesPrint("%d", AR.exprtodo);
#endif
    MesPrint("%d", AR.GetFile);
    MesPrint("%d", AR.KeptInHold);
    MesPrint("%d", AR.BracketOn);
    MesPrint("%d", AR.MaxBracket);
    MesPrint("%d", AR.CurDum);
    MesPrint("%d", AR.DeferFlag);
    MesPrint("%d", AR.TePos);
    MesPrint("%d", AR.sLevel);
    MesPrint("%d", AR.Stage4Name);
    MesPrint("%d", AR.GetOneFile);
    MesPrint("%d", AR.PolyFun);
    MesPrint("%d", AR.PolyFunInv);
    MesPrint("%d", AR.PolyFunType);
    MesPrint("%d", AR.PolyFunExp);
    MesPrint("%d", AR.PolyFunVar);
    MesPrint("%d", AR.PolyFunPow);
    MesPrint("%d", AR.Eside);
    MesPrint("%d", AR.MaxDum);
    MesPrint("%d", AR.level);
    MesPrint("%d", AR.expchanged);
    MesPrint("%d", AR.expflags);
    MesPrint("%d", AR.CurExpr);
    MesPrint("%d", AR.SortType);
    MesPrint("%d", AR.ShortSortCount);
    MesPrint("%%%% END R_const");
/*  fflush(0); */
}

#endif /* ifdef PRINTDEBUG */

/*
    #] Debugging :
    #[ Cached file operation functions :
*/

#define CACHED_SNAPSHOT

#define CACHE_SIZE 4096

```

```
#ifdef CACHED_SNAPSHOT
unsigned char cache_buffer[CACHE_SIZE];
size_t cache_fill = 0;

size_t fwrite_cached(const void *ptr, size_t size, size_t nmemb, FILE *fd)
{
    size_t fullsize = size*nmemb;
    if ( fullsize+cache_fill >= CACHE_SIZE ) {
        size_t overlap = CACHE_SIZE-cache_fill;
        memcpy(cache_buffer+cache_fill, (unsigned char*)ptr, overlap);
        if ( fwrite(cache_buffer, 1, CACHE_SIZE, fd) != CACHE_SIZE ) return 0;
        fullsize -= overlap;
        if ( fullsize >= CACHE_SIZE ) {
            cache_fill = fullsize % CACHE_SIZE;
            if ( cache_fill ) memcpy(cache_buffer, (unsigned char*)ptr+overlap+fullsize-cache_fill, cache_fill);
            if ( fwrite((unsigned char*)ptr+overlap, 1, fullsize-cache_fill, fd) != fullsize-cache_fill ) return 0;
        }
        else {
            memcpy(cache_buffer, (unsigned char*)ptr+overlap, fullsize);
            cache_fill = fullsize;
        }
    }
    else {
        memcpy(cache_buffer+cache_fill, (unsigned char*)ptr, fullsize);
        cache_fill += fullsize;
    }
    return nmemb;
}

size_t flush_cache(FILE *fd)
{
    if ( cache_fill ) {
        size_t retval = fwrite(cache_buffer, 1, cache_fill, fd);
        if ( retval != cache_fill ) {
            cache_fill = 0;
            return 0;
        }
        cache_fill = 0;
    }
    return 1;
}
#else
size_t fwrite_cached(const void *ptr, size_t size, size_t nmemb, FILE *fd)
{
    return fwrite(ptr, size, nmemb, fd);
}

size_t flush_cache(FILE *fd)
{
    DUMMYUSE(fd)
    return 1;
}
#endif

/*
    #] Cached file operation functions :
    #[ Helper Macros :
*/

/* some helper macros to streamline the code in DoSnapshot() and DoRecovery() */

/* freeing memory */

#define R_FREE(ARG) \
    if ( ARG ) M_free(ARG, #ARG);

#define R_FREE_NAMETREE(ARG) \
    R_FREE(ARG->namenode); \
    R_FREE(ARG->namebuffer); \
    R_FREE(ARG);

#define R_FREE_STREAM(ARG) \
    R_FREE(ARG.buffer); \
    R_FREE(ARG.FoldName); \
    R_FREE(ARG.name);

/* reading a single variable */

#define R_SET(VAR,TYPE) \
    VAR = *((TYPE*)p); p = (unsigned char*)p + sizeof(TYPE);

/* general buffer */

#define R_COPY_B(VAR,SIZE,CAST) \
    VAR = (CAST)Malloc1(SIZE,#VAR); \
```

```
    memcpy(VAR, p, SIZE); p = (unsigned char*)p + SIZE;

#define S_WRITE_B(BUF,LEN) \
    if ( fwrite_cached(BUF, 1, LEN, fd) != (size_t)(LEN) ) return(__LINE__);

#define S_FLUSH_B \
    if ( flush_cache(fd) != 1 ) return(__LINE__);

/* character strings */

#define R_COPY_S(VAR,CAST) \
    if ( VAR ) { \
        VAR = (CAST)Malloc1(strlen(p)+1,"R_COPY_S"); \
        strcpy((char*)VAR, p); p = (unsigned char*)p + strlen(p) + 1; \
    }

#define S_WRITE_S(STR) \
    if ( STR ) { \
        l = strlen((char*)STR) + 1; \
        if ( fwrite_cached(STR, 1, l, fd) != (size_t)l ) return(__LINE__); \
    }

/* LIST */

#define R_COPY_LIST(ARG) \
    if ( ARG.maxnum ) { \
        R_COPY_B(ARG.lijst, ARG.size*ARG.maxnum, void*) \
    }

#define S_WRITE_LIST(LST) \
    if ( LST.maxnum ) { \
        S_WRITE_B((char*)LST.lijst, LST.maxnum*LST.size) \
    }

/* NAMETREE */

#define R_COPY_NAMETREE(ARG) \
    R_COPY_B(ARG, sizeof(NAMETREE), NAMETREE*); \
    if ( ARG->namenode ) { \
        R_COPY_B(ARG->namenode, ARG->nodesize*sizeof(NAMENODE), NAMENODE*); \
    } \
    if ( ARG->namebuffer ) { \
        R_COPY_B(ARG->namebuffer, ARG->namesize, UBYTE*); \
    }

#define S_WRITE_NAMETREE(ARG) \
    S_WRITE_B(ARG, sizeof(NAMETREE)); \
    if ( ARG->namenode ) { \
        S_WRITE_B(ARG->namenode, ARG->nodesize*sizeof(struct NaMeNode)); \
    } \
    if ( ARG->namebuffer ) { \
        S_WRITE_B(ARG->namebuffer, ARG->namesize); \
    }

/* DOLLAR */

#define S_WRITE_DOLLAR(ARG) \
    if ( ARG.size && ARG.where && ARG.where != &(AM.dollarzero) ) { \
        S_WRITE_B(ARG.where, ARG.size*sizeof(WORD)) \
    }

/* Printing time marks with ANNOUNCE macro */

#ifdef PRINTTIMEMARKS
time_t announce_time;
#define ANNOUNCE(str) time(&announce_time); MesPrint("TIMEMARK %s  %s", ctime(&announce_time), #str);
#else
#define ANNOUNCE(str)
#endif

/*
    #] Helper Macros :
    #[ DoRecovery :
*/

int DoRecovery(int *moduletype)
{
    GETIDENTITY
    FILE *fd;
    POSITION pos;
    void *buf, *p;
    LONG size, l;
    int i, j;
    UBYTE *org;
    char *namebufout, *namebufhide;
    LONG ofs;
    void *oldAMdollarzero;
```

```
    LIST PotModDolListBackup;
    LIST ModOptDolListBackup;
    WORD oldLogHandle;

    MesPrint("Recovering ... %"); fflush(0);

    if ( !(fd = fopen(recoveryfile, "r")) ) return(__LINE__);

    /* load the complete recovery file into a buffer */
    if ( fread(&pos, sizeof(POSITION), 1, fd) != 1 ) return(__LINE__);
    size = BASEPOSITION(pos) - sizeof(POSITION);
    buf = Malloc1(size, "recovery buffer");
    if ( fread(buf, size, 1, fd) != 1 ) return(__LINE__);

    /* pointer p will go through the buffer in the following */
    p = buf;

    /* read moduletype */
    R_SET(*moduletype, int);

    /*#[ AM : */

    /* only certain elements will be restored. the rest of AM should have gotten
     * the correct values at startup. */

    R_SET(AM.hparallelflag, int);
    R_SET(AM.gparallelflag, int);
    R_SET(AM.gCodesFlag, int);
    R_SET(AM.gNamesFlag, int);
    R_SET(AM.gStatsFlag, int);
    R_SET(AM.gTokensWriteFlag, int);
    R_SET(AM.gNoSpacesInNumbers, int);
    R_SET(AM.gIndentSpace, WORD);
    R_SET(AM.gUnitTrace, WORD);
    R_SET(AM.gDefDim, int);
    R_SET(AM.gDefDim4, int);
    R_SET(AM.gncmod, WORD);
    R_SET(AM.gnpowmod, WORD);
    R_SET(AM.gmodmode, WORD);
    R_SET(AM.gOutputMode, WORD);
    R_SET(AM.gCnumpows, WORD);
    R_SET(AM.gOutputSpaces, WORD);
    R_SET(AM.gOutNumberType, WORD);
    R_SET(AM.gfunpowers, int);
    R_SET(AM.gPolyFun, WORD);
    R_SET(AM.gPolyFunInv, WORD);
    R_SET(AM.gPolyFunType, WORD);
    R_SET(AM.gPolyFunExp, WORD);
    R_SET(AM.gPolyFunVar, WORD);
    R_SET(AM.gPolyFunPow, WORD);
    R_SET(AM.gProcessBucketSize, LONG);
    R_SET(AM.OldChildTime, LONG);
    R_SET(AM.OldSecTime, LONG);
    R_SET(AM.OldMilliTime, LONG);
    R_SET(AM.gproperorderflag, int);
    R_SET(AM.gThreadBucketSize, LONG);
    R_SET(AM.gSizeCommuteInSet, int);
    R_SET(AM.gThreadStats, int);
    R_SET(AM.gFinalStats, int);
    R_SET(AM.gThreadsFlag, int);
    R_SET(AM.gThreadBalancing, int);
    R_SET(AM.gThreadSortFileSynch, int);
    R_SET(AM.gProcessStats, int);
    R_SET(AM.gOldParallelStats, int);
    R_SET(AM.gSortType, int);
    R_SET(AM.gShortStatsMax, WORD);
    R_SET(AM.gIsFortran90, int);
    R_SET(oldAMdollarzero, void*);
    R_FREE(AM.gFortran90Kind);
    R_SET(AM.gFortran90Kind,UBYTE *);
    R_COPY_S(AM.gFortran90Kind,UBYTE *);

    R_COPY_S(AM.gextrasym,UBYTE *);
    R_COPY_S(AM.ggextrasym,UBYTE *);

    R_SET(AM.PrintTotalSize,int);
    R_SET(AM.fbuffersize,int);
    R_SET(AM.gOldFactArgFlag,int);
    R_SET(AM.ggOldFactArgFlag,int);

    R_SET(AM.gnumextrasym,int);
    R_SET(AM.ggnumextrasym,int);
    R_SET(AM.NumSpectatorFiles,int);
    R_SET(AM.SizeForSpectatorFiles,int);
    R_SET(AM.gOldGCDflag,int);
    R_SET(AM.ggOldGCDflag,int);
    R_SET(AM.gWTimeStatsFlag, int);
```

```c

    R_FREE(AM.Path);
    R_SET(AM.Path,UBYTE *);
    R_COPY_S(AM.Path,UBYTE *);

#ifdef PRINTDEBUG
    print_M();
#endif

    /*#] AM : */
    /*#[ AC : */

    /* #[ AC free pointers */

    /* AC will be overwritten by data from the recovery file, therefore
     * dynamically allocated memory must be freed first. */

    R_FREE_NAMETREE(AC.dollarnames);
    R_FREE_NAMETREE(AC.exprnames);
    R_FREE_NAMETREE(AC.varnames);
    for ( i=0; i<AC.ChannelList.num; ++i ) {
        R_FREE(channels[i].name);
    }
    R_FREE(AC.ChannelList.lijst);
    R_FREE(AC.DubiousList.lijst);
    for ( i=0; i<AC.FunctionList.num; ++i ) {
        TABLES T = functions[i].tabl;
        if ( T ) {
            R_FREE(T->buffers);
            R_FREE(T->mm);
            R_FREE(T->flags);
            R_FREE(T->prototype);
            R_FREE(T->tablepointers);
            if ( T->sparse ) {
                R_FREE(T->boomlijst);
                R_FREE(T->argtail);
            }
            if ( T->spare ) {
                R_FREE(T->spare->buffers);
                R_FREE(T->spare->mm);
                R_FREE(T->spare->flags);
                R_FREE(T->spare->tablepointers);
                if ( T->spare->sparse ) {
                    R_FREE(T->spare->boomlijst);
                }
                R_FREE(T->spare);
            }
            R_FREE(T);
        }
    }
    R_FREE(AC.FunctionList.lijst);
    for ( i=0; i<AC.ExpressionList.num; ++i ) {
        if ( Expressions[i].renum ) {
            R_FREE(Expressions[i].renum->symb.lo);
            R_FREE(Expressions[i].renum);
        }
        if ( Expressions[i].bracketinfo ) {
            R_FREE(Expressions[i].bracketinfo->indexbuffer);
            R_FREE(Expressions[i].bracketinfo->bracketbuffer);
            R_FREE(Expressions[i].bracketinfo);
        }
        if ( Expressions[i].newbracketinfo ) {
            R_FREE(Expressions[i].newbracketinfo->indexbuffer);
            R_FREE(Expressions[i].newbracketinfo->bracketbuffer);
            R_FREE(Expressions[i].newbracketinfo);
        }
        if ( Expressions[i].renumlists != AN.dummyrenumlist ) {
            R_FREE(Expressions[i].renumlists);
        }
        R_FREE(Expressions[i].inmem);
    }
    R_FREE(AC.ExpressionList.lijst);
    R_FREE(AC.IndexList.lijst);
    R_FREE(AC.SetElementList.lijst);
    R_FREE(AC.SetList.lijst);
    R_FREE(AC.SymbolList.lijst);
    R_FREE(AC.VectorList.lijst);
    for ( i=0; i<AC.TableBaseList.num; ++i ) {
        R_FREE(tablebases[i].iblocks);
        R_FREE(tablebases[i].nblocks);
        R_FREE(tablebases[i].name);
        R_FREE(tablebases[i].fullname);
        R_FREE(tablebases[i].tablenames);
    }
    R_FREE(AC.TableBaseList.lijst);
    for ( i=0; i<AC.cbufList.num; ++i ) {
        R_FREE(cbuf[i].Buffer);
```

```
        R_FREE(cbuf[i].lhs);
        R_FREE(cbuf[i].rhs);
        R_FREE(cbuf[i].boomlijst);
    }
    R_FREE(AC.cbufList.lijst);
    R_FREE(AC.AutoSymbolList.lijst);
    R_FREE(AC.AutoIndexList.lijst);
    R_FREE(AC.AutoVectorList.lijst);
    /* Tables cannot be auto-declared, therefore no extra code here */
    R_FREE(AC.AutoFunctionList.lijst);
    R_FREE_NAMETREE(AC.autonames);
    for ( i=0; i<AC.NumStreams; ++i ) {
        R_FREE_STREAM(AC.Streams[i]);
    }
    R_FREE(AC.Streams);
    R_FREE(AC.termstack);
    R_FREE(AC.termsortstack);
    R_FREE(AC.cmod);
    R_FREE(AC.modpowers);
    R_FREE(AC.halfmod);
    R_FREE(AC.IfHeap);
    R_FREE(AC.IfCount);
    R_FREE(AC.iBuffer);
    for ( i=0; i<AC.NumLabels; ++i ) {
        R_FREE(AC.LabelNames[i]);
    }
    R_FREE(AC.LabelNames);
    R_FREE(AC.FixIndices);
    R_FREE(AC.termsumcheck);
    R_FREE(AC.WildcardNames);
    R_FREE(AC.tokens);
    R_FREE(AC.tokenarglevel);
    R_FREE(AC.modinverses);
    R_FREE(AC.Fortran90Kind);
#ifdef WITHPTHREADS
    R_FREE(AC.inputnumbers);
#endif
    R_FREE(AC.IfSumCheck);
    R_FREE(AC.CommuteInSet);
    R_FREE(AC.CheckpointRunAfter);
    R_FREE(AC.CheckpointRunBefore);

    /* #] AC free pointers */

    /* backup some lists in order to restore it to the initial setup */
    PotModDolListBackup = AC.PotModDolList;
    ModOptDolListBackup = AC.ModOptDolList;
    oldLogHandle = AC.LogHandle;

    /* first we copy AC as a whole and then restore the pointer structures step
       by step. */

    AC = *((struct C_const*)p); p = (unsigned char*)p + sizeof(struct C_const);

    R_COPY_NAMETREE(AC.dollarnames);
    R_COPY_NAMETREE(AC.exprnames);
    R_COPY_NAMETREE(AC.varnames);

    R_COPY_LIST(AC.ChannelList);
    for ( i=0; i<AC.ChannelList.num; ++i ) {
        R_COPY_S(channels[i].name,char*);
        channels[i].handle = ReOpenFile(channels[i].name);
    }
    AC.ChannelList.message = "channel buffer";

    AC.DubiousList.lijst = 0;
    AC.DubiousList.message = "ambiguous variable";
    AC.DubiousList.num =
    AC.DubiousList.maxnum =
    AC.DubiousList.numglobal =
    AC.DubiousList.numtemp =
    AC.DubiousList.numclear = 0;

    R_COPY_LIST(AC.FunctionList);
    for ( i=0; i<AC.FunctionList.num; ++i ) {
        if ( functions[i].tabl ) {
            TABLES tabl;
            R_COPY_B(tabl, sizeof(struct TaBlEs), TABLES);
            functions[i].tabl = tabl;
            if ( tabl->tablepointers ) {
                if ( tabl->sparse ) {
                    R_COPY_B(tabl->tablepointers,
                        tabl->reserved*sizeof(WORD)*(tabl->numind+TABLEEXTENSION),
                        WORD*);
                }
                else {
                    R_COPY_B(tabl->tablepointers,
```

```
                        TABLEEXTENSION*sizeof(WORD)*(tabl->totind), WORD*);
                    }
            }
            org = (UBYTE*)tabl->prototype;
#ifdef WITHPTHREADS
            R_COPY_B(tabl->prototype, tabl->prototypeSize, WORD**);
            ofs = (UBYTE*)tabl->prototype - org;
            for ( j=0; j<AM.totalnumberofthreads; ++j ) {
                if ( tabl->prototype[j] ) {
                    tabl->prototype[j] = (WORD*)((UBYTE*)tabl->prototype[j] + ofs);
                }
            }
            if ( tabl->pattern ) {
                tabl->pattern = (WORD**)((UBYTE*)tabl->pattern + ofs);
                for ( j=0; j<AM.totalnumberofthreads; ++j ) {
                    if ( tabl->pattern[j] ) {
                        tabl->pattern[j] = (WORD*)((UBYTE*)tabl->pattern[j] + ofs);
                    }
                }
            }
#else
            ofs = tabl->pattern - tabl->prototype;
            R_COPY_B(tabl->prototype, tabl->prototypeSize, WORD*);
            if ( tabl->pattern ) {
                tabl->pattern = tabl->prototype + ofs;
            }
#endif
            R_COPY_B(tabl->mm, tabl->numind*(LONG)sizeof(MINMAX), MINMAX*);
            R_COPY_B(tabl->flags, tabl->numind*(LONG)sizeof(WORD), WORD*);
            if ( tabl->sparse ) {
                R_COPY_B(tabl->boomlijst, tabl->MaxTreeSize*(LONG)sizeof(COMPTREE), COMPTREE*);
                R_COPY_S(tabl->argtail,UBYTE*);
            }
            R_COPY_B(tabl->buffers, tabl->bufferssize*(LONG)sizeof(WORD), WORD*);
            if ( tabl->spare ) {
                TABLES spare;
                R_COPY_B(spare, sizeof(struct TaBlEs), TABLES);
                tabl->spare = spare;
                if ( spare->tablepointers ) {
                    if ( spare->sparse ) {
                        R_COPY_B(spare->tablepointers,
                            spare->reserved*sizeof(WORD)*(spare->numind+TABLEEXTENSION),
                            WORD*);
                    }
                    else {
                        R_COPY_B(spare->tablepointers,
                            TABLEEXTENSION*sizeof(WORD)*(spare->totind), WORD*);
                    }
                }
                spare->prototype = tabl->prototype;
                spare->pattern = tabl->pattern;
                R_COPY_B(spare->mm, spare->numind*(LONG)sizeof(MINMAX), MINMAX*);
                R_COPY_B(spare->flags, spare->numind*(LONG)sizeof(WORD), WORD*);
                if ( tabl->sparse ) {
                    R_COPY_B(spare->boomlijst, spare->MaxTreeSize*(LONG)sizeof(COMPTREE), COMPTREE*);
                    spare->argtail = tabl->argtail;
                }
                spare->spare = tabl;
                R_COPY_B(spare->buffers, spare->bufferssize*(LONG)sizeof(WORD), WORD*);
            }
        }
    }
    AC.FunctionList.message = "function";

    R_COPY_LIST(AC.ExpressionList);
    for ( i=0; i<AC.ExpressionList.num; ++i ) {
        EXPRESSIONS ex = Expressions + i;
        if ( ex->renum ) {
            R_COPY_B(ex->renum, sizeof(struct ReNuMbEr), RENUMBER);
            org = (UBYTE*)ex->renum->symb.lo;
            R_SET(size, size_t);
            R_COPY_B(ex->renum->symb.lo, size, WORD*);
            ofs = (UBYTE*)ex->renum->symb.lo - org;
            ex->renum->symb.start = (WORD*)((UBYTE*)ex->renum->symb.start + ofs);
            ex->renum->symb.hi = (WORD*)((UBYTE*)ex->renum->symb.hi + ofs);
            ex->renum->indi.lo = (WORD*)((UBYTE*)ex->renum->indi.lo + ofs);
            ex->renum->indi.start = (WORD*)((UBYTE*)ex->renum->indi.start + ofs);
            ex->renum->indi.hi = (WORD*)((UBYTE*)ex->renum->indi.hi + ofs);
            ex->renum->vect.lo = (WORD*)((UBYTE*)ex->renum->vect.lo + ofs);
            ex->renum->vect.start = (WORD*)((UBYTE*)ex->renum->vect.start + ofs);
            ex->renum->vect.hi = (WORD*)((UBYTE*)ex->renum->vect.hi + ofs);
            ex->renum->func.lo = (WORD*)((UBYTE*)ex->renum->func.lo + ofs);
            ex->renum->func.start = (WORD*)((UBYTE*)ex->renum->func.start + ofs);
            ex->renum->func.hi = (WORD*)((UBYTE*)ex->renum->func.hi + ofs);
            ex->renum->symnum = (WORD*)((UBYTE*)ex->renum->symnum + ofs);
            ex->renum->indnum = (WORD*)((UBYTE*)ex->renum->indnum + ofs);
            ex->renum->vecnum = (WORD*)((UBYTE*)ex->renum->vecnum + ofs);
```

```
            ex->renum->funnum = (WORD*)((UBYTE*)ex->renum->funnum + ofs);
        }
        if ( ex->bracketinfo ) {
            R_COPY_B(ex->bracketinfo, sizeof(BRACKETINFO), BRACKETINFO*);
            R_COPY_B(ex->bracketinfo->indexbuffer, ex->bracketinfo->indexbuffersize*sizeof(BRACKETINDEX), BRACKETINDEX*);
            R_COPY_B(ex->bracketinfo->bracketbuffer, ex->bracketinfo->bracketbuffersize*sizeof(WORD), WORD*);
        }
        if ( ex->newbracketinfo ) {
            R_COPY_B(ex->newbracketinfo, sizeof(BRACKETINFO), BRACKETINFO*);
            R_COPY_B(ex->newbracketinfo->indexbuffer, ex->newbracketinfo->indexbuffersize*sizeof(BRACKETINDEX), BRACKETINDEX*);
            R_COPY_B(ex->newbracketinfo->bracketbuffer, ex->newbracketinfo->bracketbuffersize*sizeof(WORD), WORD*);
        }
#ifdef WITHPTHREADS
        ex->renumlists = 0;
#else
        ex->renumlists = AN.dummyrenumlist;
#endif
        if ( ex->inmem ) {
            R_SET(size, size_t);
            R_COPY_B(ex->inmem, size, WORD*);
        }
    }
    AC.ExpressionList.message = "expression";

    R_COPY_LIST(AC.IndexList);
    AC.IndexList.message = "index";
    R_COPY_LIST(AC.SetElementList);
    AC.SetElementList.message = "set element";
    R_COPY_LIST(AC.SetList);
    AC.SetList.message = "set";
    R_COPY_LIST(AC.SymbolList);
    AC.SymbolList.message = "symbol";
    R_COPY_LIST(AC.VectorList);
    AC.VectorList.message = "vector";

    AC.PotModDolList = PotModDolListBackup;
    AC.ModOptDolList = ModOptDolListBackup;

    R_COPY_LIST(AC.TableBaseList);
    for ( i=0; i<AC.TableBaseList.num; ++i ) {
        if ( tablebases[i].iblocks ) {
            R_COPY_B(tablebases[i].iblocks, tablebases[i].info.numberofindexblocks*sizeof(INDEXBLOCK*),INDEXBLOCK**);
            for ( j=0; j<tablebases[i].info.numberofindexblocks; ++j ) {
                if ( tablebases[i].iblocks[j] ) {
                    R_COPY_B(tablebases[i].iblocks[j], sizeof(INDEXBLOCK), INDEXBLOCK*);
                }
            }
        }
        if ( tablebases[i].nblocks ) {
            R_COPY_B(tablebases[i].nblocks, tablebases[i].info.numberofnamesblocks*sizeof(NAMESBLOCK*),NAMESBLOCK**);
            for ( j=0; j<tablebases[i].info.numberofindexblocks; ++j ) {
                if ( tablebases[i].nblocks[j] ) {
                    R_COPY_B(tablebases[i].nblocks[j], sizeof(NAMESBLOCK), NAMESBLOCK*);
                }
            }
        }
        /* reopen file */
        if ( ( tablebases[i].handle = fopen(tablebases[i].fullname, "r+b") ) == NULL ) {
            MesPrint("ERROR: Could not reopen tablebase %s!",tablebases[i].name);
            Terminate(-1);
        }
        R_COPY_S(tablebases[i].name,char*);
        R_COPY_S(tablebases[i].fullname,char*);
        R_COPY_S(tablebases[i].tablenames,char*);
    }
    AC.TableBaseList.message = "list of tablebases";

    R_COPY_LIST(AC.cbufList);
    for ( i=0; i<AC.cbufList.num; ++i ) {
        org = (UBYTE*)cbuf[i].Buffer;
        R_COPY_B(cbuf[i].Buffer, cbuf[i].BufferSize*sizeof(WORD), WORD*);
        ofs = (UBYTE*)cbuf[i].Buffer - org;
        cbuf[i].Top = (WORD*)((UBYTE*)cbuf[i].Top + ofs);
        cbuf[i].Pointer = (WORD*)((UBYTE*)cbuf[i].Pointer + ofs);
        R_COPY_B(cbuf[i].lhs, cbuf[i].maxlhs*(LONG)sizeof(WORD*), WORD**);
        for ( j=1; j<=cbuf[i].numlhs; ++j ) {
            if ( cbuf[i].lhs[j] ) cbuf[i].lhs[j] = (WORD*)((UBYTE*)cbuf[i].lhs[j] + ofs);
        }
        org = (UBYTE*)cbuf[i].rhs;
        R_COPY_B(cbuf[i].rhs, cbuf[i].maxrhs*(LONG)(sizeof(WORD*)+2*sizeof(LONG)+2*sizeof(WORD)), WORD**);
```

```
        for ( j=1; j<=cbuf[i].numrhs; ++j ) {
            if ( cbuf[i].rhs[j] ) cbuf[i].rhs[j] = (WORD*)((UBYTE*)cbuf[i].rhs[j] + ofs);
        }
        ofs = (UBYTE*)cbuf[i].rhs - org;
        cbuf[i].CanCommu = (LONG*)((UBYTE*)cbuf[i].CanCommu + ofs);
        cbuf[i].NumTerms = (LONG*)((UBYTE*)cbuf[i].NumTerms + ofs);
        cbuf[i].numdum = (WORD*)((UBYTE*)cbuf[i].numdum + ofs);
        cbuf[i].dimension = (WORD*)((UBYTE*)cbuf[i].dimension + ofs);
        if ( cbuf[i].boomlijst ) {
            R_COPY_B(cbuf[i].boomlijst, cbuf[i].MaxTreeSize*sizeof(COMPTREE), COMPTREE*);
        }
    }
    AC.cbufList.message = "compiler buffer";

    R_COPY_LIST(AC.AutoSymbolList);
    AC.AutoSymbolList.message = "autosymbol";
    R_COPY_LIST(AC.AutoIndexList);
    AC.AutoIndexList.message = "autoindex";
    R_COPY_LIST(AC.AutoVectorList);
    AC.AutoVectorList.message = "autovector";
    R_COPY_LIST(AC.AutoFunctionList);
    AC.AutoFunctionList.message = "autofunction";

    R_COPY_NAMETREE(AC.autonames);

    AC.Symbols = &(AC.SymbolList);
    AC.Indices = &(AC.IndexList);
    AC.Vectors = &(AC.VectorList);
    AC.Functions = &(AC.FunctionList);
    AC.activenames = &(AC.varnames);

    org = (UBYTE*)AC.Streams;
    R_COPY_B(AC.Streams, AC.MaxNumStreams*(LONG)sizeof(STREAM), STREAM*);
    for ( i=0; i<AC.NumStreams; ++i ) {
        if ( AC.Streams[i].type != FILESTREAM ) {
            UBYTE *org2;
            org2 = AC.Streams[i].buffer;
            if ( AC.Streams[i].inbuffer ) {
                R_COPY_B(AC.Streams[i].buffer, AC.Streams[i].inbuffer, UBYTE*);
            }
            ofs = AC.Streams[i].buffer - org2;
            AC.Streams[i].pointer += ofs;
            AC.Streams[i].top += ofs;
        }
        else {
            p = (unsigned char*)p + AC.Streams[i].inbuffer;
        }
        AC.Streams[i].buffersize = AC.Streams[i].inbuffer;
        R_COPY_S(AC.Streams[i].FoldName,UBYTE*);
        R_COPY_S(AC.Streams[i].name,UBYTE*);
        if ( AC.Streams[i].type == PREVARSTREAM || AC.Streams[i].type == DOLLARSTREAM ) {
            AC.Streams[i].pname = AC.Streams[i].name;
        }
        else if ( AC.Streams[i].type == FILESTREAM ) {
            UBYTE *org2;
            org2 = AC.Streams[i].buffer;
            AC.Streams[i].buffer = (UBYTE*)Malloc1(AC.Streams[i].buffersize, "buffer");
            ofs = AC.Streams[i].buffer - org2;
            AC.Streams[i].pointer += ofs;
            AC.Streams[i].top += ofs;

            /* open file except for already opened main input file */
            if ( i ) {
                AC.Streams[i].handle = OpenFile((char *)(AC.Streams[i].name));
                if ( AC.Streams[i].handle == -1 ) {
                    MesPrint("ERROR: Could not reopen stream %s!",AC.Streams[i].name);
                    Terminate(-1);
                }
            }

            PUTZERO(pos);
            ADDPOS(pos, AC.Streams[i].bufferposition);
            SeekFile(AC.Streams[i].handle, &pos, SEEK_SET);

            AC.Streams[i].inbuffer = ReadFile(AC.Streams[i].handle, AC.Streams[i].buffer,
AC.Streams[i].inbuffer);

            SETBASEPOSITION(pos, AC.Streams[i].fileposition);
            SeekFile(AC.Streams[i].handle, &pos, SEEK_SET);
        }
        /*
         * Ideally, we should check if we have a type PREREADSTREAM, PREREADSTREAM2, and
         * PRECALCSTREAM here. If so, we should free element name and point it
         * to the name element of the embracing stream's struct. In practice,
         * this is undoable without adding new data to STREAM. Since we create
         * only a small memory leak here (some few byte for each existing
         * stream of these types) and only once when we do a recovery, we
```

```
         * tolerate this leak and keep it STREAM as it is.
         */
    }
    ofs = (UBYTE*)AC.Streams - org;
    AC.CurrentStream = (STREAM*)((UBYTE*)AC.CurrentStream + ofs);

    if ( AC.termstack ) {
        R_COPY_B(AC.termstack, AC.maxtermlevel*(LONG)sizeof(LONG), LONG*);
    }

    if ( AC.termsortstack ) {
        R_COPY_B(AC.termsortstack, AC.maxtermlevel*(LONG)sizeof(LONG), LONG*);
    }

    /* exception: here we also change values from struct AM */
    R_COPY_B(AC.cmod, AM.MaxTal*4*(LONG)sizeof(UWORD), UWORD*);
    AM.gcmod = AC.cmod + AM.MaxTal;
    AC.powmod = AM.gcmod + AM.MaxTal;
    AM.gpowmod = AC.powmod + AM.MaxTal;

    AC.modpowers = 0;
    AC.halfmod = 0;

    /* we don't care about AC.ProtoType/WildC */

    if ( AC.IfHeap ) {
        ofs = AC.IfStack - AC.IfHeap;
        R_COPY_B(AC.IfHeap, (LONG)sizeof(LONG)*(AC.MaxIf+1), LONG*);
        AC.IfStack = AC.IfHeap + ofs;
        R_COPY_B(AC.IfCount, (LONG)sizeof(LONG)*(AC.MaxIf+1), LONG*);
    }

    org = AC.iBuffer;
    size = AC.iStop - AC.iBuffer + 2;
    R_COPY_B(AC.iBuffer, size, UBYTE*);
    ofs = AC.iBuffer - org;
    AC.iPointer += ofs;
    AC.iStop += ofs;

    if ( AC.LabelNames ) {
        org = (UBYTE*)AC.LabelNames;
        R_COPY_B(AC.LabelNames, AC.MaxLabels*(LONG)(sizeof(UBYTE*)+sizeof(WORD)), UBYTE**);
        for ( i=0; i<AC.NumLabels; ++i ) {
            R_COPY_S(AC.LabelNames[i],UBYTE*);
        }
        ofs = (UBYTE*)AC.LabelNames - org;
        AC.Labels = (int*)((UBYTE*)AC.Labels + ofs);
    }

    R_COPY_B(AC.FixIndices, AM.OffsetIndex*(LONG)sizeof(WORD), WORD*);

    if ( AC.termsumcheck ) {
        R_COPY_B(AC.termsumcheck, AC.maxtermlevel*(LONG)sizeof(WORD), WORD*);
    }

    R_COPY_B(AC.WildcardNames, AC.WildcardBufferSize, UBYTE*);

    if ( AC.tokens ) {
        size = AC.toptokens - AC.tokens;
        if ( size ) {
            org = (UBYTE*)AC.tokens;
            R_COPY_B(AC.tokens, size, SBYTE*);
            ofs = (UBYTE*)AC.tokens - org;
            AC.endoftokens += ofs;
            AC.toptokens += ofs;
        }
        else {
            AC.tokens = 0;
            AC.endoftokens = AC.tokens;
            AC.toptokens = AC.tokens;
        }
    }

    R_COPY_B(AC.tokenarglevel, AM.MaxParLevel*(LONG)sizeof(WORD), WORD*);

    AC.modinverses = 0;

    R_COPY_S(AC.Fortran90Kind,UBYTE *);

#ifdef WITHPTHREADS
    if ( AC.inputnumbers ) {
        org = (UBYTE*)AC.inputnumbers;
        R_COPY_B(AC.inputnumbers, AC.sizepfirstnum*(LONG)(sizeof(WORD)+sizeof(LONG)), LONG*);
        ofs = (UBYTE*)AC.inputnumbers - org;
        AC.pfirstnum = (WORD*)((UBYTE*)AC.pfirstnum + ofs);
    }
    AC.halfmodlock = dummylock;
```

```
#endif /* ifdef WITHPTHREADS */

    if ( AC.IfSumCheck ) {
        R_COPY_B(AC.IfSumCheck, (LONG)sizeof(WORD)*(AC.MaxIf+1), WORD*);
    }
    if ( AC.CommuteInSet ) {
        R_COPY_B(AC.CommuteInSet, (LONG)sizeof(WORD)*(AC.SizeCommuteInSet+1), WORD*);
    }

    AC.LogHandle = oldLogHandle;

    R_COPY_S(AC.CheckpointRunAfter,char*);
    R_COPY_S(AC.CheckpointRunBefore,char*);

    R_COPY_S(AC.extrasym,UBYTE *);

#ifdef PRINTDEBUG
    print_C();
#endif

    /*#] AC : */
    /*#[ AP : */

    /* #[ AP free pointers */

    /* AP will be overwritten by data from the recovery file, therefore
     * dynamically allocated memory must be freed first. */

    for ( i=0; i<AP.DollarList.num; ++i ) {
        if ( Dollars[i].size ) {
            R_FREE(Dollars[i].where);
        }
        CleanDollarFactors(Dollars+i);
    }
    R_FREE(AP.DollarList.lijst);

    for ( i=0; i<AP.PreVarList.num; ++i ) {
        R_FREE(PreVar[i].name);
    }
    R_FREE(AP.PreVarList.lijst);

    for ( i=0; i<AP.LoopList.num; ++i ) {
        R_FREE(DoLoops[i].p.buffer);
        if ( DoLoops[i].dollarname ) {
            R_FREE(DoLoops[i].dollarname);
        }
    }
    R_FREE(AP.LoopList.lijst);

    for ( i=0; i<AP.ProcList.num; ++i ) {
        R_FREE(Procedures[i].p.buffer);
        R_FREE(Procedures[i].name);
    }
    R_FREE(AP.ProcList.lijst);

    for ( i=1; i<=AP.PreSwitchLevel; ++i ) {
        R_FREE(AP.PreSwitchStrings[i]);
    }
    R_FREE(AP.PreSwitchStrings);
    R_FREE(AP.preStart);
    R_FREE(AP.procedureExtension);
    R_FREE(AP.cprocedureExtension);
    R_FREE(AP.PreIfStack);
    R_FREE(AP.PreSwitchModes);
    R_FREE(AP.PreTypes);

    /* #] AP free pointers */

    /* first we copy AP as a whole and then restore the pointer structures step
       by step. */

    AP = *((struct P_const*)p); p = (unsigned char*)p + sizeof(struct P_const);
#ifdef WITHPTHREADS
    AP.PreVarLock = dummylock;
#endif

    R_COPY_LIST(AP.DollarList);
    for ( i=0; i<AP.DollarList.num; ++i ) {
        DOLLARS d = Dollars + i;
        size = d->size * sizeof(WORD);
        if ( size && d->where && d->where != oldAMdollarzero ) {
            R_COPY_B(d->where, size, void*);
        }
#ifdef WITHPTHREADS
        d->pthreadslockread = dummylock;
        d->pthreadslockwrite = dummylock;
#endif
```

```
        if ( d->nfactors > 1 ) {
            R_COPY_B(d->factors,sizeof(FACDOLLAR)*d->nfactors,FACDOLLAR*);
            for ( j = 0; j < d->nfactors; j++ ) {
                if ( d->factors[j].size > 0 ) {
                    R_COPY_B(d->factors[i].where,sizeof(WORD)*(d->factors[j].size+1),WORD*);
                }
            }
        }
    }
    AP.DollarList.message = "$-variable";

    R_COPY_LIST(AP.PreVarList);
    for ( i=0; i<AP.PreVarList.num; ++i ) {
        R_SET(size, size_t);
        org = PreVar[i].name;
        R_COPY_B(PreVar[i].name, size, UBYTE*);
        ofs = PreVar[i].name - org;
        if ( PreVar[i].value ) PreVar[i].value += ofs;
        if ( PreVar[i].argnames ) PreVar[i].argnames += ofs;
    }
    AP.PreVarList.message = "PreVariable";

    R_COPY_LIST(AP.LoopList);
    for ( i=0; i<AP.LoopList.num; ++i ) {
        org = DoLoops[i].p.buffer;
        R_COPY_B(DoLoops[i].p.buffer, DoLoops[i].p.size, UBYTE*);
        ofs = DoLoops[i].p.buffer - org;
        if ( DoLoops[i].name ) DoLoops[i].name += ofs;
        if ( DoLoops[i].vars ) DoLoops[i].vars += ofs;
        if ( DoLoops[i].contents ) DoLoops[i].contents += ofs;
        if ( DoLoops[i].type == ONEEXPRESSION ) {
            R_COPY_S(DoLoops[i].dollarname,UBYTE*);
        }
    }
    AP.LoopList.message = "doloop";

    R_COPY_LIST(AP.ProcList);
    for ( i=0; i<AP.ProcList.num; ++i ) {
        if ( Procedures[i].p.size ) {
            if ( Procedures[i].loadmode != 1 ) {
                R_COPY_B(Procedures[i].p.buffer, Procedures[i].p.size, UBYTE*);
            }
            else {
                R_SET(j, int);
                Procedures[i].p.buffer = Procedures[j].p.buffer;
            }
        }
        R_COPY_S(Procedures[i].name,UBYTE*);
    }
    AP.ProcList.message = "procedure";

    size = (AP.NumPreSwitchStrings+1)*(LONG)sizeof(UBYTE*);
    R_COPY_B(AP.PreSwitchStrings, size, UBYTE**);
    for ( i=1; i<=AP.PreSwitchLevel; ++i ) {
        R_COPY_S(AP.PreSwitchStrings[i],UBYTE*);
    }

    org = AP.preStart;
    R_COPY_B(AP.preStart, AP.pSize, UBYTE*);
    ofs = AP.preStart - org;
    if ( AP.preFill ) AP.preFill += ofs;
    if ( AP.preStop ) AP.preStop += ofs;

    R_COPY_S(AP.procedureExtension,UBYTE*);
    R_COPY_S(AP.cprocedureExtension,UBYTE*);

    R_COPY_B(AP.PreAssignStack,AP.MaxPreAssignLevel*(LONG)sizeof(LONG),LONG*);
    R_COPY_B(AP.PreIfStack, AP.MaxPreIfLevel*(LONG)sizeof(int), int*);
    R_COPY_B(AP.PreSwitchModes, (AP.NumPreSwitchStrings+1)*(LONG)sizeof(int), int*);
    R_COPY_B(AP.PreTypes, (AP.MaxPreTypes+1)*(LONG)sizeof(int), int*);

#ifdef PRINTDEBUG
    print_P();
#endif

    /*#] AP : */
    /*#[ AR : */

    R_SET(ofs,LONG);
    if ( ofs ) {
        AR.infile = AR.Fscr+1;
        AR.outfile = AR.Fscr;
        AR.hidefile = AR.Fscr+2;
    }
    else {
        AR.infile = AR.Fscr;
        AR.outfile = AR.Fscr+1;
```

```
        AR.hidefile = AR.Fscr+2;
    }

    /* #[ AR free pointers */

    /* Parts of AR will be overwritten by data from the recovery file, therefore
     * dynamically allocated memory must be freed first. */

    namebufout = AR.outfile->name;
    namebufhide = AR.hidefile->name;
    R_FREE(AR.outfile->PObuffer);
#ifdef WITHZLIB
    R_FREE(AR.outfile->zsp);
    R_FREE(AR.outfile->ziobuffer);
#endif
    namebufhide = AR.hidefile->name;
    R_FREE(AR.hidefile->PObuffer);
#ifdef WITHZLIB
    R_FREE(AR.hidefile->zsp);
    R_FREE(AR.hidefile->ziobuffer);
#endif
    /* no files should be opened -> nothing to do with handle */

    /* #] AR free pointers */

    /* outfile */
    R_SET(*AR.outfile, FILEHANDLE);
    org = (UBYTE*)AR.outfile->PObuffer;
    size = AR.outfile->POfull - AR.outfile->PObuffer;
    AR.outfile->PObuffer = (WORD*)Malloc1(AR.outfile->POsize, "PObuffer");
    if ( size ) {
        memcpy(AR.outfile->PObuffer, p, size*sizeof(WORD));
        p = (unsigned char*)p + size*sizeof(WORD);
    }
    ofs = (UBYTE*)AR.outfile->PObuffer - org;
    AR.outfile->POstop = (WORD*)((UBYTE*)AR.outfile->POstop + ofs);
    AR.outfile->POfill = (WORD*)((UBYTE*)AR.outfile->POfill + ofs);
    AR.outfile->POfull = (WORD*)((UBYTE*)AR.outfile->POfull + ofs);
    AR.outfile->name = namebufout;
#ifdef WITHPTHREADS
    AR.outfile->wPObuffer = AR.outfile->PObuffer;
    AR.outfile->wPOstop = AR.outfile->POstop;
    AR.outfile->wPOfill = AR.outfile->POfill;
    AR.outfile->wPOfull = AR.outfile->POfull;
#endif
#ifdef WITHZLIB
    /* zsp and ziobuffer will be allocated when used */
    AR.outfile->zsp = 0;
    AR.outfile->ziobuffer = 0;
#endif
    /* reopen old outfile */
#ifdef WITHMPI
    if(PF.me==MASTER)
#endif
    if ( AR.outfile->handle >= 0 ) {
        if ( CopyFile(sortfile, AR.outfile->name) ) {
            MesPrint("ERROR: Could not copy old output sort file %s!",sortfile);
            Terminate(-1);
        }
        AR.outfile->handle = ReOpenFile(AR.outfile->name);
        if ( AR.outfile->handle == -1 ) {
            MesPrint("ERROR: Could not reopen output sort file %s!",AR.outfile->name);
            Terminate(-1);
        }
        SeekFile(AR.outfile->handle, &AR.outfile->POposition, SEEK_SET);
    }

    /* hidefile */
    R_SET(*AR.hidefile, FILEHANDLE);
    AR.hidefile->name = namebufhide;
    if ( AR.hidefile->PObuffer ) {
        org = (UBYTE*)AR.hidefile->PObuffer;
        size = AR.hidefile->POfull - AR.hidefile->PObuffer;
        AR.hidefile->PObuffer = (WORD*)Malloc1(AR.hidefile->POsize, "PObuffer");
        if ( size ) {
            memcpy(AR.hidefile->PObuffer, p, size*sizeof(WORD));
            p = (unsigned char*)p + size*sizeof(WORD);
        }
        ofs = (UBYTE*)AR.hidefile->PObuffer - org;
        AR.hidefile->POstop = (WORD*)((UBYTE*)AR.hidefile->POstop + ofs);
        AR.hidefile->POfill = (WORD*)((UBYTE*)AR.hidefile->POfill + ofs);
        AR.hidefile->POfull = (WORD*)((UBYTE*)AR.hidefile->POfull + ofs);
#ifdef WITHPTHREADS
        AR.hidefile->wPObuffer = AR.hidefile->PObuffer;
        AR.hidefile->wPOstop = AR.hidefile->POstop;
        AR.hidefile->wPOfill = AR.hidefile->POfill;
        AR.hidefile->wPOfull = AR.hidefile->POfull;
```

```
#endif
    }
#ifdef WITHZLIB
        /* zsp and ziobuffer will be allocated when used */
        AR.hidefile->zsp = 0;
        AR.hidefile->ziobuffer = 0;
#endif
    /* reopen old hidefile */
    if ( AR.hidefile->handle >= 0 ) {
        if ( CopyFile(hidefile, AR.hidefile->name) ) {
            MesPrint("ERROR: Could not copy old hide file %s!",hidefile);
            Terminate(-1);
        }
        AR.hidefile->handle = ReOpenFile(AR.hidefile->name);
        if ( AR.hidefile->handle == -1 ) {
            MesPrint("ERROR: Could not reopen hide file %s!",AR.hidefile->name);
            Terminate(-1);
        }
        SeekFile(AR.hidefile->handle, &AR.hidefile->POposition, SEEK_SET);
    }

    /* store file */
    R_SET(pos, POSITION);
    if ( ISNOTZEROPOS(pos) ) {
        CloseFile(AR.StoreData.Handle);
        R_SET(AR.StoreData, FILEDATA);
        if ( CopyFile(storefile, FG.fname) ) {
            MesPrint("ERROR: Could not copy old store file %s!",storefile);
            Terminate(-1);
        }
        AR.StoreData.Handle = (WORD)ReOpenFile(FG.fname);
        SeekFile(AR.StoreData.Handle, &AR.StoreData.Position, SEEK_SET);
    }

    R_SET(AR.DefPosition, POSITION);
    R_SET(AR.OldTime, LONG);
    R_SET(AR.InInBuf, LONG);
    R_SET(AR.InHiBuf, LONG);

    R_SET(AR.NoCompress, int);
    R_SET(AR.gzipCompress, int);

    R_SET(AR.outtohide, int);

    R_SET(AR.GetFile, WORD);
    R_SET(AR.KeptInHold, WORD);
    R_SET(AR.BracketOn, WORD);
    R_SET(AR.MaxBracket, WORD);
    R_SET(AR.CurDum, WORD);
    R_SET(AR.DeferFlag, WORD);
    R_SET(AR.TePos, WORD);
    R_SET(AR.sLevel, WORD);
    R_SET(AR.Stage4Name, WORD);
    R_SET(AR.GetOneFile, WORD);
    R_SET(AR.PolyFun, WORD);
    R_SET(AR.PolyFunInv, WORD);
    R_SET(AR.PolyFunType, WORD);
    R_SET(AR.PolyFunExp, WORD);
    R_SET(AR.PolyFunVar, WORD);
    R_SET(AR.PolyFunPow, WORD);
    R_SET(AR.Eside, WORD);
    R_SET(AR.MaxDum, WORD);
    R_SET(AR.level, WORD);
    R_SET(AR.expchanged, WORD);
    R_SET(AR.expflags, WORD);
    R_SET(AR.CurExpr, WORD);
    R_SET(AR.SortType, WORD);
    R_SET(AR.ShortSortCount, WORD);

    /* this is usually done in Process(), but sometimes FORM doesn't
       end up executing Process() before it uses the AR.CompressPointer,
       so we need to explicitely set it here. */
    AR.CompressPointer = AR.CompressBuffer;

#ifdef WITHPTHREADS
    for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
        R_SET(AB[j]->R.wranfnpair1, int);
        R_SET(AB[j]->R.wranfnpair2, int);
        R_SET(AB[j]->R.wranfcall, int);
        R_SET(AB[j]->R.wranfseed, ULONG);
        R_SET(AB[j]->R.wranfia,ULONG*);
        if ( AB[j]->R.wranfia ) {
            R_COPY_B(AB[j]->R.wranfia, sizeof(ULONG)*AB[j]->R.wranfnpair2, ULONG*);
        }
    }
#else
    R_SET(AR.wranfnpair1, int);
```

```
    R_SET(AR.wranfnpair2, int);
    R_SET(AR.wranfcall, int);
    R_SET(AR.wranfseed, ULONG);
    R_SET(AR.wranfia,ULONG*);
    if ( AR.wranfia ) {
        R_COPY_B(AR.wranfia, sizeof(ULONG)*AR.wranfnpair2, ULONG*);
    }
#endif

#ifdef PRINTDEBUG
    print_R();
#endif

    /*#] AR : */
    /*#[ AO :*/
/*
    We copy all non-pointer variables.
*/
    l = sizeof(A.O) - ((UBYTE *)(&(A.O.NumInBrack))-(UBYTE *)(&A.O));
    memcpy(&(A.O.NumInBrack), p, l); p = (unsigned char*)p + l;
/*
    Now the variables in OptimizeResult
*/
    memcpy(&(A.O.OptimizeResult),p,sizeof(OPTIMIZERESULT));
        p = (unsigned char*)p + sizeof(OPTIMIZERESULT);

    if ( A.O.OptimizeResult.codesize > 0 ) {
        R_COPY_B(A.O.OptimizeResult.code,A.O.OptimizeResult.codesize*sizeof(WORD),WORD *);
    }
    R_COPY_S(A.O.OptimizeResult.nameofexpr,UBYTE *);
/*
    And now the dictionaries. We know how many there are. We also know
    how many elements the array AO.Dictionaries should have.
*/
    if ( AO.SizeDictionaries > 0 ) {
        AO.Dictionaries = (DICTIONARY **)Malloc1(AO.SizeDictionaries*sizeof(DICTIONARY *),
                    "Dictionaries");
        for ( i = 0; i < AO.NumDictionaries; i++ ) {
            R_SET(l,LONG)
            AO.Dictionaries[i] = DictFromBytes(p);
            p = (char *)p + l;
        }
    }
    /*#] AO :*/
#ifdef WITHMPI
    /*#[ PF : */
    {/*Block*/
        int numtasks;
        R_SET(numtasks, int);
        if(numtasks!=PF.numtasks){
            MesPrint("%d number of tasks expected instead of %d; use mpirun -np %d",
                            numtasks,PF.numtasks,numtasks);
            if(PF.me!=MASTER)
                remove(RecoveryFilename());
            Terminate(-1);
        }
    }/*Block*/
    R_SET(PF.rhsInParallel, int);
    R_SET(PF.exprbufsize, int);
    R_SET(PF.log, int);
    /*#] PF : */
#endif

#ifdef WITHPTHREADS
    /* read timing information of individual threads */
    R_SET(i, int);
    for ( j=1; j<AM.totalnumberofthreads; ++j ) {
        /* ... and correcting OldTime */
        AB[j]->R.OldTime = -(*((LONG*)p+j));
    }
    WriteTimerInfo((LONG*)p,(LONG *)((unsigned char*)p + i*(LONG)sizeof(LONG)));
    p = (unsigned char*)p + 2*i*(LONG)sizeof(LONG);
#endif /* ifdef WITHPTHREADS */

    if ( fclose(fd) ) return(__LINE__);

    M_free(buf,"recovery buffer");

    /* cares about data in S_const */
    UpdatePositions();
    AT.SS = AT.S0;
/*
    Set the checkpoint parameter right for the next checkpoint.
*/
    AC.CheckpointStamp = TimeWallClock(1);

    done_snapshot = 1;
```

```
    MesPrint("done."); fflush(0);

    return(0);
}

/*
    #] DoRecovery :
    #[ DoSnapshot :
*/

static int DoSnapshot(int moduletype)
{
    GETIDENTITY
    FILE *fd;
    POSITION pos;
    int i, j;
    LONG l;
    WORD *w;
    void *adr;
#ifdef WITHPTHREADS
    LONG *longp,*longpp;
#endif /* ifdef WITHPTHREADS */

    MesPrint("Saving recovery point ... %"); fflush(0);
#ifdef PRINTTIMEMARKS
    MesPrint("\n");
#endif

    if ( !(fd = fopen(intermedfile, "wb")) ) return(__LINE__);

    /* reserve space in the file for a length field */
    if ( fwrite(&pos, 1, sizeof(POSITION), fd) != sizeof(POSITION) ) return(__LINE__);

    /* write moduletype */
    if ( fwrite(&moduletype, 1, sizeof(int), fd) != sizeof(int) ) return(__LINE__);

    /*#[ AM :*/

    /* since most values don't change during execution, AM doesn't need to be
     * written as a whole. all values will be correctly set when starting up
     * anyway. only the exceptions need to be taken care of. see MakeGlobal()
     * and PopVariables() in execute.c. */

    ANNOUNCE(AM)
    S_WRITE_B(&AM.hparallelflag, sizeof(int));
    S_WRITE_B(&AM.gparallelflag, sizeof(int));
    S_WRITE_B(&AM.gCodesFlag, sizeof(int));
    S_WRITE_B(&AM.gNamesFlag, sizeof(int));
    S_WRITE_B(&AM.gStatsFlag, sizeof(int));
    S_WRITE_B(&AM.gTokensWriteFlag, sizeof(int));
    S_WRITE_B(&AM.gNoSpacesInNumbers, sizeof(int));
    S_WRITE_B(&AM.gIndentSpace, sizeof(WORD));
    S_WRITE_B(&AM.gUnitTrace, sizeof(WORD));
    S_WRITE_B(&AM.gDefDim, sizeof(int));
    S_WRITE_B(&AM.gDefDim4, sizeof(int));
    S_WRITE_B(&AM.gncmod, sizeof(WORD));
    S_WRITE_B(&AM.gnpowmod, sizeof(WORD));
    S_WRITE_B(&AM.gmodmode, sizeof(WORD));
    S_WRITE_B(&AM.gOutputMode, sizeof(WORD));
    S_WRITE_B(&AM.gCnumpows, sizeof(WORD));
    S_WRITE_B(&AM.gOutputSpaces, sizeof(WORD));
    S_WRITE_B(&AM.gOutNumberType, sizeof(WORD));
    S_WRITE_B(&AM.gfunpowers, sizeof(int));
    S_WRITE_B(&AM.gPolyFun, sizeof(WORD));
    S_WRITE_B(&AM.gPolyFunInv, sizeof(WORD));
    S_WRITE_B(&AM.gPolyFunType, sizeof(WORD));
    S_WRITE_B(&AM.gPolyFunExp, sizeof(WORD));
    S_WRITE_B(&AM.gPolyFunVar, sizeof(WORD));
    S_WRITE_B(&AM.gPolyFunPow, sizeof(WORD));
    S_WRITE_B(&AM.gProcessBucketSize, sizeof(LONG));
    S_WRITE_B(&AM.OldChildTime, sizeof(LONG));
    S_WRITE_B(&AM.OldSecTime, sizeof(LONG));
    S_WRITE_B(&AM.OldMilliTime, sizeof(LONG));
    S_WRITE_B(&AM.gproperorderflag, sizeof(int));
    S_WRITE_B(&AM.gThreadBucketSize, sizeof(LONG));
    S_WRITE_B(&AM.gSizeCommuteInSet, sizeof(int));
    S_WRITE_B(&AM.gThreadStats, sizeof(int));
    S_WRITE_B(&AM.gFinalStats, sizeof(int));
    S_WRITE_B(&AM.gThreadsFlag, sizeof(int));
    S_WRITE_B(&AM.gThreadBalancing, sizeof(int));
    S_WRITE_B(&AM.gThreadSortFileSynch, sizeof(int));
    S_WRITE_B(&AM.gProcessStats, sizeof(int));
    S_WRITE_B(&AM.gOldParallelStats, sizeof(int));
    S_WRITE_B(&AM.gSortType, sizeof(int));
    S_WRITE_B(&AM.gShortStatsMax, sizeof(WORD));
    S_WRITE_B(&AM.gIsFortran90, sizeof(int));
    adr = &AM.dollarzero;
```

```
    S_WRITE_B(&adr, sizeof(void*));
    S_WRITE_B(&AM.gFortran90Kind,sizeof(UBYTE *));
    S_WRITE_S(AM.gFortran90Kind);

    S_WRITE_S(AM.gextrasym);
    S_WRITE_S(AM.ggextrasym);

    S_WRITE_B(&AM.PrintTotalSize,sizeof(int));
    S_WRITE_B(&AM.fbuffersize,sizeof(int));
    S_WRITE_B(&AM.gOldFactArgFlag,sizeof(int));
    S_WRITE_B(&AM.ggOldFactArgFlag,sizeof(int));

    S_WRITE_B(&AM.gnumextrasym,sizeof(int));
    S_WRITE_B(&AM.ggnumextrasym,sizeof(int));
    S_WRITE_B(&AM.NumSpectatorFiles,sizeof(int));
    S_WRITE_B(&AM.SizeForSpectatorFiles,sizeof(int));
    S_WRITE_B(&AM.gOldGCDflag,sizeof(int));
    S_WRITE_B(&AM.ggOldGCDflag,sizeof(int));
    S_WRITE_B(&AM.gWTimeStatsFlag, sizeof(int));

    S_WRITE_B(&AM.Path,sizeof(UBYTE *));
    S_WRITE_S(AM.Path);

    /*#] AM :*/
    /*#[ AC :*/

    /* we write AC as a whole and then write all additional data step by step.
     * AC.DubiousList doesn't need to be treated, because it should be empty. */

    ANNOUNCE(AC)
    S_WRITE_B(&AC, sizeof(struct C_const));

    S_WRITE_NAMETREE(AC.dollarnames);
    S_WRITE_NAMETREE(AC.exprnames);
    S_WRITE_NAMETREE(AC.varnames);

    S_WRITE_LIST(AC.ChannelList);
    for ( i=0; i<AC.ChannelList.num; ++i ) {
        S_WRITE_S(channels[i].name);
    }

    ANNOUNCE(AC.FunctionList)
    S_WRITE_LIST(AC.FunctionList);
    for ( i=0; i<AC.FunctionList.num; ++i ) {
        /* if the function is a table */
        if ( functions[i].tabl ) {
            TABLES tabl = functions[i].tabl;
            S_WRITE_B(tabl, sizeof(struct TaBlEs));
            if ( tabl->tablepointers ) {
                if ( tabl->sparse ) {
                    /* sparse tables. reserved holds number of allocated
                     * elements. the size of an element is numind plus
                     * TABLEEXTENSION times the size of WORD. */
                    S_WRITE_B(tabl->tablepointers,
                        tabl->reserved*sizeof(WORD)*(tabl->numind+TABLEEXTENSION));
                }
                else {
                    /* matrix like tables. */
                    S_WRITE_B(tabl->tablepointers,
                        TABLEEXTENSION*sizeof(WORD)*(tabl->totind));
                }
            }
            S_WRITE_B(tabl->prototype, tabl->prototypeSize);
            S_WRITE_B(tabl->mm, tabl->numind*(LONG)sizeof(MINMAX));
            S_WRITE_B(tabl->flags, tabl->numind*(LONG)sizeof(WORD));
            if ( tabl->sparse ) {
                S_WRITE_B(tabl->boomlijst, tabl->MaxTreeSize*(LONG)sizeof(COMPTREE));
                S_WRITE_S(tabl->argtail);
            }
            S_WRITE_B(tabl->buffers, tabl->bufferssize*(LONG)sizeof(WORD));
            if ( tabl->spare ) {
                TABLES spare = tabl->spare;
                S_WRITE_B(spare, sizeof(struct TaBlEs));
                if ( spare->tablepointers ) {
                    if ( spare->sparse ) {
                        /* sparse tables */
                        S_WRITE_B(spare->tablepointers,
                            spare->reserved*sizeof(WORD)*(spare->numind+TABLEEXTENSION));
                    }
                    else {
                        /* matrix like tables */
                        S_WRITE_B(spare->tablepointers,
                            TABLEEXTENSION*sizeof(WORD)*(spare->totind));
                    }
                }
                S_WRITE_B(spare->mm, spare->numind*(LONG)sizeof(MINMAX));
                S_WRITE_B(spare->flags, spare->numind*(LONG)sizeof(WORD));
```

```
                if ( spare->sparse ) {
                    S_WRITE_B(spare->boomlijst, spare->MaxTreeSize*(LONG)sizeof(COMPTREE));
                }
                S_WRITE_B(spare->buffers, spare->bufferssize*(LONG)sizeof(WORD));
            }
        }
    }

    ANNOUNCE(AC.ExpressionList)
    S_WRITE_LIST(AC.ExpressionList);
    for ( i=0; i<AC.ExpressionList.num; ++i ) {
        EXPRESSIONS ex = Expressions + i;
        if ( ex->renum ) {
            S_WRITE_B(ex->renum, sizeof(struct ReNuMbEr));
            /* there is one dynamically allocated buffer for struct ReNuMbEr and
             * symb.lo points to its beginning. the size of the buffer is not
             * stored anywhere but we know it is 2*sizeof(WORD)*N, where N is
             * the number of all vectors, indices, functions and symbols. since
             * funum points into the buffer at a distance 2N-[Number of
             * functions] from symb.lo (see GetTable() in store.c), we can
             * calculate the buffer size by some pointer arithmetic. the size is
             * then written to the file. */
            l = ex->renum->funnum - ex->renum->symb.lo;
            l += ex->renum->funnum - ex->renum->func.lo;
            S_WRITE_B(&l, sizeof(size_t));
            S_WRITE_B(ex->renum->symb.lo, l);
        }
        if ( ex->bracketinfo ) {
            S_WRITE_B(ex->bracketinfo, sizeof(BRACKETINFO));
            S_WRITE_B(ex->bracketinfo->indexbuffer,
ex->bracketinfo->indexbuffersize*sizeof(BRACKETINDEX));
            S_WRITE_B(ex->bracketinfo->bracketbuffer,
ex->bracketinfo->bracketbuffersize*sizeof(WORD));
        }
        if ( ex->newbracketinfo ) {
            S_WRITE_B(ex->newbracketinfo, sizeof(BRACKETINFO));
            S_WRITE_B(ex->newbracketinfo->indexbuffer,
ex->newbracketinfo->indexbuffersize*sizeof(BRACKETINDEX));
            S_WRITE_B(ex->newbracketinfo->bracketbuffer,
ex->newbracketinfo->bracketbuffersize*sizeof(WORD));
        }
        /* don't need to write ex->renumlists */
        if ( ex->inmem ) {
            /* size of the inmem buffer has to be determined. we use the fact
             * that the end of an expression is marked by a zero. */
            w = ex->inmem;
            while ( *w++ ) ;
            l = w - ex->inmem;
            S_WRITE_B(&l, sizeof(size_t));
            S_WRITE_B(ex->inmem, l);
        }
    }

    ANNOUNCE(AC.IndexList)
    S_WRITE_LIST(AC.IndexList);
    S_WRITE_LIST(AC.SetElementList);
    S_WRITE_LIST(AC.SetList);
    S_WRITE_LIST(AC.SymbolList);
    S_WRITE_LIST(AC.VectorList);

    ANNOUNCE(AC.TableBaseList)
    S_WRITE_LIST(AC.TableBaseList);
    for ( i=0; i<AC.TableBaseList.num; ++i ) {
        /* see struct dbase in minos.h */
        if ( tablebases[i].iblocks ) {
            S_WRITE_B(tablebases[i].iblocks, tablebases[i].info.numberofindexblocks *
sizeof(INDEXBLOCK*));
            for ( j=0; j<tablebases[i].info.numberofindexblocks; ++j ) {
                if ( tablebases[i].iblocks[j] ) {
                    S_WRITE_B(tablebases[i].iblocks[j], sizeof(INDEXBLOCK));
                }
            }
        }
        if ( tablebases[i].nblocks ) {
            S_WRITE_B(tablebases[i].nblocks, tablebases[i].info.numberofnamesblocks *
sizeof(NAMESBLOCK*));
            for ( j=0; j<tablebases[i].info.numberofnamesblocks; ++j ) {
                if ( tablebases[i].nblocks[j] ) {
                    S_WRITE_B(tablebases[i].nblocks[j], sizeof(NAMESBLOCK));
                }
            }
        }
        S_WRITE_S(tablebases[i].name);
        S_WRITE_S(tablebases[i].fullname);
        S_WRITE_S(tablebases[i].tablenames);
    }

```

```
    ANNOUNCE(AC.cbufList)
    S_WRITE_LIST(AC.cbufList);
    for ( i=0; i<AC.cbufList.num; ++i ) {
        S_WRITE_B(cbuf[i].Buffer, cbuf[i].BufferSize*sizeof(WORD));
        /* see inicbufs in comtool.c */
        S_WRITE_B(cbuf[i].lhs, cbuf[i].maxlhs*(LONG)sizeof(WORD*));
        S_WRITE_B(cbuf[i].rhs, cbuf[i].maxrhs*(LONG)(sizeof(WORD*)+2*sizeof(LONG)+2*sizeof(WORD)));
        if ( cbuf[i].boomlijst ) {
            S_WRITE_B(cbuf[i].boomlijst, cbuf[i].MaxTreeSize*(LONG)sizeof(COMPTREE));
        }
    }

    S_WRITE_LIST(AC.AutoSymbolList);
    S_WRITE_LIST(AC.AutoIndexList);
    S_WRITE_LIST(AC.AutoVectorList);
    S_WRITE_LIST(AC.AutoFunctionList);

    S_WRITE_NAMETREE(AC.autonames);

    ANNOUNCE(AC.Streams)
    S_WRITE_B(AC.Streams, AC.MaxNumStreams*(LONG)sizeof(STREAM));
    for ( i=0; i<AC.NumStreams; ++i ) {
        if ( AC.Streams[i].inbuffer ) {
            S_WRITE_B(AC.Streams[i].buffer, AC.Streams[i].inbuffer);
        }
        S_WRITE_S(AC.Streams[i].FoldName);
        S_WRITE_S(AC.Streams[i].name);
    }

    if ( AC.termstack ) {
        S_WRITE_B(AC.termstack, AC.maxtermlevel*(LONG)sizeof(LONG));
    }

    if ( AC.termsortstack ) {
        S_WRITE_B(AC.termsortstack, AC.maxtermlevel*(LONG)sizeof(LONG));
    }

    S_WRITE_B(AC.cmod, AM.MaxTal*4*(LONG)sizeof(UWORD));

    if ( AC.IfHeap ) {
        S_WRITE_B(AC.IfHeap, (LONG)sizeof(LONG)*(AC.MaxIf+1));
        S_WRITE_B(AC.IfCount, (LONG)sizeof(LONG)*(AC.MaxIf+1));
    }

    l = AC.iStop - AC.iBuffer + 2;
    S_WRITE_B(AC.iBuffer, l);

    if ( AC.LabelNames ) {
        S_WRITE_B(AC.LabelNames, AC.MaxLabels*(LONG)(sizeof(UBYTE*)+sizeof(WORD)));
        for ( i=0; i<AC.NumLabels; ++i ) {
            S_WRITE_S(AC.LabelNames[i]);
        }
    }

    S_WRITE_B(AC.FixIndices, AM.OffsetIndex*(LONG)sizeof(WORD));

    if ( AC.termsumcheck ) {
        S_WRITE_B(AC.termsumcheck, AC.maxtermlevel*(LONG)sizeof(WORD));
    }

    S_WRITE_B(AC.WildcardNames, AC.WildcardBufferSize);

    if ( AC.tokens ) {
        l = AC.toptokens - AC.tokens;
        if ( l ) {
            S_WRITE_B(AC.tokens, l);
        }
    }

    S_WRITE_B(AC.tokenarglevel, AM.MaxParLevel*(LONG)sizeof(WORD));

    S_WRITE_S(AC.Fortran90Kind);

#ifdef WITHPTHREADS
    if ( AC.inputnumbers ) {
        S_WRITE_B(AC.inputnumbers, AC.sizepfirstnum*(LONG)(sizeof(WORD)+sizeof(LONG)));
    }
#endif /* ifdef WITHPTHREADS */

    if ( AC.IfSumCheck ) {
        S_WRITE_B(AC.IfSumCheck, (LONG)sizeof(WORD)*(AC.MaxIf+1));
    }
    if ( AC.CommuteInSet ) {
        S_WRITE_B(AC.CommuteInSet, (LONG)sizeof(WORD)*(AC.SizeCommuteInSet+1));
    }

    S_WRITE_S(AC.CheckpointRunAfter);
```

```
    S_WRITE_S(AC.CheckpointRunBefore);

    S_WRITE_S(AC.extrasym);

    /*#] AC :*/
    /*#[ AP :*/

    /* we write AP as a whole and then write all additional data step by step. */

    ANNOUNCE(AP)
    S_WRITE_B(&AP, sizeof(struct P_const));

    S_WRITE_LIST(AP.DollarList);
    for ( i=0; i<AP.DollarList.num; ++i ) {
        DOLLARS d = Dollars + i;
        S_WRITE_DOLLAR(Dollars[i]);
        if ( d->nfactors > 1 ) {
            S_WRITE_B(&(d->factors),sizeof(FACDOLLAR)*d->nfactors);
            for ( j = 0; j < d->nfactors; j++ ) {
                if ( d->factors[j].size > 0 ) {
                    S_WRITE_B(&(d->factors[i].where),sizeof(WORD)*(d->factors[j].size+1));
                }
            }
        }
    }

    S_WRITE_LIST(AP.PreVarList);
    for ( i=0; i<AP.PreVarList.num; ++i ) {
        /* there is one dynamically allocated buffer in struct pReVaR holding
         * the strings name, value and several argnames. the size of the buffer
         * can be calculated by adding up their string lengths. */
        l = strlen((char*)PreVar[i].name) + 1;
        if ( PreVar[i].value ) {
            l += strlen((char*)(PreVar[i].name+l)) + 1;
        }
        for ( j=0; j<PreVar[i].nargs; ++j ) {
            l += strlen((char*)(PreVar[i].name+l)) + 1;
        }
        S_WRITE_B(&l, sizeof(size_t));
        S_WRITE_B(PreVar[i].name, l);
    }

    ANNOUNCE(AP.LoopList)
    S_WRITE_LIST(AP.LoopList);
    for ( i=0; i<AP.LoopList.num; ++i ) {
        S_WRITE_B(DoLoops[i].p.buffer, DoLoops[i].p.size);
        if ( DoLoops[i].type == ONEEXPRESSION ) {
            /* do loops with an expression keep this expression in a dynamically
             * allocated buffer in dollarname */
            S_WRITE_S(DoLoops[i].dollarname);
        }
    }

    S_WRITE_LIST(AP.ProcList);
    for ( i=0; i<AP.ProcList.num; ++i ) {
        if ( Procedures[i].p.size ) {
            if ( Procedures[i].loadmode != 1 ) {
                S_WRITE_B(Procedures[i].p.buffer, Procedures[i].p.size);
            }
            else {
                for ( j=0; j<AP.ProcList.num; ++j ) {
                    if ( Procedures[i].p.buffer == Procedures[j].p.buffer ) {
                        break;
                    }
                }
                if ( j == AP.ProcList.num ) {
                    MesPrint("Error writing procedures to recovery file!");
                }
                S_WRITE_B(&j, sizeof(int));
            }
        }
        S_WRITE_S(Procedures[i].name);
    }

    S_WRITE_B(AP.PreSwitchStrings, (AP.NumPreSwitchStrings+1)*(LONG)sizeof(UBYTE*));
    for ( i=1; i<=AP.PreSwitchLevel; ++i ) {
        S_WRITE_S(AP.PreSwitchStrings[i]);
    }

    S_WRITE_B(AP.preStart, AP.pSize);

    S_WRITE_S(AP.procedureExtension);
    S_WRITE_S(AP.cprocedureExtension);

    S_WRITE_B(AP.PreAssignStack, AP.MaxPreAssignLevel*(LONG)sizeof(LONG));
    S_WRITE_B(AP.PreIfStack, AP.MaxPreIfLevel*(LONG)sizeof(int));
    S_WRITE_B(AP.PreSwitchModes, (AP.NumPreSwitchStrings+1)*(LONG)sizeof(int));
```

```
    S_WRITE_B(AP.PreTypes, (AP.MaxPreTypes+1)*(LONG)sizeof(int));

    /*#] AP :*/
    /*#[ AR :*/

    ANNOUNCE(AR)
    /* to remember which entry in AR.Fscr corresponds to the infile */
    l = AR.infile - AR.Fscr;
    S_WRITE_B(&l, sizeof(LONG));

    /* write the FILEHANDLEs */
    S_WRITE_B(AR.outfile, sizeof(FILEHANDLE));
    l = AR.outfile->POfull - AR.outfile->PObuffer;
    if ( l ) {
        S_WRITE_B(AR.outfile->PObuffer, l*sizeof(WORD));
    }
    S_WRITE_B(AR.hidefile, sizeof(FILEHANDLE));
    l = AR.hidefile->POfull - AR.hidefile->PObuffer;
    if ( l ) {
        S_WRITE_B(AR.hidefile->PObuffer, l*sizeof(WORD));
    }

    S_WRITE_B(&AR.StoreData.Fill, sizeof(POSITION));
    if ( ISNOTZEROPOS(AR.StoreData.Fill) ) {
        S_WRITE_B(&AR.StoreData, sizeof(FILEDATA));
    }

    S_WRITE_B(&AR.DefPosition, sizeof(POSITION));

    l = TimeCPU(1); l = -l;
    S_WRITE_B(&l, sizeof(LONG));

    ANNOUNCE(AR.InInBuf)
    S_WRITE_B(&AR.InInBuf, sizeof(LONG));
    S_WRITE_B(&AR.InHiBuf, sizeof(LONG));

    S_WRITE_B(&AR.NoCompress, sizeof(int));
    S_WRITE_B(&AR.gzipCompress, sizeof(int));

    S_WRITE_B(&AR.outtohide, sizeof(int));

    S_WRITE_B(&AR.GetFile, sizeof(WORD));
    S_WRITE_B(&AR.KeptInHold, sizeof(WORD));
    S_WRITE_B(&AR.BracketOn, sizeof(WORD));
    S_WRITE_B(&AR.MaxBracket, sizeof(WORD));
    S_WRITE_B(&AR.CurDum, sizeof(WORD));
    S_WRITE_B(&AR.DeferFlag, sizeof(WORD));
    S_WRITE_B(&AR.TePos, sizeof(WORD));
    S_WRITE_B(&AR.sLevel, sizeof(WORD));
    S_WRITE_B(&AR.Stage4Name, sizeof(WORD));
    S_WRITE_B(&AR.GetOneFile, sizeof(WORD));
    S_WRITE_B(&AR.PolyFun, sizeof(WORD));
    S_WRITE_B(&AR.PolyFunInv, sizeof(WORD));
    S_WRITE_B(&AR.PolyFunType, sizeof(WORD));
    S_WRITE_B(&AR.PolyFunExp, sizeof(WORD));
    S_WRITE_B(&AR.PolyFunVar, sizeof(WORD));
    S_WRITE_B(&AR.PolyFunPow, sizeof(WORD));
    S_WRITE_B(&AR.Eside, sizeof(WORD));
    S_WRITE_B(&AR.MaxDum, sizeof(WORD));
    S_WRITE_B(&AR.level, sizeof(WORD));
    S_WRITE_B(&AR.expchanged, sizeof(WORD));
    S_WRITE_B(&AR.expflags, sizeof(WORD));
    S_WRITE_B(&AR.CurExpr, sizeof(WORD));
    S_WRITE_B(&AR.SortType, sizeof(WORD));
    S_WRITE_B(&AR.ShortSortCount, sizeof(WORD));

#ifdef WITHPTHREADS
    for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
        S_WRITE_B(&(AB[j]->R.wranfnpair1), sizeof(int));
        S_WRITE_B(&(AB[j]->R.wranfnpair2), sizeof(int));
        S_WRITE_B(&(AB[j]->R.wranfcall), sizeof(int));
        S_WRITE_B(&(AB[j]->R.wranfseed), sizeof(ULONG));
        S_WRITE_B(&(AB[j]->R.wranfia),sizeof(ULONG *));
        if ( AB[j]->R.wranfia ) {
            S_WRITE_B(AB[j]->R.wranfia, sizeof(ULONG)*AB[j]->R.wranfnpair2);
        }
    }
#else
    S_WRITE_B(&(AR.wranfnpair1), sizeof(int));
    S_WRITE_B(&(AR.wranfnpair2), sizeof(int));
    S_WRITE_B(&(AR.wranfcall), sizeof(int));
    S_WRITE_B(&(AR.wranfseed), sizeof(ULONG));
    S_WRITE_B(&(AR.wranfia),sizeof(ULONG *));
    if ( AR.wranfia ) {
        S_WRITE_B(AR.wranfia, sizeof(ULONG)*AR.wranfnpair2);
    }
#endif
```

```
    /*#] AR :*/
    /*#[ AO :*/
/*
    We copy all non-pointer variables.
*/
    ANNOUNCE(AO)
    l = sizeof(A.O) - ((UBYTE *)(&(A.O.NumInBrack))-(UBYTE *)(&A.O));
    S_WRITE_B(&(A.O.NumInBrack),l);
/*
    Now the variables in OptimizeResult
*/
    S_WRITE_B(&(A.O.OptimizeResult),sizeof(OPTIMIZERESULT));
    if ( A.O.OptimizeResult.codesize > 0 ) {
        S_WRITE_B(A.O.OptimizeResult.code,A.O.OptimizeResult.codesize*sizeof(WORD));
    }
    S_WRITE_S(A.O.OptimizeResult.nameofexpr);
/*
    And now the dictionaries.
    We write each dictionary to a buffer and get the size of that buffer.
    Then we write the size and the buffer.
*/
    for ( i = 0; i < AO.NumDictionaries; i++ ) {
        l = DictToBytes(AO.Dictionaries[i],(UBYTE *)(AT.WorkPointer));
        S_WRITE_B(&l,sizeof(LONG));
        S_WRITE_B(AT.WorkPointer,l);
    }

    /*#] AO :*/
    /*#[ PF :*/
#ifdef WITHMPI
    S_WRITE_B(&PF.numtasks, sizeof(int));
    S_WRITE_B(&PF.rhsInParallel, sizeof(int));
    S_WRITE_B(&PF.exprbufsize, sizeof(int));
    S_WRITE_B(&PF.log, sizeof(int));
#endif
    /*#] PF :*/

#ifdef WITHPTHREADS

    ANNOUNCE(GetTimerInfo)
/*
    write timing information of individual threads
*/
    i = GetTimerInfo(&longp,&longpp);
    S_WRITE_B(&i, sizeof(int));
    S_WRITE_B(longp, i*(LONG)sizeof(LONG));
    S_WRITE_B(&i, sizeof(int));
    S_WRITE_B(longpp, i*(LONG)sizeof(LONG));

#endif

    S_FLUSH_B /* because we will call fwrite() directly in the following code */

    /* save length of data at the beginning of the file */
    ANNOUNCE(file length)
    SETBASEPOSITION(pos, (ftell(fd)));
    fseek(fd, 0, SEEK_SET);
    if ( fwrite(&pos, 1, sizeof(POSITION), fd) != sizeof(POSITION) ) return(__LINE__);
    fseek(fd, BASEPOSITION(pos), SEEK_SET);

    ANNOUNCE(file close)
    if ( fclose(fd) ) return(__LINE__);
#ifdef WITHMPI
    if ( PF.me == MASTER ) {
#endif
/*
        copy store file if necessary
*/
        ANNOUNCE(copy store file)
        if ( ISNOTZEROPOS(AR.StoreData.Fill) ) {
            if ( CopyFile(FG.fname, storefile) ) return(__LINE__);
        }
/*
        copy sort file if necessary
*/
        ANNOUNCE(copy sort file)
        if ( AR.outfile->handle >= 0 ) {
            if ( CopyFile(AR.outfile->name, sortfile) ) return(__LINE__);
        }
/*
        copy hide file if necessary
*/
        ANNOUNCE(copy hide file)
        if ( AR.hidefile->handle >= 0 ) {
            if ( CopyFile(AR.hidefile->name, hidefile) ) return(__LINE__);
        }
```

```
#ifdef WITHMPI
    }
    /*
     * For ParFORM, the renaming will be performed after the master got
     * all recovery files from the slaves.
     */
#else
/*
    make the intermediate file the recovery file
*/
    ANNOUNCE(rename intermediate file)
    if ( rename(intermedfile, recoveryfile) ) return(__LINE__);

    done_snapshot = 1;

    MesPrint("done."); fflush(0);
#endif

#ifdef PRINTDEBUG
    print_M();
    print_C();
    print_P();
    print_R();
#endif

    return(0);
}

/*
    #] DoSnapshot :
    #[ DoCheckpoint :
*/

void DoCheckpoint(int moduletype)
{
    int error;
    LONG timestamp = TimeWallClock(1);
#ifdef WITHMPI
    if(PF.me == MASTER){
#endif
    if ( timestamp - AC.CheckpointStamp >= AC.CheckpointInterval ) {
        char argbuf[20];
        int retvalue = 0;
        if ( AC.CheckpointRunBefore ) {
            size_t l, l2;
            char *str;
            l = strlen(AC.CheckpointRunBefore);
            NumToStr((UBYTE*)argbuf, AC.CModule);
            l2 = strlen(argbuf);
            str = (char*)Malloc1(l+l2+2, "callbefore");
            strcpy(str, AC.CheckpointRunBefore);
            *(str+l) = ' ';
            strcpy(str+l+1, argbuf);
            retvalue = system(str);
            M_free(str, "callbefore");
            if ( retvalue ) {
                MesPrint("Script returned error -> no recovery file will be created.");
            }
        }
#ifdef WITHMPI
        /* Confirm slaves to make snapshots. */
        PF_BroadcastNumber(retvalue == 0);
#endif
        if ( retvalue == 0 ) {
            if ( (error = DoSnapshot(moduletype)) ) {
                MesPrint("Error creating recovery files: %d", error);
            }
#ifdef WITHMPI
            {
            int i;
            /*get recovery files from slaves:*/
            for(i=1; i<PF.numtasks;i++){
                FILE *fd;
                const char *tmpnam = PF_recoveryfile('m', i, 1);
                fd = fopen(tmpnam, "w");
                if(fd == NULL){
                    MesPrint("Error opening recovery file for slave %d",i);
                    Terminate(-1);
                }/*if(fd == NULL)*/
                retvalue=PF_RecvFile(i,fd);
                if(retvalue<=0){
                    MesPrint("Error receiving recovery file from slave %d",i);
                    Terminate(-1);
                }/*if(retvalue<=0)*/
                fclose(fd);
            }/*for(i=0; i<PF.numtasks;i++)*/
            /*
```

```
             * Make the intermediate files the recovery files.
             */
            ANNOUNCE(rename intermediate file)
            for ( i = 0; i < PF.numtasks; i++ ) {
                const char *src = PF_recoveryfile('m', i, 1);
                const char *dst = PF_recoveryfile('m', i, 0);
                if ( rename(src, dst) ) {
                    MesPrint("Error renaming recovery file %s -> %s", src, dst);
                }
            }
            done_snapshot = 1;
            MesPrint("done."); fflush(0);
            }
#endif
        }
        if ( AC.CheckpointRunAfter ) {
            size_t l, l2;
            char *str;
            l = strlen(AC.CheckpointRunAfter);
            NumToStr((UBYTE*)argbuf, AC.CModule);
            l2 = strlen(argbuf);
            str = (char*)Malloc1(l+l2+2, "callafter");
            strcpy(str, AC.CheckpointRunAfter);
            *(str+l) = ' ';
            strcpy(str+l+1, argbuf);
            retvalue = system(str);
            M_free(str, "callafter");
            if ( retvalue ) {
                MesPrint("Error calling script after recovery.");
            }
        }
        AC.CheckpointStamp = TimeWallClock(1);
    }
#ifdef WITHMPI
    else{/* timestamp - AC.CheckpointStamp < AC.CheckpointInterval*/
        /* The slaves don't need to make snapshots. */
        PF_BroadcastNumber(0);
    }
    }/*if(PF.me == MASTER)*/
    else{/*Slave*/
        int i;
        /* Check if the slave needs to make a snapshot. */
        if ( PF_BroadcastNumber(0) ) {
            error = DoSnapshot(moduletype);
            if(error == 0){
                FILE *fd;
                /*
                 * Send the recovery file to the master. Note that no renaming
                 * has been performed and what we have to send is actually sitting
                 * in the intermediate file.
                 */
                fd = fopen(intermedfile, "r");
                i=PF_SendFile(MASTER, fd);/*if fd==NULL, PF_SendFile seds to a slave the failure tag*/
                if(fd == NULL)
                    Terminate(-1);
                fclose(fd);
                if(i<=0)
                    Terminate(-1);
                /*Now the slave need not the recovery file so remove it:*/
                remove(intermedfile);
            }
            else{
                /*send the error tag to the master:*/
                PF_SendFile(MASTER,NULL);/*if fd==NULL, PF_SendFile seds to a slave the failure tag*/
                Terminate(-1);
            }
            done_snapshot = 1;
        }/*if(tag=PF_DATA_MSGTAG)*/
    }/*if(PF.me != MASTER)*/
#endif
}

/*
    #] DoCheckpoint :
*/
```

4.7 comexpr.c File Reference

`#include "form3.h"`
Include dependency graph for comexpr.c:

Data Structures

- struct **id_options**

Functions

- int CoLocal (UBYTE ∗inp)
- int CoGlobal (UBYTE ∗inp)
- int CoLocalFactorized (UBYTE ∗inp)
- int CoGlobalFactorized (UBYTE ∗inp)
- int DoExpr (UBYTE ∗inp, int type, int par)
- int CoIdOld (UBYTE ∗inp)
- int CoId (UBYTE ∗inp)
- int CoIdNew (UBYTE ∗inp)
- int CoDisorder (UBYTE ∗inp)
- int CoMany (UBYTE ∗inp)
- int CoMulti (UBYTE ∗inp)
- int CoIfMatch (UBYTE ∗inp)
- int CoIfNoMatch (UBYTE ∗inp)
- int CoOnce (UBYTE ∗inp)
- int CoOnly (UBYTE ∗inp)
- int CoSelect (UBYTE ∗inp)
- int CoIdExpression (UBYTE ∗inp, int type)
- int CoMultiply (UBYTE ∗inp)
- int CoFill (UBYTE ∗inp)
- int CoFillExpression (UBYTE ∗inp)
- int CoPrintTable (UBYTE ∗inp)
- int CoAssign (UBYTE ∗inp)
- int CoDeallocateTable (UBYTE ∗inp)

4.7.1 Detailed Description

Compiler routines for statements that involve algebraic expressions. These involve definitions, id-statements, the multiply statement and the fill statement.

Definition in file comexpr.c.

4.7.2 Function Documentation

4.7.2.1 CoLocal()

```
int CoLocal (
            UBYTE * inp )
```

Definition at line 66 of file comexpr.c.

4.7.2.2 CoGlobal()

```
int CoGlobal (
            UBYTE * inp )
```

Definition at line 73 of file comexpr.c.

4.7.2.3 CoLocalFactorized()

```
int CoLocalFactorized (
            UBYTE * inp )
```

Definition at line 80 of file comexpr.c.

4.7.2.4 CoGlobalFactorized()

```
int CoGlobalFactorized (
            UBYTE * inp )
```

Definition at line 87 of file comexpr.c.

4.7.2.5 DoExpr()

```
int DoExpr (
            UBYTE * inp,
            int type,
            int par )
```

Definition at line 96 of file comexpr.c.

4.7.2.6 CoIdOld()

```
int CoIdOld (
            UBYTE * inp )
```

Definition at line 380 of file comexpr.c.

4.7.2.7 CoId()

```
int CoId (
            UBYTE * inp )
```

Definition at line 391 of file comexpr.c.

4.7.2.8 CoIdNew()

```
int CoIdNew (
            UBYTE * inp )
```

Definition at line 402 of file comexpr.c.

4.7.2.9 CoDisorder()

```
int CoDisorder (
            UBYTE * inp )
```

Definition at line 413 of file comexpr.c.

4.7.2.10 CoMany()

```
int CoMany (
            UBYTE * inp )
```

Definition at line 424 of file comexpr.c.

4.7.2.11 CoMulti()

```
int CoMulti (
            UBYTE * inp )
```

Definition at line 435 of file comexpr.c.

4.7.2.12 CoIfMatch()

```
int CoIfMatch (
            UBYTE * inp )
```

Definition at line 446 of file comexpr.c.

4.7.2.13 CoIfNoMatch()

```
int CoIfNoMatch (
            UBYTE * inp )
```

Definition at line 457 of file comexpr.c.

4.7.2.14 CoOnce()

```
int CoOnce (
            UBYTE * inp )
```

Definition at line 468 of file comexpr.c.

4.7.2.15 CoOnly()

```
int CoOnly (
            UBYTE * inp )
```

Definition at line 479 of file comexpr.c.

4.7.2.16 CoSelect()

```
int CoSelect (
            UBYTE * inp )
```

Definition at line 490 of file comexpr.c.

4.7.2.17 CoIdExpression()

```
int CoIdExpression (
            UBYTE * inp,
            int type )
```

Definition at line 503 of file comexpr.c.

4.7.2.18 CoMultiply()

```
int CoMultiply (
            UBYTE * inp )
```

Definition at line 1024 of file comexpr.c.

4.7.2.19 CoFill()

```
int CoFill (
            UBYTE * inp )
```

Definition at line 1063 of file comexpr.c.

4.7.2.20 CoFillExpression()

```
int CoFillExpression (
            UBYTE * inp )
```

Definition at line 1339 of file comexpr.c.

4.7.2.21 CoPrintTable()

```
int CoPrintTable (
            UBYTE * inp )
```

Definition at line 1725 of file comexpr.c.

4.7.2.22 CoAssign()

```
int CoAssign (
            UBYTE * inp )
```

Definition at line 1901 of file comexpr.c.

4.7.2.23 CoDeallocateTable()

```
int CoDeallocateTable (
            UBYTE * inp )
```

Definition at line 1959 of file comexpr.c.

4.8 comexpr.c

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034
00035 /*
00036     #[ Includes : compi2.c
00037
00038     File contains most of what has to do with compiling expressions.
00039     Main supporting file: token.c
00040 */
00041
00042 #include "form3.h"
00043
00044 static struct id_options {
00045     UBYTE *name;
00046     int code;
00047     int dummy;
00048 } IdOptions[] = {
```

```
 {(UBYTE *)"multi",       SUBMULTI    ,0}
,{(UBYTE *)"many",        SUBMANY     ,0}
,{(UBYTE *)"only",        SUBONLY     ,0}
,{(UBYTE *)"once",        SUBONCE     ,0}
,{(UBYTE *)"ifmatch",     SUBAFTER    ,0}
,{(UBYTE *)"ifnomatch",   SUBAFTERNOT ,0}
,{(UBYTE *)"ifnotmatch",  SUBAFTERNOT ,0}
,{(UBYTE *)"disorder",    SUBDISORDER ,0}
,{(UBYTE *)"select",      SUBSELECT   ,0}
,{(UBYTE *)"all",         SUBALL      ,0}
};

/*
    #] Includes :
    #[ CoLocal :
*/

int CoLocal(UBYTE *inp) { return(DoExpr(inp,LOCALEXPRESSION,0)); }

/*
    #] CoLocal :
    #[ CoGlobal :
*/

int CoGlobal(UBYTE *inp) { return(DoExpr(inp,GLOBALEXPRESSION,0)); }

/*
    #] CoGlobal :
    #[ CoLocalFactorized :
*/

int CoLocalFactorized(UBYTE *inp) { return(DoExpr(inp,LOCALEXPRESSION,1)); }

/*
    #] CoLocalFactorized :
    #[ CoGlobalFactorized :
*/

int CoGlobalFactorized(UBYTE *inp) { return(DoExpr(inp,GLOBALEXPRESSION,1)); }

/*
    #] CoGlobalFactorized :
    #[ DoExpr:


*/

int DoExpr(UBYTE *inp, int type, int par)
{
    GETIDENTITY
    int error = 0;
    UBYTE *p, *q, c;
    WORD *w, i, j = 0, c1, c2, *OldWork = AT.WorkPointer, osize;
    WORD jold = 0;
    POSITION pos;
    while ( *inp == ',' ) inp++;
    if ( par ) AC.ToBeInFactors = 1;
    else       AC.ToBeInFactors = 0;
    p = inp;
    while ( *p && *p != '=' ) {
        if ( *p == '(' ) SKIPBRA4(p)
        else if ( *p == '{' ) SKIPBRA5(p)
        else if ( *p == '[' ) SKIPBRA1(p)
        else p++;
    }
    if ( *p ) {     /* Variety with the = sign */
        if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
            MesPrint("&Illegal name for expression");
            error = 1;
            if ( q[-1] == '_' ) {
                while ( FG.cTable[*q] < 2 || *q == '_' ) q++;
            }
        }
        else {
            c = *q; *q = 0;
            if ( GetVar(inp,&c1,&c2,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
                if ( c1 == CEXPRESSION ) {
                    if ( Expressions[c2].status == STOREDEXPRESSION ) {
                        MesPrint("&Illegal attempt to overwrite a stored expression");
                        error = 1;
                    }
                    else {
                        HighWarning("Expression is replaced by new definition");
                        if ( AO.OptimizeResult.nameofexpr != NULL &&
                            StrCmp(inp,AO.OptimizeResult.nameofexpr) == 0 ) {
                            ClearOptimize();
                        }
```

```
                        if ( Expressions[c2].status != DROPPEDEXPRESSION ) {
                            w = &(Expressions[c2].status);
                            if ( *w == LOCALEXPRESSION || *w == SKIPLEXPRESSION )
                                *w = DROPLEXPRESSION;
                            else if ( *w == GLOBALEXPRESSION || *w == SKIPGEXPRESSION )
                                *w = DROPGEXPRESSION;
                            else if ( *w == HIDDENLEXPRESSION )
                                *w = DROPHLEXPRESSION;
                            else if ( *w == HIDDENGEXPRESSION )
                                *w = DROPHGEXPRESSION;
                        }
                        AC.TransEname = Expressions[c2].name;
                        j = EntVar(CEXPRESSION,0,type,0,0,0);
                        Expressions[j].node = Expressions[c2].node;
                        Expressions[c2].replace = j;
                    }
                }
                else {
                    MesPrint("&name of expression is also name of a variable");
                    error = 1;
                    j = EntVar(CEXPRESSION,inp,type,0,0,0);
                }
                jold = c2;
            }
            else {
/*
                Here we have to worry about reuse of the expression in the
                same module. That will need AS.Oldvflags but that may not
                be defined or have the proper value.
*/
                j = EntVar(CEXPRESSION,inp,type,0,0,0);
                jold = j;
            }
            *q = c;
            OldWork = w = AT.WorkPointer;
            *w++ = TYPEEXPRESSION;
            *w++ = 3+SUBEXPSIZE;
            *w++ = j;
            AC.ProtoType = w;
            AR.CurExpr = j;             /* Block expression j */
            *w++ = SUBEXPRESSION;
            *w++ = SUBEXPSIZE;
            *w++ = j;
            *w++ = 1;
            *w++ = AC.cbufnum;
            FILLSUB(w)

            if ( c == '(' ) {
                while ( *q == ',' || *q == '(' ) {
                    inp = q+1;
                    if ( ( q = SkipAName(inp) ) == 0 ) {
                        MesPrint("&Illegal name for expression argument");
                        error = 1;
                        q = p - 1;
                        break;
                    }
                    c = *q; *q = 0;
                    if ( GetVar(inp,&c1,&c2,ALLVARIABLES,WITHAUTO) < 0 ) c1 = -1;
                    switch ( c1 ) {
                        case CSYMBOL :
                            *w++ = SYMTOSYM; *w++ = 4; *w++ = c2; *w++ = 0;
                            break;
                        case CINDEX :
                            *w++ = INDTOIND; *w++ = 4;
                            *w++ = c2 + AM.OffsetIndex; *w++ = 0;
                            break;
                        case CVECTOR :
                            *w++ = VECTOVEC; *w++ = 4;
                            *w++ = c2 + AM.OffsetVector; *w++ = 0;
                            break;
                        case CFUNCTION :
                            *w++ = FUNTOFUN; *w++ = 4; *w++ = c2 + FUNCTION; *w++ = 0;
                            break;
                        default :
                            MesPrint("&Illegal expression parameter: %s",inp);
                            error = 1;
                            break;
                    }
                    *q = c;
                }
                if ( *q != ')' || q+1 != p ) {
                    MesPrint("&Illegal use of arguments for expression");
                    error = 1;
                }
                AC.ProtoType[1] = w - AC.ProtoType;
            }
            else if ( c != '=' ) {
```

```
/*
                The dummy accepted L F := RHS;
*/
                MesPrint("&Illegal LHS for expression definition");
                error = 1;
            }
            *w++ = 1;
            *w++ = 1;
            *w++ = 3;
            *w++ = 0;
            SeekScratch(AR.outfile,&pos);
            Expressions[j].counter = 1;
            Expressions[j].onfile = pos;
            Expressions[j].whichbuffer = 0;
#ifdef PARALLELCODE
            Expressions[j].partodo = AC.inparallelflag;
#endif
            OldWork[2] = w - OldWork - 3;
            AT.WorkPointer = w;
/*
            Writing the expression prototype to disk and to the compiler
            buffer is done only after the RHS has been compiled because
            we don't know the number of the main level RHS yet.
*/
        }
        inp = p+1;
        ClearWildcardNames();
        osize = AC.ProtoType[1]; AC.ProtoType[1] = SUBEXPSIZE;
        PutInVflags(jold);
        if ( ( i = CompileAlgebra(inp,RHSIDE,AC.ProtoType) ) < 0 ) {
            AC.ProtoType[1] = osize;
            error = 1;
        }
        else if ( error == 0 ) {
            AC.ProtoType[1] = osize;
            AC.ProtoType[2] = i;
            if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0 ) {
                MesPrint("&Cannot create expression");
                error = -1;
            }
            else {
                Expressions[j].sizeprototype = OldWork[2];
                OldWork[2] = 4+SUBEXPSIZE;
                OldWork[4] = SUBEXPSIZE;
                OldWork[5] = i;
                OldWork[SUBEXPSIZE+3] = 1;
                OldWork[SUBEXPSIZE+4] = 1;
                OldWork[SUBEXPSIZE+5] = 3;
                OldWork[SUBEXPSIZE+6] = 0;
                if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0
                || FlushOut(&pos,AR.outfile,0) ) {
                    MesPrint("&Cannot create expression");
                    error = -1;
                }
                AR.outfile->POfull = AR.outfile->POfill;
            }
            OldWork[2] = j;
            AddNtoL(OldWork[1],OldWork);
            AT.WorkPointer = OldWork;
            if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
        }
        AC.ToBeInFactors = 0;
    }
    else {  /* Variety in which expressions change property */
/*
            This code got a major revision because it didn't
            take hidden expressions into account. (1-jun-2010 JV)
*/
        do {
            if ( ( q = SkipAName(inp) ) == 0 ) {
                MesPrint("&Illegal name(s) for expression(s)");
                return(1);
            }
            c = *q; *q = 0;
            if ( GetName(AC.exprnames,inp,&c2,NOAUTO) == NAMENOTFOUND ) {
                MesPrint("&%s is not a valid expression",inp);
                error = 1;
            }
            else {
                w = &(Expressions[c2].status);
                if ( type == LOCALEXPRESSION ) {
                    switch ( *w ) {
                        case GLOBALEXPRESSION:
                            *w = LOCALEXPRESSION;
                            break;
                        case SKIPGEXPRESSION:
                            *w = SKIPLEXPRESSION;
```

```
                            break;
                        case DROPGEXPRESSION:
                            *w = DROPLEXPRESSION;
                            break;
                        case HIDDENGEXPRESSION:
                            *w = HIDDENLEXPRESSION;
                            break;
                        case HIDEGEXPRESSION:
                            *w = HIDELEXPRESSION;
                            break;
                        case UNHIDEGEXPRESSION:
                            *w = UNHIDELEXPRESSION;
                            break;
                        case INTOHIDEGEXPRESSION:
                            *w = INTOHIDELEXPRESSION;
                            break;
                        case DROPHGEXPRESSION:
                            *w = DROPHLEXPRESSION;
                            break;
                    }
                }
                else if ( type == GLOBALEXPRESSION ) {
                    switch ( *w ) {
                        case LOCALEXPRESSION:
                            *w = GLOBALEXPRESSION;
                            break;
                        case SKIPLEXPRESSION:
                            *w = SKIPGEXPRESSION;
                            break;
                        case DROPLEXPRESSION:
                            *w = DROPGEXPRESSION;
                            break;
                        case HIDDENLEXPRESSION:
                            *w = HIDDENGEXPRESSION;
                            break;
                        case HIDELEXPRESSION:
                            *w = HIDEGEXPRESSION;
                            break;
                        case UNHIDELEXPRESSION:
                            *w = UNHIDEGEXPRESSION;
                            break;
                        case INTOHIDELEXPRESSION:
                            *w = INTOHIDEGEXPRESSION;
                            break;
                        case DROPHLEXPRESSION:
                            *w = DROPHGEXPRESSION;
                            break;
                    }
                }
/*
                old code
                if ( type != LOCALEXPRESSION || *w != STOREDEXPRESSION )
                        *w = type;
*/
            }
            *q = c; inp = q+1;
        } while ( c == ',' );
        if ( c ) {
            MesPrint("&Illegal object in local or global redefinition");
            error = 1;
        }
    }
    return(error);
}

/*
    #] DoExpr:
    #[ CoIdOld :
*/

int CoIdOld(UBYTE *inp)
{
    AC.idoption = 0;
    return(CoIdExpression(inp,TYPEIDOLD));
}

/*
    #] CoIdOld :
    #[ CoId :
*/

int CoId(UBYTE *inp)
{
    AC.idoption = 0;
    return(CoIdExpression(inp,TYPEIDNEW));
}
```

```
/*
    #] CoId :
    #[ CoIdNew :
*/

int CoIdNew(UBYTE *inp)
{
    AC.idoption = 0;
    return(CoIdExpression(inp,TYPEIDNEW));
}

/*
    #] CoIdNew :
    #[ CoDisorder :
*/

int CoDisorder(UBYTE *inp)
{
    AC.idoption = SUBDISORDER;
    return(CoIdExpression(inp,TYPEIDNEW));
}

/*
    #] CoDisorder :
    #[ CoMany :
*/

int CoMany(UBYTE *inp)
{
    AC.idoption = SUBMANY;
    return(CoIdExpression(inp,TYPEIDNEW));
}

/*
    #] CoMany :
    #[ CoMulti :
*/

int CoMulti(UBYTE *inp)
{
    AC.idoption = SUBMULTI;
    return(CoIdExpression(inp,TYPEIDNEW));
}

/*
    #] CoMulti :
    #[ CoIfMatch :
*/

int CoIfMatch(UBYTE *inp)
{
    AC.idoption = SUBAFTER;
    return(CoIdExpression(inp,TYPEIDNEW));
}

/*
    #] CoIfMatch :
    #[ CoIfNoMatch :
*/

int CoIfNoMatch(UBYTE *inp)
{
    AC.idoption = SUBAFTERNOT;
    return(CoIdExpression(inp,TYPEIDNEW));
}

/*
    #] CoIfNoMatch :
    #[ CoOnce :
*/

int CoOnce(UBYTE *inp)
{
    AC.idoption = SUBONCE;
    return(CoIdExpression(inp,TYPEIDNEW));
}

/*
    #] CoOnce :
    #[ CoOnly :
*/

int CoOnly(UBYTE *inp)
{
    AC.idoption = SUBONLY;
    return(CoIdExpression(inp,TYPEIDNEW));
}
```

```

/*
    #] CoOnly :
    #[ CoSelect :
*/

int CoSelect(UBYTE *inp)
{
    AC.idoption = SUBSELECT;
    return(CoIdExpression(inp,TYPEIDNEW));
}

/*
    #] CoSelect :
    #[ CoIdExpression :

    First finish dealing with secondary keywords
*/

int CoIdExpression(UBYTE *inp, int type)
{
    GETIDENTITY
    int i, j, idhead, error = 0, MinusSign = 0, opt, retcode;
    WORD *w, *s, *m, *mm, *ww, *FirstWork, *OldWork, c1, numsets = 0,
         oldnumrhs, *ow, oldEside;
    UBYTE *p, *pp, c;
    CBUF *C = cbuf+AC.cbufnum;
    LONG oldcpointer, x;
    FirstWork = OldWork = AT.WorkPointer;
/*
    Don't forget to change in StudyPattern if we change/add_to the
    following setup.
    if ( type == TYPEIF ) idhead = IDHEAD-1;
    else
*/
    idhead = IDHEAD;
    AR.CurExpr = -1;
    w = AT.WorkPointer;
    *w++ = type;
    *w++ = idhead + SUBEXPSIZE;
    w++;
    if ( idhead >= IDHEAD ) *w++ = -1;
#if IDHEAD > 4
    for ( i = 4; i < idhead; i++ ) *w++ = 0;
#endif
    while ( *inp == ',' ) inp++;
    p = inp;
    if ( AC.idoption == SUBSELECT ) {
        p--;
        goto findsets;
    }
    else if ( ( AC.idoption == SUBAFTER ) || ( AC.idoption == SUBAFTERNOT ) ) {
        while ( *p && *p != '=' && *p != ',' ) {
            if ( *p == '(' ) SKIPBRA4(p)
            else if ( *p == '{' ) SKIPBRA5(p)
            else if ( *p == '[' ) SKIPBRA1(p)
            else p++;
        }
        if ( *p == '=' || *inp != '-' || inp[1] != '>' ) {
            MesPrint("&Illegal use if if[no]match in id statement");
            error = 1; goto AllDone;
        }
        if ( *p == 0 ) {
            MesPrint("&id-statement without = sign");
            error = 1; goto AllDone;
        }
        inp += 2; pp = inp;
        goto readlabel;
    }
    for(;;) {
        while ( *p && *p != '=' && *p != ',' ) {
            if ( *p == '(' ) SKIPBRA4(p)
            else if ( *p == '{' ) SKIPBRA5(p)
            else if ( *p == '[' ) SKIPBRA1(p)
            else p++;
        }
        if ( *p == '=' ) break;
        if ( *p == 0 ) {
            MesPrint("&id-statement without = sign");
            error = 1; goto AllDone;
        }
/*
        We have either a secondary option or a syntax error
*/
        pp = inp;
        while ( FG.cTable[*pp] == 0 ) pp++;
        c = *pp; *pp = 0;
```

```
        i = sizeof(IdOptions)/sizeof(struct id_options);
        while ( --i >= 0 ) {
            if ( StrICmp(inp,IdOptions[i].name) == 0 ) break;
        }
        if ( i < 0 ) {
            MesPrint("&Illegal option %s in id-statement",inp);
            *pp = c; error = 1; p++; inp = p; continue;
        }
        opt = IdOptions[i].code;
        *pp = c;
        inp = pp+1;
        switch ( opt ) {
            case SUBDISORDER:
                if ( pp != p ) goto IllField;
                AC.idoption |= SUBDISORDER;
                p++; inp = p;
                break;
            case SUBSELECT:
                if ( p != pp ) goto IllField;
                if ( ( AC.idoption & SUBMASK ) != 0 ) {
                    if ( AC.idoption == SUBMULTI && type == TYPEIF ) {}
                    else {
                        MesPrint("&Conflicting options in id-statement");
                        error = 1;
                    }
                }
findsets:;
/*
                Now we read the sets
*/
                numsets = 0;
                for(;;) {
                    inp = ++p;
                    while ( *p && *p != '=' && *p != ',' ) {
                        if ( *p == '(' ) SKIPBRA4(p)
                        else if ( *p == '{' ) SKIPBRA5(p)
                        else if ( *p == '[' ) SKIPBRA1(p)
                        else p++;
                    }
                    if ( *p == '=' ) break;
                    if ( *p == 0 ) {
                        MesPrint("&id-statement without = sign");
                        error = 1; goto AllDone;
                    }
/*
                    We have a set at inp.
*/
                    if ( *inp == '{' ) {
                        if ( p[-1] != '}' ) {
                            c = *p; *p = 0;
                            MesPrint("&Illegal temporary set: %s",inp);
                            error = 1; *p = c;
                        }
                        else {
                            inp++;
                            c = p[-1]; p[-1] = 0;
                            c1 = DoTempSet(inp,p-1);
                            *w++ = c1;
                            p[-1] = c;
                            numsets++;
                            if ( w[-1] < 0 ) error = 1;
                        }
                    }
                    else {
                        c = *p; *p = 0;
                        if ( GetName(AC.varnames,inp,&c1,NOAUTO) != CSET ) {
                            MesPrint("&%s is not a set",inp);
                            error = 1;
                        }
                        else {
                            if ( c1 < AM.NumFixedSets ) {
                                MesPrint("&Built in sets are not allowed in the select option");
                                error = 1;
                            }
                            else if ( Sets[c1].type == CRANGE ) {
                                MesPrint("&Ranged sets are not allowed in the select option");
                                error = 1;
                            }
                            numsets++;
                            *w++ = c1;
                        }
                        *p = c;
                    }
                }
/*
                Now exchange the positions a bit.
                Regular stuff at OldWork, numsets sets at FirstWork[idhead]
```

```
*/
                OldWork = w;
                for ( i = 0; i < idhead; i++ ) *w++ = FirstWork[i];
                AC.idoption = SUBSELECT;
                break;
            case SUBAFTER:
            case SUBAFTERNOT:
                if ( type == TYPEIF ) {
                    MesPrint("&The if[no]match->label option is not allowed in an if statement");
                    error = 1; goto AllDone;
                }
                if ( pp[0] != '-' || pp[1] != '>' ) goto IllField;
                pp += 2;    /* points now at the label */
                inp = pp;
                AC.idoption |= opt;
readlabel:
                while ( FG.cTable[*pp] <= 1 ) pp++;
                if ( pp != p ) {
                    c = *p; *p = 0;
                    MesPrint("&Illegal label %s in if[no]match option of id-statement",inp);
                    *p = c; error = 1; inp = p+1; continue;
                }
                c = *p; *p = 0;
                OldWork[3] = GetLabel(inp);
                *p++ = c; inp = p;
                break;
            case SUBALL:
                x = 0;
                if ( *pp == '(' ) {
                    if ( FG.cTable[*inp] == 1 ) {
                        while ( *inp >= '0' && *inp <= '9' ) x = 10*x+*inp++ - '0';
                    }
                    else {
                        pp++;
                        while ( FG.cTable[*inp] == 0 ) inp++;
                        c = *inp; *inp = 0;
                        if ( StrICont(pp,(UBYTE *)"normalize") != 0 ) goto IllOpt;
                        *inp = c;
                        OldWork[4] |= NORMALIZEFLAG;
                    }
                    if ( *inp != ')' || inp+1 != p ) {
                        c = *inp; *inp = 0;
IllOpt:
                        MesPrint("&Illegal ALL option in id-statement: ",pp);
                        *inp++ = c;
                        error = 1;
                        continue;
                    }
                    pp = inp;
                    inp = pp+1;
                }
/*
                Note that the following statement limits x to
*/
                if ( x > MAXPOSITIVE ) {
                    MesPrint("&Requested maximum number of matches %l in ALL option in id-statement is
    greater than %l ",x,MAXPOSITIVE);
                    error = 1;
                }
                OldWork[5] = x;
                if ( type != TYPEIDNEW ) {
                  if ( type == TYPEIDOLD ) {
                    MesPrint("&Requested ALL option not allowed in idold/also statement.");
                    error = 1;
                  }
                  else if ( type == TYPEIF ) {
                    MesPrint("&Requested ALL option not allowed in if(match())");
                    error = 1;
                  }
                  else {
                    MesPrint("&ALL option only allowed in regular id-statement.");
                    error = 1;
                  }
                }
                p++; inp = p;
                AC.idoption = opt;
                break;
            default:
                if ( pp != p ) {
IllField:           c = *p; *p = 0;
                    MesPrint("&Illegal optionfield %s in id-statement",inp);
                    *p = c; error = 1; inp = p+1; continue;
                }
                i = AC.idoption & SUBMASK;
                if ( i && i != opt ) {
                    MesPrint("&Conflicting options in id-statement");
                    error = 1;  continue;
```

```
                }
                else AC.idoption |= opt;
                while ( *p == ',' ) p++;
                inp = p;
                break;
        }
    }
    if ( ( AC.idoption & SUBMASK ) == 0 ) AC.idoption |= SUBMULTI;
    OldWork[2] = AC.idoption;
/*
    Now we have a field till the = sign
    Now the subexpression prototype
*/
    AC.ProtoType = w;
    *w++ = SUBEXPRESSION;
    *w++ = SUBEXPSIZE;
    *w++ = C->numrhs+1;
    *w++ = 1;
    *w++ = AC.cbufnum;
    FILLSUB(w)
    AC.WildC = w;
    AC.NwildC = 0;
    AT.WorkPointer = s = w + 4*AM.MaxWildcards + 8;
/*
    Now read the LHS
*/
    ClearWildcardNames();
    oldcpointer = AddLHS(AC.cbufnum) - C->Buffer;

    *p = 0;
    oldnumrhs = C->numrhs;
    if ( ( retcode = CompileAlgebra(inp,LHSIDE,AC.ProtoType) ) < 0 ) { error = 1; }
    else AC.ProtoType[2] = retcode;
    *p = '='; inp = p+1;
    AT.WorkPointer = s;
    if ( AC.NwildC && SortWild(w,AC.NwildC) ) error = 1;

        /* Make the LHS pointers ready */

    OldWork[1] = AC.WildC-OldWork;
    OldWork[idhead+1] = OldWork[1] - idhead;
    w = AC.WildC;
    AT.WorkPointer = w;
    s = C->rhs[C->numrhs];
/*
    Now check whether wildcards get converted to dollars (for PARALLEL)
*/
    {
        WORD *tw, *twstop;
        tw = AC.ProtoType; twstop = tw + tw[1]; tw += SUBEXPSIZE;
        while ( tw < twstop ) {
            if ( *tw == LOADDOLLAR ) {
                AddPotModdollar(tw[2]);
            }
            tw += tw[1];
        }
    }
/*
    We have the expression in the compiler buffers.
    The main level is at lhs[numlhs]
    The partial lhs (including ProtoType) is in OldWork (in WorkSpace)
    We need to load the result at w after the prototype
    Because these sort routines don't use the WorkSpace
    there should not be a conflict
*/
    if ( !error && *s == 0 ) {
IllLeft:MesPrint("&Illegal LHS");
        AC.lhdollarflag = 0;
        return(1);
    }
    if ( !error && *(s+*s) != 0 ) {
        MesPrint("&LHS should be one term only");
        return(1);
    }
    if ( error == 0 ) {
        if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) {
            if ( !error ) error = 1;
            return(error);
        }
        AN.RepPoint = AT.RepCount + 1;
        ow = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
        mm = s; ww = ow; i = *mm;
        while ( --i >= 0 ) *ww++ = *mm++; AT.WorkPointer = ww;
        AC.lhdollarflag = 0; oldEside = AR.Eside; AR.Eside = LHSIDE;
        AR.Cnumlhs = C->numlhs;
        if ( Generator(BHEAD ow,C->numlhs) ) {
            AR.Eside = oldEside;
```

```
            LowerSortLevel(); LowerSortLevel(); goto IllLeft;
        }
        AR.Eside = oldEside;
        AT.WorkPointer = w;
        if ( EndSort(BHEAD w,0) < 0 ) { LowerSortLevel(); goto IllLeft; }
        if ( *w == 0 || *(w+*w) != 0 ) {
            MesPrint("&LHS must be one term");
            AC.lhdollarflag = 0;
            return(1);
        }
        LowerSortLevel();
        if ( AC.lhdollarflag ) MarkDirty(w,DIRTYFLAG);
    }
    AT.WorkPointer = w + *w;
    AC.DumNum = 0;
/*
    Everything is now after OldWork. We can pop the compilerbuffer.
    Next test for illegal things like a coefficient
    At this point we have:
    w = the term of the LHS
*/
    C->Pointer = C->Buffer + oldcpointer;
    C->numrhs = oldnumrhs;
    C->numlhs--;

    m = w + *w - 3;
    AC.vectorlikeLHS = 0;
    if ( !error ) {
      if ( m[2] != 3 || m[1] != 1 || *m != 1 ) {
        if ( *m == 1 && m[1] == 1 && m[2] == -3 ) {
            MinusSign = 1;
        }
        else {
            MesPrint("&Coefficient in LHS");
            error = 1;
            AC.DumNum = 0;
            *w -= ABS(m[2])-3;
        }
      }
      if ( *w == 7 && w[1] == INDEX && w[3] < 0 ) {
        if ( ( AC.idoption & SUBMASK ) != 0 &&  ( AC.idoption & SUBMASK ) !=
        SUBMULTI ) {
            MesPrint("&Illegal option for substitution of a vector");
            error = 1;
        }
        AC.DumNum = AM.IndDum;
        OldWork[2] = ( OldWork[2] - ( OldWork[2] & SUBMASK ) ) | SUBVECTOR;
        c1 = w[3];
            /* We overwrite the LHS */
        *w++ = INDTOIND;
        *w++ = 4;
        *w++ = AC.DumNum + WILDOFFSET;
        *w++ = 0;
        w[0] = 5;
        w[1] = VECTOR;
        w[2] = 4;
        w[3] = c1;
        w[4] = AC.DumNum + WILDOFFSET;
        OldWork[idhead+1] = w - OldWork - idhead;
        AC.vectorlikeLHS = 1;
      }
      else {
        AC.DumNum = 0;
        *w -= 3;
        i = OldWork[2] & SUBMASK;
        m = w + *w;
        if ( i == 0 || i == SUBMULTI ) {
            s = w+1;
            while ( s < m ) {
                if ( *s == SYMBOL ) {
                    j = s[1]/2; s += 2;
                    while ( --j >= 0 ) {
                        if ( ABS(s[1]) > 2*MAXPOWER ) {
                            OldWork[2] = ( OldWork[2] - i ) | SUBONCE;
                            break;
                        }
                        s += 2;
                    }
                    if ( j >= 0 ) break;
                }
                else if ( *s == DOTPRODUCT ) {
                    j = s[1]/3; s += 2;
                    while ( --j >= 0 ) {
                        if ( ABS(s[2]) > 2*MAXPOWER ) {
                            OldWork[2] = ( OldWork[2] - i ) | SUBONCE;
                            break;
                        }
```

```
                        else if ( s[1] >= -(2*WILDOFFSET) || s[0] >= -(2*WILDOFFSET) ) {
                            OldWork[2] = ( OldWork[2] - i ) | SUBMANY;
                            i = SUBMANY;
                        }
                        s += 3;
                    }
                    if ( j >= 0 ) break;
                }
                else {
                    OldWork[2] = ( OldWork[2] - i ) | SUBMANY;
                    break;
                }
            }
        }
        if ( ( OldWork[2] & SUBMASK ) == 0 ) OldWork[2] |= SUBMULTI;
      }
      if ( ( OldWork[2] & SUBMASK ) == SUBSELECT ) {
/*
        Paste the SETSET information after the pattern.
        Important note: We will still get function information for the
        smart patternmatching after it. To distinguish them we need to have
        that SETSET != m*n+1 in which m is the number of words per function
        and n the number of functions. Currently (29-may-1997) m = 4.
*/
        *m++ = SETSET;
        *m++ = numsets+2;
        s = FirstWork + idhead;
        while ( --numsets >= 0 ) *m++ = *s++;
      }
      else {
        m = w + *w;
      }
    }
/*
    We keep the whole thing in OldWork for the moment.
    We still have to add the number of the RHS expression.
    There is also some opportunity now to be smart about the pattern.
    This is needed for complicated wildcarding with symmetric functions.
    We do this in a special routine during compile time to make sure
    that we loose as little time as possible (during running) if there
    is no need to be smart.
*/
    *m++ = 0;
    OldWork[1] = m - OldWork;
    AC.ProtoType = OldWork+idhead;
    if ( !error ) {
        if ( StudyPattern(OldWork) ) error = 1;
    }
    AT.WorkPointer = OldWork + OldWork[1];
    if ( AC.lhdollarflag ) OldWork[4] |= DOLLARFLAG;
    AC.lhdollarflag = 0;
/*
    Test whether the id/idold configuration is fine.
*/
    if ( type == TYPEIDOLD ) {
        WORD ci = C->numlhs;
        while ( ci >= 1 ) {
            if ( C->lhs[ci][0] == TYPEIDNEW ) {
                if ( (C->lhs[ci][2] & SUBMASK) == SUBALL ) {
                    MesPrint("&Idold/also cannot follow an id,all statement.");
                    error = 1;
                }
                break;
            }
            else if ( C->lhs[ci][0] == TYPEDETCURDUM ) { ci--; continue; }
            else if ( C->lhs[ci][0] == TYPEIDOLD ) { ci--; continue; }
            else ci = 0;
        }
        if ( ci < 1 ) {
            MesPrint("&Idold/also should follow an id/idnew statement.");
            error = 1;
        }
    }
/*
    Now the right hand side.
*/
    if ( type != TYPEIF ) {
        if ( ( retcode = CompileAlgebra(inp,RHSIDE,AC.ProtoType) ) < 0 ) error = 1;
        else {
            AC.ProtoType[2] = retcode;
            AC.DumNum = 0;
            if ( MinusSign ) {  /* Flip the sign of the RHS */
                w = C->rhs[retcode];
                while ( *w ) { w += *w; w[-1] = -w[-1]; }
            }
            if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
        }
```

```
    }
/*
    Actual adding happens only now after numrhs insertion
*/
    if ( !error ) { AddNtoL(OldWork[1],OldWork); }
AllDone:
    AC.lhdollarflag = 0;
    AT.WorkPointer = FirstWork;
    return(error);
}

/*
    #] CoIdExpression :
    #[ CoMultiply :
*/

static WORD mularray[13] = { TYPEMULT, SUBEXPSIZE+3, 0, SUBEXPRESSION,
        SUBEXPSIZE, 0, 1, 0, 0, 0, 0, 0, 0 };

int CoMultiply(UBYTE *inp)
{
    UBYTE *p;
    int error = 0, RetCode;
    mularray[2] = 0;        /* right multiply is default */
    while ( *inp == ',' ) inp++;
/*  if ( inp[-1] == '-' || inp[-1] == '+' ) inp--; */
    p = SkipField(inp,0);
    if ( *p ) {
        *p = 0;
        if ( StrICont(inp,(UBYTE *)"left") == 0 )       mularray[2] = 1;
        else if ( StrICont(inp,(UBYTE *)"right") == 0 ) mularray[2] = 0;
        else {
            MesPrint("&Illegal option in multiply statement or ; forgotten.");
            return(1);
        }
        *p = ',';
        inp = p + 1;
    }
    ClearWildcardNames();
    while ( *inp == ',' ) inp++;
    AC.ProtoType = mularray+3;
    mularray[7] = AC.cbufnum;
    if ( ( RetCode = CompileAlgebra(inp,RHSIDE,AC.ProtoType) ) < 0 ) error = 1;
    else {
        mularray[5] = RetCode;
        AddNtoL(SUBEXPSIZE+3,mularray);
        if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
    }
    return(error);
}

/*
    #] CoMultiply :
    #[ CoFill :

    Special additions for tablebase-like tables added 12-aug-2002
*/

int CoFill(UBYTE *inp)
{
    GETIDENTITY
    WORD error = 0, x, funnum, type, *oldwp = AT.WorkPointer;
    int i, oldcbufnum = AC.cbufnum, nofill = 0, numover, redef = 0;
    WORD *w, *wold, *Tprototype;
    UBYTE *p = inp, c, *inp1;
    TABLES T = 0, oldT;
    LONG newreservation, sum = 0;
    UBYTE *p1, *p2, *p3, *p4, *fake = 0;
    int tablestub = 0;
    if ( AC.exprfillwarning == 1 ) AC.exprfillwarning = 0;
/*
    Read the name of the function and test that it is in the table.
*/
    p1 = inp;
    if ( ( p = SkipAName(inp) ) == 0 ) return(1);
    p2 = p;
    c = *p; *p = 0;
    if ( ( GetVar(inp,&type,&funnum,CFUNCTION,WITHAUTO) == NAMENOTFOUND )
    || ( T = functions[funnum].tabl ) == 0 || ( T->numind > 0 && c != '(' ) ) {
        MesPrint("&%s should be a table with argument(s)",inp);
        *p = c; return(1);
    }
    oldT = T;
    *p++ = c;
    if ( T->numind == 0 ) {
        if ( c == '(' ) {
            if ( *p != ')' ) {
```

```
                c = *p; *p = 0;
                MesPrint("&%s should be a table without arguments",inp);
                *p = c; return(1);
            }
            else { p++; }
        }
        else { p--; }
        sum = 0;
        p3 = p;
        goto andagain;
    }
    for ( sum = 0, i = 0, w = oldwp; i < T->numind; i++ ) {
        ParseSignedNumber(x,p);
        if ( FG.cTable[p[-1]] != 1 || ( *p != ',' && *p != ')' ) ) {
            MesPrint("&Table arguments in fill statement should be numbers");
            return(1);
        }
        if ( T->sparse ) *w++ = x;
        else if ( x < T->mm[i].mini || x > T->mm[i].maxi ) {
            MesPrint("&Value %d for argument %d of table out of bounds",x,i+1);
            error = 1; nofill = 1;
        }
        else sum += ( x - T->mm[i].mini ) * T->mm[i].size;
        if ( *p == ')' ) break;
        p++;
    }
    p3 = p;
    if ( *p != ')' || i < ( T->numind - 1 ) ) {
        MesPrint("&Incorrect number of table arguments in fill statement. Should be %d"
        ,T->numind);
        error = 1; nofill = 1;
    }
    AT.WorkPointer = w;
    if ( T->sparse == 0 ) sum *= TABLEEXTENSION;
andagain:;
    AC.cbufnum = T->bufnum;
    if ( T->sparse ) {
        i = FindTableTree(T,oldwp,1);
        if ( i >= 0 ) {
            sum = i + T->numind;
            if ( tablestub == 0 && ( ( T->sparse & 2 ) == 2 ) && ( T->mode != 0 )
            && ( AC.vetotablebasefill == 0 ) ) {
/*
                This redefinition does not need a new stub
*/
                functions[funnum].tabl = T = T->spare;
                tablestub = 1;
                goto andagain;
            }
            redef = 1;
            goto redef;
        }
        if ( T->totind >= T->reserved ) {
            if ( T->reserved == 0 ) newreservation = 20;
            else newreservation = T->reserved;
/*
            while ( T->totind >= newreservation && newreservation <
                    MAXTABLECOMBUF*(T->numind+TABLEEXTENSION) )
            if ( newreservation > MAXTABLECOMBUF*T->numind ) newreservation =
                    5*(T->numind+TABLEEXTENSION);
*/
            while ( T->totind >= newreservation && newreservation < MAXTABLECOMBUF )
                    newreservation = 2*newreservation;
            if ( newreservation > MAXTABLECOMBUF ) newreservation = MAXTABLECOMBUF;
            if ( T->totind >= newreservation ) {
                MesPrint("@More than %ld elements in sparse table",MAXTABLECOMBUF);
                AC.cbufnum = oldcbufnum;
                Terminate(-1);
            }
            wold = (WORD *)Malloc1(newreservation*sizeof(WORD)*
                                (T->numind+TABLEEXTENSION),"tablepointers");
            for ( i = T->reserved*(T->numind+TABLEEXTENSION)-1; i >= 0; i-- )
                wold[i] = T->tablepointers[i];
            if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
            T->tablepointers = wold;
            T->reserved = newreservation;
        }
        w = oldwp;
        for ( sum = T->totind*(T->numind+TABLEEXTENSION), i = 0; i < T->numind; i++ ) {
            T->tablepointers[sum++] = *w++;
        }
        InsTableTree(T,T->tablepointers+sum-T->numind);
#if TABLEEXTENSION == 2
        T->tablepointers[sum+TABLEEXTENSION-1] = -1;  /* New element! */
#else
        T->tablepointers[sum+1] = T->bufnum;
        T->tablepointers[sum+2] = -1;
```

```
        T->tablepointers[sum+3] = -1;
        T->tablepointers[sum+4] = 0;
        T->tablepointers[sum+5] = 0;
#endif
    }
    else {
        if ( !nofill && T->tablepointers[sum] >= 0 ) {
redef:;
            if ( AC.vetofilling ) nofill = 1;
            else {
                Warning("Table element was already defined. New definition will be used");
            }
        }
#if TABLEEXTENSION == 2
        T->tablepointers[sum+TABLEEXTENSION-1] = -1;  /* New element! */
#else
        T->tablepointers[sum+1] = T->bufnum;
        T->tablepointers[sum+2] = -1;
        T->tablepointers[sum+3] = -1;
        T->tablepointers[sum+4] = 0;
        T->tablepointers[sum+5] = 0;
#endif
    }
    if ( T->numind ) { p++; }
    if ( *p != '=' ) {
        MesPrint("&Fill statement misses = sign after the table element");
        AC.cbufnum = oldcbufnum;
        AT.WorkPointer = oldwp;
        functions[funnum].tabl = oldT;
        return(1);
    }
    if ( tablestub == 0 && T->mode == 1 && AC.vetotablebasefill == 0 ) {
/*
        Here we construct a righthandside from the indices and the wildcards
*/
        int numfake;
        tablestub = 1;
        p4 = T->argtail;
        while ( *p4 ) p4++;
        numfake = (p4-T->argtail)+(p3-p1)+10;

        fake = (UBYTE *)Malloc1(numfake*sizeof(UBYTE),"Fill fake rhs");
        p = fake;
        *p++ = 't'; *p++ = 'b'; *p++ = 'l'; *p++ = '_'; *p++ = '(';
        p4 = p1; while ( p4 < p2 ) *p++ = *p4++; *p++ = ',';
        p4 = p2+1; while ( p4 < p3 ) *p++ = *p4++;
        if ( T->argtail ) {
            p4 = T->argtail + 1;
            while ( FG.cTable[*p4] == 1 ) p4++;
            while ( *p4 ) {
                if ( *p4 == '?' && p[-1] != ',' ) {
                    p4++;
                    if ( FG.cTable[*p4] == 0 || *p4 == '$' || *p4 == '[' ) {
                        p4 = SkipAName(p4);
                        if ( *p4 == '[' ) {
                            SKIPBRA1(p4);
                        }
                    }
                    else if ( *p4 == '{' ) {
                        SKIPBRA2(p4);
                    }
                    else if ( *p4 ) { *p++ = *p4++; continue; }
                }
                else *p++ = *p4++;
            }
        }
        *p++ = ')';
        *p = 0;
        inp1 = fake;
/*      AT.WorkPointer += T->numind; */
    }
    else
    inp1 = ++p; c = 0;
/*
    Now we have the indices and p points to the rhs.
*/
    numover = 0;
    AC.tablefilling = funnum;
    while ( *inp1 ) {
        p = SkipField(inp1,0);
        c = *p; *p = 0;
#ifdef WITHPTHREADS
        Tprototype = T->prototype[0];
#else
        Tprototype = T->prototype;
#endif
        if ( ( i = CompileAlgebra(inp1,RHSIDE,Tprototype) ) < 0 ) { error = 1; i = 0; }
```

```
        if ( !nofill ) {
            T->tablepointers[sum] = i;
            T->tablepointers[sum+1] = T->bufnum;
        }
        AC.DumNum = 0;
        *p = c;
        if ( T->sparse || c == 0 ) break;
        inp1 = ++p;
#if ( TABLEEXTENSION == 2 )
        sum++;
#else
        sum += 2;
#endif
        if ( !nofill && T->tablepointers[sum] >= 0 ) numover++;
#if ( TABLEEXTENSION == 2 )
        sum++;
#else
        sum += TABLEEXTENSION-2;
#endif
    }
    if ( AC.exprfillwarning == 1 ) {
        AC.exprfillwarning = 2;
        Warning("Use of expressions and/or $variables in Fill statements is potentially very dangerous.");
    }
    AC.tablefilling = 0;
    if ( T->sparse && c != 0 ) {
        MesPrint("&In sparse tables one can fill only one element at a time");
        error = 1;
    }
    else if ( numover ) {
        if ( numover == 1 )
            Warning("one element was overwritten. New definition will be used");
        else if ( AC.WarnFlag )
            MesPrint("&Warning: %d elements were overwritten. New definitions will be used",numover);
    }
    if ( T->sparse ) {
        if ( redef == 0 ) T->totind++;
    }
    else T->defined++;
/*
    NumSets = AC.SetList.numtemp;
    NumSetElements = AC.SetElementList.numtemp;
*/
    if ( fake ) {
        M_free(fake,"Fill fake rhs");
        fake = 0;
        functions[funnum].tabl = T = T->spare;
        p = p3;
        goto andagain;
    }
    AC.cbufnum = oldcbufnum;
    AC.SymChangeFlag = 1;
    AT.WorkPointer = oldwp;
    functions[funnum].tabl = oldT;
    return(error);
}

/*
    #] CoFill :
    #[ CoFillExpression :

    Syntax: FillExpression table = expression(x1,...,xn);
    The arguments should have been bracketed. Each corresponds to one
    of the dimensions of the table. Then the bracket with x1^2*x3^4
    will fill the (2,0,4) element of the table (if n=3 of course).
    Brackets that don't fit will be skipped. It just gives a warning.

    New option (13-jul-2005)
    Syntax: FillExpression table = expression(f);
    The table indices are arguments of the function f which should
    have been bracketed before.
*/

int CoFillExpression(UBYTE *inp)
{
    GETIDENTITY
    UBYTE *p, c;
    WORD type, funnum, expnum, symnum, numsym = 0, *oldwork = AT.WorkPointer;
    WORD *brackets, *term, brasize, *b, *m, *w, *pw, *tstop, zero = 0;
    WORD oldcbuf = AC.cbufnum, curelement = 0;
    int weneedit, i, j, numzero, pow;
    TABLES T = 0;
    LONG newreservation, numcommu, sum;
    POSITION oldposition;
    FILEHANDLE *fi;
    CBUF *C;
```

```
    WORD numdummies;

    AN.IndDum = AM.IndDum;
    if ( ( p = SkipAName(inp) ) == 0 ) return(1);
    c = *p; *p = 0;
    if ( ( GetVar(inp,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
    || ( T = functions[funnum].tabl ) == 0 ) {
        MesPrint("&%s should be a previously declared table",inp);
        *p = c; return(1);
    }
    *p++ = c;
    if ( T->spare ) T = T->spare;
    C = cbuf + T->bufnum;
    if ( c != '=' ) {
        MesPrint("&No = sign in FillExpression statement");
        return(1);
    }
    inp = p;
    if ( ( p = SkipAName(inp) ) == 0 ) return(1);
    c = *p; *p = 0;
    if ( ( type = GetName(AC.exprnames,inp,&expnum,NOAUTO) ) == NAMENOTFOUND
    || c != '(' || (
        Expressions[expnum].status != LOCALEXPRESSION &&
        Expressions[expnum].status != SKIPLEXPRESSION &&
        Expressions[expnum].status != DROPLEXPRESSION &&
        Expressions[expnum].status != GLOBALEXPRESSION &&
        Expressions[expnum].status != SKIPGEXPRESSION &&
        Expressions[expnum].status != DROPGEXPRESSION ) ) {
        MesPrint("&%s should be an active expression with arguments",inp);
        *p = c; return(1);
    }
    if ( Expressions[expnum].inmem ) {
        MesPrint("&%s cannot be used in a FillExpression statement in the same %n\
        module that it has been redefined",inp);
        *p = c; return(1);
    }
    *p++ = c;
    while ( *p ) {
        inp = p;
        if ( ( p = SkipAName(inp) ) == 0 ) return(1);
        c = *p; *p = 0;

        if ( GetVar(inp,&type,&symnum,-1,NOAUTO) == NAMENOTFOUND ) {
            MesPrint("&%s should be a previously declared symbol or function",inp);
            *p = c; return(1);
        }
        else if ( type == CSYMBOL ) {
            *p++ = c;
            *AT.WorkPointer++ = symnum;
            numsym++;
        }
        else if ( type == CFUNCTION ) {
            numsym = -1;
            *p++ = c;
            if ( c != ')' ) {
                MesPrint("&Argument should be a single function or a list of symbols");
                return(1);
            }
            symnum += FUNCTION;
            *AT.WorkPointer++ = symnum;
        }
        else {
            MesPrint("&%s should be a previously declared symbol or function",inp);
            *p = c; return(1);
        }
/*
        if ( GetVar(inp,&type,&symnum,CSYMBOL,NOAUTO) == NAMENOTFOUND ) {
            if ( numsym > 0 ) {
                MesPrint("&%s should be a previously declared symbol",inp);
                *p = c; return(1);
            }
            else {
                if ( GetVar(inp,&type,&symnum,CFUNCTION,NOAUTO) == NAMENOTFOUND ) {
                    MesPrint("&%s should be a previously declared symbol or function",inp);
                    *p = c; return(1);
                }
                numsym = -1;
                *p++ = c;
                if ( c != ')' ) {
                    MesPrint("&Argument should be a single function or a list of symbols");
                    *p = c; return(1);
                }
                symnum += FUNCTION;
                *AT.WorkPointer++ = symnum;
                break;
            }
        }
```

```
        *p++ = c;
        *AT.WorkPointer++ = symnum;
        numsym++;
*/
        if ( c == ')' ) break;
        if ( c != ',' ) {
            MesPrint("&Illegal separator in FillExpression statement");
            goto noway;
        }
    }
    if ( *p ) {
        MesPrint("&Illegal end of FillExpression statement");
        goto noway;
    }
/*
    We have the number of the table in funnum.
    The number of the expression in expnum, the table struct in T
    and either the numbers of the symbols in oldwork (there are numsym of them)
    or the number of the function in oldwork (just one and numsym = -1).
    We don't sort them!!!!
*/
    if ( ( numsym > 0 ) && ( T->numind != numsym ) ) {
        MesPrint("&This table needs %d symbols for its array indices");
        goto noway;
    }
    EXCHINOUT
#ifdef WITHMPI
    /*
     * The workers can't access to the data of the input expression. We need to
     * broadcast it to all the workers.
     */
    PF_BroadcastExpr(&Expressions[expnum], AR.infile);
    if ( PF.me == MASTER ) {
        /*
         * Restore the file position on the master.
         */
        POSITION pos;
        SetEndScratch(AR.infile, &pos);
    }
#endif
    fi = AR.infile;
    if ( fi->handle >= 0 ) {
        PUTZERO(oldposition);
        SeekFile(fi->handle,&oldposition,SEEK_CUR);
        SetScratch(fi,&(Expressions[expnum].onfile));
/*      SeekFile(fi->handle,&(Expressions[expnum].onfile),SEEK_SET); */
        if ( ISNEGPOS(Expressions[expnum].onfile) ) {
            MesPrint("&File error in FillExpression");
            BACKINOUT
            goto noway;
        }
    }
    else {
/*
        Note: Because everything fits inside memory we never get problems
        with excessive file sizes.
*/
        SETBASEPOSITION(oldposition,(UBYTE *)(fi->POfill)-(UBYTE *)(fi->PObuffer));
        fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + BASEPOSITION(Expressions[expnum].onfile));
    }
    pw = AT.WorkPointer;
    if ( numsym < 0 ) { brackets = pw + 1; }
    else { brackets = pw + numsym; }
    brasize = -1; weneedit = 0; /* stands for we need it */
    term = (WORD *)(((UBYTE *)(brackets)) + AM.MaxTer);
    AT.WorkPointer = (WORD *)(((UBYTE *)(term)) + AM.MaxTer);
    AC.cbufnum = T->bufnum;
    AC.tablefilling = funnum;
    if ( GetTerm(BHEAD term) > 0 ) {            /* Skip prototype */
        while ( GetTerm(BHEAD term) > 0 ) {
            GETSTOP(term,tstop);
            w = m = term + 1;
            while ( m < tstop && *m != HAAKJE ) m += m[1];
            if ( *m != HAAKJE ) {
                MesPrint("&Illegal attempt to put an expression without brackets in a table");
                BACKINOUT
                goto noway;
            }
            if ( brasize == m - w ) {
                b = brackets;
                while ( *b == *w && w < m ) { b++; w++; }
                if ( w == m ) { /* Same as current bracket. Copy. */
                    if ( weneedit ) {
                        m += m[1] - 1;
                        *m = *term - (m-term);
                        AddNtoC(AC.cbufnum,*m,m,3);
                        numdummies = DetCurDum(BHEAD term) - AM.IndDum;
```

```
                        if ( numdummies > T->numdummies ) T->numdummies = numdummies;
                    }
                    continue; /* Next term */
                }
            }
            if ( weneedit ) {
                AddNtoC(AC.cbufnum,1,&zero,4);   /* Terminate old bracket */
                numcommu = numcommute(C->rhs[curelement],&(C->NumTerms[curelement]));
                C->CanCommu[curelement] = numcommu;
            }
            b = brackets; w = term + 1;
            if ( numsym < 0 ) pw = oldwork + 1;
            else              pw = oldwork + numsym;
            while ( w < m ) *b++ = *w++;
            brasize = b - brackets;
/*
            Now compute the element. See whether we need it
*/
            if ( numsym < 0 ) {
                WORD *bb;
                if ( *brackets != symnum || brasize != brackets[1] ) {
                    weneedit = 0; continue; /* Cannot work! */
                }
/*
                Now count the number of arguments and whether they are numbers
*/
                b = brackets + FUNHEAD;
                bb = brackets+brackets[1];
                i = 0;
                while ( b < bb ) {
                    if ( *b != -SNUMBER ) break;
                    i++;
                    b += 2;
                }
                if ( b < bb || i != T->numind ) {
                    weneedit = 0; continue; /* Cannot work! */
                }
            }
            else if ( brasize > 0 && ( *brackets != SYMBOL
            || brackets[1] < brasize || (brackets[1]-2) > numsym*2 ) ) {
                weneedit = 0; continue; /* Cannot work! */
            }
            numzero = 0; sum = 0;
            if ( numsym > 0 ) {
              for ( i = 0; i < numsym; i++ ) {
                if ( brasize > 0 ) {
                    b = brackets + 2; j = brackets[1]-2;
                    while ( j > 0 ) {
                        if ( *b == oldwork[i] ) break;
                        j -= 2; b += 2;
                    }
                    if ( j <= 0 ) {  /* it was not there */
                        numzero++; pow = 0;
                        if ( 2*numzero+brackets[1]-2 > numsym*2 ) {
                            weneedit = 0; goto nextterm;
                        }
                    }
                    else pow = b[1];
                }
                else pow = 0;
                if ( T->sparse ) *pw++ = pow;
                else if ( pow < T->mm[i].mini || pow > T->mm[i].maxi ) {
                    weneedit = 0; goto nextterm;
                }
                else sum += ( pow - T->mm[i].mini ) * T->mm[i].size;
              }
            }
            else {
              b = brackets + FUNHEAD;
              sum = 0;
              for ( i = 0; i < T->numind; i++ ) {
                pow = b[1];
                b += 2;
                if ( T->sparse ) { *pw++ = pow; }
                else if ( pow < T->mm[i].mini || pow > T->mm[i].maxi ) {
                    weneedit = 0; goto nextterm;
                }
                else sum += ( pow - T->mm[i].mini ) * T->mm[i].size;
              }
            }
            weneedit = 1;
            if ( T->sparse ) {
                if ( numsym < 0 ) pw = oldwork + 1;
                else              pw = oldwork + T->numind;
                i = FindTableTree(T,pw,1);
                if ( i >= 0 ) {
                    sum = i+T->numind;
```

```
/*
Wrong!!!!           C->rhs[T->tablepointers[sum]] = C->Pointer;
*/
                    C->Pointer--; /* Back up over the zero */
                    goto newentry;
                }
                if ( T->totind >= T->reserved ) {
                    if ( T->reserved == 0 ) newreservation = 20;
                    else newreservation = T->reserved;
/*
                    while ( T->totind >= newreservation && newreservation <
                                MAXTABLECOMBUF*(T->numind+TABLEEXTENSION) )
                            newreservation = 2*newreservation;
                    if ( newreservation > MAXTABLECOMBUF*T->numind ) newreservation =
                                MAXTABLECOMBUF*(T->numind+TABLEEXTENSION);
*/
/*---Copied from Fill---------------------------*/
                    while ( T->totind >= newreservation && newreservation < MAXTABLECOMBUF )
                            newreservation = 2*newreservation;
                    if ( newreservation > MAXTABLECOMBUF ) newreservation = MAXTABLECOMBUF;
                    if ( T->totind >= newreservation ) {
                        MesPrint("@More than %ld elements in sparse table",MAXTABLECOMBUF);
                        AC.cbufnum = oldcbuf;
                        AT.WorkPointer = oldwork;
                        Terminate(-1);
                    }
/*---Copied from Fill---------------------------*/
                    if ( T->totind >= newreservation ) {
                        MesPrint("@More than %ld elements in sparse table",MAXTABLECOMBUF);
                        AC.cbufnum = oldcbuf;
                        AT.WorkPointer = oldwork;
                        Terminate(-1);
                    }
                    w = (WORD *)Malloc1(newreservation*sizeof(WORD)*
                            (T->numind+TABLEEXTENSION),"tablepointers");
                    for ( i = T->reserved*(T->numind+TABLEEXTENSION)-1; i >= 0; i-- )
                        w[i] = T->tablepointers[i];
                    if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
                    T->tablepointers = w;
                    T->reserved = newreservation;
                }
                if ( numsym < 0 ) pw = oldwork + 1;
                else              pw = oldwork + numsym;
                for ( sum = T->totind*(T->numind+TABLEEXTENSION), i = 0; i < T->numind; i++ ) {
                    T->tablepointers[sum++] = *pw++;
                }
                InsTableTree(T,T->tablepointers+sum-T->numind);
                (T->totind)++;
            }
#if ( TABLEEXTENSION != 2 )
            else {
                sum *= TABLEEXTENSION;
            }
#endif
/*
            Start a new entry. Copy the element.
*/
            AddRHS(T->bufnum,0);
            T->tablepointers[sum] = C->numrhs;
#if ( TABLEEXTENSION == 2 )
            T->tablepointers[sum+TABLEEXTENSION-1] = -1;
#else
            T->tablepointers[sum+1] = T->bufnum;
            T->tablepointers[sum+2] = -1;
            T->tablepointers[sum+3] = -1;
            T->tablepointers[sum+4] = 0;
            T->tablepointers[sum+5] = 0;
#endif
newentry:   if ( *m == HAAKJE ) { m += m[1] - 1; }
            else m--;
            *m = *term - (m-term);
            AddNtoC(AC.cbufnum,*m,m,5);
            curelement = T->tablepointers[sum];
nextterm:;
        }
        if ( weneedit ) {
            AddNtoC(AC.cbufnum,1,&zero,6);   /* Terminate old bracket */
            numcommu = numcommute(C->rhs[curelement],&(C->NumTerms[curelement]));
            C->CanCommu[curelement] = numcommu;
        }
    }
    if ( fi->handle >= 0 ) {
        SetScratch(fi,&(oldposition));
    }
    else {
        fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + BASEPOSITION(oldposition));
    }
```

```
    BACKINOUT
    AC.cbufnum = oldcbuf;
    AC.tablefilling = 0;
    AT.WorkPointer = oldwork;
    return(0);
noway:
    BACKINOUT
    AC.cbufnum = oldcbuf;
    AC.tablefilling = 0;
    AT.WorkPointer = oldwork;
    return(1);
}

/*
    #] CoFillExpression :
    #[ CoPrintTable :

    Syntax
        PrintTable [+f] [+s] tablename [>[>] file];
    All defined elements are written with individual Fill statements.
    If a file is specified, the result is written to file only.
    The flags of the print statement apply as much as possible.
    We make use of the regular write routines.
*/

int CoPrintTable(UBYTE *inp)
{
    GETIDENTITY
    int fflag = 0, sflag = 0, addflag = 0, error = 0, sum, i, j;
    UBYTE *filename, *p, c, buffer[100], *s, *oldoutputline = AO.OutputLine;
    WORD type, funnum, *expr, *m, num;
    TABLES T = 0;
    WORD oldSkip = AO.OutSkip, oldMode = AC.OutputMode, oldHandle = AC.LogHandle;
    WORD oldType = AO.PrintType, *oldwork = AT.WorkPointer;
    UBYTE *oldFill = AO.OutFill, *oldLine = AO.OutputLine;
#ifdef WITHMPI
    if ( PF.me != MASTER ) return 0;
#endif
/*
    First the flags
*/
    while ( *inp == '+' ) {
        inp++;
        if ( *inp == 'f' || *inp == 'F' ) { fflag = 1; inp++; }
        else if ( *inp == 's' || *inp == 'S' ) { sflag = PRINTONETERM; inp++; }
        else {
            MesPrint("&Illegal + option in PrintTable statement");
            error = 1; inp++;
        }
        while ( *inp != ',' && *inp && *inp != '+' ) {
            if ( !error ) {
                if ( *inp ) {
                    MesPrint("&Illegal + option in PrintTable statement");
                    inp++;
                }
                else {
                    MesPrint("&Unfinished PrintTable statement");
                    return(1);
                }
                error = 1;
            }
            inp++;
        }
        if ( *inp == ',' ) inp++;
    }
/*
    Now the name of the table
*/
    if ( ( p = SkipAName(inp) ) == 0 ) return(1);
    c = *p; *p = 0;
    if ( ( GetVar(inp,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
    || ( T = functions[funnum].tabl ) == 0 ) {
        MesPrint("&%s should be a previously declared table",inp);
        *p = c; return(1);
    }
    if ( T->spare && T->mode == 1 ) T = T->spare;
    *p++ = c;
/*
    Check for a filename. Runs to the end of the statement.
*/
    filename = 0;
    if ( c == '>' ) {
        if ( *p == '>' ) { addflag = 1; p++; }
        filename = p;
    }
    else filename = 0;

```

```
    if ( filename ) {
        if ( addflag ) AC.LogHandle = OpenAddFile((char *)filename);
        else           AC.LogHandle = CreateFile((char *)filename);
        if ( AC.LogHandle < 0 ) {
            MesPrint("&Cannot open file '%s' properly",filename);
            error = 1; goto finally;
        }
        AO.PrintType = PRINTLFILE;
    }
    else if ( fflag && AC.LogHandle >= 0 ) {
        AO.PrintType = PRINTLFILE;
    }
    AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
    AT.WorkPointer += 2*AC.LineLength;

    AO.PrintType |= sflag;
    AC.OutputMode = 0;
    AO.IsBracket = 0;
    AO.OutSkip = 0;
    AR.DeferFlag = 0;
    AC.outsidefun = 1;
    if ( AC.LogHandle == oldHandle ) FiniLine();
    AO.OutputLine = AO.OutFill = (UBYTE *)Malloc1(AC.LineLength+20,"PrintTable");
    AO.OutStop = AO.OutFill + AC.LineLength;
    for ( i = 0; i < T->totind; i++ ) {
        if ( !T->sparse && T->tablepointers[i*TABLEEXTENSION] < 0 ) continue;
        TokenToLine((UBYTE *)"Fill ");
        TokenToLine((UBYTE *)(VARNAME(functions,funnum)));
        TokenToLine((UBYTE *)"(");
        AO.OutSkip = 3;
        if ( T->sparse ) {
            sum = i * ( T->numind + TABLEEXTENSION );
            for ( j = 0; j < T->numind; j++, sum++ ) {
                if ( j > 0 ) TokenToLine((UBYTE *)",");
                num = T->tablepointers[sum];
                s = buffer; s = NumCopy(num,s);
                TokenToLine(buffer);
            }
            expr = cbuf[T->bufnum].rhs[T->tablepointers[sum]];
        }
        else {
            for ( j = 0; j < T->numind; j++ ) {
                if ( j > 0 ) {
                    TokenToLine((UBYTE *)",");
                    num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
                }
                else {
                    num = T->mm[j].mini + i / T->mm[j].size;
                }
                s = buffer; s = NumCopy(num,s);
                TokenToLine(buffer);
            }
            expr = cbuf[T->bufnum].rhs[T->tablepointers[TABLEEXTENSION*i]];
        }
        TOKENTOLINE(") =",")=");
        if ( sflag ) {
            FiniLine();
            if ( AC.OutputSpaces != NOSPACEFORMAT ) TokenToLine((UBYTE *)"   ");
        }
        m = expr;
/*
        WORD lbrac, first;
        lbrac = 0; first = 1;
        while ( *m ) {
            if ( WriteTerm(m,&lbrac,first,1,0) ) {
                MesPrint("Error while writing table");
                error = 1;
                goto finally;
            }
            first = 0;
            m += *m;
        }
        if ( first ) { TOKENTOLINE(" 0","0") }
        else if ( lbrac ) { TOKENTOLINE(" )",")") }
*/
        while ( *m ) m += *m;
        if ( m > expr ) {
            if ( WriteExpression(expr,(LONG)(m-expr)) ) { error = 1; goto finally; }
            AO.OutSkip = 0;
        }
        else {
            TokenToLine((UBYTE *)"0");
        }
        TokenToLine((UBYTE *)";");
        FiniLine();
    }
    M_free(AO.OutputLine,"PrintTable");
```

```
    AO.OutputLine = AO.OutFill = oldoutputline;
/*
    Reset the file pointers and parameters if any. Close file if needed.
*/
finally:
    AO.OutSkip     = oldSkip;
    AC.OutputMode  = oldMode;
    AC.LogHandle   = oldHandle;
    AO.PrintType   = oldType;
    AO.OutFill     = oldFill;
    AO.OutputLine  = oldLine;
    AT.WorkPointer = oldwork;
    AC.outsidefun  = 0;
    return(error);
}

/*
    #] CoPrintTable :
    #[ CoAssign :

    This statement has an easy syntax:
        $name = expression
*/

static WORD AssignLHS[14] = { TYPEASSIGN, 3+SUBEXPSIZE, 0,
                              SUBEXPRESSION, SUBEXPSIZE, 0, 1, 0, 0,0,0,0,0 };

int CoAssign(UBYTE *inp)
{
    int error = 0, retcode;
    UBYTE *name, c;
    WORD number;
    if ( *inp != '$' ) {
nolhs:  MesPrint("&assign statement should have a dollar variable in the LHS");
        return(1);
    }
    inp++; name = inp;
    if ( FG.cTable[*inp] != 0 ) goto nolhs;
    while ( FG.cTable[*inp] < 2 ) inp++;
    if ( AP.PreAssignFlag == 2 ) {
        if ( *inp == '_' ) inp++;
    }
    if ( ( *inp == ',' && inp[1] != '=' ) && ( *inp != '=' ) ) {
        MesPrint("&assign statement should have only a dollar variable in the LHS");
        return(1);
    }
    c = *inp;
    *inp = 0;
    if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
        number = AddDollar(name,DOLUNDEFINED,0,0);
    }
    *inp = c;
    if ( c == ',' ) inp++;
    *inp++ = '=';
    if ( *inp == ',' ) inp++;
/*
    Fake a Prototype and read the RHS
*/
    AssignLHS[7] = AC.cbufnum;
    retcode = CompileAlgebra(inp,RHSIDE,(AssignLHS+3));
    if ( retcode < 0 ) error = 1;
    AC.DumNum = 0;
/*
    Now add the LHS
*/
    AssignLHS[2] = number;
    AssignLHS[5] = retcode;
    AddNtoL(AssignLHS[1],AssignLHS);
/*
    Add to the list of potentially modified dollars (for PARALLEL)
*/
    AddPotModdollar(number);
    return(error);
}

/*
    #] CoAssign :
    #[ CoDeallocateTable :

    Syntax: DeallocateTable tablename(s);
    Should work only for sparse tables.
    Action: Cleans all definitions of elements of a table as if there have
            never been any fill statements.
*/

int CoDeallocateTable(UBYTE *inp)
{
```

```
    UBYTE *p, c;
    TABLES T = 0;
    WORD type, funnum, i;
    c = *inp;
    while ( c ) {
        while ( *inp == ',' ) inp++;
        if ( *inp == 0 ) break;
        if ( ( p = SkipAName(inp) ) == 0 ) return(1);
        c = *p; *p = 0;
        if ( ( GetVar(inp,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
        || ( T = functions[funnum].tabl ) == 0 ) {
            MesPrint("&%s should be a previously declared table",inp);
            *p = c; return(1);
        }
        if ( T->sparse == 0 ) {
            MesPrint("&%s should be a sparse table",inp);
            *p = c; return(1);
        }
        if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
        ClearTableTree(T);
        for (i = 0; i < T->buffersfill; i++ ) { /* was <= */
            finishcbuf(T->buffers[i]);
        }
        T->bufnum = inicbufs();
        T->buffersfill = 0;
        T->buffers[T->buffersfill++] = T->bufnum;
        T->tablepointers = 0;
        T->boomlijst = 0;
        T->totind = 0;
        T->reserved = 0;

        if ( T->spare ) {
            TABLES TT = T->spare;
            if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
            ClearTableTree(TT);
            for (i = 0; i < TT->buffersfill; i++ ) { /* was <= */
                finishcbuf(TT->buffers[i]);
            }
            TT->bufnum = inicbufs();
            TT->buffersfill = 0;
            TT->buffers[T->buffersfill++] = T->bufnum;
            TT->tablepointers = 0;
            TT->boomlijst = 0;
            TT->totind = 0;
            TT->reserved = 0;
        }
        *p++ = c;
        inp = p;
    }
    return(0);
}

/*
    #] CoDeallocateTable :
    #[ CoFactorCache :
*/
/*
int CoFactorCache(UBYTE *inp)
{
    Code to be added in due time
    We need to read 'expression', get its terms through Generator and sort them.
    We store the result in the WorkSpace in argument notation.
    This will be argin.
    Then we do the same with the sequence of factors. They form argout.
    The whole is put in the buffer with the call
        InsertArg(BHEAD argin,argout,1)
    return(0);
}
*/

/*
    #] CoFactorCache :
*/
```

4.9 compcomm.c File Reference

```
#include "form3.h"
#include "comtool.h"
```

Include dependency graph for compcomm.c:

Functions

- int CoCollect (UBYTE ∗s)
- int setonoff (UBYTE ∗s, int ∗flag, int onvalue, int offvalue)
- int CoCompress (UBYTE ∗s)
- int CoFlags (UBYTE ∗s, int value)
- int CoOff (UBYTE ∗s)
- int CoOn (UBYTE ∗s)
- int CoInsideFirst (UBYTE ∗s)
- int CoProperCount (UBYTE ∗s)
- int CoDelete (UBYTE ∗s)
- int CoFormat (UBYTE ∗s)
- int CoKeep (UBYTE ∗s)
- int CoFixIndex (UBYTE ∗s)
- int CoMetric (UBYTE ∗s)
- int DoPrint (UBYTE ∗s, int par)
- int CoPrint (UBYTE ∗s)
- int CoPrintB (UBYTE ∗s)
- int CoNPrint (UBYTE ∗s)
- int CoPushHide (UBYTE ∗s)
- int CoPopHide (UBYTE ∗s)
- int SetExprCases (int par, int setunset, int val)
- int SetExpr (UBYTE ∗s, int setunset, int par)
- int CoDrop (UBYTE ∗s)
- int CoNoDrop (UBYTE ∗s)
- int CoSkip (UBYTE ∗s)
- int CoNoSkip (UBYTE ∗s)
- int CoHide (UBYTE ∗inp)
- int CoIntoHide (UBYTE ∗inp)
- int CoNoHide (UBYTE ∗inp)
- int CoUnHide (UBYTE ∗inp)
- int CoNoUnHide (UBYTE ∗inp)
- void AddToCom (int n, WORD ∗array)
- int AddComString (int n, WORD ∗array, UBYTE ∗thestring, int par)
- int Add2ComStrings (int n, WORD ∗array, UBYTE ∗string1, UBYTE ∗string2)
- int CoDiscard (UBYTE ∗s)
- int CoContract (UBYTE ∗s)
- int CoGoTo (UBYTE ∗inp)

- int CoLabel (UBYTE *inp)
- int DoArgument (UBYTE *s, int par)
- int CoArgument (UBYTE *s)
- int CoEndArgument (UBYTE *s)
- int CoInside (UBYTE *s)
- int CoEndInside (UBYTE *s)
- int CoNormalize (UBYTE *s)
- int CoMakeInteger (UBYTE *s)
- int CoSplitArg (UBYTE *s)
- int CoSplitFirstArg (UBYTE *s)
- int CoSplitLastArg (UBYTE *s)
- int CoFactArg (UBYTE *s)
- int DoSymmetrize (UBYTE *s, int par)
- int CoSymmetrize (UBYTE *s)
- int CoAntiSymmetrize (UBYTE *s)
- int CoCycleSymmetrize (UBYTE *s)
- int CoRCycleSymmetrize (UBYTE *s)
- int CoWrite (UBYTE *s)
- int CoNWrite (UBYTE *s)
- int CoRatio (UBYTE *s)
- int CoRedefine (UBYTE *s)
- int CoRenumber (UBYTE *s)
- int CoSum (UBYTE *s)
- int CoToTensor (UBYTE *s)
- int CoToVector (UBYTE *s)
- int CoTrace4 (UBYTE *s)
- int CoTraceN (UBYTE *s)
- int CoChisholm (UBYTE *s)
- int DoChain (UBYTE *s, int option)
- int CoChainin (UBYTE *s)
- int CoChainout (UBYTE *s)
- int CoExit (UBYTE *s)
- int CoInParallel (UBYTE *s)
- int CoNotInParallel (UBYTE *s)
- int DoInParallel (UBYTE *s, int par)
- int CoInExpression (UBYTE *s)
- int CoEndInExpression (UBYTE *s)
- int CoSetExitFlag (UBYTE *s)
- int CoTryReplace (UBYTE *p)
- int CoModulus (UBYTE *inp)
- int CoRepeat (UBYTE *inp)
- int CoEndRepeat (UBYTE *inp)
- int DoBrackets (UBYTE *inp, int par)
- int CoBracket (UBYTE *inp)
- int CoAntiBracket (UBYTE *inp)
- int CoMultiBracket (UBYTE *inp)
- WORD * CountComp (UBYTE *inp, WORD *to)
- int CoIf (UBYTE *inp)
- int CoElse (UBYTE *p)
- int CoElseIf (UBYTE *inp)
- int CoEndIf (UBYTE *inp)
- int CoWhile (UBYTE *inp)
- int CoEndWhile (UBYTE *inp)
- int DoFindLoop (UBYTE *inp, int mode)
- int CoFindLoop (UBYTE *inp)

- int CoReplaceLoop (UBYTE *inp)
- int CoFunPowers (UBYTE *inp)
- int CoUnitTrace (UBYTE *s)
- int CoTerm (UBYTE *s)
- int CoEndTerm (UBYTE *s)
- int CoSort (UBYTE *s)
- int CoPolyFun (UBYTE *s)
- int CoPolyRatFun (UBYTE *s)
- int CoMerge (UBYTE *inp)
- int CoStuffle (UBYTE *inp)
- int CoProcessBucket (UBYTE *s)
- int CoThreadBucket (UBYTE *s)
- int DoArgPlode (UBYTE *s, int par)
- int CoArgExplode (UBYTE *s)
- int CoArgImplode (UBYTE *s)
- int CoClearTable (UBYTE *s)
- int CoDenominators (UBYTE *s)
- int CoDropCoefficient (UBYTE *s)
- int CoDropSymbols (UBYTE *s)
- int CoToPolynomial (UBYTE *inp)
- int CoFromPolynomial (UBYTE *inp)
- int CoArgToExtraSymbol (UBYTE *s)
- int CoExtraSymbols (UBYTE *inp)
- WORD * GetIfDollarFactor (UBYTE **inp, WORD *w)
- UBYTE * GetDoParam (UBYTE *inp, WORD **wp, int par)
- int CoDo (UBYTE *inp)
- int CoEndDo (UBYTE *inp)
- int CoFactDollar (UBYTE *inp)
- int CoFactorize (UBYTE *s)
- int CoNFactorize (UBYTE *s)
- int CoUnFactorize (UBYTE *s)
- int CoNUnFactorize (UBYTE *s)
- int DoFactorize (UBYTE *s, int par)
- int CoOptimizeOption (UBYTE *s)
- int CoPutInside (UBYTE *inp)
- int CoAntiPutInside (UBYTE *inp)
- int DoPutInside (UBYTE *inp, int par)

4.9.1 Detailed Description

Compiler routines for most statements that don't involve algebraic expressions. Exceptions: all routines involving declarations are in the file names.c When making new statements one can add the compiler routines here and have a look whether there is already a routine that is similar. In that case one can make a copy and modify it.

Definition in file compcomm.c.

4.9.2 Function Documentation

4.9.2.1 CoCollect()

```
int CoCollect (
            UBYTE * s )
```

Definition at line 143 of file compcomm.c.

4.9.2.2 setonoff()

```
int setonoff (
            UBYTE * s,
            int * flag,
            int onvalue,
            int offvalue )
```

Definition at line 205 of file compcomm.c.

4.9.2.3 CoCompress()

```
int CoCompress (
            UBYTE * s )
```

Definition at line 221 of file compcomm.c.

4.9.2.4 CoFlags()

```
int CoFlags (
            UBYTE * s,
            int value )
```

Definition at line 270 of file compcomm.c.

4.9.2.5 CoOff()

```
int CoOff (
            UBYTE * s )
```

Definition at line 305 of file compcomm.c.

4.9.2.6 CoOn()

```
int CoOn (
            UBYTE * s )
```

Definition at line 362 of file compcomm.c.

4.9.2.7 CoInsideFirst()

```
int CoInsideFirst (
            UBYTE * s )
```

Definition at line 609 of file compcomm.c.

4.9.2.8 CoProperCount()

```
int CoProperCount (
            UBYTE * s )
```

Definition at line 616 of file compcomm.c.

4.9.2.9 CoDelete()

```
int CoDelete (
            UBYTE * s )
```

Definition at line 623 of file compcomm.c.

4.9.2.10 CoFormat()

```
int CoFormat (
            UBYTE * s )
```

Definition at line 668 of file compcomm.c.

4.9.2.11 CoKeep()

```
int CoKeep (
            UBYTE * s )
```

Definition at line 900 of file compcomm.c.

4.9.2.12 CoFixIndex()

```
int CoFixIndex (
            UBYTE * s )
```

Definition at line 912 of file compcomm.c.

4.9.2.13 CoMetric()

```
int CoMetric (
            UBYTE * s )
```

Definition at line 946 of file compcomm.c.

4.9.2.14 DoPrint()

```
int DoPrint (
            UBYTE * s,
            int par )
```

Definition at line 954 of file compcomm.c.

4.9.2.15 CoPrint()

```
int CoPrint (
            UBYTE * s )
```

Definition at line 1142 of file compcomm.c.

4.9.2.16 CoPrintB()

```
int CoPrintB (
            UBYTE * s )
```

Definition at line 1149 of file compcomm.c.

4.9.2.17 CoNPrint()

```
int CoNPrint (
            UBYTE * s )
```

Definition at line 1156 of file compcomm.c.

4.9.2.18 CoPushHide()

```
int CoPushHide (
            UBYTE * s )
```

Definition at line 1163 of file compcomm.c.

4.9.2.19 CoPopHide()

```
int CoPopHide (
            UBYTE * s )
```

Definition at line 1207 of file compcomm.c.

4.9.2.20 SetExprCases()

```
int SetExprCases (
            int par,
            int setunset,
            int val )
```

Definition at line 1242 of file compcomm.c.

4.9.2.21 SetExpr()

```
int SetExpr (
            UBYTE * s,
            int setunset,
            int par )
```

Definition at line 1388 of file compcomm.c.

4.9.2.22 CoDrop()

```
int CoDrop (
            UBYTE * s )
```

Definition at line 1433 of file compcomm.c.

4.9.2.23 CoNoDrop()

```
int CoNoDrop (
            UBYTE * s )
```

Definition at line 1440 of file compcomm.c.

4.9.2.24 CoSkip()

```
int CoSkip (
            UBYTE * s )
```

Definition at line 1447 of file compcomm.c.

4.9.2.25 CoNoSkip()

```
int CoNoSkip (
            UBYTE * s )
```

Definition at line 1454 of file compcomm.c.

4.9.2.26 CoHide()

```
int CoHide (
            UBYTE * inp )
```

Definition at line 1461 of file compcomm.c.

4.9.2.27 CoIntoHide()

```
int CoIntoHide (
            UBYTE * inp )
```

Definition at line 1479 of file compcomm.c.

4.9.2.28 CoNoHide()

```
int CoNoHide (
            UBYTE * inp )
```

Definition at line 1497 of file compcomm.c.

4.9.2.29 CoUnHide()

```
int CoUnHide (
            UBYTE * inp )
```

Definition at line 1504 of file compcomm.c.

4.9.2.30 CoNoUnHide()

```
int CoNoUnHide (
            UBYTE * inp )
```

Definition at line 1511 of file compcomm.c.

4.9.2.31 AddToCom()

```
void AddToCom (
            int n,
            WORD * array )
```

Definition at line 1518 of file compcomm.c.

4.9.2.32 AddComString()

```
int AddComString (
            int n,
            WORD * array,
            UBYTE * thestring,
            int par )
```

Definition at line 1533 of file compcomm.c.

4.9.2.33 Add2ComStrings()

```
int Add2ComStrings (
            int n,
            WORD * array,
            UBYTE * string1,
            UBYTE * string2 )
```

Definition at line 1592 of file compcomm.c.

4.9.2.34 CoDiscard()

```
int CoDiscard (
            UBYTE * s )
```

Definition at line 1633 of file compcomm.c.

4.9.2.35 CoContract()

```
int CoContract (
            UBYTE * s )
```

Definition at line 1656 of file compcomm.c.

4.9.2.36 CoGoTo()

```
int CoGoTo (
            UBYTE * inp )
```

Definition at line 1685 of file compcomm.c.

4.9.2.37 CoLabel()

```
int CoLabel (
            UBYTE * inp )
```

Definition at line 1704 of file compcomm.c.

4.9.2.38 DoArgument()

```
int DoArgument (
            UBYTE * s,
            int par )
```

Definition at line 1731 of file compcomm.c.

4.9.2.39 CoArgument()

```
int CoArgument (
            UBYTE * s )
```

Definition at line 1965 of file compcomm.c.

4.9.2.40 CoEndArgument()

```
int CoEndArgument (
            UBYTE * s )
```

Definition at line 1972 of file compcomm.c.

4.9.2.41 CoInside()

```
int CoInside (
            UBYTE * s )
```

Definition at line 1998 of file compcomm.c.

4.9.2.42 CoEndInside()

```
int CoEndInside (
            UBYTE * s )
```

Definition at line 2005 of file compcomm.c.

4.9.2.43 CoNormalize()

```
int CoNormalize (
            UBYTE * s )
```

Definition at line 2031 of file compcomm.c.

4.9.2.44 CoMakeInteger()

```
int CoMakeInteger (
            UBYTE * s )
```

Definition at line 2038 of file compcomm.c.

4.9.2.45 CoSplitArg()

```
int CoSplitArg (
            UBYTE * s )
```

Definition at line 2045 of file compcomm.c.

4.9.2.46 CoSplitFirstArg()

```
int CoSplitFirstArg (
            UBYTE * s )
```

Definition at line 2052 of file compcomm.c.

4.9.2.47 CoSplitLastArg()

```
int CoSplitLastArg (
            UBYTE * s )
```

Definition at line 2059 of file compcomm.c.

4.9.2.48 CoFactArg()

```
int CoFactArg (
            UBYTE * s )
```

Definition at line 2066 of file compcomm.c.

4.9.2.49 DoSymmetrize()

```
int DoSymmetrize (
            UBYTE * s,
            int par )
```

Definition at line 2086 of file compcomm.c.

4.9.2.50 CoSymmetrize()

```
int CoSymmetrize (
            UBYTE * s )
```

Definition at line 2206 of file compcomm.c.

4.9.2.51 CoAntiSymmetrize()

```
int CoAntiSymmetrize (
            UBYTE * s )
```

Definition at line 2213 of file compcomm.c.

4.9.2.52 CoCycleSymmetrize()

```
int CoCycleSymmetrize (
            UBYTE * s )
```

Definition at line 2220 of file compcomm.c.

4.9.2.53 CoRCycleSymmetrize()

```
int CoRCycleSymmetrize (
            UBYTE * s )
```

Definition at line 2227 of file compcomm.c.

4.9.2.54 CoWrite()

```
int CoWrite (
            UBYTE * s )
```

Definition at line 2234 of file compcomm.c.

4.9.2.55 CoNWrite()

```
int CoNWrite (
            UBYTE * s )
```

Definition at line 2259 of file compcomm.c.

4.9.2.56 CoRatio()

```
int CoRatio (
            UBYTE * s )
```

Definition at line 2286 of file compcomm.c.

4.9.2.57 CoRedefine()

```
int CoRedefine (
            UBYTE * s )
```

Definition at line 2326 of file compcomm.c.

4.9.2.58 CoRenumber()

```
int CoRenumber (
            UBYTE * s )
```

Definition at line 2431 of file compcomm.c.

4.9.2.59 CoSum()

```
int CoSum (
            UBYTE * s )
```

Definition at line 2452 of file compcomm.c.

4.9.2.60 CoToTensor()

```
int CoToTensor (
            UBYTE * s )
```

Definition at line 2572 of file compcomm.c.

4.9.2.61 CoToVector()

```
int CoToVector (
            UBYTE * s )
```

Definition at line 2740 of file compcomm.c.

4.9.2.62 CoTrace4()

```
int CoTrace4 (
            UBYTE * s )
```

Definition at line 2810 of file compcomm.c.

4.9.2.63 CoTraceN()

```
int CoTraceN (
            UBYTE * s )
```

Definition at line 2897 of file compcomm.c.

4.9.2.64 CoChisholm()

```
int CoChisholm (
            UBYTE * s )
```

Definition at line 2957 of file compcomm.c.

4.9.2.65 DoChain()

```
int DoChain (
            UBYTE * s,
            int option )
```

Definition at line 3039 of file compcomm.c.

4.9.2.66 CoChainin()

```
int CoChainin (
            UBYTE * s )
```

Definition at line 3083 of file compcomm.c.

4.9.2.67 CoChainout()

```
int CoChainout (
            UBYTE * s )
```

Definition at line 3095 of file compcomm.c.

4.9.2.68 CoExit()

```
int CoExit (
            UBYTE * s )
```

Definition at line 3105 of file compcomm.c.

4.9.2.69 CoInParallel()

```
int CoInParallel (
            UBYTE * s )
```

Definition at line 3132 of file compcomm.c.

4.9.2.70 CoNotInParallel()

```
int CoNotInParallel (
            UBYTE * s )
```

Definition at line 3142 of file compcomm.c.

4.9.2.71 DoInParallel()

```
int DoInParallel (
            UBYTE * s,
            int par )
```

Definition at line 3157 of file compcomm.c.

4.9.2.72 CoInExpression()

```
int CoInExpression (
            UBYTE * s )
```

Definition at line 3226 of file compcomm.c.

4.9.2.73 CoEndInExpression()

```
int CoEndInExpression (
            UBYTE * s )
```

Definition at line 3279 of file compcomm.c.

4.9.2.74 CoSetExitFlag()

```
int CoSetExitFlag (
            UBYTE * s )
```

Definition at line 3305 of file compcomm.c.

4.9.2.75 CoTryReplace()

```
int CoTryReplace (
            UBYTE * p )
```

Definition at line 3319 of file compcomm.c.

4.9.2.76 CoModulus()

```
int CoModulus (
            UBYTE * inp )
```

Definition at line 3422 of file compcomm.c.

4.9.2.77 CoRepeat()

```
int CoRepeat (
            UBYTE * inp )
```

Definition at line 3612 of file compcomm.c.

4.9.2.78 CoEndRepeat()

```
int CoEndRepeat (
            UBYTE * inp )
```

Definition at line 3635 of file compcomm.c.

4.9.2.79 DoBrackets()

```
int DoBrackets (
            UBYTE * inp,
            int par )
```

Definition at line 3673 of file compcomm.c.

4.9.2.80 CoBracket()

```
int CoBracket (
            UBYTE * inp )
```

Definition at line 3800 of file compcomm.c.

4.9.2.81 CoAntiBracket()

```
int CoAntiBracket (
            UBYTE * inp )
```

Definition at line 3808 of file compcomm.c.

4.9.2.82 CoMultiBracket()

```
int CoMultiBracket (
            UBYTE * inp )
```

Definition at line 3819 of file compcomm.c.

4.9.2.83 CountComp()

```
WORD * CountComp (
            UBYTE * inp,
            WORD * to )
```

Definition at line 3951 of file compcomm.c.

4.9.2.84 CoIf()

```
int CoIf (
            UBYTE * inp )
```

Definition at line 4141 of file compcomm.c.

4.9.2.85 CoElse()

```
int CoElse (
            UBYTE * p )
```

Definition at line 4704 of file compcomm.c.

4.9.2.86 CoElseIf()

```
int CoElseIf (
            UBYTE * inp )
```

Definition at line 4731 of file compcomm.c.

4.9.2.87 CoEndIf()

```
int CoEndIf (
            UBYTE * inp )
```

Definition at line 4758 of file compcomm.c.

4.9.2.88 CoWhile()

```
int CoWhile (
            UBYTE * inp )
```

Definition at line 4803 of file compcomm.c.

4.9.2.89 CoEndWhile()

```
int CoEndWhile (
            UBYTE * inp )
```

Definition at line 4824 of file compcomm.c.

4.9.2.90 DoFindLoop()

```
int DoFindLoop (
            UBYTE * inp,
            int mode )
```

Definition at line 4852 of file compcomm.c.

4.9.2.91 CoFindLoop()

```
int CoFindLoop (
            UBYTE * inp )
```

Definition at line 4966 of file compcomm.c.

4.9.2.92 CoReplaceLoop()

```
int CoReplaceLoop (
            UBYTE * inp )
```

Definition at line 4974 of file compcomm.c.

4.9.2.93 CoFunPowers()

```
int CoFunPowers (
            UBYTE * inp )
```

Definition at line 4988 of file compcomm.c.

4.9.2.94 CoUnitTrace()

```
int CoUnitTrace (
            UBYTE * s )
```

Definition at line 5015 of file compcomm.c.

4.9.2.95 CoTerm()

```
int CoTerm (
            UBYTE * s )
```

Definition at line 5050 of file compcomm.c.

4.9.2.96 CoEndTerm()

```
int CoEndTerm (
            UBYTE * s )
```

Definition at line 5098 of file compcomm.c.

4.9.2.97 CoSort()

```
int CoSort (
            UBYTE * s )
```

Definition at line 5125 of file compcomm.c.

4.9.2.98 CoPolyFun()

```
int CoPolyFun (
            UBYTE * s )
```

Definition at line 5164 of file compcomm.c.

4.9.2.99 CoPolyRatFun()

```
int CoPolyRatFun (
            UBYTE * s )
```

Definition at line 5203 of file compcomm.c.

4.9.2.100 CoMerge()

```
int CoMerge (
            UBYTE * inp )
```

Definition at line 5367 of file compcomm.c.

4.9.2.101 CoStuffle()

```
int CoStuffle (
            UBYTE * inp )
```

Definition at line 5423 of file compcomm.c.

4.9.2.102 CoProcessBucket()

```
int CoProcessBucket (
            UBYTE * s )
```

Definition at line 5478 of file compcomm.c.

4.9.2.103 CoThreadBucket()

```
int CoThreadBucket (
            UBYTE * s )
```

Definition at line 5496 of file compcomm.c.

4.9.2.104 DoArgPlode()

```
int DoArgPlode (
            UBYTE * s,
            int par )
```

Definition at line 5526 of file compcomm.c.

4.9.2.105 CoArgExplode()

```
int CoArgExplode (
            UBYTE * s )
```

Definition at line 5582 of file compcomm.c.

4.9.2.106 CoArgImplode()

```
int CoArgImplode (
            UBYTE * s )
```

Definition at line 5589 of file compcomm.c.

4.9.2.107 CoClearTable()

```
int CoClearTable (
            UBYTE * s )
```

Definition at line 5596 of file compcomm.c.

4.9.2.108 CoDenominators()

```
int CoDenominators (
            UBYTE * s )
```

Definition at line 5678 of file compcomm.c.

4.9.2.109 CoDropCoefficient()

```
int CoDropCoefficient (
            UBYTE * s )
```

Definition at line 5707 of file compcomm.c.

4.9.2.110 CoDropSymbols()

```
int CoDropSymbols (
            UBYTE * s )
```

Definition at line 5721 of file compcomm.c.

4.9.2.111 CoToPolynomial()

```
int CoToPolynomial (
            UBYTE * inp )
```

Definition at line 5744 of file compcomm.c.

4.9.2.112 CoFromPolynomial()

```
int CoFromPolynomial (
            UBYTE * inp )
```

Definition at line 5819 of file compcomm.c.

4.9.2.113 CoArgToExtraSymbol()

```
int CoArgToExtraSymbol (
            UBYTE * s )
```

Definition at line 5845 of file compcomm.c.

4.9.2.114 CoExtraSymbols()

```
int CoExtraSymbols (
            UBYTE * inp )
```

Definition at line 5895 of file compcomm.c.

4.9.2.115 GetIfDollarFactor()

```
WORD * GetIfDollarFactor (
            UBYTE ** inp,
            WORD * w )
```

Definition at line 5968 of file compcomm.c.

4.9.2.116 GetDoParam()

```
UBYTE * GetDoParam (
            UBYTE * inp,
            WORD ** wp,
            int par )
```

Definition at line 6026 of file compcomm.c.

4.9.2.117 CoDo()

```
int CoDo (
            UBYTE * inp )
```

Definition at line 6103 of file compcomm.c.

4.9.2.118 CoEndDo()

```
int CoEndDo (
            UBYTE * inp )
```

Definition at line 6206 of file compcomm.c.

4.9.2.119 CoFactDollar()

```
int CoFactDollar (
            UBYTE * inp )
```

Definition at line 6237 of file compcomm.c.

4.9.2.120 CoFactorize()

```
int CoFactorize (
            UBYTE * s )
```

Definition at line 6266 of file compcomm.c.

4.9.2.121 CoNFactorize()

```
int CoNFactorize (
            UBYTE * s )
```

Definition at line 6273 of file compcomm.c.

4.9.2.122 CoUnFactorize()

```
int CoUnFactorize (
            UBYTE * s )
```

Definition at line 6280 of file compcomm.c.

4.9.2.123 CoNUnFactorize()

```
int CoNUnFactorize (
            UBYTE * s )
```

Definition at line 6287 of file compcomm.c.

4.9.2.124 DoFactorize()

```
int DoFactorize (
            UBYTE * s,
            int par )
```

Definition at line 6294 of file compcomm.c.

4.9.2.125 CoOptimizeOption()

```
int CoOptimizeOption (
            UBYTE * s )
```

Definition at line 6427 of file compcomm.c.

4.9.2.126 CoPutInside()

```
int CoPutInside (
            UBYTE * inp )
```

Definition at line 6863 of file compcomm.c.

4.9.2.127 CoAntiPutInside()

```
int CoAntiPutInside (
            UBYTE * inp )
```

Definition at line 6864 of file compcomm.c.

4.9.2.128 DoPutInside()

```
int DoPutInside (
            UBYTE * inp,
            int par )
```

Definition at line 6866 of file compcomm.c.

4.10 compcomm.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ includes :
*/

#include "form3.h"
#include "comtool.h"

static KEYWORD formatoptions[] = {
     {"c",               (TFUN)0,    CMODE,              0}
    ,{"doublefortran",   (TFUN)0,    DOUBLEFORTRANMODE,  0}
    ,{"float",           (TFUN)0,    0,                  2}
    ,{"fortran",         (TFUN)0,    FORTRANMODE,        0}
    ,{"fortran90",       (TFUN)0,    FORTRANMODE,        4}
    ,{"maple",           (TFUN)0,    MAPLEMODE,          0}
    ,{"mathematica",     (TFUN)0,    MATHEMATICAMODE,    0}
    ,{"normal",          (TFUN)0,    NORMALFORMAT,       1}
    ,{"nospaces",        (TFUN)0,    NOSPACEFORMAT,      3}
    ,{"pfortran",        (TFUN)0,    PFORTRANMODE,       0}
    ,{"quadfortran",     (TFUN)0,    QUADRUPLEFORTRANMODE,   0}
    ,{"quadruplefortran",   (TFUN)0,    QUADRUPLEFORTRANMODE,   0}
    ,{"rational",        (TFUN)0,    RATIONALMODE,       1}
    ,{"reduce",          (TFUN)0,    REDUCEMODE,         0}
    ,{"spaces",          (TFUN)0,    NORMALFORMAT,       3}
    ,{"vortran",         (TFUN)0,    VORTRANMODE,        0}
};

static KEYWORD trace4options[] = {
     {"contract",     (TFUN)0,   CHISHOLM,       0          }
    ,{"nocontract",   (TFUN)0,   0,              CHISHOLM   }
    ,{"nosymmetrize",(TFUN)0,   0,              ALSOREVERSE}
    ,{"notrick",      (TFUN)0,   NOTRICK,        0          }
    ,{"symmetrize",   (TFUN)0,   ALSOREVERSE,    0          }
    ,{"trick",        (TFUN)0,   0,              NOTRICK    }
};

static KEYWORD chisoptions[] = {
     {"nosymmetrize",(TFUN)0,   0,              ALSOREVERSE}
    ,{"symmetrize",   (TFUN)0,   ALSOREVERSE,    0          }
};

static KEYWORD writeoptions[] = {
     {"stats",           (TFUN)&(AC.StatsFlag),  1,      0}
    ,{"statistics",      (TFUN)&(AC.StatsFlag),  1,      0}
    ,{"shortstats",      (TFUN)&(AC.ShortStats), 1,      0}
    ,{"shortstatistics",(TFUN)&(AC.ShortStats), 1,      0}
    ,{"warnings",        (TFUN)&(AC.WarnFlag),   1,      0}
    ,{"allwarnings",     (TFUN)&(AC.WarnFlag),   2,      0}
    ,{"setup",           (TFUN)&(AC.SetupFlag),  1,      0}
    ,{"names",           (TFUN)&(AC.NamesFlag),  1,      0}
    ,{"allnames",        (TFUN)&(AC.NamesFlag),  2,      0}
    ,{"codes",           (TFUN)&(AC.CodesFlag),  1,      0}
    ,{"highfirst",       (TFUN)&(AC.SortType),   SORTHIGHFIRST,      SORTLOWFIRST}
    ,{"lowfirst",        (TFUN)&(AC.SortType),   SORTLOWFIRST,       SORTHIGHFIRST}
    ,{"powerfirst",      (TFUN)&(AC.SortType),   SORTPOWERFIRST,     SORTHIGHFIRST}
    ,{"tokens",          (TFUN)&(AC.TokensWriteFlag),1,  0}
```

```
};

static KEYWORD onoffoptions[] = {
     {"compress",        (TFUN)&(AC.NoCompress),  0,  1}
    ,{"checkpoint",      (TFUN)&(AC.CheckpointFlag),  1,  0}
    ,{"insidefirst",     (TFUN)&(AC.insidefirst), 1,  0}
    ,{"propercount",     (TFUN)&(AC.BottomLevel), 1,  0}
    ,{"stats",           (TFUN)&(AC.StatsFlag),  1,  0}
    ,{"statistics",      (TFUN)&(AC.StatsFlag),  1,  0}
    ,{"shortstats",      (TFUN)&(AC.ShortStats), 1,  0}
    ,{"shortstatistics",(TFUN)&(AC.ShortStats), 1,  0}
    ,{"names",           (TFUN)&(AC.NamesFlag),  1,  0}
    ,{"allnames",        (TFUN)&(AC.NamesFlag),  2,  0}
    ,{"warnings",        (TFUN)&(AC.WarnFlag),   1,  0}
    ,{"allwarnings",     (TFUN)&(AC.WarnFlag),   2,  0}
    ,{"highfirst",       (TFUN)&(AC.SortType),   SORTHIGHFIRST,  SORTLOWFIRST}
    ,{"lowfirst",        (TFUN)&(AC.SortType),   SORTLOWFIRST,   SORTHIGHFIRST}
    ,{"powerfirst",      (TFUN)&(AC.SortType),   SORTPOWERFIRST, SORTHIGHFIRST}
    ,{"setup",           (TFUN)&(AC.SetupFlag),  1,  0}
    ,{"codes",           (TFUN)&(AC.CodesFlag),  1,  0}
    ,{"tokens",          (TFUN)&(AC.TokensWriteFlag),1,0}
    ,{"properorder",     (TFUN)&(AC.properorderflag),1,0}
    ,{"threadloadbalancing",(TFUN)&(AC.ThreadBalancing),1,  0}
    ,{"threads",         (TFUN)&(AC.ThreadsFlag),1,  0}
    ,{"threadsortfilesynch",(TFUN)&(AC.ThreadSortFileSynch),1,  0}
    ,{"threadstats",     (TFUN)&(AC.ThreadStats),1,  0}
    ,{"finalstats",      (TFUN)&(AC.FinalStats),1,   0}
    ,{"fewerstats",      (TFUN)&(AC.ShortStatsMax),  10,     0}
    ,{"fewerstatistics",(TFUN)&(AC.ShortStatsMax),  10,     0}
    ,{"processstats",    (TFUN)&(AC.ProcessStats),1, 0}
    ,{"oldparallelstats",(TFUN)&(AC.OldParallelStats),1,0}
    ,{"parallel",        (TFUN)&(AC.parallelflag),PARALLELFLAG,NOPARALLEL_USER}
    ,{"nospacesinnumbers",(TFUN)&(AO.NoSpacesInNumbers),1,0}
    ,{"indentspace",     (TFUN)&(AO.IndentSpace),INDENTSPACE,0}
    ,{"totalsize",       (TFUN)&(AM.PrintTotalSize), 1,  0}
    ,{"flag",            (TFUN)&(AC.debugFlags), 1,  0}
    ,{"oldfactarg",      (TFUN)&(AC.OldFactArgFlag), 1,  0}
    ,{"memdebugflag",    (TFUN)&(AC.MemDebugFlag),   1,  0}
    ,{"oldgcd",          (TFUN)&(AC.OldGCDflag), 1,  0}
    ,{"innertest",       (TFUN)&(AC.InnerTest),  1,  0}
    ,{"wtimestats",      (TFUN)&(AC.WTimeStatsFlag),  1,  0}
};

static WORD one = 1;

/*
    #] includes :
    #[ CoCollect :

    Collect,functionname
*/

int CoCollect(UBYTE *s)
{
/*  --------------change 17-feb-2003 Added percentage */
    WORD numfun;
    int type,x = 0;
    UBYTE *t = SkipAName(s), *t1, *t2;
    AC.AltCollectFun = 0;
    if ( t == 0 ) goto syntaxerror;
    t1 = t; while ( *t1 == ',' || *t1 == ' ' || *t1 == '\t' ) t1++;
    *t = 0; t = t1;
    if ( *t1 && ( FG.cTable[*t1] == 0 || *t1 == '[' ) ) {
        t2 = SkipAName(t1);
        if ( t2 == 0 ) goto syntaxerror;
        t = t2;
        while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
        *t2 = 0;
    }
    else t1 = 0;
    if ( *t && FG.cTable[*t] == 1 ) {
        while ( *t >= '0' && *t <= '9' ) x = 10*x + *t++ - '0';
        if ( x > 100 ) x = 100;
        while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
        if ( *t ) goto syntaxerror;
    }
    else {
        if ( *t ) goto syntaxerror;
        x = 100;
    }
    if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
    || ( functions[numfun].spec != 0 ) ) {
        MesPrint("&%s should be a regular function",s);
        if ( type < 0 ) {
            if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
                AddFunction(s,0,0,0,0,0,-1,-1);
        }
```

```
        return(1);
    }
    AC.CollectFun = numfun+FUNCTION;
    AC.CollectPercentage = (WORD)x;
    if ( t1 ) {
        if ( ( ( type = GetName(AC.varnames,t1,&numfun,WITHAUTO) ) != CFUNCTION )
        || ( functions[numfun].spec != 0 ) ) {
            MesPrint("&%s should be a regular function",t1);
            if ( type < 0 ) {
                if ( GetName(AC.exprnames,t1,&numfun,NOAUTO) == NAMENOTFOUND )
                    AddFunction(t1,0,0,0,0,0,-1,-1);
            }
            return(1);
        }
        AC.AltCollectFun = numfun+FUNCTION;
    }
    return(0);
syntaxerror:
    MesPrint("&Collect statement needs one or two functions (and a percentage) for its argument(s)");
    return(1);
}

/*
    #] CoCollect :
    #[ setonoff :
*/

int setonoff(UBYTE *s, int *flag, int onvalue, int offvalue)
{
    if ( StrICmp(s,(UBYTE *)"on") == 0 ) *flag = onvalue;
    else if ( StrICmp(s,(UBYTE *)"off") == 0 ) *flag = offvalue;
    else {
        MesPrint("&Unknown option: %s, on or off expected",s);
        return(1);
    }
    return(0);
}

/*
    #] setonoff :
    #[ CoCompress :
*/

int CoCompress(UBYTE *s)
{
    GETIDENTITY
    UBYTE *t, c;
    if ( StrICmp(s,(UBYTE *)"on") == 0 ) {
        AC.NoCompress = 0;
        AR.gzipCompress = 0;
    }
    else if ( StrICmp(s,(UBYTE *)"off") == 0 ) {
        AC.NoCompress = 1;
        AR.gzipCompress = 0;
    }
    else {
        t = s; while ( FG.cTable[*t] <= 1 ) t++;
        c = *t; *t = 0;
        if ( StrICmp(s,(UBYTE *)"gzip") == 0 ) {
#ifndef WITHZLIB
            Warning("gzip compression not supported on this platform");
#endif
            s = t; *s = c;
            if ( *s == 0 ) {
                AR.gzipCompress = GZIPDEFAULT;  /* Normally should be 6 */
                return(0);
            }
            while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
            t = s;
            if ( FG.cTable[*s] == 1 ) {
                AR.gzipCompress = *s - '0';
                s++;
                while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
                if ( *s == 0 ) return(0);
            }
            MesPrint("&Unknown gzip option: %s, a digit was expected",t);
            return(1);

        }
        else {
            MesPrint("&Unknown option: %s, on, off or gzip expected",s);
            return(1);
        }
    }
    return(0);
}

```

```
/*
    #] CoCompress :
    #[ CoFlags :
*/

int CoFlags(UBYTE *s,int value)
{
    int i, error = 0;
    if ( *s != ',' ) {
        MesPrint("&Proper syntax is: On/Off Flag,number[s];");
        error = 1;
    }
    while ( *s == ',' ) {
        do { s++; } while ( *s == ',' );
        i = 0;
        if ( FG.cTable[*s] != 1 ) {
            MesPrint("&Proper syntax is: On/Off Flag,number[s];");
            error = 1;
            break;
        }
        while ( FG.cTable[*s] == 1 ) { i = 10*i + *s++ - '0'; }
        if ( i <= 0 || i > MAXFLAGS ) {
            MesPrint("&The number of a flag in On/Off Flag should be in the range 0-%d",(int)MAXFLAGS);
            error = 1;
            break;
        }
        AC.debugFlags[i] = value;
    }
    if ( *s ) {
        MesPrint("&Proper syntax is: On/Off Flag,number[s];");
        error = 1;
    }
    return(error);
}

/*
    #] CoFlags :
    #[ CoOff :
*/

int CoOff(UBYTE *s)
{
    GETIDENTITY
    UBYTE *t, c;
    int i, num = sizeof(onoffoptions)/sizeof(KEYWORD);
    for (;;) {
        while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
        if ( *s == 0 ) return(0);
        if ( chartype[*s] != 0 ) {
            MesPrint("&Illegal character or option encountered in OFF statement");
            return(-1);
        }
        t = s;  while ( chartype[*s] == 0 ) s++;
        c = *s; *s = 0;
        for ( i = 0; i < num; i++ ) {
            if ( StrICont(t,(UBYTE *)(onoffoptions[i].name)) == 0 ) break;
        }
        if ( i >= num ) {
            MesPrint("&Unrecognized option in OFF statement: %s",t);
            *s = c; return(-1);
        }
        else if ( StrICont(t,(UBYTE *)"compress") == 0 ) {
            AR.gzipCompress = 0;
        }
        else if ( StrICont(t,(UBYTE *)"checkpoint") == 0 ) {
            AC.CheckpointInterval = 0;
            if ( AC.CheckpointRunBefore ) { free(AC.CheckpointRunBefore); AC.CheckpointRunBefore = NULL; }
            if ( AC.CheckpointRunAfter ) { free(AC.CheckpointRunAfter); AC.CheckpointRunAfter = NULL; }
            if ( AC.NoShowInput == 0 ) MesPrint("Checkpoints deactivated.");
        }
        else if ( StrICont(t,(UBYTE *)"threads") == 0 ) {
            AS.MultiThreaded = 0;
        }
        else if ( StrICont(t,(UBYTE *)"flag") == 0 ) {
            *s = c;
            return(CoFlags(s,0));
        }
        else if ( StrICont(t,(UBYTE *)"innertest") == 0 ) {
            *s = c;
            AC.InnerTest = 0;
            if ( AC.TestValue ) {
                M_free(AC.TestValue,"InnerTest");
                AC.TestValue = 0;
            }
```

```
        }
        *s = c;
        *((int *)(onoffoptions[i].func)) = onoffoptions[i].flags;
        AR.SortType = AC.SortType;
        AC.mparallelflag = AC.parallelflag | AM.hparallelflag;
    }
}

/*
    #] CoOff :
    #[ CoOn :
*/

int CoOn(UBYTE *s)
{
    GETIDENTITY
    UBYTE *t, c;
    int i, num = sizeof(onoffoptions)/sizeof(KEYWORD);
    LONG interval;
    for (;;) {
        while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
        if ( *s == 0 ) return(0);
        if ( chartype[*s] != 0 ) {
            MesPrint("&Illegal character or option encountered in ON statement");
            return(-1);
        }
        t = s;  while ( chartype[*s] == 0 ) s++;
        c = *s; *s = 0;
        for ( i = 0; i < num; i++ ) {
            if ( StrICont(t,(UBYTE *)(onoffoptions[i].name)) == 0 ) break;
        }
        if ( i >= num ) {
            MesPrint("&Unrecognized option in ON statement: %s",t);
            *s = c; return(-1);
        }
        if ( StrICont(t,(UBYTE *)"compress") == 0 ) {
            AR.gzipCompress = 0;
            *s = c;
            while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
            if ( *s ) {
              t = s;
              while ( FG.cTable[*s] <= 1 ) s++;
              c = *s; *s = 0;
              if ( StrICmp(t,(UBYTE *)"gzip") == 0 ) {}
              else {
                MesPrint("&Unrecognized option in ON compress statement: %s",t);
                return(-1);
              }
              *s = c;
              while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
#ifndef WITHZLIB
              Warning("gzip compression not supported on this platform");
#endif
              if ( FG.cTable[*s] == 1 ) {
                AR.gzipCompress = *s++ - '0';
                while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
                if ( *s ) {
                    MesPrint("&Unrecognized option in ON compress gzip statement: %s",t);
                    return(-1);
                }
              }
              else if ( *s == 0 ) {
                AR.gzipCompress = GZIPDEFAULT;
              }
              else {
                MesPrint("&Unrecognized option in ON compress gzip statement: %s, single digit expected",t);
                return(-1);
              }
            }
        }
        else if ( StrICont(t,(UBYTE *)"checkpoint") == 0 ) {
            AC.CheckpointInterval = 0;
            if ( AC.CheckpointRunBefore ) { free(AC.CheckpointRunBefore); AC.CheckpointRunBefore = NULL; }
            if ( AC.CheckpointRunAfter ) { free(AC.CheckpointRunAfter); AC.CheckpointRunAfter = NULL; }
            *s = c;
            while ( *s ) {
                while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
                if ( FG.cTable[*s] == 1 ) {
                    interval = 0;
                    t = s;
                    do { interval = 10*interval + *s++ - '0'; } while ( FG.cTable[*s] == 1 );
                    if ( *s == 's' || *s == 'S' ) {
                        s++;
                    }
```

```
                    else if ( *s == 'm' || *s == 'M' ) {
                        interval *= 60; s++;
                    }
                    else if ( *s == 'h' || *s == 'H' ) {
                        interval *= 3600; s++;
                    }
                    else if ( *s == 'd' || *s == 'D' ) {
                        interval *= 86400; s++;
                    }
                    if ( *s != ',' && FG.cTable[*s] != 6 && FG.cTable[*s] != 10 ) {
                        MesPrint("&Unrecognized time interval in ON Checkpoint statement: %s", t);
                        return(-1);
                    }
                    AC.CheckpointInterval = interval * 100; /* in 1/100 of seconds */
                }
                else if ( FG.cTable[*s] == 0 ) {
                    int type;
                    t = s;
                    while ( FG.cTable[*s] == 0 ) s++;
                    c = *s; *s = 0;
                    if ( StrICmp(t,(UBYTE *)"run") == 0 ) {
                        type = 3;
                    }
                    else if ( StrICmp(t,(UBYTE *)"runafter") == 0 ) {
                        type = 2;
                    }
                    else if ( StrICmp(t,(UBYTE *)"runbefore") == 0 ) {
                        type = 1;
                    }
                    else {
                        MesPrint("&Unrecognized option in ON Checkpoint statement: %s", t);
                        *s = c; return(-1);
                    }
                    *s = c;
                    if ( *s != '=' && FG.cTable[*(s+1)] != 9 ) {
                        MesPrint("&Unrecognized option in ON Checkpoint statement: %s", t);
                        return(-1);
                    }
                    ++s;
                    t = ++s;
                    while ( *s ) {
                        if ( FG.cTable[*s] == 9 ) {
                            c = *s; *s = 0;
                            if ( type & 1 ) {
                                if ( AC.CheckpointRunBefore ) {
                                    free(AC.CheckpointRunBefore); AC.CheckpointRunBefore = NULL;
                                }
                                if ( s-t > 0 ) {
                                    AC.CheckpointRunBefore = Malloc1(s-t+1, "AC.CheckpointRunBefore");
                                    StrCopy(t, (UBYTE*)AC.CheckpointRunBefore);
                                }
                            }
                            if ( type & 2 ) {
                                if ( AC.CheckpointRunAfter ) {
                                    free(AC.CheckpointRunAfter); AC.CheckpointRunAfter = NULL;
                                }
                                if ( s-t > 0 ) {
                                    AC.CheckpointRunAfter = Malloc1(s-t+1, "AC.CheckpointRunAfter");
                                    StrCopy(t, (UBYTE*)AC.CheckpointRunAfter);
                                }
                            }
                            *s = c;
                            break;
                        }
                        ++s;
                    }
                    if ( FG.cTable[*s] != 9 ) {
                        MesPrint("&Unrecognized option in ON Checkpoint statement: %s", t);
                        return(-1);
                    }
                    ++s;
                }
            }
/*
            if ( AC.NoShowInput == 0 ) {
                MesPrint("Checkpoints activated.");
                if ( AC.CheckpointInterval ) {
                    MesPrint("-> Minimum saving interval: %l seconds.", AC.CheckpointInterval/100);
                }
                else {
                    MesPrint("-> No minimum saving interval given. Saving after EVERY module.");
                }
                if ( AC.CheckpointRunBefore ) {
                    MesPrint("-> Calling script \"%s\" before saving.", AC.CheckpointRunBefore);
                }
                if ( AC.CheckpointRunAfter ) {
                    MesPrint("-> Calling script \"%s\" after saving.", AC.CheckpointRunAfter);
```

```
                }
            }
*/
        }
        else if ( StrICont(t,(UBYTE *)"indentspace") == 0 ) {
            *s = c;
            while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
            if ( *s ) {
                i = 0;
                while ( FG.cTable[*s] == 1 ) { i = 10*i + *s++ - '0'; }
                if ( *s ) {
                    MesPrint("&Unrecognized option in ON IndentSpace statement: %s",t);
                    return(-1);
                }
                if ( i > 40 ) {
                    Warning("IndentSpace parameter adjusted to 40");
                    i = 40;
                }
                AO.IndentSpace = i;
            }
            else {
                AO.IndentSpace = AM.ggIndentSpace;
            }
            return(0);
        }
        else if ( ( StrICont(t,(UBYTE *)"fewerstats") == 0 ) ||
                  ( StrICont(t,(UBYTE *)"fewerstatistics") == 0 ) ) {
            *s = c;
            while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
            if ( *s ) {
                i = 0;
                while ( FG.cTable[*s] == 1 ) { i = 10*i + *s++ - '0'; }
                if ( *s ) {
                    MesPrint("&Unrecognized option in ON FewerStatistics statement: %s",t);
                    return(-1);
                }
                if ( i > AM.S0->MaxPatches ) {
                    if ( AC.WarnFlag )
                    MesPrint("&Warning: FewerStatistics parameter greater than MaxPatches(=%d).
    Adjusted to %d"
                    ,AM.S0->MaxPatches,(AM.S0->MaxPatches+1)/2);
                    i = (AM.S0->MaxPatches+1)/2;
                }
                AC.ShortStatsMax = i;
            }
            else {
                AC.ShortStatsMax = 10; /* default value */
            }
            return(0);
        }
        else if ( StrICont(t,(UBYTE *)"threads") == 0 ) {
            if ( AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
        }
        else if ( StrICont(t,(UBYTE *)"flag") == 0 ) {
            *s = c;
            return(CoFlags(s,1));
        }
        else if ( StrICont(t,(UBYTE *)"innertest") == 0 ) {
            UBYTE *t;
            *s = c;
            while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
            if ( *s ) {
                t = s; while ( *t ) t++;
                while ( t[-1] == ' ' || t[-1] == '\t' ) t--;
                c = *t; *t = 0;
                if ( AC.TestValue ) M_free(AC.TestValue,"InnerTest");
                AC.TestValue = strDupl(s,"InnerTest");
                *t = c;
                s = t;
                while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
            }
            else {
                if ( AC.TestValue ) {
                    M_free(AC.TestValue,"InnerTest");
                    AC.TestValue = 0;
                }
            }
        }
        else { *s = c; }
        *((int *)(onoffoptions[i].func)) = onoffoptions[i].type;
        AR.SortType = AC.SortType;
        AC.mparallelflag = AC.parallelflag | AM.hparallelflag;
    }
}

/*
    #] CoOn :
```

```
    #[ CoInsideFirst :
*/

int CoInsideFirst(UBYTE *s) { return(setonoff(s,&AC.insidefirst,1,0)); }

/*
    #] CoInsideFirst :
    #[ CoProperCount :
*/

int CoProperCount(UBYTE *s) { return(setonoff(s,&AC.BottomLevel,1,0)); }

/*
    #] CoProperCount :
    #[ CoDelete :
*/

int CoDelete(UBYTE *s)
{
    int error = 0;
    if ( StrICmp(s,(UBYTE *)"storage") == 0 ) {
        if ( DeleteStore(1) < 0 ) {
            MesPrint("&Cannot restart storage file");
            error = 1;
        }
    }
    else {
        UBYTE *t = s, c;
        while ( *t && *t != ',' && *t != '>' ) t++;
        c = *t; *t = 0;
        if ( ( StrICmp(s,(UBYTE *)"extrasymbols") == 0 )
        || ( StrICmp(s,(UBYTE *)"extrasymbol") == 0 ) ) {
            WORD x = 0;
/*
            Either deletes all extra symbols or deletes above a given number
*/
            *t = c; s = t;
            if ( *s == '>' ) {
                s++;
                if ( FG.cTable[*s] != 1 ) goto unknown;
                while ( *s <= '9' && *s >= '0' ) x = 10*x + *s++ - '0';
                if ( *s ) goto unknown;
            }
            else if ( *s ) goto unknown;
            if ( x < AM.gnumextrasym ) x = AM.gnumextrasym;
            PruneExtraSymbols(x);
        }
        else {
            *t = c;
unknown:
            MesPrint("&Unknown option: %s",s);
            error = 1;
        }
    }
    return(error);
}

/*
    #] CoDelete :
    #[ CoFormat :
*/

int CoFormat(UBYTE *s)
{
    int error = 0, x;
    KEYWORD *key;
    UBYTE *ss;
    while ( *s == ' ' || *s == ',' ) s++;
    if ( *s == 0 ) {
        AC.OutputMode = 72;
        AC.OutputSpaces = NORMALFORMAT;
        return(error);
    }
/*
    First the optimization level
*/
    if ( *s == 'O' || *s == 'o' ) {
        if ( ( FG.cTable[s[1]] == 1 ) ||
             ( s[1] == '=' && FG.cTable[s[2]] == 1 ) ) {
            s++; if ( *s == '=' ) s++;
            x = 0;
            while ( *s >= '0' && *s <= '9' ) x = 10*x + *s++ - '0';
            while ( *s == ',' ) s++;
            AO.OptimizationLevel = x;
            AO.Optimize.greedytimelimit = 0;
            AO.Optimize.mctstimelimit = 0;
            AO.Optimize.printstats = 0;
```

```
            AO.Optimize.debugflags = 0;
            AO.Optimize.schemeflags = 0;
            AO.Optimize.mctsdecaymode = 1; // default is decreasing C_p with iteration number
            if ( AO.inscheme ) {
                M_free(AO.inscheme,"Horner input scheme");
                AO.inscheme = 0; AO.schemenum = 0;
            }
            switch ( x ) {
                case 0:
                    break;
                case 1:
                    AO.Optimize.mctsconstant.fval = -1.0;
                    AO.Optimize.horner = O_OCCURRENCE;
                    AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
                    AO.Optimize.method = O_CSE;
                    break;
                case 2:
                    AO.Optimize.horner = O_OCCURRENCE;
                    AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
                    AO.Optimize.method = O_GREEDY;
                    AO.Optimize.greedyminnum = 10;
                    AO.Optimize.greedymaxperc = 5;
                    break;
                case 3:
                    AO.Optimize.mctsconstant.fval = 1.0;
                    AO.Optimize.horner = O_MCTS;
                    AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
                    AO.Optimize.method = O_GREEDY;
                    AO.Optimize.mctsnumexpand = 1000;
                    AO.Optimize.mctsnumkeep = 10;
                    AO.Optimize.mctsnumrepeat = 1;
                    AO.Optimize.greedyminnum = 10;
                    AO.Optimize.greedymaxperc = 5;
                    break;
                case 4:
                    AO.Optimize.horner = O_SIMULATED_ANNEALING;
                    AO.Optimize.saIter = 1000;
                    AO.Optimize.saMaxT.fval = 2000;
                    AO.Optimize.saMinT.fval = 1;
                    break;
                default:
                    error = 1;
                    MesPrint("&Illegal optimization specification in format statement");
                    break;
            }
            if ( error == 0 && *s != 0 && x > 0 ) return(CoOptimizeOption(s));
            return(error);
        }
#ifdef EXPOPT
        { UBYTE c;
        ss = s;
        while ( FG.cTable[*s] == 0 ) s++;
        c = *s; *s = 0;
        if ( StrICont(ss,(UBYTE *)"optimize") == 0 ) {
            *s = c;
            while ( *s == ',' ) s++;
            if ( *s == '=' ) s++;
            AO.OptimizationLevel = 3;
            AO.Optimize.mctsconstant.fval = 1.0;
            AO.Optimize.horner = O_MCTS;
            AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
            AO.Optimize.method = O_GREEDY;
            AO.Optimize.mctstimelimit = 0;
            AO.Optimize.mctsnumexpand = 1000;
            AO.Optimize.mctsnumkeep = 10;
            AO.Optimize.mctsnumrepeat = 1;
            AO.Optimize.greedytimelimit = 0;
            AO.Optimize.greedyminnum = 10;
            AO.Optimize.greedymaxperc = 5;
            AO.Optimize.printstats = 0;
            AO.Optimize.debugflags = 0;
            AO.Optimize.schemeflags = 0;
            AO.Optimize.mctsdecaymode = 1;
            if ( AO.inscheme ) {
                M_free(AO.inscheme,"Horner input scheme");
                AO.inscheme = 0; AO.schemenum = 0;
            }
            return(CoOptimizeOption(s));
        }
        else {
            error = 1;
            MesPrint("&Illegal optimization specification in format statement");
            return(error);
        }
        }
#endif
    }
```

```
    else if ( FG.cTable[*s] == 1 ) {
        x = 0;
        while ( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
        if ( x <= 0 || x >= MAXLINELENGTH ) {
            x = 72;
            error = 1;
            MesPrint("&Illegal value for linesize: %d",x);
        }
        if ( x < 39 ) {
            MesPrint(" ... Too small value for linesize corrected to 39");
            x = 39;
        }
        AO.DoubleFlag = 0;
/*
        The next line resets the mode to normal. Because the special modes
        reset the line length we have a little problem with the special modes
        and customized line length. We try to improve by removing the next line
*/
/*      AC.OutputMode = 0;  */
        AC.LineLength = x;
        if ( *s != 0 ) {
            error = 1;
            MesPrint("&Illegal linesize field in format statement");
        }
    }
    else {
        key = FindKeyWord(s,formatoptions,
            sizeof(formatoptions)/sizeof(KEYWORD));
        if ( key ) {
            if ( key->flags == 0 ) {
                if ( key->type == FORTRANMODE || key->type == PFORTRANMODE
                || key->type == DOUBLEFORTRANMODE
                || key->type == QUADRUPLEFORTRANMODE || key->type == VORTRANMODE ) {
                    AC.IsFortran90 = ISNOTFORTRAN90;
                    if ( AC.Fortran90Kind ) {
                        M_free(AC.Fortran90Kind,"Fortran90 Kind");
                        AC.Fortran90Kind = 0;
                    }
                }
                AO.DoubleFlag = 0;
                AC.OutputMode = key->type & NODOUBLEMASK;
                if ( ( key->type & DOUBLEPRECISIONFLAG ) != 0 ) {
                    AO.DoubleFlag = 1;
                }
                else if ( ( key->type & QUADRUPLEPRECISIONFLAG ) != 0 ) {
                    AO.DoubleFlag = 2;
                }
            }
            else if ( key->flags == 1 ) {
                AC.OutputMode = AC.OutNumberType = key->type;
            }
            else if ( key->flags == 2 ) {
                while ( FG.cTable[*s] == 0 ) s++;
                if ( *s == 0 ) AC.OutNumberType = 10;
                else if ( *s == ',' ) {
                    s++;
                    x = 0;
                    while ( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
                    if ( *s != 0 ) {
                        error = 1;
                        MesPrint("&Illegal float format specifier");
                    }
                    else {
                        if ( x < 3 ) {
                            x = 3;
                            MesPrint("& ... float format value corrected to 3");
                        }
                        if ( x > 100 ) {
                            x = 100;
                            MesPrint("& ... float format value corrected to 100");
                        }
                        AC.OutNumberType = x;
                    }
                }
            }
            else if ( key->flags == 3 ) {
                AC.OutputSpaces = key->type;
            }
            else if ( key->flags == 4 ) {
                AC.IsFortran90 = ISFORTRAN90;
                if ( AC.Fortran90Kind ) {
                    M_free(AC.Fortran90Kind,"Fortran90 Kind");
                    AC.Fortran90Kind = 0;
                }
                while ( FG.cTable[*s] <= 1 ) s++;
                if ( *s == ',' ) {
                    s++; ss = s;
```

```
                    while ( *ss && *ss != ',' ) ss++;
                    if ( *ss == ',' ) {
                        MesPrint("&No white space or comma's allowed in Fortran90 option: %s",s);
    error = 1;
                    }
                    else {
                        AC.Fortran90Kind = strDup1(s,"Fortran90 Kind");
                    }
                }
                AO.DoubleFlag = 0;
                AC.OutputMode = key->type & NODOUBLEMASK;
            }
        }
        else if ( ( *s == 'c' || *s == 'C' ) && ( FG.cTable[s[1]] == 1 ) ) {
            UBYTE *ss = s+1;
            WORD x = 0;
            while ( *ss >= '0' && *ss <= '9' ) x = 10*x + *ss++ - '0';
            if ( *ss != 0 ) goto Unknown;
            AC.OutputMode = CMODE;
            AC.Cnumpows = x;
        }
        else {
Unknown:    MesPrint("&Unknown option: %s",s); error = 1;
        }
        AC.LineLength = 72;
    }
    return(error);
}

/*
    #] CoFormat :
    #[ CoKeep :
*/

int CoKeep(UBYTE *s)
{
    if ( StrICmp(s,(UBYTE *)"brackets") == 0 ) AC.ComDefer = 1;
    else { MesPrint("&Unknown option: '%s'",s); return(1); }
    return(0);
}

/*
    #] CoKeep :
    #[ CoFixIndex :
*/

int CoFixIndex(UBYTE *s)
{
    int x, y, error = 0;
    while ( *s ) {
        if ( FG.cTable[*s] != 1 ) {
proper:     MesPrint("&Proper syntax is: FixIndex,number:value[,number,value];");
            return(1);
        }
        ParseNumber(x,s)
        if ( *s != ':' ) goto proper;
        s++;
        if ( *s != '-' && *s != '+' && FG.cTable[*s] != 1 ) goto proper;
        ParseSignedNumber(y,s)
        if ( *s && *s != ',' ) goto proper;
        while ( *s == ',' ) s++;
        if ( x >= AM.OffsetIndex ) {
            MesPrint("&Fixed index out of allowed range. Change ConstIndex in setup file?");
            MesPrint("&Current value of ConstIndex = %d",AM.OffsetIndex-1);
            error = 1;
        }
        if ( y != (int)((WORD)y) ) {
            MesPrint("&Value of d_(%d,%d) outside range for this computer",x,x);
            error = 1;
        }
        if ( error == 0 ) AC.FixIndices[x] = y;
    }
    return(error);
}

/*
    #] CoFixIndex :
    #[ CoMetric :
*/

int CoMetric(UBYTE *s)
{ DUMMYUSE(s); MesPrint("&The metric statement does not do anything yet"); return(1); }

/*
    #] CoMetric :
    #[ DoPrint :
*/
```

```
int DoPrint(UBYTE *s, int par)
{
    int i, error = 0, numdol = 0, type;
    UBYTE *name, c, *t;
    EXPRESSIONS e;
    WORD numexpr, tofile = 0, *w;
    CBUF *C = cbuf + AC.cbufnum;
    while ( *s == ',' ) s++;
/*  if ( s[-1] == '+' || s[-1] == '-' ) s--; */
    if ( ( *s == '+' || *s == '-' ) && ( s[1] == 'f' || s[1] == 'F' ) ) {
        t = s + 2; while ( *t == ' ' || *t == ',' ) t++;
        if ( *t == '"' ) {
            if ( *s == '+' ) tofile = 1;
            s = t;
        }
    }
    if ( par == PRINTON && *s == '"' ) {
        WORD code;
        if ( tofile == 1 ) code = TYPEFPRINT;
        else code = TYPEPRINT;
        s++; name = s;
        while ( *s && *s != '"' ) {
            if ( *s == '\\' ) s++;
            if ( *s == '%' && s[1] == '$' ) numdol++;
            s++;
        }
        if ( *s != '"' ) {
            MesPrint("&String in print statement should be enclosed in \"");
            return(1);
        }
        *s = 0;
        AddComString(1,&code,name,1);
        *s++ = '"';
        while ( *s == ',' ) {
            s++;
            if ( *s == '$' ) {
                s++; name = s; while ( FG.cTable[*s] <= 1 ) s++;
                c = *s; *s = 0;
                type = GetName(AC.dollarnames,name,&numexpr,NOAUTO);
                if ( type == NAMENOTFOUND ) {
                    MesPrint("&$ variable %s not (yet) defined",name);
                    error = 1;
                }
                else {
                    C->lhs[C->numlhs][1] += 2;
                    *(C->Pointer)++ = DOLLAREXPRESSION;
                    *(C->Pointer)++ = numexpr;
                    numdol--;
                }
            }
            else {
                MesPrint("&Illegal object in print statement");
                error = 1;
                return(error);
            }
            *s = c;
            if ( c == '[' ) {
                w = C->Pointer;
                s++;
                s = GetDoParam(s,&(C->Pointer),-1);
                if ( s == 0 ) return(1);
                if ( *s != ']' ) {
                    MesPrint("&unmatched [] in $ factor");
                    return(1);
                }
                C->lhs[C->numlhs][1] += C->Pointer - w;
                s++;
            }
        }
        if ( *s != 0 ) {
            MesPrint("&Illegal object in print statement");
            error = 1;
        }
        if ( numdol > 0 ) {
            MesPrint("&More $ variables asked for than provided");
            error = 1;
        }
        *(C->Pointer)++ = 0;
        return(error);
    }
    if ( *s == 0 ) {    /* All active expressions */
AllExpr:
        for ( e = Expressions, i = NumExpressions; i > 0; i--, e++ ) {
            if ( e->status == LOCALEXPRESSION || e->status ==
            GLOBALEXPRESSION || e->status == UNHIDELEXPRESSION
            || e->status == UNHIDEGEXPRESSION ) e->printflag = par;
```

```
        }
        return(error);
    }
    while ( *s ) {
        if ( *s == '+' ) {
            s++;
            if ( tolower(*s) == 'f' ) par |= PRINTLFILE;
            else if ( tolower(*s) == 's' ) {
                if ( tolower(s[1]) == 's' ) {
                    if ( tolower(s[2]) == 's' ) {
                        par |= PRINTONEFUNCTION | PRINTONETERM | PRINTALL;
                        s++;
                    }
                    else if ( ( par & 3 ) < 2 ) par |= PRINTONEFUNCTION | PRINTONETERM;
                    s++;
                }
                else {
                    if ( ( par & 3 ) < 2 ) par |= PRINTONETERM;
                }
            }
            else {
illeg:              MesPrint("&Illegal option in (n)print statement");
                error = 1;
            }
            s++;
            if ( *s == 0 ) goto AllExpr;
        }
        else if ( *s == '-' ) {
            s++;
            if ( tolower(*s) == 'f' ) par &= ~PRINTLFILE;
            else if ( tolower(*s) == 's' ) {
                if ( tolower(s[1]) == 's' ) {
                    if ( tolower(s[2]) == 's' ) {
                        par &= ~PRINTALL;
                        s++;
                    }
                    else if ( ( par & 3 ) < 2 ) {
                        par &= ~PRINTONEFUNCTION;
                        par &= ~PRINTALL;
                    }
                    s++;
                }
                else {
                    if ( ( par & 3 ) < 2 ) {
                        par &= ~PRINTONETERM;
                        par &= ~PRINTONEFUNCTION;
                        par &= ~PRINTALL;
                    }
                }
            }
            else goto illeg;
            s++;
            if ( *s == 0 ) goto AllExpr;
        }
        else if ( FG.cTable[*s] == 0 || *s == '[' ) {
            name = s;
            if ( ( s = SkipAName(s) ) == 0 ) {
                MesPrint("&Improper name in (n)print statement");
                return(1);
            }
            c = *s; *s = 0;
            if ( ( GetName(AC.exprnames,name,&numexpr,NOAUTO) == CEXPRESSION )
            && ( Expressions[numexpr].status == LOCALEXPRESSION
            || Expressions[numexpr].status == GLOBALEXPRESSION ) ) {
FoundExpr:;
                if ( c == '[' && s[1] == ']' ) {
                    Expressions[numexpr].printflag = par | PRINTCONTENTS;
                    *s++ = c; c = *++s;
                }
                else
                    Expressions[numexpr].printflag = par;
            }
            else if ( GetLastExprName(name,&numexpr)
            && ( Expressions[numexpr].status == LOCALEXPRESSION
            || Expressions[numexpr].status == GLOBALEXPRESSION
            || Expressions[numexpr].status == UNHIDELEXPRESSION
            || Expressions[numexpr].status == UNHIDEGEXPRESSION
            ) ) {
                goto FoundExpr;
            }
            else {
                MesPrint("&%s is not the name of an active expression",name);
                error = 1;
            }
            *s++ = c;
            if ( c == 0 ) return(0);
            if ( c == '-' || c == '+' ) s--;
```

```
        }
        else if ( *s == ',' ) s++;
        else {
            MesPrint("&Illegal object in (n)print statement");
            return(1);
        }
    }
    return(0);
}

/*
    #] DoPrint :
    #[ CoPrint :
*/

int CoPrint(UBYTE *s) { return(DoPrint(s,PRINTON)); }

/*
    #] CoPrint :
    #[ CoPrintB :
*/

int CoPrintB(UBYTE *s) { return(DoPrint(s,PRINTCONTENT)); }

/*
    #] CoPrintB :
    #[ CoNPrint :
*/

int CoNPrint(UBYTE *s) { return(DoPrint(s,PRINTOFF)); }

/*
    #] CoNPrint :
    #[ CoPushHide :
*/

int CoPushHide(UBYTE *s)
{
    GETIDENTITY
    WORD *ScratchBuf;
    int i;
    if ( AR.Fscr[2].PObuffer == 0 ) {
        ScratchBuf = (WORD *)Malloc1(AM.HideSize*sizeof(WORD),"hidesize");
        AR.Fscr[2].POsize = AM.HideSize * sizeof(WORD);
        AR.Fscr[2].POfull = AR.Fscr[2].POfill = AR.Fscr[2].PObuffer = ScratchBuf;
        AR.Fscr[2].POstop = AR.Fscr[2].PObuffer + AM.HideSize;
        PUTZERO(AR.Fscr[2].POposition);
    }
    while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
    AC.HideLevel += 2;
    if ( *s ) {
        MesPrint("&PushHide statement should have no arguments");
        return(-1);
    }
    for ( i = 0; i < NumExpressions; i++ ) {
        switch ( Expressions[i].status ) {
            case DROPLEXPRESSION:
            case SKIPLEXPRESSION:
            case LOCALEXPRESSION:
                Expressions[i].status = HIDELEXPRESSION;
                Expressions[i].hidelevel = AC.HideLevel-1;
                break;
            case DROPGEXPRESSION:
            case SKIPGEXPRESSION:
            case GLOBALEXPRESSION:
                Expressions[i].status = HIDEGEXPRESSION;
                Expressions[i].hidelevel = AC.HideLevel-1;
                break;
            default:
                break;
        }
    }
    return(0);
}

/*
    #] CoPushHide :
    #[ CoPopHide :
*/

int CoPopHide(UBYTE *s)
{
    int i;
    while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
    if ( AC.HideLevel <= 0 ) {
        MesPrint("&PopHide statement without corresponding PushHide statement");
        return(-1);
```

```
    }
    AC.HideLevel -= 2;
    if ( *s ) {
        MesPrint("&PopHide statement should have no arguments");
        return(-1);
    }
    for ( i = 0; i < NumExpressions; i++ ) {
        switch ( Expressions[i].status ) {
            case HIDDENLEXPRESSION:
                if ( Expressions[i].hidelevel > AC.HideLevel )
                    Expressions[i].status = UNHIDELEXPRESSION;
                break;
            case HIDDENGEXPRESSION:
                if ( Expressions[i].hidelevel > AC.HideLevel )
                    Expressions[i].status = UNHIDEGEXPRESSION;
                break;
            default:
                break;
        }
    }
    return(0);
}

/*
    #] CoPopHide :
    #[ SetExprCases :
*/

int SetExprCases(int par, int setunset, int val)
{
    switch ( par ) {
        case SKIP:
            switch ( val ) {
                case SKIPLEXPRESSION:
                    if ( !setunset ) val = LOCALEXPRESSION;
                    break;
                case SKIPGEXPRESSION:
                    if ( !setunset ) val = GLOBALEXPRESSION;
                    break;
                case LOCALEXPRESSION:
                    if ( setunset ) val = SKIPLEXPRESSION;
                    break;
                case GLOBALEXPRESSION:
                    if ( setunset ) val = SKIPGEXPRESSION;
                    break;
                case INTOHIDEGEXPRESSION:
                case INTOHIDELEXPRESSION:
                default:
                    break;
            }
            break;
        case DROP:
            switch ( val ) {
                case SKIPLEXPRESSION:
                case LOCALEXPRESSION:
                case HIDELEXPRESSION:
                    if ( setunset ) val = DROPLEXPRESSION;
                    break;
                case DROPLEXPRESSION:
                    if ( !setunset ) val = LOCALEXPRESSION;
                    break;
                case SKIPGEXPRESSION:
                case GLOBALEXPRESSION:
                case HIDEGEXPRESSION:
                    if ( setunset ) val = DROPGEXPRESSION;
                    break;
                case DROPGEXPRESSION:
                    if ( !setunset ) val = GLOBALEXPRESSION;
                    break;
                case HIDDENLEXPRESSION:
                case UNHIDELEXPRESSION:
                    if ( setunset ) val = DROPHLEXPRESSION;
                    break;
                case HIDDENGEXPRESSION:
                case UNHIDEGEXPRESSION:
                    if ( setunset ) val = DROPHGEXPRESSION;
                    break;
                case DROPHLEXPRESSION:
                    if ( !setunset ) val = HIDDENLEXPRESSION;
                    break;
                case DROPHGEXPRESSION:
                    if ( !setunset ) val = HIDDENGEXPRESSION;
                    break;
                case INTOHIDEGEXPRESSION:
                case INTOHIDELEXPRESSION:
                default:
                    break;
```

```
            }
            break;
        case HIDE:
            switch ( val ) {
                case DROPLEXPRESSION:
                case SKIPLEXPRESSION:
                case LOCALEXPRESSION:
                    if ( setunset ) val = HIDELEXPRESSION;
                    break;
                case HIDELEXPRESSION:
                    if ( !setunset ) val = LOCALEXPRESSION;
                    break;
                case DROPGEXPRESSION:
                case SKIPGEXPRESSION:
                case GLOBALEXPRESSION:
                    if ( setunset ) val = HIDEGEXPRESSION;
                    break;
                case HIDEGEXPRESSION:
                    if ( !setunset ) val = GLOBALEXPRESSION;
                    break;
                case INTOHIDEGEXPRESSION:
                case INTOHIDELEXPRESSION:
                default:
                    break;
            }
            break;
        case UNHIDE:
            switch ( val ) {
                case HIDDENLEXPRESSION:
                case DROPHLEXPRESSION:
                    if ( setunset ) val = UNHIDELEXPRESSION;
                    break;
                case UNHIDELEXPRESSION:
                    if ( !setunset ) val = HIDDENLEXPRESSION;
                    break;
                case HIDDENGEXPRESSION:
                case DROPHGEXPRESSION:
                    if ( setunset ) val = UNHIDEGEXPRESSION;
                    break;
                case UNHIDEGEXPRESSION:
                    if ( !setunset ) val = HIDDENGEXPRESSION;
                    break;
                case INTOHIDEGEXPRESSION:
                case INTOHIDELEXPRESSION:
                default:
                    break;
            }
            break;
        case INTOHIDE:
            switch ( val ) {
                case HIDDENLEXPRESSION:
                case HIDDENGEXPRESSION:
                    MesPrint("&Expression is already hidden");
                    return(-1);
                case DROPHLEXPRESSION:
                case DROPHGEXPRESSION:
                case UNHIDELEXPRESSION:
                case UNHIDEGEXPRESSION:
                    MesPrint("&Cannot unhide and put intohide expression in the same module");
                    return(-1);
                case LOCALEXPRESSION:
                case DROPLEXPRESSION:
                case SKIPLEXPRESSION:
                case HIDELEXPRESSION:
                    if ( setunset ) val = INTOHIDELEXPRESSION;
                    break;
                case GLOBALEXPRESSION:
                case DROPGEXPRESSION:
                case SKIPGEXPRESSION:
                case HIDEGEXPRESSION:
                    if ( setunset ) val = INTOHIDEGEXPRESSION;
                    break;
                default:
                    break;
            }
            break;
        default:
            break;
    }
    return(val);
}

/*
    #] SetExprCases :
    #[ SetExpr :
*/

```

```c
int SetExpr(UBYTE *s, int setunset, int par)
{
    WORD *w, numexpr;
    int error = 0, i;
    UBYTE *name, c;
    if ( *s == 0 && ( par != INTOHIDE ) ) {
        for ( i = 0; i < NumExpressions; i++ ) {
            w = &(Expressions[i].status);
            *w = SetExprCases(par,setunset,*w);
            if ( *w < 0 ) error = 1;
            if ( par == HIDE && setunset == 1 )
                Expressions[i].hidelevel = AC.HideLevel;
        }
        return(0);
    }
    while ( *s ) {
        if ( *s == ',' ) { s++; continue; }
        if ( *s == '0' ) { s++; continue; }
        name = s;
        if ( ( s = SkipAName(s) ) == 0 ) {
            MesPrint("&Improper name for an expression: '%s'",name);
            return(1);
        }
        c = *s; *s = 0;
        if ( GetName(AC.exprnames,name,&numexpr,NOAUTO) == CEXPRESSION ) {
            w = &(Expressions[numexpr].status);
            *w = SetExprCases(par,setunset,*w);
            if ( *w < 0 ) error = 1;
            if ( ( par == HIDE || par == INTOHIDE ) && setunset == 1 )
                Expressions[numexpr].hidelevel = AC.HideLevel;
        }
        else if ( GetName(AC.varnames,name,&numexpr,NOAUTO) != NAMENOTFOUND ) {
            MesPrint("&%s is not an expression",name);
            error = 1;
        }
        *s = c;
    }
    return(error);
}

/*
    #] SetExpr :
    #[ CoDrop :
*/

int CoDrop(UBYTE *s) { return(SetExpr(s,1,DROP)); }

/*
    #] CoDrop :
    #[ CoNoDrop :
*/

int CoNoDrop(UBYTE *s) { return(SetExpr(s,0,DROP)); }

/*
    #] CoNoDrop :
    #[ CoSkip :
*/

int CoSkip(UBYTE *s) { return(SetExpr(s,1,SKIP)); }

/*
    #] CoSkip :
    #[ CoNoSkip :
*/

int CoNoSkip(UBYTE *s) { return(SetExpr(s,0,SKIP)); }

/*
    #] CoNoSkip :
    #[ CoHide :
*/

int CoHide(UBYTE *inp) {
    GETIDENTITY
    WORD *ScratchBuf;
    if ( AR.Fscr[2].PObuffer == 0 ) {
        ScratchBuf = (WORD *)Malloc1(AM.HideSize*sizeof(WORD),"hidesize");
        AR.Fscr[2].POsize = AM.HideSize * sizeof(WORD);
        AR.Fscr[2].POfull = AR.Fscr[2].POfill = AR.Fscr[2].PObuffer = ScratchBuf;
        AR.Fscr[2].POstop = AR.Fscr[2].PObuffer + AM.HideSize;
        PUTZERO(AR.Fscr[2].POposition);
    }
    return(SetExpr(inp,1,HIDE));
}

/*
```

```
    #] CoHide :
    #[ CoIntoHide :
*/

int CoIntoHide(UBYTE *inp) {
    GETIDENTITY
    WORD *ScratchBuf;
    if ( AR.Fscr[2].PObuffer == 0 ) {
        ScratchBuf = (WORD *)Malloc1(AM.HideSize*sizeof(WORD),"hidesize");
        AR.Fscr[2].POsize = AM.HideSize * sizeof(WORD);
        AR.Fscr[2].POfull = AR.Fscr[2].POfill = AR.Fscr[2].PObuffer = ScratchBuf;
        AR.Fscr[2].POstop = AR.Fscr[2].PObuffer + AM.HideSize;
        PUTZERO(AR.Fscr[2].POposition);
    }
    return(SetExpr(inp,1,INTOHIDE));
}

/*
    #] CoIntoHide :
    #[ CoNoHide :
*/

int CoNoHide(UBYTE *inp) { return(SetExpr(inp,0,HIDE)); }

/*
    #] CoNoHide :
    #[ CoUnHide :
*/

int CoUnHide(UBYTE *inp) { return(SetExpr(inp,1,UNHIDE)); }

/*
    #] CoUnHide :
    #[ CoNoUnHide :
*/

int CoNoUnHide(UBYTE *inp) { return(SetExpr(inp,0,UNHIDE)); }

/*
    #] CoNoUnHide :
    #[ AddToCom :
*/

void AddToCom(int n, WORD *array)
{
    CBUF *C = cbuf+AC.cbufnum;
#ifdef COMPBUFDEBUG
    MesPrint("  %a",n,array);
#endif
    while ( C->Pointer+n >= C->Top ) DoubleCbuffer(AC.cbufnum,C->Pointer,18);
    while ( --n >= 0 ) *(C->Pointer)++ = *array++;
}

/*
    #] AddToCom :
    #[ AddComString :
*/

int AddComString(int n, WORD *array, UBYTE *thestring, int par)
{
    CBUF *C = cbuf+AC.cbufnum;
    UBYTE *s = thestring, *w;
#ifdef COMPBUFDEBUG
    WORD *cc;
    UBYTE *ww;
#endif
    int i, numchars = 0, size, zeroes;
    while ( *s ) {
        if ( *s == '\\' ) s++;
        else if ( par == 1 &&
        ( ( *s == '%' && s[1] != 't' && s[1] != 'T' && s[1] != '$' &&
         s[1] != 'w' && s[1] != 'W' && s[1] != 'r' && s[1] != 0 ) || *s == '#'
        || *s == '@' || *s == '&' ) ) {
            numchars++;
        }
        s++; numchars++;
    }
    AddLHS(AC.cbufnum);
    size = numchars/sizeof(WORD)+1;
    while ( C->Pointer+size+n+2 >= C->Top ) DoubleCbuffer(AC.cbufnum,C->Pointer,19);
#ifdef COMPBUFDEBUG
    cc = C->Pointer;
#endif
    *(C->Pointer)++ = array[0];
    *(C->Pointer)++ = size+n+2;
    for ( i = 1; i < n; i++ ) *(C->Pointer)++ = array[i];
    *(C->Pointer)++ = size;
```

```
#ifdef COMPBUFDEBUG
    ww =
#endif
    w = (UBYTE *)(C->Pointer);
    zeroes = size*sizeof(WORD)-numchars;
    s = thestring;
    while ( *s ) {
        if ( *s == '\\' ) s++;
        else if ( par == 1 && ( ( *s == '%' &&
        s[1] != 't' && s[1] != 'T' && s[1] != '$' &&
        s[1] != 'w' && s[1] != 'W' && s[1] != 'r' && s[1] != 0 ) || *s == '#'
        || *s == '@' || *s == '&' ) ) {
            *w++ = '%';
        }
        *w++ = *s++;
    }
    while ( --zeroes >= 0 ) *w++ = 0;
    C->Pointer += size;
#ifdef COMPBUFDEBUG
    MesPrint("LH: %a",size+1+n,cc);
    MesPrint("        %s",thestring);
#endif
    return(0);
}

/*
    #] AddComString :
    #[ Add2ComStrings :
*/

int Add2ComStrings(int n, WORD *array, UBYTE *string1, UBYTE *string2)
{
    CBUF *C = cbuf+AC.cbufnum;
    UBYTE *s1 = string1, *s2 = string2, *w;
    int i, num1chars = 0, num2chars = 0, size1, size2, zeroes1, zeroes2;
    AddLHS(AC.cbufnum);
    while ( *s1 ) { s1++; num1chars++; }
    size1 = num1chars/sizeof(WORD)+1;
    if ( s2 ) {
        while ( *s2 ) { s2++; num2chars++; }
        size2 = num2chars/sizeof(WORD)+1;
    }
    else size2 = 0;
    while ( C->Pointer+size1+size2+n+3 >= C->Top ) DoubleCbuffer(AC.cbufnum,C->Pointer,20);
    *(C->Pointer)++ = array[0];
    *(C->Pointer)++ = size1+size2+n+3;
    for ( i = 1; i < n; i++ ) *(C->Pointer)++ = array[i];
    *(C->Pointer)++ = size1;
    w = (UBYTE *)(C->Pointer);
    zeroes1 = size1*sizeof(WORD)-num1chars;
    s1 = string1;
    while ( *s1 ) { *w++ = *s1++; }
    while ( --zeroes1 >= 0 ) *w++ = 0;
    C->Pointer += size1;
    *(C->Pointer)++ = size2;
    if ( size2 ) {
        w = (UBYTE *)(C->Pointer);
        zeroes2 = size2*sizeof(WORD)-num2chars;
        s2 = string2;
        while ( *s2 ) { *w++ = *s2++; }
        while ( --zeroes2 >= 0 ) *w++ = 0;
        C->Pointer += size2;
    }
    return(0);
}

/*
    #] Add2ComStrings :
    #[ CoDiscard :
*/

int CoDiscard(UBYTE *s)
{
    if ( *s == 0 ) {
        Add2Com(TYPEDISCARD)
        return(0);
    }
    MesPrint("&Illegal argument in discard statement: '%s'",s);
    return(1);
}

/*
    #] CoDiscard :
    #[ CoContract :

    Syntax:
        Contract
```

```
        Contract:#
        Contract #
        Contract:#,#
*/

static WORD ccarray[5] = { TYPEOPERATION,5,CONTRACT,0,0 };

int CoContract(UBYTE *s)
{
    int x;
    if ( *s == ':' ) {
        s++;
        ParseNumber(x,s)
        if ( *s != ',' && *s ) {
proper:     MesPrint("&Illegal number in contract statement");
            return(1);
        }
        if ( *s ) s++;
        ccarray[4] = x;
    }
    else ccarray[4] = 0;
    if ( FG.cTable[*s] == 1 ) {
        ParseNumber(x,s)
        if ( *s ) goto proper;
        ccarray[3] = x;
    }
    else if ( *s ) goto proper;
    else ccarray[3] = -1;
    return(AddNtoL(5,ccarray));
}

/*
    #] CoContract :
    #[ CoGoTo :
*/

int CoGoTo(UBYTE *inp)
{
    UBYTE *s = inp;
    int x;
    while ( FG.cTable[*s] <= 1 ) s++;
    if ( *s ) {
        MesPrint("&Label should be an alpha-numeric string");
        return(1);
    }
    x = GetLabel(inp);
    Add3Com(TYPEGOTO,x);
    return(0);
}

/*
    #] CoGoTo :
    #[ CoLabel :
*/

int CoLabel(UBYTE *inp)
{
    UBYTE *s = inp;
    int x;
    while ( FG.cTable[*s] <= 1 ) s++;
    if ( *s ) {
        MesPrint("&Label should be an alpha-numeric string");
        return(1);
    }
    x = GetLabel(inp);
    if ( AC.Labels[x] >= 0 ) {
        MesPrint("&Label %s defined more than once",AC.LabelNames[x]);
        return(1);
    }
    AC.Labels[x] = cbuf[AC.cbufnum].numlhs;
    return(0);
}

/*
    #] CoLabel :
    #[ DoArgument :

    Layout:
        par,full size,numlhs(+1),par,scale
        scale is for normalize
*/

int DoArgument(UBYTE *s, int par)
{
    GETIDENTITY
    UBYTE *name, *t, *v, c;
    WORD *oldworkpointer = AT.WorkPointer, *w, *ww, number, *scale;
```

```
    int error = 0, zeroflag, type, x;
    AC.lhdollarflag = 0;
    while ( *s == ',' ) s++;
    w = AT.WorkPointer;
    *w++ = par;
    w++;
    switch ( par ) {
        case TYPEARG:
            if ( AC.arglevel >= MAXNEST ) {
                MesPrint("@Nesting of argument statements more than %d levels"
                ,(WORD)MAXNEST);
                return(-1);
            }
            AC.argsumcheck[AC.arglevel] = NestingChecksum();
            AC.argstack[AC.arglevel] = cbuf[AC.cbufnum].Pointer
                                    - cbuf[AC.cbufnum].Buffer + 2;
            AC.arglevel++;
            *w++ = cbuf[AC.cbufnum].numlhs;
            break;
        case TYPENORM:
        case TYPENORM4:
        case TYPESPLITARG:
        case TYPESPLITFIRSTARG:
        case TYPESPLITLASTARG:
        case TYPEFACTARG:
        case TYPEARGTOEXTRASYMBOL:
            *w++ = cbuf[AC.cbufnum].numlhs+1;
            break;
    }
    *w++ = par;
    scale = w;
    *w++ = 1;
    *w++ = 0;
    if ( *s == '^' ) {
        s++; ParseSignedNumber(x,s)
        while ( *s == ',' ) s++;
        *scale = x;
    }
    if ( *s == '(' ) {
        t = s+1; SKIPBRA3(s)    /* We did check the brackets already */
        if ( par == TYPEARG ) {
            MesPrint("&Illegal () entry in argument statement");
            error = 1; s++; goto skipbracks;
        }
        else if ( par == TYPESPLITFIRSTARG ) {
            MesPrint("&Illegal () entry in splitfirstarg statement");
            error = 1; s++; goto skipbracks;
        }
        else if ( par == TYPESPLITLASTARG ) {
            MesPrint("&Illegal () entry in splitlastarg statement");
            error = 1; s++; goto skipbracks;
        }
        v = t;
        while ( v < s ) {
            if ( *v == '?' ) {
                MesPrint("&Wildcarding not allowed in this type of statement");
                error = 1; break;
            }
            v++;
        }
        v = s++;
        if ( *t == '(' && v[-1] == ')' ) {
            t++; v--;
            if ( par == TYPESPLITARG ) oldworkpointer[0] = TYPESPLITARG2;
            else if ( par == TYPEFACTARG ) oldworkpointer[0] = TYPEFACTARG2;
            else if ( par == TYPENORM4 ) oldworkpointer[0] = TYPENORM4;
            else if ( par == TYPENORM ) {
                if ( *t == '-' ) { oldworkpointer[0] = TYPENORM3; t++; }
                else             { oldworkpointer[0] = TYPENORM2; *scale = 0; }
            }
        }
        if ( error == 0 ) {
            CBUF *C = cbuf+AC.cbufnum;
            WORD oldnumrhs = C->numrhs, oldnumlhs = C->numlhs;
            WORD prototype[SUBEXPSIZE+40]; /* Up to 10 nested sums! */
            WORD *m, *mm;
            int i, retcode;
            LONG oldpointer = C->Pointer - C->Buffer;
            *v = 0;
            prototype[0] = SUBEXPRESSION;
            prototype[1] = SUBEXPSIZE;
            prototype[2] = C->numrhs+1;
            prototype[3] = 1;
            prototype[4] = AC.cbufnum;
            AT.WorkPointer += TYPEARGHEADSIZE+1;
            AddLHS(AC.cbufnum);
            if ( ( retcode = CompileAlgebra(t,LHSIDE,prototype) ) < 0 )
```

```
                error = 1;
            else {
                prototype[2] = retcode;
                ww = C->lhs[retcode];
                AC.lhdollarflag = 0;
                if ( *ww == 0 ) {
                    *w++ = -2; *w++ = 0;
                }
                else if ( ww[ww[0]] != 0 ) {
                    MesPrint("&There should be only one term between ()");
                    error = 1;
                }
                else if ( NewSort(BHEAD0) ) { if ( !error ) error = 1; }
                else if ( NewSort(BHEAD0) ) {
                    LowerSortLevel();
                    if ( !error ) error = 1;
                }
                else {
                    AN.RepPoint = AT.RepCount + 1;
                    m = AT.WorkPointer;
                    mm = ww; i = *mm;
                    while ( --i >= 0 ) *m++ = *mm++;
                    mm = AT.WorkPointer; AT.WorkPointer = m;
                    AR.Cnumlhs = C->numlhs;
                    if ( Generator(BHEAD mm,C->numlhs) ) {
                        LowerSortLevel(); error = 1;
                    }
                    else if ( EndSort(BHEAD mm,0) < 0 ) {
                        error = 1;
                        AT.WorkPointer = mm;
                    }
                    else if ( *mm == 0 ) {
                        *w++ = -2; *w++ = 0;
                        AT.WorkPointer = mm;
                    }
                    else if ( mm[mm[0]] != 0 ) {
                        error = 1;
                        AT.WorkPointer = mm;
                    }
                    else {
                        AT.WorkPointer = mm;
                        m = mm+*mm;
                        if ( par == TYPEFACTARG ) {
                            if ( *mm != ABS(m[-1])+1 ) {
                                *mm -= ABS(m[-1]);  /* Strip coefficient */
                            }
                            mm[-1] = -*mm-1; w += *mm+1;
                        }
                        else {
                            *mm -= ABS(m[-1]);  /* Strip coefficient */
/*
                            if ( *mm == 1 ) { *w++ = -2; *w++ = 0; }
                            else
*/
                            { mm[-1] = -*mm-1; w += *mm+1; }
                        }
                        oldworkpointer[1] = w - oldworkpointer;
                    }
                    LowerSortLevel();
                }
                oldworkpointer[5] = AC.lhdollarflag;
            }
            *v = ')';
            C->numrhs = oldnumrhs;
            C->numlhs = oldnumlhs;
            C->Pointer = C->Buffer + oldpointer;
        }
    }
skipbracks:
    if ( *s == 0 ) { *w++ = 0; *w++ = 2; *w++ = 1; }
    else {
        do {
            if ( *s == ',' ) { s++; continue; }
            ww = w; *w++ = 0; w++;
            if ( FG.cTable[*s] > 1 && *s != '[' && *s != '{' ) {
                MesPrint("&Illegal parameters in statement");
                error = 1;
                break;
            }
            while ( FG.cTable[*s] == 0 || *s == '[' || *s == '{' ) {
                if ( *s == '{' ) {
                    name = s+1;
                    SKIPBRA2(s)
                    c = *s; *s = 0;
                    number = DoTempSet(name,s);
                    name--; *s++ = c; c = *s; *s = 0;
                    goto doset;
```

```
                }
                else {
                    name = s;
                    if ( ( s = SkipAName(s) ) == 0 ) {
                        MesPrint("&Illegal name '%s'",name);
                        return(1);
                    }
                    c = *s; *s = 0;
                    if ( ( type = GetName(AC.varnames,name,&number,WITHAUTO) ) == CSET ) {
doset:                  if ( Sets[number].type != CFUNCTION ) goto nofun;
                        *w++ = CSET; *w++ = number;
                    }
                    else if ( type == CFUNCTION ) {
                        *w++ = CFUNCTION; *w++ = number + FUNCTION;
                    }
                    else {
nofun:                  MesPrint("&%s is not a function or a set of functions"
                        ,name);
                        error = 1;
                    }
                }
                *s = c;
                while ( *s == ',' ) s++;
            }
            ww[1] = w - ww;
            ww = w; w++; zeroflag = 0;
            while ( FG.cTable[*s] == 1 ) {
                ParseNumber(x,s)
                if ( *s && *s != ',' ) {
                    MesPrint("&Illegal separator after number");
                    error = 1;
                    while ( *s && *s != ',' ) s++;
                }
                while ( *s == ',' ) s++;
                if ( x == 0 ) zeroflag = 1;
                if ( !zeroflag ) *w++ = (WORD)x;
            }
            *ww = w - ww;
        } while ( *s );
    }
    oldworkpointer[1] = w - oldworkpointer;
    if ( par == TYPEARG ) {  /* To make sure. The Pointer might move in the future */
        AC.argstack[AC.arglevel-1] = cbuf[AC.cbufnum].Pointer
                                   - cbuf[AC.cbufnum].Buffer + 2;
    }
    AddNtoL(oldworkpointer[1],oldworkpointer);
    AT.WorkPointer = oldworkpointer;
    return(error);
}

/*
    #] DoArgument :
    #[ CoArgument :
*/

int CoArgument(UBYTE *s) { return(DoArgument(s,TYPEARG)); }

/*
    #] CoArgument :
    #[ CoEndArgument :
*/

int CoEndArgument(UBYTE *s)
{
    CBUF *C = cbuf+AC.cbufnum;
    while ( *s == ',' ) s++;
    if ( *s ) {
        MesPrint("&Illegal syntax for EndArgument statement");
        return(1);
    }
    if ( AC.arglevel <= 0 ) {
        MesPrint("&EndArgument without corresponding Argument statement");
        return(1);
    }
    AC.arglevel--;
    cbuf[AC.cbufnum].Buffer[AC.argstack[AC.arglevel]] = C->numlhs;
    if ( AC.argsumcheck[AC.arglevel] != NestingChecksum() ) {
        MesNesting();
        return(1);
    }
    return(0);
}

/*
    #] CoEndArgument :
    #[ CoInside :
*/
```

```

int CoInside(UBYTE *s) { return(ExecInside(s)); }

/*
    #] CoInside :
    #[ CoEndInside :
*/

int CoEndInside(UBYTE *s)
{
    CBUF *C = cbuf+AC.cbufnum;
    while ( *s == ',' ) s++;
    if ( *s ) {
        MesPrint("&Illegal syntax for EndInside statement");
        return(1);
    }
    if ( AC.insidelevel <= 0 ) {
        MesPrint("&EndInside without corresponding Inside statement");
        return(1);
    }
    AC.insidelevel--;
    cbuf[AC.cbufnum].Buffer[AC.insidestack[AC.insidelevel]] = C->numlhs;
    if ( AC.insidesumcheck[AC.insidelevel] != NestingChecksum() ) {
        MesNesting();
        return(1);
    }
    return(0);
}

/*
    #] CoEndInside :
    #[ CoNormalize :
*/

int CoNormalize(UBYTE *s) { return(DoArgument(s,TYPENORM)); }

/*
    #] CoNormalize :
    #[ CoMakeInteger :
*/

int CoMakeInteger(UBYTE *s) { return(DoArgument(s,TYPENORM4)); }

/*
    #] CoMakeInteger :
    #[ CoSplitArg :
*/

int CoSplitArg(UBYTE *s) { return(DoArgument(s,TYPESPLITARG)); }

/*
    #] CoSplitArg :
    #[ CoSplitFirstArg :
*/

int CoSplitFirstArg(UBYTE *s) { return(DoArgument(s,TYPESPLITFIRSTARG)); }

/*
    #] CoSplitFirstArg :
    #[ CoSplitLastArg :
*/

int CoSplitLastArg(UBYTE *s) { return(DoArgument(s,TYPESPLITLASTARG)); }

/*
    #] CoSplitLastArg :
    #[ CoFactArg :
*/

int CoFactArg(UBYTE *s) {
    if ( ( AC.topolynomialflag & TOPOLYNOMIALFLAG ) != 0 ) {
        MesPrint("&ToPolynomial statement and FactArg statement are not allowed in the same module");
        return(1);
    }
    AC.topolynomialflag |= FACTARGFLAG;
    return(DoArgument(s,TYPEFACTARG));
}

/*
    #] CoFactArg :
    #[ DoSymmetrize :

        Syntax:
        Symmetrize Fun[:[number]] [Fields]      -> par = 0;
        AntiSymmetrize Fun[:[number]] [Fields]  -> par = 1;
        CycleSymmetrize Fun[:[number]] [Fields] -> par = 2;
        RCycleSymmetrize Fun[:[number]] [Fields]-> par = 3;
```

```
*/

int DoSymmetrize(UBYTE *s, int par)
{
    GETIDENTITY
    int extra = 0, error = 0, err, fix, x, groupsize, num, i;
    UBYTE *name, c;
    WORD funnum, *w, *ww, type;
    for(;;) {
        name = s;
        if ( ( s = SkipAName(s) ) == 0 ) {
            MesPrint("&Improper function name");
            return(1);
        }
        c = *s; *s = 0;
        if ( c != ',' || ( FG.cTable[s[1]] != 0 && s[1] != '[' ) ) break;
        if ( par <= 0 && StrICmp(name,(UBYTE *)"cyclic") == 0 ) extra = 2;
        else if ( par <= 0 && StrICmp(name,(UBYTE *)"rcyclic") == 0 ) extra = 6;
        else {
            MesPrint("&Illegal option: '%s'",name);
            error = 1;
        }
        *s++ = c;
    }
    if ( ( err = GetVar(name,&type,&funnum,CFUNCTION,WITHAUTO) ) == NAMENOTFOUND ) {
        MesPrint("&Undefined function: %s",name);
        AddFunction(name,0,0,0,0,0,-1,-1);
        *s++ = c;
        return(1);
    }
    funnum += FUNCTION;
    if ( err == -1 ) error = 1;
    *s = c;
    if ( *s == ':' ) {
        s++;
        if ( *s == ',' || *s == '(' || *s == 0 ) fix = -1;
        else if ( FG.cTable[*s] == 1 ) {
            ParseNumber(fix,s)
            if ( fix == 0 )
                Warning("Restriction to zero arguments removed");
        }
        else {
            MesPrint("&Illegal character after :");
            return(1);
        }
    }
    else fix = 0;
    w = AT.WorkPointer;
    *w++ = TYPEOPERATION;
    w++;
    *w++ = SYMMETRIZE;
    *w++ = par | extra;
    *w++ = funnum;
    *w++ = fix;
/*
    And now the argument lists. We have either ,#,#,... or (#,#,..,#),(#,...
*/
    w += 2; ww = w; groupsize = -1;
    while ( *s == ',' ) s++;
    while ( *s ) {
        if ( *s == '(' ) {
            s++; num = 0;
            while ( *s && *s != ')' ) {
                if ( *s == ',' ) { s++; continue; }
                if ( FG.cTable[*s] != 1 ) goto illarg;
                ParseNumber(x,s)
                if ( x <= 0 || ( fix > 0 && x > fix ) ) goto illnum;
                num++;
                *w++ = x-1;
            }
            if ( *s == 0 ) {
                MesPrint("&Improper termination of statement");
                return(1);
            }
            if ( groupsize < 0 ) groupsize = num;
            else if ( groupsize != num ) goto group;
            s++;
        }
        else if ( FG.cTable[*s] == 1 ) {
            if ( groupsize < 0 ) groupsize = 1;
            else if ( groupsize != 1 ) {
group:          MesPrint("&All groups should have the same number of arguments");
                return(1);
            }
            ParseNumber(x,s)
            if ( x <= 0 || ( fix > 0 && x > fix ) ) {
illnum:         MesPrint("&Illegal argument number: %d",x);
```

```
                return(1);
            }
            *w++ = x-1;
        }
        else {
illarg:     MesPrint("&Illegal argument");
            return(1);
        }
        while ( *s == ',' ) s++;
    }
/*
    Now the completion
*/
    if ( w == ww ) {
        ww[-1] = 1;
        ww[-2] = 0;
        if ( fix > 0 ) {
            for ( i = 0; i < fix; i++ ) *w++ = i;
            ww[-2] = fix; /* Bugfix 31-oct-2001. Reported by York Schroeder */
        }
    }
    else {
        ww[-1] = groupsize;
        ww[-2] = (w-ww)/groupsize;
    }
    AT.WorkPointer[1] = w - AT.WorkPointer;
    AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
    return(error);
}

/*
    #] DoSymmetrize :
    #[ CoSymmetrize :
*/

int CoSymmetrize(UBYTE *s) { return(DoSymmetrize(s,SYMMETRIC)); }

/*
    #] CoSymmetrize :
    #[ CoAntiSymmetrize :
*/

int CoAntiSymmetrize(UBYTE *s) { return(DoSymmetrize(s,ANTISYMMETRIC)); }

/*
    #] CoAntiSymmetrize :
    #[ CoCycleSymmetrize :
*/

int CoCycleSymmetrize(UBYTE *s) { return(DoSymmetrize(s,CYCLESYMMETRIC)); }

/*
    #] CoCycleSymmetrize :
    #[ CoRCycleSymmetrize :
*/

int CoRCycleSymmetrize(UBYTE *s) { return(DoSymmetrize(s,RCYCLESYMMETRIC)); }

/*
    #] CoRCycleSymmetrize :
    #[ CoWrite :
*/

int CoWrite(UBYTE *s)
{
    GETIDENTITY
    UBYTE *option;
    KEYWORD *key;
    option = s;
    if ( ( ( s = SkipAName(s) ) == 0 ) || *s != 0 ) {
        MesPrint("&Proper use of write statement is: write option");
        return(1);
    }
    key = FindInKeyWord(option,writeoptions,sizeof(writeoptions)/sizeof(KEYWORD));
    if ( key == 0 ) {
        MesPrint("&Unrecognized option in write statement");
        return(1);
    }
    *((int *)(key->func)) = key->type;
    AR.SortType = AC.SortType;
    return(0);
}

/*
    #] CoWrite :
    #[ CoNWrite :
*/
```

```
int CoNWrite(UBYTE *s)
{
    GETIDENTITY
    UBYTE *option;
    KEYWORD *key;
    option = s;
    if ( ( ( s = SkipAName(s) ) == 0 ) || *s != 0 ) {
        MesPrint("&Proper use of nwrite statement is: nwrite option");
        return(1);
    }
    key = FindInKeyWord(option,writeoptions,sizeof(writeoptions)/sizeof(KEYWORD));
    if ( key == 0 ) {
        MesPrint("&Unrecognized option in nwrite statement");
        return(1);
    }
    *((int *)(key->func)) = key->flags;
    AR.SortType = AC.SortType;
    return(0);
}

/*
    #] CoNWrite :
    #[ CoRatio :
*/

static WORD ratstring[6] = { TYPEOPERATION, 6, RATIO, 0, 0, 0 };

int CoRatio(UBYTE *s)
{
    UBYTE c, *t;
    int i, type, error = 0;
    WORD numsym, *rs;
    rs = ratstring+3;
    for ( i = 0; i < 3; i++ ) {
        if ( *s ) {
            t = s;
            s = SkipAName(s);
            c = *s; *s = 0;
            if ( ( ( type = GetName(AC.varnames,t,&numsym,WITHAUTO) ) != CSYMBOL )
            && type != CDUBIOUS ) {
                MesPrint("&%s is not a symbol",t);
                error = 4;
                if ( type < 0 ) numsym = AddSymbol(t,-MAXPOWER,MAXPOWER,0,0);
            }
            *s = c;
            if ( *s == ',' ) s++;
        }
        else {
            if ( error == 0 )
                MesPrint("&The ratio statement needs three symbols for its arguments");
            error++;
            numsym = 0;
        }
        *rs++ = numsym;
    }
    AddNtoL(6,ratstring);
    return(error);
}

/*
    #] CoRatio :
    #[ CoRedefine :

    We have a preprocessor variable and a (new) value for it.
    This value is inside a string that must be stored.
*/

int CoRedefine(UBYTE *s)
{
    UBYTE *name, c, *args = 0;
    int numprevar;
    WORD code[2];
    name = s;
    if ( FG.cTable[*s] || ( s = SkipAName(s) ) == 0 || s[-1] == '_' ) {
        MesPrint("&Illegal name for preprocessor variable in redefine statement");
        return(1);
    }
    c = *s; *s = 0;
    for ( numprevar = NumPre-1; numprevar >= 0; numprevar-- ) {
        if ( StrCmp(name,PreVar[numprevar].name) == 0 ) break;
    }
    if ( numprevar < 0 ) {
        MesPrint("&There is no preprocessor variable with the name `%s'",name);
        *s = c;
        return(1);
    }
```

```
    *s = c;
/*
    The next code worries about arguments.
    It is a direct copy of the code in TheDefine in the preprocessor.
*/
    if ( *s == '(' ) {  /* arguments. scan for correctness */
        s++; args = s;
        for (;;) {
            if ( chartype[*s] != 0 ) goto illarg;
            s++;
            while ( chartype[*s] <= 1 ) s++;
            while ( *s == ' ' || *s == '\t' ) s++;
            if ( *s == ')' ) break;
            if ( *s != ',' ) goto illargs;
            s++;
            while ( *s == ' ' || *s == '\t' ) s++;
        }
        *s++ = 0;
        while ( *s == ' ' || *s == '\t' ) s++;
    }
    while ( *s == ',' ) s++;
    if ( *s != '"' ) {
encl:   MesPrint("&Value for %s should be enclosed in double quotes"
        ,PreVar[numprevar].name);
        return(1);
    }
    s++; name = s; /* actually name points to the new string */
    while ( *s && *s != '"' ) { if ( *s == '\\' ) s++; s++; }
    if ( *s != '"' ) goto encl;
    *s = 0;
    code[0] = TYPEREDEFPRE; code[1] = numprevar;
/*
    AddComString(2,code,name,0);
*/
    Add2ComStrings(2,code,name,args);
    *s = '"';
#ifdef PARALLELCODE
/*
    Now we prepare the input numbering system for pthreads.
    We need a list of preprocessor variables that are redefined in this
    module.
*/
    {
      int j;
      WORD *newpf;
      LONG *newin;
      for ( j = 0; j < AC.numpfirstnum; j++ ) {
        if ( numprevar == AC.pfirstnum[j] ) break;
      }
      if ( j >= AC.numpfirstnum ) {  /* add to list */
        if ( j >= AC.sizepfirstnum ) {
            if ( AC.sizepfirstnum <= 0 ) { AC.sizepfirstnum = 10; }
            else { AC.sizepfirstnum = 2 * AC.sizepfirstnum; }
            newin = (LONG *)Malloc1(AC.sizepfirstnum*(sizeof(WORD)+sizeof(LONG)),"AC.pfirstnum");
            newpf = (WORD *)(newin+AC.sizepfirstnum);
            for ( j = 0; j < AC.numpfirstnum; j++ ) {
                newpf[j] = AC.pfirstnum[j];
                newin[j] = AC.inputnumbers[j];
            }
            if ( AC.inputnumbers ) M_free(AC.inputnumbers,"AC.pfirstnum");
            AC.inputnumbers = newin;
            AC.pfirstnum = newpf;
        }
        AC.pfirstnum[AC.numpfirstnum] = numprevar;
        AC.inputnumbers[AC.numpfirstnum] = -1;
        AC.numpfirstnum++;
      }
    }
#endif
    return(0);
illarg:;
    MesPrint("&Illegally formed name in argument of redefine statement");
    return(1);
illargs:;
    MesPrint("&Illegally formed arguments in redefine statement");
    return(1);
}

/*
    #] CoRedefine :
    #[ CoRenumber :

    renumber    or renumber,0     Only exchanges (n^2 until no improvement)
    renumber,1                    All permutations (could be slow)
*/

int CoRenumber(UBYTE *s)
```

```
{
    int x;
    UBYTE *inp;
    while ( *s == ',' ) s++;
    inp = s;
    if ( *s == 0 ) { x = 0; }
    else ParseNumber(x,s)
    if ( *s == 0 && x >= 0 && x <= 1 ) {
        Add3Com(TYPERENUMBER,x);
        return(0);
    }
    MesPrint("&Illegal argument in Renumber statement: '%s'",inp);
    return(1);
}

/*
    #] CoRenumber :
    #[ CoSum :
*/

int CoSum(UBYTE *s)
{
    CBUF *C = cbuf+AC.cbufnum;
    UBYTE *ss = 0, c, *t;
    int error = 0, i = 0, type, x;
    WORD numindex,number;
    while ( *s ) {
        t = s;
        if ( *s == '$' ) {
            t++; s++; while ( FG.cTable[*s] < 2 ) s++;
            c = *s; *s = 0;
            if ( ( number = GetDollar(t) ) < 0 ) {
                MesPrint("&Undefined variable $%s",t);
                if ( !error ) error = 1;
                number = AddDollar(t,0,0,0);
            }
            numindex = -number;
        }
        else {
            if ( ( s = SkipAName(s) ) == 0 ) return(1);
            c = *s; *s = 0;
            if ( ( ( type = GetOName(AC.exprnames,t,&numindex,NOAUTO) ) != NAMENOTFOUND )
            || ( ( type = GetOName(AC.varnames,t,&numindex,WITHAUTO) ) != CINDEX ) ) {
                if ( type != NAMENOTFOUND ) error = NameConflict(type,t);
                else {
                    MesPrint("&%s should have been declared as an index",t);
                    error = 1;
                    numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
                }
            }
        }
        Add3Com(TYPESUM,numindex);
        i = 3; *s = c;
        if ( *s == 0 ) break;
        if ( *s != ',' ) {
            MesPrint("&Illegal separator between objects in sum statement.");
            return(1);
        }
        s++;
        if ( FG.cTable[*s] == 0 || *s == '[' || *s == '$' ) {
            while ( FG.cTable[*s] == 0 || *s == '[' || *s == '$' ) {
                if ( *s == '$' ) {
                    s++;
                    ss = t = s;
                    while ( FG.cTable[*s] < 2 ) s++;
                    c = *s; *s = 0;
                    if ( ( number = GetDollar(t) ) < 0 ) {
                        MesPrint("&Undefined variable $%s",t);
                        if ( !error ) error = 1;
                        number = AddDollar(t,0,0,0);
                    }
                    numindex = -number;
                }
                else {
                    ss = t = s;
                    if ( ( s = SkipAName(s) ) == 0 ) return(1);
                    c = *s; *s = 0;
                    if ( ( ( type = GetOName(AC.exprnames,t,&numindex,NOAUTO) ) != NAMENOTFOUND )
                    || ( ( type = GetOName(AC.varnames,t,&numindex,WITHAUTO) ) != CINDEX ) ) {
                        if ( type != NAMENOTFOUND ) error = NameConflict(type,t);
                        else {
                            MesPrint("&%s should have been declared as an index",t);
                            error = 1;
                            numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
                        }
                    }
                }
```

```
                AddToCB(C,numindex)
                i++;
                C->Pointer[-i+1] = i;
                *s = c;
                if ( *s == 0 ) return(error);
                if ( *s != ',' ) {
                    MesPrint("&Illegal separator between objects in sum statement.");
                    return(1);
                }
                s++;
            }
            if ( FG.cTable[*s] == 1 ) {
                C->Pointer[-i+1]--; C->Pointer--; s = ss;
            }
        }
        else if ( FG.cTable[*s] == 1 ) {
            while ( FG.cTable[*s] == 1 ) {
                t = s;
                x = *s++ - '0';
                while( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
                if ( *s && *s != ',' ) {
                    MesPrint("&%s is not a legal fixed index",t);
                    return(1);
                }
                else if ( x >= AM.OffsetIndex ) {
                    MesPrint("&%d is too large to be a fixed index",x);
                    error = 1;
                }
                else {
                    AddToCB(C,x)
                    i++;
                    C->Pointer[-i] = TYPESUMFIX;
                    C->Pointer[-i+1] = i;
                }
                if ( *s == 0 ) break;
                s++;
            }
        }
        else {
            MesPrint("&Illegal object in sum statement");
            error = 1;
        }
    }
    return(error);
}

/*
    #] CoSum :
    #[ CoToTensor :
*/

static WORD cttarray[7] = { TYPEOPERATION,7,TENVEC,0,0,1,0 };

int CoToTensor(UBYTE *s)
{
    UBYTE c, *t;
    int type, j, nargs, error = 0;
    WORD number, dol[2] = { 0, 0 };
    cttarray[1] = 6;  /* length */
    cttarray[3] = 0;  /* tensor */
    cttarray[4] = 0;  /* vector */
    cttarray[5] = 1;  /* option flags */
/*  cttarray[6] = 0;     set veto */
/*
    Count the number of the arguments. The validity of them is not checked here.
*/
    nargs = 0;
    t = s;
    for (;;) {
        while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
        if ( *s == 0 ) break;
        if ( *s == '!' ) {
            s++;
            if ( *s == '{' ) {
                SKIPBRA2(s)
                s++;
            } else {
                if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
            }
        } else {
            if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
        }
        nargs++;
    }
    if ( nargs < 2 ) goto not_enough_arguments;
    s = t;
/*
```

```
    Parse options, which are given as the arguments except the last two.
*/
    for ( j = 2; j < nargs; j++ ) {
        while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
        if ( *s == '!' ) {
/*
            Handle !set or !{vector,...}. Note: If two or more sets are
            specified, then only the last one is used.
*/
            s++;
            cttarray[1] = 7;
            cttarray[5] |= 8;
            if ( FG.cTable[*s] == 0 || *s == '[' || *s == '_' ) {
                t = s;
                if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
                c = *s; *s = 0;
                type = GetName(AC.varnames,t,&number,WITHAUTO);
                if ( type == CVECTOR ) {
/*
                    As written in the manual, "!p" (without "{}") should work.
*/
                    cttarray[6] = DoTempSet(t,s);
                    *s = c;
                    goto check_tempset;
                }
                else if ( type != CSET ) {
                    MesPrint("&%s is not the name of a set or a vector",t);
                    error = 1;
                }
                *s = c;
                cttarray[6] = number;
            }
            else if ( *s == '{' ) {
                t = ++s; SKIPBRA2(s) *s = 0;
                cttarray[6] = DoTempSet(t,s);
                *s++ = '}';
check_tempset:
                if ( cttarray[6] < 0 ) {
                    error = 1;
                }
                if ( AC.wildflag ) {
                    MesPrint("&Improper use of wildcard(s) in set specification");
                    error = 1;
                }
            }
        } else {
/*
            Other options.
*/
            t = s;
            if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
            c = *s; *s = 0;
            if ( StrICmp(t,(UBYTE *)"nosquare") == 0 ) cttarray[5] |= 2;
            else if ( StrICmp(t,(UBYTE *)"functions") == 0 ) cttarray[5] |= 4;
            else {
                MesPrint("&Unrecognized option in ToTensor statement: '%s'",t);
                *s = c;
                return(1);
            }
            *s = c;
        }
    }
/*
    Now parse a vector and a tensor. The ordering doesn't matter.
*/
    for ( j = 0; j < 2; j++ ) {
        while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
        t = s;
        if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
        c = *s; *s = 0;
        if ( t[0] == '$' ) {
            dol[j] = GetDollar(t+1);
            if ( dol[j] < 0 ) dol[j] = AddDollar(t+1,DOLUNDEFINED,0,0);
        } else {
            type = GetName(AC.varnames,t,&number,WITHAUTO);
            if ( type == CVECTOR ) {
                cttarray[4] = number + AM.OffsetVector;
            }
            else if ( type == CFUNCTION && ( functions[number].spec > 0 ) ) {
                cttarray[3] = number + FUNCTION;
            }
            else {
                MesPrint("&%s is not a vector or a tensor",t);
                error = 1;
            }
        }
        *s = c;
```

```
    }
    if ( cttarray[3] == 0 || cttarray[4] == 0 ) {
        if ( dol[0] == 0 && dol[1] == 0 ) {
            goto not_enough_arguments;
        }
        else if ( cttarray[3] ) {
            if ( dol[1] )        cttarray[4] = dol[1];
            else if ( dol[0] ) { cttarray[4] = dol[0]; }
            else {
                goto not_enough_arguments;
            }
        }
        else if ( cttarray[4] ) {
            if ( dol[1] )    { cttarray[3] = -dol[1]; }
            else if ( dol[0] ) cttarray[3] = -dol[0];
            else {
                goto not_enough_arguments;
            }
        }
        else {
            if ( dol[0] == 0 || dol[1] == 0 ) {
                goto not_enough_arguments;
            }
            else {
                cttarray[3] = -dol[0]; cttarray[4] = dol[1];
            }
        }
    }
    AddNtoL(cttarray[1],cttarray);
    return(error);

syntax_error:
    MesPrint("&Syntax error in ToTensor statement");
    return(1);

not_enough_arguments:
    MesPrint("&ToTensor statement needs a vector and a tensor");
    return(1);
}

/*
    #] CoToTensor :
    #[ CoToVector :
*/

static WORD ctvarray[6] = { TYPEOPERATION,6,TENVEC,0,0,0 };

int CoToVector(UBYTE *s)
{
    UBYTE *t, c;
    int j, type, error = 0;
    WORD number, dol[2];
    dol[0] = dol[1] = 0;
    ctvarray[3] = ctvarray[4] = ctvarray[5] = 0;
    for ( j = 0; j < 2; j++ ) {
        t = s;
        if ( ( s = SkipAName(s) ) == 0 ) {
proper:     MesPrint("&Arguments of ToVector statement should be a vector and a tensor");
            return(1);
        }
        c = *s; *s = 0;
        if ( *t == '$' ) {
            dol[j] = GetDollar(t+1);
            if ( dol[j] < 0 ) dol[j] = AddDollar(t+1,DOLUNDEFINED,0,0);
        }
        else if ( ( type = GetName(AC.varnames,t,&number,WITHAUTO) ) == CVECTOR )
            ctvarray[4] = number + AM.OffsetVector;
        else if ( type == CFUNCTION && ( functions[number].spec > 0 ) )
            ctvarray[3] = number+FUNCTION;
        else {
            MesPrint("&%s is not a vector or a tensor",t);
            error = 1;
        }
        *s = c; if ( *s && *s != ',' ) goto proper;
        if ( *s ) s++;
    }
    if ( *s != 0 ) goto proper;
    if ( ctvarray[3] == 0 || ctvarray[4] == 0 ) {
        if ( dol[0] == 0 && dol[1] == 0 ) {
            MesPrint("&ToVector statement needs a vector and a tensor");
            error = 1;
        }
        else if ( ctvarray[3] ) {
            if ( dol[1] )      ctvarray[4] = dol[1];
            else if ( dol[0] ) ctvarray[4] = dol[0];
            else {
                MesPrint("&ToVector statement needs a vector and a tensor");
```

```
                error = 1;
            }
        }
        else if ( ctvarray[4] ) {
            if ( dol[1] )      ctvarray[3] = -dol[1];
            else if ( dol[0] ) ctvarray[3] = -dol[0];
            else {
                MesPrint("&ToVector statement needs a vector and a tensor");
                error = 1;
            }
        }
        else {
            if ( dol[0] == 0 || dol[1] == 0 ) {
                MesPrint("&ToVector statement needs a vector and a tensor");
                error = 1;
            }
            else {
                ctvarray[3] = -dol[0]; ctvarray[4] = dol[1];
            }
        }
    }
    AddNtoL(6,ctvarray);
    return(error);
}

/*
    #] CoToVector :
    #[ CoTrace4 :
*/

int CoTrace4(UBYTE *s)
{
    int error = 0, type, option = CHISHOLM;
    UBYTE *t, c;
    WORD numindex, one = 1;
    KEYWORD *key;
    for (;;) {
        t = s;
        if ( FG.cTable[*s] == 1 ) break;
        if ( ( s = SkipAName(s) ) == 0 ) {
proper:     MesPrint("&Proper syntax for Trace4 is 'Trace4[,options],index;'");
            return(1);
        }
        if ( *s == 0 ) break;
        c = *s; *s = 0;
        if ( ( key = FindKeyWord(t,trace4options,
            sizeof(trace4options)/sizeof(KEYWORD)) ) == 0 ) break;
        else {
            option |=  key->type;
            option &= ~key->flags;
        }
        if ( ( *s++ = c ) != ',' ) {
            MesPrint("&Illegal separator in Trace4 statement");
            return(1);
        }
        if ( *s == 0 ) goto proper;
    }
    s = t;
    if ( FG.cTable[*s] == 1 ) {
retry:
        ParseNumber(numindex,s)
        if ( *s != 0 ) {
            MesPrint("&Last argument of Trace4 should be an index");
            return(1);
        }
        if ( numindex >= AM.OffsetIndex ) {
            MesPrint("&fixed index >= %d. Change value of OffsetIndex in setup file"
            ,AM.OffsetIndex);
            return(1);
        }
    }
    else if ( *s == '$' ) {
        if ( ( type = GetName(AC.dollarnames,s+1,&numindex,NOAUTO) ) == CDOLLAR )
            numindex = -numindex;
        else {
            MesPrint("&%s is undefined",s);
            numindex = AddDollar(s+1,DOLINDEX,&one,1);
            return(1);
        }
tests:  s = SkipAName(s);
        if ( *s != 0 ) {
            MesPrint("&Trace4 should have a single index or $variable for its argument");
            return(1);
        }
    }
    else if ( ( type = GetName(AC.varnames,s,&numindex,WITHAUTO) ) == CINDEX ) {
        numindex += AM.OffsetIndex;
```

```
        goto tests;
    }
    else if ( type != -1 ) {
        if ( type != CDUBIOUS ) {
            if ( ( FG.cTable[*s] != 0 ) && ( *s != '[' ) ) {
                if ( *s == '+' && FG.cTable[s[1]] == 1 ) { s++; goto retry; }
                goto proper;
            }
            NameConflict(type,s);
            type = MakeDubious(AC.varnames,s,&numindex);
        }
        return(1);
    }
    else {
        MesPrint("&%s is not an index",s);
        numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
        return(1);
    }
    if ( error ) return(error);
    if ( ( option & CHISHOLM ) != 0 )
        Add4Com(TYPECHISHOLM,numindex,(option & ALSOREVERSE));
    Add5Com(TYPEOPERATION,TAKETRACE,4 + (option & NOTRICK),numindex);
    return(0);
}

/*
    #] CoTrace4 :
    #[ CoTraceN :
*/

int CoTraceN(UBYTE *s)
{
    WORD numindex, one = 1;
    int type;
    if ( FG.cTable[*s] == 1 ) {
retry:
        ParseNumber(numindex,s)
        if ( *s != 0 ) {
proper:     MesPrint("&TraceN should have a single index for its argument");
            return(1);
        }
        if ( numindex >= AM.OffsetIndex ) {
            MesPrint("&fixed index >= %d. Change value of OffsetIndex in setup file"
            ,AM.OffsetIndex);
            return(1);
        }
    }
    else if ( *s == '$' ) {
        if ( ( type = GetName(AC.dollarnames,s+1,&numindex,NOAUTO) ) == CDOLLAR )
            numindex = -numindex;
        else {
            MesPrint("&%s is undefined",s);
            numindex = AddDollar(s+1,DOLINDEX,&one,1);
            return(1);
        }
tests:  s = SkipAName(s);
        if ( *s != 0 ) {
            MesPrint("&TraceN should have a single index or $variable for its argument");
            return(1);
        }
    }
    else if ( ( type = GetName(AC.varnames,s,&numindex,WITHAUTO) ) == CINDEX ) {
        numindex += AM.OffsetIndex;
        goto tests;
    }
    else if ( type != -1 ) {
        if ( type != CDUBIOUS ) {
            if ( ( FG.cTable[*s] != 0 ) && ( *s != '[' ) ) {
                if ( *s == '+' && FG.cTable[s[1]] == 1 ) { s++; goto retry; }
                goto proper;
            }
            NameConflict(type,s);
            type = MakeDubious(AC.varnames,s,&numindex);
        }
        return(1);
    }
    else {
        MesPrint("&%s is not an index",s);
        numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
        return(1);
    }
    Add5Com(TYPEOPERATION,TAKETRACE,0,numindex);
    return(0);
}

/*
    #] CoTraceN :
```

```
    #[ CoChisholm :
*/

int CoChisholm(UBYTE *s)
{
    int error = 0, type, option = CHISHOLM;
    UBYTE *t, c;
    WORD numindex, one = 1;
    KEYWORD *key;
    for (;;) {
        t = s;
        if ( FG.cTable[*s] == 1 ) break;
        if ( ( s = SkipAName(s) ) == 0 ) {
proper:     MesPrint("&Proper syntax for Chisholm is 'Chisholm[,options],index;'");
            return(1);
        }
        if ( *s == 0 ) break;
        c = *s; *s = 0;
        if ( ( key = FindKeyWord(t,chisoptions,
            sizeof(chisoptions)/sizeof(KEYWORD)) ) == 0 ) break;
        else {
            option |=  key->type;
            option &= ~key->flags;
        }
        if ( ( *s++ = c ) != ',' ) {
            MesPrint("&Illegal separator in Chisholm statement");
            return(1);
        }
        if ( *s == 0 ) goto proper;
    }
    s = t;
    if ( FG.cTable[*s] == 1 ) {
        ParseNumber(numindex,s)
        if ( *s != 0 ) {
            MesPrint("&Last argument of Chisholm should be an index");
            return(1);
        }
        if ( numindex >= AM.OffsetIndex ) {
            MesPrint("&fixed index >= %d. Change value of OffsetIndex in setup file"
            ,AM.OffsetIndex);
            return(1);
        }
    }
    else if ( *s == '$' ) {
        if ( ( type = GetName(AC.dollarnames,s+1,&numindex,NOAUTO) ) == CDOLLAR )
            numindex = -numindex;
        else {
            MesPrint("&%s is undefined",s);
            numindex = AddDollar(s+1,DOLINDEX,&one,1);
            return(1);
        }
tests:  s = SkipAName(s);
        if ( *s != 0 ) {
            MesPrint("&Chisholm should have a single index or $variable for its argument");
            return(1);
        }
    }
    else if ( ( type = GetName(AC.varnames,s,&numindex,WITHAUTO) ) == CINDEX ) {
        numindex += AM.OffsetIndex;
        goto tests;
    }
    else if ( type != -1 ) {
        if ( type != CDUBIOUS ) {
            NameConflict(type,s);
            type = MakeDubious(AC.varnames,s,&numindex);
        }
        return(1);
    }
    else {
        MesPrint("&%s is not an index",s);
        numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
        return(1);
    }
    if ( error ) return(error);
    Add4Com(TYPECHISHOLM,numindex,(option & ALSOREVERSE));
    return(0);
}

/*
    #] CoChisholm :
    #[ DoChain :

    Syntax: Chainxx functionname;
*/

int DoChain(UBYTE *s, int option)
{
```

```
    WORD numfunc,type;
    if ( *s == '$' ) {
        if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
            numfunc = -numfunc;
        else {
            MesPrint("&%s is undefined",s);
            numfunc = AddDollar(s+1,DOLINDEX,&one,1);
            return(1);
        }
tests:  s = SkipAName(s);
        if ( *s != 0 ) {
            MesPrint("&ChainIn/ChainOut should have a single function or $variable for its argument");
            return(1);
        }
    }
    else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
        numfunc += FUNCTION;
        goto tests;
    }
    else if ( type != -1 ) {
        if ( type != CDUBIOUS ) {
            NameConflict(type,s);
            type = MakeDubious(AC.varnames,s,&numfunc);
        }
        return(1);
    }
    else {
        MesPrint("&%s is not a function",s);
        numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
        return(1);
    }
    Add3Com(option,numfunc);
    return(0);
}

/*
    #] DoChain :
    #[ CoChainin :

    Syntax: Chainin functionname;
*/

int CoChainin(UBYTE *s)
{
    return(DoChain(s,TYPECHAININ));
}

/*
    #] CoChainin :
    #[ CoChainout :

    Syntax: Chainout functionname;
*/

int CoChainout(UBYTE *s)
{
    return(DoChain(s,TYPECHAINOUT));
}

/*
    #] CoChainout :
    #[ CoExit :
*/

int CoExit(UBYTE *s)
{
    UBYTE *name;
    WORD code = TYPEEXIT;
    while ( *s == ',' ) s++;
    if ( *s == 0 ) {
        Add3Com(TYPEEXIT,0);
        return(0);
    }
    name = s+1;
    s++;
    while ( *s ) { if ( *s == '\\' ) s++; s++; }
    if ( name[-1] != '"' || s[-1] != '"' ) {
        MesPrint("&Illegal syntax for exit statement");
        return(1);
    }
    s[-1] = 0;
    AddComString(1,&code,name,0);
    s[-1] = '"';
    return(0);
}

/*
```

```
    #] CoExit :
    #[ CoInParallel :
*/

int CoInParallel(UBYTE *s)
{
    return(DoInParallel(s,1));
}

/*
    #] CoInParallel :
    #[ CoNotInParallel :
*/

int CoNotInParallel(UBYTE *s)
{
    return(DoInParallel(s,0));
}

/*
    #] CoNotInParallel :
    #[ DoInParallel :

    InParallel;
    InParallel,names;
    NotInParallel;
    NotInParallel,names;
*/

int DoInParallel(UBYTE *s, int par)
{
#ifdef PARALLELCODE
    EXPRESSIONS e;
    WORD i;
#endif
    WORD number;
    UBYTE *t, c;
    int error = 0;
#ifndef WITHPTHREADS
    DUMMYUSE(par);
#endif
    if ( *s == 0 ) {
        AC.inparallelflag = par;
#ifdef PARALLELCODE
        for ( i = NumExpressions-1; i >= 0; i-- ) {
            e = Expressions+i;
            if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
            || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
            ) {
                e->partodo = par;
            }
        }
#endif
    }
    else {
        for(;;) {   /* Look for a (comma separated) list of variables */
            while ( *s == ',' ) s++;
            if ( *s == 0 ) break;
            if ( *s == '[' || FG.cTable[*s] == 0 ) {
                t = s;
                if ( ( s = SkipAName(s) ) == 0 ) {
                    MesPrint("&Improper name for an expression: '%s'",t);
                    return(1);
                }
                c = *s; *s = 0;
                if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
#ifdef PARALLELCODE
                    e = Expressions+number;
                    if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
                    || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
                    ) {
                        e->partodo = par;
                    }
#endif
                }
                else if ( GetName(AC.varnames,t,&number,NOAUTO) != NAMENOTFOUND ) {
                    MesPrint("&%s is not an expression",t);
                    error = 1;
                }
                *s = c;
            }
            else {
                MesPrint("&Illegal object in InExpression statement");
                error = 1;
                while ( *s && *s != ',' ) s++;
                if ( *s == 0 ) break;
            }
```

```
        }

    }
    return(error);
}

/*
    #] DoInParallel :
    #[ CoInExpression :
*/

int CoInExpression(UBYTE *s)
{
    GETIDENTITY
    UBYTE *t, c;
    WORD *w, number;
    int error = 0;
    w = AT.WorkPointer;
    if ( AC.inexprlevel >= MAXNEST ) {
        MesPrint("@Nesting of inexpression statements more than %d levels",(WORD)MAXNEST);
        return(-1);
    }
    AC.inexprsumcheck[AC.inexprlevel] = NestingChecksum();
    AC.inexprstack[AC.inexprlevel] = cbuf[AC.cbufnum].Pointer
                                - cbuf[AC.cbufnum].Buffer + 2;
    AC.inexprlevel++;
    *w++ = TYPEINEXPRESSION;
    w++; w++;
    for(;;) {   /* Look for a (comma separated) list of variables */
        while ( *s == ',' ) s++;
        if ( *s == 0 ) break;
        if ( *s == '[' || FG.cTable[*s] == 0 ) {
            t = s;
            if ( ( s = SkipAName(s) ) == 0 ) {
                MesPrint("&Improper name for an expression: '%s'",t);
                return(1);
            }
            c = *s; *s = 0;
            if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
                *w++ = number;
            }
            else if ( GetName(AC.varnames,t,&number,NOAUTO) != NAMENOTFOUND ) {
                MesPrint("&%s is not an expression",t);
                error = 1;
            }
            *s = c;
        }
        else {
            MesPrint("&Illegal object in InExpression statement");
            error = 1;
            while ( *s && *s != ',' ) s++;
            if ( *s == 0 ) break;
        }
    }
    AT.WorkPointer[1] = w - AT.WorkPointer;
    AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
    return(error);
}

/*
    #] CoInExpression :
    #[ CoEndInExpression :
*/

int CoEndInExpression(UBYTE *s)
{
    CBUF *C = cbuf+AC.cbufnum;
    while ( *s == ',' ) s++;
    if ( *s ) {
        MesPrint("&Illegal syntax for EndInExpression statement");
        return(1);
    }
    if ( AC.inexprlevel <= 0 ) {
        MesPrint("&EndInExpression without corresponding InExpression statement");
        return(1);
    }
    AC.inexprlevel--;
    cbuf[AC.cbufnum].Buffer[AC.inexprstack[AC.inexprlevel]] = C->numlhs;
    if ( AC.inexprsumcheck[AC.inexprlevel] != NestingChecksum() ) {
        MesNesting();
        return(1);
    }
    return(0);
}

/*
    #] CoEndInExpression :
```

```
    #[ CoSetExitFlag :
*/

int CoSetExitFlag(UBYTE *s)
{
    if ( *s ) {
        MesPrint("&Illegal syntax for the SetExitFlag statement");
        return(1);
    }
    Add2Com(TYPESETEXIT);
    return(0);
}

/*
    #] CoSetExitFlag :
    #[ CoTryReplace :
*/
int CoTryReplace(UBYTE *p)
{
    GETIDENTITY
    UBYTE *name, c;
    WORD *w, error = 0, i, which = -1, c1, minvec = 0;
    w = AT.WorkPointer;
    *w++ = TYPETRY;
    *w++ = 3;
    *w++ = 0;
    *w++ = REPLACEMENT;
    *w++ = FUNHEAD;
    FILLFUN(w)
/*
    Now we have to read a function argument for the replace_ function.
    Current arguments that we allow involve only single arguments
    that do not expand further. No brackets!
*/
    while ( *p ) {
/*
        No numbers yet
*/
        if ( *p == '-' && minvec == 0 && which == (CVECTOR+1) ) {
            minvec = 1; p++;
        }
        if ( *p == '[' || FG.cTable[*p] == 0 ) {
            name = p;
            if ( ( p = SkipAName(p) )  == 0 ) return(1);
            c = *p; *p = 0;
            i = GetName(AC.varnames,name,&c1,WITHAUTO);
            if ( which >= 0 && i >= 0 && i != CDUBIOUS && which != (i+1) ) {
                MesPrint("&Illegal combination of objects in TryReplace");
                error = 1;
            }
            else if ( minvec && i != CVECTOR && i != CDUBIOUS ) {
                MesPrint("&Currently a - sign can be used only with a vector in TryReplace");
                error = 1;
            }
            else switch ( i ) {
                case CSYMBOL: *w++ = -SYMBOL; *w++ = c1; break;
                case CVECTOR:
                    if ( minvec ) *w++ = -MINVECTOR;
                    else          *w++ = -VECTOR;
                    *w++ = c1 + AM.OffsetVector;
                    minvec = 0;
                    break;
                case CINDEX: *w++ = -INDEX; *w++ = c1 + AM.OffsetIndex;
                    if ( c1 >= AM.WilInd && c == '?' ) { *p++ = c; c = *p; }
                    break;
                case CFUNCTION: *w++ = -c1-FUNCTION; break;
                case CDUBIOUS: minvec = 0; error = 1; break;
                default:
                    MesPrint("&Illegal object type in TryReplace: %s",name);
                    error = 1;
                    i = 0;
                    break;
            }
            if ( which < 0 ) which = i+1;
            else which = -1;
            *p = c;
            if ( *p == ',' ) p++;
            continue;
        }
        else {
            MesPrint("&Illegal object in TryReplace");
            error = 1;
            while ( *p && *p != ',' ) {
                if ( *p == '(' ) SKIPBRA3(p)
                else if ( *p == '{' ) SKIPBRA2(p)
                else if ( *p == '[' ) SKIPBRA1(p)
                else p++;
```

```
            }
        }
        if ( *p == ',' ) p++;
        if ( which < 0 ) which = 0;
        else which = -1;
    }
    if ( which >= 0 ) {
        MesPrint("&Odd number of arguments in TryReplace");
        error = 1;
    }
    i = w - AT.WorkPointer;
    AT.WorkPointer[1] = i;
    AT.WorkPointer[2] = i - 3;
    AT.WorkPointer[4] = i - 3;
    AddNtoL((int)i,AT.WorkPointer);
    return(error);
}

/*
    #] CoTryReplace :
    #[ CoModulus :

    Old syntax:  Modulus [-] number [:number]
    New syntax:  Modulus [option(s)] number
        Options are: NoFunctions/CoefficientsOnly/AlsoFunctions
                     PlusMin/Positive
                     InverseTable
                     PrintPowersOf(number)
                     AlsoPowers/NoPowers
                     AlsoDollars/NoDollars
    Notice: We change the defaults. This may cause problems to some.
*/

int CoModulus(UBYTE *inp)
{
#ifdef OLDMODULUS
/*  #[ Old Syntax : */
    UBYTE *p, c;
    WORD sign = 1, Retval;
    while ( *inp == '-' || *inp == '+' ) {
        if ( *inp == '-' ) sign = -sign;
        inp++;
    }
    p = inp;
    if ( FG.cTable[*inp] != 1 ) {
        MesPrint("&Invalid value for modulus:%s",inp);
        if ( AC.modpowers ) M_free(AC.modpowers,"AC.modpowers");
        AC.modpowers = 0;
        return(1);
    }
    do { inp++; } while ( FG.cTable[*inp] == 1 );
    c = *inp; *inp = 0;
    Retval = GetLong(p,(UWORD *)AC.cmod,&AC.ncmod);
    if ( sign < 0 ) AC.ncmod = -AC.ncmod;
    *p = c;
    if ( c == 0 ) goto regular;
    else if ( c != ':' ) {
        MesPrint("&Illegal option for modulus %s",inp);
        if ( AC.modpowers ) M_free(AC.modpowers,"AC.modpowers");
        AC.modpowers = 0;
        return(1);
    }
    inp++;
    p = inp;
    while ( FG.cTable[*inp] == 1 ) inp++;
    if ( *inp ) {
        MesPrint("&Illegal character in option for modulus %s",inp);
        if ( AC.modpowers ) M_free(AC.modpowers,"AC.modpowers");
        AC.modpowers = 0;
        return(1);
    }
    if ( GetLong(p,(UWORD *)AC.powmod,&AC.npowmod) ) Retval = -1;
    if ( TakeModulus((UWORD *)AC.powmod,&AC.npowmod,AC.cmod,AC.ncmod,NOUNPACK) ) Retval = -1;
    if ( AC.npowmod == 0 ) {
        MesPrint("&Improper value for generator");
        Retval = -1;
    }
    if ( MakeModTable() ) Retval = -1;
    AC.DirtPow = 1;
regular:
    AN.ncmod = AC.ncmod;
    if ( AC.halfmod ) {
        M_free(AC.halfmod,"halfmod");
        AC.halfmod = 0; AC.nhalfmod = 0;
    }
    if ( AC.modinverses ) {
        M_free(AC.halfmod,"modinverses");
```

```
        AC.modinverses = 0;
    }
    return(Retval);
/*  #] Old Syntax : */
#else
    GETIDENTITY
    int Retval = 0, sign = 1;
    UBYTE *p, c;
    while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
    if ( *inp == 0 ) {
SwitchOff:
        if ( AC.modpowers ) M_free(AC.modpowers,"AC.modpowers");
        AC.modpowers = 0;
        AN.ncmod = AC.ncmod = 0;
        if ( AC.halfmod ) M_free(AC.halfmod,"halfmod");
        AC.halfmod = 0; AC.nhalfmod = 0;
        if ( AC.modinverses ) M_free(AC.modinverses,"modinverses");
        AC.modinverses = 0;
        AC.modmode = 0;
        return(0);
    }
    AC.modmode = 0;
    if ( *inp == '-' ) {
        sign = -1;
        inp++;
    }
    else {
      while ( FG.cTable[*inp] == 0 ) {
        p = inp;
        while ( FG.cTable[*inp] == 0 ) inp++;
        c = *inp; *inp = 0;
        if ( StrICmp(p,(UBYTE *)"nofunctions") == 0 ) {
            AC.modmode &= ~ALSOFUNARGS;
        }
        else if ( StrICmp(p,(UBYTE *)"alsofunctions") == 0 ) {
            AC.modmode |= ALSOFUNARGS;
        }
        else if ( StrICmp(p,(UBYTE *)"coefficientsonly") == 0 ) {
            AC.modmode &= ~ALSOFUNARGS;
            AC.modmode &= ~ALSOPOWERS;
            sign = -1;
        }
        else if ( StrICmp(p,(UBYTE *)"plusmin") == 0 ) {
            AC.modmode |= POSNEG;
        }
        else if ( StrICmp(p,(UBYTE *)"positive") == 0 ) {
            AC.modmode &= ~POSNEG;
        }
        else if ( StrICmp(p,(UBYTE *)"inversetable") == 0 ) {
            AC.modmode |= INVERSETABLE;
        }
        else if ( StrICmp(p,(UBYTE *)"noinversetable") == 0 ) {
            AC.modmode &= ~INVERSETABLE;
        }
        else if ( StrICmp(p,(UBYTE *)"nodollars") == 0 ) {
            AC.modmode &= ~ALSODOLLARS;
        }
        else if ( StrICmp(p,(UBYTE *)"alsodollars") == 0 ) {
            AC.modmode |= ALSODOLLARS;
        }
        else if ( StrICmp(p,(UBYTE *)"printpowersof") == 0 ) {
            *inp = c;
            if ( *inp != '(' ) {
badsyntax:
                MesPrint("&Bad syntax in argument of PrintPowersOf(number) in Modulus statement");
                return(1);
            }
            while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
            inp++; p = inp;
            if ( FG.cTable[*inp] != 1 ) goto badsyntax;
            do { inp++; } while ( FG.cTable[*inp] == 1 );
            c = *inp; *inp = 0;
            if ( GetLong(p,(UWORD *)AC.powmod,&AC.npowmod) ) Retval = -1;
            if ( TakeModulus((UWORD *)AC.powmod,&AC.npowmod,AC.cmod,AC.ncmod,NOUNPACK) ) Retval = -1;
            if ( AC.npowmod == 0 ) {
                MesPrint("&Improper value for generator");
                Retval = -1;
            }
            if ( MakeModTable() ) Retval = -1;
            AC.DirtPow = 1;
            *inp = c;
            while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
            if ( *inp != ')' ) goto badsyntax;
            inp++;
            c = *inp;
        }
        else if ( StrICmp(p,(UBYTE *)"alsopowers") == 0 ) {
```

```
            AC.modmode |= ALSOPOWERS;
            sign =  1;
        }
        else if ( StrICmp(p,(UBYTE *)"nopowers") == 0 ) {
            AC.modmode &= ~ALSOPOWERS;
            sign = -1;
        }
        else {
            MesPrint("&Unrecognized option %s in Modulus statement",inp);
            return(1);
        }
        *inp = c;
        while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
        if ( *inp == 0 ) {
            MesPrint("&Modulus statement with no value!!!");
            return(1);
        }
      }
    }
    p = inp;
    if ( FG.cTable[*inp] != 1 ) {
        MesPrint("&Invalid value for modulus:%s",inp);
        if ( AC.modpowers ) M_free(AC.modpowers,"AC.modpowers");
        AC.modpowers = 0;
        AN.ncmod = AC.ncmod = 0;
        if ( AC.halfmod ) M_free(AC.halfmod,"halfmod");
        AC.halfmod = 0; AC.nhalfmod = 0;
        if ( AC.modinverses ) M_free(AC.modinverses,"modinverses");
        AC.modinverses = 0;
        return(1);
    }
    do { inp++; } while ( FG.cTable[*inp] == 1 );
    c = *inp; *inp = 0;
    Retval = GetLong(p,(UWORD *)AC.cmod,&AC.ncmod);
    if ( Retval == 0 && AC.ncmod == 0 ) goto SwitchOff;
    if ( sign < 0 ) AC.ncmod = -AC.ncmod;
    AN.ncmod = AC.ncmod;
    if ( ( AC.modmode & INVERSETABLE ) != 0 ) MakeInverses();
    if ( AC.halfmod ) M_free(AC.halfmod,"halfmod");
    AC.halfmod = 0; AC.nhalfmod = 0;
    return(Retval);
#endif
}

/*
    #] CoModulus :
    #[ CoRepeat :
*/

int CoRepeat(UBYTE *inp)
{
    int error = 0;
    AC.RepSumCheck[AC.RepLevel] = NestingChecksum();
    AC.RepLevel++;
    if ( AC.RepLevel > AM.RepMax ) {
        MesPrint("&Too many repeat levels. Maximum is %d",AM.RepMax);
        return(1);
    }
    Add3Com(TYPEREPEAT,-1)   /* Means indefinite */
    while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
    if ( *inp ) {
        error = CompileStatement(inp);
        if ( CoEndRepeat(inp) ) error = 1;
    }
    return(error);
}

/*
    #] CoRepeat :
    #[ CoEndRepeat :
*/

int CoEndRepeat(UBYTE *inp)
{
    CBUF *C = cbuf+AC.cbufnum;
    int level, error = 0, repeatlevel = 0;
    DUMMYUSE(inp);
    AC.RepLevel--;
    if ( AC.RepLevel < 0 ) {
        MesPrint("&EndRepeat without Repeat");
        AC.RepLevel = 0;
        return(1);
    }
    else if ( AC.RepSumCheck[AC.RepLevel] != NestingChecksum() ) {
        MesNesting();
        error = 1;
    }
```

```
    level = C->numlhs+1;
    while ( level > 0 ) {
        if ( C->lhs[--level][0] == TYPEREPEAT ) {
            if ( repeatlevel == 0 ) {
                Add3Com(TYPEENDREPEAT,level)
                return(error);
            }
            repeatlevel--;
        }
        else if ( C->lhs[level][0] == TYPEENDREPEAT ) repeatlevel++;
    }
    return(1);
}

/*
    #] CoEndRepeat :
    #[ DoBrackets :

        Reads in the bracket information.
        Storage is in the form of a regular term.
        No subterms and arguments are allowed.
*/

int DoBrackets(UBYTE *inp, int par)
{
    GETIDENTITY
    UBYTE *p, *pp, c;
    WORD *to, i, type, *w, error = 0;
    WORD c1,c2, *WorkSave;
    int biflag;
    p = inp;
    WorkSave = to = AT.WorkPointer;
    to++;
    if ( AT.BrackBuf == 0 ) {
        AR.MaxBracket = 100;
        AT.BrackBuf = (WORD *)Malloc1(sizeof(WORD)*(AR.MaxBracket+1),"bracket buffer");
    }
    *AT.BrackBuf = 0;
    AR.BracketOn = 0;
    AC.bracketindexflag = 0;
    AT.bracketindexflag = 0;
    if ( *p == '+' || *p == '-' ) p++;
    if ( p[-1] == ',' && *p ) p--;
    if ( p[-1] == '+' && *p ) { biflag = 1;  if ( *p != ',' ) { *--p = ','; } }
    else if ( p[-1] == '-' && *p ) { biflag = -1; if ( *p != ',' ) { *--p = ','; } }
    else biflag = 0;
    while ( *p == ',' ) {
redo:   AR.BracketOn++;
        while ( *p == ',' ) p++;
        if ( *p == 0 ) break;
        if ( *p == '0' ) {
            p++; while ( *p == '0' ) p++;
            continue;
        }
        inp = pp = p;
        p = SkipAName(p);
        if ( p == 0 ) return(1);
        c = *p;
        *p = 0;
        type = GetName(AC.varnames,inp,&c1,WITHAUTO);
        if ( c == '.' ) {
            if ( type == CVECTOR || type == CDUBIOUS ) {
                *p++ = c;
                inp = p;
                p = SkipAName(p);
                if ( p == 0 ) return(1);
                c = *p;
                *p = 0;
                type = GetName(AC.varnames,inp,&c2,WITHAUTO);
                if ( type != CVECTOR && type != CDUBIOUS ) {
                    MesPrint("&Not a vector in dotproduct in bracket statement: %s",inp);
                    error = 1;
                }
                else type = CDOTPRODUCT;
            }
            else {
                MesPrint("&Illegal use of . after %s in bracket statement",inp);
                error = 1;
                *p++ = c;
                goto redo;
            }
        }
        switch ( type ) {
            case CSYMBOL :
                *to++ = SYMBOL; *to++ = 4; *to++ = c1; *to++ = 1; break;
            case CVECTOR :
                *to++ = INDEX; *to++ = 3; *to++ = AM.OffsetVector + c1; break;
```

```
            case CFUNCTION :
                *to++ = c1+FUNCTION; *to++ = FUNHEAD; *to++ = 0;
                FILLFUN3(to)
                break;
            case CDOTPRODUCT :
                *to++ = DOTPRODUCT; *to++ = 5; *to++ = c1 + AM.OffsetVector;
                *to++ = c2 + AM.OffsetVector; *to++ = 1; break;
            case CDELTA :
                *to++ = DELTA; *to++ = 4; *to++ = EMPTYINDEX; *to++ = EMPTYINDEX; break;
            case CSET :
                *to++ = SETSET; *to++ = 4; *to++ = c1; *to++ = Sets[c1].type; break;
            default :
                MesPrint("&Illegal bracket request for %s",pp);
                error = 1; break;
        }
        *p = c;
    }
    if ( *p ) {
        MesCerr("separator",p);
        AC.BracketNormalize = 0;
        AT.WorkPointer = WorkSave;
        error = 1;
        return(error);
    }
    *to++ = 1; *to++ = 1; *to++ = 3;
    *AT.WorkPointer = to - AT.WorkPointer;
    AT.WorkPointer = to;
    AC.BracketNormalize = 1;
    if ( BracketNormalize(BHEAD WorkSave) ) { error = 1; AR.BracketOn = 0; }
    else {
        w = WorkSave;
        if ( *w == 4 || !*w ) { AR.BracketOn = 0; }
        else {
            i = *(w+*w-1);
            if ( i < 0 ) i = -i;
            *w -= i;
            i = *w;
            if ( i > AR.MaxBracket ) {
                WORD *newbuf;
                newbuf = (WORD *)Malloc1(sizeof(WORD)*(i+1),"bracket buffer");
                AR.MaxBracket = i;
                if ( AT.BrackBuf != 0 ) M_free(AT.BrackBuf,"bracket buffer");
                AT.BrackBuf = newbuf;
            }
            to = AT.BrackBuf;
            NCOPY(to,w,i);
        }
    }
    AC.BracketNormalize = 0;
    if ( par == 1 ) AR.BracketOn = -AR.BracketOn;
    if ( error == 0 ) {
        AC.bracketindexflag = biflag;
        AT.bracketindexflag = biflag;
    }
    AT.WorkPointer = WorkSave;
    return(error);
}

/*
    #] DoBrackets :
    #[ CoBracket :
*/

int CoBracket(UBYTE *inp)
{ return(DoBrackets(inp,0)); }

/*
    #] CoBracket :
    #[ CoAntiBracket :
*/

int CoAntiBracket(UBYTE *inp)
{ return(DoBrackets(inp,1)); }

/*
    #] CoAntiBracket :
    #[ CoMultiBracket :

    Syntax:
        MultiBracket:{A|B} bracketinfo:...:{A|B} bracketinfo;
*/

int CoMultiBracket(UBYTE *inp)
{
    GETIDENTITY
    int i, error = 0, error1, type, num;
    UBYTE *s, c;
```

```
    WORD *to, *from;

    if ( *inp != ':' ) {
        MesPrint("&Illegal Multiple Bracket separator: %s",inp);
        return(1);
    }
    inp++;
    if ( AC.MultiBracketBuf == 0 ) {
        AC.MultiBracketBuf = (WORD **)Malloc1(sizeof(WORD *)*MAXMULTIBRACKETLEVELS,"multi bracket buffer");
        for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
            AC.MultiBracketBuf[i] = 0;
        }
    }
    else {
      for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
        if ( AC.MultiBracketBuf[i] ) {
            M_free(AC.MultiBracketBuf[i],"bracket buffer i");
            AC.MultiBracketBuf[i] = 0;
        }
      }
      AC.MultiBracketLevels = 0;
    }
    AC.MultiBracketLevels = 0;
/*
        Start with disabling the regular brackets.
*/
    if ( AT.BrackBuf == 0 ) {
        AR.MaxBracket = 100;
        AT.BrackBuf = (WORD *)Malloc1(sizeof(WORD)*(AR.MaxBracket+1),"bracket buffer");
    }
    *AT.BrackBuf = 0;
    AR.BracketOn = 0;
    AC.bracketindexflag = 0;
    AT.bracketindexflag = 0;
/*
    Now loop through the various levels, separated by the colons.
*/
    for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
        if ( *inp == 0 ) goto RegEnd;
/*
        1: skip to ':', determine bracket or antibracket
*/
        s = inp;
        while ( *s && *s != ':' ) {
            if ( *s == '[' ) { SKIPBRA1(s) s++; }
            else if ( *s == '{' ) { SKIPBRA2(s) s++; }
            else s++;
        }
        c = *s; *s = 0;
        if ( StrICont(inp,(UBYTE *)"antibrackets") == 0 ) { type = 1; }
        else if ( StrICont(inp,(UBYTE *)"brackets") == 0 ) { type = 0; }
        else {
            MesPrint("&Illegal (anti)bracket specification in MultiBracket statement");
            if ( error == 0 ) error = 1;
            goto NextLevel;
        }
        while ( FG.cTable[*inp] == 0 ) inp++;
        if ( *inp != ',' ) {
            MesPrint("&Illegal separator after (anti)bracket specification in MultiBracket statement");
            if ( error == 0 ) error = 1;
            goto NextLevel;
        }
        inp++;
/*
        2: call DoBrackets.
*/
        error1 = DoBrackets(inp, type);
        if ( error < 0 ) return(error1);
        if ( error1 > error ) error = error1;
/*
        3: copy bracket information to the multi bracket arrays
*/
        if ( AR.BracketOn ) {
            num = AT.BrackBuf[0];
            to = AC.MultiBracketBuf[i] = (WORD *)Malloc1((num+2)*sizeof(WORD),"bracket buffer i");
            from = AT.BrackBuf;
            *to++ = AR.BracketOn;
            NCOPY(to,from,num);
            *to = 0;
        }
/*
        4: set ready for the next level
*/
NextLevel:
        *s = c; if ( c == ':' ) s++;
```

```
        inp = s;
        *AT.BrackBuf = 0;
        AR.BracketOn = 0;
    }
    if ( *inp != 0 ) {
        MesPrint("&More than %d levels in MultiBracket statement",(WORD)MAXMULTIBRACKETLEVELS);
        if ( error == 0 ) error = 1;
    }
RegEnd:
    AC.MultiBracketLevels = i;
    *AT.BrackBuf = 0;
    AR.BracketOn = 0;
    AC.bracketindexflag = 0;
    AT.bracketindexflag = 0;
    return(error);
}

/*
    #] CoMultiBracket :
    #[ CountComp :

        This routine reads the count statement. The syntax is:
        count minimum,object,size[,object,size]
        Objects can be:
            symbol
            dotproduct
            vector
            function
        Vectors can have the auxiliary flags:
            +v +f +d +?setname

        Output for the compiler:
        TYPECOUNT,size,minimum,objects
        with the objects:
        SYMBOL,4,number,size
        DOTPRODUCT,5,v1,v2,size
        FUNCTION,4,number,size
        VECTOR,5,number,bits,size or VECTOR,6,number,bits,setnumber,size

        Currently only used in the if statement
*/

WORD *CountComp(UBYTE *inp, WORD *to)
{
    GETIDENTITY
    UBYTE *p, c;
    WORD *w, mini = 0, type, c1, c2;
    int error = 0;
    p = inp;
    w = to;
    AR.Eside = 2;
    *w++ = TYPECOUNT;
    *w++ = 0;
    *w++ = 0;
    while ( *p == ',' ) {
        p++; inp = p;
        if ( *p == '[' || FG.cTable[*p] == 0 ) {
            if ( ( p = SkipAName(inp) ) == 0 ) return(0);
            c = *p; *p = 0;
            type = GetName(AC.varnames,inp,&c1,WITHAUTO);
            if ( c == '.' ) {
                if ( type == CVECTOR || type == CDUBIOUS ) {
                    *p++ = c;
                    inp = p;
                    p = SkipAName(p);
                    if ( p == 0 ) return(0);
                    c = *p;
                    *p = 0;
                    type = GetName(AC.varnames,inp,&c2,WITHAUTO);
                    if ( type != CVECTOR && type != CDUBIOUS ) {
                        MesPrint("&Not a vector in dotproduct in if statement: %s",inp);
                        error = 1;
                    }
                    else type = CDOTPRODUCT;
                }
                else {
                    MesPrint("&Illegal use of . after %s in if statement",inp);
                    if ( type == NAMENOTFOUND )
                        MesPrint("&%s is not a properly declared variable",inp);
                    error = 1;
                    *p++ = c;
                    while ( *p && *p != ')' && *p != ',' ) p++;
                    if ( *p == ',' && FG.cTable[p[1]] == 1 ) {
                        p++;
                        while ( *p && *p != ')' && *p != ',' ) p++;
                    }
                    continue;
```

```
                }
            }
            *p = c;
            switch ( type ) {
                case CSYMBOL:
                    *w++ = SYMBOL; *w++ = 4; *w++ = c1;
Sgetnum:            if ( *p != ',' ) {
                        MesCerr("sequence",p);
                        while ( *p && *p != ')' && *p != ',' ) p++;
                        error = 1;
                    }
                    p++; inp = p;
                    ParseSignedNumber(mini,p)
                    if ( FG.cTable[p[-1]] != 1 || ( *p && *p != ')' && *p != ',' ) ) {
                        while ( *p && *p != ')' && *p != ',' ) p++;
                        error = 1;
                        c = *p; *p = 0;
                        MesPrint("&Improper value in count: %s",inp);
                        *p = c;
                        while ( *p && *p != ')' && *p != ',' ) p++;
                    }
                    *w++ = mini;
                    break;
                case CFUNCTION:
                    *w++ = FUNCTION; *w++ = 4; *w++ = c1+FUNCTION; goto Sgetnum;
                case CDOTPRODUCT:
                    *w++ = DOTPRODUCT; *w++ = 5;
                    *w++ = c2 + AM.OffsetVector;
                    *w++ = c1 + AM.OffsetVector;
                    goto Sgetnum;
                case CVECTOR:
                    *w++ = VECTOR; *w++ = 5;
                    *w++ = c1 + AM.OffsetVector;
                    if ( *p == ',' ) {
                        *w++ = VECTBIT | DOTPBIT | FUNBIT;
                        goto Sgetnum;
                    }
                    else if ( *p == '+' ) {
                        p++;
                        *w = 0;
                        while ( *p && *p != ',' ) {
                            if ( *p == 'v' || *p == 'V' ) {
                                *w |= VECTBIT; p++;
                            }
                            else if ( *p == 'd' || *p == 'D' ) {
                                *w |= DOTPBIT; p++;
                            }
                            else if ( *p == 'f' || *p == 'F'
                            || *p == 't' || *p == 'T' ) {
                                *w |= FUNBIT; p++;
                            }
                            else if ( *p == '?' ) {
                                p++; inp = p;
                                if ( *p == '{' ) { /* } */
                                    SKIPBRA2(p)
                                    if ( p == 0 ) return(0);
                                    if ( ( c1 = DoTempSet(inp+1,p) ) < 0 ) return(0);
                                    if ( Sets[c1].type != CFUNCTION ) {
                                        MesPrint("&set type conflict: Function expected");
                                        return(0);
                                    }
                                    type = CSET;
                                    c = *++p;
                                }
                                else {
                                    p = SkipAName(p);
                                    if ( p == 0 ) return(0);
                                    c = *p; *p = 0;
                                    type = GetName(AC.varnames,inp,&c1,WITHAUTO);
                                }
                                if ( type != CSET && type != CDUBIOUS ) {
                                    MesPrint("&%s is not a set",inp);
                                    error = 1;
                                }
                                w[-2] = 6;
                                *w++ |= SETBIT;
                                *w++ = c1;
                                *p = c;
                                goto Sgetnum;
                            }
                            else {
                                MesCerr("specifier for vector",p);
                                error = 1;
                            }
                        }
                        w++;
                        goto Sgetnum;
```

```
                    }
                    else {
                        MesCerr("specifier for vector",p);
                        while ( *p && *p != ')' && *p != ',' ) p++;
                        error = 1;
                        *w++ = VECTBIT | DOTPBIT | FUNBIT;
                        goto Sgetnum;
                    }
                case CDUBIOUS:
                    goto skipfield;
                default:
                    *p = 0;
                    MesPrint("&%s is not a symbol, function, vector or dotproduct",inp);
                    error = 1;
skipfield:          while ( *p && *p != ')' && *p != ',' ) p++;
                    if ( *p && FG.cTable[p[1]] == 1 ) {
                        p++;
                        while ( *p && *p != ')' && *p != ',' ) p++;
                    }
                    break;
            }
        }
        else {
            MesCerr("name",p);
            while ( *p && *p != ',' ) p++;
            error = 1;
        }
    }
    to[1] = w-to;
    if ( *p == ')' ) p++;
    if ( *p ) { MesCerr("end of statement",p); return(0); }
    if ( error ) return(0);
    return(w);
}

/*
    #] CountComp :
    #[ CoIf :

        Reads the if statement: There must be a pair of parentheses.
        Much work is delegated to the routines in compi2 and CountComp.
        The goto is kept hanging as it is forward.
        The address in which the label must be written is pushed on
        the AC.IfStack.

        Here we allow statements of the type
        if ( condition ) single statement;
        compile the if statement.
        test character at end
        if not ; or )
        copy the statement after the proper parenthesis to the
        beginning of the AC.iBuffer.
        Have it compiled.
        generate an endif statement.
*/

static UWORD *CIscratC = 0;

int CoIf(UBYTE *inp)
{
    GETIDENTITY
    int error = 0, level;
    WORD *w, *ww, *u, *s, *OldWork, *OldSpace = AT.WorkSpace;
    WORD gotexp = 0;        /* Indicates whether there can be a condition */
    WORD lenpp, lenlev, ncoef, i, number;
    UBYTE *p, *pp, *ppp, c;
    CBUF *C = cbuf+AC.cbufnum;
    LONG x;
    if ( *inp == '(' && inp[1] == ',' ) inp += 2;
    else if ( *inp == '(' ) inp++;  /* Usually we enter at the bracket */

    if ( CIscratC == 0 )
        CIscratC = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"CoIf");
    lenpp = 0;
    lenlev = 1;
    if ( AC.IfLevel >= AC.MaxIf ) DoubleIfBuffers();
    AC.IfCount[lenpp++] = 0;
/*
    IfStack is used for organizing the 'go to' for the various if levels
*/
    *AC.IfStack++ = C->Pointer-C->Buffer+2;
/*
    IfSumCheck is used to test for illegal nesting of if, argument or repeat.
*/
    AC.IfSumCheck[AC.IfLevel] = NestingChecksum();
    AC.IfLevel++;
    w = OldWork = AT.WorkPointer;
```

```
    *w++ = TYPEIF;
    w += 2;
    p = inp;
    for(;;) {
        inp = p;
        level = 0;
ReDo:
        if ( FG.cTable[*p] == 1 ) {     /* Number */
            if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
            u = w;
            *w++ = LONGNUMBER;
            w += 2;
            if ( GetLong(p,(UWORD *)w,&ncoef) ) { ncoef = 1; error = 1; }
            w[-1] = ncoef;
            while ( FG.cTable[*++p] == 1 );
            if ( *p == '/' ) {
                p++;
                if ( FG.cTable[*p] != 1 ) {
                    MesCerr("sequence",p); error = 1; goto OnlyNum;
                }
                if ( GetLong(p,CIscratC,&ncoef) ) {
                    ncoef = 1; error = 1;
                }
                while ( FG.cTable[*++p] == 1 );
                if ( ncoef == 0 ) {
                    MesPrint("&Division by zero!");
                    error = 1;
                }
                else {
                    if ( w[-1] != 0 ) {
                        if ( Simplify(BHEAD (UWORD *)w,(WORD *)(w-1),
                        CIscratC,&ncoef) ) error = 1;
                        else {
                            i = w[-1];
                            if ( i >= ncoef ) {
                                i = w[-1];
                                w += i;
                                i -= ncoef;
                                s = (WORD *)CIscratC;
                                NCOPY(w,s,ncoef);
                                while ( --i >= 0 ) *w++ = 0;
                            }
                            else {
                                w += i;
                                i = ncoef - i;
                                while ( --i >= 0 ) *w++ = 0;
                                s = (WORD *)CIscratC;
                                NCOPY(w,s,ncoef);
                            }
                        }
                    }
                }
            }
            else {
OnlyNum:
                w += ncoef;
                if ( ncoef > 0 ) {
                    ncoef--; *w++ = 1;
                    while ( --ncoef >= 0 ) *w++ = 0;
                }
            }
            u[1] = WORDDIF(w,u);
            u[2] = (u[1] - 3)»1;
            if ( level ) u[2] = -u[2];
            gotexp = 1;
        }
        else if ( *p == '+' ) { p++; goto ReDo; }
        else if ( *p == '-' ) { level ^= 1; p++; goto ReDo; }
        else if ( *p == 'c' || *p == 'C' ) {    /* Count or Coefficient */
            if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
            while ( FG.cTable[*++p] == 0 );
            c = *p; *p = 0;
            if ( !StrICmp(inp,(UBYTE *)"count") ) {
                *p = c;
                if ( c != '(' ) {
                    MesPrint("&no ( after count");
                    error = 1;
                    goto endofif;
                }
                inp = p;
                SKIPBRA4(p);
                c = *++p; *p = 0; *inp = ',';
                w = CountComp(inp,w);
                *p = c; *inp = '(';
                if ( w == 0 ) { error = 1; goto endofif; }
                gotexp = 1;
            }
```

```
            else if ( ConWord(inp,(UBYTE *)"coefficient") && ( p - inp ) > 3 ) {
                *w++ = COEFFI;
                *w++ = 2;
                *p = c;
                gotexp = 1;
            }
            else goto NoGood;
            inp = p;
        }
        else if ( *p == 'm' || *p == 'M' ) {    /* match */
            if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
            while ( !FG.cTable[*++p] );
            c = *p; *p = 0;
            if ( !StrICmp(inp,(UBYTE *)"match") ) {
                *p = c;
                if ( c != '(' ) {
                    MesPrint("&no ( after match");
                    error = 1;
                    goto endofif;
                }
                p++; inp = p;
                SKIPBRA4(p);
                *p = '=';
/*
                Now we can call the reading of the lhs of an id statement.
                This has to be modified in the future.
*/
                AT.WorkSpace = AT.WorkPointer = w;
                ppp = inp;
                while ( FG.cTable[*ppp] == 0 && ppp < p ) ppp++;
                if ( *ppp == ',' ) AC.idoption = 0;
                else AC.idoption = SUBMULTI;
                level = CoIdExpression(inp,TYPEIF);
                AT.WorkSpace = OldSpace;
                AT.WorkPointer = OldWork;
                if ( level != 0 ) {
                    if ( level < 0 ) { error = -1; goto endofif; }
                    error = 1;
                }
/*
                If we pop numlhs we are in good shape
*/
                s = u = C->lhs[C->numlhs];
                while ( u < C->Pointer ) *w++ = *u++;
                C->numlhs--; C->Pointer = s;
                *p++ = ')';
                inp = p;
                gotexp = 1;
            }
            else if ( !StrICmp(inp,(UBYTE *)"multipleof") ) {
            if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
                *p = c;
                if ( c != '(' ) {
                    MesPrint("&no ( after multipleof");
                    error = 1; goto endofif;
                }
                p++;
                if ( FG.cTable[*p] != 1 ) {
Nomulof:            MesPrint("&multipleof needs a short positive integer argument");
                    error = 1; goto endofif;
                }
                ParseNumber(x,p)
                if ( *p != ')' || x <= 0 || x > MAXPOSITIVE ) goto Nomulof;
                p++;
                *w++ = MULTIPLEOF; *w++ = 3; *w++ = (WORD)x;
                inp = p;
                gotexp = 1;
            }
            else {
NoGood:         MesPrint("&Unrecognized word: %s",inp);
                *p = c;
                error = 1;
                level = 0;
                if ( c == '(' ) SKIPBRA4(p)
                inp = ++p;
                gotexp = 1;
            }
        }
        else if ( *p == 'f' || *p == 'F' ) {    /* FindLoop */
            if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
            while ( FG.cTable[*++p] == 0 );
            c = *p; *p = 0;
            if ( !StrICmp(inp,(UBYTE *)"findloop") ) {
                *p = c;
                if ( c != '(' ) {
                    MesPrint("&no ( after findloop");
                    error = 1;
```

```
                    goto endofif;
                }
                inp = p;
                SKIPBRA4(p);
                c = *++p; *p = 0; *inp = ',';
                if ( CoFindLoop(inp) ) goto endofif;
                s = u = C->lhs[C->numlhs];
                while ( u < C->Pointer ) *w++ = *u++;
                C->numlhs--; C->Pointer = s;
                *p = c; *inp = '(';
                if ( w == 0 ) { error = 1; goto endofif; }
                gotexp = 1;
            }
            else goto NoGood;
            inp = p;
        }
        else if ( *p == 'e' || *p == 'E' ) { /* Expression */
            if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
            while ( FG.cTable[*++p] == 0 );
            c = *p; *p = 0;
            if ( !StrICmp(inp,(UBYTE *)"expression") ) {
                *p = c;
                if ( c != '(' ) {
                    MesPrint("&no ( after expression");
                    error = 1;
                    goto endofif;
                }
                p++; ww = w; *w++ = IFEXPRESSION; w++;
                while ( *p != ')' ) {
                    if ( *p == ',' ) { p++; continue; }
                    if ( *p == '[' || FG.cTable[*p] == 0 ) {
                        pp = p;
                        if ( ( p = SkipAName(p) ) == 0 ) {
                            MesPrint("&Improper name for an expression: '%s'",pp);
                            error = 1;
                            goto endofif;
                        }
                        c = *p; *p = 0;
                        if ( GetName(AC.exprnames,pp,&number,NOAUTO) == CEXPRESSION ) {
                            *w++ = number;
                        }
                        else if ( GetName(AC.varnames,pp,&number,NOAUTO) != NAMENOTFOUND ) {
                            MesPrint("&%s is not an expression",pp);
                            error = 1;
                            *w++ = number;
                        }
                        *p = c;
                    }
                    else {
                        MesPrint("&Illegal object in Expression in if-statement");
                        error = 1;
                        while ( *p && *p != ',' && *p != ')' ) p++;
                        if ( *p == 0 || *p == ')' ) break;
                    }
                }
                ww[1] = w - ww;
                p++;
                gotexp = 1;
            }
            else goto NoGood;
            inp = p;
        }
        else if ( *p == 'i' || *p == 'I' ) { /* IsFactorized */
            if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
            while ( FG.cTable[*++p] == 0 );
            c = *p; *p = 0;
            if ( !StrICmp(inp,(UBYTE *)"isfactorized") ) {
                *p = c;
                if ( c != '(' ) { /* No expression means current expression */
                  ww = w; *w++ = IFISFACTORIZED; w++;
                }
                else {
                  p++; ww = w; *w++ = IFISFACTORIZED; w++;
                  while ( *p != ')' ) {
                    if ( *p == ',' ) { p++; continue; }
                    if ( *p == '[' || FG.cTable[*p] == 0 ) {
                        pp = p;
                        if ( ( p = SkipAName(p) ) == 0 ) {
                            MesPrint("&Improper name for an expression: '%s'",pp);
                            error = 1;
                            goto endofif;
                        }
                        c = *p; *p = 0;
                        if ( GetName(AC.exprnames,pp,&number,NOAUTO) == CEXPRESSION ) {
                            *w++ = number;
                        }
                        else if ( GetName(AC.varnames,pp,&number,NOAUTO) != NAMENOTFOUND ) {
```

```
                            MesPrint("&%s is not an expression",pp);
                            error = 1;
                            *w++ = number;
                        }
                        *p = c;
                    }
                    else {
                        MesPrint("&Illegal object in IsFactorized in if-statement");
                        error = 1;
                        while ( *p && *p != ',' && *p != ')' ) p++;
                        if ( *p == 0 || *p == ')' ) break;
                    }
                  }
                  p++;
                }
                ww[1] = w - ww;
                gotexp = 1;
            }
            else goto NoGood;
            inp = p;
        }
        else if ( *p == 'o' || *p == 'O' ) { /* Occurs */
/*
            Tests whether variables occur inside a term.
            At the moment this is done one by one.
            If we want to do them in groups we should do the reading
            a bit different: each as a variable in a term, and then
            use Normalize to get the variables grouped and in order.
            That way FindVar (in if.c) can work more efficiently.
            Still to be done!!!
            TASK: Nice little task for someone to learn.
*/
            UBYTE cc;
            if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
            while ( FG.cTable[*++p] == 0 );
            c = cc = *p; *p = 0;
            if ( !StrICmp(inp,(UBYTE *)"occurs") ) {
                WORD c1, c2, type;
                *p = cc;
                if ( cc != '(' ) {
                    MesPrint("&no ( after occurs");
                    error = 1;
                    goto endofif;
                }
                inp = p;
                SKIPBRA4(p);
                cc = *++p; *p = 0; *inp = ','; pp = p;
                ww = w;
                *w++ = IFOCCURS; *w++ = 0;
                while ( *inp ) {
                    while ( *inp == ',' ) inp++;
                    if ( *inp == 0 || *inp == ')' ) break;
/*
                    Now read a list of names
                    We can have symbols, vectors, dotproducts, indices, functions.
                    There could also be dummy indices and/or extra symbols.
*/
                    if ( *inp == '[' || FG.cTable[*inp] == 0 ) {
                        if ( ( p = SkipAName(inp) ) == 0 ) return(0);
                        c = *p; *p = 0;
                        type = GetName(AC.varnames,inp,&c1,WITHAUTO);
                        if ( c == '.' ) {
                            if ( type == CVECTOR || type == CDUBIOUS ) {
                                *p++ = c;
                                inp = p;
                                p = SkipAName(p);
                                if ( p == 0 ) return(0);
                                c = *p;
                                *p = 0;
                                type = GetName(AC.varnames,inp,&c2,WITHAUTO);
                                if ( type != CVECTOR && type != CDUBIOUS ) {
                                    MesPrint("&Not a vector in dotproduct in if statement: %s",inp);
                                    error = 1;
                                }
                                else type = CDOTPRODUCT;
                            }
                            else {
                                MesPrint("&Illegal use of . after %s in if statement",inp);
                                if ( type == NAMENOTFOUND )
                                    MesPrint("&%s is not a properly declared variable",inp);
                                error = 1;
                                *p++ = c;
                                while ( *p && *p != ')' && *p != ',' ) p++;
                                if ( *p == ',' && FG.cTable[p[1]] == 1 ) {
                                    p++;
                                    while ( *p && *p != ')' && *p != ',' ) p++;
                                }
```

```
                                continue;
                            }
                        }
                        *p = c;
                        switch ( type ) {
                            case CSYMBOL: /* To worry about extra symbols */
                                *w++ = SYMBOL;
                                *w++ = c1;
                            break;
                            case CINDEX:
                                *w++ = INDEX;
                                *w++ = c1 + AM.OffsetIndex;
                            break;
                            case CVECTOR:
                                *w++ = VECTOR;
                                *w++ = c1 + AM.OffsetVector;
                            break;
                            case CDOTPRODUCT:
                                *w++ = DOTPRODUCT;
                                *w++ = c1 + AM.OffsetVector;
                                *w++ = c2 + AM.OffsetVector;
                            break;
                            case CFUNCTION:
                                *w++ = FUNCTION;
                                *w++ = c1+FUNCTION;
                            break;
                            default:
                                MesPrint("&Illegal variable %s in occurs condition in if statement",inp);
                                error = 1;
                            break;
                        }
                        inp = p;
                    }
                    else {
                        MesPrint("&Illegal object %s in occurs condition in if statement",inp);
                        error = 1;
                        break;
                    }
                }
                ww[1] = w-ww;
                p = pp; *p = cc; *inp = '(';
                gotexp = 1;
                if ( ww[1] <= 2 ) {
                    MesPrint("&The occurs condition in the if statement needs arguments.");
                    error = 1;
                }
            }
            else goto NoGood;
            inp = p;
        }
        else if ( *p == '$' ) {
            if ( gotexp == 1 ) { MesCerr("position for )",p); error = 1; }
            p++; inp = p;
            while ( FG.cTable[*p] == 0 || FG.cTable[*p] == 1 ) p++;
            c = *p; *p = 0;
            if ( ( i = GetDollar(inp) ) < 0 ) {
                MesPrint("&undefined dollar expression %s",inp);
                error = 1;
                i = AddDollar(inp,DOLUNDEFINED,0,0);
            }
            *p = c;
            *w++ = IFDOLLAR; *w++ = 3; *w++ = i;
/*
            And then the IFDOLLAREXTRA pieces for [1] [$y] etc
*/
            if ( *p == '[' ) {
                p++;
                if ( ( w = GetIfDollarFactor(&p,w) ) == 0 ) {
                    error = 1;
                    goto endofif;
                }
                else if ( *p != ']' ) {
                    error = 1;
                    goto endofif;
                }
                p++;
            }
            inp = p;
            gotexp = 1;
        }
        else if ( *p == '(' ) {
            if ( gotexp ) {
                MesCerr("parenthesis",p);
                error = 1;
                goto endofif;
            }
```

```
            gotexp = 0;
            if ( ++lenlev >= AC.MaxIf ) DoubleIfBuffers();
            AC.IfCount[lenpp++] = w-OldWork;
            *w++ = SUBEXPR;
            w += 2;
            p++;
        }
        else if ( *p == ')' ) {
            if ( gotexp == 0 ) { MesCerr("position for )",p); error = 1; }
            gotexp = 1;
            u = AC.IfCount[--lenpp]+OldWork;
            lenlev--;
            u[1] = w - u;
            if ( lenlev <= 0 ) {    /* End if condition */
                AT.WorkSpace = OldSpace;
                AT.WorkPointer = OldWork;
                AddNtoL(OldWork[1],OldWork);
                p++;
                if ( *p == ')' ) {
                    MesPrint("&unmatched parenthesis in if/while ()");
                    error = 1;
                    while ( *++p == ')' );
                }
                if ( *p ) {
                    level = CompileStatement(p);
                    if ( level ) error = level;
                    while ( *p ) p++;
                    if ( CoEndIf(p) && error == 0 ) error = 1;
                }
                return(error);
            }
            p++;
        }
        else if ( *p == '>' ) {
            if ( gotexp == 0 ) goto NoExp;
            if ( p[1] == '=' ) { *w++ = GREATEREQUAL; *w++ = 2; p += 2; }
            else               { *w++ = GREATER;      *w++ = 2; p++; }
            gotexp = 0;
        }
        else if ( *p == '<' ) {
            if ( gotexp == 0 ) goto NoExp;
            if ( p[1] == '=' ) { *w++ = LESSEQUAL; *w++ = 2; p += 2; }
            else               { *w++ = LESS;      *w++ = 2; p++; }
            gotexp = 0;
        }
        else if ( *p == '=' ) {
            if ( gotexp == 0 ) goto NoExp;
            if ( p[1] == '=' ) p++;
            *w++ = EQUAL; *w++ = 2; p++;
            gotexp = 0;
        }
        else if ( *p == '!' && p[1] == '=' ) {
            if ( gotexp == 0 ) { p++; goto NoExp; }
            *w++ = NOTEQUAL; *w++ = 2; p += 2;
            gotexp = 0;
        }
        else if ( *p == '|' && p[1] == '|' ) {
            if ( gotexp == 0 ) { p++; goto NoExp; }
            *w++ = ORCOND; *w++ = 2; p += 2;
            gotexp = 0;
        }
        else if ( *p == '&' && p[1] == '&' ) {
            if ( gotexp == 0 ) {
                p++;
NoExp:          p++;
                MesCerr("sequence",p);
                error = 1;
            }
            else {
                *w++ = ANDCOND; *w++ = 2; p += 2;
                gotexp = 0;
            }
        }
        else if ( *p == 0 ) {
            MesPrint("&Unmatched parentheses");
            error = 1;
            goto endofif;
        }
        else {
            if ( FG.cTable[*p] == 0 ) {
                WORD ij;
                inp = p;
                while ( ( ij = FG.cTable[*++p] ) == 0 || ij == 1 );
                c = *p; *p = 0;
                goto NoGood;
            }
            MesCerr("sequence",p);
```

```
            error = 1;
            p++;
        }
    }
endofif:;
    return(error);
}

/*
    #] CoIf :
    #[ CoElse :
*/

int CoElse(UBYTE *p)
{
    int error = 0;
    CBUF *C = cbuf+AC.cbufnum;
    if ( *p != 0 ) {
        while ( *p == ',' ) p++;
        if ( tolower(*p) == 'i' && tolower(p[1]) == 'f' && p[2] == '(' )
                                                    return(CoElseIf(p+2));
        MesPrint("&No extra text allowed as part of an else statement");
        error = 1;
    }
    if ( AC.IfLevel <= 0 ) { MesPrint("&else statement without if"); return(1); }
    if ( AC.IfSumCheck[AC.IfLevel-1] != NestingChecksum() - 1 ) {
        MesNesting();
        error = 1;
    }
    Add3Com(TYPEELSE,AC.IfLevel)
    C->Buffer[AC.IfStack[-1]] = C->numlhs;
    AC.IfStack[-1] = C->Pointer - C->Buffer - 1;
    return(error);
}

/*
    #] CoElse :
    #[ CoElseIf :
*/

int CoElseIf(UBYTE *inp)
{
    CBUF *C = cbuf+AC.cbufnum;
    if ( AC.IfLevel <= 0 ) { MesPrint("&elseif statement without if"); return(1); }
    Add3Com(TYPEELSE,-AC.IfLevel)
    AC.IfLevel--;
    C->Buffer[*--AC.IfStack] = C->numlhs;
    return(CoIf(inp));
}

/*
    #] CoElseIf :
    #[ CoEndIf :

        It puts a RHS-level at the position indicated in the AC.IfStack.
        This corresponds to the label belonging to a forward goto.
        It is the goto that belongs either to the failing condition
        of the if (no else statement), or the completion of the
        success path (with else statement)
        The code is a jump to the next statement. It is there to prevent
        problems with
        if ( .. )
            if ( .. )
            endif;
        elseif ( .. )
*/

int CoEndIf(UBYTE *inp)
{
    CBUF *C = cbuf+AC.cbufnum;
    WORD i = C->numlhs, to, k = -AC.IfLevel;
    int error = 0;
    while ( *inp == ',' ) inp++;
    if ( *inp != 0 ) {
        error = 1;
        MesPrint("&No extra text allowed as part of an endif/elseif statement");
    }
    if ( AC.IfLevel <= 0 ) {
        MesPrint("&Endif statement without corresponding if"); return(1);
    }
    AC.IfLevel--;
    C->Buffer[*--AC.IfStack] = i+1;
    if ( AC.IfSumCheck[AC.IfLevel] != NestingChecksum() ) {
        MesNesting();
        error = 1;
    }
    Add3Com(TYPEENDIF,i+1)
```

```
/*
    Now the search for the TYPEELSE in front of the elseif statements
*/
    to = C->numlhs;
    while ( i > 0 ) {
        if ( C->lhs[i][0] == TYPEELSE && C->lhs[i][2] == to ) to = i;
        if ( C->lhs[i][0] == TYPEIF ) {
            if ( C->lhs[i][2] == to ) {
                i--;
                if ( i <= 0 || C->lhs[i][0] != TYPEELSE
                || C->lhs[i][2] != k ) break;
                C->lhs[i][2] = C->numlhs;
                to = i;
            }
        }
        i--;
    }
    return(error);
}

/*
    #] CoEndIf :
    #[ CoWhile :
*/

int CoWhile(UBYTE *inp)
{
    CBUF *C = cbuf+AC.cbufnum;
    WORD startnum = C->numlhs + 1;
    int error;
    AC.WhileLevel++;
    error = CoIf(inp);
    if ( C->numlhs > startnum && C->lhs[startnum][2] == C->numlhs
                            && C->lhs[C->numlhs][0] == TYPEENDIF ) {
        C->lhs[C->numlhs][2] = startnum-1;
        AC.WhileLevel--;
    }
    else C->lhs[startnum][2] = startnum;
    return(error);
}

/*
    #] CoWhile :
    #[ CoEndWhile :
*/

int CoEndWhile(UBYTE *inp)
{
    int error = 0;
    WORD i;
    CBUF *C = cbuf+AC.cbufnum;
    if ( AC.WhileLevel <= 0 ) {
        MesPrint("&EndWhile statement without corresponding While"); return(1);
    }
    AC.WhileLevel--;
    i = C->Buffer[AC.IfStack[-1]];
    error = CoEndIf(inp);
    C->lhs[C->numlhs][2] = i - 1;
    return(error);
}

/*
    #] CoEndWhile :
    #[ DoFindLoop :

    Function,arguments=number,loopsize=number,outfun=function,include=index;
*/

static char *messfind[] = {
    "Findloop(function,arguments=#,loopsize(=#|<#)[,include=index])"
   ,"Replaceloop,function,arguments=#,loopsize(=#|<#),outfun=function[,include=index]"
    };
static WORD comfindloop[7] = { TYPEFINDLOOP,7,0,0,0,0,0 };

int DoFindLoop(UBYTE *inp, int mode)
{
    UBYTE *s, c;
    WORD funnum, nargs = 0, nloop = 0, indexnum = 0, outfun = 0;
    int type, aflag, lflag, indflag, outflag, error = 0, sym;
    while ( *inp == ',' ) inp++;
    if ( ( s = SkipAName(inp) ) == 0 ) {
syntax: MesPrint("&Proper syntax is:");
        MesPrint("%s",messfind[mode]);
        return(1);
    }
    c = *s; *s = 0;
    if ( ( ( type = GetName(AC.varnames,inp,&funnum,WITHAUTO) ) == NAMENOTFOUND )
```

```
        || type != CFUNCTION || ( ( sym = (functions[funnum].symmetric) & ~REVERSEORDER )
        != SYMMETRIC && sym != ANTISYMMETRIC ) ) {
        MesPrint("&%s should be a (anti)symmetric function or tensor",inp);
    }
    funnum += FUNCTION;
    *s = c; inp = s;
    aflag = lflag = indflag = outflag = 0;
    while ( *inp == ',' ) {
        while ( *inp == ',' ) inp++;
        s = inp;
        if ( ( s = SkipAName(inp) ) == 0 ) goto syntax;
        c = *s; *s = 0;
        if ( StrICont(inp,(UBYTE *)"arguments") == 0 ) {
            if ( c != '=' ) goto syntax;
            *s++ = c;
            NeedNumber(nargs,s,syntax)
            aflag++;
            inp = s;
        }
        else if ( StrICont(inp,(UBYTE *)"loopsize") == 0 ) {
            if ( c != '=' && c != '<' ) goto syntax;
            *s++ = c;
            if ( FG.cTable[*s] == 1 ) {
                NeedNumber(nloop,s,syntax)
                if ( nloop < 2 ) {
                    MesPrint("&loopsize should be at least 2");
                    error = 1;
                }
                if ( c == '<' ) nloop = -nloop;
            }
            else if ( tolower(*s) == 'a' && tolower(s[1]) == 'l'
            && tolower(s[2]) == 'l' && FG.cTable[s[3]] > 1 ) {
                nloop = -1; s += 3;
                if ( c != '=' ) goto syntax;
            }
            inp = s;
            lflag++;
        }
        else if ( StrICont(inp,(UBYTE *)"include") == 0 ) {
            if ( c != '=' ) goto syntax;
            *s++ = c;
            if ( ( inp = SkipAName(s) ) == 0 ) goto syntax;
            c = *inp; *inp = 0;
            if ( ( type = GetName(AC.varnames,s,&indexnum,WITHAUTO) ) != CINDEX ) {
                MesPrint("&%s is not a proper index",s);
                error = 1;
            }
            else if ( indexnum < WILDOFFSET
            && indices[indexnum].dimension == 0 ) {
                MesPrint("&%s should be a summable index",s);
                error = 1;
            }
            indexnum += AM.OffsetIndex;
            *inp = c;
            indflag++;
        }
        else if ( StrICont(inp,(UBYTE *)"outfun") == 0 ) {
            if ( c != '=' ) goto syntax;
            *s++ = c;
            if ( ( inp = SkipAName(s) ) == 0 ) goto syntax;
            c = *inp; *inp = 0;
            if ( ( type = GetName(AC.varnames,s,&outfun,WITHAUTO) ) != CFUNCTION ) {
                MesPrint("&%s is not a proper function or tensor",s);
                error = 1;
            }
            outfun += FUNCTION;
            outflag++;
            *inp = c;
        }
        else {
            MesPrint("&Unrecognized option in FindLoop or ReplaceLoop: %s",inp);
            *s = c; inp = s;
            while ( *inp && *inp != ',' ) inp++;
        }
    }
    if ( *inp != 0 && mode == REPLACELOOP ) goto syntax;
    if ( mode == FINDLOOP && outflag > 0 ) {
        MesPrint("&outflag option is illegal in FindLoop");
        error = 1;
    }
    if ( mode == REPLACELOOP && outflag == 0 ) goto syntax;
    if ( aflag == 0 || lflag == 0 ) goto syntax;
    comfindloop[3] = funnum;
    comfindloop[4] = nloop;
    comfindloop[5] = nargs;
    comfindloop[6] = outfun;
    comfindloop[1] = 7;
```

```
    if ( indflag ) {
        if ( mode == 0 ) comfindloop[2] =  indexnum + 5;
        else             comfindloop[2] = -indexnum - 5;
    }
    else comfindloop[2] = mode;
    AddNtoL(comfindloop[1],comfindloop);
    return(error);
}

/*
    #] DoFindLoop :
    #[ CoFindLoop :
*/

int CoFindLoop(UBYTE *inp)
{ return(DoFindLoop(inp,FINDLOOP)); }

/*
    #] CoFindLoop :
    #[ CoReplaceLoop :
*/

int CoReplaceLoop(UBYTE *inp)
{ return(DoFindLoop(inp,REPLACELOOP)); }

/*
    #] CoReplaceLoop :
    #[ CoFunPowers :
*/

static UBYTE *FunPowOptions[] = {
     (UBYTE *)"nofunpowers"
    ,(UBYTE *)"commutingonly"
    ,(UBYTE *)"allfunpowers"
    };

int CoFunPowers(UBYTE *inp)
{
    UBYTE *option, c;
    int i, maxoptions = sizeof(FunPowOptions)/sizeof(UBYTE *);
    while ( *inp == ',' ) inp++;
    option = inp;
    inp = SkipAName(inp); c = *inp; *inp = 0;
    for ( i = 0; i < maxoptions; i++ ) {
        if ( StrICont(option,FunPowOptions[i]) == 0 ) {
            if ( c ) {
                *inp = c;
                MesPrint("&Illegal FunPowers statement");
                return(1);
            }
            AC.funpowers = i;
            return(0);
        }
    }
    MesPrint("&Illegal option in FunPowers statement: %s",option);
    return(1);
}

/*
    #] CoFunPowers :
    #[ CoUnitTrace :
*/

int CoUnitTrace(UBYTE *s)
{
    WORD num;
    if ( FG.cTable[*s] == 1 ) {
        ParseNumber(num,s)
        if ( *s != 0 ) {
nogood:     MesPrint("&Value of UnitTrace should be a (positive) number or a symbol");
            return(1);
        }
        AC.lUniTrace[0] = SNUMBER;
        AC.lUniTrace[2] = num;
    }
    else {
        if ( GetName(AC.varnames,s,&num,WITHAUTO) == CSYMBOL ) {
            AC.lUniTrace[0] = SYMBOL;
            AC.lUniTrace[2] = num;
            num = -num;
        }
        else goto nogood;
        s = SkipAName(s);
        if ( *s ) goto nogood;
    }
    AC.lUnitTrace = num;
    return(0);
```

```
}

/*
    #] CoUnitTrace :
    #[ CoTerm :

    Note: termstack holds the offset of the term statement in the compiler
    buffer. termsortstack holds the offset of the last sort statement
        (or the corresponding term statement)
*/

int CoTerm(UBYTE *s)
{
    GETIDENTITY
    WORD *w = AT.WorkPointer;
    int error = 0;
    while ( *s == ',' ) s++;
    if ( *s ) {
        MesPrint("&Illegal syntax for Term statement");
        return(1);
    }
    if ( AC.termlevel+1 >= AC.maxtermlevel ) {
        if ( AC.maxtermlevel <= 0 ) {
            AC.maxtermlevel = 20;
            AC.termstack = (LONG *)Malloc1(AC.maxtermlevel*sizeof(LONG),"termstack");
            AC.termsortstack = (LONG *)Malloc1(AC.maxtermlevel*sizeof(LONG),"termsortstack");
            AC.termsumcheck = (WORD *)Malloc1(AC.maxtermlevel*sizeof(WORD),"termsumcheck");
        }
        else {
            DoubleBuffer((void **)AC.termstack,(void **)AC.termstack+AC.maxtermlevel,
                sizeof(LONG),"doubling termstack");
            DoubleBuffer((void **)AC.termsortstack,
                (void **)AC.termsortstack+AC.maxtermlevel,
                sizeof(LONG),"doubling termsortstack");
            DoubleBuffer((void **)AC.termsumcheck,
                (void **)AC.termsumcheck+AC.maxtermlevel,
                sizeof(LONG),"doubling termsumcheck");
            AC.maxtermlevel *= 2;
        }
    }
    AC.termsumcheck[AC.termlevel] = NestingChecksum();
    AC.termstack[AC.termlevel] = cbuf[AC.cbufnum].Pointer
                                - cbuf[AC.cbufnum].Buffer + 2;
    AC.termsortstack[AC.termlevel] = AC.termstack[AC.termlevel] + 1;
    AC.termlevel++;
    *w++ = TYPETERM;
    w++;
    *w++ = cbuf[AC.cbufnum].numlhs;
    *w++ = cbuf[AC.cbufnum].numlhs;
    AT.WorkPointer[1] = w - AT.WorkPointer;
    AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
    return(error);
}

/*
    #] CoTerm :
    #[ CoEndTerm :
*/

int CoEndTerm(UBYTE *s)
{
    CBUF *C = cbuf+AC.cbufnum;
    while ( *s == ',' ) s++;
    if ( *s ) {
        MesPrint("&Illegal syntax for EndTerm statement");
        return(1);
    }
    if ( AC.termlevel <= 0 ) {
        MesPrint("&EndTerm without corresponding Argument statement");
        return(1);
    }
    AC.termlevel--;
    cbuf[AC.cbufnum].Buffer[AC.termstack[AC.termlevel]] = C->numlhs;
    cbuf[AC.cbufnum].Buffer[AC.termsortstack[AC.termlevel]] = C->numlhs;
    if ( AC.termsumcheck[AC.termlevel] != NestingChecksum() ) {
        MesNesting();
        return(1);
    }
    return(0);
}

/*
    #] CoEndTerm :
    #[ CoSort :
*/

int CoSort(UBYTE *s)
```

```
{
    GETIDENTITY
    WORD *w = AT.WorkPointer;
    int error = 0;
    while ( *s == ',' ) s++;
    if ( *s ) {
        MesPrint("&Illegal syntax for Sort statement");
        error = 1;
    }
    if ( AC.termlevel <= 0 ) {
        MesPrint("&The Sort statement can only be used inside a term environment");
        error = 1;
    }
    if ( error ) return(error);
    *w++ = TYPESORT;
    w++;
    w++;
    cbuf[AC.cbufnum].Buffer[AC.termsortstack[AC.termlevel-1]] =
                                        *w = cbuf[AC.cbufnum].numlhs+1;
    w++;
    AC.termsortstack[AC.termlevel-1] = cbuf[AC.cbufnum].Pointer
                            - cbuf[AC.cbufnum].Buffer + 3;
    if ( AC.termsumcheck[AC.termlevel-1] != NestingChecksum() - 1 ) {
        MesNesting();
        return(1);
    }
    AT.WorkPointer[1] = w - AT.WorkPointer;
    AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
    return(error);
}

/*
    #] CoSort :
    #[ CoPolyFun :

    Collect,functionname
*/

int CoPolyFun(UBYTE *s)
{
    GETIDENTITY
    WORD numfun;
    int type;
    UBYTE *t;
    AR.PolyFun = AC.lPolyFun = 0;
    AR.PolyFunInv = AC.lPolyFunInv = 0;
    AR.PolyFunType = AC.lPolyFunType = 0;
    AR.PolyFunExp = AC.lPolyFunExp = 0;
    AR.PolyFunVar = AC.lPolyFunVar = 0;
    AR.PolyFunPow = AC.lPolyFunPow = 0;
    if ( *s == 0 ) { return(0); }
    t = SkipAName(s);
    if ( t == 0 || *t != 0 ) {
        MesPrint("&PolyFun statement needs a single commuting function for its argument");
        return(1);
    }
    if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
    || ( functions[numfun].spec != 0 ) || ( functions[numfun].commute != 0 ) ) {
        MesPrint("&%s should be a regular commuting function",s);
        if ( type < 0 ) {
            if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
                AddFunction(s,0,0,0,0,0,-1,-1);
        }
        return(1);
    }
    AR.PolyFun = AC.lPolyFun = numfun+FUNCTION;
    AR.PolyFunType = AC.lPolyFunType = 1;
    return(0);
}

/*
    #] CoPolyFun :
    #[ CoPolyRatFun :

    PolyRatFun [,functionname[,functionname](option)]
*/

int CoPolyRatFun(UBYTE *s)
{
    GETIDENTITY
    WORD numfun;
    int type;
    UBYTE *t, c;
    AR.PolyFun = AC.lPolyFun = 0;
    AR.PolyFunInv = AC.lPolyFunInv = 0;
    AR.PolyFunType = AC.lPolyFunType = 0;
    AR.PolyFunExp = AC.lPolyFunExp = 0;
```

```
    AR.PolyFunVar = AC.lPolyFunVar = 0;
    AR.PolyFunPow = AC.lPolyFunPow = 0;
    if ( *s == 0 ) return(0);
    t = SkipAName(s);
    if ( t == 0 ) goto NumErr;
    c = *t; *t = 0;
    if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
    || ( functions[numfun].spec != 0 ) || ( functions[numfun].commute != 0 ) ) {
        MesPrint("&%s should be a regular commuting function",s);
        if ( type < 0 ) {
            if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
                AddFunction(s,0,0,0,0,0,-1,-1);
        }
        return(1);
    }
    AR.PolyFun = AC.lPolyFun = numfun+FUNCTION;
    AR.PolyFunInv = AC.lPolyFunInv = 0;
    AR.PolyFunType = AC.lPolyFunType = 2;
    AC.PolyRatFunChanged = 1;
    if ( c == 0 ) return(0);
    *t = c;
    if ( *t == '-' ) { AC.PolyRatFunChanged = 0; t++; }
    while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
    if ( *t == 0 ) return(0);
    if ( *t != '(' ) {
        s = t;
        t = SkipAName(s);
        if ( t == 0 ) goto NumErr;
        c = *t; *t = 0;
        if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
        || ( functions[numfun].spec != 0 ) || ( functions[numfun].commute != 0 ) ) {
            MesPrint("&%s should be a regular commuting function",s);
            if ( type < 0 ) {
                if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
                    AddFunction(s,0,0,0,0,0,-1,-1);
            }
            return(1);
        }
        AR.PolyFunInv = AC.lPolyFunInv = numfun+FUNCTION;
        if ( c == 0 ) return(0);
        *t = c;
        if ( *t == '-' ) { AC.PolyRatFunChanged = 0; t++; }
        while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
        if ( *t == 0 ) return(0);
    }
    if ( *t == '(' ) {
        t++;
        while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
/*
        Next we need a keyword like
            (divergence,ep)
            (expand,ep,maxpow)
*/
        s = t;
        t = SkipAName(s);
        if ( t == 0 ) goto NumErr;
        c = *t; *t = 0;
        if ( ( StrICmp(s,(UBYTE *)"divergence") == 0 )
        || ( StrICmp(s,(UBYTE *)"finddivergence") == 0 ) ) {
            if ( c != ',' ) {
                MesPrint("&Illegal option field in PolyRatFun statement.");
                return(1);
            }
            *t = c;
            while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
            s = t;
            t = SkipAName(s);
            if ( t == 0 ) goto NumErr;
            c = *t; *t = 0;
            if ( ( type = GetName(AC.varnames,s,&AC.lPolyFunVar,WITHAUTO) ) != CSYMBOL ) {
                MesPrint("&Illegal symbol %s in option field in PolyRatFun statement.",s);
                return(1);
            }
            *t = c;
            while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
            if ( *t != ')' ) {
                MesPrint("&Illegal termination of option in PolyRatFun statement.");
                return(1);
            }
            AR.PolyFunExp = AC.lPolyFunExp = 1;
            AR.PolyFunVar = AC.lPolyFunVar;
            symbols[AC.lPolyFunVar].minpower = -MAXPOWER;
            symbols[AC.lPolyFunVar].maxpower =  MAXPOWER;
        }
        else if ( StrICmp(s,(UBYTE *)"expand") == 0 ) {
            WORD x = 0, etype = 2;
            if ( c != ',' ) {
```

```
                MesPrint("&Illegal option field in PolyRatFun statement.");
                return(1);
            }
            *t = c;
            while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
            s = t;
            t = SkipAName(s);
            if ( t == 0 ) goto NumErr;
            c = *t; *t = 0;
            if ( ( type = GetName(AC.varnames,s,&AC.lPolyFunVar,WITHAUTO) ) != CSYMBOL ) {
                MesPrint("&Illegal symbol %s in option field in PolyRatFun statement.",s);
                return(1);
            }
            *t = c;
            while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
            if ( *t > '9' || *t < '0' ) {
                MesPrint("&Illegal option field in PolyRatFun statement.");
                return(1);
            }
            while ( *t <= '9' && *t >= '0' ) x = 10*x + *t++ - '0';
            while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
            if ( *t != ')' ) {
                s = t;
                t = SkipAName(s);
                if ( t == 0 ) goto ParErr;
                c = *t; *t = 0;
                if ( StrICmp(s,(UBYTE *)"fixed") == 0 ) {
                    etype = 3;
                }
                else if ( StrICmp(s,(UBYTE *)"relative") == 0 ) {
                    etype = 2;
                }
                else {
                    MesPrint("&Illegal termination of option in PolyRatFun statement.");
                    return(1);
                }
                *t = c;
                while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
                if ( *t != ')' ) {
                    MesPrint("&Illegal termination of option in PolyRatFun statement.");
                    return(1);
                }
            }
            AR.PolyFunExp = AC.lPolyFunExp = etype;
            AR.PolyFunVar = AC.lPolyFunVar;
            AR.PolyFunPow = AC.lPolyFunPow = x;
            symbols[AC.lPolyFunVar].minpower = -MAXPOWER;
            symbols[AC.lPolyFunVar].maxpower =  MAXPOWER;
        }
        else {
ParErr:     MesPrint("&Illegal option %s in PolyRatFun statement.",s);
            return(1);
        }
        t++;
        while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
        if ( *t == 0 ) return(0);
    }
NumErr:;
    MesPrint("&PolyRatFun statement needs one or two commuting function(s) for its argument(s)");
    return(1);
}

/*
    #] CoPolyRatFun :
    #[ CoMerge :
*/

int CoMerge(UBYTE *inp)
{
    UBYTE *s = inp;
    int type;
    WORD numfunc, option = 0;
    if ( tolower(s[0]) == 'o' && tolower(s[1]) == 'n' && tolower(s[2]) == 'c' &&
         tolower(s[3]) == 'e' && tolower(s[4]) == ',' ) {
        option = 1; s += 5;
    }
    else if ( tolower(s[0]) == 'a' && tolower(s[1]) == 'l' && tolower(s[2]) == 'l' &&
         tolower(s[3]) == ',' ) {
        option = 0; s += 4;
    }
    if ( *s == '$' ) {
        if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
            numfunc = -numfunc;
        else {
            MesPrint("&%s is undefined",s);
            numfunc = AddDollar(s+1,DOLINDEX,&one,1);
            return(1);
```

```
        }
tests:  s = SkipAName(s);
        if ( *s != 0 ) {
            MesPrint("&Merge/shuffle should have a single function or $variable for its argument");
            return(1);
        }
    }
    else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
        numfunc += FUNCTION;
        goto tests;
    }
    else if ( type != -1 ) {
        if ( type != CDUBIOUS ) {
            NameConflict(type,s);
            type = MakeDubious(AC.varnames,s,&numfunc);
        }
        return(1);
    }
    else {
        MesPrint("&%s is not a function",s);
        numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
        return(1);
    }
    Add4Com(TYPEMERGE,numfunc,option);
    return(0);
}

/*
    #] CoMerge :
    #[ CoStuffle :

    Important for future options: The bit, given by 256 (bit 8) is reserved
    internally for keeping track of the sign in the number of Stuffle
    additions.
*/

int CoStuffle(UBYTE *inp)
{
    UBYTE *s = inp, *ss, c;
    int type;
    WORD numfunc, option = 0;
    if ( tolower(s[0]) == 'o' && tolower(s[1]) == 'n' && tolower(s[2]) == 'c' &&
         tolower(s[3]) == 'e' && tolower(s[4]) == ',' ) {
        option = 1; s += 5;
    }
    else if ( tolower(s[0]) == 'a' && tolower(s[1]) == 'l' && tolower(s[2]) == 'l' &&
         tolower(s[3]) == ',' ) {
        option = 0; s += 4;
    }
    ss = SkipAName(s);
    c = *ss; *ss = 0;
    if ( *s == '$' ) {
        if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
            numfunc = -numfunc;
        else {
            MesPrint("&%s is undefined",s);
            numfunc = AddDollar(s+1,DOLINDEX,&one,1);
            return(1);
        }
tests:  *ss = c;
        if ( *ss != '+' && *ss != '-' && ss[1] != 0 ) {
            MesPrint("&Stuffle should have a single function or $variable for its argument, followed by either + or -");
            return(1);
        }
        if ( *ss == '-' ) option += 2;
    }
    else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
        numfunc += FUNCTION;
        goto tests;
    }
    else if ( type != -1 ) {
        if ( type != CDUBIOUS ) {
            NameConflict(type,s);
            type = MakeDubious(AC.varnames,s,&numfunc);
        }
        return(1);
    }
    else {
        MesPrint("&%s is not a function",s);
        numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
        return(1);
    }
    Add4Com(TYPESTUFFLE,numfunc,option);
    return(0);
}

```

```
/*
    #] CoStuffle :
    #[ CoProcessBucket :
*/

int CoProcessBucket(UBYTE *s)
{
    LONG x;
    while ( *s == ',' || *s == '=' ) s++;
    ParseNumber(x,s)
    if ( *s && *s != ' ' && *s != '\t' ) {
        MesPrint("&Numerical value expected for ProcessBucketSize");
        return(1);
    }
    AC.ProcessBucketSize = x;
    return(0);
}

/*
    #] CoProcessBucket :
    #[ CoThreadBucket :
*/

int CoThreadBucket(UBYTE *s)
{
    LONG x;
    while ( *s == ',' || *s == '=' ) s++;
    ParseNumber(x,s)
    if ( *s && *s != ' ' && *s != '\t' ) {
        MesPrint("&Numerical value expected for ThreadBucketSize");
        return(1);
    }
    if ( x <= 0 ) {
        Warning("Negative of zero value not allowed for ThreadBucketSize. Adjusted to 1.");
        x = 1;
    }
    AC.ThreadBucketSize = x;
#ifdef WITHPTHREADS
    if ( AS.MultiThreaded ) MakeThreadBuckets(-1,1);
#endif
    return(0);
}

/*
    #] CoThreadBucket :
    #[ DoArgPlode :

    Syntax: a list of functions.
    If the functions have an argument it must be a function.
    In the case f(g) we treat f(g(...)) with g any argument.
      (not yet implemented)
*/

int DoArgPlode(UBYTE *s, int par)
{
    GETIDENTITY
    WORD numfunc, type, error = 0, *w, n;
    UBYTE *t,c;
    int i;
    w = AT.WorkPointer;
    *w++ = par;
    w++;
    while ( *s == ',' ) s++;
    while ( *s ) {
        if ( *s == '$' ) {
            MesPrint("&We don't do dollar variables yet in ArgImplode/ArgExplode");
            return(1);
        }
        t = s;
        if ( ( s = SkipAName(s) ) == 0 ) return(1);
        c = *s; *s = 0;
        if ( ( type = GetName(AC.varnames,t,&numfunc,WITHAUTO) ) == CFUNCTION ) {
            numfunc += FUNCTION;
        }
        else if ( type != -1 ) {
            if ( type != CDUBIOUS ) {
                NameConflict(type,t);
                type = MakeDubious(AC.varnames,t,&numfunc);
            }
            error = 1;
        }
        else {
            MesPrint("&%s is not a function",t);
            numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
            return(1);
        }
        *s = c;
```

```
        *w++ = numfunc;
        *w++ = FUNHEAD;
#if FUNHEAD > 2
        for ( i = 2; i < FUNHEAD; i++ ) *w++ = 0;
#endif
        if ( *s && *s != ',' ) {
            MesPrint("&Illegal character in ArgImplode/ArgExplode statement: %s",s);
            return(1);
        }
        while ( *s == ',' ) s++;
    }
    n = w - AT.WorkPointer;
    AT.WorkPointer[1] = n;
    AddNtoL(n,AT.WorkPointer);
    return(error);
}

/*
    #] DoArgPlode :
    #[ CoArgExplode :
*/

int CoArgExplode(UBYTE *s) { return(DoArgPlode(s,TYPEARGEXPLODE)); }

/*
    #] CoArgExplode :
    #[ CoArgImplode :
*/

int CoArgImplode(UBYTE *s) { return(DoArgPlode(s,TYPEARGIMPLODE)); }

/*
    #] CoArgImplode :
    #[ CoClearTable :
*/

int CoClearTable(UBYTE *s)
{
    UBYTE c, *t;
    int j, type, error = 0;
    WORD numfun;
    TABLES T, TT;
    if ( *s == 0 ) {
        MesPrint("&The ClearTable statement needs at least one (table) argument.");
        return(1);
    }
    while ( *s ) {
        t = s;
        s = SkipAName(s);
        c = *s; *s = 0;
        if ( ( ( type = GetName(AC.varnames,t,&numfun,WITHAUTO) ) != CFUNCTION )
        && type != CDUBIOUS ) {
nofunc:     MesPrint("&%s is not a table",t);
            error = 4;
            if ( type < 0 ) numfun = AddFunction(t,0,0,0,0,0,-1,-1);
            *s = c;
            if ( *s == ',' ) s++;
            continue;
        }
/*
        else if ( ( ( T = functions[numfun].tabl ) == 0 )
         || ( T->sparse == 0 ) ) goto nofunc;
*/
        else if ( ( T = functions[numfun].tabl ) == 0 ) goto nofunc;
        numfun += FUNCTION;
        *s = c;
        if ( *s == ',' ) s++;
/*
        Now we clear the table.
*/
        if ( T->sparse ) {
        if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
        for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
            finishcbuf(T->buffers[j]);
        }
        if ( T->buffers ) M_free(T->buffers,"Table buffers");
        finishcbuf(T->bufnum);

        T->boomlijst = 0;
        T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
        T->boomlijst = 0;
        T->bufnum = inicbufs();
        T->bufferssize = 8;
        T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"Table buffers");
        T->buffersfill = 0;
        T->buffers[T->buffersfill++] = T->bufnum;

```

```
        T->totind = 0;            /* At the moment there are this many */
        T->reserved = 0;

        ClearTableTree(T);

        if ( T->spare ) {
            if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
            T->tablepointers = 0;
            TT = T->spare;
            if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
            for (j = 0; j < TT->buffersfill; j++ ) {
                finishcbuf(TT->buffers[j]);
            }
            if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
            if ( TT->buffers )M_free(TT->buffers,"Table buffers");
            if ( TT->mm ) M_free(TT->mm,"tableminmax");
            if ( TT->flags ) M_free(TT->flags,"tableflags");
            M_free(TT,"table");
            SpareTable(T);
        }
        }
        else EmptyTable(T);
    }
    return(error);
}

/*
    #] CoClearTable :
    #[ CoDenominators :
*/

int CoDenominators(UBYTE *s)
{
    WORD numfun;
    int type;
    UBYTE *t = SkipAName(s), *t1;
    if ( t == 0 ) goto syntaxerror;
    t1 = t; while ( *t1 == ',' || *t1 == ' ' || *t1 == '\t' ) t1++;
    if ( *t1 ) goto syntaxerror;
    *t = 0;
    if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
    || ( functions[numfun].spec != 0 ) ) {
        if ( type < 0 ) {
            if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
                AddFunction(s,0,0,0,0,0,-1,-1);
        }
        goto syntaxerror;
    }
    Add3Com(TYPEDENOMINATORS,numfun+FUNCTION);
    return(0);
syntaxerror:
    MesPrint("&Denominators statement needs one regular function for its argument");
    return(1);
}

/*
    #] CoDenominators :
    #[ CoDropCoefficient :
*/

int CoDropCoefficient(UBYTE *s)
{
    if ( *s == 0 ) {
        Add2Com(TYPEDROPCOEFFICIENT)
        return(0);
    }
    MesPrint("&Illegal argument in DropCoefficient statement: '%s'",s);
    return(1);
}
/*
    #] CoDropCoefficient :
    #[ CoDropSymbols :
*/

int CoDropSymbols(UBYTE *s)
{
    if ( *s == 0 ) {
        Add2Com(TYPEDROPSYMBOLS)
        return(0);
    }
    MesPrint("&Illegal argument in DropSymbols statement: '%s'",s);
    return(1);
}
/*
    #] CoDropSymbols :
    #[ CoToPolynomial :

```

```
    Converts the current term as much as possible to symbols.
    Keeps a list of all objects converted to symbols in AM.sbufnum.
    Note that this cannot be executed in parallel because we have only
    a single compiler buffer for this. Hence we switch on the noparallel
    module option.

    Option(s):
        OnlyFunctions [,name1][,name2][,...,namem];
*/

int CoToPolynomial(UBYTE *inp)
{
    int error = 0;
    while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
    if ( ( AC.topolynomialflag & ~TOPOLYNOMIALFLAG ) != 0 ) {
        MesPrint("&ToPolynomial statement and FactArg statement are not allowed in the same module");
        return(1);
    }
    if ( AO.OptimizeResult.code != NULL ) {
        MesPrint("&Using ToPolynomial statement when there are still optimization results active.");
        MesPrint("&Please use #ClearOptimize instruction first.");
        MesPrint("&This will loose the optimized expression.");
        return(1);
    }
    if ( *inp == 0 ) {
        Add3Com(TYPETOPOLYNOMIAL,DOALL)
    }
    else {
        int numargs = 0;
        WORD *funnums = 0, type, num;
        UBYTE *s, c;
        s = SkipAName(inp);
        if ( s == 0 ) return(1);
        c = *s; *s = 0;
        if ( StrICmp(inp,(UBYTE *)"onlyfunctions") ) {
            MesPrint("&Illegal option %s in ToPolynomial statement",inp);
            *s = c;
            return(1);
        }
        *s = c;
        inp = s;
        while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
        s = inp;
        while ( *s ) s++;
/*
        Get definitely enough space for the numbers of the functions
*/
        funnums = (WORD *)Malloc1(((LONG)(s-inp)+3)*sizeof(WORD),"ToPlynomial");
        while ( *inp ) {
            s = SkipAName(inp);
            if ( s == 0 ) return(1);
            c = *s; *s = 0;
            type = GetName(AC.varnames,inp,&num,WITHAUTO);
            if ( type != CFUNCTION ) {
                MesPrint("&%s is not a function in ToPolynomial statement",inp);
                error = 1;
            }
            funnums[3+numargs++] = num+FUNCTION;
            *s = c;
            inp = s;
            while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
        }
        funnums[0] = TYPETOPOLYNOMIAL;
        funnums[1] = numargs+3;
        funnums[2] = ONLYFUNCTIONS;

        AddNtoL(numargs+3,funnums);
        if ( funnums ) M_free(funnums,"ToPolynomial");
    }
    AC.topolynomialflag |= TOPOLYNOMIALFLAG;
#ifdef WITHMPI
    /* In ParFORM, ToPolynomial has to be executed on the master. */
    AC.mparallelflag |= NOPARALLEL_CONVPOLY;
#endif
    return(error);
}

/*
    #] CoToPolynomial :
    #[ CoFromPolynomial :

    Converts the current term as much as possible back from extra symbols
    to their original values. Does not look inside functions.
*/

int CoFromPolynomial(UBYTE *inp)
{
```

```
    while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
    if ( *inp == 0 ) {
        if ( AO.OptimizeResult.code != NULL ) {
            MesPrint("&Using FromPolynomial statement when there are still optimization results active.");
            MesPrint("&Please use #ClearOptimize instruction first.");
            MesPrint("&This will loose the optimized expression.");
            return(1);
        }
        Add2Com(TYPEFROMPOLYNOMIAL)
        return(0);
    }
    MesPrint("&Illegal argument in FromPolynomial statement: '%s'",inp);
    return(1);
}

/*
    #] CoFromPolynomial :
    #[ CoArgToExtraSymbol :

    Converts the specified function arguments into extra symbols.

    Syntax: ArgToExtraSymbol [ToNumber] [<argument specifications>]
*/

int CoArgToExtraSymbol(UBYTE *s)
{
    CBUF *C = cbuf + AC.cbufnum;
    WORD *lhs;

    /* TODO: resolve interference with rational arithmetic. (#138) */
    if ( ( AC.topolynomialflag & ~TOPOLYNOMIALFLAG ) != 0 ) {
        MesPrint("&ArgToExtraSymbol statement and FactArg statement are not allowed in the same module");
        return(1);
    }
    if ( AO.OptimizeResult.code != NULL ) {
        MesPrint("&Using ArgToExtraSymbol statement when there are still optimization results active.");
        MesPrint("&Please use #ClearOptimize instruction first.");
        MesPrint("&This will loose the optimized expression.");
        return(1);
    }

    SkipSpaces(&s);
    int tonumber = ConsumeOption(&s, "tonumber");

    int ret = DoArgument(s,TYPEARGTOEXTRASYMBOL);
    if ( ret ) return(ret);

    /*
     * The "scale" parameter is unused. Instead, we put the "tonumber"
     * parameter.
     */
    lhs = C->lhs[C->numlhs];
    if ( lhs[4] != 1 ) {
        Warning("scale parameter (^n) is ignored in ArgToExtraSymbol");
    }
    lhs[4] = tonumber;

    AC.topolynomialflag |= TOPOLYNOMIALFLAG;  /* This flag is also used in ParFORM. */
#ifdef WITHMPI
    /*
     * In ParFORM, the conversion to extra symbols has to be performed on
     * the master.
     */
    AC.mparallelflag |= NOPARALLEL_CONVPOLY;
#endif

    return(0);
}

/*
    #] CoArgToExtraSymbol :
    #[ CoExtraSymbols :
*/

int CoExtraSymbols(UBYTE *inp)
{
    UBYTE *arg1, *arg2, c, *s;
    WORD i, j, type, number;
    while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
    if ( FG.cTable[*inp] != 0 ) {
        MesPrint("&Illegal argument in ExtraSymbols statement: '%s'",inp);
        return(1);
    }
    arg1 = inp;
```

```
    while ( FG.cTable[*inp] == 0 ) inp++;
    c = *inp; *inp = 0;
    if ( ( StrICmp(arg1,(UBYTE *)"array") == 0 )
            || ( StrICmp(arg1,(UBYTE *)"vector") == 0 ) ) {
        AC.extrasymbols = 1;
    }
    else if ( StrICmp(arg1,(UBYTE *)"underscore") == 0 ) {
        AC.extrasymbols = 0;
    }
/*
    else if ( StrICmp(arg1,(UBYTE *)"nothing") == 0 ) {
        AC.extrasymbols = 2;
    }
*/
    else {
        MesPrint("&Illegal keyword in ExtraSymbols statement: '%s'",arg1);
        return(1);
    }
    *inp = c;
    while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
    if ( FG.cTable[*inp] != 0 ) {
        MesPrint("&Illegal argument in ExtraSymbols statement: '%s'",inp);
        return(1);
    }
    arg2 = inp;
    while ( FG.cTable[*inp] <= 1 ) inp++;
    if ( *inp != 0 ) {
        MesPrint("&Illegal end of ExtraSymbols statement: '%s'",inp);
        return(1);
    }
/*
        Now check whether this object has been declared already.
        That would not be allowed.
*/
    if ( AC.extrasymbols == 1 ) {
        type = GetName(AC.varnames,arg2,&number,NOAUTO);
        if ( type != NAMENOTFOUND ) {
            MesPrint("&ExtraSymbols statement: '%s' has already been declared before",arg2);
            return(1);
        }
    }
    else if ( AC.extrasymbols == 0 ) {
        if ( *arg2 == 'N' ) {
            s = arg2+1;
            while ( FG.cTable[*s] == 1 ) s++;
            if ( *s == 0 ) {
                MesPrint("&ExtraSymbols statement: '%s' creates conflicts with summed indices",arg2);
                return(1);
            }
        }
    }
    if ( AC.extrasym ) { M_free(AC.extrasym,"extrasym"); AC.extrasym = 0; }
    i = inp - arg2 + 1;
    AC.extrasym = (UBYTE *)Malloc1(i*sizeof(UBYTE),"extrasym");
    for ( j = 0; j < i; j++ ) AC.extrasym[j] = arg2[j];
    return(0);
}

/*
    #] CoExtraSymbols :
    #[ GetIfDollarFactor :
*/

WORD *GetIfDollarFactor(UBYTE **inp, WORD *w)
{
    LONG x;
    WORD number;
    UBYTE *name, c, *s;
    s = *inp;
    if ( FG.cTable[*s] == 1 ) {
        x = 0;
        while ( FG.cTable[*s] == 1 ) {
            x = 10*x + *s++ - '0';
            if ( x >= MAXPOSITIVE ) {
                MesPrint("&Value in dollar factor too large");
                while ( FG.cTable[*s] == 1 ) s++;
                *inp = s;
                return(0);
            }
        }
        *w++ = IFDOLLAREXTRA;
        *w++ = 3;
        *w++ = -x-1;
        *inp = s;
        return(w);
    }
    if ( *s != '$' ) {
```

```
        MesPrint("&Factor indicator for $-variable should be a number or a $-variable.");
        return(0);
    }
    s++; name = s;
    while ( FG.cTable[*s] < 2 ) s++;
    c = *s; *s = 0;
    if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
        MesPrint("&dollar in if statement should have been defined previously");
        return(0);
    }
    *s = c;
    *w++ = IFDOLLAREXTRA;
    *w++ = 3;
    *w++ = number;
    if ( c == '[' ) {
        s++;
        *inp = s;
        if ( ( w = GetIfDollarFactor(inp,w) ) == 0 ) return(0);
        s = *inp;
        if ( *s != ']' ) {
            MesPrint("&unmatched [] in $ in if statement");
            return(0);
        }
        s++;
        *inp = s;
    }
    return(w);
}

/*
    #] GetIfDollarFactor :
    #[ GetDoParam :
*/

UBYTE *GetDoParam(UBYTE *inp, WORD **wp, int par)
{
    LONG x;
    WORD number;
    UBYTE *name, c;
    if ( FG.cTable[*inp] == 1 ) {
        x = 0;
        while ( *inp >= '0' && *inp <= '9' ) {
            x = 10*x + *inp++ - '0';
            if ( x > MAXPOSITIVE ) {
                if ( par == -1 ) {
                    MesPrint("&Value in dollar factor too large");
                }
                else {
                    MesPrint("&Value in do loop boundaries too large");
                }
                while ( FG.cTable[*inp] == 1 ) inp++;
                return(0);
            }
        }
        if ( par > 0 ) {
            *(*wp)++ = SNUMBER;
            *(*wp)++ = (WORD)x;
        }
        else {
            *(*wp)++ = DOLLAREXPR2;
            *(*wp)++ = -((WORD)x)-1;
        }
        return(inp);
    }
    if ( *inp != '$' ) {
        return(0);
    }
    inp++; name = inp;
    while ( FG.cTable[*inp] < 2 ) inp++;
    c = *inp; *inp = 0;
    if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
        if ( par == -1 ) {
            MesPrint("&dollar in print statement should have been defined previously");
        }
        else {
            MesPrint("&dollar in do loop boundaries should have been defined previously");
        }
        return(0);
    }
    *inp = c;
    if ( par > 0 ) {
        *(*wp)++ = DOLLAREXPRESSION;
        *(*wp)++ = number;
    }
    else {
        *(*wp)++ = DOLLAREXPR2;
        *(*wp)++ = number;
```

```
    }
    if ( c == '[' ) {
        inp++;
        inp = GetDoParam(inp,wp,0);
        if ( inp == 0 ) return(0);
        if ( *inp != ']' ) {
            if ( par == -1 ) {
                MesPrint("&unmatched [] in $ in print statement");
            }
            else {
                MesPrint("&unmatched [] in do loop boundaries");
            }
            return(0);
        }
        inp++;
    }
    return(inp);
}

/*
    #] GetDoParam :
    #[ CoDo :
*/

int CoDo(UBYTE *inp)
{
    GETIDENTITY
    CBUF *C = cbuf+AC.cbufnum;
    WORD *w, numparam;
    int error = 0, i;
    UBYTE *name, c;
    if ( AC.doloopstack == 0 ) {
        AC.doloopstacksize = 20;
        AC.doloopstack = (WORD *)Malloc1(AC.doloopstacksize*2*sizeof(WORD),"doloop stack");
        AC.doloopnest = AC.doloopstack + AC.doloopstacksize;
    }
    if ( AC.dolooplevel >= AC.doloopstacksize ) {
        WORD *newstack, *newnest, newsize;
        newsize = AC.doloopstacksize * 2;
        newstack = (WORD *)Malloc1(newsize*2*sizeof(WORD),"doloop stack");
        newnest = newstack + newsize;
        for ( i = 0; i < newsize; i++ ) {
            newstack[i] = AC.doloopstack[i];
            newnest[i] = AC.doloopnest[i];
        }
        M_free(AC.doloopstack,"doloop stack");
        AC.doloopstack = newstack;
        AC.doloopnest = newnest;
        AC.doloopstacksize = newsize;
    }
    AC.doloopnest[AC.dolooplevel] = NestingChecksum();

    w = AT.WorkPointer;
    *w++ = TYPEDOLOOP;
    w++; /* Space for the length of the statement */
/*
    Now the $loopvariable
*/
    while ( *inp == ',' ) inp++;
    if ( *inp != '$' ) {
        error = 1;
        MesPrint("&do loop parameter should be a dollar variable");
    }
    else {
        inp++;
        name = inp;
        if ( FG.cTable[*inp] != 0 ) {
            error = 1;
            MesPrint("&illegal name for do loop parameter");
        }
        while ( FG.cTable[*inp] < 2 ) inp++;
        c = *inp; *inp = 0;
        if ( GetName(AC.dollarnames,name,&numparam,NOAUTO) == NAMENOTFOUND ) {
            numparam = AddDollar(name,DOLUNDEFINED,0,0);
        }
        *w++ = numparam;
        *inp = c;
        AddPotModdollar(numparam);
    }
    w++;  /* space for the level of the enddo statement */
    while ( *inp == ',' ) inp++;
    if ( *inp != '=' ) goto IllSyntax;
    inp++;
    while ( *inp == ',' ) inp++;
/*
    The start value
*/
```

```
    inp = GetDoParam(inp,&w,1);
    if ( inp == 0 || *inp != ',' ) goto IllSyntax;
    while ( *inp == ',' ) inp++;
/*
    The end value
*/
    inp = GetDoParam(inp,&w,1);
    if ( inp == 0 || ( *inp != 0 && *inp != ',' ) ) goto IllSyntax;
/*
    The increment value
*/
    if ( *inp != ',' ) {
        if ( *inp == 0 ) { *w++ = SNUMBER; *w++ = 1; }
        else goto IllSyntax;
    }
    else {
        while ( *inp == ',' ) inp++;
        inp = GetDoParam(inp,&w,1);
    }
    if ( inp == 0 || *inp != 0 ) goto IllSyntax;
    *w = 0;
    AT.WorkPointer[1] = w - AT.WorkPointer;
/*
    Put away and set information for placing enddo information.
*/
    AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
    AC.doloopstack[AC.doloopleveel++] = C->numlhs;

    return(error);

IllSyntax:
    MesPrint("&Illegal syntax for do statement");
    return(1);
}

/*
    #] CoDo :
    #[ CoEndDo :
*/

int CoEndDo(UBYTE *inp)
{
    CBUF *C = cbuf+AC.cbufnum;
    WORD scratch[3];
    while ( *inp == ',' ) inp++;
    if ( *inp ) {
        MesPrint("&Illegal syntax for EndDo statement");
        return(1);
    }
    if ( AC.doloopleveel <= 0 ) {
        MesPrint("&EndDo without corresponding Do statement");
        return(1);
    }
    AC.doloopleveel--;
    scratch[0] = TYPEENDDOLOOP;
    scratch[1] = 3;
    scratch[2] = AC.doloopstack[AC.doloopleveel];
    AddNtoL(3,scratch);
    cbuf[AC.cbufnum].lhs[AC.doloopstack[AC.doloopleveel]][3] = C->numlhs;
    if ( AC.doloopnest[AC.doloopleveel] != NestingChecksum() ) {
        MesNesting();
        return(1);
    }
    return(0);
}

/*
    #] CoEndDo :
    #[ CoFactDollar :
*/

int CoFactDollar(UBYTE *inp)
{
    WORD numdollar;
    if ( *inp == '$' ) {
        if ( GetName(AC.dollarnames,inp+1,&numdollar,NOAUTO) != CDOLLAR ) {
            MesPrint("&%s is undefined",inp);
            numdollar = AddDollar(inp+1,DOLINDEX,&one,1);
            return(1);
        }
        inp = SkipAName(inp+1);
        if ( *inp != 0 ) {
            MesPrint("&FactDollar should have a single $variable for its argument");
            return(1);
        }
        AddPotModdollar(numdollar);
    }
```

```
    else {
        MesPrint("&%s is not a $-variable",inp);
        return(1);
    }
    Add3Com(TYPEFACTOR,numdollar);
    return(0);
}

/*
    #] CoFactDollar :
    #[ CoFactorize :
*/

int CoFactorize(UBYTE *s) { return(DoFactorize(s,1)); }

/*
    #] CoFactorize :
    #[ CoNFactorize :
*/

int CoNFactorize(UBYTE *s) { return(DoFactorize(s,0)); }

/*
    #] CoNFactorize :
    #[ CoUnFactorize :
*/

int CoUnFactorize(UBYTE *s) { return(DoFactorize(s,3)); }

/*
    #] CoUnFactorize :
    #[ CoNUnFactorize :
*/

int CoNUnFactorize(UBYTE *s) { return(DoFactorize(s,2)); }

/*
    #] CoNUnFactorize :
    #[ DoFactorize :
*/

int DoFactorize(UBYTE *s,int par)
{
    EXPRESSIONS e;
    WORD i;
    WORD number;
    UBYTE *t, c;
    int error = 0, keepzeroflag = 0;
    if ( *s == '(' ) {
        s++;
        while ( *s != ')' && *s ) {
            if ( FG.cTable[*s] == 0 ) {
                t = s; while ( FG.cTable[*s] == 0 ) s++;
                c = *s; *s = 0;
                if ( StrICmp((UBYTE *)"keepzero",t) == 0 ) {
                    keepzeroflag = 1;
                }
                else {
                    MesPrint("&Illegal option in [N][Un]Factorize statement: %s",t);
                    error = 1;
                }
                *s = c;
            }
            while ( *s == ',' ) s++;
            if ( *s && *s != ')' && FG.cTable[*s] != 0 ) {
                MesPrint("&Illegal character in option field of [N][Un]Factorize statement");
                error = 1;
                return(error);
            }
        }
        if ( *s ) s++;
        while ( *s == ',' || *s == ' ' ) s++;
    }
    if ( *s == 0 ) {
        for ( i = NumExpressions-1; i >= 0; i-- ) {
            e = Expressions+i;
            if ( e->replace >= 0 ) {
                e = Expressions + e->replace;
            }
            if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
            || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
          || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION
            ) {
                switch ( par ) {
                    case 0:
                        e->vflags &= ~TOBEFACTORED;
                        break;
```

```
                    case 1:
                        e->vflags |=  TOBEFACTORED;
                        e->vflags &= ~TOBEUNFACTORED;
                        break;
                    case 2:
                        e->vflags &= ~TOBEUNFACTORED;
                        break;
                    case 3:
                        e->vflags |=  TOBEUNFACTORED;
                        e->vflags &= ~TOBEFACTORED;
                        break;
                }
            }
            if ( ( e->vflags & TOBEFACTORED ) != 0 ) {
                if ( keepzeroflag ) e->vflags |=  KEEPZERO;
                else                e->vflags &= ~KEEPZERO;
            }
            else                    e->vflags &= ~KEEPZERO;
        }
    }
    else {
        for(;;) {   /* Look for a (comma separated) list of variables */
            while ( *s == ',' ) s++;
            if ( *s == 0 ) break;
            if ( *s == '[' || FG.cTable[*s] == 0 ) {
                t = s;
                if ( ( s = SkipAName(s) ) == 0 ) {
                    MesPrint("&Improper name for an expression: '%s'",t);
                    return(1);
                }
                c = *s; *s = 0;
                if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
                    e = Expressions+number;
                    if ( e->replace >= 0 ) {
                        e = Expressions + e->replace;
                    }
                    if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
                    || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
                    || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION
                    ) {
                        switch ( par ) {
                            case 0:
                                e->vflags &= ~TOBEFACTORED;
                                break;
                            case 1:
                                e->vflags |=  TOBEFACTORED;
                                e->vflags &= ~TOBEUNFACTORED;
                                break;
                            case 2:
                                e->vflags &= ~TOBEUNFACTORED;
                                break;
                            case 3:
                                e->vflags |=  TOBEUNFACTORED;
                                e->vflags &= ~TOBEFACTORED;
                                break;
                        }
                    }
                    if ( ( e->vflags & TOBEFACTORED ) != 0 ) {
                        if ( keepzeroflag ) e->vflags |=  KEEPZERO;
                        else                e->vflags &= ~KEEPZERO;
                    }
                    else                    e->vflags &= ~KEEPZERO;
                }
                else if ( GetName(AC.varnames,t,&number,NOAUTO) != NAMENOTFOUND ) {
                    MesPrint("&%s is not an expression",t);
                    error = 1;
                }
                *s = c;
            }
            else {
                MesPrint("&Illegal object in (N)Factorize statement");
                error = 1;
                while ( *s && *s != ',' ) s++;
                if ( *s == 0 ) break;
            }
        }

    }
    return(error);
}

/*
    #] DoFactorize :
    #[ CoOptimizeOption :

*/
```

```
int CoOptimizeOption(UBYTE *s)
{
    UBYTE *name, *t1, *t2, c1, c2, *value, *u;
    int error = 0, x;
    double d;
    while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
    while ( *s ) {
        name = s; while ( FG.cTable[*s] == 0 ) s++;
        t1 = s; c1 = *t1;
        while ( *s == ' ' || *s == '\t' ) s++;
        if ( *s != '=' ) {
correctuse:
            MesPrint("&Correct use in Format,Optimize statement is Optionname=value");
            error = 1;
            while ( *s == ' ' || *s == ',' || *s == '\t' || *s == '=' ) s++;
            *t1 = c1;
            continue;
        }
        *t1 = 0;
        s++;
        while ( *s == ' ' || *s == '\t' ) s++;
        if ( *s == 0 ) goto correctuse;
        value = s;
        while ( FG.cTable[*s] <= 1 || *s=='.' || *s=='*' || *s == '(' || *s == ')' ) {
            if ( *s == '(' ) { SKIPBRA4(s) }
            s++;
        }
        t2 = s; c2 = *t2;
        while ( *s == ' ' || *s == '\t' ) s++;
        if ( *s && *s != ',' ) goto correctuse;
        if ( *s ) {
            s++;
            while ( *s == ' ' || *s == '\t' ) s++;
        }
        *t2 = 0;
/*
        Now we have name=value with name and value zero terminated strings.
*/
        if ( StrICmp(name,(UBYTE *)"horner") == 0 ) {
            if ( StrICmp(value,(UBYTE *)"occurrence") == 0 ) {
                AO.Optimize.horner = O_OCCURRENCE;
            }
            else if ( StrICmp(value,(UBYTE *)"mcts") == 0 ) {
                AO.Optimize.horner = O_MCTS;
            }
            else if ( StrICmp(value,(UBYTE *)"sa") == 0 ) {
                AO.Optimize.horner = O_SIMULATED_ANNEALING;
            }
            else {
                AO.Optimize.horner = -1;
                MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
                error = 1;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"hornerdirection") == 0 ) {
            if ( StrICmp(value,(UBYTE *)"forward") == 0 ) {
                AO.Optimize.hornerdirection = O_FORWARD;
            }
            else if ( StrICmp(value,(UBYTE *)"backward") == 0 ) {
                AO.Optimize.hornerdirection = O_BACKWARD;
            }
            else if ( StrICmp(value,(UBYTE *)"forwardorbackward") == 0 ) {
                AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
            }
            else if ( StrICmp(value,(UBYTE *)"forwardandbackward") == 0 ) {
                AO.Optimize.hornerdirection = O_FORWARDANDBACKWARD;
            }
            else {
                AO.Optimize.method = -1;
                MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
                error = 1;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"method") == 0 ) {
            if ( StrICmp(value,(UBYTE *)"none") == 0 ) {
                AO.Optimize.method = O_NONE;
            }
            else if ( StrICmp(value,(UBYTE *)"cse") == 0 ) {
                AO.Optimize.method = O_CSE;
            }
            else if ( StrICmp(value,(UBYTE *)"csegreedy") == 0 ) {
                AO.Optimize.method = O_CSEGREEDY;
            }
            else if ( StrICmp(value,(UBYTE *)"greedy") == 0 ) {
                AO.Optimize.method = O_GREEDY;
            }
            else {
```

```
                AO.Optimize.method = -1;
                MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
                error = 1;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"timelimit") == 0 ) {
            x = 0;
            u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            if ( *u != 0 ) {
                MesPrint("&Option TimeLimit in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.mctstimelimit = 0;
                AO.Optimize.greedytimelimit = 0;
                error = 1;
            }
            else {
                AO.Optimize.mctstimelimit = x/2;
                AO.Optimize.greedytimelimit = x/2;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"mctstimelimit") == 0 ) {
            x = 0;
            u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            if ( *u != 0 ) {
                MesPrint("&Option MCTSTimeLimit in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.mctstimelimit = 0;
                error = 1;
            }
            else {
                AO.Optimize.mctstimelimit = x;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"mctsnumexpand") == 0 ) {
            int y;
            x = 0;
            u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            if ( *u == '*' || *u == 'x' || *u == 'X' ) {
                u++; y = x;
                x = 0;
                while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            }
            else { y = 1; }
            if ( *u != 0 ) {
                MesPrint("&Option MCTSNumExpand in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.mctsnumexpand= 0;
                AO.Optimize.mctsnumrepeat= 1;
                error = 1;
            }
            else {
                AO.Optimize.mctsnumexpand= x;
                AO.Optimize.mctsnumrepeat= y;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"mctsnumrepeat") == 0 ) {
            x = 0;
            u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            if ( *u != 0 ) {
                MesPrint("&Option MCTSNumExpand in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.mctsnumrepeat= 1;
                error = 1;
            }
            else {
                AO.Optimize.mctsnumrepeat= x;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"mctsnumkeep") == 0 ) {
            x = 0;
            u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            if ( *u != 0 ) {
                MesPrint("&Option MCTSNumKeep in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.mctsnumkeep= 0;
                error = 1;
            }
            else {
                AO.Optimize.mctsnumkeep= x;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"mctsconstant") == 0 ) {
            d = 0;
            if ( sscanf ((char*)value, "%lf", &d) != 1 ) {
                MesPrint("&Option MCTSConstant in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.mctsconstant.fval = 0;
```

```
                error = 1;
            }
            else {
                AO.Optimize.mctsconstant.fval = d;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"greedytimelimit") == 0 ) {
            x = 0;
            u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            if ( *u != 0 ) {
                MesPrint("&Option GreedyTimeLimit in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.greedytimelimit = 0;
                error = 1;
            }
            else {
                AO.Optimize.greedytimelimit = x;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"greedyminnum") == 0 ) {
            x = 0;
            u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            if ( *u != 0 ) {
                MesPrint("&Option GreedyMinNum in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.greedyminnum= 0;
                error = 1;
            }
            else {
                AO.Optimize.greedyminnum= x;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"greedymaxperc") == 0 ) {
            x = 0;
            u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            if ( *u != 0 ) {
                MesPrint("&Option GreedyMaxPerc in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.greedymaxperc= 0;
                error = 1;
            }
            else {
                AO.Optimize.greedymaxperc= x;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"stats") == 0 ) {
            if ( StrICmp(value,(UBYTE *)"on") == 0 ) {
                AO.Optimize.printstats = 1;
            }
            else if ( StrICmp(value,(UBYTE *)"off") == 0 ) {
                AO.Optimize.printstats = 0;
            }
            else {
                AO.Optimize.printstats = 0;
                MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
                error = 1;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"printscheme") == 0 ) {
            if ( StrICmp(value,(UBYTE *)"on") == 0 ) {
                AO.Optimize.schemeflags |= 1;
            }
            else if ( StrICmp(value,(UBYTE *)"off") == 0 ) {
                AO.Optimize.schemeflags &= ~1;
            }
            else {
                AO.Optimize.schemeflags &= ~1;
                MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
                error = 1;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"debugflag") == 0 ) {
/*
            This option is for debugging purposes only. Not in the manual!
            0x1: Print statements in reverse order.
            0x2: Print the scheme of the variables.
*/
            x = 0;
            u = value;
            if ( FG.cTable[*u] == 1 ) {
                while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
                if ( *u != 0 ) {
                    MesPrint("&Numerical value for DebugFlag in Format,Optimize statement should be a nonnegative number: %s",value);
                    AO.Optimize.debugflags = 0;
                    error = 1;
                }
```

```
                else {
                    AO.Optimize.debugflags = x;
                }
            }
            else if ( StrICmp(value,(UBYTE *)"on") == 0 ) {
                AO.Optimize.debugflags = 1;
            }
            else if ( StrICmp(value,(UBYTE *)"off") == 0 ) {
                AO.Optimize.debugflags = 0;
            }
            else {
                AO.Optimize.debugflags = 0;
                MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
                error = 1;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"scheme") == 0 ) {
            UBYTE *ss, *s1, c;
            WORD type, numsym;
            AO.schemenum = 0;
            u = value;
            if ( *u != '(' ) {
noscheme:
                MesPrint("&Option Scheme in Format,Optimize statement should be an array of names or integers between (): %s",value);
                error = 1;
                break;
            }
            u++; ss = u;
            while ( *ss == ' ' || *ss == '\t' || *ss == ',' ) ss++;
            if ( FG.cTable[*ss] == 0 || *ss == '$' || *ss == '[' ) { /* Name */
                s1 = u; SKIPBRA3(s1)
                if ( *s1 != ')' ) goto noscheme;
                while ( ss < s1 ) { if ( *ss++ == ',' ) AO.schemenum++; }
                *ss++ = 0; while ( *ss == ' ' ) ss++;
                if ( *ss != 0 ) goto noscheme;
                ss = u;
                if ( AO.schemenum < 1 ) {
                    MesPrint("&Option Scheme in Format,Optimize statement should have at least one name or number between ()");
                    error = 1;
                    break;
                }
                if ( AO.inscheme ) M_free(AO.inscheme,"Horner input scheme");
                AO.inscheme = (WORD *)Malloc1((AO.schemenum+1)*sizeof(WORD),"Horner input scheme");
                while ( *ss == ' ' || *ss == '\t' || *ss == ',' ) ss++;
                AO.schemenum = 0;
                for(;;) {
                    if ( *ss == 0 ) break;
                    s1 = ss; ss = SkipAName(s1); c = *ss; *ss = 0;

                    if ( ss[-1] == '_' ) {
/*
                        Now AC.extrasym followed by a number and _
*/
                        UBYTE *u1, *u2;
                        u1 = s1; u2 = AC.extrasym;
                        while ( *u1 == *u2 ) { u1++; u2++; }
                        if ( *u2 == 0 ) { /* Good start */
                            numsym = 0;
                            while ( *u1 >= '0' && *u1 <= '9' ) numsym = 10*numsym + *u1++ - '0';
                            if ( u1 != ss-1 || numsym == 0 || AC.extrasymbols != 0 ) {
                                MesPrint("&Improper use of extra symbol in scheme format option");
                                goto noscheme;
                            }
                            numsym = MAXVARIABLES-numsym;
                            ss++;
                            goto GotTheNumber;
                        }
                    }
                    else if ( *s1 == '$' ) {
                        GETIDENTITY
                        int numdollar;
                        if ( ( numdollar = GetDollar(s1+1) ) < 0 ) {
                            MesPrint("&Undefined variable %s",s1);
                            error = 5;
                        }
                        else if ( ( numsym = DolToSymbol(BHEAD numdollar) ) < 0 ) {
                            MesPrint("&$%s does not evaluate to a symbol",s1);
                            error = 5;
                        }
                        *ss = c;
                        goto GotTheNumber;
                    }
                    else if ( c == '(' ) {
                        if ( StrCmp(s1,AC.extrasym) == 0 ) {
                            if ( (AC.extrasymbols&1) != 1 ) {
```

```
                                MesPrint("&Improper use of extra symbol in scheme format option");
                                goto noscheme;
                            }
                            *ss++ = c;
                            numsym = 0;
                            while ( *ss >= '0' && *ss <= '9' ) numsym = 10*numsym + *ss++ - '0';
                            if ( *ss != ')' ) {
                                MesPrint("&Extra symbol should have a number for its argument.");
                                goto noscheme;
                            }
                            numsym = MAXVARIABLES-numsym;
                            ss++;
                            goto GotTheNumber;
                        }
                    }
                    type = GetName(AC.varnames,s1,&numsym,WITHAUTO);
                    if ( ( type != CSYMBOL ) && type != CDUBIOUS ) {
                        MesPrint("&%s is not a symbol",s1);
                        error = 4;
                        if ( type < 0 ) numsym = AddSymbol(s1,-MAXPOWER,MAXPOWER,0,0);
                    }
                    *ss = c;
GotTheNumber:
                    AO.inscheme[AO.schemenum++] = numsym;
                    while ( *ss == ' ' || *ss == '\t' || *ss == ',' ) ss++;
                }
            }
        }
        else if ( StrICmp(name,(UBYTE *)"mctsdecaymode") == 0 ) {
            x = 0;
            u = value;
            if ( FG.cTable[*u] == 1 ) {
                while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
                if ( *u != 0 ) {
                    MesPrint("&Option MCTSDecayMode in Format,Optimize statement should be a nonnegative integer: %s",value);
                    AO.Optimize.mctsdecaymode = 0;
                    error = 1;
                }
                else {
                    AO.Optimize.mctsdecaymode = x;
                }
            }
            else {
                AO.Optimize.mctsdecaymode = 0;
                MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
                error = 1;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"saiter") == 0 ) {
            x = 0;
            u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
            if ( *u != 0 ) {
                MesPrint("&Option SAIter in Format,Optimize statement should be a positive integer: %s",value);
                AO.Optimize.saIter = 0;
                error = 1;
            }
            else {
                AO.Optimize.saIter= x;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"samaxt") == 0 ) {
            d = 0;
            if ( sscanf ((char*)value, "%lf", &d) != 1 ) {
                MesPrint("&Option SAMaxT in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.saMaxT.fval = 0;
                error = 1;
            }
            else {
                AO.Optimize.saMaxT.fval = d;
            }
        }
        else if ( StrICmp(name,(UBYTE *)"samint") == 0 ) {
            d = 0;
            if ( sscanf ((char*)value, "%lf", &d) != 1 ) {
                MesPrint("&Option SAMinT in Format,Optimize statement should be a positive number: %s",value);
                AO.Optimize.saMinT.fval = 0;
                error = 1;
            }
            else {
                AO.Optimize.saMinT.fval = d;
            }
        }
        else {
```

```
            MesPrint("&Unrecognized option name in Format,Optimize statement: %s",name);
            error = 1;
        }
        *t1 = c1; *t2 = c2;
    }
    return(error);
}

/*
    #] CoOptimizeOption :
    #[ DoPutInside :

    Syntax:
        PutIn[side],functionname[,brackets]  -> par = 1
        AntiPutIn[side],functionname,antibrackets  -> par = -1
*/

int CoPutInside(UBYTE *inp) { return(DoPutInside(inp,1)); }
int CoAntiPutInside(UBYTE *inp) { return(DoPutInside(inp,-1)); }

int DoPutInside(UBYTE *inp, int par)
{
    GETIDENTITY
    UBYTE *p, c;
    WORD *to, type, c1,c2,funnum, *WorkSave;
    int error = 0;
    while ( *inp == ' ' || *inp == '\t' || *inp == ',' ) inp++;
/*
    First we need the name of a function. (Not a tensor or table!)
*/
    p = SkipAName(inp);
    if ( p == 0 ) return(1);
    c = *p; *p = 0;
    type = GetName(AC.varnames,inp,&funnum,WITHAUTO);
    if ( type != CFUNCTION || functions[funnum].tabl != 0 || functions[funnum].spec ) {
        MesPrint("&PutInside/AntiPutInside expects a regular function for its first argument");
        MesPrint("&Argument is %s",inp);
        error = 1;
    }
    funnum += FUNCTION;
    *p = c;
    inp = p;
    while ( *inp == ' ' || *inp == '\t' || *inp == ',' ) inp++;
    if ( *inp == 0 ) {
        if ( par == 1 ) {
            WORD tocompiler[4];
            tocompiler[0] = TYPEPUTINSIDE;
            tocompiler[1] = 4;
            tocompiler[2] = 0;
            tocompiler[3] = funnum;
            AddNtoL(4,tocompiler);
        }
        else {
            MesPrint("&AntiPutInside needs inside information.");
            error = 1;
        }
        return(error);
    }
    WorkSave = to = AT.WorkPointer;
    *to++ = TYPEPUTINSIDE;
    *to++ = 4;
    *to++ = par;
    *to++ = funnum;
    to++;
    while ( *inp ) {
        while ( *inp == ' ' || *inp == '\t' || *inp == ',' ) inp++;
        if ( *inp == 0 ) break;
        p = SkipAName(inp);
        if ( p == 0 ) { error = 1; break; }
        c = *p; *p = 0;
        type = GetName(AC.varnames,inp,&c1,WITHAUTO);
        if ( c == '.' ) {
            if ( type == CVECTOR || type == CDUBIOUS ) {
                *p++ = c;
                inp = p;
                p = SkipAName(inp);
                if ( p == 0 ) return(1);
                c = *p; *p = 0;
                type = GetName(AC.varnames,inp,&c2,WITHAUTO);
                if ( type != CVECTOR && type != CDUBIOUS ) {
                    MesPrint("&Not a vector in dotproduct in PutInside/AntiPutInside statement: %s",inp);
                    error = 1;
                }
                else type = CDOTPRODUCT;
            }
            else {
```

```
                MesPrint("&Illegal use of . after %s in PutInside/AntiPutInside statement",inp);
                error = 1;
                *p = c; inp = p;
                continue;
            }
        }
        switch ( type ) {
            case CSYMBOL :
                *to++ = SYMBOL; *to++ = 4; *to++ = c1; *to++ = 1; break;
            case CVECTOR :
                *to++ = INDEX; *to++ = 3; *to++ = AM.OffsetVector + c1; break;
            case CFUNCTION :
                *to++ = c1+FUNCTION; *to++ = FUNHEAD; *to++ = 0;
                FILLFUN3(to)
                break;
            case CDOTPRODUCT :
                *to++ = DOTPRODUCT; *to++ = 5; *to++ = c1 + AM.OffsetVector;
                *to++ = c2 + AM.OffsetVector; *to++ = 1; break;
            case CDELTA :
                *to++ = DELTA; *to++ = 4; *to++ = EMPTYINDEX; *to++ = EMPTYINDEX; break;
            default :
                MesPrint("&Illegal variable request for %s in PutInside/AntiPutInside statement",inp);
                error = 1; break;
        }
        *p = c;
        inp = p;
    }
    *to++ = 1; *to++ = 1; *to++ = 3;
    AT.WorkPointer[1] = to - AT.WorkPointer;
    AT.WorkPointer[4] = AT.WorkPointer[1]-4;
    AT.WorkPointer = to;
    AC.BracketNormalize = 1;
    if ( Normalize(BHEAD WorkSave+4) ) { error = 1; }
    else {
        WorkSave[1] = WorkSave[4]+4;
        to = WorkSave + WorkSave[1] - 1;
        c1 = ABS(*to);
        WorkSave[1] -= c1;
        WorkSave[4] -= c1;
        AddNtoL(WorkSave[1],WorkSave);
    }
    AC.BracketNormalize = 0;
    AT.WorkPointer = WorkSave;
    return(error);
}

/*
    #] DoPutInside :
*/
```

4.11 compiler.c File Reference

`#include "form3.h"`

Include dependency graph for compiler.c:

Data Structures

- struct SuBbUf

Macros

- #define OPTION0 1
- #define OPTION1 2
- #define OPTION2 3
- #define REDUCESUBEXPBUFFERS

Typedefs

- typedef struct SuBbUf **SUBBUF**

Functions

- VOID inictable ()
- KEYWORD ∗ findcommand (UBYTE ∗in)
- int ParenthesesTest (UBYTE ∗sin)
- UBYTE ∗ SkipAName (UBYTE ∗s)
- UBYTE ∗ IsRHS (UBYTE ∗s, UBYTE c)
- int IsIdStatement (UBYTE ∗s)
- int CompileAlgebra (UBYTE ∗s, int leftright, WORD ∗prototype)
- int CompileStatement (UBYTE ∗in)
- int TestTables ()
- int CompileSubExpressions (SBYTE ∗tokens)
- int CodeGenerator (SBYTE ∗tokens)
- int CompleteTerm (WORD ∗term, UWORD ∗numer, UWORD ∗denom, WORD nnum, WORD nden, int sign)
- int CodeFactors (SBYTE ∗tokens)
- WORD GenerateFactors (WORD n, WORD inc)

Variables

- int alfatable1 [27]
- SUBBUF ∗ subexpbuffers = 0
- SUBBUF ∗ topsubexpbuffers = 0
- LONG insubexpbuffers = 0

4.11.1 Detailed Description

The heart of the compiler. It contains the tables of statements. It finds the statements in the tables and calls the proper routines. For algebraic expressions it runs the compilation by first calling the tokenizer, splitting things into subexpressions and generating the code. There is a system for recognizing already existing subexpressions. This economizes on the length of the output.

Note: the compiler of FORM doesn't attempt to normalize the input. Hence x+1 and 1+x are different objects during compilation. Similarly (a+b-b) will not be simplified to (a).

Definition in file compiler.c.

4.11.2 Macro Definition Documentation

4.11.2.1 OPTION0

```
#define OPTION0 1
```

Definition at line 229 of file compiler.c.

4.11.2.2 OPTION1

```
#define OPTION1 2
```

Definition at line 230 of file compiler.c.

4.11.2.3 OPTION2

```
#define OPTION2 3
```

Definition at line 231 of file compiler.c.

4.11.2.4 REDUCESUBEXPBUFFERS

```
#define REDUCESUBEXPBUFFERS
```

Value:

```
{ if ( (topsubexpbuffers-subexpbuffers) > 256 ) {\
M_free(subexpbuffers,"subexpbuffers");\
subexpbuffers = (SUBBUF *)Malloc1(256*sizeof(SUBBUF),"subexpbuffers");\
topsubexpbuffers = subexpbuffers+256; } insubexpbuffers = 0; }
```

Definition at line 242 of file compiler.c.

4.11.3 Function Documentation

4.11.3.1 inictable()

```
VOID inictable ( )
```

Definition at line 286 of file compiler.c.

4.11.3.2 findcommand()

```
KEYWORD * findcommand (
            UBYTE * in )
```

Definition at line 312 of file compiler.c.

4.11.3.3 ParenthesesTest()

```
int ParenthesesTest (
            UBYTE * sin )
```

Definition at line 356 of file compiler.c.

4.11.3.4 SkipAName()

```
UBYTE * SkipAName (
            UBYTE * s )
```

Definition at line 406 of file compiler.c.

4.11.3.5 IsRHS()

```
UBYTE * IsRHS (
            UBYTE * s,
            UBYTE c )
```

Definition at line 434 of file compiler.c.

4.11.3.6 IsIdStatement()

```
int IsIdStatement (
            UBYTE * s )
```

Definition at line 480 of file compiler.c.

4.11.3.7 CompileAlgebra()

```
int CompileAlgebra (
            UBYTE * s,
            int leftright,
            WORD * prototype )
```

Definition at line 494 of file compiler.c.

4.11.3.8 CompileStatement()

```
int CompileStatement (
            UBYTE * in )
```

Definition at line 530 of file compiler.c.

4.11.3.9 TestTables()

```
int TestTables ( )
```

Definition at line 644 of file compiler.c.

4.11.3.10 CompileSubExpressions()

```
int CompileSubExpressions (
            SBYTE * tokens )
```

Definition at line 688 of file compiler.c.

4.11.3.11 CodeGenerator()

```
int CodeGenerator (
            SBYTE * tokens )
```

Definition at line 823 of file compiler.c.

4.11.3.12 CompleteTerm()

```
int CompleteTerm (
            WORD * term,
            UWORD * numer,
            UWORD * denom,
            WORD nnum,
            WORD nden,
            int sign )
```

Definition at line 1855 of file compiler.c.

4.11.3.13 CodeFactors()

```
int CodeFactors (
            SBYTE * tokens )
```

Definition at line 1892 of file compiler.c.

4.11.3.14 GenerateFactors()

```
WORD GenerateFactors (
            WORD n,
            WORD inc )
```

Definition at line 2189 of file compiler.c.

4.11.4 Variable Documentation

4.11.4.1 alfatable1

```
int alfatable1[27]
```

Definition at line 227 of file compiler.c.

4.11.4.2 subexpbuffers

```
SUBBUF* subexpbuffers = 0
```

Definition at line 238 of file compiler.c.

4.11.4.3 topsubexpbuffers

```
SUBBUF* topsubexpbuffers = 0
```

Definition at line 239 of file compiler.c.

4.11.4.4 insubexpbuffers

```
LONG insubexpbuffers = 0
```

Definition at line 240 of file compiler.c.

4.12 compiler.c

Go to the documentation of this file.

```
00001
00015 /* #[ License : */
00016 /*
00017  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00018  *   When using this file you are requested to refer to the publication
00019  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00020  *   This is considered a matter of courtesy as the development was paid
00021  *   for by FOM the Dutch physics granting agency and we would like to
00022  *   be able to track its scientific use to convince FOM of its value
00023  *   for the community.
00024  *
00025  *   This file is part of FORM.
00026  *
00027  *   FORM is free software: you can redistribute it and/or modify it under the
00028  *   terms of the GNU General Public License as published by the Free Software
00029  *   Foundation, either version 3 of the License, or (at your option) any later
00030  *   version.
00031  *
00032  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00033  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00034  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00035  *   details.
00036  *
00037  *   You should have received a copy of the GNU General Public License along
00038  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00039  */
00040 /* #] License : */
00041 /*
00042     #[ includes :
00043 */
00044
```

```
#include "form3.h"

/*
    com1commands are the commands of which only part of the word has to
    be present. The order is rather important here.
    com2commands are the commands that must have their whole word match.
    here we can do a binary search.
    {[(
*/

static KEYWORD com1commands[] = {
     {"also",            (TFUN)CoIdOld,             STATEMENT,    PARTEST}
    ,{"abrackets",       (TFUN)CoAntiBracket,       TOOUTPUT,     PARTEST}
    ,{"antisymmetrize",  (TFUN)CoAntiSymmetrize,    STATEMENT,    PARTEST}
    ,{"antibrackets",    (TFUN)CoAntiBracket,       TOOUTPUT,     PARTEST}
    ,{"brackets",        (TFUN)CoBracket,           TOOUTPUT,     PARTEST}
    ,{"cfunctions",      (TFUN)CoCFunction,         DECLARATION,  PARTEST|WITHAUTO}
    ,{"commuting",       (TFUN)CoCFunction,         DECLARATION,  PARTEST|WITHAUTO}
    ,{"compress",        (TFUN)CoCompress,          DECLARATION,  PARTEST}
    ,{"ctensors",        (TFUN)CoCTensor,           DECLARATION,  PARTEST|WITHAUTO}
    ,{"cyclesymmetrize",(TFUN)CoCycleSymmetrize,   STATEMENT,    PARTEST}
    ,{"dimension",       (TFUN)CoDimension,         DECLARATION,  PARTEST}
    ,{"discard",         (TFUN)CoDiscard,           STATEMENT,    PARTEST}
    ,{"functions",       (TFUN)CoNFunction,         DECLARATION,  PARTEST|WITHAUTO}
    ,{"format",          (TFUN)CoFormat,            TOOUTPUT,     PARTEST}
    ,{"fixindex",        (TFUN)CoFixIndex,          DECLARATION,  PARTEST}
    ,{"global",          (TFUN)CoGlobal,            DEFINITION,   PARTEST}
    ,{"gfactorized",     (TFUN)CoGlobalFactorized, DEFINITION,   PARTEST}
    ,{"globalfactorized",(TFUN)CoGlobalFactorized,DEFINITION,   PARTEST}
    ,{"goto",            (TFUN)CoGoTo,              STATEMENT,    PARTEST}
    ,{"indexes",         (TFUN)CoIndex,             DECLARATION,  PARTEST|WITHAUTO}
    ,{"indices",         (TFUN)CoIndex,             DECLARATION,  PARTEST|WITHAUTO}
    ,{"identify",        (TFUN)CoId,                STATEMENT,    PARTEST}
    ,{"idnew",           (TFUN)CoIdNew,             STATEMENT,    PARTEST}
    ,{"idold",           (TFUN)CoIdOld,             STATEMENT,    PARTEST}
    ,{"local",           (TFUN)CoLocal,             DEFINITION,   PARTEST}
    ,{"lfactorized",     (TFUN)CoLocalFactorized,  DEFINITION,   PARTEST}
    ,{"localfactorized",(TFUN)CoLocalFactorized,  DEFINITION,   PARTEST}
    ,{"load",            (TFUN)CoLoad,              DECLARATION,  PARTEST}
    ,{"label",           (TFUN)CoLabel,             STATEMENT,    PARTEST}
    ,{"modulus",         (TFUN)CoModulus,           DECLARATION,  PARTEST}
    ,{"multiply",        (TFUN)CoMultiply,          STATEMENT,    PARTEST}
    ,{"nfunctions",      (TFUN)CoNFunction,         DECLARATION,  PARTEST|WITHAUTO}
    ,{"nprint",          (TFUN)CoNPrint,            TOOUTPUT,     PARTEST}
    ,{"ntensors",        (TFUN)CoNTensor,           DECLARATION,  PARTEST|WITHAUTO}
    ,{"nwrite",          (TFUN)CoNWrite,            DECLARATION,  PARTEST}
    ,{"print",           (TFUN)CoPrint,             MIXED,        0}
    ,{"redefine",        (TFUN)CoRedefine,          STATEMENT,    0}
    ,{"rcyclesymmetrize",(TFUN)CoRCycleSymmetrize,STATEMENT,    PARTEST}
    ,{"symbols",         (TFUN)CoSymbol,            DECLARATION,  PARTEST|WITHAUTO}
    ,{"save",            (TFUN)CoSave,              DECLARATION,  PARTEST}
    ,{"symmetrize",      (TFUN)CoSymmetrize,        STATEMENT,    PARTEST}
    ,{"tensors",         (TFUN)CoCTensor,           DECLARATION,  PARTEST|WITHAUTO}
    ,{"unittrace",       (TFUN)CoUnitTrace,         DECLARATION,  PARTEST}
    ,{"vectors",         (TFUN)CoVector,            DECLARATION,  PARTEST|WITHAUTO}
    ,{"write",           (TFUN)CoWrite,             DECLARATION,  PARTEST}
};

static KEYWORD com2commands[] = {
     {"antiputinside",   (TFUN)CoAntiPutInside,     STATEMENT,    PARTEST}
    ,{"apply",           (TFUN)CoApply,             STATEMENT,    PARTEST}
    ,{"aputinside",      (TFUN)CoAntiPutInside,     STATEMENT,    PARTEST}
    ,{"argexplode",      (TFUN)CoArgExplode,        STATEMENT,    PARTEST}
    ,{"argimplode",      (TFUN)CoArgImplode,        STATEMENT,    PARTEST}
    ,{"argtoextrasymbol",(TFUN)CoArgToExtraSymbol,STATEMENT,    PARTEST}
    ,{"argument",        (TFUN)CoArgument,          STATEMENT,    PARTEST}
    ,{"assign",          (TFUN)CoAssign,            STATEMENT,    PARTEST}
    ,{"auto",            (TFUN)CoAuto,              DECLARATION,  PARTEST}
    ,{"autodeclare",     (TFUN)CoAuto,              DECLARATION,  PARTEST}
    ,{"chainin",         (TFUN)CoChainin,           STATEMENT,    PARTEST}
    ,{"chainout",        (TFUN)CoChainout,          STATEMENT,    PARTEST}
    ,{"chisholm",        (TFUN)CoChisholm,          STATEMENT,    PARTEST}
    ,{"cleartable",      (TFUN)CoClearTable,        DECLARATION,  PARTEST}
    ,{"collect",         (TFUN)CoCollect,           SPECIFICATION,PARTEST}
    ,{"commuteinset",    (TFUN)CoCommuteInSet,      DECLARATION,  PARTEST}
    ,{"contract",        (TFUN)CoContract,          STATEMENT,    PARTEST}
    ,{"copyspectator"  ,(TFUN)CoCopySpectator,     DEFINITION,   PARTEST}
    ,{"createspectator",(TFUN)CoCreateSpectator,   DECLARATION,  PARTEST}
    ,{"ctable",          (TFUN)CoCTable,            DECLARATION,  PARTEST}
    ,{"deallocatetable",(TFUN)CoDeallocateTable,   DECLARATION,  PARTEST}
    ,{"delete",          (TFUN)CoDelete,            SPECIFICATION,PARTEST}
    ,{"denominators",    (TFUN)CoDenominators,      STATEMENT,    PARTEST}
    ,{"disorder",        (TFUN)CoDisorder,          STATEMENT,    PARTEST}
    ,{"do",              (TFUN)CoDo,                STATEMENT,    PARTEST}
    ,{"drop",            (TFUN)CoDrop,              SPECIFICATION,PARTEST}
    ,{"dropcoefficient",(TFUN)CoDropCoefficient,   STATEMENT,    PARTEST}
    ,{"dropsymbols",     (TFUN)CoDropSymbols,       STATEMENT,    PARTEST}
```

```
,{"else",            (TFUN)CoElse,             STATEMENT,    PARTEST}
,{"elseif",          (TFUN)CoElseIf,           STATEMENT,    PARTEST}
,{"emptyspectator", (TFUN)CoEmptySpectator,   SPECIFICATION,PARTEST}
,{"endargument",     (TFUN)CoEndArgument,      STATEMENT,    PARTEST}
,{"enddo",           (TFUN)CoEndDo,            STATEMENT,    PARTEST}
,{"endif",           (TFUN)CoEndIf,            STATEMENT,    PARTEST}
,{"endinexpression",(TFUN)CoEndInExpression,  STATEMENT,    PARTEST}
,{"endinside",       (TFUN)CoEndInside,        STATEMENT,    PARTEST}
,{"endrepeat",       (TFUN)CoEndRepeat,        STATEMENT,    PARTEST}
,{"endterm",         (TFUN)CoEndTerm,          STATEMENT,    PARTEST}
,{"endwhile",        (TFUN)CoEndWhile,         STATEMENT,    PARTEST}
,{"exit",            (TFUN)CoExit,             STATEMENT,    PARTEST}
,{"extrasymbols",    (TFUN)CoExtraSymbols,     DECLARATION,  PARTEST}
,{"factarg",         (TFUN)CoFactArg,          STATEMENT,    PARTEST}
,{"factdollar",      (TFUN)CoFactDollar,       STATEMENT,    PARTEST}
,{"factorize",       (TFUN)CoFactorize,        TOOUTPUT,     PARTEST}
,{"fill",            (TFUN)CoFill,             DECLARATION,  PARTEST}
,{"fillexpression", (TFUN)CoFillExpression,   DECLARATION,  PARTEST}
,{"frompolynomial", (TFUN)CoFromPolynomial,   STATEMENT,    PARTEST}
,{"funpowers",       (TFUN)CoFunPowers,        DECLARATION,  PARTEST}
,{"hide",            (TFUN)CoHide,             SPECIFICATION,PARTEST}
,{"if",              (TFUN)CoIf,               STATEMENT,    PARTEST}
,{"ifmatch",         (TFUN)CoIfMatch,          STATEMENT,    PARTEST}
,{"ifnomatch",       (TFUN)CoIfNoMatch,        STATEMENT,    PARTEST}
,{"ifnotmatch",      (TFUN)CoIfNoMatch,        STATEMENT,    PARTEST}
,{"inexpression",    (TFUN)CoInExpression,     STATEMENT,    PARTEST}
,{"inparallel",      (TFUN)CoInParallel,       SPECIFICATION,PARTEST}
,{"inside",          (TFUN)CoInside,           STATEMENT,    PARTEST}
,{"insidefirst",     (TFUN)CoInsideFirst,      DECLARATION,  PARTEST}
,{"intohide",        (TFUN)CoIntoHide,         SPECIFICATION,PARTEST}
,{"keep",            (TFUN)CoKeep,             SPECIFICATION,PARTEST}
,{"makeinteger",     (TFUN)CoMakeInteger,      STATEMENT,    PARTEST}
,{"many",            (TFUN)CoMany,             STATEMENT,    PARTEST}
,{"merge",           (TFUN)CoMerge,            STATEMENT,    PARTEST}
,{"metric",          (TFUN)CoMetric,           DECLARATION,  PARTEST}
,{"moduleoption",    (TFUN)CoModuleOption,     ATENDOFMODULE,PARTEST}
,{"multi",           (TFUN)CoMulti,            STATEMENT,    PARTEST}
,{"multibracket",    (TFUN)CoMultiBracket,     STATEMENT,    PARTEST}
,{"ndrop",           (TFUN)CoNoDrop,           SPECIFICATION,PARTEST}
,{"nfactorize",      (TFUN)CoNFactorize,       TOOUTPUT,     PARTEST}
,{"nhide",           (TFUN)CoNoHide,           SPECIFICATION,PARTEST}
,{"normalize",       (TFUN)CoNormalize,        STATEMENT,    PARTEST}
,{"notinparallel",   (TFUN)CoNotInParallel,    SPECIFICATION,PARTEST}
,{"nskip",           (TFUN)CoNoSkip,           SPECIFICATION,PARTEST}
,{"ntable",          (TFUN)CoNTable,           DECLARATION,  PARTEST}
,{"nunfactorize",    (TFUN)CoNUnFactorize,     TOOUTPUT,     PARTEST}
,{"nunhide",         (TFUN)CoNoUnHide,         SPECIFICATION,PARTEST}
,{"off",             (TFUN)CoOff,              DECLARATION,  PARTEST}
,{"on",              (TFUN)CoOn,               DECLARATION,  PARTEST}
,{"once",            (TFUN)CoOnce,             STATEMENT,    PARTEST}
,{"only",            (TFUN)CoOnly,             STATEMENT,    PARTEST}
,{"polyfun",         (TFUN)CoPolyFun,          DECLARATION,  PARTEST}
,{"polyratfun",      (TFUN)CoPolyRatFun,       DECLARATION,  PARTEST}
,{"pophide",         (TFUN)CoPopHide,          SPECIFICATION,PARTEST}
,{"print[]",         (TFUN)CoPrintB,           TOOUTPUT,     PARTEST}
,{"printtable",      (TFUN)CoPrintTable,       MIXED,        PARTEST}
,{"processbucketsize",(TFUN)CoProcessBucket,  DECLARATION,  PARTEST}
,{"propercount",     (TFUN)CoProperCount,      DECLARATION,  PARTEST}
,{"pushhide",        (TFUN)CoPushHide,         SPECIFICATION,PARTEST}
,{"putinside",       (TFUN)CoPutInside,        STATEMENT,    PARTEST}
,{"ratio",           (TFUN)CoRatio,            STATEMENT,    PARTEST}
,{"removespectator",(TFUN)CoRemoveSpectator,  SPECIFICATION,PARTEST}
,{"renumber",        (TFUN)CoRenumber,         STATEMENT,    PARTEST}
,{"repeat",          (TFUN)CoRepeat,           STATEMENT,    PARTEST}
,{"replaceloop",     (TFUN)CoReplaceLoop,      STATEMENT,    PARTEST}
,{"select",          (TFUN)CoSelect,           STATEMENT,    PARTEST}
,{"set",             (TFUN)CoSet,              DECLARATION,  PARTEST}
,{"setexitflag",     (TFUN)CoSetExitFlag,      STATEMENT,    PARTEST}
,{"shuffle",         (TFUN)CoMerge,            STATEMENT,    PARTEST}
,{"skip",            (TFUN)CoSkip,             SPECIFICATION,PARTEST}
,{"sort",            (TFUN)CoSort,             STATEMENT,    PARTEST}
,{"splitarg",        (TFUN)CoSplitArg,         STATEMENT,    PARTEST}
,{"splitfirstarg",   (TFUN)CoSplitFirstArg,    STATEMENT,    PARTEST}
,{"splitlastarg",    (TFUN)CoSplitLastArg,     STATEMENT,    PARTEST}
,{"stuffle",         (TFUN)CoStuffle,          STATEMENT,    PARTEST}
,{"sum",             (TFUN)CoSum,              STATEMENT,    PARTEST}
,{"table",           (TFUN)CoTable,            DECLARATION,  PARTEST}
,{"tablebase",       (TFUN)CoTableBase,        DECLARATION,  PARTEST}
,{"tb",              (TFUN)CoTableBase,        DECLARATION,  PARTEST}
,{"term",            (TFUN)CoTerm,             STATEMENT,    PARTEST}
,{"testuse",         (TFUN)CoTestUse,          STATEMENT,    PARTEST}
,{"threadbucketsize",(TFUN)CoThreadBucket,    DECLARATION,  PARTEST}
,{"topolynomial",    (TFUN)CoToPolynomial,     STATEMENT,    PARTEST}
,{"tospectator",     (TFUN)CoToSpectator,      STATEMENT,    PARTEST}
,{"totensor",        (TFUN)CoToTensor,         STATEMENT,    PARTEST}
,{"tovector",        (TFUN)CoToVector,         STATEMENT,    PARTEST}
,{"trace4",          (TFUN)CoTrace4,           STATEMENT,    PARTEST}
```

```
    ,{"tracen",          (TFUN)CoTraceN,           STATEMENT,    PARTEST}
    ,{"transform",       (TFUN)CoTransform,        STATEMENT,    PARTEST}
    ,{"tryreplace",      (TFUN)CoTryReplace,       STATEMENT,    PARTEST}
    ,{"unfactorize",     (TFUN)CoUnFactorize,      TOOUTPUT,     PARTEST}
    ,{"unhide",          (TFUN)CoUnHide,           SPECIFICATION,PARTEST}
    ,{"while",           (TFUN)CoWhile,            STATEMENT,    PARTEST}
};

int alfatable1[27];

#define OPTION0 1
#define OPTION1 2
#define OPTION2 3

typedef struct SuBbUf {
    WORD    subexpnum;
    WORD    buffernum;
} SUBBUF;

SUBBUF *subexpbuffers = 0;
SUBBUF *topsubexpbuffers = 0;
LONG insubexpbuffers = 0;

#define REDUCESUBEXPBUFFERS { if ( (topsubexpbuffers-subexpbuffers) > 256 ) {\
    M_free(subexpbuffers,"subexpbuffers");\
    subexpbuffers = (SUBBUF *)Malloc1(256*sizeof(SUBBUF),"subexpbuffers");\
    topsubexpbuffers = subexpbuffers+256; } insubexpbuffers = 0; }

#if defined(ILP32)

#define PUTNUMBER128(t,n) { if ( n >= 16384 ) { \
                *t++ = n/(128*128); *t++ = (n/128)%128; *t++ = n%128; } \
            else if ( n >= 128 ) { *t++ = n/128; *t++ = n%128; }      \
            else *t++ = n; }
#define PUTNUMBER100(t,n) { if ( n >= 10000 ) { \
                *t++ = n/10000; *t++ = (n/100)%100; *t++ = n%100; } \
            else if ( n >= 100 ) { *t++ = n/100; *t++ = n%100; }   \
            else *t++ = n; }

#elif ( defined(LLP64) || defined(LP64) )

#define PUTNUMBER128(t,n) { if ( n >= 2097152 ) { \
                *t++ = ((n/128)/128)/128; *t++ = ((n/128)/128)%128; *t++ = (n/128)%128; *t++ = n%128;
    } \
            else if ( n >= 16384 ) { \
                *t++ = n/(128*128); *t++ = (n/128)%128; *t++ = n%128; } \
            else if ( n >= 128 ) { *t++ = n/128; *t++ = n%128; }      \
            else *t++ = n; }
#define PUTNUMBER100(t,n) { if ( n >= 1000000 ) { \
                *t++ = ((n/100)/100)/100; *t++ = ((n/100)/100)%100; *t++ = (n/100)%100; *t++ = n%100;
    } \
            else if ( n >= 10000 ) { \
                *t++ = n/10000; *t++ = (n/100)%100; *t++ = n%100; } \
            else if ( n >= 100 ) { *t++ = n/100; *t++ = n%100; }   \
            else *t++ = n; }

#endif

/*
    )]}
    #] includes :
    #[ Compiler :
        #[ inictable :

        Routine sets the table for 1-st characters that allow a faster
        start in the search in table 1 which should be sequential.
        Search in table 2 can be binary.
*/

VOID inictable()
{
    KEYWORD *k = com1commands;
    int i, j, ksize;
    ksize = sizeof(com1commands)/sizeof(KEYWORD);
    j = 0;
    alfatable1[0] = 0;
    for ( i = 0; i < 26; i++ ) {
        while ( j < ksize && k[j].name[0] == 'a'+i ) j++;
        alfatable1[i+1] = j;
    }
}

/*
        #] inictable :
        #[ findcommand :

        Checks whether a command is in the command table.
```

```
        If so a pointer to the table element is returned.
        If not we return 0.
        Note that when a command is not in the table, we have
        to test whether it is an id command without id. It should
        then have the structure pattern = rhs. This should be done
        in the calling routine.
*/

KEYWORD *findcommand(UBYTE *in)
{
    int hi, med, lo, i;
    UBYTE *s, c;
    s = in;
    while ( FG.cTable[*s] <= 1 ) s++;
    if ( s > in && *s == '[' && s[1] == ']' ) s += 2;
    if ( *s ) { c = *s; *s = 0; }
    else c = 0;
/*
    First do a binary search in the second table
*/
    lo = 0;
    hi = sizeof(com2commands)/sizeof(KEYWORD)-1;
    do {
        med = ( hi + lo ) / 2;
        i = StrICmp(in,(UBYTE *)com2commands[med].name);
        if ( i == 0 ) { if ( c ) *s = c; return(com2commands+med); }
        if ( i < 0 ) hi = med-1;
        else         lo = med+1;
    } while ( hi >= lo );
/*
    Now do a 'hashed' search in the first table. It is sequential.
*/
    i = tolower(*in) - 'a';
    med = alfatable1[i];
    hi  = alfatable1[i+1];
    while ( med < hi ) {
        if ( StrICont(in,(UBYTE *)com1commands[med].name) == 0 )
            { if ( c ) *s = c; return(com1commands+med); }
        med++;
    }
    if ( c ) *s = c;
/*
    Unrecognized. Too bad!
*/
    return(0);
}

/*
        #] findcommand :
        #[ ParenthesesTest :
*/

int ParenthesesTest(UBYTE *sin)
{
    WORD L1 = 0, L2 = 0, L3 = 0;
    UBYTE *s = sin;
    while ( *s ) {
        if ( *s == '[' ) L1++;
        else if ( *s == ']' ) {
            L1--;
            if ( L1 < 0 ) { MesPrint("&Unmatched []"); return(1); }
        }
        s++;
    }
    if ( L1 > 0 ) { MesPrint("&Unmatched []"); return(1); }
    s = sin;
    while ( *s ) {
        if ( *s == '[' ) SKIPBRA1(s)
        else if ( *s == '(' ) { L2++; s++; }
        else if ( *s == ')' ) {
            L2--; s++;
            if ( L2 < 0 ) { MesPrint("&Unmatched ()"); return(1); }
        }
        else s++;
    }
    if ( L2 > 0 ) { MesPrint("&Unmatched ()"); return(1); }
    s = sin;
    while ( *s ) {
        if ( *s == '[' ) SKIPBRA1(s)
        else if ( *s == '[' ) SKIPBRA4(s)
        else if ( *s == '{' ) { L3++; s++; }
        else if ( *s == '}' ) {
            L3--; s++;
            if ( L3 < 0 ) { MesPrint("&Unmatched {}"); return(1); }
        }
        else s++;
    }
```

```
    if ( L3 > 0 ) { MesPrint("&Unmatched {}"); return(1); }
    return(0);
}

/*
        #] ParenthesesTest :
        #[ SkipAName :

        Skips a name and gives a pointer to the object after the name.
        If there is not a proper name, it returns a zero pointer.
        In principle the brackets match already, so the `if ( *s == 0 )'
        code is not really needed, but you never know how the program
        is extended later.
*/

UBYTE *SkipAName(UBYTE *s)
{
    UBYTE *t = s;
    if ( *s == '[' ) {
        SKIPBRA1(s)
        if ( *s == 0 ) {
            MesPrint("&Illegal name: '%s'",t);
            return(0);
        }
        s++;
    }
    else if ( FG.cTable[*s] == 0 || *s == '_' || *s == '$' ) {
        if ( *s == '$' ) s++;
        while ( FG.cTable[*s] <= 1 ) s++;
        if ( *s == '_' ) s++;
    }
    else {
        MesPrint("&Illegal name: '%s'",t);
        return(0);
    }
    return(s);
}

/*
        #] SkipAName :
        #[ IsRHS :
*/

UBYTE *IsRHS(UBYTE *s, UBYTE c)
{
    while ( *s && *s != c ) {
        if ( *s == '[' ) {
            SKIPBRA1(s);
            if ( *s != ']' ) {
                MesPrint("&Unmatched []");
                return(0);
            }
        }
        else if ( *s == '{' ) {
            SKIPBRA2(s);
            if ( *s != '}' ) {
                MesPrint("&Unmatched {}");
                return(0);
            }
        }
        else if ( *s == '(' ) {
            SKIPBRA3(s);
            if ( *s != ')' ) {
                MesPrint("&Unmatched ()");
                return(0);
            }
        }
        else if ( *s == ')' ) {
            MesPrint("&Unmatched ()");
            return(0);
        }
        else if ( *s == '}' ) {
            MesPrint("&Unmatched {}");
            return(0);
        }
        else if ( *s == ']' ) {
            MesPrint("&Unmatched []");
            return(0);
        }
        s++;
    }
    return(s);
}

/*
        #] IsRHS :
        #[ IsIdStatement :
```

```
*/

int IsIdStatement(UBYTE *s)
{
    DUMMYUSE(s);
    return(0);
}

/*
        #] IsIdStatement :
        #[ CompileAlgebra :

        Returns either the number of the main level RHS (>= 0)
        or an error code (< 0)
*/

int CompileAlgebra(UBYTE *s, int leftright, WORD *prototype)
{
    GETIDENTITY
    int error;
    WORD *oldproto = AC.ProtoType;
    AC.ProtoType = prototype;
    if ( AC.TokensWriteFlag ) {
        MesPrint("To tokenize: %s",s);
        error = tokenize(s,leftright);
        MesPrint("  The contents of the token buffer are:");
        WriteTokens(AC.tokens);
    }
    else error = tokenize(s,leftright);
    if ( error == 0 ) {
        AR.Eside = leftright;
        AC.CompileLevel = 0;
        if ( leftright == LHSIDE ) { AC.DumNum = AR.CurDum = 0; }
        error = CompileSubExpressions(AC.tokens);
        REDUCESUBEXPBUFFERS
    }
    else {
        AC.ProtoType = oldproto;
        return(-1);
    }
    AC.ProtoType = oldproto;
    if ( error < 0 ) return(-1);
    else if ( leftright == LHSIDE ) return(cbuf[AC.cbufnum].numlhs);
    else                            return(cbuf[AC.cbufnum].numrhs);
}

/*
        #] CompileAlgebra :
        #[ CompileStatement :

*/

int CompileStatement(UBYTE *in)
{
    KEYWORD *k;
    UBYTE *s;
    int error1 = 0, error2;
    /* A.iStatement = */ s = in;
    if ( *s == 0 ) return(0);
    if ( *s == '$' ) {
        k = findcommand((UBYTE *)"assign");
    }
    else {
        if ( ( k = findcommand(s) ) == 0 && IsIdStatement(s) == 0 ) {
            MesPrint("&Unrecognized statement");
            return(1);
        }
        if ( k == 0 ) { /* Id statement without id. Note: id must be in table */
            k = com1commands + alfatable1['i'-'a'];
            while ( k->name[1] != 'd' || k->name[2] ) k++;
        }
        else {
            while ( FG.cTable[*s] <= 1 ) s++;
            if ( s > in && *s == '[' && s[1] == ']' ) s += 2;
/*
            The next statement is rather mysterious
            It is undone in DoPrint and CoMultiply, but it also causes effects
            in other (wrong) statements like dimension -4; or Trace4 -1;
            The code in pre.c (LoadStatement) has been changed 8-sep-2009
            to force a comma after the keyword. This means that the
            'mysterious' line is automatically inactive. Hence it is taken out.

            if ( *s == '+' || *s == '-' ) s++;
*/
            if ( *s == ',' ) s++;
        }
    }
```

```
/*
    First the test on the order of the statements.
    This is relatively new (2.2c) and may cause some problems with old
    programs. Hence the first error message should explain!
*/
    if ( AP.PreAssignFlag == 0 && AM.OldOrderFlag == 0 ) {
     if ( AP.PreInsideLevel ) {
      if ( k->type != STATEMENT && k->type != MIXED ) {
        MesPrint("&Only executable and print statements are allowed in an %#inside/%#endinside construction");
        return(-1);
      }
     }
     else {
      if ( ( AC.compiletype == DECLARATION || AC.compiletype == SPECIFICATION )
      && ( k->type == STATEMENT || k->type == DEFINITION || k->type == TOOUTPUT ) ) {
        if ( AC.tablecheck == 0 ) {
            AC.tablecheck = 1;
            if ( TestTables() ) error1 = 1;
        }
      }
      if ( k->type == MIXED ) {
        if ( AC.compiletype <= DEFINITION ) {
            AC.compiletype = STATEMENT;
        }
      }
      else if ( k->type > AC.compiletype ) {
        if ( StrCmp((UBYTE *)(k->name),(UBYTE *)"format") != 0 )
            AC.compiletype = k->type;
      }
      else if ( k->type < AC.compiletype ) {
        switch ( k->type ) {
            case DECLARATION:
                MesPrint("&Declaration out of order");
                MesPrint("&  %s",in);
                break;
            case DEFINITION:
                MesPrint("&Definition out of order");
                MesPrint("&  %s",in);
                break;
            case SPECIFICATION:
                MesPrint("&Specification out of order");
                MesPrint("&  %s",in);
                break;
            case STATEMENT:
                MesPrint("&Statement out of order");
                break;
            case TOOUTPUT:
                MesPrint("&Output control statement out of order");
                MesPrint("&  %s",in);
                break;
        }
        AC.compiletype = k->type;
        if ( AC.firstctypemessage == 0 ) {
            MesPrint("&Proper order inside a module is:");
            MesPrint("Declarations, specifications, definitions, statements, output control statements");
            AC.firstctypemessage = 1;
        }
        error1 = 1;
      }
     }
    }
/*
    Now we execute the tests that are prescribed by the flags.
*/
    if ( AC.AutoDeclareFlag && ( ( k->flags & WITHAUTO ) == 0 ) ) {
        MesPrint("&Illegal type of auto-declaration");
        return(1);
    }
    if ( ( ( k->flags & PARTEST ) != 0 ) && ParenthesesTest(s) ) return(1);
    error2 = (*k->func)(s);
    if ( error2 == 0 ) return(error1);
    return(error2);
}

/*
        #] CompileStatement :
        #[ TestTables :
*/

int TestTables()
{
    FUNCTIONS f = functions;
    TABLES t;
    WORD j;
    int error = 0, i;
```

```
    LONG x;
    i = NumFunctions + FUNCTION - MAXBUILTINFUNCTION - 1;
    f = f + MAXBUILTINFUNCTION - FUNCTION + 1;
    if ( AC.MustTestTable > 0 ) {
      while ( i > 0 ) {
        if ( ( t = f->tabl ) != 0 && t->strict > 0 && !t->sparse ) {
            for ( x = 0, j = 0; x < t->totind; x++ ) {
                if ( t->tablepointers[TABLEEXTENSION*x] < 0 ) j++;
            }
            if ( j > 0 ) {
                if ( j > 1 ) {
                    MesPrint("&In table %s there are %d unfilled elements",
                    AC.varnames->namebuffer+f->name,j);
                }
                else {
                    MesPrint("&In table %s there is one unfilled element",
                    AC.varnames->namebuffer+f->name);
                }
                error = 1;
            }
        }
        i--; f++;
      }
      AC.MustTestTable--;
    }
    return(error);
}

/*
        #] TestTables :
        #[ CompileSubExpressions :

        Now we attack the subexpressions from inside out.
        We try to see whether we had any of them already.
        We have to worry about adding the wildcard sum parameter
        to the prototype.
*/

int CompileSubExpressions(SBYTE *tokens)
{
    GETIDENTITY
    SBYTE *fill = tokens, *s = tokens, *t;
    WORD number[MAXNUMSIZE], *oldwork, *w1, *w2;
    int level, num, i, sumlevel = 0, sumtype = SYMTOSYM;
    int retval, error = 0;
/*
    Eliminate all subexpressions. They are marked by LPARENTHESIS,RPARENTHESIS
*/
    AC.CompileLevel++;
    while ( *s != TENDOFIT ) {
        if ( *s == TFUNOPEN ) {
            if ( fill < s ) *fill = TENDOFIT;
            t = fill - 1;
            while ( t >= tokens && t[0] >= 0 ) t--;
            if ( t >= tokens && *t == TFUNCTION ) {
                t++; i = 0; while ( *t >= 0 ) i = 128*i + *t++;
                if ( i == AM.sumnum || i == AM.sumpnum ) {
                    t = s + 1;
                    if ( *t == TSYMBOL || *t == TINDEX ) {
                        t++; i = 0; while ( *t >= 0 ) i = 128*i + *t++;
                        if ( s[1] == TINDEX ) {
                            i += AM.OffsetIndex;
                            sumtype = INDTOIND;
                        }
                        else sumtype = SYMTOSYM;
                        sumlevel = i;
                    }
                }
            }
            *fill++ = *s++;
        }
        else if ( *s == TFUNCLOSE ) { sumlevel = 0; *fill++ = *s++; }
        else if ( *s == LPARENTHESIS ) {
/*
            We must make an exception here.
            If the subexpression is just an integer, whatever its length,
            we should try to keep it.
            This is important when we have a function with an integer
            argument. In particular this is relevant for the MZV program.
*/
            t = s; level = 0;
            while ( level >= 0 ) {
                s++;
                if ( *s == LPARENTHESIS ) level++;
                else if ( *s == RPARENTHESIS ) level--;
                else if ( *s == TENDOFIT ) {
                    MesPrint("&Unbalanced subexpression parentheses");
```

```
                    return(-1);
                }
            }
            t++; *s = TENDOFIT;
            if ( sumlevel > 0 ) { /* Inside sum. Add wildcard to prototype */
                oldwork = w1 = AT.WorkPointer;
                w2 = AC.ProtoType;
                i = w2[1];
                while ( --i >= 0 ) *w1++ = *w2++;
                oldwork[1] += 4;
                *w1++ = sumtype; *w1++ = 4; *w1++ = sumlevel; *w1++ = sumlevel;
                w2 = AC.ProtoType; AT.WorkPointer = w1;
                AC.ProtoType = oldwork;
                num = CompileSubExpressions(t);
                AC.ProtoType = w2; AT.WorkPointer = oldwork;
            }
            else num = CompileSubExpressions(t);
            if ( num < 0 ) return(-1);
/*
            Note that the subexpression code should always fit.
            We had two parentheses and at least two bytes contents.
            There cannot be more than 2^21 subexpressions or we get outside
            this minimum. Ignoring this might lead to really rare and
            hard to find errors, years from now.
*/
            if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
                MesPrint("&More than %d subexpressions inside one
    expression",(WORD)MAXSUBEXPRESSIONS);
                Terminate(-1);
            }
            if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
                DoubleBuffer((void **)((VOID *)(&subexpbuffers))
                ,(void **)((VOID *)(&topsubexpbuffers)),sizeof(SUBBUF),"subexpbuffers");
            }
            subexpbuffers[insubexpbuffers].subexpnum = num;
            subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
            num = insubexpbuffers++;
            *fill++ = TSUBEXP;
            i = 0;
            do { number[i++] = num & 0x7F; num >>= 7; } while ( num );
            while ( --i >= 0 ) *fill++ = (SBYTE)(number[i]);
            s++;
        }
        else if ( *s == TEMPTY ) s++;
        else *fill++ = *s++;
    }
    *fill = TENDOFIT;
/*
    At this stage there are no more subexpressions.
    Hence we can do the basic compilation.
*/
    if ( AC.CompileLevel == 1 && AC.ToBeInFactors ) {
        error = CodeFactors(tokens);
    }
    AC.CompileLevel--;
    retval = CodeGenerator(tokens);
    if ( error < 0 ) return(error);
    return(retval);
}

/*
        #] CompileSubExpressions :
        #[ CodeGenerator :

        This routine does the real code generation.
        It returns the number of the rhs subexpression.
        At this point we do not have to worry about subexpressions,
        sets, setelements, simple vs complicated function arguments
        simple vs complicated powers etc.

        The variable 'first' indicates whether we are starting a new term

        The major complication are the set elements of type set[n].
        We have marked them as TSETNUM,n,Ttype,setnum
        They go into
        SETSET,size,subterm,relocation list
        in which the subterm should be ready to become a regular
        subterm in which the sets have been replaced by their element
        The relocation list consists of pairs of numbers:
        1: offset in the subterm, 2: the symbol n.
        Note that such a subterm can be a whole function with its arguments.
        We use the variable inset to indicate that we have something going.
        The relocation list is collected in the top of the WorkSpace.
*/

static UWORD *CGscrat7 = 0;

```

```
int CodeGenerator(SBYTE *tokens)
{
    GETIDENTITY
    SBYTE *s = tokens, c;
    int i, sign = 1, first = 1, deno = 1, error = 0, minus, n, needarg, numexp, cc;
    int base, sumlevel = 0, sumtype = SYMTOSYM, firstsumarg, inset = 0;
    int funflag = 0, settype, x1, x2, mulflag = 0;
    WORD *t, *v, *r, *term, nnumerator, ndenominator, *oldwork, x3, y, nin;
    WORD *w1, *w2, *tsize = 0, *relo = 0;
    UWORD *numerator, *denominator, *innum;
    CBUF *C;
    POSITION position;
    WORD TMproto[SUBEXPSIZE];
/*
#ifdef WITHPTHREADS
    RENUMBER renumber;
#endif
*/
    RENUMBER renumber;
    if ( AC.TokensWriteFlag ) WriteTokens(tokens);
    if ( CGscrat7 == 0 )
        CGscrat7 = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(WORD),"CodeGenerator");
    AddRHS(AC.cbufnum,0);
    C = cbuf + AC.cbufnum;
    numexp = C->numrhs;
    C->NumTerms[numexp] = 0;
    C->numdum[numexp] = 0;
    oldwork = AT.WorkPointer;
    numerator = (UWORD *)(AT.WorkPointer);
    denominator = numerator + 2*AM.MaxTal;
    innum = denominator + 2*AM.MaxTal;
    term = (WORD *)(innum + 2*AM.MaxTal);
    AT.WorkPointer = term + AM.MaxTer/sizeof(WORD);
    if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
    cc = 0;
    t = term+1;
    numerator[0] = denominator[0] = 1;
    nnumerator = ndenominator = 1;
    while ( *s != TENDOFIT ) {
        if ( *s == TPLUS || *s == TMINUS ) {
            if ( first || mulflag ) { if ( *s == TMINUS ) sign = -sign; }
            else {
                *term = t-term;
                C->NumTerms[numexp]++;
                if ( cc && sign ) C->CanCommu[numexp]++;
                CompleteTerm(term,numerator,denominator,nnumerator,ndenominator,sign);
                first = 1; cc = 0; t = term + 1; deno = 1;
                numerator[0] = denominator[0] = 1;
                nnumerator = ndenominator = 1;
                if ( *s == TMINUS ) sign = -1;
                else sign = 1;
            }
            s++;
        }
        else {
            mulflag = first = 0; c = *s++;
            switch ( c ) {
            case TSYMBOL:
                x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
                if ( *s == TWILDCARD ) { s++; x1 += 2*MAXPOWER; }
                *t++ = SYMBOL; *t++ = 4; *t++ = x1;
                if ( inset ) *relo = 2;
TryPower:       if ( *s == TPOWER ) {
                    s++;
                    if ( *s == TMINUS ) { s++; deno = -deno; }
                    c = *s++;
                    base = ( c == TNUMBER ) ? 100: 128;
                    x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
                    if ( c == TSYMBOL ) {
                        if ( *s == TWILDCARD ) s++;
                        x2 += 2*MAXPOWER;
                    }
                    *t++ = deno*x2;
                }
                else *t++ = deno;
fin:            deno = 1;
                if ( inset ) {
                    while ( relo < AT.WorkTop ) *t++ = *relo++;
                    inset = 0; tsize[1] = t - tsize;
                }
                break;
            case TINDEX:
                x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
                *t++ = INDEX; *t++ = 3;
                if ( *s == TWILDCARD ) { s++; x1 += WILDOFFSET; }
                if ( inset ) { *t++ = x1; *relo = 2; }
                else           *t++ = x1 + AM.OffsetIndex;
```

```
                if ( t[-1] > AM.IndDum ) {
                    x1 = t[-1] - AM.IndDum;
                    if ( x1 > C->numdum[numexp] ) C->numdum[numexp] = x1;
                }
                goto fin;
            case TGENINDEX:
                *t++ = INDEX; *t++ = 3; *t++ = AC.DumNum+WILDOFFSET;
                deno = 1;
                break;
            case TVECTOR:
                x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
dovector:       if ( inset == 0 ) x1 += AM.OffsetVector;
                if ( *s == TWILDCARD ) { s++; x1 += WILDOFFSET; }
                if ( inset ) *relo = 2;
                if ( *s == TDOT ) {     /* DotProduct ? */
                    s++;
                    if ( *s == TSETNUM || *s == TSETDOL ) {
                        settype = ( *s == TSETDOL );
                        s++; x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
                        if ( settype ) x2 = -x2;
                        if ( inset == 0 ) {
                            tsize = t; *t++ = SETSET; *t++ = 0;
                            relo = AT.WorkTop;
                        }
                        inset += 2;
                        *--relo = x2; *--relo = 3;
                    }
                    if ( *s != TVECTOR && *s != TDUBIOUS ) {
                        MesPrint("&Illegally formed dotproduct");
                        error = 1;
                    }
                    s++; x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
                    if ( inset < 2 ) x2 += AM.OffsetVector;
                    if ( *s == TWILDCARD ) { s++; x2 += WILDOFFSET; }
                    *t++ = DOTPRODUCT; *t++ = 5; *t++ = x1; *t++ = x2;
                    goto TryPower;
                }
                else if ( *s == TFUNOPEN ) {
                    s++;
                    if ( *s == TSETNUM || *s == TSETDOL ) {
                        settype = ( *s == TSETDOL );
                        s++; x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
                        if ( settype ) x2 = -x2;
                        if ( inset == 0 ) {
                            tsize = t; *t++ = SETSET; *t++ = 0;
                            relo = AT.WorkTop;
                        }
                        inset += 2;
                        *--relo = x2; *--relo = 3;
                    }
                    if ( *s == TINDEX || *s == TDUBIOUS ) {
                        s++;
                        x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
                        if ( inset < 2 ) x2 += AM.OffsetIndex;
                        if ( *s == TWILDCARD ) { s++; x2 += WILDOFFSET; }
                        *t++ = VECTOR; *t++ = 4; *t++ = x1; *t++ = x2;
                        if ( t[-1] > AM.IndDum ) {
                            x2 = t[-1] - AM.IndDum;
                            if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
                        }
                    }
                    else if ( *s == TGENINDEX ) {
                        *t++ = VECTOR; *t++ = 4; *t++ = x1;
                        *t++ = AC.DumNum + WILDOFFSET;
                    }
                    else if ( *s == TNUMBER || *s == TNUMBER1 ) {
                        base = ( *s == TNUMBER ) ? 100: 128;
                        s++;
                        x2 = 0; while ( *s >= 0 ) { x2 = x2*base + *s++; }
                        if ( x2 >= AM.OffsetIndex && inset < 2 ) {
                            MesPrint("&Fixed index in vector greater than %d",
                            AM.OffsetIndex);
                            return(-1);
                        }
                        *t++ = VECTOR; *t++ = 4; *t++ = x1; *t++ = x2;
                    }
                    else if ( *s == TVECTOR || ( *s == TMINUS && s[1] == TVECTOR ) ) {
                        if ( *s == TMINUS ) { s++; sign = -sign; }
                        s++;
                        x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
                        if ( inset < 2 ) x2 += AM.OffsetVector;
                        if ( *s == TWILDCARD ) { s++; x2 += WILDOFFSET; }
                        *t++ = DOTPRODUCT; *t++ = 5; *t++ = x1; *t++ = x2; *t++ = deno;
                    }
                    else {
                        MesPrint("&Illegal argument for vector");
                        return(-1);
```

```
                    }
                    if ( *s != TFUNCLOSE ) {
                        MesPrint("&Illegal argument for vector");
                        return(-1);
                    }
                    s++;
                }
                else {
                    if ( AC.DumNum ) {
                        *t++ = VECTOR; *t++ = 4; *t++ = x1;
                        *t++ = AC.DumNum + WILDOFFSET;
                    }
                    else {
                        *t++ = INDEX; *t++ = 3; *t++ = x1;
                    }
                }
                goto fin;
            case TDELTA:
                if ( *s != TFUNOPEN ) {
                    MesPrint("&d_ needs two arguments");
                    error = -1;
                }
                v = t; *t++ = DELTA; *t++ = 4;
                needarg = 2; x3 = x1 = -1;
                goto dotensor;
            case TFUNCTION:
                x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
                if ( x1 == AM.sumnum || x1 == AM.sumpnum ) sumlevel = x1;
                x1 += FUNCTION;
                if ( x1 == FIRSTBRACKET ) {
                    if ( s[0] == TFUNOPEN && s[1] == TEXPRESSION ) {
doexpr:                 s += 2;
                        *t++ = x1; *t++ = FUNHEAD+2; *t++ = 0;
                        if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
                                t[-1] |= MUSTCLEANPRF;
                        FILLFUN3(t)
                        x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
                        *t++ = -EXPRESSION; *t++ = x2;
/*
                        The next code is added to facilitate parallel processing
                        We need to call GetTable here to make sure all processors
                        have the same numbering of all variables.
*/
                        if ( Expressions[x2].status == STOREDEXPRESSION ) {
                            TMproto[0] = EXPRESSION;
                            TMproto[1] = SUBEXPSIZE;
                            TMproto[2] = x2;
                            TMproto[3] = 1;
                            { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
                            AT.TMaddr = TMproto;
                            PUTZERO(position);
/*
                            if ( (
#ifdef WITHPTHREADS
                                    renumber =
#endif
                                    GetTable(x2,&position,0) ) == 0 ) {
                                error = 1;
                                MesPrint("&Problems getting information about stored expression %s(1)"
                                ,EXPRNAME(x2));
                            }
#ifdef WITHPTHREADS
                            M_free(renumber->symb.lo,"VarSpace");
                            M_free(renumber,"Renumber");
#endif
*/
                            if ( ( renumber = GetTable(x2,&position,0) ) == 0 ) {
                                error = 1;
                                MesPrint("&Problems getting information about stored expression %s(1)"
                                ,EXPRNAME(x2));
                            }
                            if ( renumber->symb.lo != AN.dummyrenumlist )
                                M_free(renumber->symb.lo,"VarSpace");
                            M_free(renumber,"Renumber");
                            AR.StoreData.dirtyflag = 1;
                        }
                        if ( *s != TFUNCLOSE ) {
                            if ( x1 == FIRSTBRACKET )
                                MesPrint("&Problems with argument of FirstBracket_");
                            else if ( x1 == FIRSTTERM )
                                MesPrint("&Problems with argument of FirstTerm_");
                            else if ( x1 == CONTENTTERM )
                                MesPrint("&Problems with argument of FirstTerm_");
                            else if ( x1 == TERMSINEXPR )
                                MesPrint("&Problems with argument of TermsIn_");
                            else if ( x1 == NUMFACTORS )
                                MesPrint("&Problems with argument of NumFactors_");
```

```
                            else
                                MesPrint("&Problems with argument of FactorIn_");
                            error = 1;
                            while ( *s != TENDOFIT && *s != TFUNCLOSE ) s++;
                        }
                        if ( *s == TFUNCLOSE ) s++;
                        goto fin;
                    }
                }
                else if ( x1 == TERMSINEXPR || x1 == FACTORIN
                 || x1 == NUMFACTORS || x1 == FIRSTTERM || x1 == CONTENTTERM ) {
                    if ( s[0] == TFUNOPEN && s[1] == TEXPRESSION ) goto doexpr;
                    if ( s[0] == TFUNOPEN && s[1] == TDOLLAR ) {
                        s += 2;
                        *t++ = x1; *t++ = FUNHEAD+2; *t++ = 0;
                        FILLFUN3(t)
                        x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
                        *t++ = -DOLLAREXPRESSION; *t++ = x2;
                        if ( *s != TFUNCLOSE ) {
                            if ( x1 == TERMSINEXPR )
                                MesPrint("&Problems with argument of TermsIn_");
                            else if ( x1 == NUMFACTORS )
                                MesPrint("&Problems with argument of NumFactors_");
                            else
                                MesPrint("&Problems with argument of FactorIn_");
                            error = 1;
                            while ( *s != TENDOFIT && *s != TFUNCLOSE ) s++;
                        }
                        if ( *s == TFUNCLOSE ) s++;
                        goto fin;
                    }
                }
                x3 = x1;
                if ( inset && ( t-tsize == 2 ) ) x1 -= FUNCTION;
                if ( *s == TWILDCARD ) { x1 += WILDOFFSET; s++; }
                if ( functions[x3-FUNCTION].commute ) cc = 1;
                if ( *s != TFUNOPEN ) {
                    *t++ = x1; *t++ = FUNHEAD; *t++ = 0;
                    if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
                            t[-1] |= MUSTCLEANPRF;
                    FILLFUN3(t) sumlevel = 0; goto fin;
                }
                v = t; *t++ = x1; *t++ = FUNHEAD; *t++ = DIRTYFLAG;
                if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
                        t[-1] |= MUSTCLEANPRF;
                FILLFUN3(t)
                needarg = -1;
                if ( !inset && functions[x3-FUNCTION].spec >= TENSORFUNCTION ) {
dotensor:
                    do {
                        if ( needarg == 0 ) {
                            if ( x1 >= 0 ) {
                                x3 = x1;
                                if ( x3 >= FUNCTION+WILDOFFSET ) x3 -= WILDOFFSET;
                                MesPrint("&Too many arguments in function %s",
                                    VARNAME(functions,(x3-FUNCTION)) );
                            }
                            else
                                MesPrint("&d_ needs exactly two arguments");
                            error = -1;
                            needarg--;
                        }
                        else if ( needarg > 0 ) needarg--;
                        s++;
                        c = *s++;
                        if ( c == TMINUS && *s == TVECTOR ) { sign = -sign; c = *s++; }
                        base = ( c == TNUMBER ) ? 100: 128;
                        x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
                        if ( *s == TWILDCARD && c != TNUMBER ) { x2 += WILDOFFSET; s++; }
                        if ( c == TSETNUM || c == TSETDOL ) {
                            if ( c == TSETDOL ) x2 = -x2;
                            if ( inset == 0 ) {
                                w1 = t; t += 2; w2 = t;
                                while ( w1 > v ) *--w2 = *--w1;
                                tsize = v; relo = AT.WorkTop;
                                *v++ = SETSET; *v++ = 0;
                            }
                            inset = 2; *--relo = x2; *--relo = t - v;
                            c = *s++;
                            x2 = 0; while ( *s >= 0 ) x2 = 128*x2 + *s++;
                            switch ( c ) {
                                case TINDEX:
                                    *t++ = x2;
                                    if ( t[-1]+AM.OffsetIndex > AM.IndDum ) {
                                        x2 = t[-1]+AM.OffsetIndex - AM.IndDum;
                                        if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
                                    }
```

```
                                    break;
                                case TVECTOR:
                                    *t++ = x2; break;
                                case TNUMBER1:
                                    if ( x2 >= 0 && x2 < AM.OffsetIndex ) {
                                        *t++ = x2; break;
                                    }
                                default:
                                    MesPrint("&Illegal type of set inside tensor");
                                    error = 1;
                                    *t++ = x2;
                                    break;
                            }
                        }
                        else { switch ( c ) {
                            case TINDEX:
                                if ( inset < 2 ) *t++ = x2 + AM.OffsetIndex;
                                else *t++ = x2;
                                if ( x2+AM.OffsetIndex > AM.IndDum ) {
                                    x2 = x2+AM.OffsetIndex - AM.IndDum;
                                    if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
                                }
                                break;
                            case TGENINDEX:
                                *t++ = AC.DumNum + WILDOFFSET;
                                break;
                            case TVECTOR:
                                if ( inset < 2 ) *t++ = x2 + AM.OffsetVector;
                                else *t++ = x2;
                                break;
                            case TWILDARG:
                                *t++ = FUNNYWILD; *t++ = x2;
/*                              v[2] = 0; */
                                break;
                            case TDOLLAR:
                                *t++ = FUNNYDOLLAR; *t++ = x2;
                                break;
                            case TDUBIOUS:
                                if ( inset < 2 ) *t++ = x2 + AM.OffsetVector;
                                else *t++ = x2;
                                break;
                            case TSGAMMA:   /* Special gamma's */
                                if ( x3 != GAMMA ) {
                                    MesPrint("&5_,6_,7_ can only be used inside g_");
                                    error = -1;
                                }
                                *t++ = -x2;
                                break;
                            case TNUMBER:
                            case TNUMBER1:
                                if ( x2 >= AM.OffsetIndex && inset < 2 ) {
                                    MesPrint("&Value of constant index in tensor too large");
                                    error = -1;
                                }
                                *t++ = x2;
                                break;
                            default:
                                MesPrint("&Illegal object in tensor");
                                error = -1;
                                break;
                        }}
                        if ( inset >= 2 ) inset = 1;
                    } while ( *s == TCOMMA );
                }
                else {
dofunction:         firstsumarg = 1;
                    do {
                        s++;
                        c = *s++;
                        if ( c == TMINUS && ( *s == TVECTOR || *s == TNUMBER
                        || *s == TNUMBER1 || *s == TSUBEXP ) ) {
                            minus = 1; c = *s++;
                        }
                        else minus = 0;
                        base = ( c == TNUMBER ) ? 100: 128;
                        x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
/*
        !!!!!!!!  What if it does not fit?
*/
                        if ( firstsumarg ) {
                            firstsumarg = 0;
                            if ( sumlevel > 0 ) {
                                if ( c == TSYMBOL ) {
                                    sumlevel = x2; sumtype = SYMTOSYM;
                                }
                                else if ( c == TINDEX ) {
                                    sumlevel = x2+AM.OffsetIndex; sumtype = INDTOIND;
```

```
                                    if ( sumlevel > AM.IndDum ) {
                                        x2 = sumlevel - AM.IndDum;
                                        if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
                                    }
                                }
                            }
                        }
                        if ( *s == TWILDCARD ) {
                            if ( c == TSYMBOL ) x2 += 2*MAXPOWER;
                            else if ( c != TNUMBER ) x2 += WILDOFFSET;
                            s++;
                        }
                        switch ( c ) {
                            case TSYMBOL:
                                *t++ = -SYMBOL; *t++ = x2; break;
                            case TDOLLAR:
                                *t++ = -DOLLAREXPRESSION; *t++ = x2; break;
                            case TEXPRESSION:
                                *t++ = -EXPRESSION; *t++ = x2;
/*
                                The next code is added to facilitate parallel processing
                                We need to call GetTable here to make sure all processors
                                have the same numbering of all variables.
*/
                                if ( Expressions[x2].status == STOREDEXPRESSION ) {
                                    TMproto[0] = EXPRESSION;
                                    TMproto[1] = SUBEXPSIZE;
                                    TMproto[2] = x2;
                                    TMproto[3] = 1;
                                    { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
                                    AT.TMaddr = TMproto;
                                    PUTZERO(position);
/*
                                    if ( (
#ifdef WITHPTHREADS
                                        renumber =
#endif
                                            GetTable(x2,&position,0) ) == 0 ) {
                                        error = 1;
                                        MesPrint("&Problems getting information about stored expression %s(2)"
                                        ,EXPRNAME(x2));
                                    }
#ifdef WITHPTHREADS
                                    M_free(renumber->symb.lo,"VarSpace");
                                    M_free(renumber,"Renumber");
#endif
*/
                                    if ( ( renumber = GetTable(x2,&position,0) ) == 0 ) {
                                        error = 1;
                                        MesPrint("&Problems getting information about stored expression %s(2)"
                                        ,EXPRNAME(x2));
                                    }
                                    if ( renumber->symb.lo != AN.dummyrenumlist )
                                        M_free(renumber->symb.lo,"VarSpace");
                                    M_free(renumber,"Renumber");
                                    AR.StoreData.dirtyflag = 1;
                                }
                                break;
                            case TINDEX:
                                *t++ = -INDEX; *t++ = x2 + AM.OffsetIndex;
                                if ( t[-1] > AM.IndDum ) {
                                    x2 = t[-1] - AM.IndDum;
                                    if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
                                }
                                break;
                            case TGENINDEX:
                                *t++ = -INDEX; *t++ = AC.DumNum + WILDOFFSET;
                                break;
                            case TVECTOR:
                                if ( minus ) *t++ = -MINVECTOR;
                                else *t++ = -VECTOR;
                                *t++ = x2 + AM.OffsetVector;
                                break;
                            case TSGAMMA:   /* Special gamma's */
                                MesPrint("&5_,6_,7_ can only be used inside g_");
                                error = -1;
                                *t++ = -INDEX;
                                *t++ = -x2;
                                break;
                            case TDUBIOUS:
                                *t++ = -SYMBOL; *t++ = x2; break;
                            case TFUNCTION:
                                *t++ = -x2-FUNCTION;
                                break;
                            case TWILDARG:
```

```
                                *t++ = -ARGWILD; *t++ = x2; break;
                            case TSETDOL:
                                x2 = -x2;
                            case TSETNUM:
                                if ( inset == 0 ) {
                                    w1 = t; t += 2; w2 = t;
                                    while ( w1 > v ) *--w2 = *--w1;
                                    tsize = v; relo = AT.WorkTop;
                                    *v++ = SETSET; *v++ = 0;
                                    inset = 1;
                                }
                                *--relo = x2; *--relo = t-v+1;
                                c = *s++;
                                x2 = 0; while ( *s >= 0 ) x2 = 128*x2 + *s++;
                                switch ( c ) {
                                    case TFUNCTION:
                                        (*relo)--; *t++ = -x2-1; break;
                                    case TSYMBOL:
                                        *t++ = -SYMBOL; *t++ = x2; break;
                                    case TINDEX:
                                        *t++ = -INDEX; *t++ = x2;
                                        if ( x2+AM.OffsetIndex > AM.IndDum ) {
                                            x2 = x2+AM.OffsetIndex - AM.IndDum;
                                            if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
                                        }
                                        break;
                                    case TVECTOR:
                                        *t++ = -VECTOR; *t++ = x2; break;
                                    case TNUMBER1:
                                        *t++ = -SNUMBER; *t++ = x2; break;
                                    default:
                                        MesPrint("&Internal error 435");
                                        error = 1;
                                        *t++ = -SYMBOL; *t++ = x2; break;
                                }
                                break;
                            case TSUBEXP:
                                w2 = AC.ProtoType; i = w2[1];
                                w1 = t;
                                *t++ = i+ARGHEAD+4;
                                *t++ = 1;
                                FILLARG(t);
                                *t++ = i + 4;
                                while ( --i >= 0 ) *t++ = *w2++;
                                w1[ARGHEAD+3] = subexpbuffers[x2].subexpnum;
                                w1[ARGHEAD+5] = subexpbuffers[x2].buffernum;
                                if ( sumlevel > 0 ) {
                                    w1[0] += 4;
                                    w1[ARGHEAD] += 4;
                                    w1[ARGHEAD+2] += 4;
                                    *t++ = sumtype; *t++ = 4;
                                    *t++ = sumlevel; *t++ = sumlevel;
                                }
                                *t++ = 1; *t++ = 1;
                                if ( minus ) *t++ = -3;
                                else         *t++ =  3;
                                break;
                            case TNUMBER:
                            case TNUMBER1:
                                if ( minus ) x2 = -x2;
                                *t++ = -SNUMBER;
                                *t++ = x2;
                                break;
                            default:
                                MesPrint("&Illegal object in function");
                                error = -1;
                                break;
                        }
                    } while ( *s == TCOMMA );
                }
                if ( *s != TFUNCLOSE ) {
                    MesPrint("&Illegal argument field for function. Expected )");
                    return(-1);
                }
                s++; sumlevel = 0;
                v[1] = t-v;
/*
                if ( *v == AM.termfunnum && ( v[1] != FUNHEAD+2 ||
                v[FUNHEAD] != -DOLLAREXPRESSION ) ) {
                    MesPrint("&The function term_ can only have one argument with a single $-expression");
                    error = 1;
                }
*/
                goto fin;
            case TDUBIOUS:
                x1 = 0; while ( *s >= 0 ) x1 = 128*x1 + *s++;
```

```
                if ( *s == TWILDCARD ) s++;
                if ( *s == TDOT ) goto dovector;
                if ( *s == TFUNOPEN ) {
                    x1 += FUNCTION;
                    cc = 1;
                    v = t; *t++ = x1; *t++ = FUNHEAD; *t++ = DIRTYFLAG;
                    if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
                            t[-1] |= MUSTCLEANPRF;
                    FILLFUN3(t)
                    needarg = -1; goto dofunction;
                }
                *t++ = SYMBOL; *t++ = 4; *t++ = 0;
                if ( inset ) *relo = 2;
                goto TryPower;
            case TSUBEXP:
                x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
                if ( *s == TPOWER ) {
                    s++; c = *s++;
                    base = ( c == TNUMBER ) ? 100: 128;
                    x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
                    if ( *s == TWILDCARD ) { x2 += 2*MAXPOWER; s++; }
                    else if ( c == TSYMBOL ) x2 += 2*MAXPOWER;
                }
                else x2 = 1;
                r = AC.ProtoType; n = r[1] - 5; r += 5;
                *t++ = SUBEXPRESSION; *t++ = r[-4];
                *t++ = subexpbuffers[x1].subexpnum;
                *t++ = x2*deno;
                *t++ = subexpbuffers[x1].buffernum;
                NCOPY(t,r,n);
                if ( cbuf[subexpbuffers[x1].buffernum].CanCommu[subexpbuffers[x1].subexpnum] ) cc = 1;
                deno = 1;
                break;
            case TMULTIPLY:
                mulflag = 1;
                break;
            case TDIVIDE:
                mulflag = 1;
                deno = -deno;
                break;
            case TEXPRESSION:
                cc = 1;
                x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
                v = t;
                *t++ = EXPRESSION; *t++ = SUBEXPSIZE; *t++ = x1; *t++ = deno;
                *t++ = 0; FILLSUB(t)
/*
                Here we had some erroneous code before. It should be after
                the reading of the parameters as it is now (after 15-jan-2007).
                Thomas Hahn noticed this and reported it.
*/
                if ( *s == TFUNOPEN ) {
                    do {
                        s++; c = *s++;
                        base = ( c == TNUMBER ) ? 100: 128;
                        x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
                        switch ( c ) {
                            case TSYMBOL:
                                *t++ = SYMBOL; *t++ = 4; *t++ = x2; *t++ = 1;
                                break;
                            case TINDEX:
                                *t++ = INDEX; *t++ = 3; *t++ = x2+AM.OffsetIndex;
                                if ( t[-1] > AM.IndDum ) {
                                    x2 = t[-1] - AM.IndDum;
                                    if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
                                }
                                break;
                            case TVECTOR:
                                *t++ = INDEX; *t++ = 3; *t++ = x2+AM.OffsetVector;
                                break;
                            case TFUNCTION:
                                *t++ = x2+FUNCTION; *t++ = 2; break;
                            case TNUMBER:
                            case TNUMBER1:
                                if ( x2 >= AM.OffsetIndex || x2 < 0 ) {
                                    MesPrint("&Index as argument of expression has illegal value");
                                    error = -1;
                                }
                                *t++ = INDEX; *t++ = 3; *t++ = x2; break;
                            case TSETDOL:
                                x2 = -x2;
                            case TSETNUM:
                                if ( inset == 0 ) {
                                    w1 = t; t += 2; w2 = t;
                                    while ( w1 > v ) *--w2 = *--w1;
                                    tsize = v; relo = AT.WorkTop;
                                    *v++ = SETSET; *v++ = 0;
```

```
                                    inset = 1;
                                }
                                *--relo = x2; *--relo = t-v+2;
                                c = *s++;
                                x2 = 0; while ( *s >= 0 ) x2 = 128*x2 + *s++;
                                switch ( c ) {
                                    case TFUNCTION:
                                        *relo -= 2; *t++ = -x2-1; break;
                                    case TSYMBOL:
                                        *t++ = SYMBOL; *t++ = 4; *t++ = x2; *t++ = 1; break;
                                    case TINDEX:
                                        *t++ = INDEX; *t++ = 3; *t++ = x2;
                                        if ( x2+AM.OffsetIndex > AM.IndDum ) {
                                            x2 = x2+AM.OffsetIndex - AM.IndDum;
                                            if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
                                        }
                                        break;
                                    case TVECTOR:
                                        *t++ = VECTOR; *t++ = 3; *t++ = x2; break;
                                    case TNUMBER1:
                                        *t++ = SNUMBER; *t++ = 4; *t++ = x2; *t++ = 1; break;
                                    default:
                                        MesPrint("&Internal error 435");
                                        error = 1;
                                        *t++ = SYMBOL; *t++ = 4; *t++ = x2; *t++ = 1; break;
                                }
                                break;
                            default:
                                MesPrint("&Argument of expression can only be symbol, index, vector or function");
                                error = -1;
                                break;
                        }
                    } while ( *s == TCOMMA );
                    if ( *s != TFUNCLOSE ) {
                        MesPrint("&Illegal object in argument field for expression");
                        error = -1;
                        while ( *s != TFUNCLOSE ) s++;
                    }
                    s++;
                }
                r = AC.ProtoType; n = r[1];
                if ( n > SUBEXPSIZE ) {
                    *t++ = WILDCARDS; *t++ = n+2;
                    NCOPY(t,r,n);
                }
/*
                Code added for parallel processing.
                This is different from the other occurrences to test immediately
                for renumbering. Here we have to read the parameters first.
*/
                if ( Expressions[x1].status == STOREDEXPRESSION ) {
                    v[1] = t-v;
                    AT.TMaddr = v;
                    PUTZERO(position);
/*
                    if ( (
#ifdef WITHPTHREADS
                        renumber =
#endif
                            GetTable(x1,&position,0) ) == 0 ) {
                        error = 1;
                        MesPrint("&Problems getting information about stored expression %s(3)"
                        ,EXPRNAME(x1));
                    }
#ifdef WITHPTHREADS
                    M_free(renumber->symb.lo,"VarSpace");
                    M_free(renumber,"Renumber");
#endif
*/
                    if ( ( renumber = GetTable(x1,&position,0) ) == 0 ) {
                        error = 1;
                        MesPrint("&Problems getting information about stored expression %s(3)"
                        ,EXPRNAME(x1));
                    }
                    if ( renumber->symb.lo != AN.dummyrenumlist )
                        M_free(renumber->symb.lo,"VarSpace");
                    M_free(renumber,"Renumber");
                    AR.StoreData.dirtyflag = 1;
                }
                if ( *s == LBRACE ) {
/*
                    This should be one term that should be inserted
                    FROMBRAC size+2 ( term )
                    Because this term should have been translated
                    already we can copy it from the 'subexpression'
*/
```

```
                    s++;
                    if ( *s != TSUBEXP ) {
                        MesPrint("&Internal error 23");
                        Terminate(-1);
                    }
                    s++; x2 = 0; while ( *s >= 0 ) { x2 = 128*x2 + *s++; }
                    r = cbuf[subexpbuffers[x2].buffernum].rhs[subexpbuffers[x2].subexpnum];
                    *t++ = FROMBRAC; *t++ = *r+2;
                    n = *r;
                    NCOPY(t,r,n);
                    if ( *r != 0 ) {
                        MesPrint("&Object between [] in expression should be a single term");
                        error = -1;
                    }
                    if ( *s != RBRACE ) {
                        MesPrint("&Internal error 23b");
                        Terminate(-1);
                    }
                    s++;
                }
                if ( *s == TPOWER ) {
                    s++; c = *s++;
                    base = ( c == TNUMBER ) ? 100: 128;
                    x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
                    if ( *s == TWILDCARD || c == TSYMBOL ) { x2 += 2*MAXPOWER; s++; }
                    v[3] = x2;
                }
                v[1] = t - v;
                deno = 1;
                break;
            case TNUMBER:
                if ( *s == 0 ) {
                    s++;
                    if ( *s == TPOWER ) {
                        s++; if ( *s == TMINUS ) { s++; deno = -deno; }
                        c = *s++; base = ( c == TNUMBER ) ? 100: 128;
                        x2 = 0; while ( *s >= 0 ) { x2 = x2*base + *s++; }
                        if ( x2 == 0 ) {
                            error = -1;
                            MesPrint("&Encountered 0^0 during compilation");
                        }
                        if ( deno < 0 ) {
                            error = -1;
                            MesPrint("&Division by zero during compilation (0 to the power negative number)");
                        }
                    }
                    else if ( deno < 0 ) {
                        error = -1;
                        MesPrint("&Division by zero during compilation");
                    }
                    sign = 0; break; /* term is zero */
                }
                y = *s++;
                if ( *s >= 0 ) { y = 100*y + *s++; }
                innum[0] = y; nin = 1;
                while ( *s >= 0 ) {
                    y = *s++; x2 = 100;
                    if ( *s >= 0 ) { y = 100*y + *s++; x2 = 10000; }
                    Product(innum,&nin,(WORD)x2);
                    if ( y ) AddLong(innum,nin,(UWORD *)(&y),(WORD)1,innum,&nin);
                }
docoef:
                if ( *s == TPOWER ) {
                    s++; if ( *s == TMINUS ) { s++; deno = -deno; }
                    c = *s++; base = ( c == TNUMBER ) ? 100: 128;
                    x2 = 0; while ( *s >= 0 ) { x2 = x2*base + *s++; }
                    if ( x2 == 0 ) {
                        innum[0] = 1; nin = 1;
                    }
                    else if ( RaisPow(BHEAD innum,&nin,x2) ) {
                        error = -1; innum[0] = 1; nin = 1;
                    }
                }
                if ( deno > 0 ) {
                    Simplify(BHEAD innum,&nin,denominator,&ndenominator);
                    for ( i = 0; i < nnumerator; i++ ) CGscrat7[i] = numerator[i];
                    MulLong(innum,nin,CGscrat7,nnumerator,numerator,&nnumerator);
                }
                else if ( deno < 0 ) {
                    Simplify(BHEAD innum,&nin,numerator,&nnumerator);
                    for ( i = 0; i < ndenominator; i++ ) CGscrat7[i] = denominator[i];
                    MulLong(innum,nin,CGscrat7,ndenominator,denominator,&ndenominator);
                }
                deno = 1;
                break;
            case TNUMBER1:
```

```
                if ( *s == 0 ) { s++; sign = 0; break; /* term is zero */ }
                y = *s++;
                if ( *s >= 0 ) { y = 128*y + *s++; }
                if ( inset == 0 ) {
                    innum[0] = y; nin = 1;
                    while ( *s >= 0 ) {
                        y = *s++; x2 = 128;
                        if ( *s >= 0 ) { y = 128*y + *s++; x2 = 16384; }
                        Product(innum,&nin,(WORD)x2);
                        if ( y ) AddLong(innum,nin,(UWORD *)&y,(WORD)1,innum,&nin);
                    }
                    goto docoef;
                }
                *relo = 2; *t++ = SNUMBER; *t++ = 4; *t++ = y;
                goto TryPower;
            case TDOLLAR:
            {
                WORD *powplace;
                x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
                if ( AR.Eside != LHSIDE ) {
                    *t++ = SUBEXPRESSION; *t++ = SUBEXPSIZE; *t++ = x1;
                }
                else {
                    *t++ = DOLLAREXPRESSION; *t++ = SUBEXPSIZE; *t++ = x1;
                }
                powplace = t; t++;
                *t++ = AM.dbufnum; FILLSUB(t)
/*
                Now we have to test for factors of dollars with [ ] and [ [ ]]
*/
                if ( *s == LBRACE ) {
                    int bracelevel = 1;
                    s++;
                    while ( bracelevel > 0 ) {
                        if ( *s == RBRACE ) {
                            bracelevel--; s++;
                        }
                        else if ( *s == TNUMBER ) {
                            s++;
                            x2 = 0; while ( *s >= 0 ) { x2 = 100*x2 + *s++; }
                            *t++ = DOLLAREXPR2; *t++ = 3; *t++ = -x2-1;
CloseBraces:
                            while ( bracelevel > 0 ) {
                                if ( *s != RBRACE ) {
ErrorBraces:
                                    error = -1;
                                    MesPrint("&Improper use of [] in $-variable.");
                                    return(error);
                                }
                                else {
                                    s++; bracelevel--;
                                }
                            }
                        }
                        else if ( *s == TDOLLAR ) {
                            s++;
                            x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
                            *t++ = DOLLAREXPR2; *t++ = 3; *t++ = x1;
                            if ( *s == RBRACE ) goto CloseBraces;
                            else if ( *s == LBRACE ) {
                                s++; bracelevel++;
                            }
                        }
                        else goto ErrorBraces;
                    }
                }
/*
                Finally we can continue with the power
*/
                if ( *s == TPOWER ) {
                    s++;
                    if ( *s == TMINUS ) { s++; deno = -deno; }
                    c = *s++;
                    base = ( c == TNUMBER ) ? 100: 128;
                    x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
                    if ( c == TSYMBOL ) {
                        if ( *s == TWILDCARD ) s++;
                        x2 += 2*MAXPOWER;
                    }
                    *powplace = deno*x2;
                }
                else *powplace = deno;
                deno = 1;
/*
                if ( inset ) {
                    while ( relo < AT.WorkTop ) *t++ = *relo++;
                    inset = 0; tsize[1] = t - tsize;
```

```
                }
*/
            }
                break;
            case TSETNUM:
                inset = 1; tsize = t; relo = AT.WorkTop;
                *t++ = SETSET; *t++ = 0;
                x1 = 0; while ( *s >= 0 ) x1 = x1*128 + *s++;
                *--relo = x1; *--relo = 0;
                break;
            case TSETDOL:
                inset = 1; tsize = t; relo = AT.WorkTop;
                *t++ = SETSET; *t++ = 0;
                x1 = 0; while ( *s >= 0 ) x1 = x1*128 + *s++;
                *--relo = -x1; *--relo = 0;
                break;
            case TFUNOPEN:
                MesPrint("&Illegal use of function arguments");
                error = -1;
                funflag = 1;
                deno = 1;
                break;
            case TFUNCLOSE:
                if ( funflag == 0 )
                    MesPrint("&Illegal use of function arguments");
                error = -1;
                funflag = 0;
                deno = 1;
                break;
            case TSGAMMA:
                MesPrint("&Illegal use special gamma symbols 5_, 6_, 7_");
                error = -1;
                funflag = 0;
                deno = 1;
                break;
            default:
                MesPrint("&Internal error in code generator. Unknown object: %d",c);
                error = -1;
                deno = 1;
                break;
            }
        }
    }
    if ( mulflag ) {
        MesPrint("&Irregular end of statement.");
        error = 1;
    }
    if ( !first && error == 0 ) {
        *term = t-term;
        C->NumTerms[numexp]++;
        if ( cc && sign ) C->CanCommu[numexp]++;
        error = CompleteTerm(term,numerator,denominator,nnumerator,ndenominator,sign);
    }
    AT.WorkPointer = oldwork;
    if ( error ) return(-1);
    AddToCB(C,0)
    if ( AC.CompileLevel > 0 && AR.Eside != LHSIDE ) {
        /* See whether we have this one already */
        error = InsTree(AC.cbufnum,C->numrhs);
        if ( error < (C->numrhs) ) {
            C->Pointer = C->rhs[C->numrhs--];
            return(error);
        }
    }
    return(C->numrhs);
OverWork:
    MLOCK(ErrorMessageLock);
    MesWork();
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
        #] CodeGenerator :
        #[ CompleteTerm :

        Completes the term
        Puts it in the buffer
*/

int CompleteTerm(WORD *term, UWORD *numer, UWORD *denom, WORD nnum, WORD nden, int sign)
{
    int nsize, i;
    WORD *t;
    if ( sign == 0 ) return(0);     /* Term is zero */
    if ( nnum >= nden ) nsize = nnum;
    else                nsize = nden;
```

```
    t = term + *term;
    for ( i = 0; i < nnum; i++ ) *t++ = numer[i];
    for ( ; i < nsize; i++ ) *t++ = 0;
    for ( i = 0; i < nden; i++ ) *t++ = denom[i];
    for ( ; i < nsize; i++ ) *t++ = 0;
    *t++ = (2*nsize+1)*sign;
    *term = t - term;
    AddNtoC(AC.cbufnum,*term,term,7);
    return(0);
}

/*
        #] CompleteTerm :
        #[ CodeFactors :

        This routine does the part of reading in in terms of factors.
        If there is more than one term at this level we have only one
        factor. In that case any expression should first be unfactorized.
        Then the whole expression gets read as a new subexpression and finally
        we generate factor_*subexpression.
        If the whole has only multiplications we have factors. Then the
        nasty thing is powers of objects and in particular powers of
        factorized expressions or dollars.
        For a power we generate a new subexpression of the type
          1+factor_+...+factor_^(power-1)
        with which we multiply.

        WE HAVE NOT YET WORRIED ABOUT SETS
*/

int CodeFactors(SBYTE *tokens)
{
    GETIDENTITY
    EXPRESSIONS e = Expressions + AR.CurExpr;
    int nfactor = 1, nparenthesis, i, last = 0, error = 0;
    SBYTE *t, *startobject, *tt, *s1, *out, *outtokens;
    WORD nexp, subexp = 0, power, pow, x2, powfactor, first;
/*
    First scan the number of factors
*/
    t = tokens;
    while ( *t != TENDOFIT ) {
        if ( *t >= 0 ) { while ( *t >= 0 ) t++; continue; }
        if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN ) {
            nparenthesis = 0; t++;
            while ( nparenthesis >= 0 ) {
                if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN ) nparenthesis++;
                else if ( *t == RPARENTHESIS || *t == RBRACE || *t == TSETCLOSE || *t == TFUNCLOSE ) nparenthesis--;
                t++;
            }
            continue;
        }
        else if ( ( *t == TPLUS || *t == TMINUS ) && ( t > tokens )
        && ( t[-1] != TPLUS && t[-1] != TMINUS ) ) {
            if ( t[-1] >= 0 || t[-1] == RPARENTHESIS || t[-1] == RBRACE
            || t[-1] == TSETCLOSE || t[-1] == TFUNCLOSE ) {
                subexp = CodeGenerator(tokens);
                if ( subexp < 0 ) error = -1;
                if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
                    MesPrint("&More than %d subexpressions inside one expression",(WORD)MAXSUBEXPRESSIONS);
                    Terminate(-1);
                }
                if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
                    DoubleBuffer((void **)((VOID *)(&subexpbuffers))
                    ,(void **)((VOID *)(&topsubexpbuffers)),sizeof(SUBBUF),"subexpbuffers");
                }
                subexpbuffers[insubexpbuffers].subexpnum = subexp;
                subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
                subexp = insubexpbuffers++;
                t = tokens;
                *t++ = TSYMBOL; *t++ = FACTORSYMBOL;
                *t++ = TMULTIPLY; *t++ = TSUBEXP;
                PUTNUMBER128(t,subexp)
                *t++ = TENDOFIT;
                e->numfactors = 1;
                e->vflags |= ISFACTORIZED;
                return(subexp);
            }
        }
        else if ( ( *t == TMULTIPLY || *t == TDIVIDE ) && t > tokens ) {
            nfactor++;
        }
        else if ( *t == TEXPRESSION ) {
            t++;
```

```
            nexp = 0; while ( *t >= 0 ) { nexp = nexp*128 + *t++; }
            if ( *t == LBRACE ) continue;
            if ( ( AS.Oldvflags[nexp] & ISFACTORIZED ) != 0 ) {
                nfactor += AS.OldNumFactors[nexp];
            }
            else { nfactor++; }
            continue;
        }
        else if ( *t == TDOLLAR ) {
            t++;
            nexp = 0; while ( *t >= 0 ) { nexp = nexp*128 + *t++; }
            if ( *t == LBRACE ) continue;
            if ( Dollars[nexp].nfactors > 0 ) {
                nfactor += Dollars[nexp].nfactors;
            }
            else { nfactor++; }
            continue;
        }
        t++;
    }
/*
    Now the real pass.
    nfactor is a not so reliable measure for the space we need.
*/
    outtokens = (SBYTE *)Malloc1(((t-tokens)+(nfactor+2)*25)*sizeof(SBYTE),"CodeFactors");
    out = outtokens;
    t = tokens; first = 1; powfactor = 1;
    while ( *t == TPLUS || *t == TMINUS ) { if ( *t == TMINUS ) first = -first; t++; }
    if ( first < 0 ) {
        *out++ = TMINUS; *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
        *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
        powfactor++;
    }
    startobject = t; power = 1;
    while ( *t != TENDOFIT ) {
        if ( *t >= 0 ) { while ( *t >= 0 ) t++; continue; }
        if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN ) {
            nparenthesis = 0; t++;
            while ( nparenthesis >= 0 ) {
                if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN ) nparenthesis++;
                else if ( *t == RPARENTHESIS || *t == RBRACE || *t == TSETCLOSE || *t == TFUNCLOSE ) nparenthesis--;
                t++;
            }
            continue;
        }
        else if ( ( *t == TMULTIPLY || *t == TDIVIDE ) && ( t > tokens ) ) {
            if ( t[-1] >= 0 || t[-1] == RPARENTHESIS || t[-1] == RBRACE
            || t[-1] == TSETCLOSE || t[-1] == TFUNCLOSE ) {
dolast:
                if ( startobject ) {    /* apparently power is 1 or -1 */
                    *out++ = TPLUS;
                    if ( power < 0 ) { *out++ = TNUMBER; *out++ = 1; *out++ = TDIVIDE; }
                    s1 = startobject;
                    while ( s1 < t ) *out++ = *s1++;
                    *out++ = TMULTIPLY; *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
                    *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
                    powfactor++;
                }
                if ( last ) { startobject = 0; break; }
                startobject = t+1;
                if ( *t == TDIVIDE ) power = -1;
                if ( *t == TMULTIPLY ) power = 1;
            }
        }
        else if ( *t == TPOWER ) {
            pow = 1;
            tt = t+1;
            while ( ( *tt == TMINUS ) || ( *tt == TPLUS ) ) {
                if ( *tt == TMINUS ) pow = -pow;
                tt++;
            }
            if ( *tt == TSYMBOL ) {
                tt++; while ( *tt >= 0 ) tt++;
                t = tt; continue;
            }
            tt++; x2 = 0; while ( *tt >= 0 ) { x2 = 100*x2 + *tt++; }
/*
            We have an object in startobject till t. The power is
            power*pow*x2
*/
            power = power*pow*x2;
            if ( power < 0 ) { pow = -power; power = -1; }
            else if ( power == 0 ) { t = tt; startobject = tt; continue; }
            else { pow = power; power = 1; }
            *out++ = TPLUS;
```

```
            if ( pow > 1 ) {
                subexp = GenerateFactors(pow,1);
                if ( subexp < 0 ) { error = -1; subexp = 0; }
                *out++ = TSUBEXP; PUTNUMBER128(out,subexp);
            }
            *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
            *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
            powfactor += pow;
            if ( power > 0 ) *out++ = TMULTIPLY;
            else *out++ = TDIVIDE;
            s1 = startobject; while ( s1 < t ) *out++ = *s1++;
            startobject = 0; t = tt; continue;
        }
        else if ( *t == TEXPRESSION ) {
            startobject = t;
            t++;
            nexp = 0; while ( *t >= 0 ) { nexp = nexp*128 + *t++; }
            if ( *t == LBRACE ) continue;
            if ( *t == LPARENTHESIS ) {
                nparenthesis = 0; t++;
                while ( nparenthesis >= 0 ) {
                    if ( *t == LPARENTHESIS ) nparenthesis++;
                    else if ( *t == RPARENTHESIS ) nparenthesis--;
                    t++;
                }
            }
            if ( ( AS.Oldvflags[nexp] & ISFACTORIZED ) == 0 ) continue;
            if ( *t == TPOWER ) {
                pow = 1;
                tt = t+1;
                while ( ( *tt == TMINUS ) || ( *tt == TPLUS ) ) {
                    if ( *tt == TMINUS ) pow = -pow;
                    tt++;
                }
                if ( *tt != TNUMBER ) {
                    MesPrint("Internal problems(1) in CodeFactors");
                    return(-1);
                }
                tt++; x2 = 0; while ( *tt >= 0 ) { x2 = 100*x2 + *tt++; }
/*
                We have an object in startobject till t. The power is
                power*pow*x2
*/
dopower:
                power = power*pow*x2;
                if ( power < 0 ) { pow = -power; power = -1; }
                else if ( power == 0 ) { t = tt; startobject = tt; continue; }
                else { pow = power; power = 1; }
                *out++ = TPLUS;
                if ( pow > 1 ) {
                    subexp = GenerateFactors(pow,AS.OldNumFactors[nexp]);
                    if ( subexp < 0 ) { error = -1; subexp = 0; }
                    *out++ = TSUBEXP; PUTNUMBER128(out,subexp)
                    *out++ = TMULTIPLY;
                }
                i = powfactor-1;
                if ( i > 0 ) {
                    *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
                    if ( i > 1 ) {
                        *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,i)
                    }
                    *out++ = TMULTIPLY;
                }
                powfactor += AS.OldNumFactors[nexp]*pow;
                s1 = startobject;
                while ( s1 < t ) *out++ = *s1++;
                startobject = 0; t = tt; continue;
            }
            else {
                tt = t; pow = 1; x2 = 1; goto dopower;
            }
        }
        else if ( *t == TDOLLAR ) {
            startobject = t;
            t++;
            nexp = 0; while ( *t >= 0 ) { nexp = nexp*128 + *t++; }
            if ( *t == LBRACE ) continue;
            if ( Dollars[nexp].nfactors == 0 ) continue;
            if ( *t == TPOWER ) {
                pow = 1;
                tt = t+1;
                while ( ( *tt == TMINUS ) || ( *tt == TPLUS ) ) {
                    if ( *tt == TMINUS ) pow = -pow;
                    tt++;
                }
                if ( *tt != TNUMBER ) {
                    MesPrint("Internal problems(2) in CodeFactors");
```

```
                    return(-1);
                }
                tt++; x2 = 0; while ( *tt >= 0 ) { x2 = 100*x2 + *tt++; }
/*
                We have an object in startobject till t. The power is
                power*pow*x2
*/
dopowerd:
                power = power*pow*x2;
                if ( power < 0 ) { pow = -power; power = -1; }
                else if ( power == 0 ) { t = tt; startobject = tt; continue; }
                else { pow = power; power = 1; }
                if ( pow > 1 ) {
                    subexp = GenerateFactors(pow,1);
                    if ( subexp < 0 ) { error = -1; subexp = 0; }
                }
                for ( i = 1; i <= Dollars[nexp].nfactors; i++ ) {
                    s1 = startobject; *out++ = TPLUS;
                    while ( s1 < t ) *out++ = *s1++;
                    *out++ = LBRACE; *out++ = TNUMBER; PUTNUMBER128(out,i)
                    *out++ = RBRACE;
                    *out++ = TMULTIPLY;
                    *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
                    *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
                    powfactor += pow;
                    if ( pow > 1 ) {
                        *out++ = TSUBEXP; PUTNUMBER128(out,subexp)
                    }
                }
                startobject = 0; t = tt; continue;
            }
            else {
                tt = t; pow = 1; x2 = 1; goto dopowerd;
            }
        }
        t++;
    }
    if ( last == 0 ) { last = 1; goto dolast; }
    *out = TENDOFIT;
    e->numfactors = powfactor-1;
    e->vflags |= ISFACTORIZED;
    subexp = CodeGenerator(outtokens);
    if ( subexp < 0 ) error = -1;
    if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
        MesPrint("&More than %d subexpressions inside one expression",(WORD)MAXSUBEXPRESSIONS);
        Terminate(-1);
    }
    if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
        DoubleBuffer((void **)((VOID *)(&subexpbuffers))
        ,(void **)((VOID *)(&topsubexpbuffers)),sizeof(SUBBUF),"subexpbuffers");
    }
    subexpbuffers[insubexpbuffers].subexpnum = subexp;
    subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
    subexp = insubexpbuffers++;
    M_free(outtokens,"CodeFactors");
    s1 = tokens;
    *s1++ = TSUBEXP; PUTNUMBER128(s1,subexp); *s1++ = TENDOFIT;
    if ( error < 0 ) return(-1);
    else return(subexp);
}

/*
        #] CodeFactors :
        #[ GenerateFactors :

    Generates an expression of the type
      1+factor_+factor_^2+...+factor_^(n-1)
    (this is if inc=1)
    Returns the subexpression pointer of it.
*/

WORD GenerateFactors(WORD n,WORD inc)
{
    WORD subexp;
    int i, error = 0;
    SBYTE *s;
    SBYTE *tokenbuffer = (SBYTE *)Malloc1(8*n*sizeof(SBYTE),"GenerateFactors");
    s = tokenbuffer;
    *s++ = TNUMBER; *s++ = 1;
    for ( i = inc; i < n*inc; i += inc ) {
        *s++ = TPLUS; *s++ = TSYMBOL; *s++ = FACTORSYMBOL;
        if ( i > 1 ) {
            *s++ = TPOWER; *s++ = TNUMBER;
            PUTNUMBER100(s,i)
        }
    }
    *s++ = TENDOFIT;
```

```
    subexp = CodeGenerator(tokenbuffer);
    if ( subexp < 0 ) error = -1;
    M_free(tokenbuffer,"GenerateFactors");
    if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
        MesPrint("&More than %d subexpressions inside one expression",(WORD)MAXSUBEXPRESSIONS);
        Terminate(-1);
    }
    if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
        DoubleBuffer((void **)((VOID *)(&subexpbuffers))
        ,(void **)((VOID *)(&topsubexpbuffers)),sizeof(SUBBUF),"subexpbuffers");
    }
    subexpbuffers[insubexpbuffers].subexpnum = subexp;
    subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
    subexp = insubexpbuffers++;
    if ( error < 0 ) return(error);
    return(subexp);
}

/*
        #] GenerateFactors :
    #] Compiler :
*/
```

4.13 compress.c File Reference

`#include "form3.h"`

Include dependency graph for compress.c:

4.13.1 Detailed Description

The routines for the use of gzip (de)compression of the information in the sort file.

Definition in file compress.c.

4.14 compress.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
```

```
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

#include "form3.h"

#ifdef WITHZLIB
/*
#define GZIPDEBUG

    Low level routines for dealing with zlib during sorting and handling
    the scratch files. Work started 5-sep-2005.
    The .sor file handling was more or less completed on 8-sep-2005
    The handling of the scratch files still needs some thinking.
    Complications are:
        gzip compression should be per expression, not per buffer.
        No gzip compression for expressions with a bracket index.
        Separate decompression buffers for expressions in the rhs.
        This last one will involve more buffer work and organization.
        Information about compression should be stored for each expr.
            (including what method/program was used)
    Note: Be careful with compression. By far the most compact method
    is the original problem....

    #[ Variables :

    The following variables are to contain the intermediate buffers
    for the inflation of the various patches in the sort file.
    There can be up to MaxFpatches (FilePatches in the setup) and hence
    we can have that many streams simultaneously. We set this up once
    and only when needed.
        (in struct A.N or AB[threadnum].N)
    Bytef **AN.ziobufnum;
    Bytef *AN.ziobuffers;
*/

/*
    #] Variables :
    #[ SetupOutputGZIP :

    Routine prepares a gzip output stream for the given file.
*/

int SetupOutputGZIP(FILEHANDLE *f)
{
    GETIDENTITY

    if ( AT.SS != AT.S0 ) return(0);
    if ( AR.NoCompress == 1 ) return(0);
    if ( AR.gzipCompress <= 0 ) return(0);

    if ( f->ziobuffer == 0 ) {
/*
        1: Allocate a struct for the gzip stream:
*/
        f->zsp = Malloc1(sizeof(z_stream),"output zstream");
/*
        2: Allocate the output buffer.
*/
        f->ziobuffer =
            (Bytef *)Malloc1(f->ziosize*sizeof(char),"output zbuffer");
        if ( f->zsp == 0 || f->ziobuffer == 0 ) {
            MLOCK(ErrorMessageLock);
            MesCall("SetupOutputGZIP");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
    }
/*
    3: Set the default fields:
*/
    f->zsp->zalloc = Z_NULL;
    f->zsp->zfree  = Z_NULL;
    f->zsp->opaque = Z_NULL;
/*
    4: Set the output space:
```

```
*/
    f->zsp->next_out  = f->ziobuffer;
    f->zsp->avail_out = f->ziosize;
    f->zsp->total_out = 0;
/*
    5: Set the input space:
*/
    f->zsp->next_in  = (Bytef *)(f->PObuffer);
    f->zsp->avail_in = (Bytef *)(f->POfill) - (Bytef *)(f->PObuffer);
    f->zsp->total_in = 0;
/*
    6: Initiate the deflation
*/
    if ( deflateInit(f->zsp,AR.gzipCompress) != Z_OK ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Error from zlib: %s",f->zsp->msg);
        MesCall("SetupOutputGZIP");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }

    return(0);
}

/*
    #] SetupOutputGZIP :
    #[ PutOutputGZIP :

    Routine is called when the PObuffer of f is full.
    The contents of it will be compressed and whenever the output buffer
    f->ziobuffer is full it will be written and the output buffer
    will be reset.
    Upon exit the input buffer will be cleared.
*/

int PutOutputGZIP(FILEHANDLE *f)
{
    GETIDENTITY
    int zerror;
/*
    First set the number of bytes in the input
*/
    f->zsp->next_in  = (Bytef *)(f->PObuffer);
    f->zsp->avail_in = (Bytef *)(f->POfill) - (Bytef *)(f->PObuffer);
    f->zsp->total_in = 0;

    while ( ( zerror = deflate(f->zsp,Z_NO_FLUSH) ) == Z_OK ) {
        if ( f->zsp->avail_out == 0 ) {
/*
            ziobuffer is full. Write the output.
*/
#ifdef GZIPDEBUG
            {
                char *s = (char *)((UBYTE *)(f->ziobuffer)+f->ziosize);
                MLOCK(ErrorMessageLock);
                MesPrint("%wWriting %l bytes at %10p: %d %d %d %d %d"
                ,f->ziosize,&(f->POposition),s[-5],s[-4],s[-3],s[-2],s[-1]);
                MUNLOCK(ErrorMessageLock);
            }
#endif
#ifdef ALLLOCK
            LOCK(f->pthreadslock);
#endif
            if ( f == AR.hidefile ) {
                LOCK(AS.inputslock);
            }
            SeekFile(f->handle,&(f->POposition),SEEK_SET);
            if ( WriteFile(f->handle,(UBYTE *)(f->ziobuffer),f->ziosize)
                        != f->ziosize ) {
                if ( f == AR.hidefile ) {
                    UNLOCK(AS.inputslock);
                }
#ifdef ALLLOCK
                UNLOCK(f->pthreadslock);
#endif
                MLOCK(ErrorMessageLock);
                MesPrint("%wWrite error during compressed sort. Disk full?");
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
            if ( f == AR.hidefile ) {
                UNLOCK(AS.inputslock);
            }
#ifdef ALLLOCK
            UNLOCK(f->pthreadslock);
#endif
            ADDPOS(f->filesize,f->ziosize);
```

```
            ADDPOS(f->POposition,f->ziosize);
#ifdef WITHPTHREADS
            if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
                if ( f->handle >= 0 ) SynchFile(f->handle);
            }
#endif
/*
            Reset the output
*/
            f->zsp->next_out  = f->ziobuffer;
            f->zsp->avail_out = f->ziosize;
            f->zsp->total_out = 0;
        }
        else if ( f->zsp->avail_in == 0 ) {
/*
            We compressed everything and it sits in ziobuffer. Finish
*/
            return(0);
        }
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("%w avail_in = %d, avail_out = %d.",f->zsp->avail_in,f->zsp->avail_out);
            MUNLOCK(ErrorMessageLock);
            break;
        }
    }
    MLOCK(ErrorMessageLock);
    MesPrint("%wError in gzip handling of output. zerror = %d",zerror);
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
    #] PutOutputGZIP :
    #[ FlushOutputGZIP :

    Routine is called to flush a stream. The compression of the input buffer
    will be completed and the contents of f->ziobuffer will be written.
    Both buffers will be cleared.
*/

int FlushOutputGZIP(FILEHANDLE *f)
{
    GETIDENTITY
    int zerror;
/*
    Set the proper parameters
*/
    f->zsp->next_in  = (Bytef *)(f->PObuffer);
    f->zsp->avail_in = (Bytef *)(f->POfill) - (Bytef *)(f->PObuffer);
    f->zsp->total_in = 0;

    while ( ( zerror = deflate(f->zsp,Z_FINISH) ) == Z_OK ) {
        if ( f->zsp->avail_out == 0 ) {
/*
            Write the output
*/
#ifdef GZIPDEBUG
            MLOCK(ErrorMessageLock);
            MesPrint("%wWriting %l bytes at %10p",f->ziosize,&(f->POposition));
            MUNLOCK(ErrorMessageLock);
#endif
#ifdef ALLLOCK
            LOCK(f->pthreadslock);
#endif
            if ( f == AR.hidefile ) {
                UNLOCK(AS.inputslock);
            }
            SeekFile(f->handle,&(f->POposition),SEEK_SET);
            if ( WriteFile(f->handle,(UBYTE *)(f->ziobuffer),f->ziosize)
                        != f->ziosize ) {
                if ( f == AR.hidefile ) {
                    UNLOCK(AS.inputslock);
                }
#ifdef ALLLOCK
                UNLOCK(f->pthreadslock);
#endif
                MLOCK(ErrorMessageLock);
                MesPrint("%wWrite error during compressed sort. Disk full?");
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
            if ( f == AR.hidefile ) {
                UNLOCK(AS.inputslock);
            }
#ifdef ALLLOCK
            UNLOCK(f->pthreadslock);
```

```
#endif
            ADDPOS(f->filesize,f->ziosize);
            ADDPOS(f->POposition,f->ziosize);
#ifdef WITHPTHREADS
            if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
                if ( f->handle >= 0 ) SynchFile(f->handle);
            }
#endif
/*
            Reset the output
*/
            f->zsp->next_out  = f->ziobuffer;
            f->zsp->avail_out = f->ziosize;
            f->zsp->total_out = 0;
        }
    }
    if ( zerror == Z_STREAM_END ) {
/*
        Write the output
*/
#ifdef GZIPDEBUG
        MLOCK(ErrorMessageLock);
        MesPrint("%wWriting %l bytes at %10p",(LONG)(f->zsp->avail_out),&(f->POposition));
        MUNLOCK(ErrorMessageLock);
#endif
#ifdef ALLLOCK
        LOCK(f->pthreadslock);
#endif
        if ( f == AR.hidefile ) {
            LOCK(AS.inputslock);
        }
        SeekFile(f->handle,&(f->POposition),SEEK_SET);
        if ( WriteFile(f->handle,(UBYTE *)(f->ziobuffer),f->zsp->total_out)
                        != (LONG)(f->zsp->total_out) ) {
            if ( f == AR.hidefile ) {
                UNLOCK(AS.inputslock);
            }
#ifdef ALLLOCK
            UNLOCK(f->pthreadslock);
#endif
            MLOCK(ErrorMessageLock);
            MesPrint("%wWrite error during compressed sort. Disk full?");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        if ( f == AR.hidefile ) {
            LOCK(AS.inputslock);
        }
#ifdef ALLLOCK
        UNLOCK(f->pthreadslock);
#endif
        ADDPOS(f->filesize,f->zsp->total_out);
        ADDPOS(f->POposition,f->zsp->total_out);
#ifdef WITHPTHREADS
        if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
            if ( f->handle >= 0 ) SynchFile(f->handle);
        }
#endif
#ifdef GZIPDEBUG
        MLOCK(ErrorMessageLock);
        {  char *s = f->ziobuffer+f->zsp->total_out;
            MesPrint("%w   Last bytes written: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
        }
        MesPrint("%w     Perceived position in FlushOutputGZIP is %10p",&(f->POposition));
        MUNLOCK(ErrorMessageLock);
#endif
/*
            Reset the output
*/
        f->zsp->next_out  = f->ziobuffer;
        f->zsp->avail_out = f->ziosize;
        f->zsp->total_out = 0;
        if ( ( zerror = deflateEnd(f->zsp) ) == Z_OK ) return(0);
        MLOCK(ErrorMessageLock);
        if ( f->zsp->msg ) {
            MesPrint("%wError in finishing gzip handling of output: %s",f->zsp->msg);
        }
        else {
            MesPrint("%wError in finishing gzip handling of output.");
        }
        MUNLOCK(ErrorMessageLock);
    }
    else {
        MLOCK(ErrorMessageLock);
        MesPrint("%wError in gzip handling of output.");
        MUNLOCK(ErrorMessageLock);
    }
```

```
    return(-1);
}

/*
    #] FlushOutputGZIP :
    #[ SetupAllInputGZIP :

    Routine prepares all gzip input streams for a merge.

    Problem (29-may-2008): If we never use GZIP compression, this routine
    will still allocate the array space. This is an enormous amount!
    It places an effective restriction on the value of SortIOsize
*/

int SetupAllInputGZIP(SORTING *S)
{
    GETIDENTITY
    int i, NumberOpened = 0;
    z_streamp zsp;
/*
    This code was added 29-may-2008 by JV to prevent further processing if
    there is no compression at all (usually).
*/
    for ( i = 0; i < S->inNum; i++ ) {
        if ( S->fpincompressed[i] ) break;
    }
    if ( i >= S->inNum ) return(0);

    if ( S->zsparray == 0 ) {
        S->zsparray = (z_streamp)Malloc1(sizeof(z_stream)*S->MaxFpatches,"input zstreams");
        if ( S->zsparray == 0 ) {
            MLOCK(ErrorMessageLock);
            MesCall("SetupAllInputGZIP");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
/*
        We add 128 bytes in the hope that if it can happen that it goes
        outside the buffer during decompression, it does not do damage.
*/
        AN.ziobuffers = (Bytef *)Malloc1(S->MaxFpatches*(S->file.ziosize+128)*sizeof(Bytef),"input raw buffers");
/*
        This seems to be one of the really stupid errors:
        We allocate way too much space. Way way way too much.
        AN.ziobufnum  = (Bytef **)Malloc1(S->MaxFpatches*S->file.ziosize*sizeof(Bytef *),"input raw pointers");
*/
        AN.ziobufnum  = (Bytef **)Malloc1(S->MaxFpatches*sizeof(Bytef *),"input raw pointers");
        if ( AN.ziobuffers == 0 || AN.ziobufnum == 0 ) {
            MLOCK(ErrorMessageLock);
            MesCall("SetupAllInputGZIP");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        for ( i  = 0 ; i < S->MaxFpatches; i++ ) {
            AN.ziobufnum[i] = AN.ziobuffers + i * (S->file.ziosize+128);
        }
    }
    for ( i = 0; i < S->inNum; i++ ) {
#ifdef GZIPDEBUG
        MLOCK(ErrorMessageLock);
        MesPrint("%wPreparing z-stream %d with compression %d",i,S->fpincompressed[i]);
        MUNLOCK(ErrorMessageLock);
#endif
        if ( S->fpincompressed[i] ) {
            zsp = &(S->zsparray[i]);
/*
            1: Set the default fields:
*/
            zsp->zalloc = Z_NULL;
            zsp->zfree  = Z_NULL;
            zsp->opaque = Z_NULL;
/*
            2: Set the output space:
*/
            zsp->next_out  = Z_NULL;
            zsp->avail_out = 0;
            zsp->total_out = 0;
/*
            3: Set the input space temporarily:
*/
            zsp->next_in  = Z_NULL;
            zsp->avail_in = 0;
            zsp->total_in = 0;
/*
            4: Initiate the inflation
```

```
*/
            if ( inflateInit(zsp) != Z_OK ) {
                MLOCK(ErrorMessageLock);
                if ( zsp->msg ) MesPrint("%wError from inflateInit: %s",zsp->msg);
                else            MesPrint("%wError from inflateInit");
                MesCall("SetupAllInputGZIP");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            NumberOpened++;
        }
    }
    return(NumberOpened);
}

/*
    #] SetupAllInputGZIP :
    #[ FillInputGZIP :

    Routine is called when we need new input in the specified buffer.
    This buffer is used for the output and we keep reading and uncompressing
    input till either this buffer is full or the input stream is finished.
    The return value is the number of bytes in the buffer.
*/

LONG FillInputGZIP(FILEHANDLE *f, POSITION *position, UBYTE *buffer, LONG buffersize, int numstream)
{
    GETIDENTITY
    int zerror;
    LONG readsize, toread;
    SORTING *S = AT.SS;
    z_streamp zsp;
    POSITION pos;
    if ( S->fpincompressed[numstream] ) {
        zsp = &(S->zsparray[numstream]);
        zsp->next_out = (Bytef *)buffer;
        zsp->avail_out = buffersize;
        zsp->total_out = 0;
        if ( zsp->avail_in == 0 ) {
/*
            First loading of the input
*/
            if ( ISGEPOSINC(S->fPatchesStop[numstream],*position,f->ziosize) ) {
                toread = f->ziosize;
            }
            else {
                DIFPOS(pos,S->fPatchesStop[numstream],*position);
                toread = (LONG)(BASEPOSITION(pos));
            }
            if ( toread > 0 ) {
#ifdef GZIPDEBUG
                MLOCK(ErrorMessageLock);
                MesPrint("%w-+Reading %l bytes in stream %d at position %10p; stop at %10p",toread,numstream,position,&(S->fPatchesStop[numstream]));
                MUNLOCK(ErrorMessageLock);
#endif
#ifdef ALLLOCK
                LOCK(f->pthreadslock);
#endif
                SeekFile(f->handle,position,SEEK_SET);
                readsize = ReadFile(f->handle,(UBYTE *)(AN.ziobufnum[numstream]),toread);
                SeekFile(f->handle,position,SEEK_CUR);
#ifdef ALLLOCK
                UNLOCK(f->pthreadslock);
#endif
#ifdef GZIPDEBUG
                MLOCK(ErrorMessageLock);
                {  char *s = AN.ziobufnum[numstream]+readsize;
                    MesPrint("%w read: %l +Last bytes read: %d %d %d %d %d in %s, newpos = %10p",readsize,s[-5],s[-4],s[-3],s[-2],s[-1],f->name,position);
                }
                MUNLOCK(ErrorMessageLock);
#endif
                if ( readsize == 0 ) {
                    zsp->next_in  = AN.ziobufnum[numstream];
                    zsp->avail_in = f->ziosize;
                    zsp->total_in = 0;
                    return(zsp->total_out);
                }
                if ( readsize < 0 ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("%wFillInputGZIP: Read error during compressed sort.");
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                ADDPOS(f->filesize,readsize);
                ADDPOS(f->POposition,readsize);
```

```
/*
                Set the input
*/
                zsp->next_in  = AN.ziobufnum[numstream];
                zsp->avail_in = readsize;
                zsp->total_in = 0;
            }
        }
        while ( ( zerror = inflate(zsp,Z_NO_FLUSH) ) == Z_OK ) {
            if ( zsp->avail_out == 0 ) {
/*
                Finish
*/
                return((LONG)(zsp->total_out));
            }
            if ( zsp->avail_in == 0 ) {

                if ( ISEQUALPOS(S->fPatchesStop[numstream],*position) ) {
/*
                    We finished this stream. Try to terminate.
*/
                    if ( ( zerror = inflate(zsp,Z_SYNC_FLUSH) ) == Z_OK ) {
                        return((LONG)(zsp->total_out));
                    }
                    else
                        break;
/*
#ifdef GZIPDEBUG
                    MLOCK(ErrorMessageLock);
                    MesPrint("%wClosing stream %d",numstream);
#endif
                    readsize = zsp->total_out;
#ifdef GZIPDEBUG
                    if ( readsize > 0 ) {
                        WORD *s = (WORD *)(buffer+zsp->total_out);
                        MesPrint("%w   Last words: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
                    }
                    else {
                        MesPrint("%w No words");
                    }
                    MUNLOCK(ErrorMessageLock);
#endif
                    if ( ( zerror = inflateEnd(zsp) ) == Z_OK ) return(readsize);
                    break;
*/
                }
/*
                Read more input
*/
#ifdef GZIPDEBUG
                if ( numstream == 0 ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("%wWant to read in stream 0 at position %10p",position);
                    MUNLOCK(ErrorMessageLock);
                }
#endif
                if ( ISGEPOSINC(S->fPatchesStop[numstream],*position,f->ziosize) ) {
                    toread = f->ziosize;
                }
                else {
                    DIFPOS(pos,S->fPatchesStop[numstream],*position);
                    toread = (LONG)(BASEPOSITION(pos));
                }
#ifdef GZIPDEBUG
                MLOCK(ErrorMessageLock);
                MesPrint("%w--Reading %l bytes in stream %d at position
%10p",toread,numstream,position);
                MUNLOCK(ErrorMessageLock);
#endif
#ifdef ALLLOCK
                LOCK(f->pthreadslock);
#endif
                SeekFile(f->handle,position,SEEK_SET);
                readsize = ReadFile(f->handle,(UBYTE *)(AN.ziobufnum[numstream]),toread);
                SeekFile(f->handle,position,SEEK_CUR);
#ifdef ALLLOCK
                UNLOCK(f->pthreadslock);
#endif
#ifdef GZIPDEBUG
                MLOCK(ErrorMessageLock);
                {  char *s = AN.ziobufnum[numstream]+readsize;
                    MesPrint("%w   Last bytes read: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
                }
                MUNLOCK(ErrorMessageLock);
#endif
                if ( readsize == 0 ) {
                    zsp->next_in  = AN.ziobufnum[numstream];
```

```
                    zsp->avail_in = f->ziosize;
                    zsp->total_in = 0;
                    return(zsp->total_out);
                }
                if ( readsize < 0 ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("%wFillInputGZIP: Read error during compressed sort.");
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                ADDPOS(f->filesize,readsize);
                ADDPOS(f->POposition,readsize);
/*
                Reset the input
*/
                zsp->next_in  = AN.ziobufnum[numstream];
                zsp->avail_in = readsize;
                zsp->total_in = 0;
            }
            else {
                break;
            }
        }
#ifdef GZIPDEBUG
            MLOCK(ErrorMessageLock);
            MesPrint("%w zerror = %d in stream %d. At position %10p",zerror,numstream,position);
            MUNLOCK(ErrorMessageLock);
#endif
        if ( zerror == Z_STREAM_END ) {
/*
            Reset the input
*/
            zsp->next_in  = Z_NULL;
            zsp->avail_in = 0;
            zsp->total_in = 0;
/*
            Make the final call and finish
*/
#ifdef GZIPDEBUG
            MLOCK(ErrorMessageLock);
            MesPrint("%wClosing stream %d",numstream);
#endif
            readsize = zsp->total_out;
#ifdef GZIPDEBUG
            if ( readsize > 0 ) {
                WORD *s = (WORD *)(buffer+zsp->total_out);
                MesPrint("%w  -Last words: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
            }
            else {
                MesPrint("%w No words");
            }
            MUNLOCK(ErrorMessageLock);
#endif
            if ( inflateEnd(zsp) == Z_OK ) return(readsize);
        }

        MLOCK(ErrorMessageLock);
        MesPrint("%wFillInputGZIP: Error in gzip handling of input. zerror = %d",zerror);
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    else {
#ifdef GZIPDEBUG
        MLOCK(ErrorMessageLock);
        MesPrint("%w++Reading %l bytes at position %10p",buffersize,position);
        MUNLOCK(ErrorMessageLock);
#endif
#ifdef ALLLOCK
        LOCK(f->pthreadslock);
#endif
        SeekFile(f->handle,position,SEEK_SET);
        readsize = ReadFile(f->handle,buffer,buffersize);
        SeekFile(f->handle,position,SEEK_CUR);
#ifdef ALLLOCK
        UNLOCK(f->pthreadslock);
#endif
        if ( readsize < 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("%wFillInputGZIP: Read error during uncompressed sort.");
            MesPrint("%w++Reading %l bytes at position %10p",buffersize,position);
            MUNLOCK(ErrorMessageLock);
        }
        return(readsize);
    }
}

/*
```

```
00710     #] FillInputGZIP :
00711 */
00712 #endif
```

4.15 comtool.c File Reference

`#include "form3.h"`

Include dependency graph for comtool.c:

Functions

- int inicbufs (VOID)
- void finishcbuf (WORD num)
- void clearcbuf (WORD num)
- WORD ∗ DoubleCbuffer (int num, WORD ∗w, int par)
- WORD ∗ AddLHS (int num)
- WORD ∗ AddRHS (int num, int type)
- int AddNtoL (int n, WORD ∗array)
- int AddNtoC (int bufnum, int n, WORD ∗array, int par)
- int InsTree (int bufnum, int h)
- int FindTree (int bufnum, WORD ∗subexpr)
- void RedoTree (CBUF ∗C, int size)
- void ClearTree (int i)
- int IniFbuffer (WORD bufnum)
- LONG numcommute (WORD ∗terms, LONG ∗numterms)

4.15.1 Detailed Description

Utility routines for the compiler.

Definition in file comtool.c.

4.15.2 Function Documentation

4.15.2.1 inicbufs()

```
int inicbufs (
            VOID  )
```

Creates a new compiler buffer and returns its ID number.

Returns

The ID number for the new compiler buffer.

Definition at line 47 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::dimension, CbUf::lhs, CbUf::numdum, CbUf::NumTerms, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by AddRHS(), and StartVariables().

4.15.2.2 finishcbuf()

```
void finishcbuf (
            WORD num )
```

Frees a compiler buffer.

Parameters

num	The ID number for the buffer to be freed.

Definition at line 89 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::lhs, CbUf::NumTerms, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by PF_BroadcastCBuf().

4.15.2.3 clearcbuf()

```
void clearcbuf (
            WORD num )
```

Clears contents in a compiler buffer.

Parameters

num	The ID number for the buffer to be cleared.

Definition at line 116 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::Pointer, and CbUf::rhs.

4.15.2.4 DoubleCbuffer()

```
WORD * DoubleCbuffer (
            int num,
            WORD * w,
            int par )
```

Doubles a compiler buffer.

Parameters

num	The ID number for the buffer to be doubled.
w	The pointer to the end (exclusive) of the current buffer. The contents in the range of [cbuf[num].Buffer,w) will be kept.

Definition at line 143 of file comtool.c.

References CbUf::Buffer, CbUf::BufferSize, CbUf::lhs, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by AddNtoC(), AddNtoL(), and TestSub().

4.15.2.5 AddLHS()

```
WORD * AddLHS (
            int num )
```

Adds an LHS to a compiler buffer and returns the pointer to a buffer for the new LHS.

Parameters

num	The ID number for the buffer to get another LHS.

Definition at line 188 of file comtool.c.

References CbUf::lhs, and CbUf::Pointer.

Referenced by AddNtoL().

4.15.2.6 AddRHS()

```
WORD * AddRHS (
            int num,
            int type )
```

Adds an RHS to a compiler buffer and returns the pointer to a buffer for the new RHS.

Parameters

num	The ID number for the buffer to get another RHS.
type	If 0, the subexpression tree will be reallocated.

Definition at line 214 of file comtool.c.

References TaBlEs::buffers, TaBlEs::buffersfill, TaBlEs::bufferssize, TaBlEs::bufnum, CbUf::CanCommu, CbUf::dimension, inicbufs(), CbUf::numdum, CbUf::NumTerms, CbUf::Pointer, and CbUf::rhs.

Referenced by InsertArg(), StartVariables(), TestMatch(), and TestSub().

Here is the call graph for this function:

4.15.2.7 AddNtoL()

```
int AddNtoL (
            int n,
            WORD * array )
```

Adds an LHS with the given data to the current compiler buffer.

Parameters

n	The length of the data.
array	The data to be added.

Returns

0 if succeeds.

Definition at line 288 of file comtool.c.

References AddLHS(), DoubleCbuffer(), CbUf::Pointer, and CbUf::Top.

Here is the call graph for this function:

4.15.2.8 AddNtoC()

```
int AddNtoC (
            int bufnum,
            int n,
            WORD * array,
            int par )
```

Adds the given data to the last LHS/RHS in a compiler buffer.

Parameters

bufnum	The ID number for the buffer where the data will be added.
n	The length of the data.
array	The data to be added.

Returns

0 if succeeds.

Definition at line 317 of file comtool.c.

References DoubleCbuffer(), CbUf::Pointer, and CbUf::Top.

Referenced by InsertArg(), and StartVariables().

Here is the call graph for this function:

4.15.2.9 InsTree()

```
int InsTree (
            int bufnum,
            int h )
```

Definition at line 355 of file comtool.c.

4.15.2.10 FindTree()

```
int FindTree (
            int bufnum,
            WORD * subexpr )
```

Definition at line 533 of file comtool.c.

4.15.2.11 RedoTree()

```
void RedoTree (
            CBUF * C,
            int size )
```

Definition at line 569 of file comtool.c.

4.15.2.12 ClearTree()

```
void ClearTree (
            int i )
```

Definition at line 588 of file comtool.c.

4.15.2.13 IniFbuffer()

```
int IniFbuffer (
            WORD bufnum )
```

Initialize a factorization cache buffer. We set the size of the rhs and boomlijst buffers immediately to their final values.

Definition at line 614 of file comtool.c.

References tree::blnce, CbUf::boomlijst, CbUf::CanCommu, CbUf::dimension, tree::left, CbUf::numdum, CbUf::NumTerms, tree::parent, CbUf::rhs, tree::right, tree::usage, and tree::value.

4.15.2.14 numcommute()

```
LONG numcommute (
            WORD * terms,
            LONG * numterms )
```

Definition at line 659 of file comtool.c.

4.16 comtool.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes :
*/

#include "form3.h"

/*
    #] Includes :
    #[ inicbufs :
*/

int inicbufs(VOID)
{
    int i, num = AC.cbufList.num;
    CBUF *C = cbuf;
    for ( i = 0; i < num; i++, C++ ) {
        if ( C->Buffer == 0 ) break;
    }
    if ( i >= num ) C = (CBUF *)FromList(&AC.cbufList);
    else num = i;
    C->BufferSize = 2000;
    C->Buffer = (WORD *)Malloc1(C->BufferSize*sizeof(WORD),"compiler buffer-1");
    C->Pointer = C->Buffer;
    C->Top = C->Buffer + C->BufferSize;
    C->maxlhs = 10;
    C->lhs = (WORD **)Malloc1(C->maxlhs*sizeof(WORD *),"compiler buffer-2");
    C->numlhs = 0;
    C->mnumlhs = 0;
    C->maxrhs = 25;
    C->rhs = (WORD **)Malloc1(C->maxrhs*(sizeof(WORD *)+2*sizeof(LONG)+2*sizeof(WORD)),"compiler buffer-3");
    C->CanCommu = (LONG *)(C->rhs+C->maxrhs);
    C->NumTerms = C->CanCommu+C->maxrhs;
    C->numdum = (WORD *)(C->NumTerms+C->maxrhs);
    C->dimension = C->numdum + C->maxrhs;
    C->numrhs = 0;
    C->mnumrhs = 0;
    C->rhs[0] = C->rhs[1] = C->Pointer;
    C->boomlijst = 0;
    RedoTree(C,C->maxrhs);
    ClearTree(num);
    return(num);
}

/*
    #] inicbufs :
    #[ finishcbuf :
*/

void finishcbuf(WORD num)
{
    CBUF *C = cbuf+num;
    if ( C->Buffer ) M_free(C->Buffer,"compiler buffer-1");
    if ( C->rhs ) M_free(C->rhs,"compiler buffer-3");
    if ( C->lhs ) M_free(C->lhs,"compiler buffer-2");
```

```
    if ( C->boomlijst ) M_free(C->boomlijst,"boomlijst");
    C->Top = C->Pointer = C->Buffer = 0;
    C->rhs = C->lhs = 0;
    C->CanCommu = 0;
    C->NumTerms = 0;
    C->BufferSize = 0;
    C->boomlijst = 0;
    C->numlhs = C->numrhs = C->maxlhs = C->maxrhs = C->mnumlhs =
    C->mnumrhs = C->numtree = C->rootnum = C->MaxTreeSize = 0;
}

/*
    #] finishcbuf :
    #[ clearcbuf :
*/

void clearcbuf(WORD num)
{
    CBUF *C = cbuf+num;
    if ( C->boomlijst ) M_free(C->boomlijst,"boomlijst");
    C->Pointer = C->Buffer;
    C->numrhs = C->numlhs = 0;
    C->mnumlhs = 0;
    C->boomlijst = 0;
    C->mnumrhs = 0;
    C->rhs[0] = C->rhs[1] = C->Pointer;
    C->numtree = C->rootnum = C->MaxTreeSize = 0;
    RedoTree(C,C->maxrhs);
    ClearTree(num);
}

/*
    #] clearcbuf :
    #[ DoubleCbuffer :
*/

WORD *DoubleCbuffer(int num, WORD *w,int par)
{
    CBUF *C = cbuf + num;
    LONG newsize = C->BufferSize*2;
    WORD *newbuffer = (WORD *)Malloc1(newsize*sizeof(WORD),"compiler buffer-4");
    WORD *w1, *w2;
    LONG offset, j, i;
    DUMMYUSE(par)
/*
    MLOCK(ErrorMessageLock);
        MesPrint(" doubleCbuffer: par = %d",par);
    MUNLOCK(ErrorMessageLock);
*/
    w1 = C->Buffer; w2 = newbuffer;
    i = w - w1;
    j = i & 7;
    while ( --j >= 0 ) *w2++ = *w1++;
    i >>= 3;
    while ( --i >= 0 ) {
        *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++;
        *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++;
    }
    offset = newbuffer - C->Buffer;
    for ( i = 0; i <= C->numlhs; i++ ) C->lhs[i] += offset;
    for ( i = 1; i <= C->numrhs; i++ ) C->rhs[i] += offset;
    w1 = C->Buffer;
    C->Pointer += offset;
    C->Top = newbuffer + newsize;
    C->BufferSize = newsize;
    C->Buffer = newbuffer;
    M_free(w1,"DoubleCbuffer");
    return(w2);
}

/*
    #] DoubleCbuffer :
    #[ AddLHS :
*/

WORD *AddLHS(int num)
{
    CBUF *C = cbuf + num;
    C->numlhs++;
    if ( C->numlhs >= (C->maxlhs-2) ) {
        WORD ***ppp = &(C->lhs); /* to avoid compiler warning */
        if ( DoubleList((VOID ***)ppp,&(C->maxlhs),sizeof(WORD *),
        "statement lists") ) Terminate(-1);
    }
    C->lhs[C->numlhs] = C->Pointer;
    C->lhs[C->numlhs+1] = 0;
    return(C->Pointer);
```

```
}

/*
    #] AddLHS :
    #[ AddRHS :
*/

WORD *AddRHS(int num, int type)
{
    LONG fullsize, *lold, newsize;
    int i;
    WORD **old, *wold;
    CBUF *C;
restart:;
    C = cbuf + num;
    if ( C->numrhs >= (C->maxrhs-2) ) {
        if ( C->maxrhs == 0 ) newsize = 100;
        else                  newsize = C->maxrhs * 2;
        if ( newsize > MAXCOMBUFRHS ) newsize = MAXCOMBUFRHS;
        if ( newsize == C->maxrhs ) {
            if ( AC.tablefilling ) {
                TABLES T = functions[AC.tablefilling].tabl;
/*
                We add a compiler buffer, change a few settings and continue.
*/
                if ( T->buffersfill >= T->bufferssize ) {
                    int new1 = 2*T->bufferssize;
                    WORD *nbufs = (WORD *)Malloc1(new1*sizeof(WORD),"Table compile buffers");
                    for ( i = 0; i < T->buffersfill; i++ )
                            nbufs[i] = T->buffers[i];
                    for ( ; i < new1; i++ ) nbufs[i] = 0;
                    M_free(T->buffers,"Table compile buffers");
                    T->buffers = nbufs;
                    T->bufferssize = new1;
                }
                T->buffers[T->buffersfill++] = T->bufnum = inicbufs();
                AC.cbufnum = num = T->bufnum;
                goto restart;
            }
            else {
                MesPrint("@Compiler buffer overflow. Try to make modules smaller");
                Terminate(-1);
            }
        }
        old = C->rhs;
        fullsize = newsize * (sizeof(WORD *) + 2*sizeof(LONG) + 2*sizeof(WORD));
        C->rhs = (WORD **)Malloc1(fullsize,"subexpression lists");
        for ( i = 0; i < C->maxrhs; i++ ) C->rhs[i] = old[i];
        lold = C->CanCommu; C->CanCommu = (LONG *)(C->rhs+newsize);
        for ( i = 0; i < C->maxrhs; i++ ) C->CanCommu[i] = lold[i];
        lold = C->NumTerms; C->NumTerms = (LONG *)(C->rhs+2*newsize);
        for ( i = 0; i < C->maxrhs; i++ ) C->NumTerms[i] = lold[i];
        wold = C->numdum; C->numdum = (WORD *)(C->NumTerms+newsize);
        for ( i = 0; i < C->maxrhs; i++ ) C->numdum[i] = wold[i];
        wold = C->dimension; C->dimension = (WORD *)(C->numdum+newsize);
        for ( i = 0; i < C->maxrhs; i++ ) C->dimension[i] = wold[i];
        if ( old ) M_free(old,"subexpression lists");
        C->maxrhs = newsize;
        if ( type == 0 ) RedoTree(C,C->maxrhs);
    }
    C->numrhs++;
    C->CanCommu[C->numrhs] = 0;
    C->NumTerms[C->numrhs] = 0;
    C->numdum[C->numrhs] = 0;
    C->dimension[C->numrhs] = 0;
    C->rhs[C->numrhs] = C->Pointer;
    return(C->Pointer);
}

/*
    #] AddRHS :
    #[ AddNtoL :
*/

int AddNtoL(int n, WORD *array)
{
    int i;
    CBUF *C = cbuf+AC.cbufnum;
#ifdef COMPBUFDEBUG
    MesPrint("LH: %a",n,array);
#endif
    AddLHS(AC.cbufnum);
    while ( C->Pointer+n >= C->Top ) DoubleCbuffer(AC.cbufnum,C->Pointer,1);
    for ( i = 0; i < n; i++ ) *(C->Pointer)++ = *array++;
    return(0);
}
```

```
/*
    #] AddNtoL :
    #[ AddNtoC :

    Commentary: added the bufnum on 14-sep-2010 to make the whole a bit
    more flexible (JV). Still to do with AddNtoL.
*/

int AddNtoC(int bufnum, int n, WORD *array,int par)
{
    int i;
    WORD *w;
    CBUF *C = cbuf+bufnum;
#ifdef COMPBUFDEBUG
    MesPrint("RH: %a",n,array);
#endif
    while ( C->Pointer+n+1 >= C->Top ) DoubleCbuffer(bufnum,C->Pointer,50+par);
    w = C->Pointer;
    for ( i = 0; i < n; i++ ) *w++ = *array++;
    C->Pointer = w;
    return(0);
}

/*
    #] AddNtoC :
    #[ InsTree :

    Routines for balanced tree searching and insertion.
    Compared to Knuth we have a parent link. This minimizes the
    number of compares. That is better for anything that is more
    complicated than just single numbers.
    There are no provisions for removing elements from the tree.
    The routines are:
    void RedoTree(size) Re-allocates the tree space. There will
                        be MaxTreeSize = size elements.
    void ClearTree()    Prunes the tree down to the root element.
    int InsTree(int,int)Searches for the requested element. If not found it
                        will allocate a new element, balance the tree if
                        necessary and return the called number.
                        If it was in the tree, it returns the tree 'value'.

    Commentary: added the bufnum on 14-sep-2010 to make the whole a bit
    more flexible (JV).
*/
static COMPTREE comptreezero = {0,0,0,0,0,0};

int InsTree(int bufnum, int h)
{
    CBUF *C = cbuf + bufnum;
    COMPTREE *boomlijst = C->boomlijst, *q = boomlijst + C->rootnum, *p, *s;
    WORD *v1, *v2, *v3;
    int ip, iq, is;

    if ( C->numtree + 1 >= C->MaxTreeSize ) {
        if ( C->MaxTreeSize == 0 ) {
            COMPTREE *root;
            C->MaxTreeSize = 125;
            C->boomlijst = (COMPTREE *)Malloc1((C->MaxTreeSize+1)*sizeof(COMPTREE),
                "ClearInsTree");
            root = C->boomlijst;
            C->numtree = 0;
            C->rootnum = 0;
            root->left = -1;
            root->right = -1;
            root->parent = -1;
            root->blnce = 0;
            root->value = -1;
            root->usage = 0;
            for ( ip = 1; ip < C->MaxTreeSize; ip++ ) { C->boomlijst[ip] = comptreezero; }
        }
        else {
            is = C->MaxTreeSize * 2;
            s  = (COMPTREE *)Malloc1((is+1)*sizeof(COMPTREE),"InsTree");
            for ( ip = 0; ip < C->MaxTreeSize; ip++ ) { s[ip] = C->boomlijst[ip]; }
            for ( ip = C->MaxTreeSize; ip <= is; ip++ ) { s[ip] = comptreezero; }
            if ( C->boomlijst ) M_free(C->boomlijst,"InsTree");
            C->boomlijst = s;
            C->MaxTreeSize = is;
        }
        boomlijst = C->boomlijst;
        q = boomlijst + C->rootnum;
    }

    if ( q->right == -1 ) { /* First element */
        C->numtree++;
        s = boomlijst+C->numtree;
        q->right = C->numtree;
```

```
        s->parent = C->rootnum;
        s->left = s->right = -1;
        s->blnce = 0;
        s->value = h;
        s->usage = 1;
        return(h);
    }
    ip = q->right;
    while ( ip >= 0 ) {
        p = boomlijst + ip;
        v1 = C->rhs[p->value]; v2 = v3 = C->rhs[h];
        while ( *v3 ) v3 += *v3;  /* find the 0 that indicates end-of-expr */
        while ( *v1 == *v2 && v2 < v3 ) { v1++; v2++; }
        if ( *v1 > *v2 ) {
            iq = p->right;
            if ( iq >= 0 ) { ip = iq; }
            else {
                C->numtree++;
                is = C->numtree;
                p->right = is;
                s = boomlijst + is;
                s->parent = ip; s->left = s->right = -1;
                s->blnce = 0;   s->value = h; s->usage = 1;
                p->blnce++;
                if ( p->blnce == 0 ) return(h);
                goto balance;
            }
        }
        else if ( *v1 < *v2 ) {
            iq = p->left;
            if ( iq >= 0 ) { ip = iq; }
            else {
                C->numtree++;
                is = C->numtree;
                s = boomlijst+is;
                p->left = is;
                s->parent = ip; s->left = s->right = -1;
                s->blnce = 0;   s->value = h; s->usage = 1;
                p->blnce--;
                if ( p->blnce == 0 ) return(h);
                goto balance;
            }
        }
        else {
            p->usage++;
            return(p->value);
        }
    }
    MesPrint("We vallen uit de boom!");
    Terminate(-1);
    return(h);
balance:;
    for (;;) {
        p = boomlijst + ip;
        iq = p->parent;
        if ( iq == C->rootnum ) break;
        q = boomlijst + iq;
        if ( ip == q->left ) q->blnce--;
        else                 q->blnce++;
        if ( q->blnce == 0 ) break;
        if ( q->blnce == -2 ) {
            if ( p->blnce == -1 ) { /* single rotation */
                q->left = p->right;
                p->right = iq;
                p->parent = q->parent;
                q->parent = ip;
                if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
                else                                   boomlijst[p->parent].right = ip;
                if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
                q->blnce = p->blnce = 0;
            }
            else {  /* double rotation */
                s = boomlijst + is;
                q->left = s->right;
                p->right = s->left;
                s->right = iq;
                s->left = ip;
                if ( p->right >= 0 ) boomlijst[p->right].parent = ip;
                if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
                s->parent = q->parent;
                q->parent = is;
                p->parent = is;
                if ( boomlijst[s->parent].left == iq )
                       boomlijst[s->parent].left = is;
                else   boomlijst[s->parent].right = is;
                if ( s->blnce > 0 ) { q->blnce = s->blnce = 0; p->blnce = -1; }
                else if ( s->blnce < 0 ) { p->blnce = s->blnce = 0; q->blnce = 1; }
```

```
                else { p->blnce = s->blnce = q->blnce = 0; }
            }
            break;
        }
        else if ( q->blnce == 2 ) {
            if ( p->blnce == 1 ) {  /* single rotation */
                q->right = p->left;
                p->left = iq;
                p->parent = q->parent;
                q->parent = ip;
                if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
                else                                   boomlijst[p->parent].right = ip;
                if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
                q->blnce = p->blnce = 0;
            }
            else {  /* double rotation */
                s = boomlijst + is;
                q->right = s->left;
                p->left = s->right;
                s->left = iq;
                s->right = ip;
                if ( p->left >= 0 ) boomlijst[p->left].parent = ip;
                if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
                s->parent = q->parent;
                q->parent = is;
                p->parent = is;
                if ( boomlijst[s->parent].left == iq ) boomlijst[s->parent].left = is;
                else                                   boomlijst[s->parent].right = is;
                if ( s->blnce < 0 ) { q->blnce = s->blnce = 0; p->blnce = 1; }
                else if ( s->blnce > 0 ) { p->blnce = s->blnce = 0; q->blnce = -1; }
                else { p->blnce = s->blnce = q->blnce = 0; }
            }
            break;
        }
        is = ip; ip = iq;
    }
    return(h);
}

/*
    #] InsTree :
    #[ FindTree :

    Routines for balanced tree searching.
    Is like InsTree but without the insertions.
    Returns -1 if the element is not in the tree.
    The advantage of this routine over InsTree is that this routine
    can be run in parallel.
*/

int FindTree(int bufnum, WORD *subexpr)
{
    CBUF *C = cbuf + bufnum;
    COMPTREE *boomlijst = C->boomlijst, *q = boomlijst + C->rootnum, *p;
    WORD *v1, *v2, *v3;
    int ip, iq;

    ip = q->right;
    while ( ip >= 0 ) {
        p = boomlijst + ip;
        v1 = C->rhs[p->value]; v2 = v3 = subexpr;
        while ( *v3 ) v3 += *v3;  /* find the 0 that indicates end-of-expr */
        while ( *v1 == *v2 && v2 < v3 ) { v1++; v2++; }
        if ( *v1 > *v2 ) {
            iq = p->right;
            if ( iq >= 0 ) { ip = iq; }
            else { return(-1); }
        }
        else if ( *v1 < *v2 ) {
            iq = p->left;
            if ( iq >= 0 ) { ip = iq; }
            else { return(-1); }
        }
        else {
            p->usage++;
            return(p->value);
        }
    }
    return(-1);
}

/*
    #] FindTree :
    #[ RedoTree :
*/

void RedoTree(CBUF *C, int size)
```

```
{
    COMPTREE *newboomlijst;
    int i;
    newboomlijst = (COMPTREE *)Malloc1((size+1)*sizeof(COMPTREE),"newboomlijst");
    if ( C->boomlijst ) {
        if ( C->MaxTreeSize > size ) C->MaxTreeSize = size;
        for ( i = 0; i < C->MaxTreeSize; i++ ) newboomlijst[i] = C->boomlijst[i];
        M_free(C->boomlijst,"boomlijst");
    }
    C->boomlijst = newboomlijst;
    C->MaxTreeSize = size;
}

/*
    #] RedoTree :
    #[ ClearTree :
*/

void ClearTree(int i)
{
    CBUF *C = cbuf + i;
    COMPTREE *root = C->boomlijst;
    if ( root ) {
        C->numtree = 0;
        C->rootnum = 0;
        root->left = -1;
        root->right = -1;
        root->parent = -1;
        root->blnce = 0;
        root->value = -1;
        root->usage = 0;
    }
}

/*
    #] ClearTree :
    #[ IniFbuffer :
*/
int IniFbuffer(WORD bufnum)
{
    CBUF *C = cbuf + bufnum;
    COMPTREE *root;
    int i;
    LONG fullsize;
    C->maxrhs = AM.fbuffersize;
    C->MaxTreeSize = AM.fbuffersize;

    /*
     * Note that bufnum is a return value of inicbufs(). So C has been already
     * initialized. (TU 20 Dec 2011)
     */
    if ( C->boomlijst ) M_free(C->boomlijst, "IniFbuffer-tree");
    if ( C->rhs ) M_free(C->rhs, "IniFbuffer-rhs");

    C->boomlijst = (COMPTREE *)Malloc1((C->MaxTreeSize+1)*sizeof(COMPTREE),"IniFbuffer-tree");
    root = C->boomlijst;
    C->numtree = 0;
    C->rootnum = 0;
    root->left = -1;
    root->right = -1;
    root->parent = -1;
    root->blnce = 0;
    root->value = -1;
    root->usage = 0;
    for ( i = 1; i < C->MaxTreeSize; i++ ) { C->boomlijst[i] = comptreezero; }

    fullsize = (C->maxrhs+1) * (sizeof(WORD *) + 2*sizeof(LONG) + 2*sizeof(WORD));
    C->rhs = (WORD **)Malloc1(fullsize,"IniFbuffer-rhs");
    C->CanCommu = (LONG *)(C->rhs+C->maxrhs);
    C->NumTerms = (LONG *)(C->rhs+2*C->maxrhs);
    C->numdum = (WORD *)(C->NumTerms+C->maxrhs);
    C->dimension = (WORD *)(C->numdum+C->maxrhs);

    return(0);
}

/*
    #] IniFbuffer :
    #[ numcommute :

    Returns the number of non-commuting terms in the expression
*/

LONG numcommute(WORD *terms, LONG *numterms)
{
    LONG num = 0;
    WORD *t, *m;
```

```
    *numterms = 0;
    while ( *terms ) {
        *numterms += 1;
        t = terms + 1;
        GETSTOP(terms,m);
        while ( t < m ) {
            if ( *t >= FUNCTION ) {
                if ( functions[*t-FUNCTION].commute ) { num++; break; }
            }
            t += t[1];
        }
        terms = terms + *terms;
    }
    return(num);
}

/*
    #] numcommute :
*/
```

4.17 comtool.h File Reference

`#include "form3.h"`

Include dependency graph for comtool.h:

This graph shows which files directly or indirectly include this file:

4.17.1 Detailed Description

Utility routines for the compiler.

Definition in file comtool.h.

4.18 comtool.h

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
#ifndef FORM_COMTOOL_H_
#define FORM_COMTOOL_H_
/*
    #[ Includes :
*/

#include "form3.h"

/*
    #] Includes :
    #[ Inline functions :
*/

static inline void SkipSpaces(UBYTE **s)
{
    const char *p = (const char *)*s;
    while ( *p == ' ' || *p == ',' || *p == '\t' ) p++;
    *s = (UBYTE *)p;
}

static inline int ConsumeOption(UBYTE **s, const char *opt)
{
    const char *p = (const char *)*s;
    while ( *p && *opt && tolower(*p) == tolower(*opt) ) {
        p++;
        opt++;
    }
    /* Check if `opt' ended. */
    if ( !*opt ) {
        /* Check if `*p' is a word boundary. */
        if ( !*p || !(FG.cTable[(unsigned char)*p] == 0 ||
                      FG.cTable[(unsigned char)*p] == 1 || *p == '_' ||
                      *p == '$') ) {
            /* Consume the option. Skip the trailing spaces. */
            *s = (UBYTE *)p;
            SkipSpaces(s);
            return(1);
        }
    }
    return(0);
}

/*
    #] Inline functions :
*/
#endif  /* FORM_COMTOOL_H_ */
```

4.19 declare.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define MaX(x, y) ((x) > (y) ? (x): (y))
- #define MiN(x, y) ((x) < (y) ? (x): (y))
- #define ABS(x) ((x) < 0 ? -(x): (x))
- #define SGN(x) ((x) > 0 ? 1 : (x) < 0 ? -1 : 0)
- #define REDLENG(x) ((((x)<0)?((x)+1):((x)-1))>>1)
- #define INCLENG(x) (((x)<0)?(((x)<<1)-1):(((x)<<1)+1))
- #define GETCOEF(x, y) x += *x;y = x[-1];x -= ABS(y);y=REDLENG(y)
- #define GETSTOP(x, y) y=x+(*x)-1;y -= ABS(*y)-1
- #define StuffAdd(x, y) (((x)<0?-1:1)*(y)+((y)<0?-1:1)*(x))
- #define TOKENTOLINE(x, y)
- #define UngetFromStream(stream, c) ((stream)->nextchar[(stream)->isnextchar++]=c)
- #define AddLineFeed(s, n) { (s)[(n)++] = LINEFEED; }
- #define TryRecover(x) Terminate(-1)
- #define UngetChar(c) { pushbackchar = c; }
- #define ParseNumber(x, s) {(x)=0;while(*(s)>='0'&&*(s)<='9')(x)=10*(x)+*(s)++ -'0';}
- #define ParseSign(sgn, s)
- #define ParseSignedNumber(x, s)
- #define NCOPY(s, t, n) while (--n >= 0) *s++ = *t++;
- #define NCOPYI(s, t, n) while (--n >= 0) *s++ = *t++;
- #define NCOPYB(s, t, n) while (--n >= 0) *s++ = *t++;
- #define NCOPYI32(s, t, n) while (--n >= 0) *s++ = *t++;
- #define WCOPY(s, t, n) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; while (--nn >= 0) *ss++=*tt++; }
- #define NeedNumber(x, s, err)
- #define SKIPBLANKS(s) { while (*(s) == ' ' || *(s) == '\t') (s)++; }
- #define FLUSHCONSOLE if (AP.InOutBuf > 0) CharOut(LINEFEED)
- #define SKIPBRA1(s)
- #define SKIPBRA2(s)
- #define SKIPBRA3(s)
- #define SKIPBRA4(s)
- #define SKIPBRA5(s)
- #define CYCLE1(t, a, i) {t iX=*a; WORD jX; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
- #define AddToCB(c, wx)
- #define EXCHINOUT
- #define BACKINOUT
- #define CopyArg(to, from)
- #define FILLARG(w)
- #define COPYARG(w, t)
- #define ZEROARG(w)
- #define FILLFUN(w)
- #define COPYFUN(w, t)
- #define COPYFUN3(w, t)
- #define FILLFUN3(w)

- #define FILLSUB(w)
- #define COPYSUB(w, ww)
- #define FILLEXPR(w)
- #define NEXTARG(x) if(∗x>0) x += ∗x; else if(∗x <= -FUNCTION)x++; else x += 2;
- #define COPY1ARG(s1, t1)
- #define ZeroFillRange(w, begin, end)
- #define TABLESIZE(a, b) (((WORD)sizeof(a))/((WORD)sizeof(b)))
- #define WORDDIF(x, y) (WORD)(x-y)
- #define wsizeof(a) ((WORD)sizeof(a))
- #define VARNAME(type, num) (AC.varnames->namebuffer+type[num].name)
- #define DOLLARNAME(type, num) (AC.dollarnames->namebuffer+type[num].name)
- #define EXPRNAME(num) (AC.exprnames->namebuffer+Expressions[num].name)
- #define PREV(x) prevorder?prevorder:x
- #define SETERROR(x) { Terminate(-1); return(-1); }
- #define DUMMYUSE(x) (void)(x);
- #define ADDPOS(pp, x) (pp).p1 = ((pp).p1+(LONG)(x))
- #define SETBASELENGTH(ss, x) (ss).p1 = (LONG)(x)
- #define SETBASEPOSITION(pp, x) (pp).p1 = (LONG)(x)
- #define ISEQUALPOSINC(pp1, pp2, x) ((pp1).p1 == ((pp2).p1+(LONG)(x)))
- #define ISGEPOSINC(pp1, pp2, x) ((pp1).p1 >= ((pp2).p1+(LONG)(x)))
- #define DIVPOS(pp, n) ((pp).p1/(LONG)(n))
- #define MULPOS(pp, n) (pp).p1 ∗= (LONG)(n)
- #define DIFPOS(ss, pp1, pp2) (ss).p1 = ((pp1).p1-(pp2).p1)
- #define DIFBASE(pp1, pp2) ((pp1).p1-(pp2).p1)
- #define ADD2POS(pp1, pp2) (pp1).p1 += (pp2).p1
- #define PUTZERO(pp) (pp).p1 = 0
- #define BASEPOSITION(pp) ((pp).p1)
- #define SETSTARTPOS(pp) (pp).p1 = -2
- #define NOTSTARTPOS(pp) ((pp).p1 > -2)
- #define ISMINPOS(pp) ((pp).p1 == -1)
- #define ISEQUALPOS(pp1, pp2) ((pp1).p1 == (pp2).p1)
- #define ISNOTEQUALPOS(pp1, pp2) ((pp1).p1 != (pp2).p1)
- #define ISLESSPOS(pp1, pp2) ((pp1).p1 < (pp2).p1)
- #define ISGEPOS(pp1, pp2) ((pp1).p1 >= (pp2).p1)
- #define ISNOTZEROPOS(pp) ((pp).p1 != 0)
- #define ISZEROPOS(pp) ((pp).p1 == 0)
- #define ISPOSPOS(pp) ((pp).p1 > 0)
- #define ISNEGPOS(pp) ((pp).p1 < 0)
- #define TOLONG(x) ((LONG)(x))
- #define Add2Com(x) { WORD cod[2]; cod[0] = x; cod[1] = 2; AddNtoL(2,cod); }
- #define Add3Com(x1, x2) { WORD cod[3]; cod[0] = x1; cod[1] = 3; cod[2] = x2; AddNtoL(3,cod); }
- #define Add4Com(x1, x2, x3)
- #define Add5Com(x1, x2, x3, x4)
- #define WantAddPointers(x)
- #define WantAddLongs(x)
- #define WantAddPositions(x)
- #define FORM_INLINE inline
- #define MEMORYMACROS
- #define TermMalloc(x) ((AT.TermMemTop <= 0) ? TermMallocAddMemory(BHEAD0), AT.TermMemHeap[--AT.TermMemTop]: AT.TermMemHeap[--AT.TermMemTop])
- #define NumberMalloc(x) ((AT.NumberMemTop <= 0) ? NumberMallocAddMemory(BHEAD0), AT.↩NumberMemHeap[--AT.NumberMemTop]: AT.NumberMemHeap[--AT.NumberMemTop])
- #define CacheNumberMalloc(x) ((AT.CacheNumberMemTop <= 0) ? CacheNumberMallocAdd↩Memory(BHEAD0), AT.CacheNumberMemHeap[--AT.CacheNumberMemTop]: AT.CacheNumberMemHeap[--AT.CacheNumberMemTop])

- #define TermFree(TermMem, x) AT.TermMemHeap[AT.TermMemTop++] = (WORD ∗)(TermMem)
- #define NumberFree(NumberMem, x) AT.NumberMemHeap[AT.NumberMemTop++] = (UWORD ∗)(Number↩ Mem)
- #define CacheNumberFree(NumberMem, x) AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = (UWORD ∗)(NumberMem)
- #define NestingChecksum() (AC.IfLevel + AC.RepLevel + AC.arglevel + AC.insidelevel + AC.termlevel + AC.↩ inexprlevel + AC.dolooplevel)
- #define MesNesting() MesPrint("&Illegal nesting of if, repeat, argument, inside, term, inexpression and do")
- #define MarkPolyRatFunDirty(T)
- #define PUSHPREASSIGNLEVEL
- #define POPPREASSIGNLEVEL
- #define EXTERNLOCK(x)
- #define INILOCK(x)
- #define LOCK(x)
- #define UNLOCK(x)
- #define EXTERNRWLOCK(x)
- #define INIRWLOCK(x)
- #define RWLOCKR(x)
- #define RWLOCKW(x)
- #define UNRWLOCK(x)
- #define MLOCK(x)
- #define MUNLOCK(x)
- #define GETIDENTITY
- #define GETBIDENTITY
- #define M_alloc(x) malloc((size_t)(x))
- #define CompareTerms ((COMPARE)AR.CompareRoutine)
- #define FiniShuffle ((FINISHUFFLE)AN.SHvar.finishuf)
- #define DoShtuffle ((DO_UFFLE)AN.SHvar.do_uffle)

Typedefs

- typedef int(∗ WRITEBUFTOEXTCHANNEL) (char ∗, size_t)
- typedef int(∗ GETCFROMEXTCHANNEL) (VOID)
- typedef int(∗ SETTERMINATORFOREXTERNALCHANNEL) (char ∗)
- typedef int(∗ SETKILLMODEFOREXTERNALCHANNEL) (int, int)
- typedef LONG(∗ WRITEFILE) (int, UBYTE ∗, LONG)
- typedef WORD(∗ COMPARE) (PHEAD WORD ∗, WORD ∗, WORD)
- typedef WORD(∗ GETTERM) (PHEAD WORD ∗)
- typedef WORD(∗ FINISHUFFLE) (PHEAD WORD ∗)
- typedef WORD(∗ DO_UFFLE) (PHEAD WORD ∗, WORD, WORD, WORD)

Functions

- VOID TELLFILE (int, POSITION ∗)
- VOID StartVariables ()
- VOID **setSignalHandlers** (VOID)
- UBYTE ∗ CodeToLine (WORD, UBYTE ∗)
- UBYTE ∗ AddArrayIndex (WORD, UBYTE ∗)
- INDEXENTRY ∗ FindInIndex (WORD, FILEDATA ∗, WORD, WORD)
- INDEXENTRY ∗ NextFileIndex (POSITION ∗)
- WORD ∗ PasteTerm (PHEAD WORD, WORD ∗, WORD ∗, WORD, WORD)
- UBYTE ∗ StrCopy (UBYTE ∗, UBYTE ∗)
- UBYTE ∗ WrtPower (UBYTE ∗, WORD)

- WORD AccumGCD (PHEAD UWORD *, WORD *, UWORD *, WORD)
- VOID AddArgs (PHEAD WORD *, WORD *, WORD *)
- WORD AddCoef (PHEAD WORD **, WORD **)
- WORD AddLong (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- WORD AddPLon (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- WORD AddPoly (PHEAD WORD **, WORD **)
- WORD AddRat (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- VOID AddToLine (UBYTE *)
- WORD AddWild (PHEAD WORD, WORD, WORD)
- WORD BigLong (UWORD *, WORD, UWORD *, WORD)
- WORD BinomGen (PHEAD WORD *, WORD, WORD **, WORD, WORD, WORD, WORD, WORD, UWORD *, WORD)
- WORD CheckWild (PHEAD WORD, WORD, WORD, WORD *)
- WORD Chisholm (PHEAD WORD *, WORD)
- WORD CleanExpr (WORD)
- VOID CleanUp (WORD)
- VOID ClearWild (PHEAD0)
- WORD CompareFunctions (WORD *, WORD *)
- WORD Commute (WORD *, WORD *)
- WORD DetCommu (WORD *)
- WORD DoesCommu (WORD *)
- int CompArg (WORD *, WORD *)
- WORD CompCoef (WORD *, WORD *)
- WORD CompGroup (PHEAD WORD, WORD **, WORD *, WORD *, WORD)
- WORD Compare1 (PHEAD WORD *, WORD *, WORD)
- WORD CountDo (WORD *, WORD *)
- WORD CountFun (WORD *, WORD *)
- WORD DimensionSubterm (WORD *)
- WORD DimensionTerm (WORD *)
- WORD DimensionExpression (PHEAD WORD *)
- WORD Deferred (PHEAD WORD *, WORD)
- WORD DeleteStore (WORD)
- WORD DetCurDum (PHEAD WORD *)
- VOID DetVars (WORD *, WORD)
- WORD Distribute (DISTRIBUTE *, WORD)
- WORD DivLong (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *, UWORD *, WORD *)
- WORD DivRat (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- WORD Divvy (PHEAD UWORD *, WORD *, UWORD *, WORD)
- WORD DoDelta (WORD *)
- WORD DoDelta3 (PHEAD WORD *, WORD)
- WORD TestPartitions (PHEAD WORD *, PARTI *)
- WORD DoPartitions (PHEAD WORD *, WORD)
- WORD DoTableExpansion (WORD *, WORD)
- WORD DoDistrib (PHEAD WORD *, WORD)
- WORD DoShuffle (PHEAD WORD *, WORD, WORD, WORD)
- WORD DoPermutations (PHEAD WORD *, WORD)
- int Shuffle (PHEAD WORD *, WORD *, WORD *)
- int FinishShuffle (PHEAD WORD *)
- WORD DoStuffle (PHEAD WORD *, WORD, WORD, WORD)
- int Stuffle (PHEAD WORD *, WORD *, WORD *)
- int FinishStuffle (PHEAD WORD *)
- WORD * StuffRootAdd (WORD *, WORD *, WORD *)
- WORD TestUse (WORD *, WORD)
- DBASE * FindTB (UBYTE *)
- int CheckTableDeclarations (DBASE *)

- WORD Apply (WORD ∗, WORD)
- int ApplyExec (WORD ∗, int, WORD)
- WORD ApplyReset (WORD)
- WORD **TableReset** (VOID)
- VOID ReWorkT (WORD ∗, WORD ∗, WORD)
- WORD GetIfDollarNum (WORD ∗, WORD ∗)
- int FindVar (WORD ∗, WORD ∗)
- WORD DoIfStatement (PHEAD WORD ∗, WORD ∗)
- WORD DoOnePow (PHEAD WORD ∗, WORD, WORD, WORD ∗, WORD ∗, WORD, WORD ∗)
- void DoRevert (WORD ∗, WORD ∗)
- WORD DoSumF1 (PHEAD WORD ∗, WORD ∗, WORD, WORD)
- WORD DoSumF2 (PHEAD WORD ∗, WORD ∗, WORD, WORD)
- WORD DoTheta (PHEAD WORD ∗)
- LONG EndSort (PHEAD WORD ∗, int)
- WORD EntVar (WORD, UBYTE ∗, WORD, WORD, WORD, WORD)
- WORD EpfCon (PHEAD WORD ∗, WORD ∗, WORD, WORD)
- WORD EpfFind (PHEAD WORD ∗, WORD ∗)
- WORD EpfGen (WORD, WORD ∗, WORD ∗, WORD ∗, WORD)
- WORD EqualArg (WORD ∗, WORD, WORD)
- WORD **Evaluate** (UBYTE ∗∗)
- int Factorial (PHEAD WORD, UWORD ∗, WORD ∗)
- int Bernoulli (WORD, UWORD ∗, WORD ∗)
- int FactorIn (PHEAD WORD ∗, WORD)
- int FactorInExpr (PHEAD WORD ∗, WORD)
- WORD FindAll (PHEAD WORD ∗, WORD ∗, WORD, WORD ∗)
- WORD FindMulti (PHEAD WORD ∗, WORD ∗)
- WORD FindOnce (PHEAD WORD ∗, WORD ∗)
- WORD FindOnly (PHEAD WORD ∗, WORD ∗)
- WORD FindRest (PHEAD WORD ∗, WORD ∗)
- WORD **FindSpecial** (WORD ∗)
- WORD FindrNumber (WORD, VARRENUM ∗)
- VOID **FiniLine** (VOID)
- WORD FiniTerm (PHEAD WORD ∗, WORD ∗, WORD ∗, WORD, WORD)
- WORD FlushOut (POSITION ∗, FILEHANDLE ∗, int)
- VOID FunLevel (PHEAD WORD ∗)
- VOID AdjustRenumScratch (PHEAD0)
- VOID **GarbHand** (VOID)
- WORD GcdLong (PHEAD UWORD ∗, WORD, UWORD ∗, WORD, UWORD ∗, WORD ∗)
- WORD LcmLong (PHEAD UWORD ∗, WORD, UWORD ∗, WORD, UWORD ∗, WORD ∗)
- VOID **GCD** (UWORD ∗, WORD, UWORD ∗, WORD, UWORD ∗, WORD ∗)
- ULONG **GCD2** (ULONG, ULONG)
- WORD Generator (PHEAD WORD ∗, WORD)
- WORD GetBinom (UWORD ∗, WORD ∗, WORD, WORD)
- WORD GetFromStore (WORD ∗, POSITION ∗, RENUMBER, WORD ∗, WORD)
- WORD GetLong (UBYTE ∗, UWORD ∗, WORD ∗)
- WORD GetMoreTerms (WORD ∗)
- WORD GetMoreFromMem (WORD ∗, WORD ∗∗)
- WORD GetOneTerm (PHEAD WORD ∗, FILEHANDLE ∗, POSITION ∗, int)
- RENUMBER GetTable (WORD, POSITION ∗, WORD)
- WORD GetTerm (PHEAD WORD ∗)
- WORD Glue (PHEAD WORD ∗, WORD ∗, WORD ∗, WORD)
- WORD InFunction (PHEAD WORD ∗, WORD ∗)
- VOID IniLine (WORD)
- WORD **IniVars** (VOID)
- VOID **Initialize** (VOID)

- WORD InsertTerm (PHEAD WORD *, WORD, WORD, WORD *, WORD *, WORD)
- VOID LongToLine (UWORD *, WORD)
- WORD MakeDirty (WORD *, WORD *, WORD)
- VOID MarkDirty (WORD *, WORD)
- VOID PolyFunDirty (PHEAD WORD *)
- VOID PolyFunClean (PHEAD WORD *)
- WORD **MakeModTable** (VOID)
- WORD MatchE (PHEAD WORD *, WORD *, WORD *, WORD)
- int MatchCy (PHEAD WORD *, WORD *, WORD *, WORD)
- int FunMatchCy (PHEAD WORD *, WORD *, WORD *, WORD)
- int FunMatchSy (PHEAD WORD *, WORD *, WORD *, WORD)
- int MatchArgument (PHEAD WORD *, WORD *)
- WORD MatchFunction (PHEAD WORD *, WORD *, WORD *)
- WORD MergePatches (WORD)
- WORD MesCerr (char *, UBYTE *)
- WORD MesComp (char *, UBYTE *, UBYTE *)
- WORD Modulus (WORD *)
- VOID MoveDummies (PHEAD WORD *, WORD)
- WORD MulLong (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- WORD MulRat (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- WORD Mully (PHEAD UWORD *, WORD *, UWORD *, WORD)
- WORD MultDo (PHEAD WORD *, WORD *)
- WORD NewSort (PHEAD0)
- WORD ExtraSymbol (WORD, WORD, WORD, WORD *, WORD *)
- WORD Normalize (PHEAD WORD *)
- WORD BracketNormalize (PHEAD WORD *)
- VOID DropCoefficient (PHEAD WORD *)
- VOID DropSymbols (PHEAD WORD *)
- int PutInside (PHEAD WORD *, WORD *)
- WORD **OpenTemp** (VOID)
- VOID Pack (UWORD *, WORD *, UWORD *, WORD)
- LONG PasteFile (PHEAD WORD, WORD *, POSITION *, WORD **, RENUMBER, WORD *, WORD)
- WORD Permute (PERM *, WORD)
- WORD PermuteP (PERMP *, WORD)
- WORD PolyFunMul (PHEAD WORD *)
- WORD **PopVariables** (VOID)
- WORD PrepPoly (PHEAD WORD *, WORD)
- WORD **Processor** (VOID)
- WORD Product (UWORD *, WORD *, WORD)
- VOID PrtLong (UWORD *, WORD, UBYTE *)
- VOID **PrtTerms** (VOID)
- VOID **PrintRunningTime** (VOID)
- LONG **GetRunningTime** (VOID)
- WORD PutBracket (PHEAD WORD *)
- LONG PutIn (FILEHANDLE *, POSITION *, WORD *, WORD **, int)
- WORD PutInStore (INDEXENTRY *, WORD)
- WORD PutOut (PHEAD WORD *, POSITION *, FILEHANDLE *, WORD)
- UWORD Quotient (UWORD *, WORD *, WORD)
- WORD RaisPow (PHEAD UWORD *, WORD *, UWORD)
- VOID RaisPowCached (PHEAD WORD, WORD, UWORD **, WORD *)
- WORD RaisPowMod (WORD, WORD, WORD)
- int NormalModulus (UWORD *, WORD *)
- int **MakeInverses** (VOID)
- int GetModInverses (WORD, WORD, WORD *, WORD *)

- int GetLongModInverses (PHEAD UWORD ∗, WORD, UWORD ∗, WORD, UWORD ∗, WORD ∗, UWORD ∗, WORD ∗)
- VOID RatToLine (UWORD ∗, WORD)
- WORD RatioFind (PHEAD WORD ∗, WORD ∗)
- WORD RatioGen (PHEAD WORD ∗, WORD ∗, WORD, WORD)
- WORD ReNumber (PHEAD WORD ∗)
- WORD ReadSnum (UBYTE ∗∗)
- WORD Remain10 (UWORD ∗, WORD ∗)
- WORD Remain4 (UWORD ∗, WORD ∗)
- WORD **ResetScratch** (VOID)
- WORD ResolveSet (PHEAD WORD ∗, WORD ∗, WORD ∗)
- WORD **RevertScratch** (VOID)
- WORD ScanFunctions (PHEAD WORD ∗, WORD ∗, WORD)
- VOID SeekScratch (FILEHANDLE ∗, POSITION ∗)
- VOID SetEndScratch (FILEHANDLE ∗, POSITION ∗)
- VOID SetEndHScratch (FILEHANDLE ∗, POSITION ∗)
- WORD **SetFileIndex** (VOID)
- WORD Sflush (FILEHANDLE ∗)
- WORD Simplify (PHEAD UWORD ∗, WORD ∗, UWORD ∗, WORD ∗)
- WORD SortWild (WORD ∗, WORD)
- FILE ∗ LocateBase (char ∗∗, char ∗∗)
- VOID SplitMerge (PHEAD WORD ∗∗, LONG)
- WORD StoreTerm (PHEAD WORD ∗)
- VOID SubPLon (UWORD ∗, WORD, UWORD ∗, WORD, UWORD ∗, WORD ∗)
- VOID Substitute (PHEAD WORD ∗, WORD ∗, WORD)
- WORD SymFind (PHEAD WORD ∗, WORD ∗)
- WORD SymGen (PHEAD WORD ∗, WORD ∗, WORD, WORD)
- WORD Symmetrize (PHEAD WORD ∗, WORD ∗, WORD, WORD, WORD)
- int FullSymmetrize (PHEAD WORD ∗, int)
- WORD TakeModulus (UWORD ∗, WORD ∗, UWORD ∗, WORD, WORD)
- WORD TakeNormalModulus (UWORD ∗, WORD ∗, UWORD ∗, WORD, WORD)
- VOID TalToLine (UWORD)
- WORD TenVec (PHEAD WORD ∗, WORD ∗, WORD, WORD)
- WORD TenVecFind (PHEAD WORD ∗, WORD ∗)
- WORD TermRenumber (WORD ∗, RENUMBER, WORD)
- VOID **TestDrop** (VOID)
- VOID PutInVflags (WORD)
- WORD TestMatch (PHEAD WORD ∗, WORD ∗)
- WORD TestSub (PHEAD WORD ∗, WORD)
- LONG TimeCPU (WORD)
- LONG TimeChildren (WORD)
- LONG TimeWallClock (WORD)
- LONG **Timer** (int)
- int **GetTimerInfo** (LONG ∗∗, LONG ∗∗)
- void **WriteTimerInfo** (LONG ∗, LONG ∗)
- LONG **GetWorkerTimes** (VOID)
- WORD ToStorage (EXPRESSIONS, POSITION ∗)
- VOID TokenToLine (UBYTE ∗)
- WORD Trace4 (PHEAD WORD ∗, WORD ∗, WORD, WORD)
- WORD Trace4Gen (PHEAD TRACES ∗, WORD)
- WORD Trace4no (WORD, WORD ∗, TRACES ∗)
- WORD TraceFind (PHEAD WORD ∗, WORD ∗)
- WORD TraceN (PHEAD WORD ∗, WORD ∗, WORD, WORD)
- WORD TraceNgen (PHEAD TRACES ∗, WORD)
- WORD TraceNno (WORD, WORD ∗, TRACES ∗)

- WORD Traces (PHEAD WORD ∗, WORD ∗, WORD, WORD)
- WORD Trick (WORD ∗, TRACES ∗)
- WORD TryDo (PHEAD WORD ∗, WORD ∗, WORD)
- VOID UnPack (UWORD ∗, WORD, WORD ∗, WORD ∗)
- WORD VarStore (UBYTE ∗, WORD, WORD, WORD)
- WORD WildFill (PHEAD WORD ∗, WORD ∗, WORD ∗)
- WORD **WriteAll** (VOID)
- WORD WriteOne (UBYTE ∗, int, int)
- VOID WriteArgument (WORD ∗)
- WORD WriteExpression (WORD ∗, LONG)
- WORD WriteInnerTerm (WORD ∗, WORD)
- VOID **WriteLists** (VOID)
- VOID **WriteSetup** (VOID)
- VOID WriteStats (POSITION ∗, WORD)
- WORD WriteSubTerm (WORD ∗, WORD)
- WORD WriteTerm (WORD ∗, WORD ∗, WORD, WORD, WORD)
- WORD execarg (PHEAD WORD ∗, WORD)
- WORD execterm (PHEAD WORD ∗, WORD)
- VOID SpecialCleanup (PHEAD0)
- void SetMods ()
- void UnSetMods ()
- WORD DoExecute (WORD, WORD)
- VOID SetScratch (FILEHANDLE ∗, POSITION ∗)
- VOID Warning (char ∗)
- VOID HighWarning (char ∗)
- int SpareTable (TABLES)
- UBYTE ∗ strDup1 (UBYTE ∗, char ∗)
- VOID ∗ Malloc (LONG)
- VOID ∗ Malloc1 (LONG, const char ∗)
- int DoTail (int, UBYTE ∗∗)
- int **OpenInput** (VOID)
- int PutPreVar (UBYTE ∗, UBYTE ∗, UBYTE ∗, int)
- VOID Error0 (char ∗)
- VOID Error1 (char ∗, UBYTE ∗)
- VOID Error2 (char ∗, char ∗, UBYTE ∗)
- UBYTE ReadFromStream (STREAM ∗)
- UBYTE GetFromStream (STREAM ∗)
- UBYTE LookInStream (STREAM ∗)
- STREAM ∗ OpenStream (UBYTE ∗, int, int, int)
- int LocateFile (UBYTE ∗∗, int)
- STREAM ∗ CloseStream (STREAM ∗)
- VOID PositionStream (STREAM ∗, LONG)
- int ReverseStatements (STREAM ∗)
- int ProcessOption (UBYTE ∗, UBYTE ∗, int)
- int **DoSetups** (VOID)
- VOID Terminate (int)
- NAMENODE ∗ GetNode (NAMETREE ∗, UBYTE ∗)
- int AddName (NAMETREE ∗, UBYTE ∗, WORD, WORD, int ∗)
- int GetName (NAMETREE ∗, UBYTE ∗, WORD ∗, int)
- int GetLastExprName (UBYTE ∗, WORD ∗)
- int GetAutoName (UBYTE ∗, WORD ∗)
- int GetVar (UBYTE ∗, WORD ∗, WORD ∗, int, int)
- int MakeDubious (NAMETREE ∗, UBYTE ∗, WORD ∗)
- int GetOName (NAMETREE ∗, UBYTE ∗, WORD ∗, int)
- VOID DumpTree (NAMETREE ∗)

- VOID DumpNode (NAMETREE ∗, WORD, WORD)
- VOID LinkTree (NAMETREE ∗, WORD, WORD)
- VOID CopyTree (NAMETREE ∗, NAMETREE ∗, WORD, WORD)
- int CompactifyTree (NAMETREE ∗, WORD)
- NAMETREE ∗ **MakeNameTree** (VOID)
- VOID FreeNameTree (NAMETREE ∗)
- int AddExpression (UBYTE ∗, int, int)
- int AddSymbol (UBYTE ∗, int, int, int, int)
- int AddDollar (UBYTE ∗, WORD, WORD ∗, LONG)
- int ReplaceDollar (WORD, WORD, WORD ∗, LONG)
- int DollarRaiseLow (UBYTE ∗, LONG)
- int AddVector (UBYTE ∗, int, int)
- int AddDubious (UBYTE ∗)
- int AddIndex (UBYTE ∗, int, int)
- UBYTE ∗ DoDimension (UBYTE ∗, int ∗, int ∗)
- int AddFunction (UBYTE ∗, int, int, int, int, int, int, int)
- int CoCommuteInSet (UBYTE ∗)
- int CoFunction (UBYTE ∗, int, int)
- int TestName (UBYTE ∗)
- int AddSet (UBYTE ∗, WORD)
- int DoElements (UBYTE ∗, SETS, UBYTE ∗)
- int DoTempSet (UBYTE ∗, UBYTE ∗)
- int NameConflict (int, UBYTE ∗)
- int OpenFile (char ∗)
- int OpenAddFile (char ∗)
- int ReOpenFile (char ∗)
- int CreateFile (char ∗)
- int CreateLogFile (char ∗)
- VOID CloseFile (int)
- int CopyFile (char ∗, char ∗)
- int **CreateHandle** (VOID)
- LONG ReadFile (int, UBYTE ∗, LONG)
- LONG ReadPosFile (PHEAD FILEHANDLE ∗, UBYTE ∗, LONG, POSITION ∗)
- LONG WriteFileToFile (int, UBYTE ∗, LONG)
- VOID SeekFile (int, POSITION ∗, int)
- LONG TellFile (int)
- void FlushFile (int)
- int GetPosFile (int, fpos_t ∗)
- int SetPosFile (int, fpos_t ∗)
- VOID SynchFile (int)
- VOID TruncateFile (int)
- int GetChannel (char ∗)
- int GetAppendChannel (char ∗)
- int CloseChannel (char ∗)
- VOID **inictable** (VOID)
- KEYWORD ∗ findcommand (UBYTE ∗)
- int inicbufs (VOID)
- VOID **StartFiles** (VOID)
- UBYTE ∗ **MakeDate** (VOID)
- VOID **PreProcessor** (VOID)
- VOID ∗ FromList (LIST ∗)
- VOID ∗ From0List (LIST ∗)
- VOID ∗ FromVarList (LIST ∗)
- int DoubleList (VOID ∗∗∗, int ∗, int, char ∗)
- int DoubleLList (VOID ∗∗∗, LONG ∗, int, char ∗)

- void DoubleBuffer (void **, void **, int, char *)
- void ExpandBuffer (void **, LONG *, int)
- LONG iexp (LONG, int)
- int IsLikeVector (WORD *)
- int AreArgsEqual (WORD *, WORD *)
- int CompareArgs (WORD *, WORD *)
- UBYTE * SkipField (UBYTE *, int)
- int StrCmp (UBYTE *, UBYTE *)
- int StrICmp (UBYTE *, UBYTE *)
- int StrHICmp (UBYTE *, UBYTE *)
- int StrICont (UBYTE *, UBYTE *)
- int ConWord (UBYTE *, UBYTE *)
- int StrLen (UBYTE *)
- UBYTE * GetPreVar (UBYTE *, int)
- void ToGeneral (WORD *, WORD *, WORD)
- WORD ToPolyFunGeneral (PHEAD WORD *)
- int ToFast (WORD *, WORD *)
- SETUPPARAMETERS * GetSetupPar (UBYTE *)
- int **RecalcSetups** (VOID)
- int **AllocSetups** (VOID)
- SORTING * AllocSort (LONG, LONG, LONG, LONG, int, int, LONG)
- VOID AllocSortFileName (SORTING *)
- UBYTE * LoadInputFile (UBYTE *, int)
- UBYTE **GetInput** (VOID)
- VOID **ClearPushback** (VOID)
- UBYTE GetChar (int)
- VOID CharOut (UBYTE)
- VOID **UnsetAllowDelay** (VOID)
- VOID PopPreVars (int)
- VOID IniModule (int)
- VOID IniSpecialModule (int)
- int ModuleInstruction (int *, int *)
- int **PreProInstruction** (VOID)
- int LoadInstruction (int)
- int LoadStatement (int)
- KEYWORD * FindKeyWord (UBYTE *, KEYWORD *, int)
- KEYWORD * FindInKeyWord (UBYTE *, KEYWORD *, int)
- int DoDefine (UBYTE *)
- int DoRedefine (UBYTE *)
- int TheDefine (UBYTE *, int)
- int TheUndefine (UBYTE *)
- int ClearMacro (UBYTE *)
- int DoUndefine (UBYTE *)
- int DoInclude (UBYTE *)
- int DoReverseInclude (UBYTE *)
- int Include (UBYTE *, int)
- int DoExternal (UBYTE *)
- int DoToExternal (UBYTE *)
- int DoFromExternal (UBYTE *)
- int DoPrompt (UBYTE *)
- int DoSetExternal (UBYTE *)
- int DoSetExternalAttr (UBYTE *)
- int DoRmExternal (UBYTE *)
- int DoFactDollar (UBYTE *)
- WORD GetDollarNumber (UBYTE **, DOLLARS)

- int DoSetRandom (UBYTE *)
- int DoOptimize (UBYTE *)
- int DoClearOptimize (UBYTE *)
- int DoSkipExtraSymbols (UBYTE *)
- int DoMessage (UBYTE *)
- int DoPreOut (UBYTE *)
- int DoPreAppend (UBYTE *)
- int DoPreCreate (UBYTE *)
- int DoPreAssign (UBYTE *)
- int DoPreBreak (UBYTE *)
- int DoPreDefault (UBYTE *)
- int DoPreSwitch (UBYTE *)
- int DoPreEndSwitch (UBYTE *)
- int DoPreCase (UBYTE *)
- int DoPreShow (UBYTE *)
- int DoPreExchange (UBYTE *)
- int DoSystem (UBYTE *)
- int DoPipe (UBYTE *)
- VOID **StartPrepro** (VOID)
- int DoIfdef (UBYTE *, int)
- int DoElse (UBYTE *)
- int DoElseif (UBYTE *)
- int DoEndif (UBYTE *)
- int DoTerminate (UBYTE *)
- int DoIf (UBYTE *)
- int DoCall (UBYTE *)
- int DoDebug (UBYTE *)
- int DoDo (UBYTE *)
- int DoBreakDo (UBYTE *)
- int DoEnddo (UBYTE *)
- int DoEndprocedure (UBYTE *)
- int DoInside (UBYTE *)
- int DoEndInside (UBYTE *)
- int DoProcedure (UBYTE *)
- int DoPrePrintTimes (UBYTE *)
- int DoPreWrite (UBYTE *)
- int DoPreClose (UBYTE *)
- int DoPreRemove (UBYTE *)
- int DoCommentChar (UBYTE *)
- int DoPrcExtension (UBYTE *)
- int DoPreReset (UBYTE *)
- VOID WriteString (int, UBYTE *, int)
- VOID WriteUnfinString (int, UBYTE *, int)
- UBYTE * AddToString (UBYTE *, UBYTE *, int)
- UBYTE * **PreCalc** (VOID)
- UBYTE * PreEval (UBYTE *, LONG *)
- VOID NumToStr (UBYTE *, LONG)
- int PreCmp (int, int, UBYTE *, int, int, UBYTE *, int)
- int PreEq (int, int, UBYTE *, int, int, UBYTE *, int)
- UBYTE * pParseObject (UBYTE *, int *, LONG *)
- UBYTE * PreIfEval (UBYTE *, int *)
- int EvalPreIf (UBYTE *)
- int PreLoad (PRELOAD *, UBYTE *, UBYTE *, int, char *)
- int PreSkip (UBYTE *, UBYTE *, int)
- UBYTE * EndOfToken (UBYTE *)

- VOID SetSpecialMode (int, int)
- VOID **MakeGlobal** (VOID)
- int ExecModule (int)
- int **ExecStore** (VOID)
- VOID **FullCleanUp** (VOID)
- int **DoExecStatement** (VOID)
- int **DoPipeStatement** (VOID)
- int DoPolyfun (UBYTE ∗)
- int DoPolyratfun (UBYTE ∗)
- int CompileStatement (UBYTE ∗)
- UBYTE ∗ ToToken (UBYTE ∗)
- int GetDollar (UBYTE ∗)
- int **MesWork** (VOID)
- int **MesPrint** (const char ∗,...)
- int MesCall (char ∗)
- UBYTE ∗ NumCopy (WORD, UBYTE ∗)
- char ∗ LongCopy (LONG, char ∗)
- char ∗ LongLongCopy (off_t ∗, char ∗)
- VOID ReserveTempFiles (int)
- VOID PrintTerm (WORD ∗, char ∗)
- VOID PrintTermC (WORD ∗, char ∗)
- VOID PrintSubTerm (WORD ∗, char ∗)
- VOID PrintWords (WORD ∗, LONG)
- void PrintSeq (WORD ∗, char ∗)
- int ExpandTripleDots (int)
- LONG ComPress (WORD ∗∗, LONG ∗)
- VOID StageSort (FILEHANDLE ∗)
- void M_free (VOID ∗, const char ∗)
- void **ClearWildcardNames** (VOID)
- int AddWildcardName (UBYTE ∗)
- int GetWildcardName (UBYTE ∗)
- void Globalize (int)
- void ResetVariables (int)
- void AddToPreTypes (int)
- void MessPreNesting (int)
- LONG GetStreamPosition (STREAM ∗)
- WORD ∗ DoubleCbuffer (int, WORD ∗, int)
- WORD ∗ AddLHS (int)
- WORD ∗ AddRHS (int, int)
- int AddNtoL (int, WORD ∗)
- int AddNtoC (int, int, WORD ∗, int)
- VOID **DoubleIfBuffers** (VOID)
- STREAM ∗ CreateStream (UBYTE ∗)
- int setonoff (UBYTE ∗, int ∗, int, int)
- int DoPrint (UBYTE ∗, int)
- int SetExpr (UBYTE ∗, int, int)
- void AddToCom (int, WORD ∗)
- int Add2ComStrings (int, WORD ∗, UBYTE ∗, UBYTE ∗)
- int DoSymmetrize (UBYTE ∗, int)
- int DoArgument (UBYTE ∗, int)
- int ArgFactorize (PHEAD WORD ∗, WORD ∗)
- WORD ∗ TakeArgContent (PHEAD WORD ∗, WORD ∗)
- WORD ∗ MakeInteger (PHEAD WORD ∗, WORD ∗, WORD ∗)
- WORD ∗ MakeMod (PHEAD WORD ∗, WORD ∗, WORD ∗)
- WORD FindArg (PHEAD WORD ∗)

- WORD InsertArg (PHEAD WORD *, WORD *, int)
- int CleanupArgCache (PHEAD WORD)
- int ArgSymbolMerge (WORD *, WORD *)
- int ArgDotproductMerge (WORD *, WORD *)
- void SortWeights (LONG *, LONG *, WORD)
- int DoBrackets (UBYTE *, int)
- int DoPutInside (UBYTE *, int)
- WORD * CountComp (UBYTE *, WORD *)
- int CoAntiBracket (UBYTE *)
- int CoAntiSymmetrize (UBYTE *)
- int DoArgPlode (UBYTE *, int)
- int CoArgExplode (UBYTE *)
- int CoArgImplode (UBYTE *)
- int CoArgument (UBYTE *)
- int CoInside (UBYTE *)
- int ExecInside (UBYTE *)
- int CoInExpression (UBYTE *)
- int CoInParallel (UBYTE *)
- int CoNotInParallel (UBYTE *)
- int DoInParallel (UBYTE *, int)
- int CoEndInExpression (UBYTE *)
- int CoBracket (UBYTE *)
- int CoPutInside (UBYTE *)
- int CoAntiPutInside (UBYTE *)
- int CoMultiBracket (UBYTE *)
- int CoCFunction (UBYTE *)
- int CoCTensor (UBYTE *)
- int CoCollect (UBYTE *)
- int CoCompress (UBYTE *)
- int CoContract (UBYTE *)
- int CoCycleSymmetrize (UBYTE *)
- int CoDelete (UBYTE *)
- int CoTableBase (UBYTE *)
- int CoApply (UBYTE *)
- int CoDenominators (UBYTE *)
- int CoDimension (UBYTE *)
- int CoDiscard (UBYTE *)
- int CoDisorder (UBYTE *)
- int CoDrop (UBYTE *)
- int CoDropCoefficient (UBYTE *)
- int CoDropSymbols (UBYTE *)
- int CoElse (UBYTE *)
- int CoElseIf (UBYTE *)
- int CoEndArgument (UBYTE *)
- int CoEndInside (UBYTE *)
- int CoEndIf (UBYTE *)
- int CoEndRepeat (UBYTE *)
- int CoEndTerm (UBYTE *)
- int CoEndWhile (UBYTE *)
- int CoExit (UBYTE *)
- int CoFactArg (UBYTE *)
- int CoFactDollar (UBYTE *)
- int CoFactorize (UBYTE *)
- int CoNFactorize (UBYTE *)
- int CoUnFactorize (UBYTE *)

- int CoNUnFactorize (UBYTE *)
- int DoFactorize (UBYTE *, int)
- int CoFill (UBYTE *)
- int CoFillExpression (UBYTE *)
- int CoFixIndex (UBYTE *)
- int CoFormat (UBYTE *)
- int CoGlobal (UBYTE *)
- int CoGlobalFactorized (UBYTE *)
- int CoGoTo (UBYTE *)
- int CoId (UBYTE *)
- int CoIdNew (UBYTE *)
- int CoIdOld (UBYTE *)
- int CoIf (UBYTE *)
- int CoIfMatch (UBYTE *)
- int CoIfNoMatch (UBYTE *)
- int CoIndex (UBYTE *)
- int CoInsideFirst (UBYTE *)
- int CoKeep (UBYTE *)
- int CoLabel (UBYTE *)
- int CoLoad (UBYTE *)
- int CoLocal (UBYTE *)
- int CoLocalFactorized (UBYTE *)
- int CoMany (UBYTE *)
- int CoMerge (UBYTE *)
- int CoStuffle (UBYTE *)
- int CoMetric (UBYTE *)
- int CoModOption (UBYTE *)
- int CoModuleOption (UBYTE *)
- int CoModulus (UBYTE *)
- int CoMulti (UBYTE *)
- int CoMultiply (UBYTE *)
- int CoNFunction (UBYTE *)
- int CoNPrint (UBYTE *)
- int CoNTensor (UBYTE *)
- int CoNWrite (UBYTE *)
- int CoNoDrop (UBYTE *)
- int CoNoSkip (UBYTE *)
- int CoNormalize (UBYTE *)
- int CoMakeInteger (UBYTE *)
- int CoFlags (UBYTE *, int)
- int CoOff (UBYTE *)
- int CoOn (UBYTE *)
- int CoOnce (UBYTE *)
- int CoOnly (UBYTE *)
- int CoOptimizeOption (UBYTE *)
- int **CoOptimize** (UBYTE *)
- int CoPolyFun (UBYTE *)
- int CoPolyRatFun (UBYTE *)
- int CoPrint (UBYTE *)
- int CoPrintB (UBYTE *)
- int CoProperCount (UBYTE *)
- int CoUnitTrace (UBYTE *)
- int CoRCycleSymmetrize (UBYTE *)
- int CoRatio (UBYTE *)
- int CoRedefine (UBYTE *)

- int CoRenumber (UBYTE *)
- int CoRepeat (UBYTE *)
- int CoSave (UBYTE *)
- int CoSelect (UBYTE *)
- int CoSet (UBYTE *)
- int CoSetExitFlag (UBYTE *)
- int CoSkip (UBYTE *)
- int CoProcessBucket (UBYTE *)
- int CoPushHide (UBYTE *)
- int CoPopHide (UBYTE *)
- int CoHide (UBYTE *)
- int CoIntoHide (UBYTE *)
- int CoNoHide (UBYTE *)
- int CoUnHide (UBYTE *)
- int CoNoUnHide (UBYTE *)
- int CoSort (UBYTE *)
- int CoSplitArg (UBYTE *)
- int CoSplitFirstArg (UBYTE *)
- int CoSplitLastArg (UBYTE *)
- int CoSum (UBYTE *)
- int CoSymbol (UBYTE *)
- int CoSymmetrize (UBYTE *)
- int DoTable (UBYTE *, int)
- int CoTable (UBYTE *)
- int CoTerm (UBYTE *)
- int CoNTable (UBYTE *)
- int CoCTable (UBYTE *)
- void EmptyTable (TABLES)
- int CoToTensor (UBYTE *)
- int CoToVector (UBYTE *)
- int CoTrace4 (UBYTE *)
- int CoTraceN (UBYTE *)
- int CoChisholm (UBYTE *)
- int CoTransform (UBYTE *)
- int CoClearTable (UBYTE *)
- int DoChain (UBYTE *, int)
- int CoChainin (UBYTE *)
- int CoChainout (UBYTE *)
- int CoTryReplace (UBYTE *)
- int CoVector (UBYTE *)
- int CoWhile (UBYTE *)
- int CoWrite (UBYTE *)
- int CoAuto (UBYTE *)
- int CoTBaddto (UBYTE *)
- int CoTBaudit (UBYTE *)
- int CoTBcleanup (UBYTE *)
- int CoTBcreate (UBYTE *)
- int CoTBenter (UBYTE *)
- int CoTBhelp (UBYTE *)
- int CoTBload (UBYTE *)
- int CoTBoff (UBYTE *)
- int CoTBon (UBYTE *)
- int CoTBopen (UBYTE *)
- int CoTBreplace (UBYTE *)
- int CoTBuse (UBYTE *)

- int CoTestUse (UBYTE *)
- int CoThreadBucket (UBYTE *)
- int AddComString (int, WORD *, UBYTE *, int)
- int CompileAlgebra (UBYTE *, int, WORD *)
- int IsIdStatement (UBYTE *)
- UBYTE * IsRHS (UBYTE *, UBYTE)
- int ParenthesesTest (UBYTE *)
- int tokenize (UBYTE *, WORD)
- void WriteTokens (SBYTE *)
- int simp1token (SBYTE *)
- int simpwtoken (SBYTE *)
- int simp2token (SBYTE *)
- int simp3atoken (SBYTE *, int)
- int simp3btoken (SBYTE *, int)
- int simp4token (SBYTE *)
- int simp5token (SBYTE *, int)
- int simp6token (SBYTE *, int)
- UBYTE * SkipAName (UBYTE *)
- int **TestTables** (VOID)
- int GetLabel (UBYTE *)
- int CoIdExpression (UBYTE *, int)
- int CoAssign (UBYTE *)
- int DoExpr (UBYTE *, int, int)
- int CompileSubExpressions (SBYTE *)
- int CodeGenerator (SBYTE *)
- int CompleteTerm (WORD *, UWORD *, UWORD *, WORD, WORD, int)
- int CodeFactors (SBYTE *s)
- WORD GenerateFactors (WORD, WORD)
- int InsTree (int, int)
- int FindTree (int, WORD *)
- void RedoTree (CBUF *, int)
- void ClearTree (int)
- int CatchDollar (int)
- int AssignDollar (PHEAD WORD *, WORD)
- UBYTE * WriteDollarToBuffer (WORD, WORD)
- UBYTE * WriteDollarFactorToBuffer (WORD, WORD, WORD)
- void AddToDollarBuffer (UBYTE *)
- void TermAssign (WORD *)
- void WildDollars (PHEAD WORD *)
- LONG numcommute (WORD *, LONG *)
- int FullRenumber (PHEAD WORD *, WORD)
- int Lus (WORD *, WORD, WORD, WORD, WORD, WORD)
- int FindLus (int, int, int)
- int CoReplaceLoop (UBYTE *)
- int CoFindLoop (UBYTE *)
- int DoFindLoop (UBYTE *, int)
- int CoFunPowers (UBYTE *)
- int SortTheList (int *, int)
- int MatchIsPossible (WORD *, WORD *)
- int StudyPattern (WORD *)
- WORD DolToTensor (PHEAD WORD)
- WORD DolToFunction (PHEAD WORD)
- WORD DolToVector (PHEAD WORD)
- WORD DolToNumber (PHEAD WORD)
- WORD DolToSymbol (PHEAD WORD)

- WORD DolToIndex (PHEAD WORD)
- LONG DolToLong (PHEAD WORD)
- int DollarFactorize (PHEAD WORD)
- int CoPrintTable (UBYTE *)
- int CoDeallocateTable (UBYTE *)
- void CleanDollarFactors (DOLLARS)
- WORD * TakeDollarContent (PHEAD WORD *, WORD **)
- WORD * MakeDollarInteger (PHEAD WORD *, WORD **)
- WORD * MakeDollarMod (PHEAD WORD *, WORD **)
- int GetDolNum (PHEAD WORD *, WORD *)
- void AddPotModdollar (WORD)
- int **Optimize** (WORD, int)
- int **ClearOptimize** (VOID)
- int **LoadOpti** (WORD)
- int **PutObject** (WORD *, int)
- void **CleanOptiBuffer** (VOID)
- int **PrintOptima** (WORD)
- int **FindScratchName** (VOID)
- WORD **MaxPowerOpti** (LONG)
- WORD **HuntNumFactor** (LONG, WORD *, int)
- WORD **HuntFactor** (LONG, WORD *, int)
- void **HuntPairs** (LONG, WORD)
- void **HuntBrackets** (LONG)
- int **AddToOpti** (WORD *, int)
- LONG **TestNewSca** (LONG, WORD *, WORD *)
- void **NormOpti** (WORD *)
- void **SortOpti** (LONG)
- void **SplitOpti** (WORD **, LONG)
- void **CombiOpti** (VOID)
- int TakeLongRoot (UWORD *, WORD *, WORD)
- int TakeRatRoot (UWORD *, WORD *, WORD)
- int MakeRational (WORD, WORD, WORD *, WORD *)
- int MakeLongRational (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- void **HuntPowers** (LONG, WORD)
- void **HuntNumBrackets** (LONG)
- void ClearTableTree (TABLES)
- int InsTableTree (TABLES, WORD *)
- void RedoTableTree (TABLES, int)
- int FindTableTree (TABLES, WORD *, int)
- void finishcbuf (WORD)
- void clearcbuf (WORD)
- void CleanUpSort (int)
- FILEHANDLE * AllocFileHandle (WORD, char *)
- VOID DeAllocFileHandle (FILEHANDLE *)
- VOID **LowerSortLevel** (VOID)
- WORD * PolyRatFunSpecial (PHEAD WORD *, WORD *)
- int InsideDollar (PHEAD WORD *, WORD)
- DOLLARS DolToTerms (PHEAD WORD)
- WORD EvalDoLoopArg (PHEAD WORD *, WORD)
- int SetExprCases (int, int, int)
- int TestSelect (WORD *, WORD *)
- VOID SubsInAll (PHEAD0)
- VOID TransferBuffer (int, int, int)
- int TakeIDfunction (PHEAD WORD *)
- int **MakeSetupAllocs** (VOID)

- int **TryFileSetups** (VOID)
- void ExchangeExpressions (int, int)
- void ExchangeDollars (int, int)
- int GetFirstBracket (WORD *, int)
- int GetFirstTerm (WORD *, int)
- int GetContent (WORD *, int)
- int CleanupTerm (WORD *)
- WORD ContentMerge (PHEAD WORD *, WORD *)
- UBYTE * PreIfDollarEval (UBYTE *, int *)
- LONG TermsInDollar (WORD)
- LONG TermsInExpression (WORD)
- WORD * TranslateExpression (UBYTE *)
- int IsSetMember (WORD *, WORD)
- int IsMultipleOf (WORD *, WORD *)
- int TwoExprCompare (WORD *, WORD *, int)
- void **UpdatePositions** (VOID)
- void **M_check** (VOID)
- void **M_print** (VOID)
- void **M_check1** (VOID)
- void **PrintTime** (VOID)
- POSITION * FindBracket (WORD, WORD *)
- VOID PutBracketInIndex (PHEAD WORD *, POSITION *)
- void ClearBracketIndex (WORD)
- VOID OpenBracketIndex (WORD)
- int DoNoParallel (UBYTE *)
- int DoParallel (UBYTE *)
- int DoModSum (UBYTE *)
- int DoModMax (UBYTE *)
- int DoModMin (UBYTE *)
- int DoModLocal (UBYTE *)
- UBYTE * DoModDollar (UBYTE *, int)
- int DoProcessBucket (UBYTE *)
- int DoinParallel (UBYTE *)
- int DonotinParallel (UBYTE *)
- int FlipTable (FUNCTIONS, int)
- int ChainIn (PHEAD WORD *, WORD)
- int ChainOut (PHEAD WORD *, WORD)
- int ArgumentImplode (PHEAD WORD *, WORD *)
- int ArgumentExplode (PHEAD WORD *, WORD *)
- int DenToFunction (WORD *, WORD)
- WORD HowMany (PHEAD WORD *, WORD *)
- VOID **RemoveDollars** (VOID)
- LONG CountTerms1 (PHEAD0)
- LONG TermsInBracket (PHEAD WORD *, WORD)
- int **Crash** (VOID)
- char * str_dup (char *)
- void convertblock (INDEXBLOCK *, INDEXBLOCK *, int)
- void convertnamesblock (NAMESBLOCK *, NAMESBLOCK *, int)
- void convertiniinfo (INIINFO *, INIINFO *, int)
- int ReadIndex (DBASE *)
- int WriteIndexBlock (DBASE *, MLONG)
- int WriteNamesBlock (DBASE *, MLONG)
- int WriteIndex (DBASE *)
- int WriteIniInfo (DBASE *)
- int ReadIniInfo (DBASE *)

- int AddToIndex (DBASE ∗, MLONG)
- DBASE ∗ GetDbase (char ∗)
- DBASE ∗ OpenDbase (char ∗)
- char ∗ ReadObject (DBASE ∗, MLONG, char ∗)
- char ∗ ReadijObject (DBASE ∗, MLONG, MLONG, char ∗)
- int ExistsObject (DBASE ∗, MLONG, char ∗)
- int DeleteObject (DBASE ∗, MLONG, char ∗)
- int WriteObject (DBASE ∗, MLONG, char ∗, char ∗, MLONG)
- MLONG AddObject (DBASE ∗, MLONG, char ∗, char ∗)
- int **Cleanup** (DBASE ∗)
- DBASE ∗ NewDbase (char ∗, MLONG)
- void FreeTableBase (DBASE ∗)
- int ComposeTableNames (DBASE ∗)
- int PutTableNames (DBASE ∗)
- MLONG AddTableName (DBASE ∗, char ∗, TABLES)
- MLONG GetTableName (DBASE ∗, char ∗)
- MLONG FindTableNumber (DBASE ∗, char ∗)
- int **TryEnvironment** (VOID)
- int CopyExpression (FILEHANDLE ∗, FILEHANDLE ∗)
- int set_in (UBYTE, set_of_char)
- one_byte set_set (UBYTE, set_of_char)
- one_byte set_del (UBYTE, set_of_char)
- one_byte set_sub (set_of_char, set_of_char, set_of_char)
- int DoPreAddSeparator (UBYTE ∗)
- int DoPreRmSeparator (UBYTE ∗)
- int openExternalChannel (UBYTE ∗, int, UBYTE ∗, UBYTE ∗)
- int initPresetExternalChannels (UBYTE ∗, int)
- int closeExternalChannel (int)
- int selectExternalChannel (int)
- int **getCurrentExternalChannel** (VOID)
- VOID **closeAllExternalChannels** (VOID)
- UBYTE ∗ defineChannel (UBYTE ∗, HANDLERS ∗)
- int writeToChannel (int, UBYTE ∗, HANDLERS ∗)
- int **writeBufToExtChannelOk** (char ∗, size_t)
- int **getcFromExtChannelOk** (VOID)
- int **setKillModeForExternalChannelOk** (int, int)
- int **setTerminatorForExternalChannelOk** (char ∗)
- int **getcFromExtChannelFailure** (VOID)
- int **setKillModeForExternalChannelFailure** (int, int)
- int **setTerminatorForExternalChannelFailure** (char ∗)
- int **writeBufToExtChannelFailure** (char ∗, size_t)
- int **ReleaseTB** (VOID)
- int SymbolNormalize (WORD ∗)
- int TestFunFlag (PHEAD WORD ∗)
- int CompareSymbols (PHEAD WORD ∗, WORD ∗, WORD)
- int CompareHSymbols (PHEAD WORD ∗, WORD ∗, WORD)
- WORD NextPrime (PHEAD WORD)
- UWORD wranf (PHEAD0)
- UWORD iranf (PHEAD UWORD)
- void iniwranf (PHEAD0)
- UBYTE ∗ PreRandom (UBYTE ∗)
- WORD ∗ **PolyNormPoly** (PHEAD WORD)
- WORD ∗ **EvaluateGcd** (PHEAD WORD ∗)
- int TreatPolyRatFun (PHEAD WORD ∗)
- WORD **ReadSaveHeader** (VOID)

- WORD ReadSaveIndex (FILEINDEX *)
- WORD ReadSaveExpression (UBYTE *, UBYTE *, LONG *, LONG *)
- UBYTE * ReadSaveTerm32 (UBYTE *, UBYTE *, UBYTE **, UBYTE *, UBYTE *, int)
- WORD ReadSaveVariables (UBYTE *, UBYTE *, LONG *, LONG *, INDEXENTRY *, LONG *)
- WORD WriteStoreHeader (WORD)
- void **InitRecovery** (VOID)
- int **CheckRecoveryFile** (VOID)
- void **DeleteRecoveryFile** (VOID)
- char * **RecoveryFilename** (VOID)
- int DoRecovery (int *)
- void DoCheckpoint (int)
- VOID NumberMallocAddMemory (PHEAD0)
- VOID CacheNumberMallocAddMemory (PHEAD0)
- VOID TermMallocAddMemory (PHEAD0)
- void ExprStatus (EXPRESSIONS)
- VOID iniTools (VOID)
- int TestTerm (WORD *)
- WORD RunTransform (PHEAD WORD *term, WORD *params)
- WORD RunEncode (PHEAD WORD *fun, WORD *args, WORD *info)
- WORD RunDecode (PHEAD WORD *fun, WORD *args, WORD *info)
- WORD RunReplace (PHEAD WORD *fun, WORD *args, WORD *info)
- WORD RunImplode (WORD *fun, WORD *args)
- WORD RunExplode (PHEAD WORD *fun, WORD *args)
- int TestArgNum (int n, int totarg, WORD *args)
- WORD PutArgInScratch (WORD *arg, UWORD *scrat)
- UBYTE * ReadRange (UBYTE *s, WORD *out, int par)
- int FindRange (PHEAD WORD *, WORD *, WORD *, WORD)
- WORD RunPermute (PHEAD WORD *fun, WORD *args, WORD *info)
- WORD RunReverse (PHEAD WORD *fun, WORD *args)
- WORD RunCycle (PHEAD WORD *fun, WORD *args, WORD *info)
- WORD RunAddArg (PHEAD WORD *fun, WORD *args)
- WORD RunMulArg (PHEAD WORD *fun, WORD *args)
- WORD RunIsLyndon (PHEAD WORD *fun, WORD *args, int par)
- WORD RunToLyndon (PHEAD WORD *fun, WORD *args, int par)
- WORD RunDropArg (PHEAD WORD *fun, WORD *args)
- WORD RunSelectArg (PHEAD WORD *fun, WORD *args)
- WORD RunDedup (PHEAD WORD *fun, WORD *args)
- int NormPolyTerm (PHEAD WORD *)
- WORD **ComparePoly** (WORD *, WORD *, WORD)
- int ConvertToPoly (PHEAD WORD *, WORD *, WORD *, WORD)
- int LocalConvertToPoly (PHEAD WORD *, WORD *, WORD, WORD)
- int ConvertFromPoly (PHEAD WORD *, WORD *, WORD, WORD, WORD, WORD)
- WORD FindSubterm (WORD *)
- WORD FindLocalSubterm (PHEAD WORD *, WORD)
- void PrintSubtermList (int, int)
- void PrintExtraSymbol (int, WORD *, int)
- WORD FindSubexpression (WORD *)
- void **UpdateMaxSize** (VOID)
- int CoToPolynomial (UBYTE *)
- int CoFromPolynomial (UBYTE *)
- int CoArgToExtraSymbol (UBYTE *)
- int CoExtraSymbols (UBYTE *)
- UBYTE * GetDoParam (UBYTE *, WORD **, int)
- WORD * GetIfDollarFactor (UBYTE **, WORD *)
- int CoDo (UBYTE *)

- int CoEndDo (UBYTE *)
- int ExtraSymFun (PHEAD WORD *, WORD)
- int PruneExtraSymbols (WORD)
- int IniFbuffer (WORD)
- void **IniFbufs** (VOID)
- int GCDfunction (PHEAD WORD *, WORD)
- WORD * GCDfunction3 (PHEAD WORD *, WORD *)
- WORD * **GCDfunction4** (PHEAD WORD *, WORD *)
- int ReadPolyRatFun (PHEAD WORD *)
- int FromPolyRatFun (PHEAD WORD *, WORD **, WORD **)
- void **PRFnormalize** (PHEAD WORD *)
- WORD * **PRFadd** (PHEAD WORD *, WORD *)
- WORD * PolyDiv (PHEAD WORD *, WORD *, char *)
- WORD * **PolyGCD** (PHEAD WORD *, WORD *)
- WORD * **PolyAdd** (PHEAD WORD *, WORD *)
- void GCDclean (PHEAD WORD *, WORD *)
- int **RatFunNormalize** (PHEAD WORD *)
- WORD * TakeSymbolContent (PHEAD WORD *, WORD *)
- int GCDterms (PHEAD WORD *, WORD *, WORD *)
- WORD * PutExtraSymbols (PHEAD WORD *, WORD, int *)
- WORD * TakeExtraSymbols (PHEAD WORD *, WORD)
- WORD * MultiplyWithTerm (PHEAD WORD *, WORD *, WORD)
- WORD * TakeContent (PHEAD WORD *, WORD *)
- int MergeSymbolLists (PHEAD WORD *, WORD *, int)
- int MergeDotproductLists (PHEAD WORD *, WORD *, int)
- WORD * CreateExpression (PHEAD WORD)
- int DIVfunction (PHEAD WORD *, WORD, int)
- WORD * MULfunc (PHEAD WORD *, WORD *)
- WORD * ConvertArgument (PHEAD WORD *, int *)
- int ExpandRat (PHEAD WORD *)
- int InvPoly (PHEAD WORD *, WORD, WORD)
- WORD TestDoLoop (PHEAD WORD *, WORD)
- WORD TestEndDoLoop (PHEAD WORD *, WORD)
- WORD * **poly_gcd** (PHEAD WORD *, WORD *, WORD)
- WORD * **poly_div** (PHEAD WORD *, WORD *, WORD)
- WORD * **poly_rem** (PHEAD WORD *, WORD *, WORD)
- WORD * **poly_inverse** (PHEAD WORD *, WORD *)
- WORD * **poly_mul** (PHEAD WORD *, WORD *)
- WORD * **poly_ratfun_add** (PHEAD WORD *, WORD *)
- int **poly_ratfun_normalize** (PHEAD WORD *)
- int **poly_factorize_argument** (PHEAD WORD *, WORD *)
- WORD * **poly_factorize_dollar** (PHEAD WORD *)
- int **poly_factorize_expression** (EXPRESSIONS)
- int **poly_unfactorize_expression** (EXPRESSIONS)
- void **poly_free_poly_vars** (PHEAD const char *)
- VOID **optimize_print_code** (int)
- int DoPreAdd (UBYTE *s)
- int DoPreUseDictionary (UBYTE *s)
- int DoPreCloseDictionary (UBYTE *s)
- int DoPreOpenDictionary (UBYTE *s)
- void RemoveDictionary (DICTIONARY *dict)
- void UnSetDictionary (VOID)
- int SetDictionaryOptions (UBYTE *options)
- int **SelectDictionary** (UBYTE *name, UBYTE *options)
- int AddToDictionary (DICTIONARY *dict, UBYTE *left, UBYTE *right)

- int AddDictionary (UBYTE *name)
- int FindDictionary (UBYTE *name)
- int IsExponentSign (VOID)
- int IsMultiplySign (VOID)
- VOID TransformRational (UWORD *a, WORD na)
- void WriteDictionary (DICTIONARY *)
- void ShrinkDictionary (DICTIONARY *)
- void **MultiplyToLine** (VOID)
- UBYTE * FindSymbol (WORD num)
- UBYTE * FindVector (WORD num)
- UBYTE * FindIndex (WORD num)
- UBYTE * FindFunction (WORD num)
- UBYTE * FindFunWithArgs (WORD *t)
- UBYTE * FindExtraSymbol (WORD num)
- LONG DictToBytes (DICTIONARY *dict, UBYTE *buf)
- DICTIONARY * DictFromBytes (UBYTE *buf)
- int CoCreateSpectator (UBYTE *inp)
- int CoToSpectator (UBYTE *inp)
- int CoRemoveSpectator (UBYTE *inp)
- int CoEmptySpectator (UBYTE *inp)
- int CoCopySpectator (UBYTE *inp)
- int PutInSpectator (WORD *, WORD)
- void ClearSpectators (WORD)
- WORD GetFromSpectator (WORD *, WORD)
- void FlushSpectators (VOID)
- WORD * **PreGCD** (PHEAD WORD *, WORD *, int)
- WORD * **FindCommonVariables** (PHEAD int, int)
- VOID **AddToSymbolList** (PHEAD WORD)
- int **AddToListPoly** (PHEAD0)
- int ReadFromScratch (FILEHANDLE *, POSITION *, UBYTE *, POSITION *)
- int AddToScratch (FILEHANDLE *, POSITION *, UBYTE *, POSITION *, int)
- int DoPreAppendPath (UBYTE *)
- int DoPrePrependPath (UBYTE *)

4.19.1 Detailed Description

Contains macros and function declarations.

Definition in file declare.h.

4.19.2 Macro Definition Documentation

4.19.2.1 MaX

```
#define MaX(
            x,
            y ) ((x) > (y) ? (x): (y))
```

Definition at line 40 of file declare.h.

4.19.2.2 MiN

```
#define MiN(
            x,
            y ) ((x) < (y) ? (x): (y))
```

Definition at line 41 of file declare.h.

4.19.2.3 ABS

```
#define ABS(
            x ) ( (x) < 0 ? -(x): (x) )
```

Definition at line 42 of file declare.h.

4.19.2.4 SGN

```
#define SGN(
            x ) ( (x) > 0 ? 1 : (x) < 0 ? -1 : 0 )
```

Definition at line 43 of file declare.h.

4.19.2.5 REDLENG

```
#define REDLENG(
            x ) ((((x)<0)?((x)+1):((x)-1))>>1)
```

Definition at line 44 of file declare.h.

4.19.2.6 INCLENG

```
#define INCLENG(
            x ) (((x)<0)?(((x)<<1)-1):(((x)<<1)+1))
```

Definition at line 45 of file declare.h.

4.19.2.7 GETCOEF

```
#define GETCOEF(
            x,
            y ) x += *x;y = x[-1];x -= ABS(y);y=REDLENG(y)
```

Definition at line 46 of file declare.h.

4.19.2.8 GETSTOP

```
#define GETSTOP(
            x,
            y ) y=x+(*x)-1;y -= ABS(*y)-1
```

Definition at line 47 of file declare.h.

4.19.2.9 StuffAdd

```
#define StuffAdd(
            x,
            y ) (((x)<0?-1:1)*(y)+((y)<0?-1:1)*(x))
```

Definition at line 48 of file declare.h.

4.19.2.10 TOKENTOLINE

```
#define TOKENTOLINE(
            x,
            y )
```

Value:

```
if ( AC.OutputSpaces == NOSPACEFORMAT ) { \
TokenToLine((UBYTE *)(y)); } else { TokenToLine((UBYTE *)(x)); }
```

Definition at line 50 of file declare.h.

4.19.2.11 UngetFromStream

```
#define UngetFromStream(
            stream,
            c ) ((stream)->nextchar[(stream)->isnextchar++]=c)
```

Definition at line 53 of file declare.h.

4.19.2.12 AddLineFeed

```
#define AddLineFeed(
            s,
            n ) { (s)[(n)++] = LINEFEED; }
```

Definition at line 57 of file declare.h.

4.19.2.13 TryRecover

```
#define TryRecover(
            x ) Terminate(-1)
```

Definition at line 59 of file declare.h.

4.19.2.14 UngetChar

```
#define UngetChar(
            c ) { pushbackchar = c; }
```

Definition at line 60 of file declare.h.

4.19.2.15 ParseNumber

```
#define ParseNumber(
            x,
            s ) {(x)=0;while(*(s)>='0'&&*(s)<='9')(x)=10*(x)+*(s)++ -'0';}
```

Definition at line 61 of file declare.h.

4.19.2.16 ParseSign

```
#define ParseSign(
            sgn,
            s )
```

Value:

```
{(sgn)=0;while(*(s)=='-'||*(s)=='+'){\
if ( *(s)++ == '-' ) sgn ^= 1;}}
```

Definition at line 62 of file declare.h.

4.19.2.17 ParseSignedNumber

```
#define ParseSignedNumber(
            x,
            s )
```

Value:

```
{ int sgn; ParseSign(sgn,s)\
ParseNumber(x,s) if ( sgn ) x = -x; }
```

Definition at line 64 of file declare.h.

4.19.2.18 NCOPY

```
#define NCOPY(
            s,
            t,
            n ) while ( --n >= 0 ) *s++ = *t++;
```

Definition at line 67 of file declare.h.

4.19.2.19 NCOPYI

```
#define NCOPYI(
              s,
              t,
              n ) while ( --n >= 0 ) *s++ = *t++;
```

Definition at line 70 of file declare.h.

4.19.2.20 NCOPYB

```
#define NCOPYB(
              s,
              t,
              n ) while ( --n >= 0 ) *s++ = *t++;
```

Definition at line 71 of file declare.h.

4.19.2.21 NCOPYI32

```
#define NCOPYI32(
              s,
              t,
              n ) while ( --n >= 0 ) *s++ = *t++;
```

Definition at line 72 of file declare.h.

4.19.2.22 WCOPY

```
#define WCOPY(
              s,
              t,
              n ) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; while ( --nn >= 0 ) *ss++=*tt++;
}
```

Definition at line 73 of file declare.h.

4.19.2.23 NeedNumber

```
#define NeedNumber(
              x,
              s,
              err )
```

Value:

```
        { int sgn = 1;                                  \
        while ( *s == ' ' || *s == '\t' || *s == '-' || *s == '+' ) {     \
            if ( *s == '-' ) sgn = -sgn; s++; }                           \
        if ( chartype[*s] != 1 ) goto err;                                \
        ParseNumber(x,s)                                                  \
        if ( sgn < 0 ) (x) = -(x); while ( *s == ' ' || *s == '\t' ) s++;\
    }
```

Definition at line 74 of file declare.h.

4.19.2.24 SKIPBLANKS

```
#define SKIPBLANKS(
              s ) { while ( *(s) == ' ' || *(s) == '\t' ) (s)++; }
```

Definition at line 81 of file declare.h.

4.19.2.25 FLUSHCONSOLE

```
#define FLUSHCONSOLE if ( AP.InOutBuf > 0 ) CharOut(LINEFEED)
```

Definition at line 82 of file declare.h.

4.19.2.26 SKIPBRA1

```
#define SKIPBRA1(
              s )
```

Value:

```
{ int lev1=0; s++; while(*s) { if(*s=='[')lev1++; \
else if(*s==']'&&--lev1<0)break; s++;} }
```

Definition at line 84 of file declare.h.

4.19.2.27 SKIPBRA2

```
#define SKIPBRA2(
              s )
```

Value:

```
{ int lev2=0; s++; while(*s) { if(*s=='{')lev2++; \
else if(*s=='}'&&--lev2<0)break; \
else if(*s=='[')SKIPBRA1(s) s++;} }
```

Definition at line 86 of file declare.h.

4.19.2.28 SKIPBRA3

```
#define SKIPBRA3(
              s )
```

Value:

```
{ int lev3=0; s++; while(*s) { if(*s=='(')lev3++; \
else if(*s==')'&&--lev3<0)break; \
else if(*s=='{')SKIPBRA2(s) \
else if(*s=='[')SKIPBRA1(s) s++;} }
```

Definition at line 89 of file declare.h.

4.19.2.29 SKIPBRA4

```
#define SKIPBRA4(
            s )
```

Value:

```
{ int lev4=0; s++; while(*s) { if(*s=='(')lev4++; \
else if(*s==')'&&--lev4<0)break; \
else if(*s=='[')SKIPBRA1(s) s++;} }
```

Definition at line 93 of file declare.h.

4.19.2.30 SKIPBRA5

```
#define SKIPBRA5(
            s )
```

Value:

```
{ int lev5=0; s++; while(*s) { if(*s=='{')lev5++; \
else if(*s=='}'&&--lev5<0)break; \
else if(*s=='(')SKIPBRA4(s) \
else if(*s=='[')SKIPBRA1(s) s++;} }
```

Definition at line 96 of file declare.h.

4.19.2.31 CYCLE1

```
#define CYCLE1(
            t,
            a,
            i ) {t iX=*a; WORD jX; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
```

Definition at line 104 of file declare.h.

4.19.2.32 AddToCB

```
#define AddToCB(
            c,
            wx )
```

Value:

```
if(c->Pointer>=c->Top) \
DoubleCbuffer(c-cbuf,c->Pointer,21); \
*(c->Pointer)++ = wx;
```

Definition at line 106 of file declare.h.

4.19.2.33 EXCHINOUT

```
#define EXCHINOUT
```

Value:

```
{ FILEHANDLE *ffFi = AR.outfile; \
AR.outfile = AR.infile; AR.infile = ffFi; }
```

Definition at line 110 of file declare.h.

4.19.2.34 BACKINOUT

```
#define BACKINOUT
```

Value:

```
{ FILEHANDLE *ffFi = AR.outfile; POSITION posi; \
AR.outfile = AR.infile; AR.infile = ffFi; \
SetEndScratch(AR.infile,&posi); }
```

Definition at line 112 of file declare.h.

4.19.2.35 CopyArg

```
#define CopyArg(
            to,
            from )
```

Value:

```
{ if ( *from > 0 ) { int ica = *from; NCOPY(to,from,ica) } \
else if ( *from <= -FUNCTION ) *to++ = *from++;  \
else { *to++ = *from++; *to++ = *from++; } }
```

Definition at line 116 of file declare.h.

4.19.2.36 FILLARG

```
#define FILLARG(
            w )
```

Definition at line 125 of file declare.h.

4.19.2.37 COPYARG

```
#define COPYARG(
            w,
            t )
```

Definition at line 126 of file declare.h.

4.19.2.38 ZEROARG

```
#define ZEROARG(
            w )
```

Definition at line 127 of file declare.h.

4.19.2.39 FILLFUN

```
#define FILLFUN(
            w )
```

Definition at line 134 of file declare.h.

4.19.2.40 COPYFUN

```
#define COPYFUN(
            w,
            t )
```

Definition at line 135 of file declare.h.

4.19.2.41 COPYFUN3

```
#define COPYFUN3(
            w,
            t )
```

Definition at line 142 of file declare.h.

4.19.2.42 FILLFUN3

```
#define FILLFUN3(
            w )
```

Definition at line 143 of file declare.h.

4.19.2.43 FILLSUB

```
#define FILLSUB(
            w )
```

Definition at line 150 of file declare.h.

4.19.2.44 COPYSUB

```
#define COPYSUB(
            w,
            ww )
```

Definition at line 151 of file declare.h.

4.19.2.45 FILLEXPR

```
#define FILLEXPR(
            w )
```

Definition at line 157 of file declare.h.

4.19.2.46 NEXTARG

```
#define NEXTARG(
              x ) if(*x>0) x += *x; else if(*x <= -FUNCTION)x++; else x += 2;
```

Definition at line 160 of file declare.h.

4.19.2.47 COPY1ARG

```
#define COPY1ARG(
              s1,
              t1 )
```

Value:

```
        { int ica; if ( (ica=*t1) > 0 ) { NCOPY(s1,t1,ica) } \
        else if(*t1<=-FUNCTION){*s1++=*t1++;} else{*s1++=*t1++;*s1++=*t1++;} }
```

Definition at line 161 of file declare.h.

4.19.2.48 ZeroFillRange

```
#define ZeroFillRange(
              w,
              begin,
              end )
```

Value:

```
    do { \
    int tmp_i; \
    for ( tmp_i = begin; tmp_i < end; tmp_i++ ) { (w)[tmp_i] = 0; } \
} while (0)
```

Fills a buffer by zero in the range [begin,end).

Parameters

w	The buffer.
begin	The index for the beginning of the range.
end	The index for the end of the range (exclusive).

Definition at line 171 of file declare.h.

4.19.2.49 TABLESIZE

```
#define TABLESIZE(
              a,
              b ) (((WORD)sizeof(a))/((WORD)sizeof(b)))
```

Definition at line 176 of file declare.h.

4.19.2.50 WORDDIF

```
#define WORDDIF(
            x,
            y ) (WORD)(x-y)
```

Definition at line 177 of file declare.h.

4.19.2.51 wsizeof

```
#define wsizeof(
            a ) ((WORD)sizeof(a))
```

Definition at line 178 of file declare.h.

4.19.2.52 VARNAME

```
#define VARNAME(
            type,
            num ) (AC.varnames->namebuffer+type[num].name)
```

Definition at line 179 of file declare.h.

4.19.2.53 DOLLARNAME

```
#define DOLLARNAME(
            type,
            num ) (AC.dollarnames->namebuffer+type[num].name)
```

Definition at line 180 of file declare.h.

4.19.2.54 EXPRNAME

```
#define EXPRNAME(
            num ) (AC.exprnames->namebuffer+Expressions[num].name)
```

Definition at line 181 of file declare.h.

4.19.2.55 PREV

```
#define PREV(
            x ) prevorder?prevorder:x
```

Definition at line 183 of file declare.h.

4.19.2.56 SETERROR

```
#define SETERROR(
            x ) { Terminate(-1); return(-1); }
```

Definition at line 185 of file declare.h.

4.19.2.57 DUMMYUSE

```
#define DUMMYUSE(
            x ) (void)(x);
```

Definition at line 188 of file declare.h.

4.19.2.58 ADDPOS

```
#define ADDPOS(
            pp,
            x ) (pp).p1 = ((pp).p1+(LONG)(x))
```

Definition at line 214 of file declare.h.

4.19.2.59 SETBASELENGTH

```
#define SETBASELENGTH(
            ss,
            x ) (ss).p1 = (LONG)(x)
```

Definition at line 215 of file declare.h.

4.19.2.60 SETBASEPOSITION

```
#define SETBASEPOSITION(
            pp,
            x ) (pp).p1 = (LONG)(x)
```

Definition at line 216 of file declare.h.

4.19.2.61 ISEQUALPOSINC

```
#define ISEQUALPOSINC(
            pp1,
            pp2,
            x ) ( (pp1).p1 == ((pp2).p1+(LONG)(x)) )
```

Definition at line 217 of file declare.h.

4.19.2.62 ISGEPOSINC

```
#define ISGEPOSINC(
            pp1,
            pp2,
            x ) ( (pp1).p1 >= ((pp2).p1+(LONG)(x)) )
```

Definition at line 218 of file declare.h.

4.19.2.63 DIVPOS

```
#define DIVPOS(
            pp,
            n ) ( (pp).p1/(LONG)(n) )
```

Definition at line 219 of file declare.h.

4.19.2.64 MULPOS

```
#define MULPOS(
            pp,
            n ) (pp).p1 *= (LONG)(n)
```

Definition at line 220 of file declare.h.

4.19.2.65 DIFPOS

```
#define DIFPOS(
            ss,
            pp1,
            pp2 ) (ss).p1 = ((pp1).p1-(pp2).p1)
```

Definition at line 223 of file declare.h.

4.19.2.66 DIFBASE

```
#define DIFBASE(
            pp1,
            pp2 ) ((pp1).p1-(pp2).p1)
```

Definition at line 224 of file declare.h.

4.19.2.67 ADD2POS

```
#define ADD2POS(
            pp1,
            pp2 ) (pp1).p1 += (pp2).p1
```

Definition at line 225 of file declare.h.

4.19.2.68 PUTZERO

```
#define PUTZERO(
            pp ) (pp).p1 = 0
```

Definition at line 226 of file declare.h.

4.19.2.69 BASEPOSITION

```
#define BASEPOSITION(
            pp ) ((pp).p1)
```

Definition at line 227 of file declare.h.

4.19.2.70 SETSTARTPOS

```
#define SETSTARTPOS(
            pp ) (pp).p1 = -2
```

Definition at line 228 of file declare.h.

4.19.2.71 NOTSTARTPOS

```
#define NOTSTARTPOS(
            pp ) ( (pp).p1 > -2 )
```

Definition at line 229 of file declare.h.

4.19.2.72 ISMINPOS

```
#define ISMINPOS(
            pp ) ( (pp).p1 == -1 )
```

Definition at line 230 of file declare.h.

4.19.2.73 ISEQUALPOS

```
#define ISEQUALPOS(
            pp1,
            pp2 ) ( (pp1).p1 == (pp2).p1 )
```

Definition at line 231 of file declare.h.

4.19.2.74 ISNOTEQUALPOS

```
#define ISNOTEQUALPOS(
            pp1,
            pp2 ) ( (pp1).p1 != (pp2).p1 )
```

Definition at line 232 of file declare.h.

4.19.2.75 ISLESSPOS

```
#define ISLESSPOS(
            pp1,
            pp2 ) ( (pp1).p1 < (pp2).p1 )
```

Definition at line 233 of file declare.h.

4.19.2.76 ISGEPOS

```
#define ISGEPOS(
            pp1,
            pp2 ) ( (pp1).p1 >= (pp2).p1 )
```

Definition at line 234 of file declare.h.

4.19.2.77 ISNOTZEROPOS

```
#define ISNOTZEROPOS(
            pp ) ( (pp).p1 != 0 )
```

Definition at line 235 of file declare.h.

4.19.2.78 ISZEROPOS

```
#define ISZEROPOS(
            pp ) ( (pp).p1 == 0 )
```

Definition at line 236 of file declare.h.

4.19.2.79 ISPOSPOS

```
#define ISPOSPOS(
            pp ) ( (pp).p1 > 0 )
```

Definition at line 237 of file declare.h.

4.19.2.80 ISNEGPOS

```
#define ISNEGPOS(
            pp ) ( (pp).p1 < 0 )
```

Definition at line 238 of file declare.h.

4.19.2.81 TOLONG

```
#define TOLONG(
            x ) ((LONG)(x))
```

Definition at line 241 of file declare.h.

4.19.2.82 Add2Com

```
#define Add2Com(
              x ) { WORD cod[2]; cod[0] = x; cod[1] = 2; AddNtoL(2,cod); }
```

Definition at line 243 of file declare.h.

4.19.2.83 Add3Com

```
#define Add3Com(
              x1,
              x2 ) { WORD cod[3]; cod[0] = x1; cod[1] = 3; cod[2] = x2; AddNtoL(3,cod); }
```

Definition at line 244 of file declare.h.

4.19.2.84 Add4Com

```
#define Add4Com(
              x1,
              x2,
              x3 )
```

Value:

```
{ WORD cod[4]; cod[0] = x1; cod[1] = 4; \
cod[2] = x2; cod[3] = x3; AddNtoL(4,cod); }
```

Definition at line 245 of file declare.h.

4.19.2.85 Add5Com

```
#define Add5Com(
              x1,
              x2,
              x3,
              x4 )
```

Value:

```
{ WORD cod[5]; cod[0] = x1; cod[1] = 5; \
cod[2] = x2; cod[3] = x3; cod[4] = x4; AddNtoL(5,cod); }
```

Definition at line 247 of file declare.h.

4.19.2.86 WantAddPointers

```
#define WantAddPointers(
              x )
```

Value:

```
while((AT.pWorkPointer+(x))>AR.pWorkSize){WORD ***ppp=&AT.pWorkSpace;\
ExpandBuffer((void **)ppp,&AR.pWorkSize,(int)(sizeof(WORD *)));}
```

Definition at line 253 of file declare.h.

4.19.2.87 WantAddLongs

```
#define WantAddLongs(
            x )
```

Value:

```
while((AT.lWorkPointer+(x))>AR.lWorkSize){LONG **ppp=&AT.lWorkSpace;\
ExpandBuffer((void **)ppp,&AR.lWorkSize,sizeof(LONG));}
```

Definition at line 255 of file declare.h.

4.19.2.88 WantAddPositions

```
#define WantAddPositions(
            x )
```

Value:

```
while((AT.posWorkPointer+(x))>AR.posWorkSize){POSITION **ppp=&AT.posWorkSpace;\
ExpandBuffer((void **)ppp,&AR.posWorkSize,sizeof(POSITION));}
```

Definition at line 257 of file declare.h.

4.19.2.89 FORM_INLINE

```
#define FORM_INLINE inline
```

Definition at line 261 of file declare.h.

4.19.2.90 MEMORYMACROS

```
#define MEMORYMACROS
```

Definition at line 269 of file declare.h.

4.19.2.91 TermMalloc

```
#define TermMalloc(
            x ) ( (AT.TermMemTop <= 0 ) ?  TermMallocAddMemory(BHEAD0), AT.TermMemHeap[--AT.↩
TermMemTop]:  AT.TermMemHeap[--AT.TermMemTop] )
```

Definition at line 273 of file declare.h.

4.19.2.92 NumberMalloc

```
#define NumberMalloc(
            x ) ( (AT.NumberMemTop <= 0 ) ?  NumberMallocAddMemory(BHEAD0), AT.NumberMem↩
Heap[--AT.NumberMemTop]:  AT.NumberMemHeap[--AT.NumberMemTop] )
```

Definition at line 274 of file declare.h.

4.19.2.93 CacheNumberMalloc

```
#define CacheNumberMalloc(
            x ) ( (AT.CacheNumberMemTop <= 0 ) ?  CacheNumberMallocAddMemory(BHEAD0), AT.↩
CacheNumberMemHeap[--AT.CacheNumberMemTop]:  AT.CacheNumberMemHeap[--AT.CacheNumberMemTop] )
```

Definition at line 275 of file declare.h.

4.19.2.94 TermFree

```
#define TermFree(
            TermMem,
            x ) AT.TermMemHeap[AT.TermMemTop++] = (WORD *)(TermMem)
```

Definition at line 276 of file declare.h.

4.19.2.95 NumberFree

```
#define NumberFree(
            NumberMem,
            x ) AT.NumberMemHeap[AT.NumberMemTop++] = (UWORD *)(NumberMem)
```

Definition at line 277 of file declare.h.

4.19.2.96 CacheNumberFree

```
#define CacheNumberFree(
            NumberMem,
            x ) AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = (UWORD *)(NumberMem)
```

Definition at line 278 of file declare.h.

4.19.2.97 NestingChecksum

```
#define NestingChecksum( ) (AC.IfLevel + AC.RepLevel + AC.arglevel + AC.insidelevel + AC.↩
termlevel + AC.inexprlevel + AC.dolooplevel)
```

Definition at line 306 of file declare.h.

4.19.2.98 MesNesting

```
#define MesNesting( ) MesPrint("&Illegal nesting of if, repeat, argument, inside, term, inexpression
and do")
```

Definition at line 307 of file declare.h.

4.19.2.99 MarkPolyRatFunDirty

```
#define MarkPolyRatFunDirty(
            T )
```

Value:

```
{if(*T&&AR.PolyFunType==2){WORD *TP,*TT;TT=T++*T;TT-=ABS(TT[-1]);\
TP=T+1;while(TP<TT){if(*TP==AR.PolyFun){TP[2]|=(DIRTYFLAG|MUSTCLEANPRF);}TP+=TP[1];}}}
```

Definition at line 309 of file declare.h.

4.19.2.100 PUSHPREASSIGNLEVEL

```
#define PUSHPREASSIGNLEVEL
```

Value:

```
    AP.PreAssignLevel++; { GETIDENTITY \
    if ( AP.PreAssignLevel >= AP.MaxPreAssignLevel ) { int i; \
        LONG *ap = (LONG *)Malloc1(2*AP.MaxPreAssignLevel*sizeof(LONG *),"PreAssignStack"); \
        for ( i = 0; i < AP.MaxPreAssignLevel; i++ ) ap[i] = AP.PreAssignStack[i]; \
        M_free(AP.PreAssignStack,"PreAssignStack"); \
        AP.MaxPreAssignLevel *= 2; AP.PreAssignStack = ap; \
    } \
    *AT.WorkPointer++ = AP.PreContinuation; AP.PreContinuation = 0; \
    AP.PreAssignStack[AP.PreAssignLevel] = AC.iPointer - AC.iBuffer; }
```

Definition at line 316 of file declare.h.

4.19.2.101 POPPREASSIGNLEVEL

```
#define POPPREASSIGNLEVEL
```

Value:

```
    if ( AP.PreAssignLevel > 0 ) { GETIDENTITY \
    AC.iPointer = AC.iBuffer + AP.PreAssignStack[AP.PreAssignLevel--]; \
    AP.PreContinuation = *--AT.WorkPointer; \
    *AC.iPointer = 0; }
```

Definition at line 326 of file declare.h.

4.19.2.102 EXTERNLOCK

```
#define EXTERNLOCK(
            x )
```

NOTE: We have replaced LOCK(ErrorMessageLock) and UNLOCK(ErrorMessageLock) by MLOCK(Error↩ MessageLock) and MUNLOCK(ErrorMessageLock). They are used for the synchronised output in ParFORM. (TU 28 May 2011)

Definition at line 369 of file declare.h.

4.19.2.103 INILOCK

```
#define INILOCK(
            x )
```

Definition at line 370 of file declare.h.

4.19.2.104 LOCK

```
#define LOCK(
              x )
```

Definition at line 371 of file declare.h.

4.19.2.105 UNLOCK

```
#define UNLOCK(
              x )
```

Definition at line 372 of file declare.h.

4.19.2.106 EXTERNRWLOCK

```
#define EXTERNRWLOCK(
              x )
```

Definition at line 373 of file declare.h.

4.19.2.107 INIRWLOCK

```
#define INIRWLOCK(
              x )
```

Definition at line 374 of file declare.h.

4.19.2.108 RWLOCKR

```
#define RWLOCKR(
              x )
```

Definition at line 375 of file declare.h.

4.19.2.109 RWLOCKW

```
#define RWLOCKW(
              x )
```

Definition at line 376 of file declare.h.

4.19.2.110 UNRWLOCK

```
#define UNRWLOCK(
              x )
```

Definition at line 377 of file declare.h.

4.19.2.111 MLOCK

```
#define MLOCK(
              x )
```

Definition at line 382 of file declare.h.

4.19.2.112 MUNLOCK

```
#define MUNLOCK(
              x )
```

Definition at line 383 of file declare.h.

4.19.2.113 GETIDENTITY

```
#define GETIDENTITY
```

Definition at line 385 of file declare.h.

4.19.2.114 GETBIDENTITY

```
#define GETBIDENTITY
```

Definition at line 386 of file declare.h.

4.19.2.115 M_alloc

```
#define M_alloc(
              x ) malloc((size_t)(x))
```

Definition at line 889 of file declare.h.

4.19.2.116 CompareTerms

```
#define CompareTerms ((COMPARE)AR.CompareRoutine)
```

Definition at line 1359 of file declare.h.

4.19.2.117 FiniShuffle

```
#define FiniShuffle ((FINISHUFFLE)AN.SHvar.finishuf)
```

Definition at line 1360 of file declare.h.

4.19.2.118 DoShtuffle

```
#define DoShtuffle ((DO_UFFLE)AN.SHvar.do_uffle)
```

Definition at line 1361 of file declare.h.

4.19.3 Typedef Documentation

4.19.3.1 WRITEBUFTOEXTCHANNEL

```
typedef int(* WRITEBUFTOEXTCHANNEL) (char *, size_t)
```

Definition at line 1349 of file declare.h.

4.19.3.2 GETCFROMEXTCHANNEL

```
typedef int(* GETCFROMEXTCHANNEL) (VOID)
```

Definition at line 1350 of file declare.h.

4.19.3.3 SETTERMINATORFOREXTERNALCHANNEL

```
typedef int(* SETTERMINATORFOREXTERNALCHANNEL) (char *)
```

Definition at line 1351 of file declare.h.

4.19.3.4 SETKILLMODEFOREXTERNALCHANNEL

```
typedef int(* SETKILLMODEFOREXTERNALCHANNEL) (int, int)
```

Definition at line 1352 of file declare.h.

4.19.3.5 WRITEFILE

```
typedef LONG(* WRITEFILE) (int, UBYTE *, LONG)
```

Definition at line 1353 of file declare.h.

4.19.3.6 COMPARE

```
typedef WORD(* COMPARE) (PHEAD WORD *, WORD *, WORD)
```

Definition at line 1354 of file declare.h.

4.19.3.7 GETTERM

```
typedef WORD(* GETTERM) (PHEAD WORD *)
```

Definition at line 1355 of file declare.h.

4.19.3.8 FINISHUFFLE

```
typedef WORD(* FINISHUFFLE) (PHEAD WORD *)
```

Definition at line 1356 of file declare.h.

4.19.3.9 DO_UFFLE

```
typedef WORD(* DO_UFFLE) (PHEAD WORD *, WORD, WORD, WORD)
```

Definition at line 1357 of file declare.h.

4.19.4 Function Documentation

4.19.4.1 TELLFILE()

```
VOID TELLFILE (
            int handle,
            POSITION * position )  [extern]
```

Definition at line 1334 of file tools.c.

4.19.4.2 StartVariables()

```
VOID StartVariables ( )  [extern]
```

All functions (well, nearly all) are declared here.

Definition at line 828 of file startup.c.

References AddNtoC(), AddRHS(), Compare1(), inicbufs(), FuNcTiOn::name, PF_BroadcastNumber(), PutPreVar(), and FuNcTiOn::symmetric.

Here is the call graph for this function:

4.19.4.3 CodeToLine()

```
UBYTE * CodeToLine (
            WORD number,
            UBYTE * Out )  [extern]
```

Definition at line 669 of file sch.c.

4.19.4.4 AddArrayIndex()

```
UBYTE * AddArrayIndex (
            WORD num,
            UBYTE * out )  [extern]
```

Definition at line 707 of file sch.c.

4.19.4.5 FindInIndex()

```
INDEXENTRY * FindInIndex (
            WORD expr,
            FILEDATA * f,
            WORD par,
            WORD mode )  [extern]
```

Definition at line 2644 of file store.c.

4.19.4.6 NextFileIndex()

```
INDEXENTRY * NextFileIndex (
            POSITION * indexpos )  [extern]
```

Definition at line 2237 of file store.c.

4.19.4.7 PasteTerm()

```
WORD * PasteTerm (
            PHEAD WORD number,
            WORD * accum,
            WORD * position,
            WORD times,
            WORD divby )  [extern]
```

Puts the term at position in the accumulator accum at position 'number+1'. if times $>$ 0 the coefficient of this term is multiplied by times/divby.

Parameters

number	The number of term fragments in accum that should be skipped
accum	The accumulator of term fragments
position	A position in (typically) a compiler buffer from where a (piece of a) term comes.
times	Multiply the result by this
divby	Divide the result by this.

This routine is typically used when we have to replace a (sub)expression pointer by a power of a (sub)expression. This uses mostly a binomial expansion and the new term is the old term multiplied one by one by terms of the new expression. The factors times and divby keep track of the binomial coefficient. Once this is complete, the routine FiniTerm will make the contents of the accumulator into a proper term that still needs to be normalized.

Definition at line 2772 of file proces.c.

Referenced by Generator().

4.19.4.8 StrCopy()

```
UBYTE * StrCopy (
            UBYTE * from,
            UBYTE * to )  [extern]
```

Definition at line 67 of file sch.c.

4.19.4.9 WrtPower()

```
UBYTE * WrtPower (
            UBYTE * Out,
            WORD Power )  [extern]
```

Definition at line 749 of file sch.c.

4.19.4.10 AccumGCD()

```
WORD AccumGCD (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )  [extern]
```

Definition at line 647 of file reken.c.

4.19.4.11 AddArgs()

```
VOID AddArgs (
            PHEAD WORD * s1,
            WORD * s2,
            WORD * m )  [extern]
```

Adds the arguments of two occurrences of the PolyFun.

Parameters

s1	Pointer to the first occurrence.
s2	Pointer to the second occurrence.
m	Pointer to where the answer should be.

Definition at line 2224 of file sort.c.

Referenced by AddPoly(), and MergePatches().

4.19.4.12 AddCoef()

```
WORD AddCoef (
            PHEAD WORD ** ps1,
            WORD ** ps2 )  [extern]
```

Adds the coefficients of the terms *ps1 and *ps2. The problem comes when there is not enough space for a new longer coefficient. First a local solution is tried. If this is not succesfull we need to move terms around. The possibility of a garbage collection should not be ignored, as avoiding this costs very much extra space which is nearly wasted otherwise.

If the return value is zero the terms cancelled.

The resulting term is left in *ps1.

Definition at line 1935 of file sort.c.

References GarbHand(), and NormalModulus().

Referenced by SplitMerge().

Here is the call graph for this function:

4.19.4.13 AddLong()

```
WORD AddLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 705 of file reken.c.

4.19.4.14 AddPLon()

```
WORD AddPLon (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 750 of file reken.c.

4.19.4.15 AddPoly()

```
WORD AddPoly (
            PHEAD WORD ** ps1,
            WORD ** ps2 )  [extern]
```

Routine should be called when S->PolyWise != 0. It points then to the position of AR.PolyFun in both terms.

We add the contents of the arguments of the two polynomials. Special attention has to be given to special arguments. We have to reserve a space equal to the size of one term + the size of the argument of the other. The addition has to be done in this routine because not all objects are reentrant.

Newer addition (12-nov-2007). The PolyFun can have two arguments. In that case S->PolyFlag is 2 and we have to call the routine for adding rational polynomials. We have to be rather careful what happens with: The location of the output The order of the terms in the arguments At first we allow only univariate polynomials in the PolyFun. This restriction will be lifted a.s.a.p.

Parameters

ps1	A pointer to the postion of the first term
ps2	A pointer to the postion of the second term

Returns

If zero the terms cancel. Otherwise the new term is in *ps1.

Definition at line 2062 of file sort.c.

References AddArgs(), and GarbHand().

Referenced by SplitMerge().

Here is the call graph for this function:

4.19.4.16 AddRat()

```
WORD AddRat (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 210 of file reken.c.

4.19.4.17 AddToLine()

```
VOID AddToLine (
            UBYTE * s )  [extern]
```

Definition at line 82 of file sch.c.

4.19.4.18 AddWild()

```
WORD AddWild (
            PHEAD WORD,
            WORD type,
            WORD newnumber )  [extern]
```

Definition at line 1516 of file wildcard.c.

4.19.4.19 BigLong()

```
WORD BigLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb )  [extern]
```

Definition at line 944 of file reken.c.

4.19.4.20 BinomGen()

```
WORD BinomGen (
            PHEAD WORD * term,
            WORD level,
            WORD ** tstops,
            WORD x1,
            WORD x2,
            WORD pow1,
            WORD pow2,
            WORD sign,
            UWORD * coef,
            WORD ncoef )  [extern]
```

Definition at line 320 of file ratio.c.

4.19.4.21 CheckWild()

```
WORD CheckWild (
            PHEAD WORD,
            WORD type,
            WORD newnumber,
            WORD * newval )  [extern]
```

Definition at line 1761 of file wildcard.c.

4.19.4.22 Chisholm()

```
WORD Chisholm (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1963 of file opera.c.

4.19.4.23 CleanExpr()

```
WORD CleanExpr (
            WORD par )  [extern]
```

Definition at line 47 of file execute.c.

4.19.4.24 CleanUp()

```
VOID CleanUp (
            WORD par )  [extern]
```

Definition at line 1619 of file startup.c.

4.19.4.25 ClearWild()

```
VOID ClearWild (
            PHEAD0  )  [extern]
```

Definition at line 1490 of file wildcard.c.

4.19.4.26 CompareFunctions()

```
WORD CompareFunctions (
            WORD * fleft,
            WORD * fright )  [extern]
```

Definition at line 48 of file normal.c.

4.19.4.27 Commute()

```
WORD Commute (
            WORD * fleft,
            WORD * fright )  [extern]
```

Definition at line 112 of file normal.c.

4.19.4.28 DetCommu()

```
WORD DetCommu (
            WORD * terms )  [extern]
```

Definition at line 4345 of file normal.c.

4.19.4.29 DoesCommu()

```
WORD DoesCommu (
            WORD * term )  [extern]
```

Definition at line 4404 of file normal.c.

4.19.4.30 CompArg()

```
int CompArg (
            WORD * s1,
            WORD * s2 )  [extern]
```

Definition at line 3215 of file tools.c.

4.19.4.31 CompCoef()

```
WORD CompCoef (
            WORD * term1,
            WORD * term2 )  [extern]
```

```
Routine takes a1 mod m1 and a2 mod m2 and returns a mod m1*m2 with
a mod m1 = a1 and a mod m2 = a2
```

Chinese remainder: a%(m1∗m2) = q1∗m1+a1 a%(m1∗m2) = q2∗m2+a2 Compute n1 such that (n1∗m1)m2 is one Compute n2 such that (n2∗m2)m1 is one Then (a1∗n2∗m2+a2∗n1∗m1)%(m1∗m2) is a%(m1∗m2)

Definition at line 3037 of file reken.c.

Referenced by Compare1().

4.19.4.32 CompGroup()

```
WORD CompGroup (
            PHEAD WORD,
            WORD ** args,
            WORD * a1,
            WORD * a2,
            WORD num )  [extern]
```

Definition at line 373 of file function.c.

4.19.4.33 Compare1()

```
WORD Compare1 (
            PHEAD WORD * term1,
            WORD * term2,
            WORD level )  [extern]
```

Compares two terms. The answer is: 0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first. Some special precautions may be needed to keep the CompCoef routine from generating overflows, although this is very unlikely in subterms. This routine should not return an error condition.

Originally this routine was called Compare. With the treatment of special polynomials with terms that contain only symbols and the need for extreme speed for the polynomial routines we made a special compare routine and now we store the address of the current compare routine in AR.CompareRoutine and have a macro Compare which makes all existing code work properly and we can just replace the routine on a thread by thread basis (each thread has its own AR struct).

Parameters

term1	First input term
term2	Second input term
level	The sorting level (may influence on the result)

Returns

0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first.

Definition at line 2509 of file sort.c.

References CompCoef().

Referenced by InFunction(), and StartVariables().

Here is the call graph for this function:

4.19.4.34 CountDo()

```
WORD CountDo (
            WORD * term,
            WORD * instruct )  [extern]
```

Definition at line 551 of file reshuf.c.

4.19.4.35 CountFun()

```
WORD CountFun (
            WORD * term,
            WORD * countfun )  [extern]
```

Definition at line 705 of file reshuf.c.

4.19.4.36 DimensionSubterm()

```
WORD DimensionSubterm (
            WORD * subterm )  [extern]
```

Definition at line 866 of file reshuf.c.

4.19.4.37 DimensionTerm()

```
WORD DimensionTerm (
            WORD * term )  [extern]
```

Definition at line 967 of file reshuf.c.

4.19.4.38 DimensionExpression()

```
WORD DimensionExpression (
            PHEAD WORD * expr )  [extern]
```

Definition at line 999 of file reshuf.c.

4.19.4.39 Deferred()

```
WORD Deferred (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Picks up the deferred brackets. These are the bracket contents of which we postpone the reading when we use the 'Keep Brackets' statement. These contents are multiplying the terms just before they are sent to the sorting system. Special attention goes to having it thread-safe We have to lock positioning the file and reading it in a thread specific buffer.

Parameters

term	The term that must be multiplied by the contents of the current bracket
level	The compiler level. This is needed because after multiplying term by term we call Generator again.

Definition at line 4525 of file proces.c.

References Generator(), and InsertTerm().

Referenced by Generator().

Here is the call graph for this function:

4.19.4.40 DeleteStore()

```
WORD DeleteStore (
            WORD par )  [extern]
```

Definition at line 761 of file store.c.

4.19.4.41 DetCurDum()

```
WORD DetCurDum (
            PHEAD WORD * t )  [extern]
```

Definition at line 251 of file reshuf.c.

4.19.4.42 DetVars()

```
VOID DetVars (
            WORD * term,
            WORD par )  [extern]
```

Definition at line 1814 of file store.c.

4.19.4.43 Distribute()

```
WORD Distribute (
            DISTRIBUTE * d,
            WORD first )  [extern]
```

Definition at line 509 of file symmetr.c.

4.19.4.44 DivLong()

```
WORD DivLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc,
            UWORD * d,
            WORD * nd )  [extern]
```

Definition at line 972 of file reken.c.

4.19.4.45 DivRat()

```
WORD DivRat (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 500 of file reken.c.

4.19.4.46 Divvy()

```
WORD Divvy (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )  [extern]
```

Definition at line 172 of file reken.c.

4.19.4.47 DoDelta()

```
WORD DoDelta (
            WORD * t )  [extern]
```

Definition at line 4190 of file normal.c.

4.19.4.48 DoDelta3()

```
WORD DoDelta3 (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1404 of file reshuf.c.

4.19.4.49 TestPartitions()

```
WORD TestPartitions (
            PHEAD WORD * tfun,
            PARTI * parti )  [extern]
```

Definition at line 1606 of file reshuf.c.

4.19.4.50 DoPartitions()

```
WORD DoPartitions (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1721 of file reshuf.c.

4.19.4.51 DoTableExpansion()

```
WORD DoTableExpansion (
            WORD * term,
            WORD level )  [extern]
```

Definition at line 326 of file tables.c.

4.19.4.52 DoDistrib()

```
WORD DoDistrib (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1118 of file reshuf.c.

4.19.4.53 DoShuffle()

```
WORD DoShuffle (
            PHEAD WORD * term,
            WORD level,
            WORD fun,
            WORD option )  [extern]
```

Definition at line 2094 of file reshuf.c.

4.19.4.54 DoPermutations()

```
WORD DoPermutations (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 2006 of file reshuf.c.

4.19.4.55 Shuffle()

```
int Shuffle (
            PHEAD WORD * from1,
            WORD * from2,
            WORD * to )  [extern]
```

Definition at line 2228 of file reshuf.c.

4.19.4.56 FinishShuffle()

```
int FinishShuffle (
            PHEAD WORD * fini )  [extern]
```

Definition at line 2419 of file reshuf.c.

4.19.4.57 DoStuffle()

```
WORD DoStuffle (
            PHEAD WORD * term,
            WORD level,
            WORD fun,
            WORD option )  [extern]
```

Definition at line 2484 of file reshuf.c.

4.19.4.58 Stuffle()

```
int Stuffle (
            PHEAD WORD * from1,
            WORD * from2,
            WORD * to )  [extern]
```

Definition at line 2699 of file reshuf.c.

4.19.4.59 FinishStuffle()

```
int FinishStuffle (
            PHEAD WORD * fini )  [extern]
```

Definition at line 2826 of file reshuf.c.

4.19.4.60 StuffRootAdd()

```
WORD * StuffRootAdd (
            WORD * t1,
            WORD * t2,
            WORD * to )  [extern]
```

Definition at line 2873 of file reshuf.c.

4.19.4.61 TestUse()

```
WORD TestUse (
            WORD * term,
            WORD level )  [extern]
```

Definition at line 1334 of file tables.c.

4.19.4.62 FindTB()

```
DBASE * FindTB (
            UBYTE * name )  [extern]
```

Definition at line 682 of file tables.c.

4.19.4.63 CheckTableDeclarations()

```
int CheckTableDeclarations (
            DBASE * d )  [extern]
```

Definition at line 1109 of file tables.c.

4.19.4.64 Apply()

```
WORD Apply (
            WORD * term,
            WORD level )  [extern]
```

Definition at line 1897 of file tables.c.

4.19.4.65 ApplyExec()

```
int ApplyExec (
            WORD * term,
            int maxtogo,
            WORD level )  [extern]
```

Definition at line 1956 of file tables.c.

4.19.4.66 ApplyReset()

```
WORD ApplyReset (
            WORD level )  [extern]
```

Definition at line 2173 of file tables.c.

4.19.4.67 ReWorkT()

```
VOID ReWorkT (
            WORD * term,
            WORD * funs,
            WORD numfuns )  [extern]
```

Definition at line 1816 of file tables.c.

4.19.4.68 GetIfDollarNum()

```
WORD GetIfDollarNum (
            WORD * ifp,
            WORD * ifstop )  [extern]
```

Definition at line 106 of file if.c.

4.19.4.69 FindVar()

```
int FindVar (
            WORD * v,
            WORD * term )  [extern]
```

Definition at line 173 of file if.c.

4.19.4.70 DoIfStatement()

```
WORD DoIfStatement (
            PHEAD WORD * ifcode,
            WORD * term )  [extern]
```

Definition at line 280 of file if.c.

4.19.4.71 DoOnePow()

```
WORD DoOnePow (
            PHEAD WORD * term,
            WORD power,
            WORD nexp,
            WORD * accum,
            WORD * aa,
            WORD level,
            WORD * freeze )  [extern]
```

Routine gets one power of an expression in the scratch system. If there are more powers needed there will be a recursion.

No attempt is made to use binomials because we have no information about commutating properties.

There is a searching for the contents of brackets if needed. This searching may be rather slow because of the single links.

Parameters

term	is the term we are adding to.
power	is the power of the expression that we need.
nexp	is the number of the expression.
accum	is the accumulator of terms. It accepts the termfragments that are made into a proper term in FiniTerm
aa	points to the start of the entire accumulator. In ∗aa we store the number of term fragments that are in the accumulator.
level	is the current depth in the tree of statements. It is needed to continue to the next operation/substitution with each generated term
freeze	is the pointer to the bracket information that should be matched.

Definition at line 4304 of file proces.c.

References DoOnePow(), FiniTerm(), Generator(), and FiLe::handle.

Referenced by DoOnePow(), and Generator().

Here is the call graph for this function:

4.19.4.72 DoRevert()

```
void DoRevert (
            WORD * fun,
            WORD * tmp )  [extern]
```

Definition at line 4257 of file normal.c.

4.19.4.73 DoSumF1()

```
WORD DoSumF1 (
            PHEAD WORD * term,
            WORD * params,
            WORD replac,
            WORD level )  [extern]
```

Definition at line 395 of file ratio.c.

4.19.4.74 DoSumF2()

```
WORD DoSumF2 (
            PHEAD WORD * term,
            WORD * params,
            WORD replac,
            WORD level )  [extern]
```

Definition at line 520 of file ratio.c.

4.19.4.75 DoTheta()

```
WORD DoTheta (
            PHEAD WORD * t )  [extern]
```

Definition at line 4093 of file normal.c.

4.19.4.76 EndSort()

```
LONG EndSort (
            PHEAD WORD * buffer,
            int par )  [extern]
```

Finishes a sort. At AR.sLevel == 0 the output is to the regular output stream. When AR.sLevel > 0, the parameter par determines the actual output. The AR.sLevel will be popped. All ongoing stages are finished and if the sortfile is open it is closed. The statistics are printed when AR.sLevel == 0 par == 0 Output to the buffer. par == 1 Sort for function arguments. The output will be copied into the buffer. It is assumed that this is in the WorkSpace. par == 2 Sort for $-variable. We return the address of the buffer that contains the output in buffer (treated like WORD **). We first catch the output in a file (unless we can intercept things after the small buffer has been sorted) Then we read from the file into a buffer. Only when par == 0 data compression can be attempted at AT.SS==AT.S0.

Parameters

buffer	buffer for output when needed
par	See above

Returns

If negative: error. If positive: number of words in output.

Definition at line 675 of file sort.c.

References ComPress(), FlushOut(), MergePatches(), PF_EndSort(), PutOut(), SplitMerge(), StageSort(), and WriteStats().

Referenced by InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Processor(), PolyFunMul(), Processor(), TakeArgContent(), TestMatch(), and TestSub().

Here is the call graph for this function:

4.19.4.77 EntVar()

```
WORD EntVar (
            WORD type,
            UBYTE * name,
            WORD x,
            WORD y,
            WORD z,
            WORD d )  [extern]
```

Definition at line 459 of file names.c.

4.19.4.78 EpfCon()

```
WORD EpfCon (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 212 of file opera.c.

4.19.4.79 EpfFind()

```
WORD EpfFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 58 of file opera.c.

4.19.4.80 EpfGen()

```
WORD EpfGen (
            WORD number,
            WORD * inlist,
            WORD * kron,
            WORD * perm,
            WORD sgn )  [extern]
```

Definition at line 282 of file opera.c.

4.19.4.81 EqualArg()

```
WORD EqualArg (
            WORD * parms,
            WORD num1,
            WORD num2 )  [extern]
```

Definition at line 1378 of file reshuf.c.

4.19.4.82 Factorial()

```
int Factorial (
            PHEAD WORD,
            UWORD * a,
            WORD * na )  [extern]
```

Definition at line 3439 of file reken.c.

4.19.4.83 Bernoulli()

```
int Bernoulli (
            WORD n,
            UWORD * a,
            WORD * na )  [extern]
```

Definition at line 3519 of file reken.c.

4.19.4.84 FactorIn()

```
int FactorIn (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 46 of file factor.c.

4.19.4.85 FactorInExpr()

```
int FactorInExpr (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 421 of file factor.c.

4.19.4.86 FindAll()

```
WORD FindAll (
            PHEAD WORD * term,
            WORD * pattern,
            WORD level,
            WORD * par )  [extern]
```

Definition at line 1197 of file pattern.c.

4.19.4.87 FindMulti()

```
WORD FindMulti (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 954 of file findpat.c.

4.19.4.88 FindOnce()

```
WORD FindOnce (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 419 of file findpat.c.

4.19.4.89 FindOnly()

```
WORD FindOnly (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 61 of file findpat.c.

4.19.4.90 FindRest()

```
WORD FindRest (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 1070 of file findpat.c.

4.19.4.91 FindrNumber()

```
WORD FindrNumber (
            WORD n,
            VARRENUM * v )  [extern]
```

Definition at line 2567 of file store.c.

4.19.4.92 FiniTerm()

```
WORD FiniTerm (
            PHEAD WORD * term,
            WORD * accum,
            WORD * termout,
            WORD number,
            WORD tepos )  [extern]
```

Concatenates the contents of the accumulator into a single legal term, which replaces the subexpression pointer

Parameters

term	the input term with the (sub)expression subterm
accum	the accumulator with the term fragments
termout	the location where the output should be written
number	the number of term fragments in the accumulator
tepos	the position of the subterm in term to be replaced

Definition at line 2837 of file proces.c.

Referenced by DoOnePow(), and Generator().

4.19.4.93 FlushOut()

```
WORD FlushOut (
            POSITION * position,
            FILEHANDLE * fi,
            int compr )  [extern]
```

Completes output to an output file and writes the trailing zero.

Parameters

position	The position in the file after writing
fi	The file (or its cache)
compr	Indicates whether there should be compression with gzip.

Returns

Regular conventions (OK -> 0).

Definition at line 1724 of file sort.c.

References FiLe::handle, BrAcKeTiNfO::indexbuffer, and PF_ISendSbuf().

Referenced by EndSort(), MergePatches(), PF_EndSort(), and Processor().

Here is the call graph for this function:

4.19.4.94 FunLevel()

```
VOID FunLevel (
            PHEAD WORD * term )  [extern]
```

Definition at line 130 of file reshuf.c.

4.19.4.95 AdjustRenumScratch()

```
VOID AdjustRenumScratch (
            PHEAD0  )  [extern]
```

Definition at line 505 of file reshuf.c.

4.19.4.96 GcdLong()

```
WORD GcdLong (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 2266 of file reken.c.

4.19.4.97 LcmLong()

```
WORD LcmLong (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 2691 of file reken.c.

4.19.4.98 Generator()

```
WORD Generator (
            PHEAD WORD * term,
            WORD level )  [extern]
```

The heart of the program. Here the expansion tree is set up in one giant recursion

Parameters

term	the input term. may be overwritten
level	the level in the compiler buffer (number of statement)

Returns

Normal conventions (OK = 0).

The routine looks first whether there are unsubstituted (sub)expressions. If so, one of them gets inserted term by term and the new term is used in a renewed call to Generator. If there are no (sub)expressions, the term is normalized, the compiler level is raised (next statement) and the program looks what type of statement this is. If this is a special statement it is either treated on the spot or the appropriate routine is called. If it is a substitution, the pattern matcher is called (TestMatch) which tells whether there was a match. If so we need to call TestSub again to test for (sub)expressions. If we run out of levels, the term receives a final treatment for modulus calculus and/or brackets and is then sent off to the sorting routines.

Definition at line 3034 of file proces.c.

References Deferred(), DoOnePow(), FiniTerm(), Generator(), InFunction(), InsertTerm(), CbUf::lhs, VaRrEnUm::lo, PasteFile(), PasteTerm(), PF_Deferred(), CbUf::Pointer, PolyFunMul(), PrepPoly(), PutPreVar(), CbUf::rhs, StoreTerm(), ReNuMbEr::symb, TestMatch(), and TestSub().

Referenced by Deferred(), DoOnePow(), Generator(), InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Deferred(), PF_Processor(), Processor(), TestMatch(), and TestSub().

Here is the call graph for this function:

4.19.4.99 GetBinom()

```
WORD GetBinom (
            UWORD * a,
            WORD * na,
            WORD i1,
            WORD i2 )  [extern]
```

Definition at line 2657 of file reken.c.

4.19.4.100 GetFromStore()

```
WORD GetFromStore (
            WORD * to,
            POSITION * position,
            RENUMBER renumber,
            WORD * InCompState,
            WORD nexpr )  [extern]
```

Definition at line 1615 of file store.c.

4.19.4.101 GetLong()

```
WORD GetLong (
            UBYTE * s,
            UWORD * a,
            WORD * na )  [extern]
```

Definition at line 1764 of file reken.c.

4.19.4.102 GetMoreTerms()

```
WORD GetMoreTerms (
            WORD * term )  [extern]
```

Definition at line 1412 of file store.c.

4.19.4.103 GetMoreFromMem()

```
WORD GetMoreFromMem (
            WORD * term,
            WORD ** tpoin )  [extern]
```

Definition at line 1508 of file store.c.

4.19.4.104 GetOneTerm()

```
WORD GetOneTerm (
            PHEAD WORD * term,
            FILEHANDLE * fi,
            POSITION * pos,
            int par )  [extern]
```

Definition at line 1255 of file store.c.

4.19.4.105 GetTable()

```
RENUMBER GetTable (
            WORD expr,
            POSITION * position,
            WORD mode )  [extern]
```

Definition at line 2805 of file store.c.

4.19.4.106 GetTerm()

```
WORD GetTerm (
            PHEAD WORD * term )  [extern]
```

Definition at line 980 of file store.c.

4.19.4.107 Glue()

```
WORD Glue (
            PHEAD WORD * term1,
            WORD * term2,
            WORD * sub,
            WORD insert )  [extern]
```

Definition at line 461 of file ratio.c.

4.19.4.108 InFunction()

```
WORD InFunction (
            PHEAD WORD * term,
            WORD * termout )  [extern]
```

Makes the replacement of the subexpression with the number 'replac' in a function argument. Additional information is passed in some of the AR, AN, AT variables.

Parameters

term	The input term
termout	The output term

Returns

0: everything is fine, Negative: fatal, Positive: error.

Special attention should be given to nested functions!

Definition at line 1969 of file proces.c.

References Compare1(), CompareSymbols(), EndSort(), Generator(), LowerSortLevel(), NewSort(), StoreTerm(), and SymbolNormalize().

Referenced by Generator().

Here is the call graph for this function:

4.19.4.109 IniLine()

```
VOID IniLine (
            WORD extrablank ) [extern]
```

Definition at line 267 of file sch.c.

4.19.4.110 InsertTerm()

```
WORD InsertTerm (
            PHEAD WORD * term,
            WORD replac,
            WORD extractbuff,
            WORD * position,
            WORD * termout,
            WORD tepos )  [extern]
```

Puts the terms 'term' and 'position' together into a single legal term in termout. replac is the number of the subexpression that should be replaced. It must be a positive term. When action is needed in the argument of a function all terms in that argument are dealt with recursively. The subexpression is sorted. Only one subexpression is done at a time this way.

Parameters

term	the input term
replac	number of the subexpression pointer to replace
extractbuff	number of the compiler buffer replac refers to
position	position from where to take the term in the compiler buffer
termout	the output term
tepos	offset in term where the subexpression is.

Returns

Normal conventions (OK = 0).

Definition at line 2514 of file proces.c.

Referenced by Deferred(), Generator(), and PF_Deferred().

4.19.4.111 LongToLine()

```
VOID LongToLine (
            UWORD * a,
            WORD na )  [extern]
```

Definition at line 303 of file sch.c.

4.19.4.112 MakeDirty()

```
WORD MakeDirty (
            WORD * term,
            WORD * x,
            WORD par )  [extern]
```

Definition at line 51 of file function.c.

4.19.4.113 MarkDirty()

```
VOID MarkDirty (
            WORD * term,
            WORD flags )  [extern]
```

Definition at line 97 of file function.c.

4.19.4.114 PolyFunDirty()

```
VOID PolyFunDirty (
            PHEAD WORD * term )  [extern]
```

Definition at line 139 of file function.c.

4.19.4.115 PolyFunClean()

```
VOID PolyFunClean (
            PHEAD WORD * term )  [extern]
```

Definition at line 174 of file function.c.

4.19.4.116 MatchE()

```
WORD MatchE (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 56 of file symmetr.c.

4.19.4.117 MatchCy()

```
int MatchCy (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 570 of file symmetr.c.

4.19.4.118 FunMatchCy()

```
int FunMatchCy (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 1066 of file symmetr.c.

4.19.4.119 FunMatchSy()

```
int FunMatchSy (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 1524 of file symmetr.c.

4.19.4.120 MatchArgument()

```
int MatchArgument (
            PHEAD WORD * arg,
            WORD * pat )  [extern]
```

Definition at line 2051 of file symmetr.c.

4.19.4.121 MatchFunction()

```
WORD MatchFunction (
            PHEAD WORD * pattern,
            WORD * interm,
            WORD * wilds )  [extern]
```

Definition at line 817 of file function.c.

4.19.4.122 MergePatches()

```
WORD MergePatches (
            WORD par )  [extern]
```

The general merge routine. Can be used for the large buffer and the file merging. The array S->Patches tells where the patches start S->pStop tells where they end (has to be computed first). The end of a 'line to be merged' is indicated by a zero. If the end is reached without running into a zero or a term runs over the boundary of a patch it is a file merging operation and a new piece from the file is read in.

Parameters

par	If par == 0 the sort is for file -> outputfile. If par == 1 the sort is for large buffer -> sortfile. If par == 2 the sort is for large buffer -> outputfile.

Definition at line 3498 of file sort.c.

References AddArgs(), FlushOut(), FiLe::handle, NormalModulus(), PutIn(), PutOut(), Sflush(), and StageSort().

Referenced by EndSort(), and StoreTerm().

Here is the call graph for this function:

4.19.4.123 MesCerr()

```
WORD MesCerr (
            char * s,
            UBYTE * t )  [extern]
```

Definition at line 824 of file message.c.

4.19.4.124 MesComp()

```
WORD MesComp (
            char * s,
            UBYTE * p,
            UBYTE * q )  [extern]
```

Definition at line 843 of file message.c.

4.19.4.125 Modulus()

```
WORD Modulus (
            WORD * term )  [extern]
```

Definition at line 3092 of file reken.c.

4.19.4.126 MoveDummies()

```
VOID MoveDummies (
            PHEAD WORD * term,
            WORD shift )  [extern]
```

Definition at line 435 of file reshuf.c.

4.19.4.127 MulLong()

```
WORD MulLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 841 of file reken.c.

4.19.4.128 MulRat()

```
WORD MulRat (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 378 of file reken.c.

4.19.4.129 Mully()

```
WORD Mully (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )  [extern]
```

Definition at line 126 of file reken.c.

4.19.4.130 MultDo()

```
WORD MultDo (
            PHEAD WORD * term,
            WORD * pattern )  [extern]
```

Definition at line 1043 of file reshuf.c.

4.19.4.131 NewSort()

```
WORD NewSort (
            PHEAD0  )  [extern]
```

Starts a new sort. At the lowest level this is a 'main sort' with the struct according to the parameters in S0. At higher levels this is a sort for functions, subroutines or dollars. We prepare the arrays and structs.

Returns

Regular convention (OK -> 0)

Definition at line 589 of file sort.c.

Referenced by InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Processor(), PolyFunMul(), Processor(), TakeArgContent(), TestMatch(), and TestSub().

4.19.4.132 ExtraSymbol()

```
WORD ExtraSymbol (
            WORD sym,
            WORD pow,
            WORD nsym,
            WORD * ppsym,
            WORD * ncoef )  [extern]
```

Definition at line 4031 of file normal.c.

4.19.4.133 Normalize()

```
WORD Normalize (
            PHEAD WORD * term )  [extern]
```

Definition at line 187 of file normal.c.

4.19.4.134 BracketNormalize()

```
WORD BracketNormalize (
            PHEAD WORD * term )  [extern]
```

Definition at line 5109 of file normal.c.

4.19.4.135 DropCoefficient()

```
VOID DropCoefficient (
            PHEAD WORD * term )  [extern]
```

Definition at line 4931 of file normal.c.

4.19.4.136 DropSymbols()

```
VOID DropSymbols (
            PHEAD WORD * term )  [extern]
```

Definition at line 4949 of file normal.c.

4.19.4.137 PutInside()

```
int PutInside (
            PHEAD WORD * term,
            WORD * code )  [extern]
```

Definition at line 609 of file index.c.

4.19.4.138 Pack()

```
VOID Pack (
            UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )  [extern]
```

Definition at line 62 of file reken.c.

4.19.4.139 PasteFile()

```
LONG PasteFile (
            PHEAD WORD number,
            WORD * accum,
            POSITION * position,
            WORD ** accfill,
            RENUMBER renumber,
            WORD * freeze,
            WORD nexpr )  [extern]
```

Gets a term from stored expression expr and puts it in the accumulator at position number. It returns the length of the term that came from file.

Parameters

number	number of partial terms to skip in accum
accum	the accumulator
position	file position from where to get the stored term
accfill	returns tail position in accum
renumber	the renumber struct for the variables in the stored expression
freeze	information about if we need only the contents of a bracket
nexpr	the number of the stored expression

Returns

Normal conventions (OK = 0).

Definition at line 2650 of file proces.c.

Referenced by Generator().

4.19.4.140 Permute()

```
WORD Permute (
            PERM * perm,
            WORD first )  [extern]
```

Definition at line 443 of file symmetr.c.

4.19.4.141 PermuteP()

```
WORD PermuteP (
            PERMP * perm,
            WORD first )  [extern]
```

Definition at line 477 of file symmetr.c.

4.19.4.142 PolyFunMul()

```
WORD PolyFunMul (
            PHEAD WORD * term )  [extern]
```

Multiplies the arguments of multiple occurrences of the polyfun. In this routine we do the original PolyFun with one argument only. The PolyRatFun (PolyFunType = 2) is done in a dedicated routine in the file polywrap.cc The new result is written over the old result.

Parameters

term	It contains the input term and later the output.

Returns

Normal conventions (OK = 0).

Definition at line 5035 of file proces.c.

References EndSort(), LowerSortLevel(), NewSort(), and StoreTerm().

Referenced by Generator().

Here is the call graph for this function:

4.19.4.143 PrepPoly()

```
WORD PrepPoly (
            PHEAD WORD * term,
            WORD par )  [extern]
```

Routine checks whether the count of function AR.PolyFun is zero or one. If it is one and it has one scalarlike argument the coefficient of the term is pulled inside the argument. If the count is zero a new function is made with the coefficient as its only argument. The function should be placed at its proper position.

When this function is active it places the PolyFun as last object before the coefficient. This is needed because otherwise the compress algorithm has problems in MergePatches.

The bracket routine should also place the PolyFun at a comparable spot. The compression should then stop at the PolyFun. It doesn't really have to stop when writing the final result but this may be too complicated.

The parameter par tells whether we are at groundlevel or inside a function or dollar variable.

Definition at line 4653 of file proces.c.

Referenced by Generator(), and TestSub().

4.19.4.144 Product()

```
WORD Product (
            UWORD * a,
            WORD * na,
            WORD b )  [extern]
```

Definition at line 1575 of file reken.c.

4.19.4.145 PrtLong()

```
VOID PrtLong (
            UWORD * a,
            WORD na,
            UBYTE * s )  [extern]
```

Definition at line 1702 of file reken.c.

4.19.4.146 PutBracket()

```
WORD PutBracket (
            PHEAD WORD * termin )  [extern]
```

Definition at line 1030 of file execute.c.

4.19.4.147 PutIn()

```
LONG PutIn (
            FILEHANDLE * file,
            POSITION * position,
            WORD * buffer,
            WORD ** take,
            int npat )  [extern]
```

Reads a new patch from position in file handle. It is put at buffer, anything after take is moved forward. This would be part of a term that hasn't been used yet. Because of this there should be some space before the start of the buffer

Parameters

file	The file system from which to read
position	The position from which to read
buffer	The buffer into which to read
take	The unused tail should be moved before the buffer
npat	The number of the patch. Is needed if the information was compressed with gzip, because each patch has its own independent gzip encoding.

Definition at line 1241 of file sort.c.

References FiLe::handle.

Referenced by MergePatches().

4.19.4.148 PutInStore()

```
WORD PutInStore (
            INDEXENTRY * ind,
            WORD num )  [extern]
```

Definition at line 861 of file store.c.

4.19.4.149 PutOut()

```
WORD PutOut (
            PHEAD WORD * term,
            POSITION * position,
            FILEHANDLE * fi,
            WORD ncomp )  [extern]
```

Routine writes one term to file handle at position. It returns the new value of the position.

NOTE: For 'final output' we may have to index the brackets. See the struct BRACKETINDEX. We should maintain: 1: a list with brackets array with the brackets 2: a list of objects of type BRACKETINDEX. It contains array with either pointers or offsets to the list of brackets. starting positions in the file. The index may be tied to a maximum size. In that case we may have to prune the list occasionally.

Parameters

term	The term to be written
position	The position in the file. Afterwards it is updated
fi	The file (or its cache) to which should be written
ncomp	Information about what type of compression should be used

Definition at line 1387 of file sort.c.

References FiLe::handle, and PF_ISendSbuf().

Referenced by EndSort(), MergePatches(), PF_EndSort(), PF_Processor(), and Processor().

Here is the call graph for this function:

4.19.4.150 Quotient()

```
UWORD Quotient (
            UWORD * a,
            WORD * na,
            WORD b )  [extern]
```

Definition at line 1610 of file reken.c.

4.19.4.151 RaisPow()

```
WORD RaisPow (
            PHEAD UWORD * a,
            WORD * na,
            UWORD b )  [extern]
```

Definition at line 1218 of file reken.c.

4.19.4.152 RaisPowCached()

```
VOID RaisPowCached (
            PHEAD WORD x,
            WORD n,
            UWORD ** c,
            WORD * nc )  [extern]
```

Computes power x^n and caches the value

4.19.5 Description

Calculates the power x^n and stores the results for caching purposes. The pointer c (i.e., the pointer, and not what it points to) is overwritten. What it points to should not be overwritten in the calling function.

4.19.6 Notes

- Caching is done in AT.small_power[]. This array is extended if necessary.

Definition at line 1286 of file reken.c.

4.19.6.1 RaisPowMod()

```
WORD RaisPowMod (
            WORD x,
            WORD n,
            WORD m )  [extern]
```

Definition at line 1373 of file reken.c.

4.19.6.2 NormalModulus()

```
int NormalModulus (
            UWORD * a,
            WORD * na )  [extern]
```

Brings a modular representation in the range -p/2 to +p/2 The return value tells whether anything was done. Routine made in the general modulus revamp of July 2008 (JV).

Definition at line 1393 of file reken.c.

Referenced by AddCoef(), and MergePatches().

4.19.6.3 GetModInverses()

```
int GetModInverses (
            WORD m1,
            WORD m2,
            WORD * im1,
            WORD * im2 )  [extern]
```

Input m1 and m2, which are relative prime. determines a∗m1+b∗m2 = 1 (and 1 is the gcd of m1 and m2) then a∗m1 = 1 mod m2 and hence im1 = a. and b∗m2 = 1 mod m1 and hence im2 = b. Set m1 = 0∗m1+1∗m2 = a1∗m1+b1∗m2 m2 = 1∗m1+0∗m2 = a2∗m1+b2∗m2 If everything is OK, the return value is zero

Definition at line 1466 of file reken.c.

Referenced by MakeDollarMod(), MakeInverses(), MakeMod(), TakeContent(), and TakeSymbolContent().

4.19.6.4 GetLongModInverses()

```
int GetLongModInverses (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * ia,
            WORD * nia,
            UWORD * ib,
            WORD * nib )  [extern]
```

Definition at line 1498 of file reken.c.

4.19.6.5 RatToLine()

```
VOID RatToLine (
            UWORD * a,
            WORD na )  [extern]
```

Definition at line 337 of file sch.c.

4.19.6.6 RatioFind()

```
WORD RatioFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 62 of file ratio.c.

4.19.6.7 RatioGen()

```
WORD RatioGen (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 170 of file ratio.c.

4.19.6.8 ReNumber()

```
WORD ReNumber (
            PHEAD WORD * term )  [extern]
```

Definition at line 80 of file reshuf.c.

4.19.6.9 ReadSnum()

```
WORD ReadSnum (
            UBYTE ** p )  [extern]
```

Definition at line 1881 of file tools.c.

4.19.6.10 Remain10()

```
WORD Remain10 (
            UWORD * a,
            WORD * na )  [extern]
```

Definition at line 1649 of file reken.c.

4.19.6.11 Remain4()

```
WORD Remain4 (
            UWORD * a,
            WORD * na )  [extern]
```

Definition at line 1676 of file reken.c.

4.19.6.12 ResolveSet()

```
WORD ResolveSet (
            PHEAD WORD * from,
            WORD * to,
            WORD * subs )  [extern]
```

Definition at line 1333 of file wildcard.c.

4.19.6.13 ScanFunctions()

```
WORD ScanFunctions (
            PHEAD WORD * inpat,
            WORD * inter,
            WORD par )  [extern]
```

Definition at line 1604 of file function.c.

4.19.6.14 SeekScratch()

```
VOID SeekScratch (
            FILEHANDLE * fi,
            POSITION * pos )  [extern]
```

Definition at line 64 of file store.c.

4.19.6.15 SetEndScratch()

```
VOID SetEndScratch (
            FILEHANDLE * f,
            POSITION * position )  [extern]
```

Definition at line 75 of file store.c.

4.19.6.16 SetEndHScratch()

```
VOID SetEndHScratch (
            FILEHANDLE * f,
            POSITION * position )  [extern]
```

Definition at line 89 of file store.c.

4.19.6.17 Sflush()

```
WORD Sflush (
            FILEHANDLE * fi )  [extern]
```

Puts the contents of a buffer to output Only to be used when there is a single patch in the large buffer.

Parameters

fi	The filesystem (or its cache) to which the patch should be written

Definition at line 1301 of file sort.c.

References FiLe::handle.

Referenced by MergePatches().

4.19.6.18 Simplify()

```
WORD Simplify (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD * nb )  [extern]
```

Definition at line 530 of file reken.c.

4.19.6.19 SortWild()

```
WORD SortWild (
            WORD * w,
            WORD nw )  [extern]
```

Sorts the wildcard entries in the parameter w. Double entries are removed. Full space taken is nw words. Routine serves for the reading of wildcards in the compiler. The entries come in the format: (type,4,number,0) in which the zero is reserved for the future replacement of 'number'.

Parameters

w	buffer with wildcard entries.
nw	number of wildcard entries.

Returns

Normal conventions (OK -> 0)

Definition at line 4444 of file sort.c.

4.19.6.20 LocateBase()

```
FILE * LocateBase (
            char ** name,
            char ** newname )  [extern]
```

Definition at line 225 of file minos.c.

4.19.6.21 SplitMerge()

```
VOID SplitMerge (
            PHEAD WORD ** Pointer,
            LONG number )  [extern]
```

Algorithm by J.A.M.Vermaseren (31-7-1988)

Note that AN.SplitScratch and AN.InScratch are used also in GarbHand

Merge sort in memory. The input is an array of pointers. Sorting is done recursively by dividing the array in two equal parts and calling SplitMerge for each. When the parts are small enough we can do the compare and take the appropriate action. An addition is that we look for 'runs'. Sequences that are already ordered. This happens a lot when there is very little action in a module. This made FORM faster by a few percent.

Parameters

Pointer	The array of pointers to the terms to be sorted.
number	The number of pointers in Pointer.

The terms are supposed to be sitting in the small buffer and there is supposed to be an extension to this buffer for

when there are two terms that should be added and the result takes more space than each of the original terms. The notation guarantees that the result never needs more space than the sum of the spaces of the original terms.

Definition at line 3207 of file sort.c.

References AddCoef(), AddPoly(), and SplitMerge().

Referenced by EndSort(), SplitMerge(), and StoreTerm().

Here is the call graph for this function:

4.19.6.22 StoreTerm()

```
WORD StoreTerm (
            PHEAD WORD * term )  [extern]
```

The central routine to accept terms, store them and keep things at least partially sorted. A call to EndSort will then complete storing and sorting.

Parameters

term	The term to be stored

Returns

Regular return conventions (OK -> 0)

Definition at line 4246 of file sort.c.

References ComPress(), MergePatches(), SplitMerge(), and WriteStats().

Referenced by Generator(), InFunction(), PF_Processor(), PolyFunMul(), Processor(), TakeArgContent(), and TestSub().

Here is the call graph for this function:

4.19.6.23 SubPLon()

```
VOID SubPLon (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )  [extern]
```

Definition at line 803 of file reken.c.

4.19.6.24 Substitute()

```
VOID Substitute (
            PHEAD WORD * term,
            WORD * pattern,
            WORD power )  [extern]
```

Definition at line 640 of file pattern.c.

4.19.6.25 SymFind()

```
WORD SymFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 633 of file function.c.

4.19.6.26 SymGen()

```
WORD SymGen (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 518 of file function.c.

4.19.6.27 Symmetrize()

```
WORD Symmetrize (
            PHEAD WORD * func,
            WORD * Lijst,
            WORD ngroups,
            WORD gsize,
            WORD type )  [extern]
```

Definition at line 213 of file function.c.

4.19.6.28 FullSymmetrize()

```
int FullSymmetrize (
            PHEAD WORD * fun,
            int type )  [extern]
```

Definition at line 473 of file function.c.

4.19.6.29 TakeModulus()

```
WORD TakeModulus (
            UWORD * a,
            WORD * na,
            UWORD * cmodvec,
            WORD ncmod,
            WORD par )  [extern]
```

Definition at line 3139 of file reken.c.

4.19.6.30 TakeNormalModulus()

```
WORD TakeNormalModulus (
            UWORD * a,
            WORD * na,
            UWORD * c,
            WORD nc,
            WORD par )  [extern]
```

Definition at line 3311 of file reken.c.

4.19.6.31 TalToLine()

```
VOID TalToLine (
            UWORD x )  [extern]
```

Definition at line 533 of file sch.c.

4.19.6.32 TenVec()

```
WORD TenVec (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 2312 of file opera.c.

4.19.6.33 TenVecFind()

```
WORD TenVecFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 2178 of file opera.c.

4.19.6.34 TermRenumber()

```
WORD TermRenumber (
            WORD * term,
            RENUMBER renumber,
            WORD nexpr )  [extern]
```

!! WORD ∗memterm=term; static LONG ctrap=0; !!!

!! ctrap++; !!!

Definition at line 2407 of file store.c.

References ReNuMbEr::func, ReNuMbEr::funnum, ReNuMbEr::indi, ReNuMbEr::indnum, ReNuMbEr::symb, ReNuMbEr::symnum, TermRenumber(), ReNuMbEr::vecnum, and ReNuMbEr::vect.

Referenced by TermRenumber().

Here is the call graph for this function:

4.19.6.35 PutInVflags()

```
VOID PutInVflags (
            WORD nexpr )  [extern]
```

Definition at line 551 of file execute.c.

4.19.6.36 TestMatch()

```
WORD TestMatch (
            PHEAD WORD * term,
            WORD * level )  [extern]
```

This routine governs the pattern matching. If it decides that a substitution should be made, this can be either the insertion of a right hand side (C->rhs) or the automatic generation of terms as a result of an operation (like trace). The object to be replaced is removed from term and a subexpression pointer is inserted. If the substitution is made more than once there can be more subexpression pointers. Its number is positive as it corresponds to the level at which the C->rhs can be found in the compiler output. The subexpression pointer contains the wildcard substitution information. The power is found in ∗AT.TMout. For operations the subexpression pointer is negative and corresponds to an address in the array AT.TMout. In this array are then the instructions for the routine to be called and its number in the array 'Operations' The format is here: length,functionnumber,length-2 parameters

There is a certain complexity wrt repeat levels. Another complication is the poking of the wildcard values in the subexpression prototype in the compiler buffer. This was how things were done in the past with sequential FORM, but with the advent of TFORM this cannot be maintained. Now, for TFORM we make a copy of it. 7-may-2008 (JV): We cannot yet guarantee that this has been done 100% correctly. There are errors that occur in TFORM only and that may indicate problems.

Definition at line 97 of file pattern.c.

References AddRHS(), EndSort(), Generator(), CbUf::lhs, LowerSortLevel(), and NewSort().

Referenced by Generator().

Here is the call graph for this function:

4.19.6.37 TestSub()

```
WORD TestSub (
            PHEAD WORD * term,
            WORD level )  [extern]
```

TestSub hunts for subexpression pointers. If one is found its power is given in AN.TeSuOut. and the returnvalue is 'expressionnumber'. If the expression number is negative it is an expression on disk.

In addition this routine tries to locate subexpression pointers in functions. It also notices that action must be taken with any of the special functions.

Parameters

term	The term in which TestSub hunts for potential action
level	The number of the 'level' in the compiler buffer.

Returns

The number of the (sub)expression that was encountered.

Other values that are returned are in AN.TeSuOut, AR.TePos, AT.TMbuff, AN.TeInFun, AN.Frozen, AT.TMaddr

The level in the compiler buffer is more or less the number of the statement in the module. Hence it refers to the element in the lhs array.

This routine is one of the most important routines in FORM.

Definition at line 680 of file proces.c.

References AddRHS(), TaBlEs::bounds, TaBlEs::bufnum, CbUf::CanCommu, CompareSymbols(), DoubleCbuffer(), EndSort(), Generator(), LowerSortLevel(), MiNmAx::maxi, MiNmAx::mini, TaBlEs::mm, NewSort(), TaBlEs::numind, TaBlEs::pattern, CbUf::Pointer, PrepPoly(), TaBlEs::prototype, CbUf::rhs, MiNmAx::size, TaBlEs::sparse, StoreTerm(), TaBlEs::strict, TaBlEs::tablepointers, TestSub(), and CbUf::Top.

Referenced by Generator(), and TestSub().

Here is the call graph for this function:

4.19.6.38 TimeCPU()

```
LONG TimeCPU (
            WORD par )  [extern]
```

Returns the CPU time.

Parameters

par	If zero, the CPU time will be reset to 0.

Returns

The CPU time in milliseconds.

Definition at line 3418 of file tools.c.

Referenced by PF_GetSlaveTimes(), PF_Processor(), and WriteStats().

4.19.6.39 TimeChildren()

```
LONG TimeChildren (
            WORD par )  [extern]
```

Definition at line 3400 of file tools.c.

4.19.6.40 TimeWallClock()

```
LONG TimeWallClock (
            WORD par )  [extern]
```

Returns the wall-clock time.

Parameters

par	If zero, the wall-clock time will be reset to 0.

Returns

The wall-clock time in centiseconds.

Definition at line 3377 of file tools.c.

Referenced by DoCheckpoint(), DoRecovery(), and WriteStats().

4.19.6.41 ToStorage()

```
WORD ToStorage (
            EXPRESSIONS e,
            POSITION * length )  [extern]
```

Definition at line 2001 of file store.c.

4.19.6.42 TokenToLine()

```
VOID TokenToLine (
            UBYTE * s )  [extern]
```

Definition at line 562 of file sch.c.

4.19.6.43 Trace4()

```
WORD Trace4 (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 694 of file opera.c.

4.19.6.44 Trace4Gen()

```
WORD Trace4Gen (
            PHEAD TRACES * t,
            WORD number )  [extern]
```

Definition at line 856 of file opera.c.

4.19.6.45 Trace4no()

```
WORD Trace4no (
            WORD number,
            WORD * kron,
            TRACES * t )  [extern]
```

Definition at line 418 of file opera.c.

4.19.6.46 TraceFind()

```
WORD TraceFind (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 1853 of file opera.c.

4.19.6.47 TraceN()

```
WORD TraceN (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 1382 of file opera.c.

4.19.6.48 TraceNgen()

```
WORD TraceNgen (
            PHEAD TRACES * t,
            WORD number )  [extern]
```

Definition at line 1446 of file opera.c.

4.19.6.49 TraceNno()

```
WORD TraceNno (
            WORD number,
            WORD * kron,
            TRACES * t )  [extern]
```

Definition at line 1341 of file opera.c.

4.19.6.50 Traces()

```
WORD Traces (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )  [extern]
```

Definition at line 1831 of file opera.c.

4.19.6.51 Trick()

```
WORD Trick (
            WORD * in,
            TRACES * t )  [extern]
```

Definition at line 331 of file opera.c.

4.19.6.52 TryDo()

```
WORD TryDo (
            PHEAD WORD * term,
            WORD * pattern,
            WORD level )  [extern]
```

Definition at line 1071 of file reshuf.c.

4.19.6.53 UnPack()

```
VOID UnPack (
            UWORD * a,
            WORD na,
            WORD * denom,
            WORD * numer )  [extern]
```

Definition at line 100 of file reken.c.

4.19.6.54 VarStore()

```
WORD VarStore (
            UBYTE * s,
            WORD n,
            WORD name,
            WORD namesize )  [extern]
```

Definition at line 2349 of file store.c.

4.19.6.55 WildFill()

```
WORD WildFill (
            PHEAD WORD * to,
            WORD * from,
            WORD * sub )  [extern]
```

Definition at line 65 of file wildcard.c.

4.19.6.56 WriteOne()

```
WORD WriteOne (
            UBYTE * name,
            int alreadyinline,
            int nosemi )  [extern]
```

Definition at line 2634 of file sch.c.

4.19.6.57 WriteArgument()

```
VOID WriteArgument (
            WORD * t )  [extern]
```

Definition at line 1395 of file sch.c.

4.19.6.58 WriteExpression()

```
WORD WriteExpression (
            WORD * terms,
            LONG ltot )  [extern]
```

Definition at line 2383 of file sch.c.

4.19.6.59 WriteInnerTerm()

```
WORD WriteInnerTerm (
            WORD * term,
            WORD first )  [extern]
```

Definition at line 1903 of file sch.c.

4.19.6.60 WriteStats()

```
VOID WriteStats (
            POSITION * plspace,
            WORD par )  [extern]
```

Writes the statistics.

Parameters

plspace	The size in bytes currently occupied
par	par = 0 after a splitmerge. par = 1 after merge to sortfile. par = 2 after the sort

current expression is to be found in AR.CurExpr. terms are in S->TermsLeft. S->GenTerms.

Definition at line 91 of file sort.c.

References TimeCPU(), and TimeWallClock().

Referenced by EndSort(), PF_Processor(), and StoreTerm().

Here is the call graph for this function:

4.19.6.61 WriteSubTerm()

```
WORD WriteSubTerm (
            WORD * sterm,
            WORD first )  [extern]
```

Definition at line 1510 of file sch.c.

4.19.6.62 WriteTerm()

```
WORD WriteTerm (
            WORD * term,
            WORD * lbrac,
            WORD first,
            WORD prtf,
            WORD br )  [extern]
```

Definition at line 2061 of file sch.c.

4.19.6.63 execarg()

```
WORD execarg (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 56 of file argument.c.

4.19.6.64 execterm()

```
WORD execterm (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1724 of file argument.c.

4.19.6.65 SpecialCleanup()

```
VOID SpecialCleanup (
            PHEAD0  )  [extern]
```

Definition at line 1539 of file execute.c.

4.19.6.66 SetMods()

```
void SetMods ( )  [extern]
```

Definition at line 1553 of file execute.c.

4.19.6.67 UnSetMods()

```
void UnSetMods ( )  [extern]
```

Definition at line 1571 of file execute.c.

4.19.6.68 DoExecute()

```
WORD DoExecute (
            WORD par,
            WORD skip )  [extern]
```

Definition at line 599 of file execute.c.

4.19.6.69 SetScratch()

```
VOID SetScratch (
            FILEHANDLE * f,
            POSITION * position )  [extern]
```

Definition at line 115 of file store.c.

4.19.6.70 Warning()

```
VOID Warning (
            char * s )  [extern]
```

Definition at line 790 of file message.c.

4.19.6.71 HighWarning()

```
VOID HighWarning (
            char * s )  [extern]
```

Definition at line 802 of file message.c.

4.19.6.72 SpareTable()

```
int SpareTable (
            TABLES TT )  [extern]
```

Definition at line 642 of file tables.c.

4.19.6.73 strDup1()

```
UBYTE * strDup1 (
            UBYTE * instring,
            char * ifwrong )  [extern]
```

Definition at line 1812 of file tools.c.

4.19.6.74 Malloc()

```
VOID * Malloc (
            LONG size )  [extern]
```

Definition at line 2135 of file tools.c.

4.19.6.75 Malloc1()

```
VOID * Malloc1 (
            LONG size,
            const char * messageifwrong )  [extern]
```

Definition at line 2217 of file tools.c.

4.19.6.76 DoTail()

```
int DoTail (
            int argc,
            UBYTE ** argv )  [extern]
```

Definition at line 217 of file startup.c.

4.19.6.77 PutPreVar()

```
int PutPreVar (
            UBYTE * name,
            UBYTE * value,
            UBYTE * args,
            int mode )  [extern]
```

Inserts/Updates a preprocessor variable in the name administration.

Parameters

name	Character string with the variable name.
value	Character string with a possible value. Special case: if this argument is zero, then we have no value. Note: This is different from having an empty argument! This should only occur when the name starts with a ?
args	Character string with possible arguments.
mode	=0: always create a new name entry, =1: try to do a redefinition if possible.

Returns

Index of used entry in name list.

Definition at line 638 of file pre.c.

References pReVaR::argnames, pReVaR::name, pReVaR::nargs, pReVaR::value, and pReVaR::wildarg.

Referenced by Generator(), PF_BroadcastRedefinedPreVars(), StartVariables(), and TheDefine().

4.19.6.78 Error0()

```
VOID Error0 (
            char * s )  [extern]
```

Definition at line 56 of file message.c.

4.19.6.79 Error1()

```
VOID Error1 (
            char * s,
            UBYTE * t )  [extern]
```

Definition at line 67 of file message.c.

4.19.6.80 Error2()

```
VOID Error2 (
            char * s1,
            char * s2,
            UBYTE * t )  [extern]
```

Definition at line 78 of file message.c.

4.19.6.81 ReadFromStream()

```
UBYTE ReadFromStream (
            STREAM * stream )  [extern]
```

Definition at line 151 of file tools.c.

4.19.6.82 GetFromStream()

```
UBYTE GetFromStream (
            STREAM * stream )  [extern]
```

Definition at line 255 of file tools.c.

4.19.6.83 LookInStream()

```
UBYTE LookInStream (
            STREAM * stream )  [extern]
```

Definition at line 278 of file tools.c.

4.19.6.84 OpenStream()

```
STREAM * OpenStream (
            UBYTE * name,
            int type,
            int prevarmode,
            int raiselow )  [extern]
```

Definition at line 290 of file tools.c.

4.19.6.85 LocateFile()

```
int LocateFile (
            UBYTE ** name,
            int type )  [extern]
```

Definition at line 513 of file tools.c.

4.19.6.86 CloseStream()

```
STREAM * CloseStream (
            STREAM * stream )  [extern]
```

Definition at line 600 of file tools.c.

4.19.6.87 PositionStream()

```
VOID PositionStream (
            STREAM * stream,
            LONG position )  [extern]
```

Definition at line 751 of file tools.c.

4.19.6.88 ReverseStatements()

```
int ReverseStatements (
            STREAM * stream )  [extern]
```

Definition at line 783 of file tools.c.

4.19.6.89 ProcessOption()

```
int ProcessOption (
            UBYTE * s1,
            UBYTE * s2,
            int filetype )  [extern]
```

Definition at line 175 of file setfile.c.

4.19.6.90 Terminate()

```
VOID Terminate (
            int errorcode )  [extern]
```

Definition at line 1696 of file startup.c.

4.19.6.91 GetNode()

```
NAMENODE * GetNode (
            NAMETREE * nametree,
            UBYTE * name )  [extern]
```

Definition at line 49 of file names.c.

4.19.6.92 AddName()

```
int AddName (
            NAMETREE * nametree,
            UBYTE * name,
            WORD type,
            WORD number,
            int * nodenum )  [extern]
```

Definition at line 71 of file names.c.

4.19.6.93 GetName()

```
int GetName (
            NAMETREE * nametree,
            UBYTE * name,
            WORD * number,
            int par )  [extern]
```

Definition at line 252 of file names.c.

4.19.6.94 GetLastExprName()

```
int GetLastExprName (
            UBYTE * name,
            WORD * number )  [extern]
```

Definition at line 340 of file names.c.

4.19.6.95 GetAutoName()

```
int GetAutoName (
            UBYTE * name,
            WORD * number )  [extern]
```

Definition at line 381 of file names.c.

4.19.6.96 GetVar()

```
int GetVar (
            UBYTE * name,
            WORD * type,
            WORD * number,
            int wantedtype,
            int par )  [extern]
```

Definition at line 426 of file names.c.

4.19.6.97 MakeDubious()

```
int MakeDubious (
            NAMETREE * nametree,
            UBYTE * name,
            WORD * number )  [extern]
```

Definition at line 2503 of file names.c.

4.19.6.98 GetOName()

```
int GetOName (
            NAMETREE * nametree,
            UBYTE * name,
            WORD * number,
            int par )  [extern]
```

Definition at line 362 of file names.c.

4.19.6.99 DumpTree()

```
VOID DumpTree (
            NAMETREE * nametree )  [extern]
```

Definition at line 504 of file names.c.

4.19.6.100 DumpNode()

```
VOID DumpNode (
            NAMETREE * nametree,
            WORD node,
            WORD depth )  [extern]
```

Definition at line 517 of file names.c.

4.19.6.101 LinkTree()

```
VOID LinkTree (
            NAMETREE * tree,
            WORD offset,
            WORD numnodes )  [extern]
```

Definition at line 686 of file names.c.

4.19.6.102 CopyTree()

```
VOID CopyTree (
            NAMETREE * newtree,
            NAMETREE * oldtree,
            WORD node,
            WORD par )  [extern]
```

Definition at line 598 of file names.c.

4.19.6.103 CompactifyTree()

```
int CompactifyTree (
            NAMETREE * nametree,
            WORD par )  [extern]
```

Definition at line 536 of file names.c.

4.19.6.104 FreeNameTree()

```
VOID FreeNameTree (
            NAMETREE * n )  [extern]
```

Definition at line 736 of file names.c.

4.19.6.105 AddExpression()

```
int AddExpression (
            UBYTE * name,
            int x,
            int y )  [extern]
```

Definition at line 2554 of file names.c.

4.19.6.106 AddSymbol()

```
int AddSymbol (
            UBYTE * name,
            int minpow,
            int maxpow,
            int cplx,
            int dim )  [extern]
```

Definition at line 822 of file names.c.

4.19.6.107 AddDollar()

```
int AddDollar (
            UBYTE * name,
            WORD type,
            WORD * start,
            LONG size )  [extern]
```

Definition at line 2412 of file names.c.

4.19.6.108 ReplaceDollar()

```
int ReplaceDollar (
            WORD number,
            WORD newtype,
            WORD * newstart,
            LONG newsize )  [extern]
```

Definition at line 2456 of file names.c.

4.19.6.109 DollarRaiseLow()

```
int DollarRaiseLow (
            UBYTE * name,
            LONG value )  [extern]
```

Definition at line 2458 of file dollar.c.

4.19.6.110 AddVector()

```
int AddVector (
            UBYTE * name,
            int cplx,
            int dim )  [extern]
```

Definition at line 1143 of file names.c.

4.19.6.111 AddDubious()

```
int AddDubious (
            UBYTE * name )  [extern]
```

Definition at line 2489 of file names.c.

4.19.6.112 AddIndex()

```
int AddIndex (
            UBYTE * name,
            int dim,
            int dim4 )  [extern]
```

Definition at line 990 of file names.c.

4.19.6.113 DoDimension()

```
UBYTE * DoDimension (
            UBYTE * s,
            int * dim,
            int * dim4 )  [extern]
```

Definition at line 1062 of file names.c.

4.19.6.114 AddFunction()

```
int AddFunction (
            UBYTE * name,
            int comm,
            int istensor,
            int cplx,
            int symprop,
            int dim,
            int argmax,
            int argmin )  [extern]
```

Definition at line 1225 of file names.c.

4.19.6.115 CoCommuteInSet()

```
int CoCommuteInSet (
            UBYTE * s )  [extern]
```

Definition at line 1255 of file names.c.

4.19.6.116 CoFunction()

```
int CoFunction (
            UBYTE * s,
            int comm,
            int istensor )  [extern]
```

Definition at line 1345 of file names.c.

4.19.6.117 TestName()

```
int TestName (
            UBYTE * name )  [extern]
```

Definition at line 3063 of file names.c.

4.19.6.118 AddSet()

```
int AddSet (
            UBYTE * name,
            WORD dim )  [extern]
```

Definition at line 1997 of file names.c.

4.19.6.119 DoElements()

```
int DoElements (
            UBYTE * s,
            SETS set,
            UBYTE * name )  [extern]
```

Definition at line 2029 of file names.c.

4.19.6.120 DoTempSet()

```
int DoTempSet (
            UBYTE * from,
            UBYTE * to )  [extern]
```

Definition at line 2290 of file names.c.

4.19.6.121 NameConflict()

```
int NameConflict (
            int type,
            UBYTE * name )  [extern]
```

Definition at line 2538 of file names.c.

4.19.6.122 OpenFile()

```
int OpenFile (
            char * name )  [extern]
```

Definition at line 911 of file tools.c.

4.19.6.123 OpenAddFile()

```
int OpenAddFile (
            char * name )  [extern]
```

Definition at line 930 of file tools.c.

4.19.6.124 ReOpenFile()

```
int ReOpenFile (
            char * name )  [extern]
```

Definition at line 951 of file tools.c.

4.19.6.125 CreateFile()

```
int CreateFile (
            char * name )  [extern]
```

Definition at line 971 of file tools.c.

4.19.6.126 CreateLogFile()

```
int CreateLogFile (
            char * name )  [extern]
```

Definition at line 988 of file tools.c.

4.19.6.127 CloseFile()

```
VOID CloseFile (
            int handle )  [extern]
```

Definition at line 1006 of file tools.c.

4.19.6.128 CopyFile()

```
int CopyFile (
            char * source,
            char * dest )  [extern]
```

Copies a file with name ∗source to a file named ∗dest. The involved files must not be open. Returns non-zero if an error occurred. Uses if possible the combined large and small sorting buffers as cache.

Definition at line 1029 of file tools.c.

Referenced by DoRecovery().

4.19.6.129 ReadFile()

```
LONG ReadFile (
            int handle,
            UBYTE * buffer,
            LONG size )  [extern]
```

Definition at line 1145 of file tools.c.

4.19.6.130 ReadPosFile()

```
LONG ReadPosFile (
            PHEAD FILEHANDLE * fi,
            UBYTE * buffer,
            LONG size,
            POSITION * pos )  [extern]
```

Definition at line 1195 of file tools.c.

4.19.6.131 WriteFileToFile()

```
LONG WriteFileToFile (
            int handle,
            UBYTE * buffer,
            LONG size )  [extern]
```

Definition at line 1261 of file tools.c.

4.19.6.132 SeekFile()

```
VOID SeekFile (
            int handle,
            POSITION * offset,
            int origin )  [extern]
```

Definition at line 1299 of file tools.c.

4.19.6.133 TellFile()

```
LONG TellFile (
            int handle )  [extern]
```

Definition at line 1324 of file tools.c.

4.19.6.134 FlushFile()

```
void FlushFile (
            int handle )  [extern]
```

Definition at line 1348 of file tools.c.

4.19.6.135 GetPosFile()

```
int GetPosFile (
            int handle,
            fpos_t * pospointer )  [extern]
```

Definition at line 1362 of file tools.c.

4.19.6.136 SetPosFile()

```
int SetPosFile (
            int handle,
            fpos_t * pospointer )  [extern]
```

Definition at line 1376 of file tools.c.

4.19.6.137 SynchFile()

```
VOID SynchFile (
            int handle )  [extern]
```

Definition at line 1396 of file tools.c.

4.19.6.138 TruncateFile()

```
VOID TruncateFile (
            int handle )  [extern]
```

Definition at line 1418 of file tools.c.

4.19.6.139 GetChannel()

```
int GetChannel (
            char * name )  [extern]
```

Definition at line 1437 of file tools.c.

4.19.6.140 GetAppendChannel()

```
int GetAppendChannel (
            char * name )  [extern]
```

Definition at line 1468 of file tools.c.

4.19.6.141 CloseChannel()

```
int CloseChannel (
            char * name )  [extern]
```

Definition at line 1498 of file tools.c.

4.19.6.142 findcommand()

```
KEYWORD * findcommand (
            UBYTE * in )  [extern]
```

Definition at line 312 of file compiler.c.

4.19.6.143 inicbufs()

```
int inicbufs (
            VOID  )  [extern]
```

Creates a new compiler buffer and returns its ID number.

Returns

The ID number for the new compiler buffer.

Definition at line 47 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::dimension, CbUf::lhs, CbUf::numdum, CbUf::NumTerms, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by AddRHS(), and StartVariables().

4.19.6.144 FromList()

```
VOID * FromList (
            LIST * L )  [extern]
```

Definition at line 2693 of file tools.c.

4.19.6.145 From0List()

```
VOID * From0List (
            LIST * L )  [extern]
```

Definition at line 2719 of file tools.c.

4.19.6.146 FromVarList()

```
VOID * FromVarList (
            LIST * L )  [extern]
```

Definition at line 2748 of file tools.c.

4.19.6.147 DoubleList()

```
int DoubleList (
            VOID *** lijst,
            int * oldsize,
            int objectsize,
            char * nameoftype )  [extern]
```

Definition at line 2790 of file tools.c.

4.19.6.148 DoubleLList()

```
int DoubleLList (
            VOID *** lijst,
            LONG * oldsize,
            int objectsize,
            char * nameoftype )  [extern]
```

Definition at line 2842 of file tools.c.

4.19.6.149 DoubleBuffer()

```
void DoubleBuffer (
            void ** start,
            void ** stop,
            int size,
            char * text )  [extern]
```

Definition at line 2890 of file tools.c.

4.19.6.150 ExpandBuffer()

```
void ExpandBuffer (
            void ** buffer,
            LONG * oldsize,
            int type )  [extern]
```

Definition at line 2915 of file tools.c.

4.19.6.151 iexp()

```
LONG iexp (
            LONG x,
            int p )  [extern]
```

Definition at line 2939 of file tools.c.

4.19.6.152 IsLikeVector()

```
int IsLikeVector (
            WORD * arg )  [extern]
```

Definition at line 3137 of file tools.c.

4.19.6.153 AreArgsEqual()

```
int AreArgsEqual (
            WORD * arg1,
            WORD * arg2 )  [extern]
```

Definition at line 3165 of file tools.c.

4.19.6.154 CompareArgs()

```
int CompareArgs (
            WORD * arg1,
            WORD * arg2 )  [extern]
```

Definition at line 3184 of file tools.c.

4.19.6.155 SkipField()

```
UBYTE * SkipField (
            UBYTE * s,
            int level )  [extern]
```

Definition at line 1854 of file tools.c.

4.19.6.156 StrCmp()

```
int StrCmp (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1622 of file tools.c.

4.19.6.157 StrICmp()

```
int StrICmp (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1633 of file tools.c.

4.19.6.158 StrHICmp()

```
int StrHICmp (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1644 of file tools.c.

4.19.6.159 StrICont()

```
int StrICont (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1655 of file tools.c.

4.19.6.160 ConWord()

```
int ConWord (
            UBYTE * s1,
            UBYTE * s2 )  [extern]
```

Definition at line 1667 of file tools.c.

4.19.6.161 StrLen()

```
int StrLen (
            UBYTE * s )  [extern]
```

Definition at line 1679 of file tools.c.

4.19.6.162 GetPreVar()

```
UBYTE * GetPreVar (
            UBYTE * name,
            int flag )  [extern]
```

Definition at line 456 of file pre.c.

4.19.6.163 ToGeneral()

```
void ToGeneral (
            WORD * r,
            WORD * m,
            WORD par )  [extern]
```

Definition at line 2969 of file tools.c.

4.19.6.164 ToPolyFunGeneral()

```
WORD ToPolyFunGeneral (
            PHEAD WORD * term )  [extern]
```

Definition at line 3062 of file tools.c.

4.19.6.165 ToFast()

```
int ToFast (
            WORD * r,
            WORD * m )  [extern]
```

Definition at line 3009 of file tools.c.

4.19.6.166 GetSetupPar()

```
SETUPPARAMETERS * GetSetupPar (
            UBYTE * s )  [extern]
```

Definition at line 330 of file setfile.c.

4.19.6.167 AllocSort()

```
SORTING * AllocSort (
            LONG LargeSize,
            LONG SmallSize,
            LONG SmallEsize,
            LONG TermsInSmall,
            int MaxPatches,
            int MaxFpatches,
            LONG IOsize )  [extern]
```

Definition at line 835 of file setfile.c.

4.19.6.168 AllocSortFileName()

```
VOID AllocSortFileName (
            SORTING * sort )  [extern]
```

Definition at line 968 of file setfile.c.

4.19.6.169 LoadInputFile()

```
UBYTE * LoadInputFile (
            UBYTE * filename,
            int type )  [extern]
```

Definition at line 112 of file tools.c.

4.19.6.170 GetChar()

```
UBYTE GetChar (
            int level )  [extern]
```

Definition at line 165 of file pre.c.

4.19.6.171 CharOut()

```
VOID CharOut (
            UBYTE c )  [extern]
```

Definition at line 409 of file pre.c.

4.19.6.172 PopPreVars()

```
VOID PopPreVars (
            int tonumber )  [extern]
```

Definition at line 739 of file pre.c.

4.19.6.173 IniModule()

```
VOID IniModule (
            int type )  [extern]
```

Definition at line 754 of file pre.c.

4.19.6.174 IniSpecialModule()

```
VOID IniSpecialModule (
            int type )  [extern]
```

Definition at line 856 of file pre.c.

4.19.6.175 ModuleInstruction()

```
int ModuleInstruction (
            int * moduletype,
            int * specialtype )  [extern]
```

Definition at line 76 of file module.c.

4.19.6.176 LoadInstruction()

```
int LoadInstruction (
            int mode )  [extern]
```

Definition at line 1176 of file pre.c.

4.19.6.177 LoadStatement()

```
int LoadStatement (
            int type )  [extern]
```

Definition at line 1374 of file pre.c.

4.19.6.178 FindKeyWord()

```
KEYWORD * FindKeyWord (
            UBYTE * theword,
            KEYWORD * table,
            int size )  [extern]
```

Definition at line 1877 of file pre.c.

4.19.6.179 FindInKeyWord()

```
KEYWORD * FindInKeyWord (
            UBYTE * theword,
            KEYWORD * table,
            int size )  [extern]
```

Definition at line 1908 of file pre.c.

4.19.6.180 DoDefine()

```
int DoDefine (
            UBYTE * s )  [extern]
```

Definition at line 2073 of file pre.c.

4.19.6.181 DoRedefine()

```
int DoRedefine (
            UBYTE * s )  [extern]
```

Definition at line 2083 of file pre.c.

4.19.6.182 TheDefine()

```
int TheDefine (
            UBYTE * s,
            int mode )  [extern]
```

Preprocessor assignment. Possible arguments and values are treated and the new preprocessor variable is put into the name administration.

Parameters

s	Pointer to the character string following the preprocessor command.
mode	Bitmask. 0-bit clear: always create a new name entry, 0-bit set: try to redefine an existing name, 1-bit set: ignore preprocessor if/switch status.

Returns

zero: no errors, negative number: errors.

Definition at line 1938 of file pre.c.

References PutPreVar().

Here is the call graph for this function:

4.19.6.183 TheUndefine()

```
int TheUndefine (
            UBYTE * name )  [extern]
```

Definition at line 2123 of file pre.c.

4.19.6.184 ClearMacro()

```
int ClearMacro (
            UBYTE * name )  [extern]
```

Definition at line 2095 of file pre.c.

4.19.6.185 DoUndefine()

```
int DoUndefine (
            UBYTE * s )  [extern]
```

Definition at line 2177 of file pre.c.

4.19.6.186 DoInclude()

```
int DoInclude (
            UBYTE * s )  [extern]
```

Definition at line 2229 of file pre.c.

4.19.6.187 DoReverseInclude()

```
int DoReverseInclude (
            UBYTE * s )  [extern]
```

Definition at line 2236 of file pre.c.

4.19.6.188 Include()

```
int Include (
            UBYTE * s,
            int type )  [extern]
```

Definition at line 2243 of file pre.c.

4.19.6.189 DoExternal()

```
int DoExternal (
            UBYTE * s )  [extern]
```

Definition at line 5160 of file pre.c.

4.19.6.190 DoToExternal()

```
int DoToExternal (
            UBYTE * s )  [extern]
```

Definition at line 5721 of file pre.c.

4.19.6.191 DoFromExternal()

```
int DoFromExternal (
            UBYTE * s )  [extern]
```

Definition at line 5526 of file pre.c.

4.19.6.192 DoPrompt()

```
int DoPrompt (
            UBYTE * s )  [extern]
```

Definition at line 5242 of file pre.c.

4.19.6.193 DoSetExternal()

```
int DoSetExternal (
            UBYTE * s )  [extern]
```

Definition at line 5275 of file pre.c.

4.19.6.194 DoSetExternalAttr()

```
int DoSetExternalAttr (
            UBYTE * s )  [extern]
```

Definition at line 5345 of file pre.c.

4.19.6.195 DoRmExternal()

```
int DoRmExternal (
            UBYTE * s )  [extern]
```

Definition at line 5461 of file pre.c.

4.19.6.196 DoFactDollar()

```
int DoFactDollar (
            UBYTE * s )  [extern]
```

Definition at line 6355 of file pre.c.

4.19.6.197 GetDollarNumber()

```
WORD GetDollarNumber (
            UBYTE ** inp,
            DOLLARS d )  [extern]
```

Definition at line 6392 of file pre.c.

4.19.6.198 DoSetRandom()

```
int DoSetRandom (
            UBYTE * s )  [extern]
```

Definition at line 6482 of file pre.c.

4.19.6.199 DoOptimize()

```
int DoOptimize (
            UBYTE * s )  [extern]
```

Definition at line 6525 of file pre.c.

4.19.6.200 DoClearOptimize()

```
int DoClearOptimize (
            UBYTE * s )  [extern]
```

Definition at line 6658 of file pre.c.

4.19.6.201 DoSkipExtraSymbols()

```
int DoSkipExtraSymbols (
            UBYTE * s )  [extern]
```

Definition at line 6678 of file pre.c.

4.19.6.202 DoMessage()

```
int DoMessage (
            UBYTE * s )  [extern]
```

Definition at line 3544 of file pre.c.

4.19.6.203 DoPreOut()

```
int DoPreOut (
            UBYTE * s )  [extern]
```

Definition at line 3615 of file pre.c.

4.19.6.204 DoPreAppend()

```
int DoPreAppend (
            UBYTE * s )  [extern]
```

Definition at line 3655 of file pre.c.

4.19.6.205 DoPreCreate()

```
int DoPreCreate (
            UBYTE * s )  [extern]
```

Definition at line 3698 of file pre.c.

4.19.6.206 DoPreAssign()

```
int DoPreAssign (
            UBYTE * s )  [extern]
```

Definition at line 2042 of file pre.c.

4.19.6.207 DoPreBreak()

```
int DoPreBreak (
            UBYTE * s )  [extern]
```

Definition at line 3898 of file pre.c.

4.19.6.208 DoPreDefault()

```
int DoPreDefault (
            UBYTE * s )  [extern]
```

Definition at line 3961 of file pre.c.

4.19.6.209 DoPreSwitch()

```
int DoPreSwitch (
            UBYTE * s )  [extern]
```

Definition at line 4012 of file pre.c.

4.19.6.210 DoPreEndSwitch()

```
int DoPreEndSwitch (
            UBYTE * s )  [extern]
```

Definition at line 3985 of file pre.c.

4.19.6.211 DoPreCase()

```
int DoPreCase (
            UBYTE * s )  [extern]
```

Definition at line 3922 of file pre.c.

4.19.6.212 DoPreShow()

```
int DoPreShow (
            UBYTE * s )  [extern]
```

Definition at line 4066 of file pre.c.

4.19.6.213 DoPreExchange()

```
int DoPreExchange (
            UBYTE * s )  [extern]
```

Definition at line 2404 of file pre.c.

4.19.6.214 DoSystem()

```
int DoSystem (
            UBYTE * s )  [extern]
```

Definition at line 4099 of file pre.c.

4.19.6.215 DoPipe()

```
int DoPipe (
            UBYTE * s )  [extern]
```

Definition at line 3558 of file pre.c.

4.19.6.216 DoIfdef()

```
int DoIfdef (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 3279 of file pre.c.

4.19.6.217 DoElse()

```
int DoElse (
            UBYTE * s )  [extern]
```

Definition at line 2983 of file pre.c.

4.19.6.218 DoElseif()

```
int DoElseif (
            UBYTE * s )  [extern]
```

Definition at line 3020 of file pre.c.

4.19.6.219 DoEndif()

```
int DoEndif (
            UBYTE * s )  [extern]
```

Definition at line 3201 of file pre.c.

4.19.6.220 DoTerminate()

```
int DoTerminate (
            UBYTE * s )  [extern]
```

Definition at line 2671 of file pre.c.

4.19.6.221 DoIf()

```
int DoIf (
            UBYTE * s )  [extern]
```

Definition at line 3255 of file pre.c.

4.19.6.222 DoCall()

```
int DoCall (
            UBYTE * s )  [extern]
```

Definition at line 2465 of file pre.c.

4.19.6.223 DoDebug()

```
int DoDebug (
            UBYTE * s )  [extern]
```

Definition at line 2634 of file pre.c.

4.19.6.224 DoDo()

```
int DoDo (
            UBYTE * s )  [extern]
```

Definition at line 2700 of file pre.c.

4.19.6.225 DoBreakDo()

```
int DoBreakDo (
            UBYTE * s )  [extern]
```

Definition at line 2914 of file pre.c.

4.19.6.226 DoEnddo()

```
int DoEnddo (
            UBYTE * s )  [extern]
```

Definition at line 3055 of file pre.c.

4.19.6.227 DoEndprocedure()

```
int DoEndprocedure (
            UBYTE * s )  [extern]
```

Definition at line 3229 of file pre.c.

4.19.6.228 DoInside()

```
int DoInside (
            UBYTE * s )  [extern]
```

Definition at line 3319 of file pre.c.

4.19.6.229 DoEndInside()

```
int DoEndInside (
            UBYTE * s )  [extern]
```

Definition at line 3412 of file pre.c.

4.19.6.230 DoProcedure()

```
int DoProcedure (
            UBYTE * s )  [extern]
```

Definition at line 3856 of file pre.c.

4.19.6.231 DoPrePrintTimes()

```
int DoPrePrintTimes (
            UBYTE * s )  [extern]
```

Definition at line 3638 of file pre.c.

4.19.6.232 DoPreWrite()

```
int DoPreWrite (
            UBYTE * s )  [extern]
```

Definition at line 3815 of file pre.c.

4.19.6.233 DoPreClose()

```
int DoPreClose (
            UBYTE * s )  [extern]
```

Definition at line 3771 of file pre.c.

4.19.6.234 DoPreRemove()

```
int DoPreRemove (
            UBYTE * s )  [extern]
```

Definition at line 3738 of file pre.c.

4.19.6.235 DoCommentChar()

```
int DoCommentChar (
            UBYTE * s )  [extern]
```

Definition at line 2011 of file pre.c.

4.19.6.236 DoPrcExtension()

```
int DoPrcExtension (
            UBYTE * s )  [extern]
```

Definition at line 3581 of file pre.c.

4.19.6.237 DoPreReset()

```
int DoPreReset (
            UBYTE * s )  [extern]
```

Definition at line 6717 of file pre.c.

4.19.6.238 WriteString()

```
VOID WriteString (
            int type,
            UBYTE * str,
            int num )  [extern]
```

Definition at line 1715 of file tools.c.

4.19.6.239 WriteUnfinString()

```
VOID WriteUnfinString (
            int type,
            UBYTE * str,
            int num )  [extern]
```

Definition at line 1749 of file tools.c.

4.19.6.240 AddToString()

```
UBYTE * AddToString (
            UBYTE * outstring,
            UBYTE * extrastring,
            int par )  [extern]
```

Definition at line 1774 of file tools.c.

4.19.6.241 PreEval()

```
UBYTE * PreEval (
            UBYTE * s,
            LONG * x )  [extern]
```

Definition at line 4933 of file pre.c.

4.19.6.242 NumToStr()

```
VOID NumToStr (
            UBYTE * s,
            LONG x )  [extern]
```

Definition at line 1691 of file tools.c.

4.19.6.243 PreCmp()

```
int PreCmp (
            int type,
            int val,
            UBYTE * t,
            int type2,
            int val2,
            UBYTE * t2,
            int cmpop )  [extern]
```

Definition at line 4455 of file pre.c.

4.19.6.244 PreEq()

```
int PreEq (
            int type,
            int val,
            UBYTE * t,
            int type2,
            int val2,
            UBYTE * t2,
            int eqop )  [extern]
```

Definition at line 4479 of file pre.c.

4.19.6.245 pParseObject()

```
UBYTE * pParseObject (
            UBYTE * s,
            int * type,
            LONG * val2 )  [extern]
```

Definition at line 4508 of file pre.c.

4.19.6.246 PreIfEval()

```
UBYTE * PreIfEval (
            UBYTE * s,
            int * value )  [extern]
```

Definition at line 4327 of file pre.c.

4.19.6.247 EvalPreIf()

```
int EvalPreIf (
            UBYTE * s )  [extern]
```

Definition at line 4292 of file pre.c.

4.19.6.248 PreLoad()

```
int PreLoad (
            PRELOAD * p,
            UBYTE * start,
            UBYTE * stop,
            int mode,
            char * message )  [extern]
```

Definition at line 4126 of file pre.c.

4.19.6.249 PreSkip()

```
int PreSkip (
            UBYTE * start,
            UBYTE * stop,
            int mode )  [extern]
```

Definition at line 4209 of file pre.c.

4.19.6.250 EndOfToken()

```
UBYTE * EndOfToken (
            UBYTE * s )  [extern]
```

Definition at line 1827 of file tools.c.

4.19.6.251 SetSpecialMode()

```
VOID SetSpecialMode (
            int moduletype,
            int specialtype )  [extern]
```

Definition at line 255 of file module.c.

4.19.6.252 ExecModule()

```
int ExecModule (
            int moduletype )  [extern]
```

Definition at line 272 of file module.c.

4.19.6.253 DoPolyfun()

```
int DoPolyfun (
            UBYTE * s )  [extern]
```

Definition at line 374 of file module.c.

4.19.6.254 DoPolyratfun()

```
int DoPolyratfun (
            UBYTE * s )  [extern]
```

Definition at line 437 of file module.c.

4.19.6.255 CompileStatement()

```
int CompileStatement (
            UBYTE * in )  [extern]
```

Definition at line 530 of file compiler.c.

4.19.6.256 ToToken()

```
UBYTE * ToToken (
            UBYTE * s )  [extern]
```

Definition at line 1839 of file tools.c.

4.19.6.257 GetDollar()

```
int GetDollar (
            UBYTE * name )  [extern]
```

Definition at line 492 of file names.c.

4.19.6.258 MesCall()

```
int MesCall (
            char * s )  [extern]
```

Definition at line 814 of file message.c.

4.19.6.259 NumCopy()

```
UBYTE * NumCopy (
            WORD y,
            UBYTE * to )  [extern]
```

Definition at line 1905 of file tools.c.

4.19.6.260 LongCopy()

```
char * LongCopy (
            LONG y,
            char * to )  [extern]
```

Definition at line 1935 of file tools.c.

4.19.6.261 LongLongCopy()

```
char * LongLongCopy (
            off_t * y,
            char * to )  [extern]
```

Definition at line 1965 of file tools.c.

4.19.6.262 ReserveTempFiles()

```
VOID ReserveTempFiles (
            int par )  [extern]
```

Definition at line 593 of file startup.c.

4.19.6.263 PrintTerm()

```
VOID PrintTerm (
            WORD * term,
            char * where )  [extern]
```

Definition at line 857 of file message.c.

4.19.6.264 PrintTermC()

```
VOID PrintTermC (
            WORD * term,
            char * where )  [extern]
```

Definition at line 887 of file message.c.

4.19.6.265 PrintSubTerm()

```
VOID PrintSubTerm (
            WORD * term,
            char * where )  [extern]
```

Definition at line 921 of file message.c.

4.19.6.266 PrintWords()

```
VOID PrintWords (
            WORD * buffer,
            LONG number )  [extern]
```

Definition at line 943 of file message.c.

4.19.6.267 PrintSeq()

```
void PrintSeq (
            WORD * a,
            char * text )  [extern]
```

Definition at line 961 of file message.c.

4.19.6.268 ExpandTripleDots()

```
int ExpandTripleDots (
            int par )  [extern]
```

Definition at line 1509 of file pre.c.

4.19.6.269 ComPress()

```
LONG ComPress (
            WORD ** ss,
            LONG * n )  [extern]
```

Gets a list of pointers to terms and compresses the terms. In n it collects the number of terms and the return value of the function is the space that is occupied.

We have to pay some special attention to the compression of terms with a PolyFun. This PolyFun should occur only straight before the coefficient, so we can use the same trick as for the coefficient to sabotage compression of this object (Replace in the history the function pointer by zero. This is safe, because terms that would be identical otherwise would have been added).

Parameters

ss	Array of pointers to terms to be compressed.
n	Number of pointers in ss.

Returns

Total number of words needed for the compressed result.

Definition at line 3041 of file sort.c.

Referenced by EndSort(), and StoreTerm().

4.19.6.270 StageSort()

```
VOID StageSort (
            FILEHANDLE * fout )  [extern]
```

Prepares a stage 4 or higher sort. Stage 4 sorts occur when the sort file contains more patches than can be merged in one pass.

Definition at line 4364 of file sort.c.

References FiLe::handle.

Referenced by EndSort(), and MergePatches().

4.19.6.271 M_free()

```
void M_free (
            VOID * x,
            const char * where )  [extern]
```

Definition at line 2296 of file tools.c.

4.19.6.272 AddWildcardName()

```
int AddWildcardName (
            UBYTE * name )  [extern]
```

Definition at line 756 of file names.c.

4.19.6.273 GetWildcardName()

```
int GetWildcardName (
            UBYTE * name )  [extern]
```

Definition at line 800 of file names.c.

4.19.6.274 Globalize()

```
void Globalize (
            int par )  [extern]
```

Definition at line 2964 of file names.c.

4.19.6.275 ResetVariables()

```
void ResetVariables (
            int par )  [extern]
```

Definition at line 2641 of file names.c.

4.19.6.276 AddToPreTypes()

```
void AddToPreTypes (
            int type )  [extern]
```

Definition at line 5077 of file pre.c.

4.19.6.277 MessPreNesting()

```
void MessPreNesting (
            int par )  [extern]
```

Definition at line 5095 of file pre.c.

4.19.6.278 GetStreamPosition()

```
LONG GetStreamPosition (
            STREAM * stream )  [extern]
```

Definition at line 741 of file tools.c.

4.19.6.279 DoubleCbuffer()

```
WORD * DoubleCbuffer (
            int num,
            WORD * w,
            int par )  [extern]
```

Doubles a compiler buffer.

Parameters

num	The ID number for the buffer to be doubled.
w	The pointer to the end (exclusive) of the current buffer. The contents in the range of [cbuf[num].Buffer,w) will be kept.

Definition at line 143 of file comtool.c.

References CbUf::Buffer, CbUf::BufferSize, CbUf::lhs, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by AddNtoC(), AddNtoL(), and TestSub().

4.19.6.280 AddLHS()

```
WORD * AddLHS (
            int num )  [extern]
```

Adds an LHS to a compiler buffer and returns the pointer to a buffer for the new LHS.

Parameters

num	The ID number for the buffer to get another LHS.

Definition at line 188 of file comtool.c.

References CbUf::lhs, and CbUf::Pointer.

Referenced by AddNtoL().

4.19.6.281 AddRHS()

```
WORD * AddRHS (
            int num,
            int type )  [extern]
```

Adds an RHS to a compiler buffer and returns the pointer to a buffer for the new RHS.

Parameters

num	The ID number for the buffer to get another RHS.
type	If 0, the subexpression tree will be reallocated.

Definition at line 214 of file comtool.c.

References TaBlEs::buffers, TaBlEs::buffersfill, TaBlEs::bufferssize, TaBlEs::bufnum, CbUf::CanCommu, CbUf::dimension, inicbufs(), CbUf::numdum, CbUf::NumTerms, CbUf::Pointer, and CbUf::rhs.

Referenced by InsertArg(), StartVariables(), TestMatch(), and TestSub().

Here is the call graph for this function:

4.19.6.282 AddNtoL()

```
int AddNtoL (
            int n,
            WORD * array )  [extern]
```

Adds an LHS with the given data to the current compiler buffer.

Parameters

n	The length of the data.
array	The data to be added.

Returns

0 if succeeds.

Definition at line 288 of file comtool.c.

References AddLHS(), DoubleCbuffer(), CbUf::Pointer, and CbUf::Top.

Here is the call graph for this function:

4.19.6.283 AddNtoC()

```
int AddNtoC (
            int bufnum,
            int n,
            WORD * array,
            int par )  [extern]
```

Adds the given data to the last LHS/RHS in a compiler buffer.

Parameters

bufnum	The ID number for the buffer where the data will be added.
n	The length of the data.
array	The data to be added.

Returns

0 if succeeds.

Definition at line 317 of file comtool.c.

References DoubleCbuffer(), CbUf::Pointer, and CbUf::Top.

Referenced by InsertArg(), and StartVariables().

Here is the call graph for this function:

4.19.6.284 CreateStream()

```
STREAM * CreateStream (
            UBYTE * where )  [extern]
```

Definition at line 710 of file tools.c.

4.19.6.285 setonoff()

```
int setonoff (
            UBYTE * s,
            int * flag,
            int onvalue,
            int offvalue )  [extern]
```

Definition at line 205 of file compcomm.c.

4.19.6.286 DoPrint()

```
int DoPrint (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 954 of file compcomm.c.

4.19.6.287 SetExpr()

```
int SetExpr (
            UBYTE * s,
            int setunset,
            int par )  [extern]
```

Definition at line 1388 of file compcomm.c.

4.19.6.288 AddToCom()

```
void AddToCom (
            int n,
            WORD * array )  [extern]
```

Definition at line 1518 of file compcomm.c.

4.19.6.289 Add2ComStrings()

```
int Add2ComStrings (
            int n,
            WORD * array,
            UBYTE * string1,
            UBYTE * string2 )  [extern]
```

Definition at line 1592 of file compcomm.c.

4.19.6.290 DoSymmetrize()

```
int DoSymmetrize (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 2086 of file compcomm.c.

4.19.6.291 DoArgument()

```
int DoArgument (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 1731 of file compcomm.c.

4.19.6.292 ArgFactorize()

```
int ArgFactorize (
            PHEAD WORD * argin,
            WORD * argout )  [extern]
```

Definition at line 2015 of file argument.c.

4.19.6.293 TakeArgContent()

```
WORD * TakeArgContent (
            PHEAD WORD * argin,
            WORD * argout )  [extern]
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. The common pieces are put in argout as a sequence of arguments. The part with the multiple terms that are now relative prime is put in argfree which is allocated via TermMalloc and is given as the return value. The difference with the old code is that negative powers are always removed. Hence it is as in MakeInteger in which only numerators will be left: now only zero or positive powers will be remaining.

Definition at line 2716 of file argument.c.

References EndSort(), MakeInteger(), MakeMod(), NewSort(), and StoreTerm().

Here is the call graph for this function:

4.19.6.294 MakeInteger()

```
WORD * MakeInteger (
            PHEAD WORD * argin,
            WORD * argout,
            WORD * argfree )  [extern]
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in argin. The number that comes out should go to argout. The new pointer in the argout buffer is the return value. The normalized argument is in argfree.

Definition at line 3262 of file argument.c.

Referenced by TakeArgContent().

4.19.6.295 MakeMod()

```
WORD * MakeMod (
            PHEAD WORD * argin,
            WORD * argout,
            WORD * argfree )  [extern]
```

Similar to MakeInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3433 of file argument.c.

References GetModInverses().

Referenced by TakeArgContent().

Here is the call graph for this function:

4.19.6.296 FindArg()

```
WORD FindArg (
            PHEAD WORD * a )  [extern]
```

Looks the argument up in the (workers) table. If it is found the number in the table is returned (plus one to make it positive). If it is not found we look in the compiler provided table. If it is found - the number in the table is returned (minus one to make it negative). If in neither table we return zero.

Definition at line 2463 of file argument.c.

4.19.6.297 InsertArg()

```
WORD InsertArg (
            PHEAD WORD * argin,
            WORD * argout,
            int par )  [extern]
```

Inserts the argument into the (workers) table. If the table is too full we eliminate half of it. The eliminated elements are the ones that have not been used most recently, weighted by their total use and age(?). If par == 0 it inserts in the regular factorization cache If par == 1 it inserts in the cache defined with the FactorCache statement

Definition at line 2487 of file argument.c.

References AddNtoC(), AddRHS(), and CleanupArgCache().

Here is the call graph for this function:

4.19.6.298 CleanupArgCache()

```
int CleanupArgCache (
            PHEAD WORD bufnum )  [extern]
```

Cleans up the argument factorization cache. We throw half the elements. For a weight of what we want to keep we use the product of usage and the number in the buffer.

Definition at line 2522 of file argument.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::Pointer, CbUf::rhs, SortWeights(), and tree::usage.

Referenced by InsertArg().

Here is the call graph for this function:

4.19.6.299 ArgSymbolMerge()

```
int ArgSymbolMerge (
            WORD * t1,
            WORD * t2 )  [extern]
```

Definition at line 2593 of file argument.c.

4.19.6.300 ArgDotproductMerge()

```
int ArgDotproductMerge (
            WORD * t1,
            WORD * t2 )  [extern]
```

Definition at line 2648 of file argument.c.

4.19.6.301 SortWeights()

```
void SortWeights (
            LONG * weights,
            LONG * extraspace,
            WORD number )  [extern]
```

Sorts an array of LONGS in the same way SplitMerge (in sort.c) works We use gradual division in two.

Definition at line 3478 of file argument.c.

References SortWeights().

Referenced by CleanupArgCache(), and SortWeights().

Here is the call graph for this function:

4.19.6.302 DoBrackets()

```
int DoBrackets (
            UBYTE * inp,
            int par )  [extern]
```

Definition at line 3673 of file compcomm.c.

4.19.6.303 DoPutInside()

```
int DoPutInside (
            UBYTE * inp,
            int par )  [extern]
```

Definition at line 6866 of file compcomm.c.

4.19.6.304 CountComp()

```
WORD * CountComp (
            UBYTE * inp,
            WORD * to )  [extern]
```

Definition at line 3951 of file compcomm.c.

4.19.6.305 CoAntiBracket()

```
int CoAntiBracket (
            UBYTE * inp )  [extern]
```

Definition at line 3808 of file compcomm.c.

4.19.6.306 CoAntiSymmetrize()

```
int CoAntiSymmetrize (
            UBYTE * s )  [extern]
```

Definition at line 2213 of file compcomm.c.

4.19.6.307 DoArgPlode()

```
int DoArgPlode (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 5526 of file compcomm.c.

4.19.6.308 CoArgExplode()

```
int CoArgExplode (
            UBYTE * s )  [extern]
```

Definition at line 5582 of file compcomm.c.

4.19.6.309 CoArgImplode()

```
int CoArgImplode (
            UBYTE * s )  [extern]
```

Definition at line 5589 of file compcomm.c.

4.19.6.310 CoArgument()

```
int CoArgument (
            UBYTE * s )  [extern]
```

Definition at line 1965 of file compcomm.c.

4.19.6.311 CoInside()

```
int CoInside (
            UBYTE * s )  [extern]
```

Definition at line 1998 of file compcomm.c.

4.19.6.312 ExecInside()

```
int ExecInside (
            UBYTE * s )  [extern]
```

Definition at line 1648 of file dollar.c.

4.19.6.313 CoInExpression()

```
int CoInExpression (
            UBYTE * s )  [extern]
```

Definition at line 3226 of file compcomm.c.

4.19.6.314 CoInParallel()

```
int CoInParallel (
            UBYTE * s )  [extern]
```

Definition at line 3132 of file compcomm.c.

4.19.6.315 CoNotInParallel()

```
int CoNotInParallel (
            UBYTE * s )  [extern]
```

Definition at line 3142 of file compcomm.c.

4.19.6.316 DoInParallel()

```
int DoInParallel (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 3157 of file compcomm.c.

4.19.6.317 CoEndInExpression()

```
int CoEndInExpression (
            UBYTE * s )  [extern]
```

Definition at line 3279 of file compcomm.c.

4.19.6.318 CoBracket()

```
int CoBracket (
            UBYTE * inp )  [extern]
```

Definition at line 3800 of file compcomm.c.

4.19.6.319 CoPutInside()

```
int CoPutInside (
            UBYTE * inp )  [extern]
```

Definition at line 6863 of file compcomm.c.

4.19.6.320 CoAntiPutInside()

```
int CoAntiPutInside (
            UBYTE * inp )  [extern]
```

Definition at line 6864 of file compcomm.c.

4.19.6.321 CoMultiBracket()

```
int CoMultiBracket (
            UBYTE * inp )  [extern]
```

Definition at line 3819 of file compcomm.c.

4.19.6.322 CoCFunction()

```
int CoCFunction (
            UBYTE * s )  [extern]
```

Definition at line 1524 of file names.c.

4.19.6.323 CoCTensor()

```
int CoCTensor (
            UBYTE * s )  [extern]
```

Definition at line 1526 of file names.c.

4.19.6.324 CoCollect()

```
int CoCollect (
            UBYTE * s )  [extern]
```

Definition at line 143 of file compcomm.c.

4.19.6.325 CoCompress()

```
int CoCompress (
            UBYTE * s )  [extern]
```

Definition at line 221 of file compcomm.c.

4.19.6.326 CoContract()

```
int CoContract (
            UBYTE * s )  [extern]
```

Definition at line 1656 of file compcomm.c.

4.19.6.327 CoCycleSymmetrize()

```
int CoCycleSymmetrize (
            UBYTE * s )  [extern]
```

Definition at line 2220 of file compcomm.c.

4.19.6.328 CoDelete()

```
int CoDelete (
            UBYTE * s )  [extern]
```

Definition at line 623 of file compcomm.c.

4.19.6.329 CoTableBase()

```
int CoTableBase (
            UBYTE * s )  [extern]
```

Definition at line 575 of file tables.c.

4.19.6.330 CoApply()

```
int CoApply (
            UBYTE * s )  [extern]
```

Definition at line 1713 of file tables.c.

4.19.6.331 CoDenominators()

```
int CoDenominators (
            UBYTE * s )  [extern]
```

Definition at line 5678 of file compcomm.c.

4.19.6.332 CoDimension()

```
int CoDimension (
            UBYTE * s )  [extern]
```

Definition at line 1125 of file names.c.

4.19.6.333 CoDiscard()

```
int CoDiscard (
            UBYTE * s )  [extern]
```

Definition at line 1633 of file compcomm.c.

4.19.6.334 CoDisorder()

```
int CoDisorder (
            UBYTE * inp )  [extern]
```

Definition at line 413 of file comexpr.c.

4.19.6.335 CoDrop()

```
int CoDrop (
            UBYTE * s )  [extern]
```

Definition at line 1433 of file compcomm.c.

4.19.6.336 CoDropCoefficient()

```
int CoDropCoefficient (
            UBYTE * s )  [extern]
```

Definition at line 5707 of file compcomm.c.

4.19.6.337 CoDropSymbols()

```
int CoDropSymbols (
            UBYTE * s )  [extern]
```

Definition at line 5721 of file compcomm.c.

4.19.6.338 CoElse()

```
int CoElse (
            UBYTE * p )  [extern]
```

Definition at line 4704 of file compcomm.c.

4.19.6.339 CoElseIf()

```
int CoElseIf (
            UBYTE * inp )  [extern]
```

Definition at line 4731 of file compcomm.c.

4.19.6.340 CoEndArgument()

```
int CoEndArgument (
            UBYTE * s )  [extern]
```

Definition at line 1972 of file compcomm.c.

4.19.6.341 CoEndInside()

```
int CoEndInside (
            UBYTE * s )  [extern]
```

Definition at line 2005 of file compcomm.c.

4.19.6.342 CoEndIf()

```
int CoEndIf (
            UBYTE * inp )  [extern]
```

Definition at line 4758 of file compcomm.c.

4.19.6.343 CoEndRepeat()

```
int CoEndRepeat (
            UBYTE * inp )  [extern]
```

Definition at line 3635 of file compcomm.c.

4.19.6.344 CoEndTerm()

```
int CoEndTerm (
            UBYTE * s )  [extern]
```

Definition at line 5098 of file compcomm.c.

4.19.6.345 CoEndWhile()

```
int CoEndWhile (
            UBYTE * inp )  [extern]
```

Definition at line 4824 of file compcomm.c.

4.19.6.346 CoExit()

```
int CoExit (
            UBYTE * s )  [extern]
```

Definition at line 3105 of file compcomm.c.

4.19.6.347 CoFactArg()

```
int CoFactArg (
            UBYTE * s )  [extern]
```

Definition at line 2066 of file compcomm.c.

4.19.6.348 CoFactDollar()

```
int CoFactDollar (
            UBYTE * inp )  [extern]
```

Definition at line 6237 of file compcomm.c.

4.19.6.349 CoFactorize()

```
int CoFactorize (
            UBYTE * s )  [extern]
```

Definition at line 6266 of file compcomm.c.

4.19.6.350 CoNFactorize()

```
int CoNFactorize (
            UBYTE * s )  [extern]
```

Definition at line 6273 of file compcomm.c.

4.19.6.351 CoUnFactorize()

```
int CoUnFactorize (
            UBYTE * s )  [extern]
```

Definition at line 6280 of file compcomm.c.

4.19.6.352 CoNUnFactorize()

```
int CoNUnFactorize (
            UBYTE * s )  [extern]
```

Definition at line 6287 of file compcomm.c.

4.19.6.353 DoFactorize()

```
int DoFactorize (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 6294 of file compcomm.c.

4.19.6.354 CoFill()

```
int CoFill (
            UBYTE * inp )  [extern]
```

Definition at line 1063 of file comexpr.c.

4.19.6.355 CoFillExpression()

```
int CoFillExpression (
            UBYTE * inp )  [extern]
```

Definition at line 1339 of file comexpr.c.

4.19.6.356 CoFixIndex()

```
int CoFixIndex (
            UBYTE * s )  [extern]
```

Definition at line 912 of file compcomm.c.

4.19.6.357 CoFormat()

```
int CoFormat (
            UBYTE * s )  [extern]
```

Definition at line 668 of file compcomm.c.

4.19.6.358 CoGlobal()

```
int CoGlobal (
            UBYTE * inp )  [extern]
```

Definition at line 73 of file comexpr.c.

4.19.6.359 CoGlobalFactorized()

```
int CoGlobalFactorized (
            UBYTE * inp )  [extern]
```

Definition at line 87 of file comexpr.c.

4.19.6.360 CoGoTo()

```
int CoGoTo (
            UBYTE * inp )  [extern]
```

Definition at line 1685 of file compcomm.c.

4.19.6.361 CoId()

```
int CoId (
            UBYTE * inp )  [extern]
```

Definition at line 391 of file comexpr.c.

4.19.6.362 CoIdNew()

```
int CoIdNew (
            UBYTE * inp )  [extern]
```

Definition at line 402 of file comexpr.c.

4.19.6.363 CoIdOld()

```
int CoIdOld (
            UBYTE * inp )  [extern]
```

Definition at line 380 of file comexpr.c.

4.19.6.364 CoIf()

```
int CoIf (
            UBYTE * inp )  [extern]
```

Definition at line 4141 of file compcomm.c.

4.19.6.365 CoIfMatch()

```
int CoIfMatch (
            UBYTE * inp )  [extern]
```

Definition at line 446 of file comexpr.c.

4.19.6.366 CoIfNoMatch()

```
int CoIfNoMatch (
            UBYTE * inp )  [extern]
```

Definition at line 457 of file comexpr.c.

4.19.6.367 CoIndex()

```
int CoIndex (
            UBYTE * s )  [extern]
```

Definition at line 1014 of file names.c.

4.19.6.368 CoInsideFirst()

```
int CoInsideFirst (
            UBYTE * s )  [extern]
```

Definition at line 609 of file compcomm.c.

4.19.6.369 CoKeep()

```
int CoKeep (
            UBYTE * s )  [extern]
```

Definition at line 900 of file compcomm.c.

4.19.6.370 CoLabel()

```
int CoLabel (
            UBYTE * inp )  [extern]
```

Definition at line 1704 of file compcomm.c.

4.19.6.371 CoLoad()

```
int CoLoad (
            UBYTE * inp )  [extern]
```

Definition at line 561 of file store.c.

4.19.6.372 CoLocal()

```
int CoLocal (
            UBYTE * inp )  [extern]
```

Definition at line 66 of file comexpr.c.

4.19.6.373 CoLocalFactorized()

```
int CoLocalFactorized (
            UBYTE * inp )  [extern]
```

Definition at line 80 of file comexpr.c.

4.19.6.374 CoMany()

```
int CoMany (
            UBYTE * inp )  [extern]
```

Definition at line 424 of file comexpr.c.

4.19.6.375 CoMerge()

```
int CoMerge (
            UBYTE * inp )  [extern]
```

Definition at line 5367 of file compcomm.c.

4.19.6.376 CoStuffle()

```
int CoStuffle (
            UBYTE * inp )  [extern]
```

Definition at line 5423 of file compcomm.c.

4.19.6.377 CoMetric()

```
int CoMetric (
            UBYTE * s )  [extern]
```

Definition at line 946 of file compcomm.c.

4.19.6.378 CoModOption()

```
int CoModOption (
            UBYTE * s )  [extern]
```

Definition at line 210 of file module.c.

4.19.6.379 CoModuleOption()

```
int CoModuleOption (
            UBYTE * s )  [extern]
```

Definition at line 143 of file module.c.

4.19.6.380 CoModulus()

```
int CoModulus (
            UBYTE * inp )  [extern]
```

Definition at line 3422 of file compcomm.c.

4.19.6.381 CoMulti()

```
int CoMulti (
            UBYTE * inp )  [extern]
```

Definition at line 435 of file comexpr.c.

4.19.6.382 CoMultiply()

```
int CoMultiply (
            UBYTE * inp )  [extern]
```

Definition at line 1024 of file comexpr.c.

4.19.6.383 CoNFunction()

```
int CoNFunction (
            UBYTE * s )  [extern]
```

Definition at line 1523 of file names.c.

4.19.6.384 CoNPrint()

```
int CoNPrint (
            UBYTE * s )  [extern]
```

Definition at line 1156 of file compcomm.c.

4.19.6.385 CoNTensor()

```
int CoNTensor (
            UBYTE * s )  [extern]
```

Definition at line 1525 of file names.c.

4.19.6.386 CoNWrite()

```
int CoNWrite (
            UBYTE * s )  [extern]
```

Definition at line 2259 of file compcomm.c.

4.19.6.387 CoNoDrop()

```
int CoNoDrop (
            UBYTE * s )  [extern]
```

Definition at line 1440 of file compcomm.c.

4.19.6.388 CoNoSkip()

```
int CoNoSkip (
            UBYTE * s )  [extern]
```

Definition at line 1454 of file compcomm.c.

4.19.6.389 CoNormalize()

```
int CoNormalize (
            UBYTE * s )  [extern]
```

Definition at line 2031 of file compcomm.c.

4.19.6.390 CoMakeInteger()

```
int CoMakeInteger (
            UBYTE * s )  [extern]
```

Definition at line 2038 of file compcomm.c.

4.19.6.391 CoFlags()

```
int CoFlags (
            UBYTE * s,
            int value )  [extern]
```

Definition at line 270 of file compcomm.c.

4.19.6.392 CoOff()

```
int CoOff (
            UBYTE * s )  [extern]
```

Definition at line 305 of file compcomm.c.

4.19.6.393 CoOn()

```
int CoOn (
            UBYTE * s )  [extern]
```

Definition at line 362 of file compcomm.c.

4.19.6.394 CoOnce()

```
int CoOnce (
            UBYTE * inp )  [extern]
```

Definition at line 468 of file comexpr.c.

4.19.6.395 CoOnly()

```
int CoOnly (
            UBYTE * inp )  [extern]
```

Definition at line 479 of file comexpr.c.

4.19.6.396 CoOptimizeOption()

```
int CoOptimizeOption (
            UBYTE * s )  [extern]
```

Definition at line 6427 of file compcomm.c.

4.19.6.397 CoPolyFun()

```
int CoPolyFun (
            UBYTE * s )  [extern]
```

Definition at line 5164 of file compcomm.c.

4.19.6.398 CoPolyRatFun()

```
int CoPolyRatFun (
            UBYTE * s )  [extern]
```

Definition at line 5203 of file compcomm.c.

4.19.6.399 CoPrint()

```
int CoPrint (
            UBYTE * s )  [extern]
```

Definition at line 1142 of file compcomm.c.

4.19.6.400 CoPrintB()

```
int CoPrintB (
            UBYTE * s )  [extern]
```

Definition at line 1149 of file compcomm.c.

4.19.6.401 CoProperCount()

```
int CoProperCount (
            UBYTE * s )  [extern]
```

Definition at line 616 of file compcomm.c.

4.19.6.402 CoUnitTrace()

```
int CoUnitTrace (
            UBYTE * s )  [extern]
```

Definition at line 5015 of file compcomm.c.

4.19.6.403 CoRCycleSymmetrize()

```
int CoRCycleSymmetrize (
            UBYTE * s )  [extern]
```

Definition at line 2227 of file compcomm.c.

4.19.6.404 CoRatio()

```
int CoRatio (
            UBYTE * s )  [extern]
```

Definition at line 2286 of file compcomm.c.

4.19.6.405 CoRedefine()

```
int CoRedefine (
            UBYTE * s )  [extern]
```

Definition at line 2326 of file compcomm.c.

4.19.6.406 CoRenumber()

```
int CoRenumber (
            UBYTE * s )  [extern]
```

Definition at line 2431 of file compcomm.c.

4.19.6.407 CoRepeat()

```
int CoRepeat (
            UBYTE * inp )  [extern]
```

Definition at line 3612 of file compcomm.c.

4.19.6.408 CoSave()

```
int CoSave (
            UBYTE * inp )  [extern]
```

Definition at line 363 of file store.c.

4.19.6.409 CoSelect()

```
int CoSelect (
            UBYTE * inp )  [extern]
```

Definition at line 490 of file comexpr.c.

4.19.6.410 CoSet()

```
int CoSet (
            UBYTE * s )  [extern]
```

Definition at line 2217 of file names.c.

4.19.6.411 CoSetExitFlag()

```
int CoSetExitFlag (
            UBYTE * s )  [extern]
```

Definition at line 3305 of file compcomm.c.

4.19.6.412 CoSkip()

```
int CoSkip (
            UBYTE * s )  [extern]
```

Definition at line 1447 of file compcomm.c.

4.19.6.413 CoProcessBucket()

```
int CoProcessBucket (
            UBYTE * s )  [extern]
```

Definition at line 5478 of file compcomm.c.

4.19.6.414 CoPushHide()

```
int CoPushHide (
            UBYTE * s )  [extern]
```

Definition at line 1163 of file compcomm.c.

4.19.6.415 CoPopHide()

```
int CoPopHide (
            UBYTE * s )  [extern]
```

Definition at line 1207 of file compcomm.c.

4.19.6.416 CoHide()

```
int CoHide (
            UBYTE * inp )  [extern]
```

Definition at line 1461 of file compcomm.c.

4.19.6.417 CoIntoHide()

```
int CoIntoHide (
            UBYTE * inp )  [extern]
```

Definition at line 1479 of file compcomm.c.

4.19.6.418 CoNoHide()

```
int CoNoHide (
            UBYTE * inp )  [extern]
```

Definition at line 1497 of file compcomm.c.

4.19.6.419 CoUnHide()

```
int CoUnHide (
            UBYTE * inp )  [extern]
```

Definition at line 1504 of file compcomm.c.

4.19.6.420 CoNoUnHide()

```
int CoNoUnHide (
            UBYTE * inp )  [extern]
```

Definition at line 1511 of file compcomm.c.

4.19.6.421 CoSort()

```
int CoSort (
            UBYTE * s )  [extern]
```

Definition at line 5125 of file compcomm.c.

4.19.6.422 CoSplitArg()

```
int CoSplitArg (
            UBYTE * s )  [extern]
```

Definition at line 2045 of file compcomm.c.

4.19.6.423 CoSplitFirstArg()

```
int CoSplitFirstArg (
            UBYTE * s )  [extern]
```

Definition at line 2052 of file compcomm.c.

4.19.6.424 CoSplitLastArg()

```
int CoSplitLastArg (
            UBYTE * s )  [extern]
```

Definition at line 2059 of file compcomm.c.

4.19.6.425 CoSum()

```
int CoSum (
            UBYTE * s )  [extern]
```

Definition at line 2452 of file compcomm.c.

4.19.6.426 CoSymbol()

```
int CoSymbol (
            UBYTE * s )  [extern]
```

Definition at line 848 of file names.c.

4.19.6.427 CoSymmetrize()

```
int CoSymmetrize (
            UBYTE * s )  [extern]
```

Definition at line 2206 of file compcomm.c.

4.19.6.428 DoTable()

```
int DoTable (
            UBYTE * s,
            int par )  [extern]
```

Definition at line 1560 of file names.c.

4.19.6.429 CoTable()

```
int CoTable (
            UBYTE * s )  [extern]
```

Definition at line 1923 of file names.c.

4.19.6.430 CoTerm()

```
int CoTerm (
            UBYTE * s )  [extern]
```

Definition at line 5050 of file compcomm.c.

4.19.6.431 CoNTable()

```
int CoNTable (
            UBYTE * s )  [extern]
```

Definition at line 1933 of file names.c.

4.19.6.432 CoCTable()

```
int CoCTable (
            UBYTE * s )  [extern]
```

Definition at line 1943 of file names.c.

4.19.6.433 EmptyTable()

```
void EmptyTable (
            TABLES T )  [extern]
```

Definition at line 1953 of file names.c.

4.19.6.434 CoToTensor()

```
int CoToTensor (
            UBYTE * s )  [extern]
```

Definition at line 2572 of file compcomm.c.

4.19.6.435 CoToVector()

```
int CoToVector (
            UBYTE * s )  [extern]
```

Definition at line 2740 of file compcomm.c.

4.19.6.436 CoTrace4()

```
int CoTrace4 (
            UBYTE * s )  [extern]
```

Definition at line 2810 of file compcomm.c.

4.19.6.437 CoTraceN()

```
int CoTraceN (
            UBYTE * s )  [extern]
```

Definition at line 2897 of file compcomm.c.

4.19.6.438 CoChisholm()

```
int CoChisholm (
            UBYTE * s )  [extern]
```

Definition at line 2957 of file compcomm.c.

4.19.6.439 CoTransform()

```
int CoTransform (
            UBYTE * in )  [extern]
```

Definition at line 87 of file transform.c.

4.19.6.440 CoClearTable()

```
int CoClearTable (
            UBYTE * s )  [extern]
```

Definition at line 5596 of file compcomm.c.

4.19.6.441 DoChain()

```
int DoChain (
            UBYTE * s,
            int option )  [extern]
```

Definition at line 3039 of file compcomm.c.

4.19.6.442 CoChainin()

```
int CoChainin (
            UBYTE * s )  [extern]
```

Definition at line 3083 of file compcomm.c.

4.19.6.443 CoChainout()

```
int CoChainout (
            UBYTE * s )  [extern]
```

Definition at line 3095 of file compcomm.c.

4.19.6.444 CoTryReplace()

```
int CoTryReplace (
            UBYTE * p )  [extern]
```

Definition at line 3319 of file compcomm.c.

4.19.6.445 CoVector()

```
int CoVector (
            UBYTE * s )  [extern]
```

Definition at line 1166 of file names.c.

4.19.6.446 CoWhile()

```
int CoWhile (
            UBYTE * inp )  [extern]
```

Definition at line 4803 of file compcomm.c.

4.19.6.447 CoWrite()

```
int CoWrite (
            UBYTE * s )  [extern]
```

Definition at line 2234 of file compcomm.c.

4.19.6.448 CoAuto()

```
int CoAuto (
            UBYTE * inp )  [extern]
```

Definition at line 2382 of file names.c.

4.19.6.449 CoTBaddto()

```
int CoTBaddto (
            UBYTE * s )  [extern]
```

Definition at line 738 of file tables.c.

4.19.6.450 CoTBaudit()

```
int CoTBaudit (
            UBYTE * s )  [extern]
```

Definition at line 1393 of file tables.c.

4.19.6.451 CoTBcleanup()

```
int CoTBcleanup (
            UBYTE * s )  [extern]
```

Definition at line 1495 of file tables.c.

4.19.6.452 CoTBcreate()

```
int CoTBcreate (
            UBYTE * s )  [extern]
```

Definition at line 704 of file tables.c.

4.19.6.453 CoTBenter()

```
int CoTBenter (
            UBYTE * s )  [extern]
```

Definition at line 906 of file tables.c.

4.19.6.454 CoTBhelp()

```
int CoTBhelp (
            UBYTE * s )  [extern]
```

Definition at line 1800 of file tables.c.

4.19.6.455 CoTBload()

```
int CoTBload (
            UBYTE * ss )  [extern]
```

Definition at line 1178 of file tables.c.

4.19.6.456 CoTBoff()

```
int CoTBoff (
            UBYTE * s )  [extern]
```

Definition at line 1464 of file tables.c.

4.19.6.457 CoTBon()

```
int CoTBon (
            UBYTE * s )  [extern]
```

Definition at line 1433 of file tables.c.

4.19.6.458 CoTBopen()

```
int CoTBopen (
            UBYTE * s )  [extern]
```

Definition at line 720 of file tables.c.

4.19.6.459 CoTBreplace()

```
int CoTBreplace (
            UBYTE * s )  [extern]
```

Definition at line 1507 of file tables.c.

4.19.6.460 CoTBuse()

```
int CoTBuse (
            UBYTE * s )  [extern]
```

Definition at line 1522 of file tables.c.

4.19.6.461 CoTestUse()

```
int CoTestUse (
            UBYTE * s )  [extern]
```

Definition at line 1064 of file tables.c.

4.19.6.462 CoThreadBucket()

```
int CoThreadBucket (
            UBYTE * s )  [extern]
```

Definition at line 5496 of file compcomm.c.

4.19.6.463 AddComString()

```
int AddComString (
            int n,
            WORD * array,
            UBYTE * thestring,
            int par )  [extern]
```

Definition at line 1533 of file compcomm.c.

4.19.6.464 CompileAlgebra()

```
int CompileAlgebra (
            UBYTE * s,
            int leftright,
            WORD * prototype )  [extern]
```

Definition at line 494 of file compiler.c.

4.19.6.465 IsIdStatement()

```
int IsIdStatement (
            UBYTE * s )  [extern]
```

Definition at line 480 of file compiler.c.

4.19.6.466 IsRHS()

```
UBYTE * IsRHS (
            UBYTE * s,
            UBYTE c )  [extern]
```

Definition at line 434 of file compiler.c.

4.19.6.467 ParenthesesTest()

```
int ParenthesesTest (
            UBYTE * sin )  [extern]
```

Definition at line 356 of file compiler.c.

4.19.6.468 tokenize()

```
int tokenize (
            UBYTE * in,
            WORD leftright )  [extern]
```

Definition at line 58 of file token.c.

4.19.6.469 WriteTokens()

```
void WriteTokens (
            SBYTE * in )  [extern]
```

Definition at line 592 of file token.c.

4.19.6.470 simp1token()

```
int simp1token (
            SBYTE * s )  [extern]
```

Definition at line 640 of file token.c.

4.19.6.471 simpwtoken()

```
int simpwtoken (
            SBYTE * s )  [extern]
```

Definition at line 729 of file token.c.

4.19.6.472 simp2token()

```
int simp2token (
            SBYTE * s )  [extern]
```

Definition at line 883 of file token.c.

4.19.6.473 simp3atoken()

```
int simp3atoken (
            SBYTE * s,
            int mode )  [extern]
```

Definition at line 1127 of file token.c.

4.19.6.474 simp3btoken()

```
int simp3btoken (
            SBYTE * s,
            int mode )  [extern]
```

Definition at line 1350 of file token.c.

4.19.6.475 simp4token()

```
int simp4token (
            SBYTE * s )  [extern]
```

Definition at line 1726 of file token.c.

4.19.6.476 simp5token()

```
int simp5token (
            SBYTE * s,
            int mode )  [extern]
```

Definition at line 1883 of file token.c.

4.19.6.477 simp6token()

```
int simp6token (
            SBYTE * tokens,
            int mode )  [extern]
```

Definition at line 1929 of file token.c.

4.19.6.478 SkipAName()

```
UBYTE * SkipAName (
            UBYTE * s )  [extern]
```

Definition at line 406 of file compiler.c.

4.19.6.479 GetLabel()

```
int GetLabel (
            UBYTE * name )  [extern]
```

Definition at line 2596 of file names.c.

4.19.6.480 CoIdExpression()

```
int CoIdExpression (
            UBYTE * inp,
            int type )  [extern]
```

Definition at line 503 of file comexpr.c.

4.19.6.481 CoAssign()

```
int CoAssign (
            UBYTE * inp )  [extern]
```

Definition at line 1901 of file comexpr.c.

4.19.6.482 DoExpr()

```
int DoExpr (
            UBYTE * inp,
            int type,
            int par )  [extern]
```

Definition at line 96 of file comexpr.c.

4.19.6.483 CompileSubExpressions()

```
int CompileSubExpressions (
            SBYTE * tokens )  [extern]
```

Definition at line 688 of file compiler.c.

4.19.6.484 CodeGenerator()

```
int CodeGenerator (
            SBYTE * tokens )  [extern]
```

Definition at line 823 of file compiler.c.

4.19.6.485 CompleteTerm()

```
int CompleteTerm (
            WORD * term,
            UWORD * numer,
            UWORD * denom,
            WORD nnum,
            WORD nden,
            int sign )  [extern]
```

Definition at line 1855 of file compiler.c.

4.19.6.486 CodeFactors()

```
int CodeFactors (
            SBYTE * s )  [extern]
```

Definition at line 1892 of file compiler.c.

4.19.6.487 GenerateFactors()

```
WORD GenerateFactors (
            WORD n,
            WORD inc )  [extern]
```

Definition at line 2189 of file compiler.c.

4.19.6.488 InsTree()

```
int InsTree (
            int bufnum,
            int h )  [extern]
```

Definition at line 355 of file comtool.c.

4.19.6.489 FindTree()

```
int FindTree (
            int bufnum,
            WORD * subexpr )  [extern]
```

Definition at line 533 of file comtool.c.

4.19.6.490 RedoTree()

```
void RedoTree (
            CBUF * C,
            int size )  [extern]
```

Definition at line 569 of file comtool.c.

4.19.6.491 ClearTree()

```
void ClearTree (
            int i )  [extern]
```

Definition at line 588 of file comtool.c.

4.19.6.492 CatchDollar()

```
int CatchDollar (
            int par )  [extern]
```

Definition at line 54 of file dollar.c.

4.19.6.493 AssignDollar()

```
int AssignDollar (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 209 of file dollar.c.

4.19.6.494 WriteDollarToBuffer()

```
UBYTE * WriteDollarToBuffer (
            WORD numdollar,
            WORD par )  [extern]
```

Definition at line 596 of file dollar.c.

4.19.6.495 WriteDollarFactorToBuffer()

```
UBYTE * WriteDollarFactorToBuffer (
            WORD numdollar,
            WORD numfac,
            WORD par )  [extern]
```

Definition at line 687 of file dollar.c.

4.19.6.496 AddToDollarBuffer()

```
void AddToDollarBuffer (
            UBYTE * s )  [extern]
```

Definition at line 759 of file dollar.c.

4.19.6.497 TermAssign()

```
void TermAssign (
            WORD * term )  [extern]
```

Definition at line 800 of file dollar.c.

4.19.6.498 WildDollars()

```
void WildDollars (
            PHEAD WORD * term )  [extern]
```

Definition at line 862 of file dollar.c.

4.19.6.499 numcommute()

```
LONG numcommute (
            WORD * terms,
            LONG * numterms )  [extern]
```

Definition at line 659 of file comtool.c.

4.19.6.500 FullRenumber()

```
int FullRenumber (
            PHEAD WORD * term,
            WORD par )  [extern]
```

Definition at line 323 of file reshuf.c.

4.19.6.501 Lus()

```
int Lus (
            WORD * term,
            WORD funnum,
            WORD loopsize,
            WORD numargs,
            WORD outfun,
            WORD mode )  [extern]
```

Definition at line 57 of file lus.c.

4.19.6.502 FindLus()

```
int FindLus (
            int from,
            int level,
            int openindex )  [extern]
```

Definition at line 515 of file lus.c.

4.19.6.503 CoReplaceLoop()

```
int CoReplaceLoop (
            UBYTE * inp )  [extern]
```

Definition at line 4974 of file compcomm.c.

4.19.6.504 CoFindLoop()

```
int CoFindLoop (
            UBYTE * inp )  [extern]
```

Definition at line 4966 of file compcomm.c.

4.19.6.505 DoFindLoop()

```
int DoFindLoop (
            UBYTE * inp,
            int mode )  [extern]
```

Definition at line 4852 of file compcomm.c.

4.19.6.506 CoFunPowers()

```
int CoFunPowers (
            UBYTE * inp )  [extern]
```

Definition at line 4988 of file compcomm.c.

4.19.6.507 SortTheList()

```
int SortTheList (
            int * slist,
            int num )  [extern]
```

Definition at line 578 of file lus.c.

4.19.6.508 MatchIsPossible()

```
int MatchIsPossible (
            WORD * pattern,
            WORD * term )  [extern]
```

Definition at line 299 of file smart.c.

4.19.6.509 StudyPattern()

```
int StudyPattern (
            WORD * lhs )  [extern]
```

Definition at line 62 of file smart.c.

4.19.6.510 DolToTensor()

```
WORD DolToTensor (
            PHEAD WORD )  [extern]
```

Definition at line 1055 of file dollar.c.

4.19.6.511 DolToFunction()

```
WORD DolToFunction (
            PHEAD WORD )  [extern]
```

Definition at line 1116 of file dollar.c.

4.19.6.512 DolToVector()

```
WORD DolToVector (
            PHEAD WORD )  [extern]
```

Definition at line 1173 of file dollar.c.

4.19.6.513 DolToNumber()

```
WORD DolToNumber (
            PHEAD WORD )  [extern]
```

Definition at line 1237 of file dollar.c.

4.19.6.514 DolToSymbol()

```
WORD DolToSymbol (
            PHEAD WORD )  [extern]
```

Definition at line 1296 of file dollar.c.

4.19.6.515 DolToIndex()

```
WORD DolToIndex (
            PHEAD WORD )  [extern]
```

Definition at line 1350 of file dollar.c.

4.19.6.516 DolToLong()

```
LONG DolToLong (
            PHEAD WORD )  [extern]
```

Definition at line 1573 of file dollar.c.

4.19.6.517 DollarFactorize()

```
int DollarFactorize (
            PHEAD WORD )  [extern]
```

Definition at line 2861 of file dollar.c.

4.19.6.518 CoPrintTable()

```
int CoPrintTable (
            UBYTE * inp )  [extern]
```

Definition at line 1725 of file comexpr.c.

4.19.6.519 CoDeallocateTable()

```
int CoDeallocateTable (
            UBYTE * inp )  [extern]
```

Definition at line 1959 of file comexpr.c.

4.19.6.520 CleanDollarFactors()

```
void CleanDollarFactors (
            DOLLARS d )  [extern]
```

Definition at line 3460 of file dollar.c.

4.19.6.521 TakeDollarContent()

```
WORD * TakeDollarContent (
            PHEAD WORD * dollarbuffer,
            WORD ** factor )  [extern]
```

Definition at line 3481 of file dollar.c.

4.19.6.522 MakeDollarInteger()

```
WORD * MakeDollarInteger (
            PHEAD WORD * bufin,
            WORD ** bufout )  [extern]
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in bufin. The number that comes out is the return value. The normalized argument is in bufout.

Definition at line 3533 of file dollar.c.

References EndSort(), Generator(), and NewSort().

Here is the call graph for this function:

4.19.6.523 MakeDollarMod()

```
WORD * MakeDollarMod (
            PHEAD WORD * buffer,
            WORD ** bufout )  [extern]
```

Similar to MakeDollarInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3707 of file dollar.c.

References EndSort(), Generator(), GetModInverses(), and NewSort().

Here is the call graph for this function:

4.19.6.524 GetDolNum()

```
int GetDolNum (
            PHEAD WORD * t,
            WORD * tstop )  [extern]
```

Definition at line 3752 of file dollar.c.

4.19.6.525 AddPotModdollar()

```
void AddPotModdollar (
            WORD numdollar )  [extern]
```

Adds a $-variable specified by *numdollar* to the list of potentially modified $-variables unless it has already been included in the list.

Parameters

numdollar	The index of the $-variable to be added.

Definition at line 3865 of file dollar.c.

4.19.6.526 TakeLongRoot()

```
int TakeLongRoot (
            UWORD * a,
            WORD * n,
            WORD power )  [extern]
```

Definition at line 2725 of file reken.c.

4.19.6.527 TakeRatRoot()

```
int TakeRatRoot (
            UWORD * a,
            WORD * n,
            WORD power )  [extern]
```

Definition at line 680 of file reken.c.

4.19.6.528 MakeRational()

```
int MakeRational (
            WORD a,
            WORD m,
            WORD * b,
            WORD * c )  [extern]
```

Definition at line 2845 of file reken.c.

4.19.6.529 MakeLongRational()

```
int MakeLongRational (
            PHEAD UWORD * a,
            WORD na,
            UWORD * m,
            WORD nm,
            UWORD * b,
            WORD * nb )  [extern]
```

Definition at line 2893 of file reken.c.

4.19.6.530 ClearTableTree()

```
void ClearTableTree (
            TABLES T )  [extern]
```

Definition at line 71 of file tables.c.

4.19.6.531 InsTableTree()

```
int InsTableTree (
            TABLES T,
            WORD * tp )  [extern]
```

Definition at line 102 of file tables.c.

4.19.6.532 RedoTableTree()

```
void RedoTableTree (
            TABLES T,
            int newsize )  [extern]
```

Definition at line 263 of file tables.c.

4.19.6.533 FindTableTree()

```
int FindTableTree (
            TABLES T,
            WORD * tp,
            int inc )  [extern]
```

Definition at line 288 of file tables.c.

4.19.6.534 finishcbuf()

```
void finishcbuf (
            WORD num )  [extern]
```

Frees a compiler buffer.

Parameters

num	The ID number for the buffer to be freed.

Definition at line 89 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::lhs, CbUf::NumTerms, CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Referenced by PF_BroadcastCBuf().

4.19.6.535 clearcbuf()

```
void clearcbuf (
            WORD num )  [extern]
```

Clears contents in a compiler buffer.

Parameters

num	The ID number for the buffer to be cleared.

Definition at line 116 of file comtool.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::Pointer, and CbUf::rhs.

4.19.6.536 CleanUpSort()

```
void CleanUpSort (
            int num )  [extern]
```

Partially or completely frees function sort buffers.

Definition at line 4536 of file sort.c.

4.19.6.537 AllocFileHandle()

```
FILEHANDLE * AllocFileHandle (
            WORD par,
            char * name )  [extern]
```

Definition at line 993 of file setfile.c.

4.19.6.538 DeAllocFileHandle()

```
VOID DeAllocFileHandle (
            FILEHANDLE * fh )  [extern]
```

Definition at line 1053 of file setfile.c.

4.19.6.539 PolyRatFunSpecial()

```
WORD * PolyRatFunSpecial (
            PHEAD WORD * t1,
            WORD * t2 )  [extern]
```

Definition at line 4627 of file sort.c.

4.19.6.540 InsideDollar()

```
int InsideDollar (
            PHEAD WORD * ll,
            WORD level )  [extern]
```

Definition at line 1713 of file dollar.c.

4.19.6.541 DolToTerms()

```
DOLLARS DolToTerms (
            PHEAD WORD )  [extern]
```

Definition at line 1427 of file dollar.c.

4.19.6.542 EvalDoLoopArg()

```
WORD EvalDoLoopArg (
            PHEAD WORD * arg,
            WORD par )  [extern]
```

Evaluates one argument of a do loop. Such an argument is constructed from SNUMBERs DOLLAREXPRESSIONs and possibly DOLLAREXPR2s which indicate factors of the preceeding dollar. Hence we have SNUMBER,num DOLLAREXPRESSION,numdollar DOLLAREXPRESSION,numdollar,DOLLAREXPR2,numfactor DOLLAREXPRESSION,numdollar,DOLLAREXPR2,numfactor,DOLLAREXPR2,numfactor etc. Because we have a do-loop at every stage we should have a number. The notation in DOLLAREXPR2 is that $>=$ 0 is number of yet another dollar and $<$ 0 is -n-1 with n the array element or zero. The return value is the (short) number. The routine works its way through the list in a recursive manner.

Definition at line 2557 of file dollar.c.

References EvalDoLoopArg().

Referenced by EvalDoLoopArg().

Here is the call graph for this function:

4.19.6.543 SetExprCases()

```
int SetExprCases (
            int par,
            int setunset,
            int val )  [extern]
```

Definition at line 1242 of file compcomm.c.

4.19.6.544 TestSelect()

```
int TestSelect (
            WORD * term,
            WORD * setp )  [extern]
```

Definition at line 1748 of file pattern.c.

4.19.6.545 SubsInAll()

```
VOID SubsInAll (
            PHEAD0  )  [extern]
```

Definition at line 1982 of file pattern.c.

4.19.6.546 TransferBuffer()

```
VOID TransferBuffer (
            int from,
            int to,
            int spectator )  [extern]
```

Definition at line 2143 of file pattern.c.

4.19.6.547 TakeIDfunction()

```
int TakeIDfunction (
            PHEAD WORD * term )  [extern]
```

Definition at line 2213 of file pattern.c.

4.19.6.548 ExchangeExpressions()

```
void ExchangeExpressions (
            int num1,
            int num2 )  [extern]
```

Definition at line 1586 of file execute.c.

4.19.6.549 ExchangeDollars()

```
void ExchangeDollars (
            int num1,
            int num2 )  [extern]
```

Definition at line 1807 of file dollar.c.

4.19.6.550 GetFirstBracket()

```
int GetFirstBracket (
            WORD * term,
            int num )  [extern]
```

Definition at line 1660 of file execute.c.

4.19.6.551 GetFirstTerm()

```
int GetFirstTerm (
            WORD * term,
            int num )  [extern]
```

Definition at line 1770 of file execute.c.

4.19.6.552 GetContent()

```
int GetContent (
            WORD * content,
            int num )  [extern]
```

Definition at line 1865 of file execute.c.

4.19.6.553 CleanupTerm()

```
int CleanupTerm (
            WORD * term )  [extern]
```

Definition at line 1982 of file execute.c.

4.19.6.554 ContentMerge()

```
WORD ContentMerge (
            PHEAD WORD * content,
            WORD * term )  [extern]
```

Definition at line 2006 of file execute.c.

4.19.6.555 PreIfDollarEval()

```
UBYTE * PreIfDollarEval (
            UBYTE * s,
            int * value )  [extern]
```

Definition at line 1884 of file dollar.c.

4.19.6.556 TermsInDollar()

```
LONG TermsInDollar (
            WORD num )  [extern]
```

Definition at line 1825 of file dollar.c.

4.19.6.557 TermsInExpression()

```
LONG TermsInExpression (
            WORD num )  [extern]
```

Definition at line 2226 of file execute.c.

4.19.6.558 TranslateExpression()

```
WORD * TranslateExpression (
            UBYTE * s )  [extern]
```

Definition at line 2066 of file dollar.c.

4.19.6.559 IsSetMember()

```
int IsSetMember (
            WORD * buffer,
            WORD numset )  [extern]
```

Definition at line 2123 of file dollar.c.

4.19.6.560 IsMultipleOf()

```
int IsMultipleOf (
            WORD * buf1,
            WORD * buf2 )  [extern]
```

Definition at line 2304 of file dollar.c.

4.19.6.561 TwoExprCompare()

```
int TwoExprCompare (
            WORD * buf1,
            WORD * buf2,
            int oprtr )  [extern]
```

Definition at line 2379 of file dollar.c.

4.19.6.562 FindBracket()

```
POSITION * FindBracket (
            WORD nexp,
            WORD * bracket )  [extern]
```

Definition at line 65 of file index.c.

4.19.6.563 PutBracketInIndex()

```
VOID PutBracketInIndex (
            PHEAD WORD * term,
            POSITION * newpos )  [extern]
```

Definition at line 331 of file index.c.

4.19.6.564 ClearBracketIndex()

```
void ClearBracketIndex (
            WORD numexp )  [extern]
```

Definition at line 546 of file index.c.

4.19.6.565 OpenBracketIndex()

```
VOID OpenBracketIndex (
            WORD nexpr )  [extern]
```

Definition at line 578 of file index.c.

4.19.6.566 DoNoParallel()

```
int DoNoParallel (
            UBYTE * s )  [extern]
```

Definition at line 508 of file module.c.

4.19.6.567 DoParallel()

```
int DoParallel (
            UBYTE * s )  [extern]
```

Definition at line 523 of file module.c.

4.19.6.568 DoModSum()

```
int DoModSum (
            UBYTE * s )  [extern]
```

Definition at line 538 of file module.c.

4.19.6.569 DoModMax()

```
int DoModMax (
            UBYTE * s )  [extern]
```

Definition at line 558 of file module.c.

4.19.6.570 DoModMin()

```
int DoModMin (
            UBYTE * s )  [extern]
```

Definition at line 578 of file module.c.

4.19.6.571 DoModLocal()

```
int DoModLocal (
            UBYTE * s )  [extern]
```

Definition at line 598 of file module.c.

4.19.6.572 DoModDollar()

```
UBYTE * DoModDollar (
            UBYTE * s,
            int type )  [extern]
```

Definition at line 636 of file module.c.

4.19.6.573 DoProcessBucket()

```
int DoProcessBucket (
            UBYTE * s )  [extern]
```

Definition at line 618 of file module.c.

4.19.6.574 DoinParallel()

```
int DoinParallel (
            UBYTE * s )  [extern]
```

Definition at line 737 of file module.c.

4.19.6.575 DonotinParallel()

```
int DonotinParallel (
            UBYTE * s )  [extern]
```

Definition at line 747 of file module.c.

4.19.6.576 FlipTable()

```
int FlipTable (
            FUNCTIONS f,
            int type )  [extern]
```

Definition at line 619 of file tables.c.

4.19.6.577 ChainIn()

```
int ChainIn (
            PHEAD WORD * term,
            WORD funnum )  [extern]
```

Definition at line 691 of file function.c.

4.19.6.578 ChainOut()

```
int ChainOut (
            PHEAD WORD * term,
            WORD funnum )  [extern]
```

Definition at line 739 of file function.c.

4.19.6.579 ArgumentImplode()

```
int ArgumentImplode (
            PHEAD WORD * term,
            WORD * thelist )  [extern]
```

Definition at line 1800 of file argument.c.

4.19.6.580 ArgumentExplode()

```
int ArgumentExplode (
            PHEAD WORD * term,
            WORD * thelist )  [extern]
```

Definition at line 1904 of file argument.c.

4.19.6.581 DenToFunction()

```
int DenToFunction (
            WORD * term,
            WORD numfun )  [extern]
```

Definition at line 2472 of file wildcard.c.

4.19.6.582 HowMany()

```
WORD HowMany (
            PHEAD WORD * ifcode,
            WORD * term )  [extern]
```

Definition at line 816 of file if.c.

4.19.6.583 CountTerms1()

```
LONG CountTerms1 (
            PHEAD0  )  [extern]
```

Definition at line 2292 of file execute.c.

4.19.6.584 TermsInBracket()

```
LONG TermsInBracket (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 2400 of file execute.c.

4.19.6.585 str_dup()

```
char * str_dup (
            char * str )  [extern]
```

Definition at line 101 of file minos.c.

4.19.6.586 convertblock()

```
void convertblock (
            INDEXBLOCK * in,
            INDEXBLOCK * out,
            int mode )  [extern]
```

Definition at line 120 of file minos.c.

4.19.6.587 convertnamesblock()

```
void convertnamesblock (
            NAMESBLOCK * in,
            NAMESBLOCK * out,
            int mode )  [extern]
```

Definition at line 171 of file minos.c.

4.19.6.588 convertiniinfo()

```
void convertiniinfo (
            INIINFO * in,
            INIINFO * out,
            int mode )  [extern]
```

Definition at line 197 of file minos.c.

4.19.6.589 ReadIndex()

```
int ReadIndex (
            DBASE * d )  [extern]
```

Definition at line 287 of file minos.c.

4.19.6.590 WriteIndexBlock()

```
int WriteIndexBlock (
            DBASE * d,
            MLONG num )  [extern]
```

Definition at line 400 of file minos.c.

4.19.6.591 WriteNamesBlock()

```
int WriteNamesBlock (
            DBASE * d,
            MLONG num )  [extern]
```

Definition at line 421 of file minos.c.

4.19.6.592 WriteIndex()

```
int WriteIndex (
            DBASE * d )  [extern]
```

Definition at line 444 of file minos.c.

4.19.6.593 WriteIniInfo()

```
int WriteIniInfo (
            DBASE * d )  [extern]
```

Definition at line 507 of file minos.c.

4.19.6.594 ReadIniInfo()

```
int ReadIniInfo (
            DBASE * d )  [extern]
```

Definition at line 524 of file minos.c.

4.19.6.595 AddToIndex()

```
int AddToIndex (
            DBASE * d,
            MLONG number )  [extern]
```

Definition at line 1049 of file minos.c.

4.19.6.596 GetDbase()

```
DBASE * GetDbase (
            char * filename )  [extern]
```

Definition at line 547 of file minos.c.

4.19.6.597 OpenDbase()

```
DBASE * OpenDbase (
            char * filename )  [extern]
```

Definition at line 806 of file minos.c.

4.19.6.598 ReadObject()

```
char * ReadObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments )  [extern]
```

Definition at line 1269 of file minos.c.

4.19.6.599 ReadijObject()

```
char * ReadijObject (
            DBASE * d,
            MLONG i,
            MLONG j,
            char * arguments )  [extern]
```

Definition at line 1348 of file minos.c.

4.19.6.600 ExistsObject()

```
int ExistsObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments )  [extern]
```

Definition at line 1400 of file minos.c.

4.19.6.601 DeleteObject()

```
int DeleteObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments )  [extern]
```

Definition at line 1432 of file minos.c.

4.19.6.602 WriteObject()

```
int WriteObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments,
            char * rhs,
            MLONG number )  [extern]
```

Definition at line 1178 of file minos.c.

4.19.6.603 AddObject()

```
MLONG AddObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments,
            char * rhs )  [extern]
```

Definition at line 1136 of file minos.c.

4.19.6.604 NewDbase()

```
DBASE * NewDbase (
            char * name,
            MLONG number )  [extern]
```

Definition at line 591 of file minos.c.

4.19.6.605 FreeTableBase()

```
void FreeTableBase (
            DBASE * d )  [extern]
```

Definition at line 725 of file minos.c.

4.19.6.606 ComposeTableNames()

```
int ComposeTableNames (
            DBASE * d )  [extern]
```

Definition at line 757 of file minos.c.

4.19.6.607 PutTableNames()

```
int PutTableNames (
            DBASE * d )  [extern]
```

Definition at line 955 of file minos.c.

4.19.6.608 AddTableName()

```
MLONG AddTableName (
            DBASE * d,
            char * name,
            TABLES T )  [extern]
```

Definition at line 840 of file minos.c.

4.19.6.609 GetTableName()

```
MLONG GetTableName (
            DBASE * d,
            char * name )  [extern]
```

Definition at line 923 of file minos.c.

4.19.6.610 FindTableNumber()

```
MLONG FindTableNumber (
            DBASE * d,
            char * name )  [extern]
```

Definition at line 1150 of file minos.c.

4.19.6.611 CopyExpression()

```
int CopyExpression (
            FILEHANDLE * from,
            FILEHANDLE * to )  [extern]
```

Definition at line 3262 of file store.c.

4.19.6.612 set_in()

```
int set_in (
            UBYTE ch,
            set_of_char set )  [extern]
```

Definition at line 2026 of file tools.c.

4.19.6.613 set_set()

```
one_byte set_set (
            UBYTE ch,
            set_of_char set )  [extern]
```

Definition at line 2046 of file tools.c.

4.19.6.614 set_del()

```
one_byte set_del (
            UBYTE ch,
            set_of_char set )  [extern]
```

Definition at line 2067 of file tools.c.

4.19.6.615 set_sub()

```
one_byte set_sub (
            set_of_char set,
            set_of_char set1,
            set_of_char set2 )  [extern]
```

Definition at line 2089 of file tools.c.

4.19.6.616 DoPreAddSeparator()

```
int DoPreAddSeparator (
            UBYTE * s )  [extern]
```

Definition at line 5118 of file pre.c.

4.19.6.617 DoPreRmSeparator()

```
int DoPreRmSeparator (
            UBYTE * s )  [extern]
```

Definition at line 5143 of file pre.c.

4.19.6.618 openExternalChannel()

```
int openExternalChannel (
            UBYTE * cmd,
            int daemonize,
            UBYTE * shellname,
            UBYTE * stderrname )  [extern]
```

Definition at line 236 of file extcmd.c.

4.19.6.619 initPresetExternalChannels()

```
int initPresetExternalChannels (
            UBYTE * theline,
            int thetimeout )  [extern]
```

Definition at line 238 of file extcmd.c.

4.19.6.620 closeExternalChannel()

```
int closeExternalChannel (
            int n )  [extern]
```

Definition at line 239 of file extcmd.c.

4.19.6.621 selectExternalChannel()

```
int selectExternalChannel (
            int n )  [extern]
```

Definition at line 240 of file extcmd.c.

4.19.6.622 defineChannel()

```
UBYTE * defineChannel (
            UBYTE * s,
            HANDLERS * h )  [extern]
```

Definition at line 5768 of file pre.c.

4.19.6.623 writeToChannel()

```
int writeToChannel (
            int wtype,
            UBYTE * s,
            HANDLERS * h )  [extern]
```

Definition at line 5803 of file pre.c.

4.19.6.624 SymbolNormalize()

```
int SymbolNormalize (
            WORD * term )  [extern]
```

Routine normalizes terms that contain only symbols. Regular minimum and maximum properties are ignored.

We check whether there are negative powers in the output. This is not allowed.

Definition at line 4979 of file normal.c.

Referenced by InFunction().

4.19.6.625 TestFunFlag()

```
int TestFunFlag (
            PHEAD WORD * tfun )  [extern]
```

Definition at line 5075 of file normal.c.

4.19.6.626 CompareSymbols()

```
int CompareSymbols (
            PHEAD WORD * term1,
            WORD * term2,
            WORD par )  [extern]
```

Compares the terms, based on the value of AN.polysortflag. If term1 $<$ term2 the return value is -1 If term1 $>$ term2 the return value is 1 If term1 = term2 the return value is 0 The coefficients may differ. The terms contain only a single subterm of type SYMBOL. If AN.polysortflag = 0 it is a 'regular' compare. If AN.polysortflag = 1 the sum of the powers is more important par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in sort.c

Definition at line 2945 of file sort.c.

Referenced by InFunction(), and TestSub().

4.19.6.627 CompareHSymbols()

```
int CompareHSymbols (
            PHEAD WORD * term1,
            WORD * term2,
            WORD par )  [extern]
```

Compares terms that can have only SYMBOL and HAAKJE subterms. If term1 $<$ term2 the return value is -1 If term1 $>$ term2 the return value is 1 If term1 = term2 the return value is 0 par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in sort.c

Definition at line 2988 of file sort.c.

4.19.6.628 NextPrime()

```
WORD NextPrime (
            PHEAD WORD num )  [extern]
```

Gives the next prime number in the list of prime numbers.

If the list isn't long enough we expand it. For ease in ParForm and because these lists shouldn't be very big we let each worker keep its own list.

The list is cut off at MAXPOWER, because we don't want to get into trouble that the power of a variable gets larger than the prime number.

Definition at line 3654 of file reken.c.

4.19.6.629 wranf()

```
UWORD wranf (
            PHEAD0  )  [extern]
```

Definition at line 3793 of file reken.c.

4.19.6.630 iranf()

```
UWORD iranf (
            PHEAD UWORD )  [extern]
```

Definition at line 3812 of file reken.c.

4.19.6.631 iniwranf()

```
void iniwranf (
            PHEAD0  )  [extern]
```

Definition at line 3744 of file reken.c.

4.19.6.632 PreRandom()

```
UBYTE * PreRandom (
            UBYTE * s )  [extern]
```

Definition at line 3834 of file reken.c.

4.19.6.633 TreatPolyRatFun()

```
int TreatPolyRatFun (
            PHEAD WORD * prf )  [extern]
```

Definition at line 4795 of file normal.c.

4.19.6.634 ReadSaveIndex()

```
WORD ReadSaveIndex (
            FILEINDEX * fileind )  [extern]
```

Reads a FILEINDEX from the open save file specified by AO.SaveData.Handle. Translations for adjusting endianness and data sizes are done if necessary.

Depends on the assumption that sizeof(FILEINDEX) is the same everywhere. If FILEINDEX or INDEXENTRY change, then this functions has to be adjusted.

Called by CoLoad() and FindInIndex().

Parameters

fileind	contains the read FILEINDEX after succesful return. must point to allocated, big enough memory.

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4137 of file store.c.

References INFILEINDEX, FiLeInDeX::next, FiLeInDeX::number, and FuNcTiOn::number.

4.19.6.635 ReadSaveExpression()

```
WORD ReadSaveExpression (
            UBYTE * buffer,
            UBYTE * top,
            LONG * size,
            LONG * outsize )  [extern]
```

Reads an expression from the open file specified by AO.SaveData.Handle. The endianness flip and a resizing without renumbering is done in this function. Thereafter the buffer consists of chunks with a uniform maximal word

size (32bit at the moment). The actual renumbering is then done by calling the function ReadSaveTerm32(). The result is returned in *buffer*.

If the translation at some point doesn't fit into the buffer anymore, the function returns and must be called again. In any case *size* returns the number of successfully read bytes, *outsize* returns the number of successfully written bytes, and the file will be positioned at the next byte after the successfully read data.

It is called by PutInStore().

Parameters

buffer	output buffer, holds the (translated) expression
top	end of buffer
size	number of read bytes
outsize	number of written bytes

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 5092 of file store.c.

References ReadSaveTerm32().

Here is the call graph for this function:

4.19.6.636 ReadSaveTerm32()

```
UBYTE * ReadSaveTerm32 (
            UBYTE * bin,
            UBYTE * binend,
            UBYTE ** bout,
            UBYTE * boutend,
            UBYTE * top,
            int terminbuf )  [extern]
```

Reads a single term from the given buffer at *bin* and write the translated term back to this buffer at *bout*.

ReadSaveTerm32() is currently the only instantiation of a ReadSaveTerm-function. It only deals with data that already has the correct endianness and that is resized to 32bit words but without being renumbered or translated in any other way. It uses the compress buffer AR.CompressBuffer.

The function is reentrant in order to cope with nested function arguments. It is called by ReadSaveExpression() and itself.

The *return value* indicates the position in the input buffer up to which the data has already been successfully processed. The parameter *bout* returns the corresponding position in the output buffer.

Parameters

bin	start of the input buffer
binend	end of the input buffer
bout	as input points to the beginning of the output buffer, as output points behind the already translated data in the output buffer
boutend	end of already decompressed data in output buffer
top	end of output buffer
terminbuf	flag whether decompressed data is already in the output buffer. used in recursive calls

Returns

pointer to the next unprocessed data in the input buffer

Definition at line 4683 of file store.c.

References ReadSaveTerm32().

Referenced by ReadSaveExpression(), and ReadSaveTerm32().

Here is the call graph for this function:

4.19.6.637 ReadSaveVariables()

```
WORD ReadSaveVariables (
            UBYTE * buffer,
            UBYTE * top,
            LONG * size,
            LONG * outsize,
            INDEXENTRY * ind,
            LONG * stage )  [extern]
```

Reads the variables from the open file specified by AO.SaveData.Handle. It reads the ∗size bytes and writes them to the ∗buffer. It is called by PutInStore().

If translation is necessary, the data might shrink or grow in size, then ∗*size* is adjusted so that the reading and writing fits into the memory from the buffer to the top. The actual number of read bytes is returned in ∗*size*, the number of written bytes is returned in ∗*outsize*.

If the ∗size is smaller than the actual size of the variables, this function will be called several times and needs to remember the current position in the variable structure. The parameter *stage* does this job. When ReadSaveVariables() is called for the first time, this parameter should have the value -1.

The parameter *ind* is used to get the number of variables.

Parameters

buffer	read variables are written into this allocated memory
top	upper end of allocated memory
size	number of bytes to read. might return a smaller number of read bytes if translation was necessary
outsize	if translation has be done, outsize contains the number of written bytes
ind	pointer of INDEXENTRY for the current expression. read-only
stage	should be -1 for the first call, will be increased by ReadSaveVariables to memorize the position in the variable structure

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4323 of file store.c.

References InDeXeNtRy::nfunctions, InDeXeNtRy::nindices, InDeXeNtRy::nsymbols, InDeXeNtRy::nvectors, and FuNcTiOn::spec.

4.19.6.638 WriteStoreHeader()

```
WORD WriteStoreHeader (
            WORD handle )  [extern]
```

Writes header with information about system architecture and FORM revision to an open store file.

Called by SetFileIndex().

Parameters

handle	specifies open file to which header will be written

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 3926 of file store.c.

References STOREHEADER::endianness, STOREHEADER::lenLONG, STOREHEADER::lenPOINTER, STOREHEADER::lenPOS, STOREHEADER::lenWORD, STOREHEADER::maxpower, STOREHEADER::sFun, STOREHEADER::sInd, STOREHEADER::sSym, STOREHEADER::sVec, and STOREHEADER::wildoffset.

Referenced by SetFileIndex().

4.19.6.639 DoRecovery()

```
int DoRecovery (
            int * moduletype )  [extern]
```

Reads from the recovery file and restores all necessary variables and states in FORM, so that the execution can recommence in preprocessor() as if no restart of FORM had occurred.

The recovery file is read into memory as a whole. The pointer p then points into this memory at the next non-processed data. The macros by which variables are restored, like R_SET, automatically increase p appropriately.

If something goes wrong, the function returns with a non-zero value.

Allocated memory that would be lost when overwriting the global structs with data from the file is freed first. A major part of the code deals with the restoration of pointers. The idiom we use is to memorize the original pointer value (org), allocate new memory and copy the data from the file into this memory, calculate the offset between the old pointer value and the new allocated memory position (ofs), and then correct all affected pointers (+=ofs).

We rely on the fact that several variables (especially in AM) are already assigned the correct values by the startup functions. That means, in principle, that a change in the setup files between snapshot creation and recovery will be noticed.

Definition at line 1399 of file checkpoint.c.

References TaBlEs::argtail, TaBlEs::boomlijst, BrAcKeTiNfO::bracketbuffer, TaBlEs::buffers, TaBlEs::bufferssize, CopyFile(), TaBlEs::flags, ReNuMbEr::func, ReNuMbEr::funnum, VaRrEnUm::hi, BrAcKeTiNfO::indexbuffer, ReNuMbEr::indi, ReNuMbEr::indnum, VaRrEnUm::lo, TaBlEs::MaxTreeSize, TaBlEs::mm, TaBlEs::numind, TaBlEs::pattern, TaBlEs::prototype, TaBlEs::prototypeSize, RecoveryFilename(), ExPrEsSiOn::renumlists, TaBlEs::reserved, TaBlEs::spare, TaBlEs::sparse, VaRrEnUm::start, ReNuMbEr::symb, ReNuMbEr::symnum, TaBlEs::tablepointers, TimeWallClock(), TaBlEs::totind, ReNuMbEr::vecnum, and ReNuMbEr::vect.

Here is the call graph for this function:

4.19.6.640 DoCheckpoint()

```
void DoCheckpoint (
            int moduletype )  [extern]
```

Checks whether a snapshot should be done. Calls DoSnapshot() to create the snapshot.

Definition at line 3102 of file checkpoint.c.

References PF_BroadcastNumber(), PF_RecvFile(), PF_SendFile(), and TimeWallClock().

Here is the call graph for this function:

4.19.6.641 NumberMallocAddMemory()

```
VOID NumberMallocAddMemory (
            PHEAD0  )  [extern]
```

Definition at line 2567 of file tools.c.

4.19.6.642 CacheNumberMallocAddMemory()

```
VOID CacheNumberMallocAddMemory (
            PHEAD0  )  [extern]
```

Definition at line 2641 of file tools.c.

4.19.6.643 TermMallocAddMemory()

```
VOID TermMallocAddMemory (
            PHEAD0  )  [extern]
```

Definition at line 2463 of file tools.c.

4.19.6.644 ExprStatus()

```
void ExprStatus (
            EXPRESSIONS e )  [extern]
```

Definition at line 66 of file bugtool.c.

4.19.6.645 iniTools()

```
VOID iniTools (
            VOID  )  [extern]
```

Definition at line 2115 of file tools.c.

4.19.6.646 TestTerm()

```
int TestTerm (
            WORD * term )  [extern]
```

Tests the consistency of the term. Returns 0 when the term is OK. Any nonzero value is trouble. In the current version the testing isn't 100% complete. For instance, we don't check the validity of the symbols nor do we check the range of their powers. Etc. This should be extended when the need is there.

Parameters

term	the term to be tested

Definition at line 3729 of file tools.c.

References TestTerm().

Referenced by TestTerm().

Here is the call graph for this function:

4.19.6.647 RunTransform()

```
WORD RunTransform (
            PHEAD WORD * term,
            WORD * params )  [extern]
```

Definition at line 689 of file transform.c.

4.19.6.648 RunEncode()

```
WORD RunEncode (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 922 of file transform.c.

4.19.6.649 RunDecode()

```
WORD RunDecode (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 1109 of file transform.c.

4.19.6.650 RunReplace()

```
WORD RunReplace (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 1279 of file transform.c.

4.19.6.651 RunImplode()

```
WORD RunImplode (
            WORD * fun,
            WORD * args )  [extern]
```

Definition at line 1757 of file transform.c.

4.19.6.652 RunExplode()

```
WORD RunExplode (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 1958 of file transform.c.

4.19.6.653 TestArgNum()

```
int TestArgNum (
            int n,
            int totarg,
            WORD * args )  [extern]
```

Definition at line 3103 of file transform.c.

4.19.6.654 PutArgInScratch()

```
WORD PutArgInScratch (
            WORD * arg,
            UWORD * scrat )  [extern]
```

Definition at line 3172 of file transform.c.

4.19.6.655 ReadRange()

```
UBYTE * ReadRange (
            UBYTE * s,
            WORD * out,
            int par )  [extern]
```

Definition at line 3208 of file transform.c.

4.19.6.656 FindRange()

```
int FindRange (
            PHEAD WORD * args,
            WORD * arg1,
            WORD * arg2,
            WORD totarg )  [extern]
```

Definition at line 3368 of file transform.c.

4.19.6.657 RunPermute()

```
WORD RunPermute (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 2072 of file transform.c.

4.19.6.658 RunReverse()

```
WORD RunReverse (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 2299 of file transform.c.

4.19.6.659 RunCycle()

```
WORD RunCycle (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )  [extern]
```

Definition at line 2475 of file transform.c.

4.19.6.660 RunAddArg()

```
WORD RunAddArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 2627 of file transform.c.

4.19.6.661 RunMulArg()

```
WORD RunMulArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 2710 of file transform.c.

4.19.6.662 RunIsLyndon()

```
WORD RunIsLyndon (
            PHEAD WORD * fun,
            WORD * args,
            int par )  [extern]
```

Definition at line 2849 of file transform.c.

4.19.6.663 RunToLyndon()

```
WORD RunToLyndon (
            PHEAD WORD * fun,
            WORD * args,
            int par )  [extern]
```

Definition at line 2927 of file transform.c.

4.19.6.664 RunDropArg()

```
WORD RunDropArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 3039 of file transform.c.

4.19.6.665 RunSelectArg()

```
WORD RunSelectArg (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 3065 of file transform.c.

4.19.6.666 RunDedup()

```
WORD RunDedup (
            PHEAD WORD * fun,
            WORD * args )  [extern]
```

Definition at line 2386 of file transform.c.

4.19.6.667 NormPolyTerm()

```
int NormPolyTerm (
            PHEAD WORD * term )  [extern]
```

Definition at line 53 of file notation.c.

4.19.6.668 ConvertToPoly()

```
int ConvertToPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD * comlist,
            WORD par )  [extern]
```

Definition at line 307 of file notation.c.

4.19.6.669 LocalConvertToPoly()

```
int LocalConvertToPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD startebuf,
            WORD par )  [extern]
```

Converts a generic term to polynomial notation in which there are only symbols and brackets. During conversion there will be only symbols. Brackets are stripped. Objects that need 'translation' are put inside a special compiler buffer and represented by a symbol. The numbering of the extra symbols is down from the maximum. In principle there can be a problem when running into the already assigned ones. This uses the FindTree for searching in the global tree and then looks further in the AT.ebufnum. This allows fully parallel processing. Hence we need no locks. Cannot be used in the same module as ConvertToPoly.

Definition at line 510 of file notation.c.

4.19.6.670 ConvertFromPoly()

```
int ConvertFromPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD from,
            WORD to,
            WORD offset,
            WORD par )  [extern]
```

Definition at line 660 of file notation.c.

4.19.6.671 FindSubterm()

```
WORD FindSubterm (
            WORD * subterm )  [extern]
```

Definition at line 773 of file notation.c.

4.19.6.672 FindLocalSubterm()

```
WORD FindLocalSubterm (
            PHEAD WORD * subterm,
            WORD startebuf )  [extern]
```

Definition at line 842 of file notation.c.

4.19.6.673 PrintSubtermList()

```
void PrintSubtermList (
            int from,
            int to )  [extern]
```

Definition at line 895 of file notation.c.

4.19.6.674 PrintExtraSymbol()

```
void PrintExtraSymbol (
            int num,
            WORD * terms,
            int par )  [extern]
```

Definition at line 1007 of file notation.c.

4.19.6.675 FindSubexpression()

```
WORD FindSubexpression (
            WORD * subexpr )  [extern]
```

Definition at line 1093 of file notation.c.

4.19.6.676 CoToPolynomial()

```
int CoToPolynomial (
            UBYTE * inp )  [extern]
```

Definition at line 5744 of file compcomm.c.

4.19.6.677 CoFromPolynomial()

```
int CoFromPolynomial (
            UBYTE * inp )  [extern]
```

Definition at line 5819 of file compcomm.c.

4.19.6.678 CoArgToExtraSymbol()

```
int CoArgToExtraSymbol (
            UBYTE * s )  [extern]
```

Definition at line 5845 of file compcomm.c.

4.19.6.679 CoExtraSymbols()

```
int CoExtraSymbols (
            UBYTE * inp )  [extern]
```

Definition at line 5895 of file compcomm.c.

4.19.6.680 GetDoParam()

```
UBYTE * GetDoParam (
            UBYTE * inp,
            WORD ** wp,
            int par )  [extern]
```

Definition at line 6026 of file compcomm.c.

4.19.6.681 GetIfDollarFactor()

```
WORD * GetIfDollarFactor (
            UBYTE ** inp,
            WORD * w )  [extern]
```

Definition at line 5968 of file compcomm.c.

4.19.6.682 CoDo()

```
int CoDo (
            UBYTE * inp )  [extern]
```

Definition at line 6103 of file compcomm.c.

4.19.6.683 CoEndDo()

```
int CoEndDo (
            UBYTE * inp )  [extern]
```

Definition at line 6206 of file compcomm.c.

4.19.6.684 ExtraSymFun()

```
int ExtraSymFun (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 1145 of file notation.c.

4.19.6.685 PruneExtraSymbols()

```
int PruneExtraSymbols (
            WORD downto )  [extern]
```

Definition at line 1213 of file notation.c.

4.19.6.686 IniFbuffer()

```
int IniFbuffer (
            WORD bufnum )  [extern]
```

Initialize a factorization cache buffer. We set the size of the rhs and boomlijst buffers immediately to their final values.

Definition at line 614 of file comtool.c.

References tree::blnce, CbUf::boomlijst, CbUf::CanCommu, CbUf::dimension, tree::left, CbUf::numdum, CbUf::NumTerms, tree::parent, CbUf::rhs, tree::right, tree::usage, and tree::value.

4.19.6.687 GCDfunction()

```
int GCDfunction (
            PHEAD WORD * term,
            WORD level )  [extern]
```

Definition at line 609 of file ratio.c.

4.19.6.688 GCDfunction3()

```
WORD * GCDfunction3 (
            PHEAD WORD * in1,
            WORD * in2 )  [extern]
```

Definition at line 1141 of file ratio.c.

4.19.6.689 ReadPolyRatFun()

```
int ReadPolyRatFun (
            PHEAD WORD * term )  [extern]
```

Definition at line 2264 of file ratio.c.

4.19.6.690 FromPolyRatFun()

```
int FromPolyRatFun (
            PHEAD WORD * fun,
            WORD ** numout,
            WORD ** denout )  [extern]
```

Definition at line 2383 of file ratio.c.

4.19.6.691 PolyDiv()

```
WORD * PolyDiv (
            PHEAD WORD * a,
            WORD * b,
            char * text )  [extern]
```

Definition at line 2741 of file ratio.c.

4.19.6.692 GCDclean()

```
void GCDclean (
            PHEAD WORD * num,
            WORD * den )  [extern]
```

Definition at line 2644 of file ratio.c.

4.19.6.693 TakeSymbolContent()

```
WORD * TakeSymbolContent (
            PHEAD WORD * in,
            WORD * term )  [extern]
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. We allow only symbols as this code is used for the polyratfun only. We have a special routine, because the generic TakeContent does too much work and speed is at a premium here. Input: in is the input expression as a sequence of terms. Output: term: the content return value: the contentfree expression. it is in new allocation, made by TermMalloc. (should be in a TermMalloc space?)

Definition at line 2434 of file ratio.c.

References GetModInverses().

Here is the call graph for this function:

4.19.6.694 GCDterms()

```
int GCDterms (
            PHEAD WORD * term1,
            WORD * term2,
            WORD * termout )  [extern]
```

Definition at line 2076 of file ratio.c.

4.19.6.695 PutExtraSymbols()

```
WORD * PutExtraSymbols (
            PHEAD WORD * in,
            WORD startebuf,
            int * actionflag )  [extern]
```

Definition at line 1244 of file ratio.c.

4.19.6.696 TakeExtraSymbols()

```
WORD * TakeExtraSymbols (
            PHEAD WORD * in,
            WORD startebuf )  [extern]
```

Definition at line 1273 of file ratio.c.

4.19.6.697 MultiplyWithTerm()

```
WORD * MultiplyWithTerm (
            PHEAD WORD * in,
            WORD * term,
            WORD par )  [extern]
```

Definition at line 1312 of file ratio.c.

4.19.6.698 TakeContent()

```
WORD * TakeContent (
            PHEAD WORD * in,
            WORD * term )  [extern]
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. Here the input is a sequence of terms in 'in' and the answer is a content-free sequence of terms. This sequence has been allocated by the Malloc1 routine in a call to EndSort, unless the expression was already content-free. In that case the input pointer is returned. The content is returned in term. This is supposed to be a separate allocation, made by TermMalloc in the calling routine.

Definition at line 1376 of file ratio.c.

References GetModInverses().

Here is the call graph for this function:

4.19.6.699 MergeSymbolLists()

```
int MergeSymbolLists (
            PHEAD WORD * old,
            WORD * extra,
            int par )  [extern]
```

Definition at line 1808 of file ratio.c.

4.19.6.700 MergeDotproductLists()

```
int MergeDotproductLists (
            PHEAD WORD * old,
            WORD * extra,
            int par )  [extern]
```

Definition at line 1927 of file ratio.c.

4.19.6.701 CreateExpression()

```
WORD * CreateExpression (
            PHEAD WORD )  [extern]
```

Definition at line 2020 of file ratio.c.

4.19.6.702 DIVfunction()

```
int DIVfunction (
            PHEAD WORD * term,
            WORD level,
            int par )  [extern]
```

Definition at line 2788 of file ratio.c.

4.19.6.703 MULfunc()

```
WORD * MULfunc (
            PHEAD WORD * p1,
            WORD * p2 )  [extern]
```

Definition at line 2941 of file ratio.c.

4.19.6.704 ConvertArgument()

```
WORD * ConvertArgument (
            PHEAD WORD * arg,
            int * type )  [extern]
```

Definition at line 3002 of file ratio.c.

4.19.6.705 ExpandRat()

```
int ExpandRat (
            PHEAD WORD * fun )  [extern]
```

Definition at line 3097 of file ratio.c.

4.19.6.706 InvPoly()

```
int InvPoly (
            PHEAD WORD * inpoly,
            WORD maxpow,
            WORD sym )  [extern]
```

Definition at line 3576 of file ratio.c.

4.19.6.707 TestDoLoop()

```
WORD TestDoLoop (
            PHEAD WORD * lhsbuf,
            WORD level )  [extern]
```

Definition at line 2668 of file dollar.c.

4.19.6.708 TestEndDoLoop()

```
WORD TestEndDoLoop (
            PHEAD WORD * lhsbuf,
            WORD level )  [extern]
```

Definition at line 2746 of file dollar.c.

4.19.6.709 DoPreAdd()

```
int DoPreAdd (
            UBYTE * s )  [extern]
```

Definition at line 1065 of file dict.c.

4.19.6.710 DoPreUseDictionary()

```
int DoPreUseDictionary (
            UBYTE * s )  [extern]
```

Definition at line 1020 of file dict.c.

4.19.6.711 DoPreCloseDictionary()

```
int DoPreCloseDictionary (
            UBYTE * s )  [extern]
```

Definition at line 996 of file dict.c.

4.19.6.712 DoPreOpenDictionary()

```
int DoPreOpenDictionary (
            UBYTE * s )  [extern]
```

Definition at line 957 of file dict.c.

4.19.6.713 RemoveDictionary()

```
void RemoveDictionary (
            DICTIONARY * dict )  [extern]
```

Definition at line 900 of file dict.c.

4.19.6.714 UnSetDictionary()

```
void UnSetDictionary (
            VOID  )  [extern]
```

Definition at line 881 of file dict.c.

4.19.6.715 SetDictionaryOptions()

```
int SetDictionaryOptions (
            UBYTE * options )  [extern]
```

Definition at line 788 of file dict.c.

4.19.6.716 AddToDictionary()

```
int AddToDictionary (
            DICTIONARY * dict,
            UBYTE * left,
            UBYTE * right )  [extern]
```

Definition at line 489 of file dict.c.

4.19.6.717 AddDictionary()

```
int AddDictionary (
            UBYTE * name )  [extern]
```

Definition at line 447 of file dict.c.

4.19.6.718 FindDictionary()

```
int FindDictionary (
            UBYTE * name )  [extern]
```

Definition at line 432 of file dict.c.

4.19.6.719 IsExponentSign()

```
int IsExponentSign (
            VOID  )  [extern]
```

Definition at line 237 of file dict.c.

4.19.6.720 IsMultiplySign()

```
int IsMultiplySign (
            VOID  )  [extern]
```

Definition at line 218 of file dict.c.

4.19.6.721 TransformRational()

```
VOID TransformRational (
            UWORD * a,
            WORD na )  [extern]
```

Definition at line 71 of file dict.c.

4.19.6.722 WriteDictionary()

```
void WriteDictionary (
            DICTIONARY * dict )  [extern]
```

Definition at line 1278 of file sch.c.

4.19.6.723 ShrinkDictionary()

```
void ShrinkDictionary (
            DICTIONARY * dict )  [extern]
```

Definition at line 943 of file dict.c.

4.19.6.724 FindSymbol()

```
UBYTE * FindSymbol (
            WORD num )  [extern]
```

Definition at line 256 of file dict.c.

4.19.6.725 FindVector()

```
UBYTE * FindVector (
            WORD num )  [extern]
```

Definition at line 277 of file dict.c.

4.19.6.726 FindIndex()

```
UBYTE * FindIndex (
            WORD num )  [extern]
```

Definition at line 299 of file dict.c.

4.19.6.727 FindFunction()

```
UBYTE * FindFunction (
            WORD num )  [extern]
```

Definition at line 321 of file dict.c.

4.19.6.728 FindFunWithArgs()

```
UBYTE * FindFunWithArgs (
            WORD * t )  [extern]
```

Definition at line 343 of file dict.c.

4.19.6.729 FindExtraSymbol()

```
UBYTE * FindExtraSymbol (
            WORD num )  [extern]
```

Definition at line 375 of file dict.c.

4.19.6.730 DictToBytes()

```
LONG DictToBytes (
            DICTIONARY * dict,
            UBYTE * buf )  [extern]
```

Definition at line 1117 of file dict.c.

4.19.6.731 DictFromBytes()

```
DICTIONARY * DictFromBytes (
            UBYTE * buf )  [extern]
```

Definition at line 1150 of file dict.c.

4.19.6.732 CoCreateSpectator()

```
int CoCreateSpectator (
            UBYTE * inp )  [extern]
```

Definition at line 89 of file spectator.c.

4.19.6.733 CoToSpectator()

```
int CoToSpectator (
            UBYTE * inp )  [extern]
```

Definition at line 181 of file spectator.c.

4.19.6.734 CoRemoveSpectator()

```
int CoRemoveSpectator (
            UBYTE * inp )  [extern]
```

Definition at line 233 of file spectator.c.

4.19.6.735 CoEmptySpectator()

```
int CoEmptySpectator (
            UBYTE * inp )  [extern]
```

Definition at line 289 of file spectator.c.

4.19.6.736 CoCopySpectator()

```
int CoCopySpectator (
            UBYTE * inp )  [extern]
```

Definition at line 455 of file spectator.c.

4.19.6.737 PutInSpectator()

```
int PutInSpectator (
            WORD * term,
            WORD specnum )  [extern]
```

Definition at line 341 of file spectator.c.

4.19.6.738 ClearSpectators()

```
void ClearSpectators (
            WORD par )  [extern]
```

Definition at line 667 of file spectator.c.

4.19.6.739 GetFromSpectator()

```
WORD GetFromSpectator (
            WORD * term,
            WORD specnum )  [extern]
```

Definition at line 593 of file spectator.c.

4.19.6.740 FlushSpectators()

```
void FlushSpectators (
            VOID  )  [extern]
```

Definition at line 398 of file spectator.c.

4.19.6.741 ReadFromScratch()

```
int ReadFromScratch (
            FILEHANDLE * fi,
            POSITION * pos,
            UBYTE * buffer,
            POSITION * length )  [extern]
```

Definition at line 273 of file store.c.

4.19.6.742 AddToScratch()

```
int AddToScratch (
            FILEHANDLE * fi,
            POSITION * pos,
            UBYTE * buffer,
            POSITION * length,
            int withflush )  [extern]
```

Definition at line 300 of file store.c.

4.19.6.743 DoPreAppendPath()

```
int DoPreAppendPath (
            UBYTE * s )  [extern]
```

Appends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #appendpath <path>

Definition at line 6875 of file pre.c.

4.19.6.744 DoPrePrependPath()

```
int DoPrePrependPath (
            UBYTE * s )  [extern]
```

Prepends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #prependpath <path>

Definition at line 6892 of file pre.c.

4.20 declare.h

Go to the documentation of this file.

```
#ifndef __FDECLARE__
#define __FDECLARE__

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
```

```
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

/*
    #[ Macro's :
*/

#define MaX(x,y) ((x) > (y) ? (x): (y))
#define MiN(x,y) ((x) < (y) ? (x): (y))
#define ABS(x) ( (x) < 0 ? -(x): (x) )
#define SGN(x) ( (x) > 0 ? 1 : (x) < 0 ? -1 : 0 )
#define REDLENG(x) ((((x)<0)?((x)+1):((x)-1))»1)
#define INCLENG(x) (((x)<0)?(((x)«1)-1):(((x)«1)+1))
#define GETCOEF(x,y) x += *x;y = x[-1];x -= ABS(y);y=REDLENG(y)
#define GETSTOP(x,y) y=x+(*x)-1;y -= ABS(*y)-1
#define StuffAdd(x,y)  (((x)<0?-1:1)*(y)+((y)<0?-1:1)*(x))

#define TOKENTOLINE(x,y) if ( AC.OutputSpaces == NOSPACEFORMAT ) { \
        TokenToLine((UBYTE *)(y)); } else { TokenToLine((UBYTE *)(x)); }

#define UngetFromStream(stream,c) ((stream)->nextchar[(stream)->isnextchar++]=c)
#ifdef WITHRETURN
#define AddLineFeed(s,n) { (s)[(n)++] = CARRIAGERETURN; (s)[(n)++] = LINEFEED; }
#else
#define AddLineFeed(s,n) { (s)[(n)++] = LINEFEED; }
#endif
#define TryRecover(x) Terminate(-1)
#define UngetChar(c) { pushbackchar = c; }
#define ParseNumber(x,s) {(x)=0;while(*(s)>='0'&&*(s)<='9')(x)=10*(x)+*(s)++ -'0';}
#define ParseSign(sgn,s) {(sgn)=0;while(*(s)=='-'||*(s)=='+'){\
          if ( *(s)++ == '-' ) sgn ^= 1;}}
#define ParseSignedNumber(x,s) { int sgn; ParseSign(sgn,s)\
          ParseNumber(x,s) if ( sgn ) x = -x; }

#define NCOPY(s,t,n) while ( --n >= 0 ) *s++ = *t++;

/*#define NCOPY(s,t,n) { memcpy(s,t,n*sizeof(WORD)); s+=n; t+=n; n = -1; }*/
#define NCOPYI(s,t,n) while ( --n >= 0 ) *s++ = *t++;
#define NCOPYB(s,t,n) while ( --n >= 0 ) *s++ = *t++;
#define NCOPYI32(s,t,n) while ( --n >= 0 ) *s++ = *t++;
#define WCOPY(s,t,n) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; while ( --nn >= 0 ) *ss++=*tt++; }
#define NeedNumber(x,s,err) { int sgn = 1;                                  \
        while ( *s == ' ' || *s == '\t' || *s == '-' || *s == '+' ) {     \
            if ( *s == '-' ) sgn = -sgn; s++; }                           \
        if ( chartype[*s] != 1 ) goto err;                                \
        ParseNumber(x,s)                                                  \
        if ( sgn < 0 ) (x) = -(x); while ( *s == ' ' || *s == '\t' ) s++;\
    }
#define SKIPBLANKS(s) { while ( *(s) == ' ' || *(s) == '\t' ) (s)++; }
#define FLUSHCONSOLE if ( AP.InOutBuf > 0 ) CharOut(LINEFEED)

#define SKIPBRA1(s) { int lev1=0; s++; while(*s) { if(*s=='[')lev1++; \
                    else if(*s==']'&&--lev1<0)break; s++;} }
#define SKIPBRA2(s) { int lev2=0; s++; while(*s) { if(*s=='{')lev2++; \
                    else if(*s=='}'&&--lev2<0)break; \
                    else if(*s=='[')SKIPBRA1(s) s++;} }
#define SKIPBRA3(s) { int lev3=0; s++; while(*s) { if(*s=='(')lev3++; \
                    else if(*s==')'&&--lev3<0)break; \
                    else if(*s=='{')SKIPBRA2(s) \
                    else if(*s=='[')SKIPBRA1(s) s++;} }
#define SKIPBRA4(s) { int lev4=0; s++; while(*s) { if(*s=='(')lev4++; \
                    else if(*s==')'&&--lev4<0)break; \
                    else if(*s=='[')SKIPBRA1(s) s++;} }
#define SKIPBRA5(s) { int lev5=0; s++; while(*s) { if(*s=='{')lev5++; \
                    else if(*s=='}'&&--lev5<0)break; \
                    else if(*s=='(')SKIPBRA4(s) \
                    else if(*s=='[')SKIPBRA1(s) s++;} }

/*
#define CYCLE1(a,i) {WORD iX,jX; iX=*a; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
*/
#define CYCLE1(t,a,i) {t iX=*a; WORD jX; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}

#define AddToCB(c,wx) if(c->Pointer>=c->Top) \
        DoubleCbuffer(c-cbuf,c->Pointer,21); \
        *(c->Pointer)++ = wx;

#define EXCHINOUT { FILEHANDLE *ffFi = AR.outfile; \
    AR.outfile = AR.infile; AR.infile = ffFi; }
#define BACKINOUT { FILEHANDLE *ffFi = AR.outfile; POSITION posi; \
    AR.outfile = AR.infile; AR.infile = ffFi; \
    SetEndScratch(AR.infile,&posi); }

#define CopyArg(to,from) { if ( *from > 0 ) { int ica = *from; NCOPY(to,from,ica) } \
```

```
        else if ( *from <= -FUNCTION ) *to++ = *from++;  \
        else { *to++ = *from++; *to++ = *from++; } }

#if ARGHEAD > 2
#define FILLARG(w) { int i = ARGHEAD-2; while ( --i >= 0 ) *w++ = 0; }
#define COPYARG(w,t) { int i = ARGHEAD-2; while ( --i >= 0 ) *w++ = *t++; }
#define ZEROARG(w) { int i; for ( i = 2; i < ARGHEAD; i++ ) w[i] = 0; }
#else
#define FILLARG(w)
#define COPYARG(w,t)
#define ZEROARG(w)
#endif

#if FUNHEAD > 2
#define FILLFUN(w) { *w++ = 0; FILLFUN3(w) }
#define COPYFUN(w,t) { *w++ = *t++; COPYFUN3(w,t) }
#else
#define FILLFUN(w)
#define COPYFUN(w,t)
#endif

#if FUNHEAD > 3
#define FILLFUN3(w) { int ie = FUNHEAD-3; while ( --ie >= 0 ) *w++ = 0; }
#define COPYFUN3(w,t) { int ie = FUNHEAD-3; while ( --ie >= 0 ) *w++ = *t++; }
#else
#define COPYFUN3(w,t)
#define FILLFUN3(w)
#endif

#if SUBEXPSIZE > 5
#define FILLSUB(w) { int ie = SUBEXPSIZE-5; while ( --ie >= 0 ) *w++ = 0; }
#define COPYSUB(w,ww) { int ie = SUBEXPSIZE-5; while ( --ie >= 0 ) *w++ = *ww++; }
#else
#define FILLSUB(w)
#define COPYSUB(w,ww)
#endif

#if EXPRHEAD > 4
#define FILLEXPR(w) { int ie = EXPRHEAD-4; while ( --ie >= 0 ) *w++ = 0; }
#else
#define FILLEXPR(w)
#endif

#define NEXTARG(x) if(*x>0) x += *x; else if(*x <= -FUNCTION)x++; else x += 2;
#define COPY1ARG(s1,t1) { int ica; if ( (ica=*t1) > 0 ) { NCOPY(s1,t1,ica) } \
        else if(*t1<=-FUNCTION){*s1++=*t1++;} else{*s1++=*t1++;*s1++=*t1++;} }

#define ZeroFillRange(w,begin,end) do { \
    int tmp_i; \
    for ( tmp_i = begin; tmp_i < end; tmp_i++ ) { (w)[tmp_i] = 0; } \
} while (0)

#define TABLESIZE(a,b) (((WORD)sizeof(a))/((WORD)sizeof(b)))
#define WORDDIF(x,y) (WORD)(x-y)
#define wsizeof(a) ((WORD)sizeof(a))
#define VARNAME(type,num) (AC.varnames->namebuffer+type[num].name)
#define DOLLARNAME(type,num) (AC.dollarnames->namebuffer+type[num].name)
#define EXPRNAME(num) (AC.exprnames->namebuffer+Expressions[num].name)

#define PREV(x) prevorder?prevorder:x

#define SETERROR(x) { Terminate(-1); return(-1); }

/* use this macro to avoid the unused parameter warning */
#define DUMMYUSE(x) (void)(x);

#ifdef _FILE_OFFSET_BITS
#if _FILE_OFFSET_BITS==64
/*:[19mar2004 mt]*/

#define ADDPOS(pp,x) (pp).p1 = ((pp).p1+(off_t)(x))
#define SETBASELENGTH(ss,x) (ss).p1 = (off_t)(x)
#define SETBASEPOSITION(pp,x) (pp).p1 = (off_t)(x)
#define ISEQUALPOSINC(pp1,pp2,x) ( (pp1).p1 == ((pp2).p1+(off_t)(x)) )
#define ISGEPOSINC(pp1,pp2,x) ( (pp1).p1 >= ((pp2).p1+(off_t)(x)) )
#define DIVPOS(pp,n) ( (pp).p1/(off_t)(n) )
#define MULPOS(pp,n) (pp).p1 *= (off_t)(n)

#else

#define ADDPOS(pp,x) (pp).p1 = ((pp).p1+(x))
#define SETBASELENGTH(ss,x) (ss).p1 = (x)
#define SETBASEPOSITION(pp,x) (pp).p1 = (x)
#define ISEQUALPOSINC(pp1,pp2,x) ( (pp1).p1 == ((pp2).p1+(LONG)(x)) )
#define ISGEPOSINC(pp1,pp2,x) ( (pp1).p1 >= ((pp2).p1+(LONG)(x)) )
#define DIVPOS(pp,n) ( (pp).p1/(n) )
#define MULPOS(pp,n) (pp).p1 *= (n)
```

```
#endif
#else

#define ADDPOS(pp,x) (pp).p1 = ((pp).p1+(LONG)(x))
#define SETBASELENGTH(ss,x) (ss).p1 = (LONG)(x)
#define SETBASEPOSITION(pp,x) (pp).p1 = (LONG)(x)
#define ISEQUALPOSINC(pp1,pp2,x) ( (pp1).p1 == ((pp2).p1+(LONG)(x)) )
#define ISGEPOSINC(pp1,pp2,x) ( (pp1).p1 >= ((pp2).p1+(LONG)(x)) )
#define DIVPOS(pp,n) ( (pp).p1/(LONG)(n) )
#define MULPOS(pp,n) (pp).p1 *= (LONG)(n)

#endif
#define DIFPOS(ss,pp1,pp2) (ss).p1 = ((pp1).p1-(pp2).p1)
#define DIFBASE(pp1,pp2) ((pp1).p1-(pp2).p1)
#define ADD2POS(pp1,pp2) (pp1).p1 += (pp2).p1
#define PUTZERO(pp) (pp).p1 = 0
#define BASEPOSITION(pp) ((pp).p1)
#define SETSTARTPOS(pp) (pp).p1 = -2
#define NOTSTARTPOS(pp) ( (pp).p1 > -2 )
#define ISMINPOS(pp) ( (pp).p1 == -1 )
#define ISEQUALPOS(pp1,pp2) ( (pp1).p1 == (pp2).p1 )
#define ISNOTEQUALPOS(pp1,pp2) ( (pp1).p1 != (pp2).p1 )
#define ISLESSPOS(pp1,pp2) ( (pp1).p1 < (pp2).p1 )
#define ISGEPOS(pp1,pp2) ( (pp1).p1 >= (pp2).p1 )
#define ISNOTZEROPOS(pp) ( (pp).p1 != 0 )
#define ISZEROPOS(pp) ( (pp).p1 == 0 )
#define ISPOSPOS(pp) ( (pp).p1 > 0 )
#define ISNEGPOS(pp) ( (pp).p1 < 0 )
extern VOID TELLFILE(int,POSITION *);

#define TOLONG(x) ((LONG)(x))

#define Add2Com(x) { WORD cod[2]; cod[0] = x; cod[1] = 2; AddNtoL(2,cod); }
#define Add3Com(x1,x2) { WORD cod[3]; cod[0] = x1; cod[1] = 3; cod[2] = x2; AddNtoL(3,cod); }
#define Add4Com(x1,x2,x3) { WORD cod[4]; cod[0] = x1; cod[1] = 4; \
   cod[2] = x2; cod[3] = x3; AddNtoL(4,cod); }
#define Add5Com(x1,x2,x3,x4) { WORD cod[5]; cod[0] = x1; cod[1] = 5; \
   cod[2] = x2; cod[3] = x3; cod[4] = x4; AddNtoL(5,cod); }

/*
    The temporary variable ppp is to avoid a compiler warning about strict aliassing
*/
#define WantAddPointers(x) while((AT.pWorkPointer+(x))>AR.pWorkSize){WORD ***ppp=&AT.pWorkSpace;\
    ExpandBuffer((void **)ppp,&AR.pWorkSize,(int)(sizeof(WORD *)));}
#define WantAddLongs(x) while((AT.lWorkPointer+(x))>AR.lWorkSize){LONG **ppp=&AT.lWorkSpace;\
    ExpandBuffer((void **)ppp,&AR.lWorkSize,sizeof(LONG));}
#define WantAddPositions(x) while((AT.posWorkPointer+(x))>AR.posWorkSize){POSITION
**ppp=&AT.posWorkSpace;\
    ExpandBuffer((void **)ppp,&AR.posWorkSize,sizeof(POSITION));}

/* inline in form3.h (or config.h). */
#define FORM_INLINE inline

/*
    Macro's for memory management. This can be done by routines, but that
    would be slower. Inline routines could do this, but we don't want to
    leave this to the friendliness of the compiler(s).
    The routines can be found in the file tools.c
*/
#define MEMORYMACROS

#ifdef MEMORYMACROS

#define TermMalloc(x) ( (AT.TermMemTop <= 0 ) ? TermMallocAddMemory(BHEAD0),
AT.TermMemHeap[--AT.TermMemTop]: AT.TermMemHeap[--AT.TermMemTop] )
#define NumberMalloc(x) ( (AT.NumberMemTop <= 0 ) ? NumberMallocAddMemory(BHEAD0),
AT.NumberMemHeap[--AT.NumberMemTop]: AT.NumberMemHeap[--AT.NumberMemTop] )
#define CacheNumberMalloc(x) ( (AT.CacheNumberMemTop <= 0 ) ? CacheNumberMallocAddMemory(BHEAD0),
AT.CacheNumberMemHeap[--AT.CacheNumberMemTop]: AT.CacheNumberMemHeap[--AT.CacheNumberMemTop] )
#define TermFree(TermMem,x) AT.TermMemHeap[AT.TermMemTop++] = (WORD *)(TermMem)
#define NumberFree(NumberMem,x) AT.NumberMemHeap[AT.NumberMemTop++] = (UWORD *)(NumberMem)
#define CacheNumberFree(NumberMem,x) AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = (UWORD
*)(NumberMem)

#else

#define TermMalloc(x) TermMalloc2(BHEAD (char *)(x))
#define NumberMalloc(x) NumberMalloc2(BHEAD (char *)(x))
#define CacheNumberMalloc(x) CacheNumberMalloc2(BHEAD (char *)(x))
#define TermFree(x,y) TermFree2(BHEAD (WORD *)(x),(char *)(y))
#define NumberFree(x,y) NumberFree2(BHEAD (UWORD *)(x),(char *)(y))
#define CacheNumberFree(x,y) CacheNumberFree2(BHEAD (UWORD *)(x),(char *)(y))

#endif

/*
 * Macros for checking nesting levels in the compiler, used as follows:
```

```
 *
 *   AC.IfSumCheck[AC.IfLevel] = NestingChecksum();
 *   AC.IfLevel++;
 *
 *   AC.IfLevel--;
 *   if ( AC.IfSumCheck[AC.IfLevel] != NestingChecksum() ) {
 *     MesNesting();
 *   }
 *
 * Note that NestingChecksum() also contains AC.IfLevel and so in this case
 * using increment/decrement operators on it in the left-hand side may be
 * confusing.
 */
#define NestingChecksum() (AC.IfLevel + AC.RepLevel + AC.arglevel + AC.insidelevel + AC.termlevel + AC.inexprlevel + AC.dolooplevel)
#define MesNesting() MesPrint("&Illegal nesting of if, repeat, argument, inside, term, inexpression and do")

#define MarkPolyRatFunDirty(T) {if(*T&&AR.PolyFunType==2){WORD *TP,*TT;TT=T+*T;TT-=ABS(TT[-1]);\
TP=T+1;while(TP<TT){if(*TP==AR.PolyFun){TP[2]|=(DIRTYFLAG|MUSTCLEANPRF);}TP+=TP[1];}}}

/*
    Macros for nesting input levels for #$name = ...; assign instructions.
    Note that the level should never go below zero.
*/
#define PUSHPREASSIGNLEVEL AP.PreAssignLevel++; { GETIDENTITY \
    if ( AP.PreAssignLevel >= AP.MaxPreAssignLevel ) { int i; \
        LONG *ap = (LONG *)Malloc1(2*AP.MaxPreAssignLevel*sizeof(LONG *),"PreAssignStack"); \
        for ( i = 0; i < AP.MaxPreAssignLevel; i++ ) ap[i] = AP.PreAssignStack[i]; \
        M_free(AP.PreAssignStack,"PreAssignStack"); \
        AP.MaxPreAssignLevel *= 2; AP.PreAssignStack = ap; \
    } \
    *AT.WorkPointer++ = AP.PreContinuation; AP.PreContinuation = 0; \
    AP.PreAssignStack[AP.PreAssignLevel] = AC.iPointer - AC.iBuffer; }

#define POPPREASSIGNLEVEL if ( AP.PreAssignLevel > 0 ) { GETIDENTITY \
    AC.iPointer = AC.iBuffer + AP.PreAssignStack[AP.PreAssignLevel--]; \
    AP.PreContinuation = *--AT.WorkPointer; \
    *AC.iPointer = 0; }

/*
    MesPrint("P-level popped to %d with %d",AP.PreAssignLevel,(WORD)(AC.iPointer - AC.iBuffer));

    #] Macro's :
    #[ Thread objects :
*/

#ifdef WITHPTHREADS

#define EXTERNLOCK(x) extern pthread_mutex_t x;
#define INILOCK(x)    pthread_mutex_t x = PTHREAD_MUTEX_INITIALIZER
#define EXTERNRWLOCK(x) extern pthread_rwlock_t x;
#define INIRWLOCK(x)    pthread_rwlock_t x = PTHREAD_RWLOCK_INITIALIZER;
#ifdef DEBUGGINGLOCKS
#include <asm/errno.h>
#define LOCK(x)       while ( pthread_mutex_trylock(&(x)) == EBUSY ) {}
#define RWLOCKR(x)      while ( pthread_rwlock_tryrdlock(&(x)) == EBUSY ) {}
#define RWLOCKW(x)      while ( pthread_rwlock_trywrlock(&(x)) == EBUSY ) {}
#else
#define LOCK(x)       pthread_mutex_lock(&(x))
#define RWLOCKR(x)      pthread_rwlock_rdlock(&(x))
#define RWLOCKW(x)      pthread_rwlock_wrlock(&(x))
#endif
#define UNLOCK(x)     pthread_mutex_unlock(&(x))
#define UNRWLOCK(x)     pthread_rwlock_unlock(&(x))
#define MLOCK(x)      LOCK(x)
#define MUNLOCK(x)    UNLOCK(x)

#define GETBIDENTITY
#define GETIDENTITY   int identity = WhoAmI(); ALLPRIVATES *B = AB[identity];
#else

#define EXTERNLOCK(x)
#define INILOCK(x)
#define LOCK(x)
#define UNLOCK(x)
#define EXTERNRWLOCK(x)
#define INIRWLOCK(x)
#define RWLOCKR(x)
#define RWLOCKW(x)
#define UNRWLOCK(x)
#ifdef WITHMPI
#define MLOCK(x)      do { if ( PF.me != MASTER ) PF_MLock(); } while (0)
#define MUNLOCK(x)    do { if ( PF.me != MASTER ) PF_MUnlock(); } while (0)
#else
#define MLOCK(x)
#define MUNLOCK(x)
```

```
#endif
#define GETIDENTITY
#define GETBIDENTITY

#endif

/*
    #] Thread objects :
    #[ Declarations :
*/

#ifdef TERMMALLOCDEBUG
extern WORD **DebugHeap1, **DebugHeap2;
#endif

extern VOID   StartVariables();
extern VOID   setSignalHandlers(VOID);
extern UBYTE *CodeToLine(WORD,UBYTE *);
extern UBYTE *AddArrayIndex(WORD ,UBYTE *);
extern INDEXENTRY *FindInIndex(WORD,FILEDATA *,WORD,WORD);
extern INDEXENTRY *NextFileIndex(POSITION *);
extern WORD  *PasteTerm(PHEAD WORD,WORD *,WORD *,WORD,WORD);
extern UBYTE *StrCopy(UBYTE *,UBYTE *);
extern UBYTE *WrtPower(UBYTE *,WORD);
extern WORD   AccumGCD(PHEAD UWORD *,WORD *,UWORD *,WORD);
extern VOID   AddArgs(PHEAD WORD *,WORD *,WORD *);
extern WORD   AddCoef(PHEAD WORD **,WORD **);
extern WORD   AddLong(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern WORD   AddPLon(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern WORD   AddPoly(PHEAD WORD **,WORD **);
extern WORD   AddRat(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern VOID   AddToLine(UBYTE *);
extern WORD   AddWild(PHEAD WORD,WORD,WORD);
extern WORD   BigLong(UWORD *,WORD,UWORD *,WORD);
extern WORD   BinomGen(PHEAD WORD *,WORD,WORD **,WORD,WORD,WORD,WORD,WORD,UWORD *,WORD);
extern WORD   CheckWild(PHEAD WORD,WORD,WORD,WORD *);
extern WORD   Chisholm(PHEAD WORD *,WORD);
extern WORD   CleanExpr(WORD);
extern VOID   CleanUp(WORD);
extern VOID   ClearWild(PHEAD0);
extern WORD   CompareFunctions(WORD *,WORD *);
extern WORD   Commute(WORD *,WORD *);
extern WORD   DetCommu(WORD *);
extern WORD   DoesCommu(WORD *);
extern int    CompArg(WORD *,WORD *);
extern WORD   CompCoef(WORD *, WORD *);
extern WORD   CompGroup(PHEAD WORD,WORD **,WORD *,WORD *,WORD);
extern WORD   Compare1(PHEAD WORD *,WORD *,WORD);
extern WORD   CountDo(WORD *,WORD *);
extern WORD   CountFun(WORD *,WORD *);
extern WORD   DimensionSubterm(WORD *);
extern WORD   DimensionTerm(WORD *);
extern WORD   DimensionExpression(PHEAD WORD *);
extern WORD   Deferred(PHEAD WORD *,WORD);
extern WORD   DeleteStore(WORD);
extern WORD   DetCurDum(PHEAD WORD *);
extern VOID   DetVars(WORD *,WORD);
extern WORD   Distribute(DISTRIBUTE *,WORD);
extern WORD   DivLong(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *,UWORD *,WORD *);
extern WORD   DivRat(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern WORD   Divvy(PHEAD UWORD *,WORD *,UWORD *,WORD);
extern WORD   DoDelta(WORD *);
extern WORD   DoDelta3(PHEAD WORD *,WORD);
extern WORD   TestPartitions(PHEAD WORD *, PARTI *);
extern WORD   DoPartitions(PHEAD WORD *,WORD);
extern WORD   DoTableExpansion(WORD *,WORD);
extern WORD   DoDistrib(PHEAD WORD *,WORD);
extern WORD   DoShuffle(PHEAD WORD *,WORD,WORD,WORD);
extern WORD   DoPermutations(PHEAD WORD *,WORD);
extern int    Shuffle(PHEAD WORD *, WORD *, WORD *);
extern int    FinishShuffle(PHEAD WORD *);
extern WORD   DoStuffle(PHEAD WORD *,WORD,WORD,WORD);
extern int    Stuffle(PHEAD WORD *, WORD *, WORD *);
extern int    FinishStuffle(PHEAD WORD *);
extern WORD  *StuffRootAdd(WORD *, WORD *, WORD *);
extern WORD   TestUse(WORD *,WORD);
extern DBASE *FindTB(UBYTE *);
extern int    CheckTableDeclarations(DBASE *);
extern WORD   Apply(WORD *,WORD);
extern int    ApplyExec(WORD *,int,WORD);
extern WORD   ApplyReset(WORD);
extern WORD   TableReset(VOID);
extern VOID   ReWorkT(WORD *,WORD *,WORD);
extern WORD   GetIfDollarNum(WORD *, WORD *);
extern int    FindVar(WORD *,WORD *);
extern WORD   DoIfStatement(PHEAD WORD *,WORD *);
extern WORD   DoOnePow(PHEAD WORD *,WORD,WORD,WORD *,WORD *,WORD,WORD *);
```

```
extern void   DoRevert(WORD *,WORD *);
extern WORD   DoSumF1(PHEAD WORD *,WORD *,WORD,WORD);
extern WORD   DoSumF2(PHEAD WORD *,WORD *,WORD,WORD);
extern WORD   DoTheta(PHEAD WORD *);
extern LONG   EndSort(PHEAD WORD *,int);
extern WORD   EntVar(WORD,UBYTE *,WORD,WORD,WORD,WORD);
extern WORD   EpfCon(PHEAD WORD *,WORD *,WORD,WORD);
extern WORD   EpfFind(PHEAD WORD *,WORD *);
extern WORD   EpfGen(WORD,WORD *,WORD *,WORD *,WORD);
extern WORD   EqualArg(WORD *,WORD,WORD);
extern WORD   Evaluate(UBYTE **);
extern int    Factorial(PHEAD WORD,UWORD *,WORD *);
extern int    Bernoulli(WORD,UWORD *,WORD *);
extern int    FactorIn(PHEAD WORD *,WORD);
extern int    FactorInExpr(PHEAD WORD *,WORD);
extern WORD   FindAll(PHEAD WORD *,WORD *,WORD,WORD *);
extern WORD   FindMulti(PHEAD WORD *,WORD *);
extern WORD   FindOnce(PHEAD WORD *,WORD *);
extern WORD   FindOnly(PHEAD WORD *,WORD *);
extern WORD   FindRest(PHEAD WORD *,WORD *);
extern WORD   FindSpecial(WORD *);
extern WORD   FindrNumber(WORD,VARRENUM *);
extern VOID   FiniLine(VOID);
extern WORD   FiniTerm(PHEAD WORD *,WORD *,WORD *,WORD,WORD);
extern WORD   FlushOut(POSITION *,FILEHANDLE *,int);
extern VOID   FunLevel(PHEAD WORD *);
extern VOID   AdjustRenumScratch(PHEAD0);
extern VOID   GarbHand(VOID);
extern WORD   GcdLong(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern WORD   LcmLong(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern VOID   GCD(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern ULONG  GCD2(ULONG,ULONG);
extern WORD   Generator(PHEAD WORD *,WORD);
extern WORD   GetBinom(UWORD *,WORD *,WORD,WORD);
extern WORD   GetFromStore(WORD *,POSITION *,RENUMBER,WORD *,WORD);
extern WORD   GetLong(UBYTE *,UWORD *,WORD *);
extern WORD   GetMoreTerms(WORD *);
extern WORD   GetMoreFromMem(WORD *,WORD **);
extern WORD   GetOneTerm(PHEAD WORD *,FILEHANDLE *,POSITION *,int);
extern RENUMBER GetTable(WORD,POSITION *,WORD);
extern WORD   GetTerm(PHEAD WORD *);
extern WORD   Glue(PHEAD WORD *,WORD *,WORD *,WORD);
extern WORD   InFunction(PHEAD WORD *,WORD *);
extern VOID   IniLine(WORD);
extern WORD   IniVars(VOID);
extern VOID   Initialize(VOID);
extern WORD   InsertTerm(PHEAD WORD *,WORD,WORD,WORD *,WORD *,WORD);
extern VOID   LongToLine(UWORD *,WORD);
extern WORD   MakeDirty(WORD *,WORD *,WORD);
extern VOID   MarkDirty(WORD *,WORD);
extern VOID   PolyFunDirty(PHEAD WORD *);
extern VOID   PolyFunClean(PHEAD WORD *);
extern WORD   MakeModTable(VOID);
extern WORD   MatchE(PHEAD WORD *,WORD *,WORD *,WORD);
extern int    MatchCy(PHEAD WORD *,WORD *,WORD *,WORD);
extern int    FunMatchCy(PHEAD WORD *,WORD *,WORD *,WORD);
extern int    FunMatchSy(PHEAD WORD *,WORD *,WORD *,WORD);
extern int    MatchArgument(PHEAD WORD *,WORD *);
extern WORD   MatchFunction(PHEAD WORD *,WORD *,WORD *);
extern WORD   MergePatches(WORD);
extern WORD   MesCerr(char *, UBYTE *);
extern WORD   MesComp(char *, UBYTE *, UBYTE *);
extern WORD   Modulus(WORD *);
extern VOID   MoveDummies(PHEAD WORD *,WORD);
extern WORD   MulLong(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern WORD   MulRat(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern WORD   Mully(PHEAD UWORD *,WORD *,UWORD *,WORD);
extern WORD   MultDo(PHEAD WORD *,WORD *);
extern WORD   NewSort(PHEAD0);
extern WORD   ExtraSymbol(WORD,WORD,WORD,WORD *,WORD *);
extern WORD   Normalize(PHEAD WORD *);
extern WORD   BracketNormalize(PHEAD WORD *);
extern VOID   DropCoefficient(PHEAD WORD *);
extern VOID   DropSymbols(PHEAD WORD *);
extern int    PutInside(PHEAD WORD *, WORD *);
extern WORD   OpenTemp(VOID);
extern VOID   Pack(UWORD *,WORD *,UWORD *,WORD );
extern LONG   PasteFile(PHEAD WORD,WORD *,POSITION *,WORD **,RENUMBER,WORD *,WORD);
extern WORD   Permute(PERM *,WORD);
extern WORD   PermuteP(PERMP *,WORD);
extern WORD   PolyFunMul(PHEAD WORD *);
extern WORD   PopVariables(VOID);
extern WORD   PrepPoly(PHEAD WORD *,WORD);
extern WORD   Processor(VOID);
extern WORD   Product(UWORD *,WORD *,WORD);
extern VOID   PrtLong(UWORD *,WORD,UBYTE *);
extern VOID   PrtTerms(VOID);
```

```
extern VOID   PrintRunningTime(VOID);
extern LONG   GetRunningTime(VOID);
extern WORD   PutBracket(PHEAD WORD *);
extern LONG   PutIn(FILEHANDLE *,POSITION *,WORD *,WORD **,int);
extern WORD   PutInStore(INDEXENTRY *,WORD);
extern WORD   PutOut(PHEAD WORD *,POSITION *,FILEHANDLE *,WORD);
extern UWORD  Quotient(UWORD *,WORD *,WORD);
extern WORD   RaisPow(PHEAD UWORD *,WORD *,UWORD);
extern VOID   RaisPowCached (PHEAD WORD, WORD, UWORD **, WORD *);
extern WORD   RaisPowMod (WORD, WORD, WORD);
extern int    NormalModulus(UWORD *,WORD *);
extern int    MakeInverses(VOID);
extern int    GetModInverses(WORD,WORD,WORD *,WORD *);
extern int    GetLongModInverses(PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *, UWORD *, WORD
    *);
extern VOID   RatToLine(UWORD *,WORD);
extern WORD   RatioFind(PHEAD WORD *,WORD *);
extern WORD   RatioGen(PHEAD WORD *,WORD *,WORD,WORD);
extern WORD   ReNumber(PHEAD WORD *);
extern WORD   ReadSnum(UBYTE **);
extern WORD   Remain10(UWORD *,WORD *);
extern WORD   Remain4(UWORD *,WORD *);
extern WORD   ResetScratch(VOID);
extern WORD   ResolveSet(PHEAD WORD *,WORD *,WORD *);
extern WORD   RevertScratch(VOID);
extern WORD   ScanFunctions(PHEAD WORD *,WORD *,WORD);
extern VOID   SeekScratch(FILEHANDLE *,POSITION *);
extern VOID   SetEndScratch(FILEHANDLE *,POSITION *);
extern VOID   SetEndHScratch(FILEHANDLE *,POSITION *);
extern WORD   SetFileIndex(VOID);
extern WORD   Sflush(FILEHANDLE *);
extern WORD   Simplify(PHEAD UWORD *,WORD *,UWORD *,WORD *);
extern WORD   SortWild(WORD *,WORD);
extern FILE  *LocateBase(char **,char **);
#ifdef NEWSPLITMERGE
extern LONG   SplitMerge(PHEAD WORD **,LONG);
#else
extern VOID   SplitMerge(PHEAD WORD **,LONG);
#endif
extern WORD   StoreTerm(PHEAD WORD *);
extern VOID   SubPLon(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
extern VOID   Substitute(PHEAD WORD *,WORD *,WORD);
extern WORD   SymFind(PHEAD WORD *,WORD *);
extern WORD   SymGen(PHEAD WORD *,WORD *,WORD,WORD);
extern WORD   Symmetrize(PHEAD WORD *,WORD *,WORD,WORD,WORD);
extern int    FullSymmetrize(PHEAD WORD *,int);
extern WORD   TakeModulus(UWORD *,WORD *,UWORD *,WORD,WORD);
extern WORD   TakeNormalModulus(UWORD *,WORD *,UWORD *,WORD,WORD);
extern VOID   TalToLine(UWORD);
extern WORD   TenVec(PHEAD WORD *,WORD *,WORD,WORD);
extern WORD   TenVecFind(PHEAD WORD *,WORD *);
extern WORD   TermRenumber(WORD *,RENUMBER,WORD);
extern VOID   TestDrop(VOID);
extern VOID   PutInVflags(WORD);
extern WORD   TestMatch(PHEAD WORD *,WORD *);
extern WORD   TestSub(PHEAD WORD *,WORD);
extern LONG   TimeCPU(WORD);
extern LONG   TimeChildren(WORD);
extern LONG   TimeWallClock(WORD);
extern LONG   Timer(int);
extern int    GetTimerInfo(LONG **,LONG **);
extern void   WriteTimerInfo(LONG *,LONG *);
extern LONG   GetWorkerTimes(VOID);
extern WORD   ToStorage(EXPRESSIONS,POSITION *);
extern VOID   TokenToLine(UBYTE *);
extern WORD   Trace4(PHEAD WORD *,WORD *,WORD,WORD);
extern WORD   Trace4Gen(PHEAD TRACES *,WORD);
extern WORD   Trace4no(WORD,WORD *,TRACES *);
extern WORD   TraceFind(PHEAD WORD *,WORD *);
extern WORD   TraceN(PHEAD WORD *,WORD *,WORD,WORD);
extern WORD   TraceNgen(PHEAD TRACES *,WORD);
extern WORD   TraceNno(WORD,WORD *,TRACES *);
extern WORD   Traces(PHEAD WORD *,WORD *,WORD,WORD);
extern WORD   Trick(WORD *,TRACES *);
extern WORD   TryDo(PHEAD WORD *,WORD *,WORD);
extern VOID   UnPack(UWORD *,WORD,WORD *,WORD *);
extern WORD   VarStore(UBYTE *,WORD,WORD,WORD);
extern WORD   WildFill(PHEAD WORD *,WORD *,WORD *);
extern WORD   WriteAll(VOID);
extern WORD   WriteOne(UBYTE *,int,int);
extern VOID   WriteArgument(WORD *);
extern WORD   WriteExpression(WORD *,LONG);
extern WORD   WriteInnerTerm(WORD *,WORD);
extern VOID   WriteLists(VOID);
extern VOID   WriteSetup(VOID);
extern VOID   WriteStats(POSITION *,WORD);
extern WORD   WriteSubTerm(WORD *,WORD);
```

```
extern WORD   WriteTerm(WORD *,WORD *,WORD,WORD,WORD);
extern WORD   execarg(PHEAD WORD *,WORD);
extern WORD   execterm(PHEAD WORD *,WORD);
extern VOID   SpecialCleanup(PHEAD0);
extern void   SetMods();
extern void   UnSetMods();

/*-----------------------------------------------------------------------*/

extern WORD   DoExecute(WORD,WORD);
extern VOID   SetScratch(FILEHANDLE *,POSITION *);
extern VOID   Warning(char *);
extern VOID   HighWarning(char *);
extern int    SpareTable(TABLES);

extern UBYTE *strDup1(UBYTE *,char *);
extern VOID  *Malloc(LONG);
extern VOID  *Malloc1(LONG,const char *);
extern int    DoTail(int,UBYTE **);
extern int    OpenInput(VOID);
extern int    PutPreVar(UBYTE *,UBYTE *,UBYTE *,int);
extern VOID   Error0(char *);
extern VOID   Error1(char *,UBYTE *);
extern VOID   Error2(char *,char *,UBYTE *);
extern UBYTE  ReadFromStream(STREAM *);
extern UBYTE  GetFromStream(STREAM *);
extern UBYTE  LookInStream(STREAM *);
extern STREAM *OpenStream(UBYTE *,int,int,int);
extern int    LocateFile(UBYTE **,int);
extern STREAM *CloseStream(STREAM *);
extern VOID   PositionStream(STREAM *,LONG);
extern int    ReverseStatements(STREAM *);
extern int    ProcessOption(UBYTE *,UBYTE *,int);
extern int    DoSetups(VOID);
extern VOID   Terminate(int);
extern NAMENODE *GetNode(NAMETREE *,UBYTE *);
extern int    AddName(NAMETREE *,UBYTE *,WORD,WORD,int *);
extern int    GetName(NAMETREE *,UBYTE *,WORD *,int);
extern int    GetLastExprName(UBYTE *,WORD *);
extern int    GetAutoName(UBYTE *,WORD *);
extern int    GetVar(UBYTE *,WORD *,WORD *,int,int);
extern int    MakeDubious(NAMETREE *,UBYTE *,WORD *);
extern int    GetOName(NAMETREE *,UBYTE *,WORD *,int);
extern VOID   DumpTree(NAMETREE *);
extern VOID   DumpNode(NAMETREE *,WORD,WORD);
extern VOID   LinkTree(NAMETREE *,WORD,WORD);
extern VOID   CopyTree(NAMETREE *,NAMETREE *,WORD,WORD);
extern int    CompactifyTree(NAMETREE *,WORD);
extern NAMETREE *MakeNameTree(VOID);
extern VOID   FreeNameTree(NAMETREE *);
extern int    AddExpression(UBYTE *,int,int);
extern int    AddSymbol(UBYTE *,int,int,int,int);
extern int    AddDollar(UBYTE *,WORD,WORD *,LONG);
extern int    ReplaceDollar(WORD,WORD,WORD *,LONG);
extern int    DollarRaiseLow(UBYTE *,LONG);
extern int    AddVector(UBYTE *,int,int);
extern int    AddDubious(UBYTE *);
extern int    AddIndex(UBYTE *,int,int);
extern UBYTE *DoDimension(UBYTE *,int *,int *);
extern int    AddFunction(UBYTE *,int,int,int,int,int,int,int);
extern int    CoCommuteInSet(UBYTE *);
extern int    CoFunction(UBYTE *,int,int);
extern int    TestName(UBYTE *);
extern int    AddSet(UBYTE *,WORD);
extern int    DoElements(UBYTE *,SETS,UBYTE *);
extern int    DoTempSet(UBYTE *,UBYTE *);
extern int    NameConflict(int,UBYTE *);
extern int    OpenFile(char *);
extern int    OpenAddFile(char *);
extern int    ReOpenFile(char *);
extern int    CreateFile(char *);
extern int    CreateLogFile(char *);
extern VOID   CloseFile(int);
extern int    CopyFile(char *, char *);
extern int    CreateHandle(VOID);
extern LONG   ReadFile(int,UBYTE *,LONG);
extern LONG   ReadPosFile(PHEAD FILEHANDLE *,UBYTE *,LONG,POSITION *);
extern LONG   WriteFileToFile(int,UBYTE *,LONG);
extern VOID   SeekFile(int,POSITION *,int);
extern LONG   TellFile(int);
extern void   FlushFile(int);
extern int    GetPosFile(int,fpos_t *);
extern int    SetPosFile(int,fpos_t *);
extern VOID   SynchFile(int);
extern VOID   TruncateFile(int);
extern int    GetChannel(char *);
extern int    GetAppendChannel(char *);
```

```
extern int    CloseChannel(char *);
extern VOID   inictable(VOID);
extern KEYWORD *findcommand(UBYTE *);
extern int    inicbufs(VOID);
extern VOID   StartFiles(VOID);
extern UBYTE *MakeDate(VOID);
extern VOID   PreProcessor(VOID);
extern VOID  *FromList(LIST *);
extern VOID  *From0List(LIST *);
extern VOID  *FromVarList(LIST *);
extern int    DoubleList(VOID ***,int *,int,char *);
extern int    DoubleLList(VOID ***,LONG *,int,char *);
extern void   DoubleBuffer(void **,void **,int,char *);
extern void   ExpandBuffer(void **,LONG *,int);
extern LONG   iexp(LONG,int);
extern int    IsLikeVector(WORD *);
extern int    AreArgsEqual(WORD *,WORD *);
extern int    CompareArgs(WORD *,WORD *);
extern UBYTE *SkipField(UBYTE *,int);
extern int    StrCmp(UBYTE *,UBYTE *);
extern int    StrICmp(UBYTE *,UBYTE *);
extern int    StrHICmp(UBYTE *,UBYTE *);
extern int    StrICont(UBYTE *,UBYTE *);
extern int    ConWord(UBYTE *,UBYTE *);
extern int    StrLen(UBYTE *);
extern UBYTE *GetPreVar(UBYTE *,int);
extern void   ToGeneral(WORD *,WORD *,WORD);
extern WORD   ToPolyFunGeneral(PHEAD WORD *);
extern int    ToFast(WORD *,WORD *);
extern SETUPPARAMETERS *GetSetupPar(UBYTE *);
extern int    RecalcSetups(VOID);
extern int    AllocSetups(VOID);
extern SORTING *AllocSort(LONG,LONG,LONG,LONG,int,int,LONG);
extern VOID   AllocSortFileName(SORTING *);
extern UBYTE *LoadInputFile(UBYTE *,int);
extern UBYTE  GetInput(VOID);
extern VOID   ClearPushback(VOID);
extern UBYTE  GetChar(int);
extern VOID   CharOut(UBYTE);
extern VOID   UnsetAllowDelay(VOID);
extern VOID   PopPreVars(int);
extern VOID   IniModule(int);
extern VOID   IniSpecialModule(int);
extern int    ModuleInstruction(int *,int *);
extern int    PreProInstruction(VOID);
extern int    LoadInstruction(int);
extern int    LoadStatement(int);
extern KEYWORD *FindKeyWord(UBYTE *,KEYWORD *,int);
extern KEYWORD *FindInKeyWord(UBYTE *,KEYWORD *,int);
extern int    DoDefine(UBYTE *);
extern int    DoRedefine(UBYTE *);
extern int    TheDefine(UBYTE *,int);
extern int    TheUndefine(UBYTE *);
extern int    ClearMacro(UBYTE *);
extern int    DoUndefine(UBYTE *);
extern int    DoInclude(UBYTE *);
extern int    DoReverseInclude(UBYTE *);
extern int    Include(UBYTE *,int);
/*[14apr2004 mt]:*/
extern int    DoExternal(UBYTE *);
extern int    DoToExternal(UBYTE *);
extern int    DoFromExternal(UBYTE *);
extern int    DoPrompt(UBYTE *);
extern int    DoSetExternal(UBYTE *);
/*[10may2006 mt]:*/
extern int    DoSetExternalAttr(UBYTE *);
/*:[10may2006 mt]*/
extern int    DoRmExternal(UBYTE *);
/*:[14apr2004 mt]*/
extern int    DoFactDollar(UBYTE *);
extern WORD   GetDollarNumber(UBYTE **,DOLLARS);
extern int    DoSetRandom(UBYTE *);
extern int    DoOptimize(UBYTE *);
extern int    DoClearOptimize(UBYTE *);
extern int    DoSkipExtraSymbols(UBYTE *);
extern int    DoMessage(UBYTE *);
extern int    DoPreOut(UBYTE *);
extern int    DoPreAppend(UBYTE *);
extern int    DoPreCreate(UBYTE *);
extern int    DoPreAssign(UBYTE *);
extern int    DoPreBreak(UBYTE *);
extern int    DoPreDefault(UBYTE *);
extern int    DoPreSwitch(UBYTE *);
extern int    DoPreEndSwitch(UBYTE *);
extern int    DoPreCase(UBYTE *);
extern int    DoPreShow(UBYTE *);
extern int    DoPreExchange(UBYTE *);
```

```
extern int    DoSystem(UBYTE *);
extern int    DoPipe(UBYTE *);
extern VOID   StartPrepro(VOID);
extern int    DoIfdef(UBYTE *,int);
extern int    DoElse(UBYTE *);
extern int    DoElseif(UBYTE *);
extern int    DoEndif(UBYTE *);
extern int    DoTerminate(UBYTE *);
extern int    DoIf(UBYTE *);
extern int    DoCall(UBYTE *);
extern int    DoDebug(UBYTE *);
extern int    DoDo(UBYTE *);
extern int    DoBreakDo(UBYTE *);
extern int    DoEnddo(UBYTE *);
extern int    DoEndprocedure(UBYTE *);
extern int    DoInside(UBYTE *);
extern int    DoEndInside(UBYTE *);
extern int    DoProcedure(UBYTE *);
extern int    DoPrePrintTimes(UBYTE *);
extern int    DoPreWrite(UBYTE *);
extern int    DoPreClose(UBYTE *);
extern int    DoPreRemove(UBYTE *);
extern int    DoCommentChar(UBYTE *);
extern int    DoPrcExtension(UBYTE *);
extern int    DoPreReset(UBYTE *);
extern VOID   WriteString(int,UBYTE *,int);
extern VOID   WriteUnfinString(int,UBYTE *,int);
extern UBYTE *AddToString(UBYTE *,UBYTE *,int);
extern UBYTE *PreCalc(VOID);
extern UBYTE *PreEval(UBYTE *,LONG *);
extern VOID   NumToStr(UBYTE *,LONG);
extern int    PreCmp(int,int,UBYTE *,int,int,UBYTE *,int);
extern int    PreEq(int,int,UBYTE *,int,int,UBYTE *,int);
extern UBYTE *pParseObject(UBYTE *,int *,LONG *);
extern UBYTE *PreIfEval(UBYTE *,int *);
extern int    EvalPreIf(UBYTE *);
extern int    PreLoad(PRELOAD *,UBYTE *,UBYTE *,int,char *);
extern int    PreSkip(UBYTE *,UBYTE *,int);
extern UBYTE *EndOfToken(UBYTE *);
extern VOID   SetSpecialMode(int,int);
extern VOID   MakeGlobal(VOID);
extern int    ExecModule(int);
extern int    ExecStore(VOID);
extern VOID   FullCleanUp(VOID);
extern int    DoExecStatement(VOID);
extern int    DoPipeStatement(VOID);
extern int    DoPolyfun(UBYTE *);
extern int    DoPolyratfun(UBYTE *);
extern int    CompileStatement(UBYTE *);
extern UBYTE *ToToken(UBYTE *);
extern int    GetDollar(UBYTE *);
extern int    MesWork(VOID);
extern int    MesPrint(const char *,...);
extern int    MesCall(char *);
extern UBYTE *NumCopy(WORD,UBYTE *);
extern char  *LongCopy(LONG,char *);
extern char  *LongLongCopy(off_t *,char *);
extern VOID   ReserveTempFiles(int);
extern VOID   PrintTerm(WORD *,char *);
extern VOID   PrintTermC(WORD *,char *);
extern VOID   PrintSubTerm(WORD *,char *);
extern VOID   PrintWords(WORD *,LONG);
extern void   PrintSeq(WORD *,char *);
extern int    ExpandTripleDots(int);
extern LONG   ComPress(WORD **,LONG *);
extern VOID   StageSort(FILEHANDLE *);

#define M_alloc(x)      malloc((size_t)(x))

extern void   M_free(VOID *,const char *);
extern void   ClearWildcardNames(VOID);
extern int    AddWildcardName(UBYTE *);
extern int    GetWildcardName(UBYTE *);
extern void   Globalize(int);
extern void   ResetVariables(int);
extern void   AddToPreTypes(int);
extern void   MessPreNesting(int);
extern LONG   GetStreamPosition(STREAM *);
extern WORD  *DoubleCbuffer(int,WORD *,int);
extern WORD  *AddLHS(int);
extern WORD  *AddRHS(int,int);
extern int    AddNtoL(int,WORD *);
extern int    AddNtoC(int,int,WORD *,int);
extern VOID   DoubleIfBuffers(VOID);
extern STREAM *CreateStream(UBYTE *);

extern int    setonoff(UBYTE *,int *,int,int);
```

```
extern int    DoPrint(UBYTE *,int);
extern int    SetExpr(UBYTE *,int,int);
extern void   AddToCom(int,WORD *);
extern int    Add2ComStrings(int,WORD *,UBYTE *,UBYTE *);
extern int    DoSymmetrize(UBYTE *,int);
extern int    DoArgument(UBYTE *,int);
extern int    ArgFactorize(PHEAD WORD *,WORD *);
extern WORD  *TakeArgContent(PHEAD WORD *, WORD *);
extern WORD  *MakeInteger(PHEAD WORD *,WORD *,WORD *);
extern WORD  *MakeMod(PHEAD WORD *,WORD *,WORD *);
extern WORD   FindArg(PHEAD WORD *);
extern WORD   InsertArg(PHEAD WORD *,WORD *,int);
extern int    CleanupArgCache(PHEAD WORD);
extern int    ArgSymbolMerge(WORD *, WORD *);
extern int    ArgDotproductMerge(WORD *, WORD *);
extern void   SortWeights(LONG *,LONG *,WORD);
extern int    DoBrackets(UBYTE *,int);
extern int    DoPutInside(UBYTE *,int);
extern WORD  *CountComp(UBYTE *,WORD *);
extern int    CoAntiBracket(UBYTE *);
extern int    CoAntiSymmetrize(UBYTE *);
extern int    DoArgPlode(UBYTE *,int);
extern int    CoArgExplode(UBYTE *);
extern int    CoArgImplode(UBYTE *);
extern int    CoArgument(UBYTE *);
extern int    CoInside(UBYTE *);
extern int    ExecInside(UBYTE *);
extern int    CoInExpression(UBYTE *);
extern int    CoInParallel(UBYTE *);
extern int    CoNotInParallel(UBYTE *);
extern int    DoInParallel(UBYTE *,int);
extern int    CoEndInExpression(UBYTE *);
extern int    CoBracket(UBYTE *);
extern int    CoPutInside(UBYTE *);
extern int    CoAntiPutInside(UBYTE *);
extern int    CoMultiBracket(UBYTE *);
extern int    CoCFunction(UBYTE *);
extern int    CoCTensor(UBYTE *);
extern int    CoCollect(UBYTE *);
extern int    CoCompress(UBYTE *);
extern int    CoContract(UBYTE *);
extern int    CoCycleSymmetrize(UBYTE *);
extern int    CoDelete(UBYTE *);
extern int    CoTableBase(UBYTE *);
extern int    CoApply(UBYTE *);
extern int    CoDenominators(UBYTE *);
extern int    CoDimension(UBYTE *);
extern int    CoDiscard(UBYTE *);
extern int    CoDisorder(UBYTE *);
extern int    CoDrop(UBYTE *);
extern int    CoDropCoefficient(UBYTE *);
extern int    CoDropSymbols(UBYTE *);
extern int    CoElse(UBYTE *);
extern int    CoElseIf(UBYTE *);
extern int    CoEndArgument(UBYTE *);
extern int    CoEndInside(UBYTE *);
extern int    CoEndIf(UBYTE *);
extern int    CoEndRepeat(UBYTE *);
extern int    CoEndTerm(UBYTE *);
extern int    CoEndWhile(UBYTE *);
extern int    CoExit(UBYTE *);
extern int    CoFactArg(UBYTE *);
extern int    CoFactDollar(UBYTE *);
extern int    CoFactorize(UBYTE *);
extern int    CoNFactorize(UBYTE *);
extern int    CoUnFactorize(UBYTE *);
extern int    CoNUnFactorize(UBYTE *);
extern int    DoFactorize(UBYTE *,int);
extern int    CoFill(UBYTE *);
extern int    CoFillExpression(UBYTE *);
extern int    CoFixIndex(UBYTE *);
extern int    CoFormat(UBYTE *);
extern int    CoGlobal(UBYTE *);
extern int    CoGlobalFactorized(UBYTE *);
extern int    CoGoTo(UBYTE *);
extern int    CoId(UBYTE *);
extern int    CoIdNew(UBYTE *);
extern int    CoIdOld(UBYTE *);
extern int    CoIf(UBYTE *);
extern int    CoIfMatch(UBYTE *);
extern int    CoIfNoMatch(UBYTE *);
extern int    CoIndex(UBYTE *);
extern int    CoInsideFirst(UBYTE *);
extern int    CoKeep(UBYTE *);
extern int    CoLabel(UBYTE *);
extern int    CoLoad(UBYTE *);
extern int    CoLocal(UBYTE *);
```

```
extern int    CoLocalFactorized(UBYTE *);
extern int    CoMany(UBYTE *);
extern int    CoMerge(UBYTE *);
extern int    CoStuffle(UBYTE *);
extern int    CoMetric(UBYTE *);
extern int    CoModOption(UBYTE *);
extern int    CoModuleOption(UBYTE *);
extern int    CoModulus(UBYTE *);
extern int    CoMulti(UBYTE *);
extern int    CoMultiply(UBYTE *);
extern int    CoNFunction(UBYTE *);
extern int    CoNPrint(UBYTE *);
extern int    CoNTensor(UBYTE *);
extern int    CoNWrite(UBYTE *);
extern int    CoNoDrop(UBYTE *);
extern int    CoNoSkip(UBYTE *);
extern int    CoNormalize(UBYTE *);
extern int    CoMakeInteger(UBYTE *);
extern int    CoFlags(UBYTE *,int);
extern int    CoOff(UBYTE *);
extern int    CoOn(UBYTE *);
extern int    CoOnce(UBYTE *);
extern int    CoOnly(UBYTE *);
extern int    CoOptimizeOption(UBYTE *);
extern int    CoOptimize(UBYTE *);
extern int    CoPolyFun(UBYTE *);
extern int    CoPolyRatFun(UBYTE *);
extern int    CoPrint(UBYTE *);
extern int    CoPrintB(UBYTE *);
extern int    CoProperCount(UBYTE *);
extern int    CoUnitTrace(UBYTE *);
extern int    CoRCycleSymmetrize(UBYTE *);
extern int    CoRatio(UBYTE *);
extern int    CoRedefine(UBYTE *);
extern int    CoRenumber(UBYTE *);
extern int    CoRepeat(UBYTE *);
extern int    CoSave(UBYTE *);
extern int    CoSelect(UBYTE *);
extern int    CoSet(UBYTE *);
extern int    CoSetExitFlag(UBYTE *);
extern int    CoSkip(UBYTE *);
extern int    CoProcessBucket(UBYTE *);
extern int    CoPushHide(UBYTE *);
extern int    CoPopHide(UBYTE *);
extern int    CoHide(UBYTE *);
extern int    CoIntoHide(UBYTE *);
extern int    CoNoHide(UBYTE *);
extern int    CoUnHide(UBYTE *);
extern int    CoNoUnHide(UBYTE *);
extern int    CoSort(UBYTE *);
extern int    CoSplitArg(UBYTE *);
extern int    CoSplitFirstArg(UBYTE *);
extern int    CoSplitLastArg(UBYTE *);
extern int    CoSum(UBYTE *);
extern int    CoSymbol(UBYTE *);
extern int    CoSymmetrize(UBYTE *);
extern int    DoTable(UBYTE *,int);
extern int    CoTable(UBYTE *);
extern int    CoTerm(UBYTE *);
extern int    CoNTable(UBYTE *);
extern int    CoCTable(UBYTE *);
extern void   EmptyTable(TABLES);
extern int    CoToTensor(UBYTE *);
extern int    CoToVector(UBYTE *);
extern int    CoTrace4(UBYTE *);
extern int    CoTraceN(UBYTE *);
extern int    CoChisholm(UBYTE *);
extern int    CoTransform(UBYTE *);
extern int    CoClearTable(UBYTE *);
extern int    DoChain(UBYTE *,int);
extern int    CoChainin(UBYTE *);
extern int    CoChainout(UBYTE *);
extern int    CoTryReplace(UBYTE *);
extern int    CoVector(UBYTE *);
extern int    CoWhile(UBYTE *);
extern int    CoWrite(UBYTE *);
extern int    CoAuto(UBYTE *);
extern int    CoTBaddto(UBYTE *);
extern int    CoTBaudit(UBYTE *);
extern int    CoTBcleanup(UBYTE *);
extern int    CoTBcreate(UBYTE *);
extern int    CoTBenter(UBYTE *);
extern int    CoTBhelp(UBYTE *);
extern int    CoTBload(UBYTE *);
extern int    CoTBoff(UBYTE *);
extern int    CoTBon(UBYTE *);
extern int    CoTBopen(UBYTE *);
```

```
extern int    CoTBreplace(UBYTE *);
extern int    CoTBuse(UBYTE *);
extern int    CoTestUse(UBYTE *);
extern int    CoThreadBucket(UBYTE *);
extern int    AddComString(int,WORD *,UBYTE *,int);
extern int    CompileAlgebra(UBYTE *,int,WORD *);
extern int    IsIdStatement(UBYTE *);
extern UBYTE *IsRHS(UBYTE *,UBYTE);
extern int    ParenthesesTest(UBYTE *);
extern int    tokenize(UBYTE *,WORD);
extern void   WriteTokens(SBYTE *);
extern int    simp1token(SBYTE *);
extern int    simpwtoken(SBYTE *);
extern int    simp2token(SBYTE *);
extern int    simp3atoken(SBYTE *,int);
extern int    simp3btoken(SBYTE *,int);
extern int    simp4token(SBYTE *);
extern int    simp5token(SBYTE *,int);
extern int    simp6token(SBYTE *,int);
extern UBYTE *SkipAName(UBYTE *);
extern int    TestTables(VOID);
extern int    GetLabel(UBYTE *);
extern int    CoIdExpression(UBYTE *,int);
extern int    CoAssign(UBYTE *);
extern int    DoExpr(UBYTE *,int,int);
extern int    CompileSubExpressions(SBYTE *);
extern int    CodeGenerator(SBYTE *);
extern int    CompleteTerm(WORD *,UWORD *,UWORD *,WORD,WORD,int);
extern int    CodeFactors(SBYTE *s);
extern WORD   GenerateFactors(WORD,WORD);
extern int    InsTree(int,int);
extern int    FindTree(int,WORD *);
extern void   RedoTree(CBUF *,int);
extern void   ClearTree(int);
extern int    CatchDollar(int);
extern int    AssignDollar(PHEAD WORD *,WORD);
extern UBYTE *WriteDollarToBuffer(WORD,WORD);
extern UBYTE *WriteDollarFactorToBuffer(WORD,WORD,WORD);
extern void   AddToDollarBuffer(UBYTE *);
extern void   TermAssign(WORD *);
extern void   WildDollars(PHEAD WORD *);
extern LONG   numcommute(WORD *,LONG *);
extern int    FullRenumber(PHEAD WORD *,WORD);
extern int    Lus(WORD *,WORD,WORD,WORD,WORD,WORD);
extern int    FindLus(int,int,int);
extern int    CoReplaceLoop(UBYTE *);
extern int    CoFindLoop(UBYTE *);
extern int    DoFindLoop(UBYTE *,int);
extern int    CoFunPowers(UBYTE *);
extern int    SortTheList(int *,int);
extern int    MatchIsPossible(WORD *,WORD *);
extern int    StudyPattern(WORD *);
extern WORD   DolToTensor(PHEAD WORD);
extern WORD   DolToFunction(PHEAD WORD);
extern WORD   DolToVector(PHEAD WORD);
extern WORD   DolToNumber(PHEAD WORD);
extern WORD   DolToSymbol(PHEAD WORD);
extern WORD   DolToIndex(PHEAD WORD);
extern LONG   DolToLong(PHEAD WORD);
extern int    DollarFactorize(PHEAD WORD);
extern int    CoPrintTable(UBYTE *);
extern int    CoDeallocateTable(UBYTE *);
extern void   CleanDollarFactors(DOLLARS);
extern WORD   *TakeDollarContent(PHEAD WORD *,WORD **);
extern WORD   *MakeDollarInteger(PHEAD WORD *,WORD **);
extern WORD   *MakeDollarMod(PHEAD WORD *,WORD **);
extern int    GetDolNum(PHEAD WORD *, WORD *);
extern void   AddPotModdollar(WORD);

extern int    Optimize(WORD, int);
extern int    ClearOptimize(VOID);
extern int    LoadOpti(WORD);
extern int    PutObject(WORD *,int);
extern void   CleanOptiBuffer(VOID);
extern int    PrintOptima(WORD);
extern int    FindScratchName(VOID);
extern WORD   MaxPowerOpti(LONG);
extern WORD   HuntNumFactor(LONG,WORD *,int);
extern WORD   HuntFactor(LONG,WORD *,int);
extern void   HuntPairs(LONG,WORD);
extern void   HuntBrackets(LONG);
extern int    AddToOpti(WORD *,int);
extern LONG   TestNewSca(LONG,WORD *,WORD *);
extern void   NormOpti(WORD *);
extern void   SortOpti(LONG);
extern void   SplitOpti(WORD **,LONG);
extern void   CombiOpti(VOID);
```

```
extern int    TakeLongRoot(UWORD *,WORD *,WORD);
extern int    TakeRatRoot(UWORD *,WORD *,WORD);
extern int    MakeRational(WORD ,WORD , WORD *, WORD *);
extern int    MakeLongRational(PHEAD UWORD *,WORD ,UWORD *,WORD ,UWORD *,WORD *);
extern void   HuntPowers(LONG,WORD);
extern void   HuntNumBrackets(LONG);
extern void   ClearTableTree(TABLES);
extern int    InsTableTree(TABLES,WORD *);
extern void   RedoTableTree(TABLES,int);
extern int    FindTableTree(TABLES,WORD *,int);
extern void   finishcbuf(WORD);
extern void   clearcbuf(WORD);
extern void   CleanUpSort(int);
extern FILEHANDLE *AllocFileHandle(WORD,char *);
extern VOID   DeAllocFileHandle(FILEHANDLE *);
extern VOID   LowerSortLevel(VOID);
extern WORD  *PolyRatFunSpecial(PHEAD WORD *, WORD *);
extern int    InsideDollar(PHEAD WORD *,WORD);
extern DOLLARS DolToTerms(PHEAD WORD);
extern WORD   EvalDoLoopArg(PHEAD WORD *,WORD);
extern int    SetExprCases(int,int,int);
extern int    TestSelect(WORD *,WORD *);
extern VOID   SubsInAll(PHEAD0);
extern VOID   TransferBuffer(int,int,int);
extern int    TakeIDfunction(PHEAD WORD *);
extern int    MakeSetupAllocs(VOID);
extern int    TryFileSetups(VOID);
extern void   ExchangeExpressions(int,int);
extern void   ExchangeDollars(int,int);
extern int    GetFirstBracket(WORD *,int);
extern int    GetFirstTerm(WORD *,int);
extern int    GetContent(WORD *,int);
extern int    CleanupTerm(WORD *);
extern WORD   ContentMerge(PHEAD WORD *,WORD *);
extern UBYTE *PreIfDollarEval(UBYTE *,int *);
extern LONG   TermsInDollar(WORD);
extern LONG   TermsInExpression(WORD);
extern WORD   *TranslateExpression(UBYTE *);
extern int    IsSetMember(WORD *,WORD);
extern int    IsMultipleOf(WORD *,WORD *);
extern int    TwoExprCompare(WORD *,WORD *,int);
extern void   UpdatePositions(VOID);
extern void   M_check(VOID);
extern void   M_print(VOID);
extern void   M_check1(VOID);
extern void   PrintTime(VOID);

extern POSITION *FindBracket(WORD,WORD *);
extern VOID   PutBracketInIndex(PHEAD WORD *,POSITION *);
extern void   ClearBracketIndex(WORD);
extern VOID   OpenBracketIndex(WORD);
extern int    DoNoParallel(UBYTE *);
extern int    DoParallel(UBYTE *);
extern int    DoModSum(UBYTE *);
extern int    DoModMax(UBYTE *);
extern int    DoModMin(UBYTE *);
extern int    DoModLocal(UBYTE *);
extern UBYTE *DoModDollar(UBYTE *,int);
extern int    DoProcessBucket(UBYTE *);
extern int    DoinParallel(UBYTE *);
extern int    DonotinParallel(UBYTE *);

extern int    FlipTable(FUNCTIONS,int);
extern int    ChainIn(PHEAD WORD *,WORD);
extern int    ChainOut(PHEAD WORD *,WORD);
extern int    ArgumentImplode(PHEAD WORD *,WORD *);
extern int    ArgumentExplode(PHEAD WORD *,WORD *);
extern int    DenToFunction(WORD *,WORD);

extern WORD   HowMany(PHEAD WORD *,WORD *);
extern VOID   RemoveDollars(VOID);
extern LONG   CountTerms1(PHEAD0);
extern LONG   TermsInBracket(PHEAD WORD *,WORD);
extern int    Crash(VOID);

extern char  *str_dup(char *);
extern void   convertblock(INDEXBLOCK *,INDEXBLOCK *,int);
extern void   convertnamesblock(NAMESBLOCK *,NAMESBLOCK *,int);
extern void   convertiniinfo(INIINFO *,INIINFO *,int);
extern int    ReadIndex(DBASE *);
extern int    WriteIndexBlock(DBASE *,MLONG);
extern int    WriteNamesBlock(DBASE *,MLONG);
extern int    WriteIndex(DBASE *);
extern int    WriteIniInfo(DBASE *);
extern int    ReadIniInfo(DBASE *);
extern int    AddToIndex(DBASE *,MLONG);
extern DBASE *GetDbase(char *);
```

```
extern DBASE *OpenDbase(char *);
extern char  *ReadObject(DBASE *,MLONG,char *);
extern char  *ReadijObject(DBASE *,MLONG,MLONG,char *);
extern int    ExistsObject(DBASE *,MLONG,char *);
extern int    DeleteObject(DBASE *,MLONG,char *);
extern int    WriteObject(DBASE *,MLONG,char *,char *,MLONG);
extern MLONG  AddObject(DBASE *,MLONG,char *,char *);
extern int    Cleanup(DBASE *);
extern DBASE *NewDbase(char *,MLONG);
extern void   FreeTableBase(DBASE *);
extern int    ComposeTableNames(DBASE *);
extern int    PutTableNames(DBASE *);
extern MLONG  AddTableName(DBASE *,char *,TABLES);
extern MLONG  GetTableName(DBASE *,char *);
extern MLONG  FindTableNumber(DBASE *,char *);
extern int    TryEnvironment(VOID);

#ifdef WITHZLIB
extern int    SetupOutputGZIP(FILEHANDLE *);
extern int    PutOutputGZIP(FILEHANDLE *);
extern int    FlushOutputGZIP(FILEHANDLE *);
extern int    SetupAllInputGZIP(SORTING *);
extern LONG   FillInputGZIP(FILEHANDLE *,POSITION *,UBYTE *,LONG,int);
#endif

#ifdef WITHPTHREADS
extern VOID   BeginIdentities(VOID);
extern int    WhoAmI(VOID);
extern int    StartAllThreads(int);
extern void   StartHandleLock(VOID);
extern VOID   TerminateAllThreads(VOID);
extern int    GetAvailableThread(VOID);
extern int    ConditionalGetAvailableThread(VOID);
extern int    BalanceRunThread(PHEAD int,WORD *,WORD);
extern void   WakeupThread(int,int);
extern int    MasterWait(VOID);
extern int    InParallelProcessor(VOID);
extern int    ThreadsProcessor(EXPRESSIONS,WORD,WORD);
extern int    MasterMerge(VOID);
extern int    PutToMaster(PHEAD WORD *);
extern void   SetWorkerFiles(VOID);
extern int    MakeThreadBuckets(int,int);
extern int    SendOneBucket(int);
extern int    LoadOneThread(int,int,THREADBUCKET *,int);
extern void  *RunSortBot(void *);
extern void   MasterWaitAllSortBots(VOID);
extern int    SortBotMerge(PHEAD0);
extern int    SortBotOut(PHEAD WORD *);
extern void   DefineSortBotTree(VOID);
extern int    SortBotMasterMerge(VOID);
extern int    SortBotWait(int);
extern void   StartIdentity(VOID);
extern void   FinishIdentity(void *);
extern int    SetIdentity(int *);
extern ALLPRIVATES *InitializeOneThread(int);
extern void   FinalizeOneThread(int);
extern void   ClearAllThreads(VOID);
extern void  *RunThread(void *);
extern void   IAmAvailable(int);
extern int    ThreadWait(int);
extern int    ThreadClaimedBlock(int);
extern int    GetThread(int);
extern int    UpdateOneThread(int);
extern void   MasterWaitAll(VOID);
extern void   MasterWaitAllBlocks(VOID);
extern int    MasterWaitThread(int);
extern void   WakeupMasterFromThread(int,int);
extern int    LoadReadjusted(VOID);
extern int    IniSortBlocks(int);
extern int    TreatIndexEntry(PHEAD LONG);
extern WORD   GetTerm2(PHEAD WORD *);
extern void SetHideFiles(VOID);

#endif

extern int    CopyExpression(FILEHANDLE *,FILEHANDLE *);

extern int    set_in(UBYTE, set_of_char);
extern one_byte set_set(UBYTE, set_of_char);
extern one_byte set_del(UBYTE, set_of_char);
extern one_byte set_sub (set_of_char, set_of_char, set_of_char);
extern int    DoPreAddSeparator(UBYTE *);
extern int    DoPreRmSeparator(UBYTE *);

/*See the file extcmd.c*/
extern int    openExternalChannel(UBYTE *,int,UBYTE *,UBYTE *);
extern int    initPresetExternalChannels(UBYTE *, int);
```

```
extern int    closeExternalChannel(int);
extern int    selectExternalChannel(int);
extern int    getCurrentExternalChannel(VOID);
extern VOID   closeAllExternalChannels(VOID);

typedef int (*WRITEBUFTOEXTCHANNEL)(char *,size_t);
typedef int (*GETCFROMEXTCHANNEL)(VOID);
typedef int (*SETTERMINATORFOREXTERNALCHANNEL)(char *);
typedef int (*SETKILLMODEFOREXTERNALCHANNEL)(int,int);
typedef LONG (*WRITEFILE)(int,UBYTE *,LONG);
typedef WORD (*COMPARE)(PHEAD WORD *,WORD *,WORD);
typedef WORD (*GETTERM)(PHEAD WORD *);
typedef WORD (*FINISHUFFLE)(PHEAD WORD *);
typedef WORD (*DO_UFFLE)(PHEAD WORD *,WORD,WORD,WORD);

#define CompareTerms ((COMPARE)AR.CompareRoutine)
#define FiniShuffle ((FINISHUFFLE)AN.SHvar.finishuf)
#define DoShtuffle ((DO_UFFLE)AN.SHvar.do_uffle)

extern UBYTE *defineChannel(UBYTE*, HANDLERS*);
extern int    writeToChannel(int,UBYTE *,HANDLERS*);
#ifdef WITHEXTERNALCHANNEL
extern LONG   WriteToExternalChannel(int,UBYTE *,LONG);
#endif
extern int    writeBufToExtChannelOk(char *,size_t);
extern int    getcFromExtChannelOk(VOID);
extern int    setKillModeForExternalChannelOk(int,int);
extern int    setTerminatorForExternalChannelOk(char *);
extern int    getcFromExtChannelFailure(VOID);
extern int    setKillModeForExternalChannelFailure(int,int);
extern int    setTerminatorForExternalChannelFailure(char *);
extern int    writeBufToExtChannelFailure(char *,size_t);

extern int    ReleaseTB(VOID);

extern int    SymbolNormalize(WORD *);
extern int    TestFunFlag(PHEAD WORD *);
extern int    CompareSymbols(PHEAD WORD *,WORD *,WORD);
extern int    CompareHSymbols(PHEAD WORD *,WORD *,WORD);
extern WORD   NextPrime(PHEAD WORD);
extern UWORD  wranf(PHEAD0);
extern UWORD  iranf(PHEAD UWORD);
extern void   iniwranf(PHEAD0);
extern UBYTE *PreRandom(UBYTE *);

extern WORD *PolyNormPoly (PHEAD WORD);
extern WORD  *EvaluateGcd(PHEAD WORD *);
extern int TreatPolyRatFun(PHEAD WORD *);

extern WORD   ReadSaveHeader(VOID);
extern WORD   ReadSaveIndex(FILEINDEX *);
extern WORD   ReadSaveExpression(UBYTE *,UBYTE *,LONG *,LONG *);
extern UBYTE *ReadSaveTerm32(UBYTE *,UBYTE *,UBYTE **,UBYTE *,UBYTE *,int);
extern WORD   ReadSaveVariables(UBYTE *,UBYTE *,LONG *,LONG *,INDEXENTRY *,LONG *);
extern WORD   WriteStoreHeader(WORD);

extern void   InitRecovery(VOID);
extern int    CheckRecoveryFile(VOID);
extern void   DeleteRecoveryFile(VOID);
extern char  *RecoveryFilename(VOID);
extern int    DoRecovery(int *);
extern void   DoCheckpoint(int);

extern VOID NumberMallocAddMemory(PHEAD0);
extern VOID CacheNumberMallocAddMemory(PHEAD0);
extern VOID TermMallocAddMemory(PHEAD0);
#ifndef MEMORYMACROS
extern WORD *TermMalloc2(PHEAD char *text);
extern VOID TermFree2(PHEAD WORD *term,char *text);
extern UWORD *NumberMalloc2(PHEAD char *text);
extern UWORD *CacheNumberMalloc2(PHEAD char *text);
extern VOID NumberFree2(PHEAD UWORD *NumberMem,char *text);
extern VOID CacheNumberFree2(PHEAD UWORD *NumberMem,char *text);
#endif

extern void ExprStatus(EXPRESSIONS);
extern VOID iniTools(VOID);
extern int TestTerm(WORD *);

extern WORD RunTransform(PHEAD WORD *term, WORD *params);
extern WORD RunEncode(PHEAD WORD *fun, WORD *args, WORD *info);
extern WORD RunDecode(PHEAD WORD *fun, WORD *args, WORD *info);
extern WORD RunReplace(PHEAD WORD *fun, WORD *args, WORD *info);
extern WORD RunImplode(WORD *fun, WORD *args);
extern WORD RunExplode(PHEAD WORD *fun, WORD *args);
extern int TestArgNum(int n, int totarg, WORD *args);
extern WORD PutArgInScratch(WORD *arg,UWORD *scrat);
```

```
extern UBYTE *ReadRange(UBYTE *s, WORD *out, int par);
extern int  FindRange(PHEAD WORD *,WORD *,WORD *,WORD);
extern WORD RunPermute(PHEAD WORD *fun, WORD *args, WORD *info);
extern WORD RunReverse(PHEAD WORD *fun, WORD *args);
extern WORD RunCycle(PHEAD WORD *fun, WORD *args, WORD *info);
extern WORD RunAddArg(PHEAD WORD *fun, WORD *args);
extern WORD RunMulArg(PHEAD WORD *fun, WORD *args);
extern WORD RunIsLyndon(PHEAD WORD *fun, WORD *args, int par);
extern WORD RunToLyndon(PHEAD WORD *fun, WORD *args, int par);
extern WORD RunDropArg(PHEAD WORD *fun, WORD *args);
extern WORD RunSelectArg(PHEAD WORD *fun, WORD *args);
extern WORD RunDedup(PHEAD WORD *fun, WORD *args);

extern int NormPolyTerm(PHEAD WORD *);
extern WORD ComparePoly(WORD *, WORD *, WORD);
extern int ConvertToPoly(PHEAD WORD *, WORD *,WORD *,WORD);
extern int LocalConvertToPoly(PHEAD WORD *, WORD *, WORD,WORD);
extern int ConvertFromPoly(PHEAD WORD *, WORD *, WORD, WORD, WORD, WORD);
extern WORD FindSubterm(WORD *);
extern WORD FindLocalSubterm(PHEAD WORD *, WORD);
extern void PrintSubtermList(int,int);
extern void PrintExtraSymbol(int,WORD *,int);
extern WORD FindSubexpression(WORD *);

extern void UpdateMaxSize(VOID);

extern int CoToPolynomial(UBYTE *);
extern int CoFromPolynomial(UBYTE *);
extern int CoArgToExtraSymbol(UBYTE *);
extern int CoExtraSymbols(UBYTE *);
extern UBYTE *GetDoParam(UBYTE *, WORD **, int);
extern WORD *GetIfDollarFactor(UBYTE **, WORD *);
extern int CoDo(UBYTE *);
extern int CoEndDo(UBYTE *);
extern int ExtraSymFun(PHEAD WORD *,WORD);
extern int PruneExtraSymbols(WORD);
extern int IniFbuffer(WORD);
extern void IniFbufs(VOID);
extern int GCDfunction(PHEAD WORD *,WORD);
extern WORD *GCDfunction3(PHEAD WORD *,WORD *);
extern WORD *GCDfunction4(PHEAD WORD *,WORD *);
extern int ReadPolyRatFun(PHEAD WORD *);
extern int FromPolyRatFun(PHEAD WORD *, WORD **, WORD **);
extern void PRFnormalize(PHEAD WORD *);
extern WORD *PRFadd(PHEAD WORD *, WORD *);
extern WORD *PolyDiv(PHEAD WORD *,WORD *,char *);
extern WORD *PolyGCD(PHEAD WORD *,WORD *);
extern WORD *PolyAdd(PHEAD WORD *,WORD *);
extern void GCDclean(PHEAD WORD *, WORD *);
extern int RatFunNormalize(PHEAD WORD *);
extern WORD *TakeSymbolContent(PHEAD WORD *,WORD *);
extern int GCDterms(PHEAD WORD *,WORD *,WORD *);
extern WORD *PutExtraSymbols(PHEAD WORD *,WORD,int *);
extern WORD *TakeExtraSymbols(PHEAD WORD *,WORD);
extern WORD *MultiplyWithTerm(PHEAD WORD *, WORD *,WORD);
extern WORD *TakeContent(PHEAD WORD *, WORD *);
extern int MergeSymbolLists(PHEAD WORD *, WORD *, int);
extern int MergeDotproductLists(PHEAD WORD *, WORD *, int);
extern WORD *CreateExpression(PHEAD WORD);
extern int DIVfunction(PHEAD WORD *,WORD,int);
extern WORD *MULfunc(PHEAD WORD *, WORD *);
extern WORD *ConvertArgument(PHEAD WORD *,int *);
extern int ExpandRat(PHEAD WORD *);
extern int InvPoly(PHEAD WORD *,WORD,WORD);
extern WORD TestDoLoop(PHEAD WORD *,WORD);
extern WORD TestEndDoLoop(PHEAD WORD *,WORD);

extern WORD  *poly_gcd(PHEAD WORD *, WORD *, WORD);
extern WORD  *poly_div(PHEAD WORD *, WORD *, WORD);
extern WORD  *poly_rem(PHEAD WORD *, WORD *, WORD);
extern WORD  *poly_inverse(PHEAD WORD *, WORD *);
extern WORD  *poly_mul(PHEAD WORD *, WORD *);
extern WORD *poly_ratfun_add(PHEAD WORD *, WORD *);
extern int   poly_ratfun_normalize(PHEAD WORD *);
extern int   poly_factorize_argument(PHEAD WORD *, WORD *);
extern WORD *poly_factorize_dollar(PHEAD WORD *);
extern int   poly_factorize_expression(EXPRESSIONS);
extern int   poly_unfactorize_expression(EXPRESSIONS);
extern void  poly_free_poly_vars(PHEAD const char *);

extern VOID optimize_print_code (int);

#ifdef WITHPTHREADS
extern void find_Horner_MCTS_expand_tree();
extern void find_Horner_MCTS_expand_tree_threaded();
extern void optimize_expression_given_Horner();
extern void optimize_expression_given_Horner_threaded();
```

```
#endif

extern int DoPreAdd(UBYTE *s);
extern int DoPreUseDictionary(UBYTE *s);
extern int DoPreCloseDictionary(UBYTE *s);
extern int DoPreOpenDictionary(UBYTE *s);
extern void RemoveDictionary(DICTIONARY *dict);
extern void UnSetDictionary(VOID);
extern int SetDictionaryOptions(UBYTE *options);
extern int SelectDictionary(UBYTE *name,UBYTE *options);
extern int AddToDictionary(DICTIONARY *dict,UBYTE *left,UBYTE *right);
extern int AddDictionary(UBYTE *name);
extern int FindDictionary(UBYTE *name);
extern int IsExponentSign(VOID);
extern int IsMultiplySign(VOID);
extern VOID TransformRational(UWORD *a, WORD na);
extern void WriteDictionary(DICTIONARY *);
extern void ShrinkDictionary(DICTIONARY *);
extern void MultiplyToLine(VOID);
extern UBYTE *FindSymbol(WORD num);
extern UBYTE *FindVector(WORD num);
extern UBYTE *FindIndex(WORD num);
extern UBYTE *FindFunction(WORD num);
extern UBYTE *FindFunWithArgs(WORD *t);
extern UBYTE *FindExtraSymbol(WORD num);
extern LONG DictToBytes(DICTIONARY *dict,UBYTE *buf);
extern DICTIONARY *DictFromBytes(UBYTE *buf);
extern int CoCreateSpectator(UBYTE *inp);
extern int CoToSpectator(UBYTE *inp);
extern int CoRemoveSpectator(UBYTE *inp);
extern int CoEmptySpectator(UBYTE *inp);
extern int CoCopySpectator(UBYTE *inp);
extern int PutInSpectator(WORD *,WORD);
extern void ClearSpectators(WORD);
extern WORD GetFromSpectator(WORD *,WORD);
extern void FlushSpectators(VOID);

extern WORD *PreGCD(PHEAD WORD *, WORD *,int);
extern WORD *FindCommonVariables(PHEAD int,int);
extern VOID AddToSymbolList(PHEAD WORD);
extern int AddToListPoly(PHEAD0);
extern int InvPoly(PHEAD WORD *,WORD,WORD);

extern int ReadFromScratch(FILEHANDLE *,POSITION *,UBYTE *,POSITION *);
extern int AddToScratch(FILEHANDLE *,POSITION *,UBYTE *,POSITION *,int);

extern int DoPreAppendPath(UBYTE *);
extern int DoPrePrependPath(UBYTE *);

/*
    #] Declarations :
*/
#endif
```

4.21 dict.c File Reference

`#include "form3.h"`

Include dependency graph for dict.c:

Functions

- VOID TransformRational (UWORD ∗a, WORD na)
- int IsMultiplySign (VOID)
- int IsExponentSign (VOID)
- UBYTE ∗ FindSymbol (WORD num)
- UBYTE ∗ FindVector (WORD num)
- UBYTE ∗ FindIndex (WORD num)
- UBYTE ∗ FindFunction (WORD num)
- UBYTE ∗ FindFunWithArgs (WORD ∗t)
- UBYTE ∗ FindExtraSymbol (WORD num)
- int FindDictionary (UBYTE ∗name)
- int AddDictionary (UBYTE ∗name)
- int AddToDictionary (DICTIONARY ∗dict, UBYTE ∗left, UBYTE ∗right)
- int UseDictionary (UBYTE ∗name, UBYTE ∗options)
- int SetDictionaryOptions (UBYTE ∗options)
- void UnSetDictionary (VOID)
- void RemoveDictionary (DICTIONARY ∗dict)
- void ShrinkDictionary (DICTIONARY ∗dict)
- int DoPreOpenDictionary (UBYTE ∗s)
- int DoPreCloseDictionary (UBYTE ∗s)
- int DoPreUseDictionary (UBYTE ∗s)
- int DoPreAdd (UBYTE ∗s)
- LONG DictToBytes (DICTIONARY ∗dict, UBYTE ∗buf)
- DICTIONARY ∗ DictFromBytes (UBYTE ∗buf)

4.21.1 Detailed Description

Contains the code pertaining to dictionaries Commands are: #opendictionary name #closedictionary #selectdictionary name <options> There can be several dictionaries, but only one can be active. Defining elements is done with #add object: "replacement" Replacements are strings when a dictionary is for output translation. Objects can be 1: a number (rational) 2: a variable 3: ∗ ^ 4: a function with arguments

Definition in file dict.c.

4.21.2 Function Documentation

4.21.2.1 TransformRational()

```
VOID TransformRational (
            UWORD * a,
            WORD na )
```

Definition at line 71 of file dict.c.

4.21.2.2 IsMultiplySign()

```
int IsMultiplySign (
            VOID  )
```

Definition at line 218 of file dict.c.

4.21.2.3 IsExponentSign()

```
int IsExponentSign (
            VOID  )
```

Definition at line 237 of file dict.c.

4.21.2.4 FindSymbol()

```
UBYTE * FindSymbol (
            WORD num )
```

Definition at line 256 of file dict.c.

4.21.2.5 FindVector()

```
UBYTE * FindVector (
            WORD num )
```

Definition at line 277 of file dict.c.

4.21.2.6 FindIndex()

```
UBYTE * FindIndex (
            WORD num )
```

Definition at line 299 of file dict.c.

4.21.2.7 FindFunction()

```
UBYTE * FindFunction (
            WORD num )
```

Definition at line 321 of file dict.c.

4.21.2.8 FindFunWithArgs()

```
UBYTE * FindFunWithArgs (
            WORD * t )
```

Definition at line 343 of file dict.c.

4.21.2.9 FindExtraSymbol()

```
UBYTE * FindExtraSymbol (
            WORD num )
```

Definition at line 375 of file dict.c.

4.21.2.10 FindDictionary()

```
int FindDictionary (
            UBYTE * name )
```

Definition at line 432 of file dict.c.

4.21.2.11 AddDictionary()

```
int AddDictionary (
            UBYTE * name )
```

Definition at line 447 of file dict.c.

4.21.2.12 AddToDictionary()

```
int AddToDictionary (
            DICTIONARY * dict,
            UBYTE * left,
            UBYTE * right )
```

Definition at line 489 of file dict.c.

4.21.2.13 UseDictionary()

```
int UseDictionary (
            UBYTE * name,
            UBYTE * options )
```

Definition at line 764 of file dict.c.

4.21.2.14 SetDictionaryOptions()

```
int SetDictionaryOptions (
            UBYTE * options )
```

Definition at line 788 of file dict.c.

4.21.2.15 UnSetDictionary()

```
void UnSetDictionary (
            VOID  )
```

Definition at line 881 of file dict.c.

4.21.2.16 RemoveDictionary()

```
void RemoveDictionary (
            DICTIONARY * dict )
```

Definition at line 900 of file dict.c.

4.21.2.17 ShrinkDictionary()

```
void ShrinkDictionary (
            DICTIONARY * dict )
```

Definition at line 943 of file dict.c.

4.21.2.18 DoPreOpenDictionary()

```
int DoPreOpenDictionary (
            UBYTE * s )
```

Definition at line 957 of file dict.c.

4.21.2.19 DoPreCloseDictionary()

```
int DoPreCloseDictionary (
            UBYTE * s )
```

Definition at line 996 of file dict.c.

4.21.2.20 DoPreUseDictionary()

```
int DoPreUseDictionary (
            UBYTE * s )
```

Definition at line 1020 of file dict.c.

4.21.2.21 DoPreAdd()

```
int DoPreAdd (
            UBYTE * s )
```

Definition at line 1065 of file dict.c.

4.21.2.22 DictToBytes()

```
LONG DictToBytes (
            DICTIONARY * dict,
            UBYTE * buf )
```

Definition at line 1117 of file dict.c.

4.21.2.23 DictFromBytes()

```
DICTIONARY * DictFromBytes (
            UBYTE * buf )
```

Definition at line 1150 of file dict.c.

4.22 dict.c

Go to the documentation of this file.

```
00001
00018 /* #[ License : */
00019 /*
00020  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00021  *   When using this file you are requested to refer to the publication
00022  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00023  *   This is considered a matter of courtesy as the development was paid
00024  *   for by FOM the Dutch physics granting agency and we would like to
00025  *   be able to track its scientific use to convince FOM of its value
00026  *   for the community.
00027  *
00028  *   This file is part of FORM.
00029  *
00030  *   FORM is free software: you can redistribute it and/or modify it under the
00031  *   terms of the GNU General Public License as published by the Free Software
00032  *   Foundation, either version 3 of the License, or (at your option) any later
00033  *   version.
00034  *
00035  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00036  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00037  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00038  *   details.
00039  *
00040  *   You should have received a copy of the GNU General Public License along
00041  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00042  */
00043 /* #] License : */
00044 /*
00045     #[ Includes : ratio.c
00046
00047     Data setup:
00048         AO.Dictionaries          Array of pointers to DICTIONARY
00049         AO.NumDictionaries
00050         AO.SizeDictionaries
00051         AO.CurrentDictionary
00052         AO.CurDictNumbers
00053         AO.CurDictVariables
00054         AO.CurDictSpecials
00055         AP.OpenDictionary
00056 */
00057
00058 #include "form3.h"
00059
00060 /*
00061     #] Includes :
00062     #[ TransformRational:
00063
00064     Tries to transform the rational number a according to the rules of
00065     the current dictionary. Whatever cannot be translated goes to the
00066     regular output.
00067     Options for AO.CurDictNumbers are:
00068         DICT_ALLNUMBERS, DICT_RATIONALONLY, DICT_INTEGERONLY, DICT_NONUMBERS
00069 */
00070
00071 VOID TransformRational(UWORD *a, WORD na)
00072 {
00073     DICTIONARY *dict;
00074     WORD i, j, nb, i1, i2; UWORD *b;
00075     if ( AO.CurrentDictionary <= 0 ) goto NoAction;
00076     dict = AO.Dictionaries[AO.CurrentDictionary-1];
00077     if ( na < 0 ) na = -na;
00078     switch ( AO.CurDictNumbers ) {
00079         case DICT_NONUMBERS:
00080             goto NoAction;
00081         case DICT_INTEGERONLY:
00082             if ( a[na] != 1 ) goto NoAction;
00083             if ( na > 1 ) {
00084                 for ( i = 1; i < na; i++ ) {
```

```
                    if ( a[na+i] != 0 ) goto NoAction;
                }
            }
Numeratoronly:;
            for ( i = dict->numelements-1; i >= 0; i-- ) {
                if ( dict->elements[i]->type == DICT_INTEGERNUMBER ) {
                    if ( dict->elements[i]->size == na ) {
                        for ( j = 0; j < na; j++ ) {
                            if ( (UWORD)(dict->elements[i]->lhs[j]) != a[j] ) break;
                        }
                        if ( j == na ) {    /* Got it */
                            TokenToLine((UBYTE *)(dict->elements[i]->rhs));
                            return;
                        }
                    }
                }
            }
            goto NotFound;
        case DICT_RATIONALONLY:
            nb = 2*na;
            for ( i = dict->numelements-1; i >= 0; i-- ) {
                if ( dict->elements[i]->type == DICT_RATIONALNUMBER ) {
                    if ( dict->elements[i]->size == nb+2 ) {
                        for ( j = 0; j < nb; j++ ) {
                            if ( (UWORD)(dict->elements[i]->lhs[j+1]) != a[j] ) break;
                        }
                        if ( j == nb ) {    /* Got it */
                            TokenToLine((UBYTE *)(dict->elements[i]->rhs));
                            return;
                        }
                    }
                }
            }
            goto NotFound;
        case DICT_ALLNUMBERS:
/*
            First fish for rationals
*/
            nb = 2*na;
            for ( i = dict->numelements-1; i >= 0; i-- ) {
                if ( dict->elements[i]->type == DICT_RATIONALNUMBER ) {
                    if ( dict->elements[i]->size == nb+2 ) {
                        for ( j = 0; j < nb; j++ ) {
                            if ( (UWORD)(dict->elements[i]->lhs[j+1]) != a[j] ) break;
                        }
                        if ( j == nb ) {    /* Got it */
                            TokenToLine((UBYTE *)(dict->elements[i]->rhs));
                            return;
                        }
                    }
                }
            }
/*
            Now look for element[j1]/element[j2]
*/
            nb = na; b = a+na;
            while ( b[nb-1] == 0 ) nb--;
            if ( nb == 1 && b[0] == 1 ) goto Numeratoronly;
            while ( a[na-1] == 0 ) na--;
            for ( i1 = dict->numelements-1; i1 >= 0; i1-- ) {
                if ( dict->elements[i1]->type == DICT_INTEGERNUMBER ) {
                    if ( dict->elements[i1]->size == na ) {
                        for ( j = 0; j < na; j++ ) {
                            if ( (UWORD)(dict->elements[i1]->lhs[j]) != a[j] ) break;
                        }
                        if ( j == na ) break;
                    }
                }
            }
            for ( i2 = dict->numelements-1; i2 >= 0; i2-- ) {
                if ( dict->elements[i2]->type == DICT_INTEGERNUMBER ) {
                    if ( dict->elements[i2]->size == nb ) {
                        for ( j = 0; j < nb; j++ ) {
                            if ( (UWORD)(dict->elements[i2]->lhs[j]) != b[j] ) break;
                        }
                        if ( j == nb ) break;
                    }
                }
            }
            if ( i1 < 0 ) {
                if ( i2 < 0 ) goto NotFound;
                else {  /* number/replacement[i2] */
                    LongToLine(a,na);
                    if ( na > 1 ) {
                        if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
                             || AC.OutputMode == CMODE ) {
                            if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *)".Q0/"); }
```

```
                            else if ( AO.DoubleFlag == 1 ) { AddToLine((UBYTE *).D0/"); }
                            else { AddToLine((UBYTE *)"/"); }
                        }
                    }
                    else AddToLine((UBYTE *)("/"));
                    TokenToLine((UBYTE *)(dict->elements[i2]->rhs));
                }
            }
            else if ( i2 < 0 ) { /* replacement[i1]/number */
                TokenToLine((UBYTE *)(dict->elements[i1]->rhs));
                AddToLine((UBYTE *)("/"));
                LongToLine((UWORD *)(b),nb);
                if ( nb > 1 ) {
                    if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
                         || AC.OutputMode == CMODE ) {
                        if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *).Q0"); }
                        else if ( AO.DoubleFlag == 1 ) { AddToLine((UBYTE *).D0"); }
                    }
                }
            }
            else { /* replacement[i1]/replacement[i2] */
                TokenToLine((UBYTE *)(dict->elements[i1]->rhs));
                AddToLine((UBYTE *)("/"));
                TokenToLine((UBYTE *)(dict->elements[i2]->rhs));
            }
            break;
        default:
            MesPrint("Illegal code in TransformRational: %d",AO.CurDictNumbers);
            Terminate(-1);
    }
    return;
NotFound:
    if ( na != 1 || a[1] != 1 ) {
    if ( AO.CurDictNumberWarning ) {
        MesPrint(">>>>>Could not translate coefficient with dictionary %s<<<<<<<",dict->name);
    } }
NoAction:
    RatToLine(a,na);
    return;
}

/*
    #] TransformRational:
    #[ IsMultiplySign:
*/

int IsMultiplySign(VOID)
{
    DICTIONARY *dict;
    int i;
    if ( AO.CurrentDictionary <= 0 ) return(0);
    dict = AO.Dictionaries[AO.CurrentDictionary-1];
    if ( dict->characters == 0 ) return(0);
    for ( i = dict->numelements-1; i >= 0; i-- ) {
        if ( ( dict->elements[i]->type == DICT_SPECIALCHARACTER )
            && ( dict->elements[i]->lhs[0] == (WORD)('*') ) ) return(i+1);
    }
    return(0);
}

/*
    #] IsMultiplySign:
    #[ IsExponentSign:
*/

int IsExponentSign(VOID)
{
    DICTIONARY *dict;
    int i;
    if ( AO.CurrentDictionary <= 0 ) return(0);
    dict = AO.Dictionaries[AO.CurrentDictionary-1];
    if ( dict->characters == 0 ) return(0);
    for ( i = dict->numelements-1; i >= 0; i-- ) {
        if ( ( dict->elements[i]->type == DICT_SPECIALCHARACTER )
            && ( dict->elements[i]->lhs[0] == (WORD)('^') ) ) return(i+1);
    }
    return(0);
}

/*
    #] IsExponentSign:
    #[ FindSymbol :
*/

UBYTE *FindSymbol(WORD num)
{
    if ( AO.CurrentDictionary > 0 ) {
```

```
        DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
        int i;
        if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
            for ( i = dict->numelements-1; i >= 0; i-- ) {
                if ( dict->elements[i]->type == DICT_SYMBOL &&
                     dict->elements[i]->lhs[0] == num )
                        return((UBYTE *)(dict->elements[i]->rhs));
            }
        }
    }
    return(VARNAME(symbols,num));
}

/*
    #] FindSymbol :
    #[ FindVector :
*/

UBYTE *FindVector(WORD num)
{
    if ( AO.CurrentDictionary > 0 ) {
        DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
        int i;
        if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
            for ( i = dict->numelements-1; i >= 0; i-- ) {
                if ( dict->elements[i]->type == DICT_VECTOR &&
                     dict->elements[i]->lhs[0] == num )
                        return((UBYTE *)(dict->elements[i]->rhs));
            }
        }
    }
    num -= AM.OffsetVector;
    return(VARNAME(vectors,num));
}

/*
    #] FindVector :
    #[ FindIndex :
*/

UBYTE *FindIndex(WORD num)
{
    if ( AO.CurrentDictionary > 0 ) {
        DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
        int i;
        if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
            for ( i = dict->numelements-1; i >= 0; i-- ) {
                if ( dict->elements[i]->type == DICT_INDEX &&
                     dict->elements[i]->lhs[0] == num )
                        return((UBYTE *)(dict->elements[i]->rhs));
            }
        }
    }
    num -= AM.OffsetIndex;
    return(VARNAME(indices,num));
}

/*
    #] FindIndex :
    #[ FindFunction :
*/

UBYTE *FindFunction(WORD num)
{
    if ( AO.CurrentDictionary > 0 ) {
        DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
        int i;
        if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
            for ( i = dict->numelements-1; i >= 0; i-- ) {
                if ( dict->elements[i]->type == DICT_FUNCTION &&
                     dict->elements[i]->lhs[0] == num )
                        return((UBYTE *)(dict->elements[i]->rhs));
            }
        }
    }
    num -= FUNCTION;
    return(VARNAME(functions,num));
}

/*
    #] FindFunction :
    #[ FindFunWithArgs :
*/

UBYTE *FindFunWithArgs(WORD *t)
{
    if ( AO.CurrentDictionary > 0 ) {
```

```
        DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
        int i, j;
        if ( dict->funwith > 0
                    && AO.CurDictFunWithArgs == DICT_DOFUNWITHARGS ) {
            for ( i = dict->numelements-1; i >= 0; i-- ) {
                if ( dict->elements[i]->type == DICT_FUNCTION_WITH_ARGUMENTS &&
                     (WORD)(dict->elements[i]->lhs[0]) == t[0] &&
                     (WORD)(dict->elements[i]->lhs[1]) == t[1] ) {
                    for ( j = 2; j < t[1]; j++ ) {
                        if ( (WORD)(dict->elements[i]->lhs[j]) != t[j] ) break;
                    }
                    if ( j >= t[1] ) return((UBYTE *)(dict->elements[i]->rhs));
                }
            }
        }
    }
    return(0);
}

/*
    #] FindFunWithArgs :
    #[ FindExtraSymbol :

    The extra symbol is constructed in the WorkSpace. This way we do not
    have to worry about Malloc and freeing the object later.
    The input value num is already the number of the extra symbol.
    We do NOT need num = MAXVARIABLES-num;
*/

UBYTE *FindExtraSymbol(WORD num)
{
    GETIDENTITY;
    UBYTE *out = (UBYTE *)(AT.WorkPointer);
    *out = 0;
    if ( AO.CurrentDictionary > 0 ) {
        DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
        int i;
        if ( dict->ranges > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
            for ( i = dict->numelements-1; i >= 0; i-- ) {
                if ( dict->elements[i]->type == DICT_RANGE
                     && num >= dict->elements[i]->lhs[0]
                     && num <= dict->elements[i]->lhs[1] ) {
/*
                        Now we have to translate the rhs
                        %#  gives the number
                        %@  gives the number as its position in the range
*/
                        UBYTE *r = (UBYTE *)(dict->elements[i]->rhs);
                        while ( *r ) {
                            if ( *r == (UBYTE)'%' && ( r[1] == (UBYTE)'#'
                            || r[1] == (UBYTE)'@' ) ) {
                                if ( r[1] == (UBYTE)'#' ) {
                                    out = NumCopy(num,out);
                                }
                                else {
                                    out = NumCopy(num-dict->elements[i]->lhs[0]+1,out);
                                }
                                r += 2;
                            }
                            else {
                                *out++ = *r++;
                            }
                        }
                        *out = 0;
                        return((UBYTE *)(AT.WorkPointer));
                }
            }
        }
    }

    out = StrCopy((UBYTE *)AC.extrasym,out);
    if ( AC.extrasymbols == 0 ) {
        out = NumCopy(num,out);
        out = StrCopy((UBYTE *)"_",out);
    }
    else if ( AC.extrasymbols == 1 ) {
        out = AddArrayIndex(num,out);
    }
    return((UBYTE *)(AT.WorkPointer));
}

/*
    #] FindExtraSymbol :
    #[ FindDictionary :
*/

int FindDictionary(UBYTE *name)
```

```
{
    int i;
    for ( i = 0; i < AO.NumDictionaries; i++ ) {
        if ( StrCmp(AO.Dictionaries[i]->name,name) == 0 )
            return(i+1);
    }
    return(0);
}

/*
    #] FindDictionary :
    #[ AddDictionary :
*/

int AddDictionary(UBYTE *name)
{
    DICTIONARY *dict;
/*
    First make space for the pointer in the list.
*/
    if ( AO.NumDictionaries >= AO.SizeDictionaries-1 ) {
        DICTIONARY **d;
        int i;
        if ( AO.SizeDictionaries <= 0 ) AO.SizeDictionaries = 10;
        else AO.SizeDictionaries = 2*AO.SizeDictionaries;
        d = (DICTIONARY **)Malloc1(AO.SizeDictionaries*sizeof(DICTIONARY *),"Dictionaries");
        for ( i = 0; i < AO.NumDictionaries; i++ ) d[i] = AO.Dictionaries[i];
        if ( AO.Dictionaries != 0 ) M_free(AO.Dictionaries,"Dictionaries");
        AO.Dictionaries = d;
    }
/*
    Now create an empty dictionary.
*/
    dict = (DICTIONARY *)Malloc1(sizeof(DICTIONARY),"Dictionary");
    AO.Dictionaries[AO.NumDictionaries++] = dict;
    dict->elements = 0;
    dict->name = strDup1(name,"DictionaryName");
    dict->sizeelements = 0;
    dict->numelements = 0;
    dict->numbers = 0;
    dict->variables = 0;
    dict->characters = 0;
    dict->funwith = 0;
    dict->gnumelements = 0;
    dict->ranges = 0;

    return(AO.NumDictionaries);
}

/*
    #] AddDictionary :
    #[ AddToDictionary :

        To be called from #add left:right
*/

int AddToDictionary(DICTIONARY *dict,UBYTE *left,UBYTE *right)
{
    GETIDENTITY
    CBUF *C = cbuf+AC.cbufnum;
    WORD *w = AT.WorkPointer;
    WORD *OldWork = AT.WorkPointer;
    WORD *s, oldnumrhs = C->numrhs, oldnumlhs = C->numlhs;
    WORD *ow, *ww, *mm, oldEside, *where = 0, type, number, range[3];
    LONG oldcpointer;
    int error = 0, sizelhs, sizerhs, i, retcode;
    UBYTE *r;
    DICTIONARY_ELEMENT *new;
    WORD power = (WORD)('^'), times = (WORD)('*');
    if ( ( left[0] == '^' && left[1] == 0 )
      || ( left[0] == '*' && left[1] == '*' && left[2] == 0 ) ) {
        type = DICT_SPECIALCHARACTER;
        number = 1;
        where = &power;
        goto TestDouble;
    }
    else if ( left[0] == '*' && left[1] == 0 ) {
        type = DICT_SPECIALCHARACTER;
        number = 1;
        where = &times;
        goto TestDouble;
    }
    else if ( left[0] == '(' ) {  /* range of extra symbols */
        WORD x1 = 0, x2 = 0;
        r = left+1;
        while ( FG.cTable[*r] == 1 ) x1 = 10*x1 + *r++ - '0';
        if ( *r == ',' ) {
```

```
            r++;
            while ( FG.cTable[*r] == 1 ) x2 = 10*x2 + *r++ - '0';
        }
        else x2 = x1;
        number = 2;
        if ( *r != ')' ) {
            MesPrint("&Illegal range specification in LHS of %#add instruction.");
            return(1);
        }
        type = DICT_RANGE;
        if ( x1 <= 0 || x2 <= 0 || x1 > x2 ) {
            MesPrint("&Illegal range in LHS of %#add instruction.");
            return(1);
        }
        range[0] = x1;
        range[1] = x2;
        range[2] = 0;
        where = range;
        goto TestDouble;
    }
/*
    Translate the left part. Determine type.
    We follow the code in CoIdExpression and then veto what we do not like.
    Just make sure to pop what needs to be popped in the compiler buffer.
*/
    AC.ProtoType = w;
    *w++ = SUBEXPRESSION;
    *w++ = SUBEXPSIZE;
    *w++ = C->numrhs+1;
    *w++ = 1;
    *w++ = AC.cbufnum;
    FILLSUB(w)
    AC.WildC = w;
    AC.NwildC = 0;
    AT.WorkPointer = s = w + 4*AM.MaxWildcards + 8;
/*
    Now read the LHS
*/
    oldcpointer = AddLHS(AC.cbufnum) - C->Buffer;

    if ( ( retcode = CompileAlgebra(left,LHSIDE,AC.ProtoType) ) < 0 ) { error = 1; }
    else AC.ProtoType[2] = retcode;
    AT.WorkPointer = s;
    if ( AC.NwildC && SortWild(w,AC.NwildC) ) error = 1;

    OldWork[1] = AC.WildC-OldWork;
    w = AC.WildC;
    AT.WorkPointer = w;
    s = C->rhs[C->numrhs];
/*
    We have the expression in the compiler buffers.
    The main level is at lhs[numlhs]
    The partial lhs (including ProtoType) is in OldWork (in WorkSpace)
    We need to load the result at w after the prototype
    Because these sort routines don't use the WorkSpace
    there should not be a conflict
*/
    if ( !error && *s == 0 ) {
IllLeft:MesPrint("&Illegal LHS in dictionary");
        AC.lhdollarflag = 0;
        return(1);
    }
    if ( !error && *(s+*s) != 0 ) {
        MesPrint("&LHS in dictionary should be one term only");
        return(1);
    }
    if ( error == 0 ) {
        if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) {
            if ( !error ) error = 1;
            return(error);
        }
        AN.RepPoint = AT.RepCount + 1;
        ow = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
        mm = s; ww = ow; i = *mm;
        while ( --i >= 0 ) *ww++ = *mm++; AT.WorkPointer = ww;
        AC.lhdollarflag = 0; oldEside = AR.Eside; AR.Eside = LHSIDE;
        AR.Cnumlhs = C->numlhs;
        if ( Generator(BHEAD ow,C->numlhs) ) {
            AR.Eside = oldEside;
            LowerSortLevel(); LowerSortLevel(); goto IllLeft;
        }
        AR.Eside = oldEside;
        AT.WorkPointer = w;
        if ( EndSort(BHEAD w,0) < 0 ) { LowerSortLevel(); goto IllLeft; }
        if ( *w == 0 || *(w+*w) != 0 ) {
            MesPrint("&LHS must be one term");
            AC.lhdollarflag = 0;
```

```
            return(1);
        }
        LowerSortLevel();
    }
    AT.WorkPointer = w + *w;
    AC.DumNum = 0;
/*
    Everything is now after OldWork. We can pop the compilerbuffer.
    Next test for illegal things like a coefficient
    At this point we have:
    w = the term of the LHS
*/
    C->Pointer = C->Buffer + oldcpointer;
    C->numrhs = oldnumrhs;
    C->numlhs = oldnumlhs;
    AC.lhdollarflag = 0;
/*
    Test for undesirables.
        1: wildcards
        2: sign
        3: more than one term
        4: composite terms
*/
    if ( AC.ProtoType[1] != SUBEXPSIZE ) {
        MesPrint("& Currently no wildcards allowed in dictionaries.");
        return(1);
    }
    if ( w[w[0]-1] < 0 ) {
        MesPrint("& Currently no sign allowed in dictionaries.");
        return(1);
    }
    if ( w[w[0]] != 0 ) {
        MesPrint("& More than one term in dictionary element.");
        return(1);
    }
    if ( w[0] == w[w[0]-1]+1 ) {    /* Only coefficient */
        WORD *numer, *denom;
        WORD nsize, dsize;
        nsize = dsize = (w[w[0]-1]-1)/2;
        numer = w+1;
        denom = numer+nsize;
        while ( numer[nsize-1] == 0 ) nsize--;
        while ( denom[dsize-1] == 0 ) dsize--;
        if ( dsize == 1 && denom[0] == 1 ) {
            type = DICT_INTEGERNUMBER;
            number = nsize;
            where = numer;
        }
        else {
            type = DICT_RATIONALNUMBER;
            number = w[0];
            where = w;
        }
    }
    else {
        s = w + w[0]-1;
        if ( s[0] != 3 || s[-1] != 1 || s[-2] != 1 ) {
Compositeness:;
            MesPrint("& Currently no composite objects allowed in dictionaries.");
            return(1);
        }
        if ( w[0] != w[2]+4 ) goto Compositeness;
        s = w+1;
        switch ( *s ) {
            case SYMBOL:
                if ( s[1] != 4 || s[3] != 1 ) goto Compositeness;
                type = DICT_SYMBOL;
                number = 1;
                where = s+2;
                break;
            case INDEX:
                if ( s[1] != 3 ) goto Compositeness;
                if ( s[2] < 0 ) type = DICT_VECTOR;
                else            type = DICT_INDEX;
                number = 1;
                where = s+2;
                break;
            default:
                if ( *s < FUNCTION ) {
                    MesPrint("& Illegal object in dictionary.");
                    return(1);
                }
                if ( s[1] == FUNHEAD ) {
                    type = DICT_FUNCTION;
                    number = 1;
                    where = s;
                    break;
```

```
                }
                else {
                    type = DICT_FUNCTION_WITH_ARGUMENTS;
                    number = s[1];
                    where = s;
                }
                break;
        }
    }
TestDouble:;
/*
    Create a new element
*/
    if ( dict->numelements >= dict->sizeelements ) {
        DICTIONARY_ELEMENT **d;
        if ( dict->sizeelements <= 0 ) dict->sizeelements = 10;
        else                           dict->sizeelements *= 2;
        d = (DICTIONARY_ELEMENT **)Malloc1(
            sizeof(DICTIONARY_ELEMENT *)*dict->sizeelements,"Dictionary elements");
        for ( i = 0; i < dict->numelements; i++ )
            d[i] = dict->elements[i];
        if ( dict->elements ) M_free(dict->elements,"Dictionary elements");
        dict->elements = d;
    }
    sizelhs = number+1;
    sizerhs = 1; r = right; while ( *r++ ) sizerhs++;
    sizerhs = (sizerhs+sizeof(WORD)-1)/sizeof(WORD)+1;
    new = (DICTIONARY_ELEMENT *)Malloc1(sizeof(DICTIONARY_ELEMENT)
                +sizeof(WORD)*(sizelhs+sizerhs),"Dictionary element");
    new->lhs = (WORD *)(new+1);
    new->rhs = new->lhs+sizelhs;
    new->type = type;
    new->size = number;
    for ( i = 0; i < number; i++ ) new->lhs[i] = where[i];
    new->lhs[i] = 0;
    r = (UBYTE *)(new->rhs);
    while ( *right ) {
        if ( *right == '\\' && ( right[1] == '`' || right[1] == '\" ) ) right++;
        *r++ = *right++;
    }
    *r = 0;

    dict->elements[dict->numelements++] = new;

    switch ( type ) {
        case DICT_INTEGERNUMBER:
        case DICT_RATIONALNUMBER:
            dict->numbers++; break;
        case DICT_SYMBOL:
        case DICT_VECTOR:
        case DICT_INDEX:
        case DICT_FUNCTION:
            dict->variables++; break;
        case DICT_FUNCTION_WITH_ARGUMENTS:
            dict->funwith++; break;
        case DICT_SPECIALCHARACTER:
            dict->characters++; break;
        case DICT_RANGE:
            dict->ranges++; break;
    }

    AT.WorkPointer = OldWork;
    return(0);
}

/*
    #] AddToDictionary :
    #[ UseDictionary :
*/

int UseDictionary(UBYTE *name,UBYTE *options)
{
    int i;
    for ( i = 0; i < AO.NumDictionaries; i++ ) {
        if ( StrCmp(AO.Dictionaries[i]->name,name) == 0 ) {
            AO.CurrentDictionary = i+1;
            if ( SetDictionaryOptions(options) < 0 ) {
                AO.CurrentDictionary = 0;
                return(-1);
            }
            else {  /* Now test whether what is requested is really there? */
                return(0);
            }
        }
    }
    MesPrint("@There is no dictionary with the name %s",name);
    exit(-1);
```

```
}

/*
    #] UseDictionary :
    #[ SetDictionaryOptions :
*/

int SetDictionaryOptions(UBYTE *options)
{
    UBYTE *opt, *s, c;
    int retval = 0;
    s = options;
    AO.CurDictNumbers = DICT_ALLNUMBERS;
    AO.CurDictVariables = DICT_DOVARIABLES;
    AO.CurDictSpecials = DICT_DOSPECIALS;
    AO.CurDictFunWithArgs = DICT_DOFUNWITHARGS;
    AO.CurDictNumberWarning = 0;
    AO.CurDictNotInFunctions= 0;
    AO.CurDictInDollars = DICT_NOTINDOLLARS;
    while ( *s ) {
        opt = s;
        while ( *s && *s != ',' && *s != ' ' ) s++;
        c = *s; *s = 0;
        if ( opt[0] == '$' && opt[1] == 0 ) {
            AO.CurDictInDollars = DICT_INDOLLARS;
        }
        else if ( StrICmp(opt,(UBYTE *)"nonumbers") == 0 ) {
            AO.CurDictNumbers = DICT_NONUMBERS;
        }
        else if ( StrICmp(opt,(UBYTE *)"integersonly") == 0 ) {
            AO.CurDictNumbers = DICT_INTEGERONLY;
        }
        else if ( StrICmp(opt,(UBYTE *)"rationalsonly") == 0 ) {
            AO.CurDictNumbers = DICT_RATIONALONLY;
        }
        else if ( StrICmp(opt,(UBYTE *)"allnumbers") == 0 ) {
            AO.CurDictNumbers = DICT_ALLNUMBERS;
        }
        else if ( StrICmp(opt,(UBYTE *)"novariables") == 0 ) {
            AO.CurDictVariables = DICT_NOVARIABLES;
        }
        else if ( StrICmp(opt,(UBYTE *)"numbersonly") == 0 ) {
            AO.CurDictNumbers = DICT_ALLNUMBERS;
            AO.CurDictVariables = DICT_NOVARIABLES;
            AO.CurDictSpecials = DICT_NOSPECIALS;
            AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
        }
        else if ( StrICmp(opt,(UBYTE *)"variablesonly") == 0 ) {
            AO.CurDictNumbers = DICT_NONUMBERS;
            AO.CurDictVariables = DICT_DOVARIABLES;
            AO.CurDictSpecials = DICT_NOSPECIALS;
            AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
        }
        else if ( StrICmp(opt,(UBYTE *)"nospecials") == 0 ) {
            AO.CurDictSpecials = DICT_NOSPECIALS;
        }
        else if ( StrICmp(opt,(UBYTE *)"specialsonly") == 0 ) {
            AO.CurDictNumbers = DICT_NONUMBERS;
            AO.CurDictVariables = DICT_NOVARIABLES;
            AO.CurDictSpecials = DICT_DOSPECIALS;
            AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
        }
        else if ( StrICmp(opt,(UBYTE *)"nofunwithargs") == 0 ) {
            AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
        }
        else if ( StrICmp(opt,(UBYTE *)"funwithargsonly") == 0 ) {
            AO.CurDictNumbers = DICT_NONUMBERS;
            AO.CurDictVariables = DICT_NOVARIABLES;
            AO.CurDictSpecials = DICT_NOSPECIALS;
            AO.CurDictFunWithArgs = DICT_DOFUNWITHARGS;
        }
        else if ( StrICmp(opt,(UBYTE *)"warnings") == 0
                || StrICmp(opt,(UBYTE *)"warning") == 0 ) {
            AO.CurDictNumberWarning = 1;
        }
        else if ( StrICmp(opt,(UBYTE *)"nowarnings") == 0
                || StrICmp(opt,(UBYTE *)"nowarning") == 0 ) {
            AO.CurDictNumberWarning = 0;
        }
        else if ( StrICmp(opt,(UBYTE *)"infunctions") == 0 ) {
            AO.CurDictNotInFunctions= 0;
        }
        else if ( StrICmp(opt,(UBYTE *)"notinfunctions") == 0 ) {
            AO.CurDictNotInFunctions= 1;
        }
        else {
            MesPrint("@ Unrecognized option in %#SetDictionary: %s",opt);
```

```
            retval = -1;
        }
        *s = c;
        if ( c == ',' ) s++;
    }
    return(retval);
}

/*
    #] SetDictionaryOptions :
    #[ UnSetDictionary :
*/

void UnSetDictionary(VOID)
{
    AO.CurrentDictionary = 0;
    AO.CurDictNumbers = -1;
    AO.CurDictVariables = -1;
    AO.CurDictSpecials = -1;
    AO.CurDictFunWithArgs = -1;
    AO.CurDictFunWithArgs = -1;
    AO.CurDictNumberWarning = -1;
    AO.CurDictNotInFunctions= -1;
}

/*
    #] UnSetDictionary :
    #[ RemoveDictionary :

    Mostly needed for .clear
*/

void RemoveDictionary(DICTIONARY *dict)
{
    int i;
    if ( dict == 0 ) return;
    for ( i = 0; i < AO.NumDictionaries; i++ ) {
        if ( AO.Dictionaries[i] == dict ) {
            for (i++; i < AO.NumDictionaries; i++ ) {
                AO.Dictionaries[i-1] = AO.Dictionaries[i];
            }
            AO.NumDictionaries--;
            goto removeit;
        }
    }
    MesPrint("@ Dictionary not found in RemoveDictionary");
    exit(-1);
removeit:;
    for ( i = 0; i < dict->numelements; i++ )
            M_free(dict->elements[i],"Dictionary element");
    for ( i = 0; i < dict->numelements; i++ ) dict->elements[i] = 0;
    if ( dict->elements ) M_free(dict->elements,"Dictionary elements");
    if ( dict->name ) {
        M_free(dict->name,"DictionaryName");
        dict->name = 0;
    }
    dict->sizeelements = 0;
    dict->numelements = 0;
    dict->numbers = 0;
    dict->variables = 0;
    dict->characters = 0;
    dict->funwith = 0;
    dict->gnumelements = 0;
    dict->ranges = 0;
}

/*
    #] RemoveDictionary :
    #[ ShrinkDictionary :

    To be called after a .store to restore the dictionary to the state
    it had at the last .global
    We do not make the elements array shorter.
*/

void ShrinkDictionary(DICTIONARY *dict)
{
    while ( dict->numelements > dict->gnumelements ) {
        dict->numelements--;
        M_free(dict->elements[dict->numelements],"Dictionary element");
        dict->elements[dict->numelements] = 0;
    }
}

/*
    #] ShrinkDictionary :
    #[ DoPreOpenDictionary :
```

```c
*/

int DoPreOpenDictionary(UBYTE *s)
{
    UBYTE *name;
    int dict;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' ) s++;

    name = s; s = SkipAName(s);
    if ( *s != 0 && *s != ';' ) {
        MesPrint("@proper syntax is #opendictionary name");
        return(-1);
    }
    *s = 0;

    if ( AP.OpenDictionary > 0 ) {
        MesPrint("@you cannot nest #opendictionary instructions");
        MesPrint("@dictionary %s is open already",
                AO.Dictionaries[AP.OpenDictionary-1]->name);
        return(-1);
    }
    if ( AO.CurrentDictionary > 0 ) {
        MesPrint("@before opening a dictionary you have to first close the selected dictionary");
        return(-1);
    }
/*
    Do we have this dictionary already?
*/
    dict = FindDictionary(name);
    if ( dict == 0 ) dict = AddDictionary(name);
    AP.OpenDictionary = dict;
    return(0);
}

/*
    #] DoPreOpenDictionary :
    #[ DoPreCloseDictionary :
*/

int DoPreCloseDictionary(UBYTE *s)
{
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' ) s++;

    if ( AP.OpenDictionary == 0 && AO.CurrentDictionary == 0 ) {
        MesPrint("@you have neither an open, nor a selected dictionary");
        return(-1);
    }

    AP.OpenDictionary = 0;
    AO.CurrentDictionary = 0;

    AO.CurDictNotInFunctions = 0;

    return(0);
}

/*
    #] DoPreCloseDictionary :
    #[ DoPreUseDictionary :
*/

int DoPreUseDictionary(UBYTE *s)
{
    UBYTE *options, c, *ss, *sss, *name;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' ) s++;

    if ( AP.OpenDictionary > 0 ) {
        MesPrint("@before selecting a dictionary you have to first close the open dictionary");
        return(-1);
    }

    name = s; s = SkipAName(s);
    ss = s; while ( *s && *s != '(' ) s++;
    c = *ss; *ss = 0;
    if ( c == 0 ) {
        options = ss;
    }
    else {
        options = s+1; SKIPBRA3(s)
        if ( *s != ')' ) {
            MesPrint("@Irregular end of %#UseDictionary instruction");
```

```
            return(-1);
        }
        sss = s;
        s++; while ( *s == ' ' || *s == '\t' || *s == ';' ) s++;
        *sss = 0;
        if ( *s ) {
            MesPrint("@Irregular end of %#UseDictionary instruction");
            return(-1);
        }
    }
    return(UseDictionary(name,options));
}

/*
    #] DoPreUseDictionary :
    #[ DoPreAdd :

    Syntax:
        #add left :right
        #add left : "right"
    Adds to the currently open dictionary
*/

int DoPreAdd(UBYTE *s)
{
    UBYTE *left, *right;

    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' ) s++;

    if ( AP.OpenDictionary == 0 ) {
        MesPrint("@there is no open dictionary to add to");
        return(-1);
    }
/*
    Scan to the : and mark the left and right parts.
*/
    left = s;
    while ( *s && *s != ':' ) {
        if ( *s == '[' ) { SKIPBRA1(s) s++; }
        else if ( *s == '{' ) { SKIPBRA2(s) s++; }
        else if ( *s == '(' ) { SKIPBRA3(s) s++; }
        else if ( *s == ']' || *s == '}' || *s == ')' ) {
            MesPrint("@unmatched brackets in #add instruction");
            return(-1);
        }
        else s++;
    }
    if ( *s == 0 ) {
        MesPrint("@Missing : in #add instruction");
        return(-1);
    }
    *s++ = 0;
    right = s;
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == '"' && s[1] ) {
        right = s+1;
        s = s+2;
        while ( *s ) s++;
        while ( s[-1] != '"' ) s--;
        if ( s <= right ) {
            MesPrint("@Irregular use of double quotes in #add instruction");
            return(-1);
        }
        s[-1] = 0;
    }
    return(AddToDictionary(AO.Dictionaries[AP.OpenDictionary-1],left,right));
}

/*
    #] DoPreAdd :
    #[ DictToBytes :
*/

LONG DictToBytes(DICTIONARY *dict,UBYTE *buf)
{
    int numelements = dict->numelements, sizeelement, i, j, x;
    UBYTE *s1, *s2 = buf;
    DICTIONARY_ELEMENT *e;
/*
    First copy the struct
*/
    s1 = (UBYTE *)dict; j = sizeof(DICTIONARY);
    NCOPY(s2,s1,j)
/*
    Now the elements. Put a size indicator in front of each of them.
```

```
*/
    for ( i = 0; i < numelements; i++ ) {
        e = dict->elements[i];
        sizeelement = sizeof(DICTIONARY_ELEMENT)+(e->size+1)*sizeof(WORD);
        s1 = (UBYTE *)e->rhs; x = 0;
        while ( *s1 ) { s1++; x++; }
        x /= sizeof(WORD);
        sizeelement += (x+1) * sizeof(WORD);
        s1 = (UBYTE *)(&sizeelement); j = sizeof(WORD); NCOPY(s2,s1,j)
        s1 = (UBYTE *)e; j = sizeof(DICTIONARY_ELEMENT); NCOPY(s2,s1,j)
        s1 = (UBYTE *)e->lhs; j = (e->size+1)*(sizeof(WORD)); NCOPY(s2,s1,j)
        s1 = (UBYTE *)e->rhs; j = (x+1)*(sizeof(WORD)); NCOPY(s2,s1,j)
    }
    return(s2-buf);
}

/*
    #] DictToBytes :
    #[ DictFromBytes :
*/

DICTIONARY *DictFromBytes(UBYTE *buf)
{
    DICTIONARY *dict = Malloc1(sizeof(DICTIONARY),"Dictionary");
    UBYTE *s1, *s2;
    int i, j, sizeelement;
    DICTIONARY_ELEMENT *e;
/*
    First read the dictionary itself
*/
    s1 = buf;
    s2 = (UBYTE *)dict; j = sizeof(DICTIONARY); NCOPY(s2,s1,j)
/*
    Allocate the elements array:
*/
    dict->elements = (DICTIONARY_ELEMENT **)Malloc1(
            sizeof(DICTIONARY_ELEMENT *)*dict->sizeelements,"dictionary elements");
    for ( i = 0; i < dict->numelements; i++ ) {
        s2 = (UBYTE *)(&sizeelement); j = sizeof(WORD); NCOPY(s2,s1,j)
        e = (DICTIONARY_ELEMENT *)Malloc1(sizeelement*sizeof(UBYTE),"dictionary element");
        dict->elements[i] = e;
        j = sizeelement; s2 = (UBYTE *)e; NCOPY(s2,s1,j)
        e->lhs = (WORD *)(e+1);
        e->rhs = e->lhs + e->size+1;
    }
    return(dict);
}

/*
    #] DictFromBytes :
*/
```

4.23 dollar.c File Reference

`#include "form3.h"`

Include dependency graph for dollar.c:

Macros

- #define STEP2

Functions

- int CatchDollar (int par)
- int AssignDollar (PHEAD WORD *term, WORD level)
- UBYTE * WriteDollarToBuffer (WORD numdollar, WORD par)
- UBYTE * WriteDollarFactorToBuffer (WORD numdollar, WORD numfac, WORD par)
- void AddToDollarBuffer (UBYTE *s)
- void TermAssign (WORD *term)
- void WildDollars (PHEAD WORD *term)
- WORD DolToTensor (PHEAD WORD numdollar)
- WORD DolToFunction (PHEAD WORD numdollar)
- WORD DolToVector (PHEAD WORD numdollar)
- WORD DolToNumber (PHEAD WORD numdollar)
- WORD DolToSymbol (PHEAD WORD numdollar)
- WORD DolToIndex (PHEAD WORD numdollar)
- DOLLARS DolToTerms (PHEAD WORD numdollar)
- LONG DolToLong (PHEAD WORD numdollar)
- int ExecInside (UBYTE *s)
- int InsideDollar (PHEAD WORD *ll, WORD level)
- void ExchangeDollars (int num1, int num2)
- LONG TermsInDollar (WORD num)
- UBYTE * PreIfDollarEval (UBYTE *s, int *value)
- WORD * TranslateExpression (UBYTE *s)
- int IsSetMember (WORD *buffer, WORD numset)
- int IsMultipleOf (WORD *buf1, WORD *buf2)
- int TwoExprCompare (WORD *buf1, WORD *buf2, int oprtr)
- int DollarRaiseLow (UBYTE *name, LONG value)
- WORD EvalDoLoopArg (PHEAD WORD *arg, WORD par)
- WORD TestDoLoop (PHEAD WORD *lhsbuf, WORD level)
- WORD TestEndDoLoop (PHEAD WORD *lhsbuf, WORD level)
- int DollarFactorize (PHEAD WORD numdollar)
- void CleanDollarFactors (DOLLARS d)
- WORD * TakeDollarContent (PHEAD WORD *dollarbuffer, WORD **factor)
- WORD * MakeDollarInteger (PHEAD WORD *bufin, WORD **bufout)
- WORD * MakeDollarMod (PHEAD WORD *buffer, WORD **bufout)
- int GetDolNum (PHEAD WORD *t, WORD *tstop)
- void AddPotModdollar (WORD numdollar)

4.23.1 Detailed Description

The routines that deal with the dollar variables. The name administration is to be found in the file names.c

Definition in file dollar.c.

4.23.2 Macro Definition Documentation

4.23.2.1 STEP2

```
#define STEP2
```

Factors a dollar expression. Notation: d->nfactors becomes nonzero. if the number of factors is one, we leave d->factors zero. Otherwise factors is an array of pointers to the factors. These are pointers of the type FACDOLLAR. fd->where pointer to contents in term notation fd->size size of the buffer fd->where points to fd->type DOLNUMBER or DOLTERMS fd->value value if type is DOLNUMBER and it fits in a WORD.

Definition at line 2859 of file dollar.c.

4.23.3 Function Documentation

4.23.3.1 CatchDollar()

```
int CatchDollar (
            int par )
```

Definition at line 54 of file dollar.c.

4.23.3.2 AssignDollar()

```
int AssignDollar (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 209 of file dollar.c.

4.23.3.3 WriteDollarToBuffer()

```
UBYTE * WriteDollarToBuffer (
            WORD numdollar,
            WORD par )
```

Definition at line 596 of file dollar.c.

4.23.3.4 WriteDollarFactorToBuffer()

```
UBYTE * WriteDollarFactorToBuffer (
            WORD numdollar,
            WORD numfac,
            WORD par )
```

Definition at line 687 of file dollar.c.

4.23.3.5 AddToDollarBuffer()

```
void AddToDollarBuffer (
            UBYTE * s )
```

Definition at line 759 of file dollar.c.

4.23.3.6 TermAssign()

```
void TermAssign (
            WORD * term )
```

Definition at line 800 of file dollar.c.

4.23.3.7 WildDollars()

```
void WildDollars (
            PHEAD WORD * term )
```

Definition at line 862 of file dollar.c.

4.23.3.8 DolToTensor()

```
WORD DolToTensor (
            PHEAD WORD numdollar )
```

Definition at line 1055 of file dollar.c.

4.23.3.9 DolToFunction()

```
WORD DolToFunction (
            PHEAD WORD numdollar )
```

Definition at line 1116 of file dollar.c.

4.23.3.10 DolToVector()

```
WORD DolToVector (
            PHEAD WORD numdollar )
```

Definition at line 1173 of file dollar.c.

4.23.3.11 DolToNumber()

```
WORD DolToNumber (
            PHEAD WORD numdollar )
```

Definition at line 1237 of file dollar.c.

4.23.3.12 DolToSymbol()

```
WORD DolToSymbol (
            PHEAD WORD numdollar )
```

Definition at line 1296 of file dollar.c.

4.23.3.13 DolToIndex()

```
WORD DolToIndex (
            PHEAD WORD numdollar )
```

Definition at line 1350 of file dollar.c.

4.23.3.14 DolToTerms()

```
DOLLARS DolToTerms (
            PHEAD WORD numdollar )
```

Definition at line 1427 of file dollar.c.

4.23.3.15 DolToLong()

```
LONG DolToLong (
            PHEAD WORD numdollar )
```

Definition at line 1573 of file dollar.c.

4.23.3.16 ExecInside()

```
int ExecInside (
            UBYTE * s )
```

Definition at line 1648 of file dollar.c.

4.23.3.17 InsideDollar()

```
int InsideDollar (
            PHEAD WORD * ll,
            WORD level )
```

Definition at line 1713 of file dollar.c.

4.23.3.18 ExchangeDollars()

```
void ExchangeDollars (
            int num1,
            int num2 )
```

Definition at line 1807 of file dollar.c.

4.23.3.19 TermsInDollar()

```
LONG TermsInDollar (
            WORD num )
```

Definition at line 1825 of file dollar.c.

4.23.3.20 PreIfDollarEval()

```
UBYTE * PreIfDollarEval (
            UBYTE * s,
            int * value )
```

Definition at line 1884 of file dollar.c.

4.23.3.21 TranslateExpression()

```
WORD * TranslateExpression (
            UBYTE * s )
```

Definition at line 2066 of file dollar.c.

4.23.3.22 IsSetMember()

```
int IsSetMember (
            WORD * buffer,
            WORD numset )
```

Definition at line 2123 of file dollar.c.

4.23.3.23 IsMultipleOf()

```
int IsMultipleOf (
            WORD * buf1,
            WORD * buf2 )
```

Definition at line 2304 of file dollar.c.

4.23.3.24 TwoExprCompare()

```
int TwoExprCompare (
            WORD * buf1,
            WORD * buf2,
            int oprtr )
```

Definition at line 2379 of file dollar.c.

4.23.3.25 DollarRaiseLow()

```
int DollarRaiseLow (
            UBYTE * name,
            LONG value )
```

Definition at line 2458 of file dollar.c.

4.23.3.26 EvalDoLoopArg()

```
WORD EvalDoLoopArg (
            PHEAD WORD * arg,
            WORD par )
```

Evaluates one argument of a do loop. Such an argument is constructed from SNUMBERs DOLLAREXPRESSIONs and possibly DOLLAREXPR2s which indicate factors of the preceeding dollar. Hence we have SNUMBER,num DOLLAREXPRESSION,numdollar DOLLAREXPRESSION,numdollar,DOLLAREXPR2,numfactor DOLLAREXPRESSION,numdollar,DOLLAREXPR2,numfactor,DOLLAREXPR2,numfactor etc. Because we have a do-loop at every stage we should have a number. The notation in DOLLAREXPR2 is that >= 0 is number of yet another dollar and < 0 is -n-1 with n the array element or zero. The return value is the (short) number. The routine works its way through the list in a recursive manner.

Definition at line 2557 of file dollar.c.

References EvalDoLoopArg().

Referenced by EvalDoLoopArg().

Here is the call graph for this function:

4.23.3.27 TestDoLoop()

```
WORD TestDoLoop (
            PHEAD WORD * lhsbuf,
            WORD level )
```

Definition at line 2668 of file dollar.c.

4.23.3.28 TestEndDoLoop()

```
WORD TestEndDoLoop (
            PHEAD WORD * lhsbuf,
            WORD level )
```

Definition at line 2746 of file dollar.c.

4.23.3.29 DollarFactorize()

```
int DollarFactorize (
            PHEAD WORD numdollar )
```

Definition at line 2861 of file dollar.c.

4.23.3.30 CleanDollarFactors()

```
void CleanDollarFactors (
            DOLLARS d )
```

Definition at line 3460 of file dollar.c.

4.23.3.31 TakeDollarContent()

```
WORD * TakeDollarContent (
            PHEAD WORD * dollarbuffer,
            WORD ** factor )
```

Definition at line 3481 of file dollar.c.

4.23.3.32 MakeDollarInteger()

```
WORD * MakeDollarInteger (
            PHEAD WORD * bufin,
            WORD ** bufout )
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in bufin. The number that comes out is the return value. The normalized argument is in bufout.

Definition at line 3533 of file dollar.c.

References EndSort(), Generator(), and NewSort().

Here is the call graph for this function:

4.23.3.33 MakeDollarMod()

```
WORD * MakeDollarMod (
            PHEAD WORD * buffer,
            WORD ** bufout )
```

Similar to MakeDollarInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3707 of file dollar.c.

References EndSort(), Generator(), GetModInverses(), and NewSort().

Here is the call graph for this function:

4.23.3.34 GetDolNum()

```
int GetDolNum (
            PHEAD WORD * t,
            WORD * tstop )
```

Definition at line 3752 of file dollar.c.

4.23.3.35 AddPotModdollar()

```
void AddPotModdollar (
            WORD numdollar )
```

Adds a $-variable specified by *numdollar* to the list of potentially modified $-variables unless it has already been included in the list.

Parameters

numdollar	The index of the $-variable to be added.

Definition at line 3865 of file dollar.c.

4.24 dollar.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes :
*/

#include "form3.h"

/* EXTERNLOCK(dummylock) */

static UBYTE underscore[2] = {'_',0};

/*
    #] Includes :
    #[ CatchDollar :

    Works out a dollar expression during compile type.
    Steals it from the buffer and puts it in an assignment.
    At the moment we should keep this inside the small buffer.
    Later with more sort buffers we can do this better.
    Par == 0 : regular assignment
    par == -1: after error. Just make zero for now.
*/

int CatchDollar(int par)
{
    GETIDENTITY
    CBUF *C = cbuf + AC.cbufnum;
    int error = 0, numterms = 0, numdollar, resetmods = 0;
    LONG newsize, retval;
    WORD *w, *t, n, nsize, *oldwork = AT.WorkPointer, *dbuffer;
    WORD oldncmod = AN.ncmod;
    DOLLARS d;
    if ( AN.ncmod && ( ( AC.modmode & ALSODOLLARS ) == 0 ) ) AN.ncmod = 0;
    if ( AN.ncmod && AN.cmod == 0 ) { SetMods(); resetmods = 1; }

    numdollar = C->lhs[C->numlhs][2];

    d = Dollars+numdollar;
    if ( par == -1 ) {
        d->type = DOLUNDEFINED;
        cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
        cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
        if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"$-buffer old");
        d->size = 0; d->where = &(AM.dollarzero);
        cbuf[AM.dbufnum].rhs[numdollar] = d->where;
        AN.ncmod = oldncmod;
        if ( resetmods ) UnSetMods();
        return(0);
    }
#ifdef WITHMPI
    /*
     * The problem here is that only the master can make an assignment
     * like #$a=g; where g is an expression: only the master has an access to
     * the expression. So, in cases where the RHS contains expression names,
     * only the master invokes Generator() and then broadcasts the result to
     * the all slaves.
```

```
     * Broadcasting must be performed immediately; one cannot postpone it
     * to the end of the module because the dollar variable is visible
     * in the current module. For the same reason, this should be done
     * regardless of on/off parallel status.
     * If the RHS does not contain any expression names, it can be processed
     * in each slave.
     */
    if ( PF.me == MASTER || !AC.RhsExprInModuleFlag ) {
#endif

    EXCHINOUT

    if ( NewSort(BHEAD0) ) { if ( !error ) error = 1; goto onerror; }
    if ( NewSort(BHEAD0) ) {
        LowerSortLevel();
        if ( !error ) error = 1;
        goto onerror;
    }
    AN.RepPoint = AT.RepCount + 1;
    w = C->rhs[C->lhs[C->numlhs][5]];
    while ( *w ) {
        n = *w; t = oldwork;
        NCOPY(t,w,n)
        AT.WorkPointer = t;
        AR.Cnumlhs = C->numlhs;
        if ( Generator(BHEAD oldwork,C->numlhs) ) { error = 1; break; }
    }
    AT.WorkPointer = oldwork;
    AN.tryterm = 0; /* for now */
    dbuffer = 0;
    if ( ( retval = EndSort(BHEAD (WORD *)((VOID *)(&dbuffer)),2) ) < 0 ) { error = 1; }
    LowerSortLevel();
    if ( retval <= 1 || dbuffer == 0 ) {
        d->type = DOLZERO;
        if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"$-buffer old");
        d->size = 0; d->where = &(AM.dollarzero);
        cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
        cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
        goto docopy2;
    }
    w = dbuffer;
    if ( error == 0 )
        while ( *w ) { w += *w; numterms++; }
    else
        goto onerror;
    newsize = (w-dbuffer)+1;
#ifdef WITHMPI
    }
    if ( AC.RhsExprInModuleFlag )
        /* PF_BroadcastPreDollar allocates dbuffer for slaves! */
        if ( (error = PF_BroadcastPreDollar(&dbuffer, &newsize, &numterms)) != 0 )
            goto onerror;
#endif
    if ( newsize < 32 ) newsize = 32;
    newsize = ((newsize+7)/8)*8;
    if ( numterms == 0 ) {
        d->type = DOLZERO;
        goto docopy;
    }
    else if ( numterms == 1 ) {
        t = dbuffer;
        n = *t;
        nsize = t[n-1];
        if ( nsize < 0 ) { nsize = -nsize; }
        if ( nsize == (n-1) ) { /* numerical */
            nsize = (nsize-1)/2;
            w = t + 1 + nsize;
            if ( *w != 1 ) goto doterms;
            w++; while ( w < ( t + n - 1 ) ) { if ( *w ) break; w++; }
            if ( w < ( t + n - 1 ) ) goto doterms;
            d->type = DOLNUMBER;
            goto docopy;
        }
        else if ( n == 7 && t[6] == 3 && t[5] == 1 && t[4] == 1
            && t[1] == INDEX && t[2] == 3 ) {
            d->type = DOLINDEX;
            d->index = t[3];
            goto docopy;
        }
        else goto doterms;
    }
    else {
doterms:;
        d->type = DOLTERMS;
        cbuf[AM.dbufnum].CanCommu[numdollar] = numcommute(dbuffer,
                &(cbuf[AM.dbufnum].NumTerms[numdollar]));
docopy:;
```

```
        if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"$-buffer old");
        d->size = newsize; d->where = dbuffer;
docopy2:;
        cbuf[AM.dbufnum].rhs[numdollar] = d->where;
    }
    if ( C->Pointer > C->rhs[C->numrhs] ) C->Pointer = C->rhs[C->numrhs];
    C->numlhs--; C->numrhs--;
onerror:
#ifdef WITHMPI
    if ( PF.me == MASTER || !AC.RhsExprInModuleFlag )
#endif
    BACKINOUT
    AN.ncmod = oldncmod;
    if ( resetmods ) UnSetMods();
    return(error);
}

/*
    #] CatchDollar :
    #[ AssignDollar :

    To be called from Generator. Assigns an expression to a $ variable.
    This one is slightly different from CatchDollar.
    We have no easy buffer this time.
    We will have to hack our way using what we normally use for functions.

    Note that in the threaded case we trust the user. That means that
    we are not going to recheck whether there is a maximum, minimum or sum.
    If the user says it is like that, we treat it like that.
    We only check that in this centralized version MODLOCAL isn't used.

    In a later stage dtype could be used for actually checking MODMAX
    and MODMIN cases.
*/

int AssignDollar(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    CBUF *C = cbuf+AM.rbufnum;
    int numterms = 0, numdollar = C->lhs[level][2];
    LONG newsize;
    DOLLARS d = Dollars + numdollar;
    WORD *w, *t, n, nsize, *rh = cbuf[C->lhs[level][7]].rhs[C->lhs[level][5]];
    WORD *ss, *ww;
    WORD olddefer, oldcompress, oldncmod = AN.ncmod;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1, dw;
    WORD numvalue;
    if ( AN.ncmod && ( ( AC.modmode & ALSODOLLARS ) == 0 ) ) AN.ncmod = 0;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
/*
        Here we come only when the module runs with more than one thread.
        This must be a variable with a special module option.
        For the multi-threaded version we only allow MODSUM, MODMAX and MODMIN.
*/
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt >= NumModOptdollars ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Illegal attempt to change $-variable in multi-threaded module %l",AC.CModule);
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        dtype = ModOptdollars[nummodopt].type;
        if ( dtype == MODLOCAL ) {
            d = ModOptdollars[nummodopt].dstruct+AT.identity;
        }
    }
#endif
    DUMMYUSE(term);
    w = rh;
/*
    First some shortcuts
*/
    if ( *w == 0 ) {
/*
        #[ Thread version : Zero case
*/
#ifdef WITHPTHREADS
        if ( dtype > 0 ) {
/*          LOCK(d->pthreadslockwrite); */
            LOCK(d->pthreadslockread);
NewValIsZero:;
            switch ( d->type ) {
                case DOLZERO: goto NoChangeZero;
                case DOLNUMBER:
```

```
                case DOLTERMS:
                    if ( ( dw = d->where[0] ) > 0 && d->where[dw] != 0 ) {
                        break; /* was not a single number. Trust the user */
                    }
                    if ( dtype == MODMAX && d->where[dw-1] >= 0 ) goto NoChangeZero;
                    if ( dtype == MODMIN && d->where[dw-1] <= 0 ) goto NoChangeZero;
                    break;
                default:
                    numvalue = DolToNumber(BHEAD numdollar);
                    if ( AN.ErrorInDollar != 0 ) break;
                    if ( dtype == MODMAX && numvalue >= 0 ) goto NoChangeZero;
                    if ( dtype == MODMIN && numvalue <= 0 ) goto NoChangeZero;
                    break;
            }
            d->type = DOLZERO;
            d->where[0] = 0;
            cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
            cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
NoChangeZero:;
            CleanDollarFactors(d);
/*          UNLOCK(d->pthreadslockwrite); */
            UNLOCK(d->pthreadslockread);
            AN.ncmod = oldncmod;
            return(0);
        }
#endif
/*
        #] Thread version :
*/
        d->type = DOLZERO;
        d->where[0] = 0;
        cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
        cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
        CleanDollarFactors(d);
        AN.ncmod = oldncmod;
        return(0);
    }
    else if ( *w == 4 && w[4] == 0 && w[2] == 1 ) {
/*
        #[ Thread version : New value is 'single precision'
*/
#ifdef WITHPTHREADS
        if ( dtype > 0 ) {
/*          LOCK(d->pthreadslockwrite); */
            LOCK(d->pthreadslockread);
            if ( d->size < 32 ) {
                WORD oldsize, *oldwhere, i;
                oldsize = d->size; oldwhere = d->where;
                d->size = 32;
                d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
                cbuf[AM.dbufnum].rhs[numdollar] = d->where;
                if ( oldsize > 0 ) {
                    for ( i = 0; i < oldsize; i++ ) d->where[i] = oldwhere[i];
                }
                else d->where[0] = 0;
                if ( oldwhere && oldwhere != &(AM.dollarzero) ) M_free(oldwhere,"dollar contents");
            }
            switch ( d->type ) {
                case DOLZERO:
HandleDolZero:;
                    if ( dtype == MODMAX && w[3] <= 0 ) goto NoChangeOne;
                    if ( dtype == MODMIN && w[3] >= 0 ) goto NoChangeOne;
                    break;
                case DOLNUMBER:
                case DOLTERMS:
                    if ( ( dw = d->where[0] ) > 0 && d->where[dw] != 0 ) {
                        break; /* was not a single number. Trust the user */
                    }
                    if ( dtype == MODMAX && CompCoef(d->where,w) >= 0 ) goto NoChangeOne;
                    if ( dtype == MODMIN && CompCoef(d->where,w) <= 0 ) goto NoChangeOne;
                    break;
                default:
                    {
/*
                        Note that we convert the type for the next time around.
*/
                        WORD extraterm[4];
                        numvalue = DolToNumber(BHEAD numdollar);
                        if ( AN.ErrorInDollar != 0 ) break;
                        if ( numvalue == 0 ) {
                            d->type = DOLZERO;
                            d->where[0] = 0;
                            cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
                            cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
                            goto HandleDolZero;
                        }
                        d->where[0] = extraterm[0] = 4;
```

```
                        d->where[1] = extraterm[1] = ABS(numvalue);
                        d->where[2] = extraterm[2] = 1;
                        d->where[3] = extraterm[3] = numvalue > 0 ? 3 : -3;
                        d->where[4] = 0;
                        d->type = DOLNUMBER;
                        if ( dtype == MODMAX && CompCoef(extraterm,w) >= 0 ) goto NoChangeOne;
                        if ( dtype == MODMIN && CompCoef(extraterm,w) <= 0 ) goto NoChangeOne;
                        break;
                    }
            }
            d->where[0] = w[0];
            d->where[1] = w[1];
            d->where[2] = w[2];
            d->where[3] = w[3];
            d->where[4] = 0;
            d->type = DOLNUMBER;
            cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
            cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
NoChangeOne:;
            CleanDollarFactors(d);
/*          UNLOCK(d->pthreadslockwrite); */
            UNLOCK(d->pthreadslockread);
            AN.ncmod = oldncmod;
            return(0);
        }
#endif
/*
        #] Thread version :
*/
        if ( d->size < 32 ) {
            if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
            d->size = 32;
            d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
            cbuf[AM.dbufnum].rhs[numdollar] = d->where;
        }
        d->where[0] = w[0];
        d->where[1] = w[1];
        d->where[2] = w[2];
        d->where[3] = w[3];
        d->where[4] = 0;
        d->type = DOLNUMBER;
        cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
        cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
        CleanDollarFactors(d);
        AN.ncmod = oldncmod;
        return(0);
    }
/*
    Now the real evaluation.
    In the case of threads and MODSUM this requires an immediate lock.
    Otherwise the lock could be placed later.
*/
#ifdef WITHPTHREADS
    if ( dtype == MODSUM ) {
/*      LOCK(d->pthreadslockwrite); */
        LOCK(d->pthreadslockread);
    }
#endif
    CleanDollarFactors(d);
/*
    The following case cannot occur. We treated it already

    if ( *w == 0 ) {
        ss = 0; numterms = 0; newsize = 0;
        olddefer = AR.DeferFlag; AR.DeferFlag = 0;
        oldcompress = AR.NoCompress; AR.NoCompress = 1;
    }
    else
*/
    {
/*
        New value is an expression that has to be evaluated first
        This is all generic. It won't foliate due to the sort level
*/
        if ( NewSort(BHEAD0) ) {
            AN.ncmod = oldncmod;
            return(1);
        }
        olddefer = AR.DeferFlag; AR.DeferFlag = 0;
        oldcompress = AR.NoCompress; AR.NoCompress = 1;
        while ( *w ) {
            n = *w; t = ww = AT.WorkPointer;
            NCOPY(t,w,n);
            AT.WorkPointer = t;
            if ( Generator(BHEAD ww,AR.Cnumlhs) ) {
                AT.WorkPointer = ww;
                LowerSortLevel();
```

```
                AR.DeferFlag = olddefer;
                AN.ncmod = oldncmod;
                return(1);
            }
            AT.WorkPointer = ww;
        }
        AN.tryterm = 0; /* for now */
        if ( ( newsize = EndSort(BHEAD (WORD *)((VOID *)(&ss)),2) ) < 0 ) {
            AN.ncmod = oldncmod;
            return(1);
        }
        numterms = 0; t = ss; while ( *t ) { numterms++; t += *t; }
    }
#ifdef WITHPTHREADS
    if ( dtype != MODSUM ) {
/*      LOCK(d->pthreadslockwrite); */
        LOCK(d->pthreadslockread);
    }
#endif
    if ( numterms == 0 ) {
/*
        the new value evaluates to zero
*/
#ifdef WITHPTHREADS
        if ( dtype == MODMAX || dtype == MODMIN ) {
            if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
            AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
            goto NewValIsZero;
        }
        else
#endif
        {
          if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
          d->where = &(AM.dollarzero);
          d->size = 0;
          cbuf[AM.dbufnum].rhs[numdollar] = 0;
          cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
          cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
          d->type = DOLZERO;
        }
        if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
    }
    else {
/*
        #[ Thread version :
*/
#ifdef WITHPTHREADS
        if ( dtype == MODMAX || dtype == MODMIN ) {
            if ( numterms == 1 && ( *ss-1 == ABS(ss[*ss-1]) ) ) { /* is number */
              switch ( d->type ) {
                case DOLZERO:
HandleDolZero1:;
                    if ( dtype == MODMAX && ss[*ss-1] > 0 ) break;
                    if ( dtype == MODMIN && ss[*ss-1] < 0 ) break;
                    if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
                    AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
                    goto NoChange;
                case DOLTERMS:
                case DOLNUMBER:
                    if ( ( dw = d->where[0] ) > 0 && d->where[dw] != 0 ) break;
                    if ( dtype == MODMAX && CompCoef(ss,d->where) > 0 ) break;
                    if ( dtype == MODMIN && CompCoef(ss,d->where) < 0 ) break;
                    if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
                    AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
                    goto NoChange;
                default: {
                    WORD extraterm[4];
                    numvalue = DolToNumber(BHEAD numdollar);
                    if ( AN.ErrorInDollar != 0 ) break;
                    if ( numvalue == 0 ) {
                        d->type = DOLZERO;
                        d->where[0] = 0;
                        cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
                        cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
                        goto HandleDolZero1;
                    }
                    d->where[0] = extraterm[0] = 4;
                    d->where[1] = extraterm[1] = ABS(numvalue);
                    d->where[2] = extraterm[2] = 1;
                    d->where[3] = extraterm[3] = numvalue > 0 ? 3 : -3;
                    d->where[4] = 0;
                    d->type = DOLNUMBER;
                    if ( dtype == MODMAX && CompCoef(ss,extraterm) > 0 ) break;
                    if ( dtype == MODMIN && CompCoef(ss,extraterm) < 0 ) break;
                    if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
                    AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
                    goto NoChange;
```

```
                }
              }
            }
            else {
                if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
                AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
                goto NoChange;
            }
        }
#endif
/*
        #] Thread version :
*/
        d->type = DOLTERMS;
        if ( d->where && d->where != &(AM.dollarzero) ) { M_free(d->where,"dollar contents"); d->where = 0; }
        d->size = newsize + 1;
        d->where = ss;
        cbuf[AM.dbufnum].rhs[numdollar] = w = d->where;
    }
    AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
/*
    Now find the special cases
*/
    if ( numterms == 0 ) {
        d->type = DOLZERO;
    }
    else if ( numterms == 1 ) {
        t = d->where;
        n = *t;
        nsize = t[n-1];
        if ( nsize < 0 ) { nsize = -nsize; }
        if ( nsize == (n-1) ) {
            nsize = (nsize-1)/2;
            w = t + 1 + nsize;
            if ( *w == 1 ) {
                w++; while ( w < ( t + n - 1 ) ) { if ( *w ) break; w++; }
                if ( w >= ( t + n - 1 ) ) d->type = DOLNUMBER;
            }
        }
        else if ( n == 7 && t[6] == 3 && t[5] == 1 && t[4] == 1
            && t[1] == INDEX && t[2] == 3 ) {
            d->type = DOLINDEX;
            d->index = t[3];
        }
    }
    if ( d->type == DOLTERMS ) {
        cbuf[AM.dbufnum].CanCommu[numdollar] = numcommute(d->where,
            &(cbuf[AM.dbufnum].NumTerms[numdollar]));
    }
    else {
        cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
        cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
    }
#ifdef WITHPTHREADS
NoChange:;
/*  UNLOCK(d->pthreadslockwrite); */
    UNLOCK(d->pthreadslockread);
#endif
    AN.ncmod = oldncmod;
    return(0);
}

/*
    #] AssignDollar :
    #[ WriteDollarToBuffer :

    Takes the numbered dollar expression and writes it to output.
    We catch however the output in a buffer and return its address.
    This routine is needed when we need a text representation of
    a dollar expression like for the construction `$name' in the preprocessor.
    If par==0 we leave the current printing mode.
    If par==1 we insist on normal mode
*/

UBYTE *WriteDollarToBuffer(WORD numdollar, WORD par)
{
    DOLLARS d = Dollars+numdollar;
    UBYTE *s, *oldcurbufwrt = AO.CurBufWrt;
    WORD *t, lbrac = 0, first = 0, arg[2], oldOutputMode = AC.OutputMode;
    WORD oldinfbrack = AO.InFbrack;
    int error = 0;
    int dict = AO.CurrentDictionary;

    AO.DollarOutSizeBuffer = 32;
    AO.DollarOutBuffer = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,"DollarOutBuffer");
    AO.DollarInOutBuffer = 1;
```

```
    AO.PrintType = 1;
    AO.InFbrack = 0;
    s = AO.DollarOutBuffer;
    *s = 0;
    if ( par > 0 && AO.CurDictInDollars == 0 ) {
        AC.OutputMode = NORMALFORMAT;
        AO.CurrentDictionary = 0;
    }
    else {
        AO.CurBufWrt = (UBYTE *)underscore;
    }
    AO.OutInBuffer = 1;
    switch ( d->type ) {
        case DOLARGUMENT:
            WriteArgument(d->where);
            break;
        case DOLSUBTERM:
            WriteSubTerm(d->where,1);
            break;
        case DOLNUMBER:
        case DOLTERMS:
            t = d->where;
            while ( *t ) {
                if ( WriteTerm(t,&lbrac,first,PRINTON,0) ) {
                    error = 1; break;
                }
                t += *t;
            }
            break;
        case DOLWILDARGS:
            t = d->where+1;
            while ( *t ) {
                WriteArgument(t);
                NEXTARG(t)
                if ( *t ) TokenToLine((UBYTE *)(","));
            }
            break;
        case DOLINDEX:
            arg[0] = -INDEX; arg[1] = d->index;
            WriteArgument(arg);
            break;
        case DOLZERO:
            *s++ = '0'; *s = 0;
            AO.DollarInOutBuffer = 1;
            break;
        case DOLUNDEFINED:
            *s = 0;
            AO.DollarInOutBuffer = 1;
            break;
    }
    AC.OutputMode = oldOutputMode;
    AO.OutInBuffer = 0;
    AO.InFbrack = oldinfbrack;
    AO.CurBufWrt = oldcurbufwrt;
    AO.CurrentDictionary = dict;
    if ( error ) {
        MLOCK(ErrorMessageLock);
        MesPrint("&Illegal dollar object for writing");
        MUNLOCK(ErrorMessageLock);
        M_free(AO.DollarOutBuffer,"DollarOutBuffer");
        AO.DollarOutBuffer = 0;
        AO.DollarOutSizeBuffer = 0;
        return(0);
    }
    return(AO.DollarOutBuffer);
}

/*
    #] WriteDollarToBuffer :
    #[ WriteDollarFactorToBuffer :

    Takes the numbered dollar expression and writes it to output.
    We catch however the output in a buffer and return its address.
    This routine is needed when we need a text representation of
    a dollar expression like for the construction `$name' in the preprocessor.
    If par==0 we leave the current printing mode.
    If par==1 we insist on normal mode
*/

UBYTE *WriteDollarFactorToBuffer(WORD numdollar, WORD numfac, WORD par)
{
    DOLLARS d = Dollars+numdollar;
    UBYTE *s, *oldcurbufwrt = AO.CurBufWrt;
    WORD *t, lbrac = 0, first = 0, n[5], oldOutputMode = AC.OutputMode;
    WORD oldinfbrack = AO.InFbrack;
    int error = 0;
    int dict = AO.CurrentDictionary;
```

```c

    if ( numfac > d->nfactors || numfac < 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("&Illegal factor number for this dollar variable: %d",numfac);
        MesPrint("&There are %d factors",d->nfactors);
        MUNLOCK(ErrorMessageLock);
        return(0);
    }

    AO.DollarOutSizeBuffer = 32;
    AO.DollarOutBuffer = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,"DollarOutBuffer");
    AO.DollarInOutBuffer = 1;
    AO.PrintType = 1;
    AO.InFbrack = 0;
    s = AO.DollarOutBuffer;
    *s = 0;
    if ( par > 0 ) {
        AC.OutputMode = NORMALFORMAT;
        AO.CurrentDictionary = 0;
    }
    else {
        AO.CurBufWrt = (UBYTE *)underscore;
    }
    AO.OutInBuffer = 1;
    if ( numfac == 0 ) {    /* write the number d->nfactors */
        n[0] = 4; n[1] = d->nfactors; n[2] = 1; n[3] = 3; n[4] = 0; t = n;
    }
    else if ( d->factors[numfac-1].where == 0 ) {   /* write the value */
        if ( d->factors[numfac-1].value < 0 ) {
            n[0] = 4; n[1] = -d->factors[numfac-1].value; n[2] = 1; n[3] = -3; n[4] = 0; t = n;
        }
        else {
            n[0] = 4; n[1] = d->factors[numfac-1].value; n[2] = 1; n[3] = 3; n[4] = 0; t = n;
        }
    }
    else { t = d->factors[numfac-1].where; }
    while ( *t ) {
        if ( WriteTerm(t,&lbrac,first,PRINTON,0) ) {
            error = 1; break;
        }
        t += *t;
    }
    AC.OutputMode = oldOutputMode;
    AO.OutInBuffer = 0;
    AO.InFbrack = oldinfbrack;
    AO.CurBufWrt = oldcurbufwrt;
    AO.CurrentDictionary = dict;
    if ( error ) {
        MLOCK(ErrorMessageLock);
        MesPrint("&Illegal dollar object for writing");
        MUNLOCK(ErrorMessageLock);
        M_free(AO.DollarOutBuffer,"DollarOutBuffer");
        AO.DollarOutBuffer = 0;
        AO.DollarOutSizeBuffer = 0;
        return(0);
    }
    return(AO.DollarOutBuffer);
}

/*
    #] WriteDollarFactorToBuffer :
    #[ AddToDollarBuffer :
*/

void AddToDollarBuffer(UBYTE *s)
{
    int i;
    UBYTE *t = s, *u, *newdob;
    LONG j;
    while ( *t ) { t++; }
    i = t - s;
    while ( i + AO.DollarInOutBuffer >= AO.DollarOutSizeBuffer ) {
        j = AO.DollarInOutBuffer;
        AO.DollarOutSizeBuffer *= 2;
        t = AO.DollarOutBuffer;
        newdob = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,"DollarOutBuffer");
        u = newdob;
        while ( --j >= 0 ) *u++ = *t++;
        M_free(AO.DollarOutBuffer,"DollarOutBuffer");
        AO.DollarOutBuffer = newdob;
    }
    t = AO.DollarOutBuffer + AO.DollarInOutBuffer-1;
    while ( t == AO.DollarOutBuffer && ( *s == '+' || *s == ' ' ) ) s++;
    i = 0;
    if ( AO.CurrentDictionary == 0 ) {
        while ( *s ) {
            if ( *s == ' ' ) { s++; continue; }
```

```
            *t++ = *s++; i++;
        }
    }
    else {
        while ( *s ) { *t++ = *s++; i++; }
    }
    *t = 0;
    AO.DollarInOutBuffer += i;
}

/*
    #] AddToDollarBuffer :
    #[ TermAssign :

    This routine is called from a piece of code in Normalize that has been
    commented out.
*/

void TermAssign(WORD *term)
{
    DOLLARS d;
    WORD *t, *tstop, *astop, *w, *m;
    WORD i, newsize;
    for (;;) {
        astop = term + *term;
        tstop = astop - ABS(astop[-1]);
        t = term + 1;
        while ( t < tstop ) {
            if ( *t == AM.termfunnum && t[1] == FUNHEAD+2
            && t[FUNHEAD] == -DOLLAREXPRESSION ) {
                d = Dollars + t[FUNHEAD+1];
                newsize = *term - FUNHEAD - 1;
                if ( newsize < 32 ) newsize = 32;
                newsize = ((newsize+7)/8)*8;
                if ( d->size > 2*newsize && d->size > 1000 ) {
                    if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
                    d->size = 0;
                    d->where = &(AM.dollarzero);
                }
                if ( d->size < newsize ) {
                    if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
                    d->size = newsize;
                    d->where = (WORD *)Malloc1(newsize*sizeof(WORD),"dollar contents");
                }
                cbuf[AM.dbufnum].rhs[t[FUNHEAD+1]] = w = d->where;
                m = term;
                while ( m < t ) *w++ = *m++;
                m += t[1];
                while ( m < tstop ) {
                    if ( *m == AM.termfunnum && m[1] == FUNHEAD+2
                    && m[FUNHEAD] == -DOLLAREXPRESSION ) { m += m[1]; }
                    else {
                        i = m[1];
                        while ( --i >= 0 ) *w++ = *m++;
                    }
                }
                while ( m < astop ) *w++ = *m++;
                *(d->where) = w - d->where;
                *w = 0;
                d->type = DOLTERMS;
                w = t; m = t + t[1];
                while ( m < astop ) *w++ = *m++;
                *term = w - term;
                break;
            }
            t += t[1];
        }
        if ( t >= tstop ) return;
    }
}

/*
    #] TermAssign :
    #[ WildDollars :

    Note that we cannot upload wildcards into dollar variables when WITHPTHREADS.
LONG alloccounter = 0;
*/


void WildDollars(PHEAD WORD *term)
{
    GETBIDENTITY
    DOLLARS d;
    WORD *m, *t, *w, *ww, *orig = 0, *wildvalue, *wildstop;
```

```
    int numdollar;
    LONG weneed, i;
    struct DoLlArS;
#ifdef WITHPTHREADS
    int dtype = -1;
#endif
/*  alloccounter++; */
    if ( term == 0 ) {
        m = wildvalue = AN.WildValue;
        wildstop = AN.WildStop;
    }
    else {
        ww = term + *term; ww -= ABS(ww[-1]); w = term+1;
        while ( w < ww && *w != SUBEXPRESSION ) w += w[1];
        if ( w >= ww ) return;
        wildstop = w + w[1];
        w += SUBEXPSIZE;
        wildvalue = m = w;
    }
    while ( m < wildstop ) {
        if ( *m != LOADDOLLAR ) { m += m[1]; continue; }
        t = m - 4;
        while ( *t == LOADDOLLAR || *t == FROMSET || *t == SETTONUM ) t -= 4;
        if ( t < wildvalue ) {
            MLOCK(ErrorMessageLock);
            MesPrint("&Serious bug in wildcard prototype. Found in WildDollars");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        numdollar = m[2];
        d = Dollars + numdollar;
#ifdef WITHPTHREADS
        {
            int nummodopt;
            dtype = -1;
            if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
                for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                    if ( numdollar == ModOptdollars[nummodopt].number ) break;
                }
                if ( nummodopt < NumModOptdollars ) {
                    dtype = ModOptdollars[nummodopt].type;
                    if ( dtype == MODLOCAL ) {
                        d = ModOptdollars[nummodopt].dstruct+AT.identity;
                    }
                    else {
                        MLOCK(ErrorMessageLock);
                        MesPrint("&Illegal attempt to use $-variable %s in module %l",
                            DOLLARNAME(Dollars,numdollar),AC.CModule);
                        MUNLOCK(ErrorMessageLock);
                        Terminate(-1);
                    }
                }
            }
        }
#endif
/*
        The value of this wildcard goes into our $-variable
        First compute the space we need.
*/
        switch ( *t ) {
            case SYMTONUM:
                weneed = 5;
                break;
            case SYMTOSYM:
                weneed = 9;
                break;
            case SYMTOSUB:
            case VECTOSUB:
            case INDTOSUB:
                orig = cbuf[AT.ebufnum].rhs[t[3]];
                w = orig; while ( *w ) w += *w;
                weneed = w - orig + 1;
                break;
            case VECTOMIN:
            case VECTOVEC:
            case INDTOIND:
                weneed = 8;
                break;
            case FUNTOFUN:
                weneed = FUNHEAD+5;
                break;
            case ARGTOARG:
                orig = cbuf[AT.ebufnum].rhs[t[3]];
                if ( *orig > 0 ) weneed = *orig+2;
                else {
                    w = orig+1; while ( *w ) { NEXTARG(w) }
                    weneed = w - orig + 1;
```

```
                }
                break;
            default:
                weneed = 32;
                break;
        }
        if ( weneed < 32 ) weneed = 32;
        weneed = ((weneed+7)/8)*8;
        if ( d->size > 2*weneed && d->size > 1000 ) {
            if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollarspace");
            d->where = &(AM.dollarzero);
            d->size = 0;
        }
        if ( d->size < weneed ) {
            if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollarspace");
            d->where = (WORD *)Malloc1(weneed*sizeof(WORD),"dollarspace");
            d->size = weneed;
        }
/*
        It is not clear what the following code does for TFORM

        if ( dtype != MODLOCAL ) {
*/
            cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
            cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
/*          cbuf[AM.dbufnum].rhs[numdollar] = d->where; */
            cbuf[AM.dbufnum].rhs[numdollar] = (WORD *)(1);
/*
        }
        Now load up the value of the wildcard in compiler buffer format
*/
        w = d->where;
        d->type = DOLTERMS;
        switch ( *t ) {
            case SYMTONUM:
                d->where[0] = 4; d->where[2] = 1;
                if ( t[3] >= 0 ) { d->where[1] = t[3]; d->where[3] = 3; }
                else { d->where[1] = -t[3]; d->where[3] = -3; }
                if ( t[3] == 0 ) { d->type = DOLZERO; d->where[0] = 0; }
                else { d->type = DOLNUMBER; d->where[4] = 0; }
                break;
            case SYMTOSYM:
                *w++ = 8;
                *w++ = SYMBOL;
                *w++ = 4;
                *w++ = t[3];
                *w++ = 1;
                *w++ = 1;
                *w++ = 1;
                *w++ = 3;
                *w = 0;
                break;
            case SYMTOSUB:
            case VECTOSUB:
            case INDTOSUB:
                while ( *orig ) {
                    i = *orig; while ( --i >= 0 ) *w++ = *orig++;
                }
                *w = 0;
/*
                And then we have to fix up CanCommu
*/
                break;
            case VECTOMIN:
                *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[3];
                *w++ = 1; *w++ = 1; *w++ = -3; *w = 0;
                break;
            case VECTOVEC:
                *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[3];
                *w++ = 1; *w++ = 1; *w++ = 3; *w = 0;
                break;
            case INDTOIND:
                d->type = DOLINDEX; d->index = t[3]; *w = 0;
                break;
            case FUNTOFUN:
                *w++ = FUNHEAD+4; *w++ = t[3]; *w++ = FUNHEAD;
                FILLFUN(w)
                *w++ = 1; *w++ = 1; *w++ = 3; *w = 0;
                break;
            case ARGTOARG:
                if ( *orig > 0 ) ww = orig + *orig + 1;
                else {
                    ww = orig+1; while ( *ww ) { NEXTARG(ww) }
                }
                while ( orig < ww ) *w++ = *orig++;
                *w = 0;
                d->type = DOLWILDARGS;
```

```
                break;
            default:
                d->type = DOLUNDEFINED;
                break;
        }
        m += m[1];
    }
}

/*
    #] WildDollars :
    #[ DolToTensor :    with LOCK
*/

WORD DolToTensor(PHEAD WORD numdollar)
{
    GETBIDENTITY
    DOLLARS d = Dollars + numdollar;
    WORD retval;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
            else {
                LOCK(d->pthreadslockread);
            }
        }
    }
#endif
    AN.ErrorInDollar = 0;
    if ( d->type == DOLTERMS && d->where[0] == FUNHEAD+4 &&
    d->where[FUNHEAD+4] == 0 && d->where[FUNHEAD+3] == 3 &&
    d->where[FUNHEAD+2] == 1 && d->where[FUNHEAD+1] == 1 &&
    d->where[1] >= FUNCTION && d->where[1] < FUNCTION+WILDOFFSET
    && functions[d->where[1]-FUNCTION].spec >= TENSORFUNCTION ) {
        retval = d->where[1];
    }
    else if ( d->type == DOLARGUMENT &&
    d->where[0] <= -FUNCTION && d->where[0] > -FUNCTION-WILDOFFSET
    && functions[-d->where[0]-FUNCTION].spec >= TENSORFUNCTION ) {
        retval = -d->where[0];
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && d->where[1] <= -FUNCTION && d->where[1] > -FUNCTION-WILDOFFSET
    && d->where[2] == 0
    && functions[-d->where[1]-FUNCTION].spec >= TENSORFUNCTION ) {
        retval = -d->where[1];
    }
    else if ( d->type == DOLSUBTERM &&
    d->where[0] >= FUNCTION && d->where[0] < FUNCTION+WILDOFFSET
    && functions[d->where[0]-FUNCTION].spec >= TENSORFUNCTION ) {
        retval = d->where[0];
    }
    else {
        AN.ErrorInDollar = 1;
        retval = 0;
    }
#ifdef WITHPTHREADS
    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
    return(retval);
}

/*
    #] DolToTensor :
    #[ DolToFunction :  with LOCK
*/

WORD DolToFunction(PHEAD WORD numdollar)
{
    GETBIDENTITY
    DOLLARS d = Dollars + numdollar;
    WORD retval;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
```

```
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
            else {
                LOCK(d->pthreadslockread);
            }
        }
    }
#endif
    AN.ErrorInDollar = 0;
    if ( d->type == DOLTERMS && d->where[0] == FUNHEAD+4 &&
    d->where[FUNHEAD+4] == 0 && d->where[FUNHEAD+3] == 3 &&
    d->where[FUNHEAD+2] == 1 && d->where[FUNHEAD+1] == 1 &&
    d->where[1] >= FUNCTION && d->where[1] < FUNCTION+WILDOFFSET ) {
        retval = d->where[1];
    }
    else if ( d->type == DOLARGUMENT &&
    d->where[0] <= -FUNCTION && d->where[0] > -FUNCTION-WILDOFFSET ) {
        retval = -d->where[0];
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && d->where[1] <= -FUNCTION && d->where[1] > -FUNCTION-WILDOFFSET
    && d->where[2] == 0 ) {
        retval = -d->where[1];
    }
    else if ( d->type == DOLSUBTERM &&
    d->where[0] >= FUNCTION && d->where[0] < FUNCTION+WILDOFFSET ) {
        retval = d->where[0];
    }
    else {
        AN.ErrorInDollar = 1;
        retval = 0;
    }
#ifdef WITHPTHREADS
    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
    return(retval);
}

/*
    #] DolToFunction :
    #[ DolToVector :    with LOCK
*/

WORD DolToVector(PHEAD WORD numdollar)
{
    GETBIDENTITY
    DOLLARS d = Dollars + numdollar;
    WORD retval;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
            else {
                LOCK(d->pthreadslockread);
            }
        }
    }
#endif
    AN.ErrorInDollar = 0;
    if ( d->type == DOLINDEX && d->index < 0 ) {
        retval = d->index;
    }
    else if ( d->type == DOLARGUMENT && ( d->where[0] == -VECTOR
    || d->where[0] == -MINVECTOR ) ) {
        retval = d->where[1];
    }
    else if ( d->type == DOLSUBTERM && d->where[0] == INDEX
    && d->where[1] == 3 && d->where[2] < 0 ) {
        retval = d->where[2];
    }
    else if ( d->type == DOLTERMS && d->where[0] == 7 &&
    d->where[7] == 0 && d->where[6] == 3 &&
    d->where[5] == 1 && d->where[4] == 1 &&
    d->where[1] >= INDEX && d->where[3] < 0 ) {
        retval = d->where[3];
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && ( d->where[1] == -VECTOR || d->where[1] == -MINVECTOR )
```

```
    && d->where[3] == 0 ) {
        retval = d->where[2];
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 1
    && d->where[1] < 0 ) {
        retval = d->where[1];
    }
    else {
        AN.ErrorInDollar = 1;
        retval = 0;
    }
#ifdef WITHPTHREADS
    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
    return(retval);
}

/*
    #] DolToVector :
    #[ DolToNumber :
*/

WORD DolToNumber(PHEAD WORD numdollar)
{
    GETBIDENTITY
    DOLLARS d = Dollars + numdollar;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
        }
    }
#endif
    AN.ErrorInDollar = 0;
    if ( ( d->type == DOLTERMS || d->type == DOLNUMBER )
     && d->where[0] == 4 &&
    d->where[4] == 0 && ( d->where[3] == 3 || d->where[3] == -3 )
     && d->where[2] == 1 && ( d->where[1] & TOPBITONLY ) == 0 ) {
        if ( d->where[3] > 0 ) return(d->where[1]);
        else return(-d->where[1]);
    }
    else if ( d->type == DOLARGUMENT && d->where[0] == -SNUMBER ) {
        return(d->where[1]);
    }
    else if ( d->type == DOLARGUMENT && d->where[0] == -INDEX
    && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
        return(d->where[1]);
    }
    else if ( d->type == DOLZERO ) return(0);
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && d->where[1] == -SNUMBER && d->where[3] == 0 ) {
        return(d->where[2]);
    }
    else if ( d->type == DOLINDEX && d->index >= 0 && d->index < AM.OffsetIndex ) {
        return(d->index);
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 1
    && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
        return(d->where[1]);
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && d->where[1] == -INDEX && d->where[3] == 0 && d->where[2] >= 0
    && d->where[2] < AM.OffsetIndex ) {
        return(d->where[2]);
    }
    AN.ErrorInDollar = 1;
    return(0);
}

/*
    #] DolToNumber :
    #[ DolToSymbol :    with LOCK
*/

WORD DolToSymbol(PHEAD WORD numdollar)
{
    GETBIDENTITY
    DOLLARS d = Dollars + numdollar;
    WORD retval;
#ifdef WITHPTHREADS
```

```
    int nummodopt, dtype = -1;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
            else {
                LOCK(d->pthreadslockread);
            }
        }
    }
#endif
    AN.ErrorInDollar = 0;
    if ( d->type == DOLTERMS && d->where[0] == 8 &&
    d->where[8] == 0 && d->where[7] == 3 && d->where[6] == 1
     && d->where[5] == 1 && d->where[4] == 1 && d->where[1] == SYMBOL ) {
        retval = d->where[3];
    }
    else if ( d->type == DOLARGUMENT && d->where[0] == -SYMBOL ) {
        retval = d->where[1];
    }
    else if ( d->type == DOLSUBTERM && d->where[0] == SYMBOL
    && d->where[1] == 4 && d->where[3] == 1 ) {
        retval = d->where[2];
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && d->where[1] == -SYMBOL && d->where[3] == 0 ) {
        retval = d->where[2];
    }
    else {
        AN.ErrorInDollar = 1;
        retval = -1;
    }
#ifdef WITHPTHREADS
    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
    return(retval);
}

/*
    #] DolToSymbol :
    #[ DolToIndex :     with LOCK
*/

WORD DolToIndex(PHEAD WORD numdollar)
{
    GETBIDENTITY
    DOLLARS d = Dollars + numdollar;
    WORD retval;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
            else {
                LOCK(d->pthreadslockread);
            }
        }
    }
#endif
    AN.ErrorInDollar = 0;
    if ( d->type == DOLTERMS && d->where[0] == 7 &&
    d->where[7] == 0 && d->where[6] == 3 && d->where[5] == 1
     && d->where[4] == 1 && d->where[1] == INDEX && d->where[3] >= 0 ) {
        retval = d->where[3];
    }
    else if ( d->type == DOLARGUMENT && d->where[0] == -SNUMBER
    && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
        retval = d->where[1];
    }
    else if ( d->type == DOLARGUMENT && d->where[0] == -INDEX
    && d->where[1] >= 0 ) {
        retval = d->where[1];
    }
    else if ( d->type == DOLZERO ) return(0);
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && d->where[1] == -SNUMBER && d->where[3] == 0 && d->where[2] >= 0
```

```
    && d->where[2] < AM.OffsetIndex ) {
        retval = d->where[2];
    }
    else if ( d->type == DOLINDEX && d->index >= 0 ) {
        retval = d->index;
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 1
    && d->where[1] >= 0 ) {
        retval = d->where[1];
    }
    else if ( d->type == DOLSUBTERM && d->where[0] == INDEX
    && d->where[1] == 3 && d->where[2] >= 0 ) {
        retval = d->where[2];
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && d->where[1] == -INDEX && d->where[3] == 0 && d->where[2] >= 0 ) {
        retval = d->where[2];
    }
    else {
        AN.ErrorInDollar = 1;
        retval = 0;
    }
#ifdef WITHPTHREADS
    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
    return(retval);
}

/*
    #] DolToIndex :
    #[ DolToTerms :

    Returns a struct of type DOLLARS which contains a copy of the
    original dollar variable, provided it can be expressed in terms of
    an expression (type = DOLTERMS). Otherwise it returns zero.
    The dollar is expressed in terms in the buffer "where"
*/

DOLLARS DolToTerms(PHEAD WORD numdollar)
{
    GETBIDENTITY
    LONG size;
    DOLLARS d = Dollars + numdollar, newd;
    WORD *t, *w, i;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
        }
    }
#endif
    AN.ErrorInDollar = 0;
    switch ( d->type ) {
        case DOLARGUMENT:
            t = d->where;
            if ( t[0] < 0 ) {
ShortArgument:
                w = AT.WorkPointer;
                if ( t[0] <= -FUNCTION ) {
                    *w++ = FUNHEAD+4; *w++ = -t[0];
                    *w++ = FUNHEAD; FILLFUN(w)
                    *w++ = 1; *w++ = 1; *w++ = 3;
                }
                else if ( t[0] == -SYMBOL ) {
                    *w++ = 8; *w++ = SYMBOL; *w++ = 4; *w++ = t[1];
                    *w++ = 1; *w++ = 1; *w++ = 1; *w++ = 3;
                }
                else if ( t[0] == -VECTOR || t[0] == -INDEX ) {
                    *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[1];
                    *w++ = 1; *w++ = 1; *w++ = 3;
                }
                else if ( t[0] == -MINVECTOR ) {
                    *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[1];
                    *w++ = 1; *w++ = 1; *w++ = -3;
                }
                else if ( t[0] == -SNUMBER ) {
                    *w++ = 4;
                    if ( t[1] < 0 ) {
                        *w++ = -t[1]; *w++ = 1; *w++ = -3;
                    }
```

```
                    else {
                        *w++ = t[1]; *w++ = 1; *w++ = 3;
                    }
                }
                *w = 0; size = w - AT.WorkPointer;
                w = AT.WorkPointer;
                break;
            }
        case DOLNUMBER:
        case DOLTERMS:
            t = d->where;
            while ( *t ) t += *t;
            size = t - d->where;
            w = d->where;
            break;
        case DOLSUBTERM:
            w = AT.WorkPointer;
            size = d->where[1];
            *w++ = size+4; t = d->where; NCOPY(w,t,size)
            *w++ = 1; *w++ = 1; *w++ = 3;
            w = AT.WorkPointer; size = d->where[1]+4;
            break;
        case DOLINDEX:
            w = AT.WorkPointer;
            *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = d->index;
            *w++ = 1; *w++ = 1; *w++ = 3; *w = 0;
            w = AT.WorkPointer; size = 7;
            break;
        case DOLWILDARGS:
/*
            In some cases we can make a copy
*/
            t = d->where+1;
            if ( *t == 0 ) return(0);
            NEXTARG(t);
            if ( *t ) { /* More than one argument in here */
                MLOCK(ErrorMessageLock);
                MesPrint("Trying to convert a $ with an argument field into an expression");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
/*
            Now we have a single argument
*/
            t = d->where+1;
            if ( *t < 0 ) goto ShortArgument;
            size = *t - ARGHEAD;
            w = t + ARGHEAD;
            break;
        case DOLUNDEFINED:
            MLOCK(ErrorMessageLock);
            MesPrint("Trying to use an undefined $ in an expression");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        case DOLZERO:
            if ( d->where ) { d->where[0] = 0; }
            else d->where = &(AM.dollarzero);
            size = 0;
            w = d->where;
            break;
        default:
            return(0);
    }
    newd = (DOLLARS)Malloc1(sizeof(struct DoLlArS)+(size+1)*sizeof(WORD),
                "Copy of dollar variable");
    t = (WORD *)(newd+1);
    newd->where = t;
    newd->name = d->name;
    newd->node = d->node;
    newd->type = DOLTERMS;
    newd->size = size;
    newd->numdummies = d->numdummies;
#ifdef WITHPTHREADS
    newd->pthreadslockread  = dummylock;
    newd->pthreadslockwrite = dummylock;
#endif
    size++;
    NCOPY(t,w,size);
    newd->nfactors = d->nfactors;
    if ( d->nfactors > 1 ) {
        newd->factors = (FACDOLLAR *)Malloc1(d->nfactors*sizeof(FACDOLLAR),"Dollar factors");
        for ( i = 0; i < d->nfactors; i++ ) {
            newd->factors[i].where = 0;
            newd->factors[i].size = 0;
            newd->factors[i].type = DOLUNDEFINED;
            newd->factors[i].value = d->factors[i].value;
        }
```

```
    }
    else { newd->factors = 0; }
    return(newd);
}

/*
    #] DolToTerms :
    #[ DolToLong :
*/

LONG DolToLong(PHEAD WORD numdollar)
{
    GETBIDENTITY
    DOLLARS d = Dollars + numdollar;
    LONG x;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
        }
    }
#endif
    AN.ErrorInDollar = 0;
    if ( ( d->type == DOLTERMS || d->type == DOLNUMBER )
     && d->where[0] == 4 &&
    d->where[4] == 0 && ( d->where[3] == 3 || d->where[3] == -3 )
     && d->where[2] == 1 && ( d->where[1] & TOPBITONLY ) == 0 ) {
        x = d->where[1];
        if ( d->where[3] > 0 ) return(x);
        else return(-x);
    }
    else if ( ( d->type == DOLTERMS || d->type == DOLNUMBER )
     && d->where[0] == 6 &&
    d->where[6] == 0 && ( d->where[5] == 5 || d->where[5] == -5 )
     && d->where[3] == 1 && d->where[4] == 1 && ( d->where[2] & TOPBITONLY ) == 0 ) {
        x = d->where[1] + ( (LONG)(d->where[2]) « BITSINWORD );
        if ( d->where[5] > 0 ) return(x);
        else return(-x);
    }
    else if ( d->type == DOLARGUMENT && d->where[0] == -SNUMBER ) {
        x = d->where[1];
        return(x);
    }
    else if ( d->type == DOLARGUMENT && d->where[0] == -INDEX
    && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
        x = d->where[1];
        return(x);
    }
    else if ( d->type == DOLZERO ) return(0);
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && d->where[1] == -SNUMBER && d->where[3] == 0 ) {
        x = d->where[2];
        return(x);
    }
    else if ( d->type == DOLINDEX && d->index >= 0 && d->index < AM.OffsetIndex ) {
        x = d->index;
        return(x);
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 1
    && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
        x = d->where[1];
        return(x);
    }
    else if ( d->type == DOLWILDARGS && d->where[0] == 0
    && d->where[1] == -INDEX && d->where[3] == 0 && d->where[2] >= 0
    && d->where[2] < AM.OffsetIndex ) {
        x = d->where[2];
        return(x);
    }
    AN.ErrorInDollar = 1;
    return(0);
}

/*
    #] DolToLong :
    #[ ExecInside :
*/

int ExecInside(UBYTE *s)
{
```

```
    GETIDENTITY
    UBYTE *t, c;
    WORD *w, number;
    int error = 0;
    w = AT.WorkPointer;
    if ( AC.insidelevel >= MAXNEST ) {
        MLOCK(ErrorMessageLock);
        MesPrint("@Nesting of inside statements more than %d levels",(WORD)MAXNEST);
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    AC.insidesumcheck[AC.insidelevel] = NestingChecksum();
    AC.insidestack[AC.insidelevel] = cbuf[AC.cbufnum].Pointer
                                - cbuf[AC.cbufnum].Buffer + 2;
    AC.insidelevel++;
    *w++ = TYPEINSIDE;
    w++; w++;
    for(;;) {   /* Look for a (comma separated) list of dollar variables */
        while ( *s == ',' ) s++;
        if ( *s == 0 ) break;
        if ( *s == '$' ) {
            s++; t = s;
            if ( FG.cTable[*s] != 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal name for $ variable: %s",s-1);
                MUNLOCK(ErrorMessageLock);
                goto skipdol;
            }
            while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
            c = *s; *s = 0;
            if ( ( number = GetDollar(t) ) < 0 ) {
                number = AddDollar(t,0,0,0);
            }
            *s = c;
            *w++ = number;
            AddPotModdollar(number);
        }
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("&Illegal object in Inside statement");
            MUNLOCK(ErrorMessageLock);
skipdol:    error = 1;
            while ( *s && *s != ',' && s[1] != '$' ) s++;
            if ( *s == 0 ) break;
        }
    }
    AT.WorkPointer[1] = w - AT.WorkPointer;
    AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
    return(error);
}

/*
    #] ExecInside :
    #[ InsideDollar :

    Execution part of Inside $a;
    We have to take the variables one by one and then
    convert them into proper terms and call Generator for the proper levels.
    The conversion copies the whole dollar into a new buffer, making us
    insensitive to redefinitions of $a inside the Inside.
    In the end we sort and redefine $a.
*/

int InsideDollar(PHEAD WORD *ll, WORD level)
{
    GETBIDENTITY
    int numvar = (int)(ll[1]-3), j, error = 0;
    WORD numdol, *oldcterm, *oldwork = AT.WorkPointer, olddefer, *r, *m;
    WORD oldnumlhs, *dbuffer;
    DOLLARS d, newd;
    oldcterm = AN.cTerm; AN.cTerm = 0;
    oldnumlhs = AR.Cnumlhs; AR.Cnumlhs = ll[2];
    ll += 3;
    olddefer = AR.DeferFlag;
    AR.DeferFlag = 0;
    while ( --numvar >= 0 ) {
      numdol = *ll++;
      d = Dollars + numdol;
      {
#ifdef WITHPTHREADS
        int nummodopt, dtype = -1;
        if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
            for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                if ( numdol == ModOptdollars[nummodopt].number ) break;
            }
            if ( nummodopt < NumModOptdollars ) {
                dtype = ModOptdollars[nummodopt].type;
```

```
                if ( dtype == MODLOCAL ) {
                    d = ModOptdollars[nummodopt].dstruct+AT.identity;
                }
                else {
/*                  LOCK(d->pthreadslockwrite); */
                    LOCK(d->pthreadslockread);
                }
            }
        }
#endif
        newd = DolToTerms(BHEAD numdol);
        if ( newd == 0 || newd->where[0] == 0 ) continue;
        r = newd->where;
        NewSort(BHEAD0);
        while ( *r ) {  /* Sum over the terms */
            m = AT.WorkPointer;
            j = *r;
            while ( --j >= 0 ) *m++ = *r++;
            AT.WorkPointer = m;
/*
            What to do with dummy indices?
*/
            if ( Generator(BHEAD oldwork,level) ) {
                LowerSortLevel();
                error = -1; goto idcall;
            }
            AT.WorkPointer = oldwork;
        }
        AN.tryterm = 0; /* for now */
        if ( EndSort(BHEAD (WORD *)((VOID *)(&dbuffer)),2) < 0 ) { error = 1; break; }
        if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"old buffer of dollar");
        d->where = dbuffer;
        if ( dbuffer == 0 || *dbuffer == 0 ) {
            d->type = DOLZERO;
            if ( dbuffer ) M_free(dbuffer,"buffer of dollar");
            d->where = &(AM.dollarzero); d->size = 0;
        }
        else {
            d->type = DOLTERMS;
            r = d->where; while ( *r ) r += *r;
            d->size = (r-d->where)+1;
        }
/*      cbuf[AM.dbufnum].rhs[numdol] = d->where; */
        cbuf[AM.dbufnum].rhs[numdol] = (WORD *)(1);
/*
        Now we have a little cleaning up to do
*/
#ifdef WITHPTHREADS
        if ( dtype > 0 && dtype != MODLOCAL ) {
/*          UNLOCK(d->pthreadslockwrite); */
            UNLOCK(d->pthreadslockread);
        }
#endif
        if ( newd->factors ) M_free(newd->factors,"Dollar factors");
        M_free(newd,"Copy of dollar variable");
      }
    }
idcall:;
    AR.Cnumlhs = oldnumlhs;
    AR.DeferFlag = olddefer;
    AN.cTerm = oldcterm;
    AT.WorkPointer = oldwork;
    return(error);
}

/*
    #] InsideDollar :
    #[ ExchangeDollars :
*/

void ExchangeDollars(int num1, int num2)
{
    DOLLARS d1, d2;
    WORD node1, node2;
    LONG nam;
    d1 = Dollars + num1; node1 = d1->node;
    d2 = Dollars + num2; node2 = d2->node;
    nam = d1->name; d1->name = d2->name; d2->name = nam;
    d1->node = node2; d2->node = node1;
    AC.dollarnames->namenode[node1].number = num2;
    AC.dollarnames->namenode[node2].number = num1;
}

/*
    #] ExchangeDollars :
    #[ TermsInDollar :
*/
```

```
LONG TermsInDollar(WORD num)
{
    GETIDENTITY
    DOLLARS d = Dollars + num;
    WORD *t;
    LONG n;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1;
    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( num == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
            else {
                LOCK(d->pthreadslockread);
            }
        }
    }
#endif
    if ( d->type == DOLTERMS ) {
        n = 0;
        t = d->where;
        while ( *t ) { t += *t; n++; }
    }
    else if ( d->type == DOLWILDARGS ) {
        n = 0;
        if ( d->where[0] == 0 ) {
            t = d->where+1;
            while ( *t != 0 ) { NEXTARG(t); n++; }
        }
        else if ( d->where[0] == 1 ) n = 1;
    }
    else if ( d->type == DOLZERO ) n = 0;
    else n = 1;
#ifdef WITHPTHREADS
    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
    return(n);
}

/*
    #] TermsInDollar :
    #[ PreIfDollarEval :

    Routine is invoked in #if etc after $( is encountered.
    $(expr1 operator expr2) makes compares between expressions,
    $(expr1 operator _keyword) makes compares between expressions,
    interpreted as expressions. We are here mainly looking at $variables.
    First we look for the operator:
        >, <, ==, >=, <=, != : < means that it comes before.
    _keywords can be:
        _set(setname)   (does the expr belong to the set (only with == or !=))
        _productof(expr)
*/

UBYTE *PreIfDollarEval(UBYTE *s, int *value)
{
    GETIDENTITY
    UBYTE *s1,*s2,*s3,*s4,*s5,*t,c,c1,c2,c3;
    int oprtr, type;
    WORD *buf1 = 0, *buf2 = 0, numset, *oldwork = AT.WorkPointer;
    EXCHINOUT
/*
    Find the three composing objects (epxression, operator, expression or keyw
*/
    while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
    s1 = t = s;
    while ( *t != '=' && *t != '!' && *t != '>' && *t != '<' ) {
        if ( *t == '[' ) { SKIPBRA1(t) }
        else if ( *t == '{' ) { SKIPBRA2(t) }
        else if ( *t == '(' ) { SKIPBRA3(t) }
        else if ( *t == ']' || *t == '}' || *t == ')' ) {
            MLOCK(ErrorMessageLock);
            MesPrint("@Improper bracketting in #if");
            MUNLOCK(ErrorMessageLock);
            goto onerror;
        }
        t++;
    }
    s2 = t;
    while ( *t == '=' || *t == '!' || *t == '>' || *t == '<' ) t++;
    s3 = t;
```

```
    while ( *t && *t != ')' ) {
        if ( *t == '[' ) { SKIPBRA1(t) }
        else if ( *t == '{' ) { SKIPBRA2(t) }
        else if ( *t == '(' ) { SKIPBRA3(t) }
        else if ( *t == ']' || *t == '}' ) {
            MLOCK(ErrorMessageLock);
            MesPrint("@Improper brackets in #if");
            MUNLOCK(ErrorMessageLock);
            goto onerror;
        }
        t++;
    }
    if ( *t == 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("@Missing ) to match $( in #if");
        MUNLOCK(ErrorMessageLock);
        goto onerror;
    }
    s4 = t; c2 = *s4; *s4 = 0;
    if ( s2+2 < s3 || s2 == s3 ) {
IllOp:;
        MLOCK(ErrorMessageLock);
        MesPrint("@Illegal operator in $( option of #if");
        MUNLOCK(ErrorMessageLock);
        goto onerror;
    }
    if ( s2+1 == s3 ) {
        if ( *s2 == '=' ) oprtr = EQUAL;
        else if ( *s2 == '>' ) oprtr = GREATER;
        else if ( *s2 == '<' ) oprtr = LESS;
        else goto IllOp;
    }
    else if ( *s2 == '!' && s2[1] == '=' ) oprtr = NOTEQUAL;
    else if ( *s2 == '=' && s2[1] == '=' ) oprtr = EQUAL;
    else if ( *s2 == '<' && s2[1] == '=' ) oprtr = LESSEQUAL;
    else if ( *s2 == '>' && s2[1] == '=' ) oprtr = GREATEREQUAL;
    else goto IllOp;
    c1 = *s2; *s2 = 0;
/*
    The two expressions are now zero terminated
    Look for the special keywords
*/
    while ( *s3 == ' ' || *s3 == '\t' || *s3 == '\n' || *s3 == '\r' ) s3++;
    t = s3;
    while ( chartype[*t] == 0 ) t++;
    if ( *t == '_' ) {
        t++; c = *t; *t = 0;
        if ( StrICmp(s3,(UBYTE *)"set_") == 0 ) {
            if ( oprtr != EQUAL && oprtr != NOTEQUAL ) {
ImpOp:;
                MLOCK(ErrorMessageLock);
                MesPrint("@Improper operator for special keyword in $( ) option");
                MUNLOCK(ErrorMessageLock);
                goto onerror;
            }
            type = 1;
        }
        else if ( StrICmp(s3,(UBYTE *)"multipleof_") == 0 ) {
            if ( oprtr != EQUAL && oprtr != NOTEQUAL ) goto ImpOp;
            type = 2;
        }
/*
        else if ( StrICmp(s3,(UBYTE *)"productof_") == 0 ) {
            if ( oprtr != EQUAL && oprtr != NOTEQUAL ) goto ImpOp;
            type = 3;
        }
*/
        else type = 0;
    }
    else { type = 0; c = *t; }
    if ( type > 0 ) {
        *t++ = c; s3 = t; s5 = s4-1;
        while ( *s5 != ')' ) {
            if ( *s5 == ' ' || *s5 == '\t' || *s5 == '\n' || *s5 == '\r' ) s5--;
            else {
                MLOCK(ErrorMessageLock);
                MesPrint("@Improper use of special keyword in $( ) option");
                MUNLOCK(ErrorMessageLock);
                goto onerror;
            }
        }
        c3 = *s5; *s5 = 0;
    }
    else { c3 = c2; s5 = s4; }
/*
    Expand the first expression.
*/
```

```
    if ( ( buf1 = TranslateExpression(s1) ) == 0 ) {
        AT.WorkPointer = oldwork;
        goto onerror;
    }
    if ( type == 1 ) {  /* determine the set */
        if ( *s3 == '{' ) {
            t = s3+1;
            SKIPBRA2(s3)
            numset = DoTempSet(t,s3);
            s3++;
            if ( numset < 0 ) {
noset:;
                MLOCK(ErrorMessageLock);
                MesPrint("@Argument of set_ is not a valid set");
                MUNLOCK(ErrorMessageLock);
                goto onerror;
            }
        }
        else {
            t = s3;
            while ( FG.cTable[*s3] == 0 || FG.cTable[*s3] == 1
                || *s3 == '_' ) s3++;
            c = *s3; *s3 = 0;
            if ( GetName(AC.varnames,t,&numset,NOAUTO) != CSET ) {
                *s3 = c; goto noset;
            }
            *s3 = c;
        }
        while ( *s3 == ' ' || *s3 == '\t' || *s3 == '\n' || *s3 == '\r' ) s3++;
        if ( s3 != s5 ) goto noset;
        *value = IsSetMember(buf1,numset);
        if ( oprtr == NOTEQUAL ) *value ^= 1;
    }
    else {
        if ( ( buf2 = TranslateExpression(s3) ) == 0 ) goto onerror;
    }
    if ( type == 0 ) {
        *value = TwoExprCompare(buf1,buf2,oprtr);
    }
    else if ( type == 2 ) {
        *value = IsMultipleOf(buf1,buf2);
        if ( oprtr == NOTEQUAL ) *value ^= 1;
    }
/*
    else if ( type == 3 ) {
        *value = IsProductOf(buf1,buf2);
        if ( oprtr == NOTEQUAL ) *value ^= 1;
    }
*/
    if ( buf1 ) M_free(buf1,"Buffer in $()");
    if ( buf2 ) M_free(buf2,"Buffer in $()");
    *s5 = c3; *s4++ = c2; *s2 = c1;
    AT.WorkPointer = oldwork;
    BACKINOUT
    return(s4);
onerror:
    if ( buf1 ) M_free(buf1,"Buffer in $()");
    if ( buf2 ) M_free(buf2,"Buffer in $()");
    AT.WorkPointer = oldwork;
    BACKINOUT
    return(0);
}

/*
    #] PreIfDollarEval :
    #[ TranslateExpression :
*/

WORD *TranslateExpression(UBYTE *s)
{
    GETIDENTITY
    CBUF *C = cbuf+AC.cbufnum;
    WORD oldnumrhs = C->numrhs;
    LONG oldcpointer = C->Pointer - C->Buffer;
    WORD *w = AT.WorkPointer;
    WORD retcode, oldEside;
    WORD *outbuffer;
    *w++ = SUBEXPSIZE + 4;
    AC.ProtoType = w;
    *w++ = SUBEXPRESSION;
    *w++ = SUBEXPSIZE;
    *w++ = C->numrhs+1;
    *w++ = 1;
    *w++ = AC.cbufnum;
    FILLSUB(w)
    *w++ = 1; *w++ = 1; *w++ = 3; *w++ = 0;
    AT.WorkPointer = w;
```

```
    if ( ( retcode = CompileAlgebra(s,RHSIDE,AC.ProtoType) ) < 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("@Error translating first expression in $( ) option");
        MUNLOCK(ErrorMessageLock);
        return(0);
    }
    else { AC.ProtoType[2] = retcode; }
/*
    Evaluate this expression
*/
    if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) { return(0); }
    AN.RepPoint = AT.RepCount + 1;
    oldEside = AR.Eside; AR.Eside = RHSIDE;
    AR.Cnumlhs = C->numlhs;
    if ( Generator(BHEAD AC.ProtoType-1,C->numlhs) ) {
        AR.Eside = oldEside;
        LowerSortLevel(); LowerSortLevel(); return(0);
    }
    AR.Eside = oldEside;
    AT.WorkPointer = w;
    AN.tryterm = 0; /* for now */
    if ( EndSort(BHEAD (WORD *)((VOID *)(&outbuffer)),2) < 0 ) { LowerSortLevel(); return(0); }
    LowerSortLevel();
    C->Pointer = C->Buffer + oldcpointer;
    C->numrhs = oldnumrhs;
    AT.WorkPointer = AC.ProtoType - 1;
    return(outbuffer);
}

/*
    #] TranslateExpression :
    #[ IsSetMember :

    Checks whether the expression in the buffer can be seen as an element
    of the given set.
    For the special sets: if more than one term: no match!!!
*/

int IsSetMember(WORD *buffer, WORD numset)
{
    WORD *t = buffer, *tt, num, csize, num1;
    WORD bufterm[4];
    int i, j, type;
    if ( numset < AM.NumFixedSets ) {
        if ( t[*t] != 0 ) return(0);    /* More than one term */
        if ( *t == 0 ) {
            if ( numset == POS0_ || numset == NEG0_ || numset == EVEN_
            || numset == Z_ || numset == Q_ ) return(1);
            else return(0);
        }
        if ( numset == SYMBOL_ ) {
            if ( *t == 8 && t[1] == SYMBOL && t[7] == 3 && t[6] == 1
            && t[5] == 1 && t[4] == 1 ) return(1);
            else return(0);
        }
        if ( numset == INDEX_ ) {
            if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
            && t[4] == 1 && t[3] > 0 ) return(1);
            if ( *t == 4 && t[3] == 3 && t[2] == 1 && t[1] < AM.OffsetIndex)
                return(1);
            return(0);
        }
        if ( numset == FIXED_ ) {
            if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
            && t[4] == 1 && t[3] > 0 && t[3] < AM.OffsetIndex ) return(1);
            if ( *t == 4 && t[3] == 3 && t[2] == 1 && t[1] < AM.OffsetIndex)
                return(1);
            return(0);
        }
        if ( numset == DUMMYINDEX_ ) {
            if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
            && t[4] == 1 && t[3] >= AM.IndDum && t[3] < AM.IndDum+MAXDUMMIES ) return(1);
            if ( *t == 4 && t[3] == 3 && t[2] == 1
                 && t[1] >= AM.IndDum && t[1] < AM.IndDum+MAXDUMMIES ) return(1);
            return(0);
        }
        if ( numset == VECTOR_ ) {
            if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
            && t[4] == 1 && t[3] < (AM.OffsetVector+WILDOFFSET) && t[3] >= AM.OffsetVector )
return(1);
            return(0);
        }
        tt = t + *t - 1;
        if ( ABS(tt[0]) != *t-1 ) return(0);
        if ( numset == Q_ ) return(1);
        if ( numset == POS_ || numset == POS0_ ) return(tt[0]>0);
        else if ( numset == NEG_ || numset == NEG0_ ) return(tt[0]<0);
```

```
        i = (ABS(tt[0])-1)/2;
        tt -= i;
        if ( tt[0] != 1 ) return(0);
        for ( j = 1; j < i; j++ ) { if ( tt[j] != 0 ) return(0); }
        if ( numset == Z_ ) return(1);
        if ( numset == ODD_ ) return(t[1]&1);
        if ( numset == EVEN_ ) return(1-(t[1]&1));
        return(0);
    }
    if ( t[*t] != 0 ) return(0);    /* More than one term */
    type = Sets[numset].type;
    switch ( type ) {
        case CSYMBOL:
            if ( t[0] == 8 && t[1] == SYMBOL && t[7] == 3 && t[6] == 1
            && t[5] == 1 && t[4] == 1 ) {
                num = t[3];
            }
            else if ( t[0] == 4 && t[2] == 1 && t[1] <= MAXPOWER ) {
                num = t[1];
                if ( t[3] < 0 ) num = -num;
                num += 2*MAXPOWER;
            }
            else return(0);
            break;
        case CVECTOR:
            if ( t[0] == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
            && t[4] == 1 && t[3] < 0 ) {
                num = t[3];
            }
            else return(0);
            break;
        case CINDEX:
            if ( t[0] == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
            && t[4] == 1 && t[3] > 0 ) {
                num = t[3];
            }
            else if ( t[0] == 4 && t[3] == 3 && t[2] == 1 && t[1] < AM.OffsetIndex ) {
                num = t[1];
            }
            else return(0);
            break;
        case CFUNCTION:
            if ( t[0] == 4+FUNHEAD && t[3+FUNHEAD] == 3 && t[2+FUNHEAD] == 1
            && t[1+FUNHEAD] == 1 && t[1] >= FUNCTION ) {
                num = t[1];
            }
            else return(0);
            break;
        case CNUMBER:
            if ( t[0] == 4 && t[2] == 1 && t[1] <= AM.OffsetIndex && t[3] == 3 ) {
                num = t[1];
            }
            else return(0);
            break;
        case CRANGE:
            csize = t[t[0]-1];
            csize = ABS(csize);
            if ( csize != t[0]-1 ) return(0);
            if ( Sets[numset].first < 3*MAXPOWER ) {
                num1 = num = Sets[numset].first;
                if ( num >= MAXPOWER ) num -= 2*MAXPOWER;
                if ( num == 0 ) {
                    if ( num1 < MAXPOWER ) {
                        if ( t[t[0]-1] >= 0 ) return(0);
                    }
                    else if ( t[t[0]-1] > 0 ) return(0);
                }
                else {
                    bufterm[0] = 4; bufterm[1] = ABS(num);
                    bufterm[2] = 1;
                    if ( num < 0 ) bufterm[3] = -3;
                    else bufterm[3] = 3;
                    num = CompCoef(t,bufterm);
                    if ( num1 < MAXPOWER ) {
                        if ( num >= 0 ) return(0);
                    }
                    else if ( num > 0 ) return(0);
                }
            }
            if ( Sets[numset].last > -3*MAXPOWER ) {
                num1 = num = Sets[numset].last;
                if ( num <= -MAXPOWER ) num += 2*MAXPOWER;
                if ( num == 0 ) {
                    if ( num1 > -MAXPOWER ) {
                        if ( t[t[0]-1] <= 0 ) return(0);
                    }
                    else if ( t[t[0]-1] < 0 ) return(0);
```

```
                }
                else {
                    bufterm[0] = 4; bufterm[1] = ABS(num);
                    bufterm[2] = 1;
                    if ( num < 0 ) bufterm[3] = -3;
                    else bufterm[3] = 3;
                    num = CompCoef(t,bufterm);
                    if ( num1 > -MAXPOWER ) {
                        if ( num <= 0 ) return(0);
                    }
                    else if ( num < 0 ) return(0);
                }
            }
            return(1);
            break;
        default: return(0);
    }
    t  = SetElements + Sets[numset].first;
    tt = SetElements + Sets[numset].last;
    do {
        if ( num == *t ) return(1);
        t++;
    } while ( t < tt );
    return(0);
}

/*
    #] IsSetMember :
    #[ IsProductOf :

    Checks whether the expression in buf1 is a single term multiple of
    the expression in buf2.

int IsProductOf(WORD *buf1, WORD *buf2)
{
    return(0);
}


    #] IsProductOf :
    #[ IsMultipleOf :

    Checks whether the expression in buf1 is a numerical multiple of
    the expression in buf2.
*/

int IsMultipleOf(WORD *buf1, WORD *buf2)
{
    GETIDENTITY
    LONG num1, num2;
    WORD *t1, *t2, *m1, *m2, *r1, *r2, nc1, nc2, ni1, ni2;
    UWORD *IfScrat1, *IfScrat2;
    int i, j;
    if ( *buf1 == 0 && *buf2 == 0 ) return(1);
/*
    First count terms
*/
    t1 = buf1; t2 = buf2; num1 = 0; num2 = 0;
    while ( *t1 ) { t1 += *t1; num1++; }
    while ( *t2 ) { t2 += *t2; num2++; }
    if ( num1 != num2 ) return(0);
/*
    Test similarity of terms. Difference up to a number.
*/
    t1 = buf1; t2 = buf2;
    while ( *t1 ) {
        m1 = t1+1; m2 = t2+1; t1 += *t1; t2 += *t2;
        r1 = t1 - ABS(t1[-1]); r2 = t2 - ABS(t2[-1]);
        if ( r1-m1 != r2-m2 ) return(0);
        while ( m1 < r1 ) {
            if ( *m1 != *m2 ) return(0);
            m1++; m2++;
        }
    }
/*
    Now we have to test the constant factor
*/
    IfScrat1 = (UWORD *)(TermMalloc("IsMultipleOf")); IfScrat2 = (UWORD
*)(TermMalloc("IsMultipleOf"));
    t1 = buf1; t2 = buf2;
    t1 += *t1; t2 += *t2;
    if ( *t1 == 0 && *t2 == 0 ) return(1);
    r1 = t1 - ABS(t1[-1]); r2 = t2 - ABS(t2[-1]);
    nc1 = REDLENG(t1[-1]); nc2 = REDLENG(t2[-1]);
    if ( DivRat(BHEAD (UWORD *)r1,nc1,(UWORD *)r2,nc2,IfScrat1,&ni1) ) {
        MLOCK(ErrorMessageLock);
        MesPrint("@Called from MultipleOf in $( )");
```

```
        MUNLOCK(ErrorMessageLock);
        TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
        Terminate(-1);
    }
    while ( *t1 ) {
        t1 += *t1; t2 += *t2;
        r1 = t1 - ABS(t1[-1]); r2 = t2 - ABS(t2[-1]);
        nc1 = REDLENG(t1[-1]); nc2 = REDLENG(t2[-1]);
        if ( DivRat(BHEAD (UWORD *)r1,nc1,(UWORD *)r2,nc2,IfScrat2,&ni2) ) {
            MLOCK(ErrorMessageLock);
            MesPrint("@Called from MultipleOf in $( )");
            MUNLOCK(ErrorMessageLock);
            TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
            Terminate(-1);
        }
        if ( ni1 != ni2 ) return(0);
        i = 2*ABS(ni1);
        for ( j = 0; j < i; j++ ) {
            if ( IfScrat1[j] != IfScrat2[j] ) {
                TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
                return(0);
            }
        }
    }
    TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
    return(1);
}

/*
    #] IsMultipleOf :
    #[ TwoExprCompare :

    Compares the expressions in buf1 and buf2 according to oprtr
*/

int TwoExprCompare(WORD *buf1, WORD *buf2, int oprtr)
{
    GETIDENTITY
    WORD *t1, *t2, cond;
    t1 = buf1; t2 = buf2;
    while ( *t1 && *t2 ) {
        cond = CompareTerms(BHEAD t1,t2,1);
        if ( cond != 0 ) {
            if ( cond > 0 ) { /* t1 comes first */
                switch ( oprtr ) {  /* t1 is less */
                    case EQUAL: return(0);
                    case NOTEQUAL: return(1);
                    case GREATEREQUAL: return(0);
                    case GREATER: return(0);
                    case LESS: return(1);
                    case LESSEQUAL: return(1);
                }
            }
            else {
                switch ( oprtr ) {
                    case EQUAL: return(0);
                    case NOTEQUAL: return(1);
                    case GREATEREQUAL: return(1);
                    case GREATER: return(1);
                    case LESS: return(0);
                    case LESSEQUAL: return(0);
                }
            }
        }
        t1 += *t1; t2 += *t2;
    }
    if ( *t1 == *t2 ) { /* They are equal */
        switch ( oprtr ) {
            case EQUAL: return(1);
            case NOTEQUAL: return(0);
            case GREATEREQUAL: return(1);
            case GREATER: return(0);
            case LESS: return(0);
            case LESSEQUAL: return(1);
        }
    }
    else if ( *t1 ) {  /* t1 is greater */
        switch ( oprtr ) {
            case EQUAL: return(0);
            case NOTEQUAL: return(1);
            case GREATEREQUAL: return(1);
            case GREATER: return(1);
            case LESS: return(0);
            case LESSEQUAL: return(0);
        }
    }
    else {
```

```
        switch ( oprtr ) {  /* t1 is less */
            case EQUAL: return(0);
            case NOTEQUAL: return(1);
            case GREATEREQUAL: return(0);
            case GREATER: return(0);
            case LESS: return(1);
            case LESSEQUAL: return(1);
        }
    }
    MLOCK(ErrorMessageLock);
    MesPrint("@Internal problems with operator in $( )");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(0);
}

/*
    #] TwoExprCompare :
    #[ DollarRaiseLow :

    Raises or lowers the numerical value of a dollar variable
    Not to be used in parallel.
*/

static UWORD *dscrat = 0;
static WORD ndscrat;

int DollarRaiseLow(UBYTE *name, LONG value)
{
    GETIDENTITY
    int num;
    DOLLARS d;
    int sgn = 1;
    WORD lnum[4], nnum, *t1, *t2, i;
    UBYTE *s, c;
    s = name; while ( *s ) s++;
    if ( s[-1] == '-' && s[-2] == '-' && s > name+2 ) s -= 2;
    else if ( s[-1] == '+' && s[-2] == '+' && s > name+2 ) s -= 2;
    c = *s; *s = 0;
    num = GetDollar(name);
    *s = c;
    d = Dollars + num;
    if ( value < 0 ) { value = -value; sgn = -1; }
    if ( d->type == DOLZERO ) {
        if ( d->where ) M_free(d->where,"DollarRaiseLow");
        d->size = 32;
        d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"DollarRaiseLow");
        if ( ( value & AWORDMASK ) != 0 ) {
            d->where[0] = 6; d->where[1] = value » BITSINWORD;
            d->where[2] = (WORD)value; d->where[3] = 1; d->where[4] = 0;
            d->where[5] = 5*sgn; d->where[6] = 0;
            d->type = DOLTERMS;
        }
        else {
            d->where[0] = 4; d->where[1] = (WORD)value; d->where[2] = 1;
            d->where[3] = 3*sgn; d->where[4] = 0;
            d->type = DOLNUMBER;
        }
    }
    else if ( d->type == DOLNUMBER || ( d->type == DOLTERMS
    && d->where[d->where[0]] == 0
    && d->where[0] == ABS(d->where[d->where[0]-1])+1 ) ) {
        if ( ( value & AWORDMASK ) != 0 ) {
            lnum[0] = value » BITSINWORD;
            lnum[1] = (WORD)value; lnum[2] = 1; lnum[3] = 0;
            nnum = 2*sgn;
        }
        else {
            lnum[0] = (WORD)value; lnum[1] = 1; nnum = sgn;
        }
        i = d->where[d->where[0]-1];
        i = REDLENG(i);
        if ( dscrat == 0 ) {
            dscrat = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"DollarRaiseLow");
        }
        if ( AddRat(BHEAD (UWORD *)(d->where+1),i,
            (UWORD *)lnum,nnum,dscrat,&ndscrat) ) {
                MLOCK(ErrorMessageLock);
                MesCall("DollarRaiseLow");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
        }
        ndscrat = INCLENG(ndscrat);
        i = ABS(ndscrat);
        if ( i == 0 ) {
            M_free(d->where,"DollarRaiseLow");
            d->where = 0;
```

```
            d->type = DOLZERO;
            d->size = 0;
            return(0);
        }
        if ( i+2 > d->size ) {
            M_free(d->where,"DollarRaiseLow");
            d->size = i+2;
            if ( d->size < 32 ) d->size = 32;
            d->size = ((d->size+7)/8)*8;
            d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"DollarRaiseLow");
        }
        t1 = d->where; *t1++ = i+1; t2 = (WORD *)dscrat;
        while ( --i > 0 ) *t1++ = *t2++;
        *t1++ = ndscrat; *t1 = 0;
        d->type = DOLTERMS;
    }
    return(0);
}

/*
    #] DollarRaiseLow :
    #[ EvalDoLoopArg :
*/
WORD EvalDoLoopArg(PHEAD WORD *arg, WORD par)
{
    WORD num, type, *td;
    DOLLARS d;
    if ( *arg == SNUMBER ) return(arg[1]);
    if ( *arg == DOLLAREXPR2 && arg[1] < 0 ) return(-arg[1]-1);
    d = Dollars + arg[1];
#ifdef WITHPTHREADS
    {
        int nummodopt, dtype = -1;
        if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
            for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                if ( arg[1] == ModOptdollars[nummodopt].number ) break;
            }
            if ( nummodopt < NumModOptdollars ) {
                dtype = ModOptdollars[nummodopt].type;
                if ( dtype == MODLOCAL ) {
                    d = ModOptdollars[nummodopt].dstruct+AT.identity;
                }
            }
        }
    }
#endif
    if ( *arg == DOLLAREXPRESSION ) {
        if ( arg[2] != DOLLAREXPR2 ) {  /* end of chain */
endofchain:
            type = d->type;
            if ( type == DOLZERO ) {}
            else if ( type == DOLNUMBER ) {
                td = d->where;
                if ( ( td[0] != 4 ) || ( (td[1]&SPECMASK) != 0 ) || ( td[2] != 1 ) ) {
                    MLOCK(ErrorMessageLock);
                    if ( par == -1 ) {
                        MesPrint("$-variable is not a short number in print statement");
                    }
                    else {
                        MesPrint("$-variable is not a short number in do loop");
                    }
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                return( td[3] > 0 ? td[1]: -td[1] );
            }
            else {
                MLOCK(ErrorMessageLock);
                if ( par == -1 ) {
                    MesPrint("$-variable is not a number in print statement");
                }
                else {
                    MesPrint("$-variable is not a number in do loop");
                }
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            return(0);
        }
        num = EvalDoLoopArg(BHEAD arg+2,par);
    }
    else if ( *arg == DOLLAREXPR2 ) {
        if ( arg[1] < 0 ) { num = -arg[1]-1; }
        else if ( arg[2] != DOLLAREXPR2 && par == -1 ) {
            goto endofchain;
        }
        else              { num = EvalDoLoopArg(BHEAD arg+2,par); }
```

```
    }
    else {
        MLOCK(ErrorMessageLock);
        if ( par == -1 ) {
            MesPrint("Invalid $-variable in print statement");
        }
        else {
            MesPrint("Invalid $-variable in do loop");
        }
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
        return(0);
    }
    if ( num == 0 ) return(d->nfactors);
    if ( num > d->nfactors || num < 1 ) {
        MLOCK(ErrorMessageLock);
        if ( par == -1 ) {
            MesPrint("Not a valid factor number for $-variable in print statement");
        }
        else {
            MesPrint("Not a valid factor number for $-variable in do loop");
        }
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
        return(0);
    }
    if ( d->factors[num].type == DOLNUMBER )
        return(d->factors[num].value);
    else {  /* If correct, type can only be DOLNUMBER or DOLTERMS */
        MLOCK(ErrorMessageLock);
        if ( par == -1 ) {
            MesPrint("$-variable in print statement is not a number");
        }
        else {
            MesPrint("$-variable in do loop is not a number");
        }
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
        return(0);
    }
}

/*
    #] EvalDoLoopArg :
    #[ TestDoLoop :
*/

WORD TestDoLoop(PHEAD WORD *lhsbuf, WORD level)
{
    GETBIDENTITY
    WORD start,finish,incr;
    WORD *h;
    DOLLARS d;
    h = lhsbuf + 4; /* address of the start value */
    start = EvalDoLoopArg(BHEAD h,0);
    while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
        && ( h[2] == DOLLAREXPR2 ) ) h += 2;
    h += 2;
    finish = EvalDoLoopArg(BHEAD h,0);
    while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
        && ( h[2] == DOLLAREXPR2 ) ) h += 2;
    h += 2;
    incr = EvalDoLoopArg(BHEAD h,0);

    if ( ( finish == start ) || ( finish > start && incr > 0 )
    || ( finish < start && incr < 0 ) ) {}
    else { level = lhsbuf[3]; } /* skips the loop */
/*
    Put start in the dollar variable indicated by lhsbuf[2]
*/
    d = Dollars + lhsbuf[2];
#ifdef WITHPTHREADS
    {
        int nummodopt, dtype = -1;
        if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
            for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                if ( lhsbuf[2] == ModOptdollars[nummodopt].number ) break;
            }
            if ( nummodopt < NumModOptdollars ) {
                dtype = ModOptdollars[nummodopt].type;
                if ( dtype == MODLOCAL ) {
                    d = ModOptdollars[nummodopt].dstruct+AT.identity;
                }
            }
        }
    }
#endif
```

```

    if ( d->size < 32 ) {
        if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
        d->size = 32;
        d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
    }
    if ( start > 0 ) {
        d->where[0] = 4;
        d->where[1] = start;
        d->where[2] = 1;
        d->where[3] = 3;
        d->where[4] = 0;
        d->type = DOLNUMBER;
    }
    else if ( start < 0 ) {
        d->where[0] = 4;
        d->where[1] = -start;
        d->where[2] = 1;
        d->where[3] = -3;
        d->where[4] = 0;
        d->type = DOLNUMBER;
    }
    else
        d->type = DOLZERO;

    if ( d == Dollars + lhsbuf[2] ) {
        cbuf[AM.dbufnum].CanCommu[lhsbuf[2]] = 0;
        cbuf[AM.dbufnum].NumTerms[lhsbuf[2]] = 1;
        cbuf[AM.dbufnum].rhs[lhsbuf[2]] = d->where;
    }
    return(level);
}

/*
    #] TestDoLoop :
    #[ TestEndDoLoop :
*/

WORD TestEndDoLoop(PHEAD WORD *lhsbuf, WORD level)
{
    GETBIDENTITY
    WORD start,finish,incr,value;
    WORD *h;
    DOLLARS d;
    h = lhsbuf + 4; /* address of the start value */
    start = EvalDoLoopArg(BHEAD h,0);
    while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
        && ( h[2] == DOLLAREXPR2 ) ) h += 2;
    h += 2;
    finish = EvalDoLoopArg(BHEAD h,0);
    while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
        && ( h[2] == DOLLAREXPR2 ) ) h += 2;
    h += 2;
    incr = EvalDoLoopArg(BHEAD h,0);

    if ( ( finish == start ) || ( finish > start && incr > 0 )
    || ( finish < start && incr < 0 ) ) {}
    else { level = lhsbuf[3]; } /* skips the loop */
/*
    Put start in the dollar variable indicated by lhsbuf[2]
*/
    d = Dollars + lhsbuf[2];
#ifdef WITHPTHREADS
    {
        int nummodopt, dtype = -1;
        if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
            for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                if ( lhsbuf[2] == ModOptdollars[nummodopt].number ) break;
            }
            if ( nummodopt < NumModOptdollars ) {
                dtype = ModOptdollars[nummodopt].type;
                if ( dtype == MODLOCAL ) {
                    d = ModOptdollars[nummodopt].dstruct+AT.identity;
                }
            }
        }
    }
#endif
/*
    Get the value
*/
    if ( d->type == DOLZERO ) {
        value = 0;
    }
    else if ( ( d->type == DOLNUMBER || d->type == DOLTERMS )
    && ( d->where[4] == 0 ) && ( d->where[0] == 4 )
    && ( d->where[1] > 0 ) && ( d->where[2] == 1 ) ) {
```

```
        value = ( d->where[3] < 0 ) ? -d->where[1]: d->where[1];
    }
    else {
        MLOCK(ErrorMessageLock);
        MesPrint("Wrong type of object in do loop parameter");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
        return(level);
    }
    value += incr;
    if ( ( finish > start && value <= finish ) ||
         ( finish < start && value >= finish ) ||
         ( finish == start && value == finish ) ) {}
    else level = lhsbuf[3];

    if ( d->size < 32 ) {
        if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
        d->size = 32;
        d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
    }
    if ( value > 0 ) {
        d->where[0] = 4;
        d->where[1] = value;
        d->where[2] = 1;
        d->where[3] = 3;
        d->where[4] = 0;
        d->type = DOLNUMBER;
    }
    else if ( start < 0 ) {
        d->where[0] = 4;
        d->where[1] = -value;
        d->where[2] = 1;
        d->where[3] = -3;
        d->where[4] = 0;
        d->type = DOLNUMBER;
    }
    else
        d->type = DOLZERO;

    if ( d == Dollars + lhsbuf[2] ) {
        cbuf[AM.dbufnum].CanCommu[lhsbuf[2]] = 0;
        cbuf[AM.dbufnum].NumTerms[lhsbuf[2]] = 1;
        cbuf[AM.dbufnum].rhs[lhsbuf[2]] = d->where;
    }
    return(level);
}

/*
    #] TestEndDoLoop :
    #[ DollarFactorize :
*/
/* #define STEP2 */
#define STEP2

int DollarFactorize(PHEAD WORD numdollar)
{
    GETBIDENTITY
    DOLLARS d = Dollars + numdollar;
    CBUF *C, *CC;
    WORD *oldworkpointer;
    WORD *buf1, *t, *term, *buf1content, *buf2, *termextra;
    WORD *buf3, *argextra;
#ifdef STEP2
    WORD *tstop, pow, *r;
#endif
    int i, j, jj, action = 0, sign = 1;
    LONG insize, ii;
    WORD startebuf = cbuf[AT.ebufnum].numrhs;
    WORD nfactors, factorsincontent, extrafactor = 0;
    WORD oldsorttype = AR.SortType;

#ifdef WITHPTHREADS
    int nummodopt, dtype;
    dtype = -1;
    if ( AS.MultiThreaded ) {
        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
            if ( numdollar == ModOptdollars[nummodopt].number ) break;
        }
        if ( nummodopt < NumModOptdollars ) {
            dtype = ModOptdollars[nummodopt].type;
            if ( dtype == MODLOCAL ) {
                d = ModOptdollars[nummodopt].dstruct+AT.identity;
            }
            else {
                LOCK(d->pthreadslockread);
            }
        }
```

```
    }
#endif
    CleanDollarFactors(d);
#ifdef WITHPTHREADS
    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
    if ( d->type != DOLTERMS ) {    /* only one term */
        if ( d->type != DOLZERO ) d->nfactors = 1;
        return(0);
    }
    if ( d->where[d->where[0]] == 0 ) { /* only one term. easy */
    }
/*
    Here should come the code for the factorization
    We copied the routine ArgFactorize in argument.c and changed the
    memory management completely. For the actual factorization it
    calls WORD *DoFactorizeDollar(PHEAD WORD *expr) which allocates
    space for the answer. Notation:
     term,...,term,0,term,...,term,0,term,...,term,0,0

        #[ Step 1: sort the terms properly and/or make copy  --> buf1,insize
*/
    term = d->where;
    AR.SortType = SORTHIGHFIRST;
    if ( oldsorttype != AR.SortType ) {
        NewSort(BHEAD0);
        while ( *term ) {
            t = term + *term;
            if ( AN.ncmod != 0 ) {
                if ( AN.ncmod != 1 || ( (WORD)AN.cmod[0] < 0 ) ) {
                    AR.SortType = oldsorttype;
                    MLOCK(ErrorMessageLock);
                    MesPrint("Factorization modulus a number, greater than a WORD not implemented.");
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                if ( Modulus(term) ) {
                    AR.SortType = oldsorttype;
                    MLOCK(ErrorMessageLock);
                    MesCall("DollarFactorize");
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                if ( !*term) { term = t; continue; }
            }
            StoreTerm(BHEAD term);
            term = t;
        }
        AN.tryterm = 0; /* for now */
        EndSort(BHEAD (WORD *)((void *)(&buf1)),2);
        t = buf1; while ( *t ) t += *t;
        insize = t - buf1;
    }
    else {
        t = term; while ( *t ) t += *t;
        ii = insize = t - term;
        buf1 = (WORD *)Malloc1((insize+1)*sizeof(WORD),"DollarFactorize-1");
        t = buf1;
        NCOPY(t,term,ii);
        *t++ = 0;
    }
/*
        #] Step 1:
        #[ Step 2: take out the 'content'.
*/
#ifdef STEP2
    buf1content = TermMalloc("DollarContent");
    AN.tryterm = -1;
    if ( ( buf2 = TakeContent(BHEAD buf1,buf1content) ) == 0 ) {
        AN.tryterm = 0;
        TermFree(buf1content,"DollarContent");
        M_free(buf1,"DollarFactorize-1");
        AR.SortType = oldsorttype;
        MLOCK(ErrorMessageLock);
        MesCall("DollarFactorize");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
        return(1);
    }
    else if ( ( buf1content[0] == 4 ) && ( buf1content[1] == 1 ) &&
              ( buf1content[2] == 1 ) && ( buf1content[3] == 3 ) ) { /* Nothing happened */
        AN.tryterm = 0;
        if ( buf2 != buf1 ) {
            M_free(buf2,"DollarFactorize-2");
            buf2 = buf1;
        }
        factorsincontent = 0;
```

```
    }
    else {
/*
        The way we took out objects is rather brutish. We have to normalize
*/
        AN.tryterm = 0;
        if ( buf2 != buf1 ) M_free(buf1,"DollarFactorize-1");
        buf1 = buf2;
        t = buf1; while ( *t ) t += *t;
        insize = t - buf1;
/*
        Now analyse how many factors there are in the content
*/
        factorsincontent = 0;
        term = buf1content;
        tstop = term + *term;
        if ( tstop[-1] < 0 ) factorsincontent++;
        if ( ABS(tstop[-1]) == 3 && tstop[-2] == 1 && tstop[-3] == 1 ) {
            tstop -= ABS(tstop[-1]);
        }
        else {
            factorsincontent++;
            tstop -= ABS(tstop[-1]);
        }
        term++;
        while ( term < tstop ) {
            switch ( *term ) {
                case SYMBOL:
                    t = term+2; i = (term[1]-2)/2;
                    while ( i > 0 ) {
                        factorsincontent += ABS(t[1]);
                        i--; t += 2;
                    }
                    break;
                case DOTPRODUCT:
                    t = term+2; i = (term[1]-2)/3;
                    while ( i > 0 ) {
                        factorsincontent += ABS(t[2]);
                        i--; t += 3;
                    }
                    break;
                case VECTOR:
                case DELTA:
                    factorsincontent += (term[1]-2)/2;
                    break;
                case INDEX:
                    factorsincontent += term[1]-2;
                    break;
                default:
                    if ( *term >= FUNCTION ) factorsincontent++;
                    break;
            }
            term += term[1];
        }
    }
#else
    factorsincontent = 0;
    buf1content = 0;
#endif
/*
        #] Step 2: take out the 'content'.
        #[ Step 3: ConvertToPoly
                if there are objects that are not SYMBOLs,
                invoke ConvertToPoly
                We keep the original in buf1 in case there are no factors
*/
    t = buf1;
    while ( *t ) {
        if ( ( t[1] != SYMBOL ) && ( *t != (ABS(t[*t-1])+1) ) ) {
            action = 1; break;
        }
        t += *t;
    }
    if ( DetCommu(buf1) > 1 ) {
        MesPrint("Cannot factorize a $-expression with more than one noncommuting object");
        AR.SortType = oldsorttype;
        M_free(buf1,"DollarFactorize-2");
        if ( buf1content ) TermFree(buf1content,"DollarContent");
        MesCall("DollarFactorize");
        Terminate(-1);
        return(-1);
    }
    if ( action ) {
        t = buf1;
        termextra = AT.WorkPointer;
        NewSort(BHEAD0);
        NewSort(BHEAD0);
```

```
        while ( *t ) {
            if ( LocalConvertToPoly(BHEAD t,termextra,startebuf,0) < 0 ) {
getout:
                AR.SortType = oldsorttype;
                M_free(buf1,"DollarFactorize-2");
                if ( buf1content ) TermFree(buf1content,"DollarContent");
                MesCall("DollarFactorize");
                Terminate(-1);
                return(-1);
            }
            StoreTerm(BHEAD termextra);
            t += *t;
        }
        AN.tryterm = 0; /* for now */
        if ( EndSort(BHEAD (WORD *)((void *)(&buf2)),2) < 0 ) { goto getout; }
        LowerSortLevel();
        t = buf2; while ( *t > 0 ) t += *t;
    }
    else {
        buf2 = buf1;
    }
/*
        #] Step 3: ConvertToPoly
        #[ Step 4: Now the hard work.
*/
    if ( ( buf3 = poly_factorize_dollar(BHEAD buf2) ) == 0 ) {
        MesCall("DollarFactorize");
        AR.SortType = oldsorttype;
        if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-3");
        M_free(buf1,"DollarFactorize-3");
        if ( buf1content ) TermFree(buf1content,"DollarContent");
        Terminate(-1);
        return(-1);
    }
    if ( buf2 != buf1 && buf2 ) {
        M_free(buf2,"DollarFactorize-3");
        buf2 = 0;
    }
    term = buf3;
    AR.SortType = oldsorttype;
/*
    Count the factors and strip a factor -1
*/
    nfactors = 0;
    while ( *term ) {
#ifdef STEP2
        if ( *term == 4 && term[4] == 0 && term[3] == -3 && term[2] == 1
            && term[1] == 1 ) {
            WORD *tt1, *tt2, *ttstop;
            sign = -sign;
            tt1 = term; tt2 = term + *term + 1;
            ttstop = tt2;
            while ( *ttstop ) {
                while ( *ttstop ) ttstop += *ttstop;
                ttstop++;
            }
            while ( tt2 < ttstop ) *tt1++ = *tt2++;
            *tt1 = 0;
            factorsincontent++;
            extrafactor++;
        }
        else
#endif
        {
            term += *term;
            while ( *term ) { term += *term; }
            nfactors++; term++;
        }
    }
/*
    We have now:
        buf1: the original before ConvertToPoly for if only one factor
        buf3: the factored expression with nfactors factors

        #] Step 4:
        #[ Step 5: ConvertFromPoly
                If ConvertToPoly was used, use now ConvertFromPoly
                Be careful: there should be more than one factor now.
*/
#ifdef WITHPTHREADS
    if ( dtype > 0 && dtype != MODLOCAL ) { LOCK(d->pthreadslockread); }
#endif
    if ( nfactors ==  1 && extrafactor == 0 ) { /* we can use the buf1 contents */
        if ( factorsincontent == 0 ) {
            d->nfactors = 1;
#ifdef WITHPTHREADS
            if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
```

```
#endif
/*
            We used here (before 3-sep-2015) the original and did not make
            provisions for having a factors struct, figuring that all info
            is identical to the full dollar. This makes things too
            complicated at later stages.
*/
            d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR),"factors in dollar");
            term = buf1; while ( *term ) term += *term;
            d->factors[0].size = i = term - buf1;
            d->factors[0].where = t = (WORD *)Malloc1(sizeof(WORD)*(i+1),"DollarFactorize-5");
            term = buf1; NCOPY(t,term,i); *t = 0;
            AR.SortType = oldsorttype;
            M_free(buf3,"DollarFactorize-4");
            if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-4");
            M_free(buf1,"DollarFactorize-4");
            if ( buf1content ) TermFree(buf1content,"DollarContent");
            return(0);
        }
        else {
            d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR)*(nfactors+factorsincontent),"factors in dollar");
            term = buf1; while ( *term ) term += *term;
            d->factors[0].size = i = term - buf1;
            d->factors[0].where = t = (WORD *)Malloc1(sizeof(WORD)*(i+1),"DollarFactorize-5");
            term = buf1; NCOPY(t,term,i); *t = 0;
            M_free(buf3,"DollarFactorize-4");
            buf3 = 0;
            if ( buf2 != buf1 && buf2 ) {
                M_free(buf2,"DollarFactorize-4");
                buf2 = 0;
            }
        }
    }
    else if ( action ) {
        C = cbuf+AC.cbufnum;
        CC = cbuf+AT.ebufnum;
        oldworkpointer = AT.WorkPointer;
        d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR)*(nfactors+factorsincontent),"factors in dollar");
        term = buf3;
        for ( i = 0; i < nfactors; i++ ) {
            argextra = AT.WorkPointer;
            NewSort(BHEAD0);
            NewSort(BHEAD0);
            while ( *term ) {
                if ( ConvertFromPoly(BHEAD term,argextra,numxsymbol,CC->numrhs-startebuf+numxsymbol
                ,startebuf-numxsymbol,1) <= 0 ) {
                    LowerSortLevel();
getout2:            AR.SortType = oldsorttype;
                    M_free(d->factors,"factors in dollar");
                    d->factors = 0;
#ifdef WITHPTHREADS
                    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                    M_free(buf3,"DollarFactorize-4");
                    if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-4");
                    M_free(buf1,"DollarFactorize-4");
                    if ( buf1content ) TermFree(buf1content,"DollarContent");
                    return(-3);
                }
                AT.WorkPointer = argextra + *argextra;
/*
                ConvertFromPoly leaves terms with subexpressions. Hence:
*/
                if ( Generator(BHEAD argextra,C->numlhs+1) ) {
                    goto getout2;
                }
                term += *term;
            }
            term++;
            AT.WorkPointer = oldworkpointer;
            AN.tryterm = 0; /* for now */
            EndSort(BHEAD (WORD *)((void *)(&(d->factors[i].where))),2);
            LowerSortLevel();
            d->factors[i].type = DOLTERMS;
            t = d->factors[i].where;
            while ( *t ) t += *t;
            d->factors[i].size = t - d->factors[i].where;
        }
        CC->numrhs = startebuf;
    }
    else {
        C = cbuf+AC.cbufnum;
        oldworkpointer = AT.WorkPointer;
        d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR)*(nfactors+factorsincontent),"factors in dollar");
```

```
        term = buf3;
        for ( i = 0; i < nfactors; i++ ) {
            NewSort(BHEAD0);
            while ( *term ) {
                argextra = oldworkpointer;
                j = *term;
                NCOPY(argextra,term,j)
                AT.WorkPointer = argextra;
                if ( Generator(BHEAD oldworkpointer,C->numlhs+1) ) {
                    goto getout2;
                }
            }
            term++;
            AT.WorkPointer = oldworkpointer;
            AN.tryterm = 0; /* for now */
            EndSort(BHEAD (WORD *)((void *)(&(d->factors[i].where))),2);
            d->factors[i].type = DOLTERMS;
            t = d->factors[i].where;
            while ( *t ) t += *t;
            d->factors[i].size = t - d->factors[i].where;
        }
    }
    d->nfactors = nfactors + factorsincontent;
/*
        #] Step 5: ConvertFromPoly
        #[ Step 6: The factors of the content
*/
    if ( buf3 ) M_free(buf3,"DollarFactorize-5");
    if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-5");
    M_free(buf1,"DollarFactorize-5");
    j = nfactors;
#ifdef STEP2
    term = buf1content;
    tstop = term + *term;
    if ( tstop[-1] < 0 ) { tstop[-1] = -tstop[-1]; sign = -sign; }
    tstop -= tstop[-1];
    term++;
    while ( term < tstop ) {
        switch ( *term ) {
            case SYMBOL:
                t = term+2; i = (term[1]-2)/2;
                while ( i > 0 ) {
                    if ( t[1] < 0 ) { t[1] = -t[1]; pow = -1; }
                    else { pow = 1; }
                    for ( jj = 0; jj < t[1]; jj++ ) {
                        r = d->factors[j].where = (WORD *)Malloc1(9*sizeof(WORD),"factor");
                        r[0] = 8; r[1] = SYMBOL; r[2] = 4; r[3] = *t; r[4] = pow;
                        r[5] = 1; r[6] = 1; r[7] = 3; r[8] = 0;
                        d->factors[j].type = DOLTERMS;
                        d->factors[j].size = 8;
                        j++;
                    }
                    i--; t += 2;
                }
                break;
            case DOTPRODUCT:
                t = term+2; i = (term[1]-2)/3;
                while ( i > 0 ) {
                    if ( t[2] < 0 ) { t[2] = -t[2]; pow = -1; }
                    else { pow = 1; }
                    for ( jj = 0; jj < t[2]; jj++ ) {
                        r = d->factors[j].where = (WORD *)Malloc1(10*sizeof(WORD),"factor");
                        r[0] = 9; r[1] = DOTPRODUCT; r[2] = 5; r[3] = t[0]; r[4] = t[1];
                        r[5] = pow; r[6] = 1; r[7] = 1; r[8] = 3; r[9] = 0;
                        d->factors[j].type = DOLTERMS;
                        d->factors[j].size = 9;
                        j++;
                    }
                    i--; t += 3;
                }
                break;
            case VECTOR:
            case DELTA:
                t = term+2; i = (term[1]-2)/2;
                while ( i > 0 ) {
                    for ( jj = 0; jj < t[1]; jj++ ) {
                        r = d->factors[j].where = (WORD *)Malloc1(9*sizeof(WORD),"factor");
                        r[0] = 8; r[1] = *term; r[2] = 4; r[3] = *t; r[4] = t[1];
                        r[5] = 1; r[6] = 1; r[7] = 3; r[8] = 0;
                        d->factors[j].type = DOLTERMS;
                        d->factors[j].size = 8;
                        j++;
                    }
                    i--; t += 2;
                }
                break;
            case INDEX:
```

```
                t = term+2; i = term[1]-2;
                while ( i > 0 ) {
                    for ( jj = 0; jj < t[1]; jj++ ) {
                        r = d->factors[j].where = (WORD *)Malloc1(8*sizeof(WORD),"factor");
                        r[0] = 7; r[1] = *term; r[2] = 3; r[3] = *t;
                        r[4] = 1; r[5] = 1; r[6] = 3; r[7] = 0;
                        d->factors[j].type = DOLTERMS;
                        d->factors[j].size = 7;
                        j++;
                    }
                    i--; t++;
                }
                break;
            default:
                if ( *term >= FUNCTION ) {
                    r = d->factors[j].where = (WORD *)Malloc1((term[1]+5)*sizeof(WORD),"factor");
                    *r++ = d->factors[j].size = term[1]+4;
                    for ( jj = 0; jj < t[1]; jj++ ) *r++ = term[jj];
                    *r++ = 1; *r++ = 1; *r++ = 3; *r = 0;
                    j++;
                }
                break;
        }
        term += term[1];
    }
#endif
/*
        #] Step 6:
        #[ Step 7: Numerical factors
*/
#ifdef STEP2
    term = buf1content;
    tstop = term + *term;
    if ( tstop[-1] == 3 && tstop[-2] == 1 && tstop[-3] == 1 ) {}
    else if ( tstop[-1] == 3 && tstop[-2] == 1 && (UWORD)(tstop[-3]) <= MAXPOSITIVE ) {
        d->factors[j].where = 0;
        d->factors[j].size  = 0;
        d->factors[j].type  = DOLNUMBER;
        d->factors[j].value = sign*tstop[-3];
        sign = 1;
        j++;
    }
    else {
        d->factors[j].where = r = (WORD *)Malloc1((tstop[-1]+2)*sizeof(WORD),"numfactor");
        d->factors[j].size  = tstop[-1]+1;
        d->factors[j].type  = DOLTERMS;
        d->factors[j].value = 0;
        i = tstop[-1];
        t = tstop - i;
        *r++ = tstop[-1]+1;
        NCOPY(r,t,i);
        *r = 0;
        if ( sign < 0 ) {
            r = d->factors[j].where;
            while ( *r ) {
                r += *r; r[-1] = -r[-1];
            }
            sign = 1;
        }
        j++;
    }
#endif
    if ( sign < 0 ) { /* Note that this guy should come first */
        for ( jj = j; jj > 0; jj-- ) {
            d->factors[jj] = d->factors[jj-1];
        }
        d->factors[0].where = 0;
        d->factors[0].size  = 0;
        d->factors[0].type  = DOLNUMBER;
        d->factors[0].value = -1;
        j++;
    }
    d->nfactors = j;
    if ( buf1content ) TermFree(buf1content,"DollarContent");
/*
        #] Step 7:
        #[ Step 8: Sorting the factors

    There are d->nfactors factors. Look which ones have a 'where'
    Sort them by bubble sort
*/
    if ( d->nfactors > 1 ) {
        WORD ***fac, j1, j2, k, ret, *s1, *s2, *s3;
        LONG **facsize, x;
        facsize = (LONG **)Malloc1((sizeof(WORD **)+sizeof(LONG *))*d->nfactors,"SortDollarFactors");
        fac = (WORD ***)(facsize+d->nfactors);
        k = 0;
```

```
        for ( j = 0; j < d->nfactors; j++ ) {
            if ( d->factors[j].where ) {
                fac[k] = &(d->factors[j].where);
                facsize[k] = &(d->factors[j].size);
                k++;
            }
        }
        if ( k > 1 ) {
            for ( j = 1; j < k; j++ ) { /* bubble sort */
                j1 = j; j2 = j1-1;
nextj1:;
                s1 = *(fac[j1]); s2 = *(fac[j2]);
                while ( *s1 && *s2 ) {
                    if ( ( ret = CompareTerms(BHEAD s2, s1, (WORD)2) ) == 0 ) {
                        s1 += *s1; s2 += *s2;
                    }
                    else if ( ret > 0 ) goto nextj;
                    else {
exch:
                        s3 = *(fac[j1]); *(fac[j1]) = *(fac[j2]); *(fac[j2]) = s3;
                        x = *(facsize[j1]); *(facsize[j1]) = *(facsize[j2]); *(facsize[j2]) = x;
                        j1--; j2--;
                        if ( j1 > 0 ) goto nextj1;
                        goto nextj;
                    }
                }
                if ( *s1 ) goto nextj;
                if ( *s2 ) goto exch;
nextj:;
            }
        }
        M_free(facsize,"SortDollarFactors");
    }
/*
        #] Step 8:
*/
#ifdef WITHPTHREADS
    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
    return(0);
}

/*
    #] DollarFactorize :
    #[ CleanDollarFactors :
*/

void CleanDollarFactors(DOLLARS d)
{
    int i;
    if ( d->nfactors > 1 ) {
        for ( i = 0; i < d->nfactors; i++ ) {
            if ( d->factors[i].where )
                M_free(d->factors[i].where,"dollar factors");
        }
    }
    if ( d->factors ) {
        M_free(d->factors,"dollar factors");
        d->factors = 0;
    }
    d->nfactors = 0;
}

/*
    #] CleanDollarFactors :
    #[ TakeDollarContent :
*/

WORD *TakeDollarContent(PHEAD WORD *dollarbuffer, WORD **factor)
{
    WORD *remain, *t;
    int pow;
/*
    We force the sign of the first term to be positive.
*/
    t = dollarbuffer; pow = 1;
    t += *t;
    if ( t[-1] < 0 ) {
        pow = 0;
        t[-1] = -t[-1];
        while ( *t ) {
            t += *t; t[-1] = -t[-1];
        }
    }
/*
    Now the GCD of the numerators and the LCM of the denominators:
*/
```

```
    if ( AN.cmod != 0 ) {
        if ( ( *factor = MakeDollarMod(BHEAD dollarbuffer,&remain) ) == 0 ) {
            Terminate(-1);
        }
        if ( pow == 0 ) {
            (*factor)[**factor-1] = -(*factor)[**factor-1];
            (*factor)[**factor-1] += AN.cmod[0];
        }
    }
    else {
        if ( ( *factor = MakeDollarInteger(BHEAD dollarbuffer,&remain) ) == 0 ) {
            Terminate(-1);
        }
        if ( pow == 0 ) {
            (*factor)[**factor-1] = -(*factor)[**factor-1];
        }
    }
    return(remain);
}

/*
    #] TakeDollarContent :
    #[ MakeDollarInteger :
*/
WORD *MakeDollarInteger(PHEAD WORD *bufin,WORD **bufout)
{
    GETBIDENTITY
    UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
    WORD *r, *r1, *r2, *r3, *rnext, i, k, j, *oldworkpointer, *factor;
    WORD kGCD, kLCM, kGCD2, kkLCM, jLCM, jGCD;
    CBUF *C = cbuf+AC.cbufnum;

    GCDbuffer = NumberMalloc("MakeDollarInteger");
    GCDbuffer2 = NumberMalloc("MakeDollarInteger");
    LCMbuffer = NumberMalloc("MakeDollarInteger");
    LCMb = NumberMalloc("MakeDollarInteger");
    LCMc = NumberMalloc("MakeDollarInteger");
    r = bufin;
/*
    First take the first term to load up the LCM and the GCD
*/
    r2 = r + *r;
    j = r2[-1];
    r3 = r2 - ABS(j);
    k = REDLENG(j);
    if ( k < 0 ) k = -k;
    while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
    for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
    k = REDLENG(j);
    if ( k < 0 ) k = -k;
    r3 += k;
    while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
    for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
    r1 = r2;
/*
    Now go through the rest of the terms in this argument.
*/
    while ( *r1 ) {
        r2 = r1 + *r1;
        j = r2[-1];
        r3 = r2 - ABS(j);
        k = REDLENG(j);
        if ( k < 0 ) k = -k;
        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
        if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
/*
            GCD is already 1
*/
        }
        else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
            if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
                goto MakeDollarIntegerErr;
            }
            kGCD = kGCD2;
            for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
        }
        else {
            kGCD = 1; GCDbuffer[0] = 1;
        }
        k = REDLENG(j);
        if ( k < 0 ) k = -k;
        r3 += k;
        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
        if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
            for ( kLCM = 0; kLCM < k; kLCM++ )
                LCMbuffer[kLCM] = r3[kLCM];
        }
```

```
        else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
            if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
                goto MakeDollarIntegerErr;
            }
            DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
            MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
            for ( kLCM = 0; kLCM < jLCM; kLCM++ )
                LCMbuffer[kLCM] = LCMc[kLCM];
        }
        else {} /* LCM doesn't change */
        r1 = r2;
    }
/*
    Now put the factor together: GCD/LCM
*/
    r3 = (WORD *)(GCDbuffer);
    if ( kGCD == kLCM ) {
        for ( jGCD = 0; jGCD < kGCD; jGCD++ )
            r3[jGCD+kGCD] = LCMbuffer[jGCD];
        k = kGCD;
    }
    else if ( kGCD > kLCM ) {
        for ( jGCD = 0; jGCD < kLCM; jGCD++ )
            r3[jGCD+kGCD] = LCMbuffer[jGCD];
        for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
            r3[jGCD+kGCD] = 0;
        k = kGCD;
    }
    else {
        for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
            r3[jGCD] = 0;
        for ( jGCD = 0; jGCD < kLCM; jGCD++ )
            r3[jGCD+kLCM] = LCMbuffer[jGCD];
        k = kLCM;
    }
    j = 2*k+1;
/*
    Now we have to write this to factor
*/
    factor = r1 = (WORD *)Malloc1((j+2)*sizeof(WORD),"MakeDollarInteger");
    *r1++ = j+1; r2 = r3;
    for ( i = 0; i < k; i++ ) { *r1++ = *r2++; *r1++ = *r2++; }
    *r1++ = j;
    *r1 = 0;
/*
    Next we have to take the factor out from the argument.
    This cannot be done in location, because the denominator stuff can make
    coefficients longer.

    We do this via a sort because the things may be jumbled any way and we
    do not know in advance how much space we need.
*/
    NewSort(BHEAD0);
    r = bufin;
    oldworkpointer = AT.WorkPointer;
    while ( *r ) {
        rnext = r + *r;
        j = ABS(rnext[-1]);
        r3 = rnext - j;
        r2 = oldworkpointer;
        while ( r < r3 ) *r2++ = *r++;
        j = (j-1)/2;    /* reduced length. Remember, k is the other red length */
        if ( DivRat(BHEAD (UWORD *)r3,j,GCDbuffer,k,(UWORD *)r2,&i) ) {
            goto MakeDollarIntegerErr;
        }
        i = 2*i+1;
        r2 = r2 + i;
        if ( rnext[-1] < 0 ) r2[-1] = -i;
        else                 r2[-1] =  i;
        *oldworkpointer = r2-oldworkpointer;
        AT.WorkPointer = r2;
        if ( Generator(BHEAD oldworkpointer,C->numlhs) ) {
            goto MakeDollarIntegerErr;
        }
        r = rnext;
    }
    AT.WorkPointer = oldworkpointer;
    AN.tryterm = 0; /* for now */
    EndSort(BHEAD (WORD *)bufout,2);
/*
    Cleanup
*/
    NumberFree(LCMc,"MakeDollarInteger");
    NumberFree(LCMb,"MakeDollarInteger");
    NumberFree(LCMbuffer,"MakeDollarInteger");
    NumberFree(GCDbuffer2,"MakeDollarInteger");
    NumberFree(GCDbuffer,"MakeDollarInteger");
```

```
    return(factor);

MakeDollarIntegerErr:
    NumberFree(LCMc,"MakeDollarInteger");
    NumberFree(LCMb,"MakeDollarInteger");
    NumberFree(LCMbuffer,"MakeDollarInteger");
    NumberFree(GCDbuffer2,"MakeDollarInteger");
    NumberFree(GCDbuffer,"MakeDollarInteger");
    MesCall("MakeDollarInteger");
    Terminate(-1);
    return(0);
}

/*
    #] MakeDollarInteger :
    #[ MakeDollarMod :
*/
WORD *MakeDollarMod(PHEAD WORD *buffer, WORD **bufout)
{
    GETBIDENTITY
    WORD *r, *r1, x, xx, ix, ip;
    WORD *factor, *oldworkpointer;
    int i;
    CBUF *C = cbuf+AC.cbufnum;
    r = buffer;
    x = r[*r-3];
    if ( r[*r-1] < 0 ) x += AN.cmod[0];
    if ( GetModInverses(x,(WORD)(AN.cmod[0]),&ix,&ip) ) {
        Terminate(-1);
    }
    factor = (WORD *)Malloc1(5*sizeof(WORD),"MakeDollarMod");
    factor[0] = 4; factor[1] = x; factor[2] = 1; factor[3] = 3; factor[4] = 0;
/*
    Now we have to multiply all coefficients by ix.
    This does not make things longer, but we should keep to the conventions
    of MakeDollarInteger.
*/
    NewSort(BHEAD0);
    r = buffer;
    oldworkpointer = AT.WorkPointer;
    while ( *r ) {
        r1 = oldworkpointer; i = *r;
        NCOPY(r1,r,i);
        xx = r1[-3]; if ( r1[-1] < 0 ) xx += AN.cmod[0];
        r1[-1] = (WORD)((((LONG)xx)*ix) % AN.cmod[0]);
        *r1 = 0; AT.WorkPointer = r1;
        if ( Generator(BHEAD oldworkpointer,C->numlhs) ) {
            Terminate(-1);
        }
    }
    AT.WorkPointer = oldworkpointer;
    AN.tryterm = 0; /* for now */
    EndSort(BHEAD (WORD *)bufout,2);
    return(factor);
}
/*
    #] MakeDollarMod :
    #[ GetDolNum :

    Evaluates a chain of DOLLAREXPR2 into a number
*/

int GetDolNum(PHEAD WORD *t, WORD *tstop)
{
    DOLLARS d;
    WORD num, *w;
    if ( t+3 < tstop && t[3] == DOLLAREXPR2 ) {
        d = Dollars + t[2];
#ifdef WITHPTHREADS
        {
            int nummodopt, dtype;
            dtype = -1;
            if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
                for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                    if ( t[2] == ModOptdollars[nummodopt].number ) break;
                }
                if ( nummodopt < NumModOptdollars ) {
                    dtype = ModOptdollars[nummodopt].type;
                    if ( dtype == MODLOCAL ) {
                        d = ModOptdollars[nummodopt].dstruct+AT.identity;
                    }
                    else {
                        MLOCK(ErrorMessageLock);
                        MesPrint("&Illegal attempt to use $-variable %s in module %l",
                            DOLLARNAME(Dollars,t[2]),AC.CModule);
                        MUNLOCK(ErrorMessageLock);
                        Terminate(-1);
```

```
                    }
                }
            }
        }
#endif
        if ( d->factors == 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Attempt to use a factor of an unfactored $-variable");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        num = GetDolNum(BHEAD t+t[1],tstop);
        if ( num == 0 ) return(d->nfactors);
        if ( num > d->nfactors ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Attempt to use an nonexisting factor %d of a $-variable",num);
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        w = d->factors[num-1].where;
        if ( w == 0 ) return(d->factors[num-1].value);
        if ( w[0] == 4 && w[4] == 0 && w[3] == 3 && w[2] == 1 && w[1] > 0
        && w[1] < MAXPOSITIVE ) return(w[1]);
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("Illegal type of factor number of a $-variable");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
    }
    else if ( t[2] < 0 ) {
        return(-t[2]-1);
    }
    else {
        d = Dollars + t[2];
#ifdef WITHPTHREADS
        {
            int nummodopt, dtype;
            dtype = -1;
            if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
                for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                    if ( t[2] == ModOptdollars[nummodopt].number ) break;
                }
                if ( nummodopt < NumModOptdollars ) {
                    dtype = ModOptdollars[nummodopt].type;
                    if ( dtype == MODLOCAL ) {
                        d = ModOptdollars[nummodopt].dstruct+AT.identity;
                    }
                    else {
                        MLOCK(ErrorMessageLock);
                        MesPrint("&Illegal attempt to use $-variable %s in module %l",
                            DOLLARNAME(Dollars,t[2]),AC.CModule);
                        MUNLOCK(ErrorMessageLock);
                        Terminate(-1);
                    }
                }
            }
        }
#endif
        if ( d->type == DOLZERO ) return(0);
        if ( d->type == DOLTERMS || d->type == DOLNUMBER ) {
            if ( d->where[0] == 4 && d->where[4] == 0 && d->where[3] == 3
                && d->where[2] == 1 && d->where[1] > 0
                && d->where[1] < MAXPOSITIVE ) return(d->where[1]);
            MLOCK(ErrorMessageLock);
            MesPrint("Attempt to use an nonexisting factor of a $-variable");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal type of factor number of a $-variable");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    return(0);
}

/*
    #] GetDolNum :
    #[ AddPotModdollar :
*/

void AddPotModdollar(WORD numdollar)
{
    int i, n = NumPotModdollars;
    for ( i = 0; i < n; i++ ) {
        if ( numdollar == PotModdollars[i] ) break;
```

```
    }
    if ( i >= n ) {
        *(WORD *)FromList(&AC.PotModDolList) = numdollar;
    }
}

/*
    #] AddPotModdollar :
*/
```

4.25 execute.c File Reference

`#include "form3.h"`

Include dependency graph for execute.c:

Macros

- #define CURRENTBRACKET 1
- #define BRACKETCURRENTEXPR 2
- #define BRACKETOTHEREXPR 3
- #define NOBRACKETACTIVE 4

Functions

- WORD CleanExpr (WORD par)
- WORD PopVariables ()
- VOID MakeGlobal ()
- VOID TestDrop ()
- void PutInVflags (WORD nexpr)
- WORD DoExecute (WORD par, WORD skip)
- WORD PutBracket (PHEAD WORD *termin)
- VOID SpecialCleanup (PHEAD0)
- void SetMods ()
- void UnSetMods ()
- void ExchangeExpressions (int num1, int num2)
- int GetFirstBracket (WORD *term, int num)
- int GetFirstTerm (WORD *term, int num)
- int GetContent (WORD *content, int num)
- int CleanupTerm (WORD *term)
- WORD ContentMerge (PHEAD WORD *content, WORD *term)
- LONG TermsInExpression (WORD num)
- void UpdatePositions ()
- LONG CountTerms1 (PHEAD0)
- LONG TermsInBracket (PHEAD WORD *term, WORD level)

4.25.1 Detailed Description

The routines that start the execution phase of a module. It also contains the routines for placing the bracket subterm.

Definition in file execute.c.

4.25.2 Macro Definition Documentation

4.25.2.1 CURRENTBRACKET

```
#define CURRENTBRACKET 1
```

Definition at line 2395 of file execute.c.

4.25.2.2 BRACKETCURRENTEXPR

```
#define BRACKETCURRENTEXPR 2
```

Definition at line 2396 of file execute.c.

4.25.2.3 BRACKETOTHEREXPR

```
#define BRACKETOTHEREXPR 3
```

Definition at line 2397 of file execute.c.

4.25.2.4 NOBRACKETACTIVE

```
#define NOBRACKETACTIVE 4
```

Definition at line 2398 of file execute.c.

4.25.3 Function Documentation

4.25.3.1 CleanExpr()

```
WORD CleanExpr (
            WORD par )
```

Definition at line 47 of file execute.c.

4.25.3.2 PopVariables()

```
WORD PopVariables ( )
```

Definition at line 209 of file execute.c.

4.25.3.3 MakeGlobal()

```
VOID MakeGlobal ( )
```

Definition at line 376 of file execute.c.

4.25.3.4 TestDrop()

```
VOID TestDrop ( )
```

Definition at line 473 of file execute.c.

4.25.3.5 PutInVflags()

```
void PutInVflags (
            WORD nexpr )
```

Definition at line 551 of file execute.c.

4.25.3.6 DoExecute()

```
WORD DoExecute (
            WORD par,
            WORD skip )
```

Definition at line 599 of file execute.c.

4.25.3.7 PutBracket()

```
WORD PutBracket (
            PHEAD WORD * termin )
```

Definition at line 1030 of file execute.c.

4.25.3.8 SpecialCleanup()

```
VOID SpecialCleanup (
            PHEAD0  )
```

Definition at line 1539 of file execute.c.

4.25.3.9 SetMods()

```
void SetMods ( )
```

Definition at line 1553 of file execute.c.

4.25.3.10 UnSetMods()

```
void UnSetMods ( )
```

Definition at line 1571 of file execute.c.

4.25.3.11 ExchangeExpressions()

```
void ExchangeExpressions (
            int num1,
            int num2 )
```

Definition at line 1586 of file execute.c.

4.25.3.12 GetFirstBracket()

```
int GetFirstBracket (
            WORD * term,
            int num )
```

Definition at line 1660 of file execute.c.

4.25.3.13 GetFirstTerm()

```
int GetFirstTerm (
            WORD * term,
            int num )
```

Definition at line 1770 of file execute.c.

4.25.3.14 GetContent()

```
int GetContent (
            WORD * content,
            int num )
```

Definition at line 1865 of file execute.c.

4.25.3.15 CleanupTerm()

```
int CleanupTerm (
            WORD * term )
```

Definition at line 1982 of file execute.c.

4.25.3.16 ContentMerge()

```
WORD ContentMerge (
            PHEAD WORD * content,
            WORD * term )
```

Definition at line 2006 of file execute.c.

4.25.3.17 TermsInExpression()

```
LONG TermsInExpression (
            WORD num )
```

Definition at line 2226 of file execute.c.

4.25.3.18 UpdatePositions()

```
void UpdatePositions ( )
```

Definition at line 2238 of file execute.c.

4.25.3.19 CountTerms1()

```
LONG CountTerms1 (
            PHEAD0  )
```

Definition at line 2292 of file execute.c.

4.25.3.20 TermsInBracket()

```
LONG TermsInBracket (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 2400 of file execute.c.

4.26 execute.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes : execute.c
*/

#include "form3.h"

/*
    #] Includes :
    #[ DoExecute :
        #[ CleanExpr :

        par == 1 after .store or .clear
        par == 0 after .sort
*/

WORD CleanExpr(WORD par)
{
    GETIDENTITY
    WORD j, n, i;
    POSITION length;
    EXPRESSIONS e_in, e_out, e;
    int numhid = 0;
    NAMENODE *node;
    n = NumExpressions;
    j = 0;
    e_in = e_out = Expressions;
    if ( n > 0 ) { do {
        e_in->vflags &= ~( TOBEFACTORED | TOBEUNFACTORED );
        if ( par ) {
            if ( e_in->renumlists ) {
                if ( e_in->renumlists != AN.dummyrenumlist )
                        M_free(e_in->renumlists,"Renumber-lists");
                e_in->renumlists = 0;
            }
            if ( e_in->renum ) {
                M_free(e_in->renum,"Renumber"); e_in->renum = 0;
            }
        }
        if ( e_in->status == HIDDENLEXPRESSION
        || e_in->status == HIDDENGEXPRESSION ) numhid++;
        switch ( e_in->status ) {
            case SPECTATOREXPRESSION:
            case LOCALEXPRESSION:
            case HIDDENLEXPRESSION:
                if ( par ) {
                    AC.exprnames->namenode[e_in->node].type = CDELETE;
                    AC.DidClean = 1;
                    if ( e_in->status != HIDDENLEXPRESSION )
                        ClearBracketIndex(e_in-Expressions);
                    break;
                }
            case GLOBALEXPRESSION:
            case HIDDENGEXPRESSION:
                if ( par ) {
#ifdef WITHMPI
```

```
                    /*
                     * Broadcast the global expression from the master to the all workers.
                     */
                    if ( PF_BroadcastExpr(e_in, e_in->status == HIDDENGEXPRESSION ? AR.hidefile : AR.outfile) ) return -1;
                    if ( PF.me == MASTER ) {
#endif
                    e = e_in;
                    i = n-1;
                    while ( --i >= 0 ) {
                        e++;
                        if ( e_in->status == HIDDENGEXPRESSION ) {
                            if ( e->status == HIDDENGEXPRESSION
                            || e->status == HIDDENLEXPRESSION ) break;
                        }
                        else {
                            if ( e->status == GLOBALEXPRESSION
                            || e->status == LOCALEXPRESSION ) break;
                        }
                    }
#ifdef WITHMPI
                    }
                    else {
                        /*
                         * On the slaves, the broadcast expression is sitting at the end of the file.
                         */
                        e = e_in;
                        i = -1;
                    }
#endif
                    if ( i >= 0 ) {
                        DIFPOS(length,e->onfile,e_in->onfile);
                    }
                    else {
                        FILEHANDLE *f = e_in->status == HIDDENGEXPRESSION ? AR.hidefile : AR.outfile;
                        if ( f->handle < 0 ) {
                            SETBASELENGTH(length,TOLONG(f->POfull)
                             - TOLONG(f->PObuffer)
                             - BASEPOSITION(e_in->onfile));
                        }
                        else {
                            SeekFile(f->handle,&(f->filesize),SEEK_SET);
                            DIFPOS(length,f->filesize,e_in->onfile);
                        }
                    }
                    if ( ToStorage(e_in,&length) ) {
                        return(MesCall("CleanExpr"));
                    }
                    e_in->status = STOREDEXPRESSION;
                    if ( e_in->status != HIDDENGEXPRESSION )
                        ClearBracketIndex(e_in-Expressions);
                }
                /* Fall through is intentional */
            case SKIPLEXPRESSION:
            case DROPLEXPRESSION:
            case DROPHLEXPRESSION:
            case DROPGEXPRESSION:
            case DROPHGEXPRESSION:
            case STOREDEXPRESSION:
            case DROPSPECTATOREXPRESSION:
                if ( e_out != e_in ) {
                    node = AC.exprnames->namenode + e_in->node;
                    node->number = e_out - Expressions;

                    e_out->onfile = e_in->onfile;
                    e_out->size = e_in->size;
                    e_out->printflag = 0;
                    if ( par ) e_out->status = STOREDEXPRESSION;
                    else e_out->status = e_in->status;
                    e_out->name = e_in->name;
                    e_out->node = e_in->node;
                    e_out->renum = e_in->renum;
                    e_out->renumlists = e_in->renumlists;
                    e_out->counter = e_in->counter;
                    e_out->hidelevel = e_in->hidelevel;
                    e_out->inmem = e_in->inmem;
                    e_out->bracketinfo = e_in->bracketinfo;
                    e_out->newbracketinfo = e_in->newbracketinfo;
                    e_out->numdummies = e_in->numdummies;
                    e_out->numfactors = e_in->numfactors;
                    e_out->vflags = e_in->vflags;
                    e_out->sizeprototype = e_in->sizeprototype;
                }
#ifdef PARALLELCODE
                e_out->partodo = 0;
#endif
                e_out++;
```

```
                j++;
                break;
            case DROPPEDEXPRESSION:
                break;
            default:
                AC.exprnames->namenode[e_in->node].type = CDELETE;
                AC.DidClean = 1;
                break;
        }
        e_in++;
    } while ( --n > 0 ); }
    UpdateMaxSize();
    NumExpressions = j;
    if ( numhid == 0 && AR.hidefile->PObuffer ) {
        if ( AR.hidefile->handle >= 0 ) {
            CloseFile(AR.hidefile->handle);
            remove(AR.hidefile->name);
            AR.hidefile->handle = -1;
        }
        AR.hidefile->POfull =
        AR.hidefile->POfill = AR.hidefile->PObuffer;
        PUTZERO(AR.hidefile->POposition);
    }
    FlushSpectators();
    return(0);
}

/*
        #] CleanExpr :
        #[ PopVariables :

    Pops the local variables from the tables.
    The Expressions are reprocessed and their tables are compactified.

*/

WORD PopVariables()
{
    GETIDENTITY
    WORD i, j, retval;
    UBYTE *s;

    retval = CleanExpr(1);
    ResetVariables(1);

    if ( AC.DidClean ) CompactifyTree(AC.exprnames,EXPRNAMES);

    AC.CodesFlag = AM.gCodesFlag;
    AC.NamesFlag = AM.gNamesFlag;
    AC.StatsFlag = AM.gStatsFlag;
    AC.OldFactArgFlag = AM.gOldFactArgFlag;
    AC.TokensWriteFlag = AM.gTokensWriteFlag;
    AC.extrasymbols = AM.gextrasymbols;
    if ( AC.extrasym ) { M_free(AC.extrasym,"extrasym"); AC.extrasym = 0; }
    i = 1; s = AM.gextrasym; while ( *s ) { s++; i++; }
    AC.extrasym = (UBYTE *)Malloc1(i*sizeof(UBYTE),"extrasym");
    for ( j = 0; j < i; j++ ) AC.extrasym[j] = AM.gextrasym[j];
    AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers;
    AO.IndentSpace = AM.gIndentSpace;
    AC.lUnitTrace = AM.gUnitTrace;
    AC.lDefDim = AM.gDefDim;
    AC.lDefDim4 = AM.gDefDim4;
    if ( AC.halfmod ) {
        if ( AC.ncmod == AM.gncmod && AC.modmode == AM.gmodmode ) {
            j = ABS(AC.ncmod);
            while ( --j >= 0 ) {
                if ( AC.cmod[j] != AM.gcmod[j] ) break;
            }
            if ( j >= 0 ) {
                M_free(AC.halfmod,"halfmod");
                AC.halfmod = 0; AC.nhalfmod = 0;
            }
        }
        else {
            M_free(AC.halfmod,"halfmod");
            AC.halfmod = 0; AC.nhalfmod = 0;
        }
    }
    if ( AC.modinverses ) {
        if ( AC.ncmod == AM.gncmod && AC.modmode == AM.gmodmode ) {
            j = ABS(AC.ncmod);
            while ( --j >= 0 ) {
                if ( AC.cmod[j] != AM.gcmod[j] ) break;
            }
            if ( j >= 0 ) {
                M_free(AC.modinverses,"modinverses");
                AC.modinverses = 0;
```

```
            }
        }
        else {
            M_free(AC.modinverses,"modinverses");
            AC.modinverses = 0;
        }
    }
    AN.ncmod = AC.ncmod = AM.gncmod;
    AC.npowmod = AM.gnpowmod;
    AC.modmode = AM.gmodmode;
    if ( ( ( AC.modmode & INVERSETABLE ) != 0 ) && ( AC.modinverses == 0 ) )
            MakeInverses();
    AC.funpowers = AM.gfunpowers;
    AC.lPolyFun = AM.gPolyFun;
    AC.lPolyFunInv = AM.gPolyFunInv;
    AC.lPolyFunType = AM.gPolyFunType;
    AC.lPolyFunExp = AM.gPolyFunExp;
    AR.PolyFunVar = AC.lPolyFunVar = AM.gPolyFunVar;
    AC.lPolyFunPow = AM.gPolyFunPow;
    AC.parallelflag = AM.gparallelflag;
    AC.ProcessBucketSize = AC.mProcessBucketSize = AM.gProcessBucketSize;
    AC.properorderflag = AM.gproperorderflag;
    AC.ThreadBucketSize = AM.gThreadBucketSize;
    AC.ThreadStats = AM.gThreadStats;
    AC.FinalStats = AM.gFinalStats;
    AC.OldGCDflag = AM.gOldGCDflag;
    AC.WTimeStatsFlag = AM.gWTimeStatsFlag;
    AC.ThreadsFlag = AM.gThreadsFlag;
    AC.ThreadBalancing = AM.gThreadBalancing;
    AC.ThreadSortFileSynch = AM.gThreadSortFileSynch;
    AC.ProcessStats = AM.gProcessStats;
    AC.OldParallelStats = AM.gOldParallelStats;
    AC.IsFortran90 = AM.gIsFortran90;
    AC.SizeCommuteInSet = AM.gSizeCommuteInSet;
    PruneExtraSymbols(AM.gnumextrasym);

    if ( AC.Fortran90Kind ) {
        M_free(AC.Fortran90Kind,"Fortran90 Kind");
        AC.Fortran90Kind = 0;
    }
    if ( AM.gFortran90Kind ) {
        AC.Fortran90Kind = strDup1(AM.gFortran90Kind,"Fortran90 Kind");
    }
    if ( AC.ThreadsFlag && AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
    {
        UWORD *p, *m;
        p = AM.gcmod;
        m = AC.cmod;
        j = ABS(AC.ncmod);
        NCOPY(m,p,j);
        p = AM.gpowmod;
        m = AC.powmod;
        j = AC.npowmod;
        NCOPY(m,p,j);
        if ( AC.DirtPow ) {
            if ( MakeModTable() ) {
                MesPrint("===No printing in powers of generator");
            }
            AC.DirtPow = 0;
        }
    }
    {
        WORD *p, *m;
        p = AM.gUniTrace;
        m = AC.lUniTrace;
        j = 4;
        NCOPY(m,p,j);
    }
    AC.Cnumpows = AM.gCnumpows;
    AC.OutputMode = AM.gOutputMode;
    AC.OutputSpaces = AM.gOutputSpaces;
    AC.OutNumberType = AM.gOutNumberType;
    AR.SortType = AC.SortType = AM.gSortType;
    AC.ShortStatsMax = AM.gShortStatsMax;
/*
    Now we have to clean up the commutation properties
*/
    for ( i = 0; i < NumFunctions; i++ ) functions[i].flags &= ~COULDCOMMUTE;
    if ( AC.CommuteInSet ) {
        WORD *g, *gg;
        g = AC.CommuteInSet;
        while ( *g ) {
            gg = g+1; g += *g;
            while ( gg < g ) {
                if ( *gg <= GAMMASEVEN && *gg >= GAMMA ) {
                    functions[GAMMA-FUNCTION].flags |= COULDCOMMUTE;
                    functions[GAMMAI-FUNCTION].flags |= COULDCOMMUTE;
```

```
                    functions[GAMMAFIVE-FUNCTION].flags |= COULDCOMMUTE;
                    functions[GAMMASIX-FUNCTION].flags |= COULDCOMMUTE;
                    functions[GAMMASEVEN-FUNCTION].flags |= COULDCOMMUTE;
                }
                else {
                    functions[*gg-FUNCTION].flags |= COULDCOMMUTE;
                }
            }
        }
    }
/*
    Clean up the dictionaries.
*/
    for ( i = AO.NumDictionaries-1; i >= AO.gNumDictionaries; i-- ) {
        RemoveDictionary(AO.Dictionaries[i]);
        M_free(AO.Dictionaries[i],"Dictionary");
    }
    for( ; i >= 0; i-- ) {
        ShrinkDictionary(AO.Dictionaries[i]);
    }
    AO.NumDictionaries = AO.gNumDictionaries;
    return(retval);
}

/*
        #] PopVariables :
        #[ MakeGlobal :
*/

VOID MakeGlobal()
{
    WORD i, j, *pp, *mm;
    UWORD *p, *m;
    UBYTE *s;
    Globalize(0);

    AM.gCodesFlag = AC.CodesFlag;
    AM.gNamesFlag = AC.NamesFlag;
    AM.gStatsFlag = AC.StatsFlag;
    AM.gOldFactArgFlag = AC.OldFactArgFlag;
    AM.gextrasymbols = AC.extrasymbols;
    if ( AM.gextrasym ) { M_free(AM.gextrasym,"extrasym"); AM.gextrasym = 0; }
    i = 1; s = AC.extrasym; while ( *s ) { s++; i++; }
    AM.gextrasym = (UBYTE *)Malloc1(i*sizeof(UBYTE),"extrasym");
    for ( j = 0; j < i; j++ ) AM.gextrasym[j] = AC.extrasym[j];
    AM.gTokensWriteFlag= AC.TokensWriteFlag;
    AM.gNoSpacesInNumbers = AO.NoSpacesInNumbers;
    AM.gIndentSpace = AO.IndentSpace;
    AM.gUnitTrace = AC.lUnitTrace;
    AM.gDefDim = AC.lDefDim;
    AM.gDefDim4 = AC.lDefDim4;
    AM.gncmod = AC.ncmod;
    AM.gnpowmod = AC.npowmod;
    AM.gmodmode = AC.modmode;
    AM.gCnumpows = AC.Cnumpows;
    AM.gOutputMode = AC.OutputMode;
    AM.gOutputSpaces = AC.OutputSpaces;
    AM.gOutNumberType = AC.OutNumberType;
    AM.gfunpowers = AC.funpowers;
    AM.gPolyFun = AC.lPolyFun;
    AM.gPolyFunInv = AC.lPolyFunInv;
    AM.gPolyFunType = AC.lPolyFunType;
    AM.gPolyFunExp = AC.lPolyFunExp;
    AM.gPolyFunVar = AC.lPolyFunVar;
    AM.gPolyFunPow = AC.lPolyFunPow;
    AM.gparallelflag = AC.parallelflag;
    AM.gProcessBucketSize = AC.ProcessBucketSize;
    AM.gproperorderflag = AC.properorderflag;
    AM.gThreadBucketSize = AC.ThreadBucketSize;
    AM.gThreadStats = AC.ThreadStats;
    AM.gFinalStats = AC.FinalStats;
    AM.gOldGCDflag = AC.OldGCDflag;
    AM.gWTimeStatsFlag = AC.WTimeStatsFlag;
    AM.gThreadsFlag = AC.ThreadsFlag;
    AM.gThreadBalancing = AC.ThreadBalancing;
    AM.gThreadSortFileSynch = AC.ThreadSortFileSynch;
    AM.gProcessStats = AC.ProcessStats;
    AM.gOldParallelStats = AC.OldParallelStats;
    AM.gIsFortran90 = AC.IsFortran90;
    AM.gSizeCommuteInSet = AC.SizeCommuteInSet;
    AM.gnumextrasym = (cbuf+AM.sbufnum)->numrhs;
    if ( AM.gFortran90Kind ) {
        M_free(AM.gFortran90Kind,"Fortran 90 Kind");
        AM.gFortran90Kind = 0;
    }
    if ( AC.Fortran90Kind ) {
        AM.gFortran90Kind = strDup1(AC.Fortran90Kind,"Fortran 90 Kind");
```

```
    }
    p = AM.gcmod;
    m = AC.cmod;
    i = ABS(AC.ncmod);
    NCOPY(p,m,i);
    p = AM.gpowmod;
    m = AC.powmod;
    i = AC.npowmod;
    NCOPY(p,m,i);
    pp = AM.gUniTrace;
    mm = AC.lUniTrace;
    i = 4;
    NCOPY(pp,mm,i);
    AM.gSortType = AC.SortType;
    AM.gShortStatsMax = AC.ShortStatsMax;

    if ( AO.CurrentDictionary > 0 || AP.OpenDictionary > 0 ) {
        Warning("You cannot have an open or selected dictionary at a .global. Dictionary closed.");
        AP.OpenDictionary = 0;
        AO.CurrentDictionary = 0;
    }

    AO.gNumDictionaries = AO.NumDictionaries;
    for ( i = 0; i < AO.NumDictionaries; i++ ) {
        AO.Dictionaries[i]->gnumelements = AO.Dictionaries[i]->numelements;
    }
    if ( AM.NumSpectatorFiles > 0 ) {
        for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
            if ( AM.SpectatorFiles[i].name != 0 )
                    AM.SpectatorFiles[i].flags |= GLOBALSPECTATORFLAG;
        }
    }
}

/*
        #] MakeGlobal :
        #[ TestDrop :
*/

VOID TestDrop()
{
    EXPRESSIONS e;
    WORD j;
    for ( j = 0, e = Expressions; j < NumExpressions; j++, e++ ) {
        switch ( e->status ) {
            case SKIPLEXPRESSION:
                e->status = LOCALEXPRESSION;
                break;
            case UNHIDELEXPRESSION:
                e->status = LOCALEXPRESSION;
                ClearBracketIndex(j);
                e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
                break;
            case HIDELEXPRESSION:
                e->status = HIDDENLEXPRESSION;
                break;
            case SKIPGEXPRESSION:
                e->status = GLOBALEXPRESSION;
                break;
            case UNHIDEGEXPRESSION:
                e->status = GLOBALEXPRESSION;
                ClearBracketIndex(j);
                e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
                break;
            case HIDEGEXPRESSION:
                e->status = HIDDENGEXPRESSION;
                break;
            case DROPLEXPRESSION:
            case DROPGEXPRESSION:
            case DROPHLEXPRESSION:
            case DROPHGEXPRESSION:
            case DROPSPECTATOREXPRESSION:
                e->status = DROPPEDEXPRESSION;
                ClearBracketIndex(j);
                e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
                if ( e->replace >= 0 ) {
                    Expressions[e->replace].replace = REGULAREXPRESSION;
                    AC.exprnames->namenode[e->node].number = e->replace;
                    e->replace = REGULAREXPRESSION;
                }
                else {
                    AC.exprnames->namenode[e->node].type = CDELETE;
                    AC.DidClean = 1;
                }
                break;
            case LOCALEXPRESSION:
            case GLOBALEXPRESSION:
```

```
                ClearBracketIndex(j);
                e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
                break;
            case HIDDENLEXPRESSION:
            case HIDDENGEXPRESSION:
                break;
            case INTOHIDELEXPRESSION:
                ClearBracketIndex(j);
                e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
                e->status = HIDDENLEXPRESSION;
                break;
            case INTOHIDEGEXPRESSION:
                ClearBracketIndex(j);
                e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
                e->status = HIDDENGEXPRESSION;
                break;
            default:
                ClearBracketIndex(j);
                e->bracketinfo = 0;
                break;
        }
        if ( e->replace == NEWLYDEFINEDEXPRESSION ) e->replace = REGULAREXPRESSION;
    }
}

/*
        #] TestDrop :
        #[ PutInVflags :
*/

void PutInVflags(WORD nexpr)
{
    EXPRESSIONS e = Expressions + nexpr;
    POSITION *old;
    WORD *oldw;
    int i;
restart:;
    if ( AS.OldOnFile == 0 ) {
        AS.NumOldOnFile = 20;
        AS.OldOnFile = (POSITION *)Malloc1(AS.NumOldOnFile*sizeof(POSITION),"file pointers");
    }
    else if ( nexpr >= AS.NumOldOnFile ) {
        old = AS.OldOnFile;
        AS.OldOnFile = (POSITION *)Malloc1(2*AS.NumOldOnFile*sizeof(POSITION),"file pointers");
        for ( i = 0; i < AS.NumOldOnFile; i++ ) AS.OldOnFile[i] = old[i];
        AS.NumOldOnFile = 2*AS.NumOldOnFile;
        M_free(old,"process file pointers");
    }
    if ( AS.OldNumFactors == 0 ) {
        AS.NumOldNumFactors = 20;
        AS.OldNumFactors = (WORD *)Malloc1(AS.NumOldNumFactors*sizeof(WORD),"numfactors pointers");
        AS.Oldvflags = (WORD *)Malloc1(AS.NumOldNumFactors*sizeof(WORD),"vflags pointers");
    }
    else if ( nexpr >= AS.NumOldNumFactors ) {
        oldw = AS.OldNumFactors;
        AS.OldNumFactors = (WORD *)Malloc1(2*AS.NumOldNumFactors*sizeof(WORD),"numfactors pointers");
        for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.OldNumFactors[i] = oldw[i];
        M_free(oldw,"numfactors pointers");
        oldw = AS.Oldvflags;
        AS.Oldvflags = (WORD *)Malloc1(2*AS.NumOldNumFactors*sizeof(WORD),"vflags pointers");
        for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Oldvflags[i] = oldw[i];
        AS.NumOldNumFactors = 2*AS.NumOldNumFactors;
        M_free(oldw,"vflags pointers");
    }
/*
    The next is needed when we Load a .sav file with lots of expressions.
*/
    if ( nexpr >= AS.NumOldOnFile || nexpr >= AS.NumOldNumFactors ) goto restart;
    AS.OldOnFile[nexpr] = e->onfile;
    AS.OldNumFactors[nexpr] = e->numfactors;
    AS.Oldvflags[nexpr] = e->vflags;
}

/*
        #] PutInVflags :
        #[ DoExecute :
*/

WORD DoExecute(WORD par, WORD skip)
{
    GETIDENTITY
    WORD RetCode = 0;
    int i, oldmultithreaded = AS.MultiThreaded;
#ifdef PARALLELCODE
    int j;
#endif

```

```
    SpecialCleanup(BHEAD0);
    if ( skip ) goto skipexec;
    if ( AC.IfLevel > 0 ) {
        MesPrint(" %d endif statement(s) missing",AC.IfLevel);
        RetCode = 1;
    }
    if ( AC.WhileLevel > 0 ) {
        MesPrint(" %d endwhile statement(s) missing",AC.WhileLevel);
        RetCode = 1;
    }
    if ( AC.arglevel > 0 ) {
        MesPrint(" %d endargument statement(s) missing",AC.arglevel);
        RetCode = 1;
    }
    if ( AC.termlevel > 0 ) {
        MesPrint(" %d endterm statement(s) missing",AC.termlevel);
        RetCode = 1;
    }
    if ( AC.insidelevel > 0 ) {
        MesPrint(" %d endinside statement(s) missing",AC.insidelevel);
        RetCode = 1;
    }
    if ( AC.inexprlevel > 0 ) {
        MesPrint(" %d endinexpression statement(s) missing",AC.inexprlevel);
        RetCode = 1;
    }
    if ( AC.NumLabels > 0 ) {
        for ( i = 0; i < AC.NumLabels; i++ ) {
            if ( AC.Labels[i] < 0 ) {
                MesPrint(" -->Label %s missing",AC.LabelNames[i]);
                RetCode = 1;
            }
        }
    }
    if ( AC.dolooplevel > 0 ) {
        MesPrint(" %d enddo statement(s) missing",AC.dolooplevel);
        RetCode = 1;
    }
    if ( AP.OpenDictionary > 0 ) {
        MesPrint(" Dictionary %s has not been closed.",
            AO.Dictionaries[AP.OpenDictionary-1]->name);
        AP.OpenDictionary = 0;
        RetCode = 1;
    }
    if ( RetCode ) return(RetCode);
    AR.Cnumlhs = cbuf[AM.rbufnum].numlhs;

    if ( ( AS.ExecMode = par ) == GLOBALMODULE ) AS.ExecMode = 0;
#ifdef PARALLELCODE
/*
        Now check whether we have either the regular parallel flag or the
        mparallel flag set.
        Next check whether any of the expressions has partodo set.
        If any of the above we need to check what the dollar status is.
*/
    AC.partodoflag = -1;
    if ( NumPotModdollars >= 0 ) {
        for ( i = 0; i < NumExpressions; i++ ) {
            if ( Expressions[i].partodo ) { AC.partodoflag = 1; break; }
        }
    }
#ifdef WITHMPI
    if ( AC.partodoflag > 0 && PF.numtasks < 3 ) {
        AC.partodoflag = 0;
    }
#endif
    if ( AC.partodoflag > 0 || ( NumPotModdollars > 0 && AC.mparallelflag == PARALLELFLAG ) ) {
      if ( NumPotModdollars > NumModOptdollars ) {
        AC.mparallelflag |= NOPARALLEL_DOLLAR;
#ifdef WITHPTHREADS
        AS.MultiThreaded = 0;
#endif
        AC.partodoflag = 0;
      }
      else {
        for ( i = 0; i < NumPotModdollars; i++ ) {
          for ( j = 0; j < NumModOptdollars; j++ )
            if ( PotModdollars[i] == ModOptdollars[j].number ) break;
          if ( j >= NumModOptdollars ) {
            AC.mparallelflag |= NOPARALLEL_DOLLAR;
#ifdef WITHPTHREADS
            AS.MultiThreaded = 0;
#endif
            AC.partodoflag = 0;
            break;
          }
          switch ( ModOptdollars[j].type ) {
```

```
            case MODSUM:
            case MODMAX:
            case MODMIN:
            case MODLOCAL:
                break;
            default:
                AC.mparallelflag |= NOPARALLEL_DOLLAR;
                AS.MultiThreaded = 0;
                AC.partodoflag = 0;
                break;
          }
        }
      }
    }
    else if ( ( AC.mparallelflag & NOPARALLEL_USER ) != 0 ) {
#ifdef WITHPTHREADS
        AS.MultiThreaded = 0;
#endif
        AC.partodoflag = 0;
    }
    if ( AC.partodoflag == 0 ) {
        for ( i = 0; i < NumExpressions; i++ ) {
            Expressions[i].partodo = 0;
        }
    }
    else if ( AC.partodoflag == -1 ) {
        AC.partodoflag = 0;
    }
#endif
#ifdef WITHMPI
    /*
     * Check RHS expressions.
     */
    if ( AC.RhsExprInModuleFlag && (AC.mparallelflag == PARALLELFLAG || AC.partodoflag) ) {
        if (PF.rhsInParallel) {
            PF.mkSlaveInfile=1;
            if(PF.me != MASTER){
                PF.slavebuf.PObuffer=(WORD *)Malloc1(AM.ScratSize*sizeof(WORD),"PF inbuf");
                PF.slavebuf.POsize=AM.ScratSize*sizeof(WORD);
                PF.slavebuf.POfull = PF.slavebuf.POfill = PF.slavebuf.PObuffer;
                PF.slavebuf.POstop= PF.slavebuf.PObuffer+AM.ScratSize;
                PUTZERO(PF.slavebuf.POposition);
            }/*if(PF.me != MASTER)*/
        }
        else {
            AC.mparallelflag |= NOPARALLEL_RHS;
            AC.partodoflag = 0;
            for ( i = 0; i < NumExpressions; i++ ) {
                Expressions[i].partodo = 0;
            }
        }
    }
    /*
     * Set $-variables with MODSUM to zero on the slaves.
     */
    if ( (AC.mparallelflag == PARALLELFLAG || AC.partodoflag) && PF.me != MASTER ) {
        for ( i = 0; i < NumModOptdollars; i++ ) {
            if ( ModOptdollars[i].type == MODSUM ) {
                DOLLARS d = Dollars + ModOptdollars[i].number;
                d->type = DOLZERO;
                if ( d->where && d->where != &AM.dollarzero ) M_free(d->where, "old content of dollar");
                d->where = &AM.dollarzero;
                d->size = 0;
                CleanDollarFactors(d);
            }
        }
    }
#endif
    AR.SortType = AC.SortType;
#ifdef WITHMPI
    if ( PF.me == MASTER )
#endif
    {
        if ( AC.SetupFlag ) WriteSetup();
        if ( AC.NamesFlag || AC.CodesFlag ) WriteLists();
    }
    if ( par == GLOBALMODULE ) MakeGlobal();
    if ( RevertScratch() ) return(-1);
    if ( AC.ncmod ) SetMods();
/*
    Warn if the module has to run in sequential mode due to some problems.
*/
#ifdef WITHMPI
    if ( PF.me == MASTER )
#endif
    {
```

```c
        if ( !AC.ThreadsFlag || AC.mparallelflag & NOPARALLEL_USER ) {
            /* The user switched off the parallel execution explicitly. */
        }
        else if ( AC.mparallelflag & NOPARALLEL_DOLLAR ) {
            if ( AC.WarnFlag >= 2 ) {  /* HighWarning */
                int i, j, k, n;
                UBYTE *s, *s1;
                s = strDup1((UBYTE *)"","NOPARALLEL_DOLLAR s");
                n = 0;
                j = NumPotModdollars;
                for ( i = 0; i < j; i++ ) {
                    for ( k = 0; k < NumModOptdollars; k++ )
                        if ( ModOptdollars[k].number == PotModdollars[i] ) break;
                    if ( k >= NumModOptdollars ) {
                        /* global $-variable */
                        if ( n > 0 )
                            s = AddToString(s,(UBYTE *)", ",0);
                        s = AddToString(s,(UBYTE *)"$",0);
                        s = AddToString(s,DOLLARNAME(Dollars,PotModdollars[i]),0);
                        n++;
                    }
                }
                s1 = strDup1((UBYTE *)"This module is forced to run in sequential mode due to $-variable","NOPARALLEL_DOLLAR s1");
                if ( n != 1 )
                    s1 = AddToString(s1,(UBYTE *)"s",0);
                s1 = AddToString(s1,(UBYTE *)": ",0);
                s1 = AddToString(s1,s,0);
                HighWarning((char *)s1);
                M_free(s,"NOPARALLEL_DOLLAR s");
                M_free(s1,"NOPARALLEL_DOLLAR s1");
            }
        }
        else if ( AC.mparallelflag & NOPARALLEL_RHS ) {
            HighWarning("This module is forced to run in sequential mode due to RHS expression names");
        }
        else if ( AC.mparallelflag & NOPARALLEL_CONVPOLY ) {
            HighWarning("This module is forced to run in sequential mode due to conversion to extra symbols");
        }
        else if ( AC.mparallelflag & NOPARALLEL_SPECTATOR ) {
            HighWarning("This module is forced to run in sequential mode due to tospectator/copyspectator");
        }
        else if ( AC.mparallelflag & NOPARALLEL_TBLDOLLAR ) {
            HighWarning("This module is forced to run in sequential mode due to $-variable assignments in tables");
        }
        else if ( AC.mparallelflag & NOPARALLEL_NPROC ) {
            HighWarning("This module is forced to run in sequential mode because there is only one processor");
        }
    }
/*
    Now the actual execution
*/
#ifdef WITHMPI
    /*
     * Turn on AS.printflag to print runtime errors occurring on slaves.
     */
    AS.printflag = 1;
#endif
    if ( AP.preError == 0 && ( Processor() || WriteAll() ) ) RetCode = -1;
#ifdef WITHMPI
    AS.printflag = 0;
#endif
/*
    That was it. Next is cleanup.
*/
    if ( AC.ncmod ) UnSetMods();
    AS.MultiThreaded = oldmultithreaded;
    TableReset();

/*[28sep2005 mt]:*/
#ifdef WITHMPI
    /* Combine and then broadcast modified dollar variables. */
    if ( NumPotModdollars > 0 ) {
        RetCode = PF_CollectModifiedDollars();
        if ( RetCode ) return RetCode;
        RetCode = PF_BroadcastModifiedDollars();
        if ( RetCode ) return RetCode;
    }
    /* Broadcast redefined preprocessor variables. */
    if ( AC.numpfirstnum > 0 ) {
        RetCode = PF_BroadcastRedefinedPreVars();
        if ( RetCode ) return RetCode;
```

```
    }
    /* Broadcast the list of objects converted to symbols in AM.sbufnum. */
    if ( AC.topolynomialflag & TOPOLYNOMIALFLAG ) {
        RetCode = PF_BroadcastCBuf(AM.sbufnum);
        if ( RetCode ) return RetCode;
    }
    /*
     * Broadcast AR.expflags, which may be used on the slaves in the next module
     * via ZERO_ or UNCHANGED_. It also broadcasts several flags of each expression.
     */
    RetCode = PF_BroadcastExpFlags();
    if ( RetCode ) return RetCode;
    /*
     * Clean the hide file on the slaves, which was used for RHS expressions
     * broadcast from the master at the beginning of the module.
     */
    if ( PF.me != MASTER && AR.hidefile->PObuffer ) {
        if ( AR.hidefile->handle >= 0 ) {
            CloseFile(AR.hidefile->handle);
            AR.hidefile->handle = -1;
            remove(AR.hidefile->name);
        }
        AR.hidefile->POfull = AR.hidefile->POfill = AR.hidefile->PObuffer;
        PUTZERO(AR.hidefile->POposition);
    }
#endif
#ifdef WITHPTHREADS
    for ( j = 0; j < NumModOptdollars; j++ ) {
        if ( ModOptdollars[j].dstruct ) {
/*
            First clean up dollar values.
*/
            for ( i = 0; i < AM.totalnumberofthreads; i++ ) {
                if ( ModOptdollars[j].dstruct[i].size > 0 ) {
                    CleanDollarFactors(&(ModOptdollars[j].dstruct[i]));
                    M_free(ModOptdollars[j].dstruct[i].where,"Local dollar value");
                }
            }
/*
            Now clean up the whole array.
*/
            M_free(ModOptdollars[j].dstruct,"Local DOLLARS");
            ModOptdollars[j].dstruct = 0;
        }
    }
#endif
/*:[28sep2005 mt]*/

/*
    @@@@@@@@@@@@@@@
    Now follows the code to invalidate caches for all objects in the
    PotModdollars. There are NumPotModdollars of them and PotModdollars
    is an array of WORD.
*/
/*
    Cleanup:
*/
#ifdef JV_IS_WRONG
/*
    Giving back this memory gives way too much activity with Malloc1
    Better to keep it and just put the number of used objects to zero (JV)
    If you put the lijst equal to NULL, please also make maxnum = 0
*/
    if ( ModOptdollars ) M_free(ModOptdollars, "ModOptdollars pointer");
    if ( PotModdollars ) M_free(PotModdollars, "PotModdollars pointer");

    /* ModOptdollars changed to AC.ModOptDolList.lijst because AIX C compiler complained. MF 30/07/2003. */
    AC.ModOptDolList.lijst = NULL;
    /* PotModdollars changed to AC.PotModDolList.lijst because AIX C compiler complained. MF 30/07/2003. */
    AC.PotModDolList.lijst = NULL;
#endif
    NumPotModdollars = 0;
    NumModOptdollars = 0;

skipexec:
#ifdef PARALLELCODE
    AC.numpfirstnum = 0;
#endif
    AC.DidClean = 0;
    AC.PolyRatFunChanged = 0;
    TestDrop();
    if ( par == STOREMODULE || par == CLEARMODULE ) {
        ClearOptimize();
        if ( par == STOREMODULE && PopVariables() ) RetCode = -1;
        if ( AR.infile->handle >= 0 ) {
```

```
            CloseFile(AR.infile->handle);
            remove(AR.infile->name);
            AR.infile->handle = -1;
        }
        AR.infile->POfill = AR.infile->PObuffer;
        PUTZERO(AR.infile->POposition);
        AR.infile->POfull = AR.infile->PObuffer;
        if ( AR.outfile->handle >= 0 ) {
            CloseFile(AR.outfile->handle);
            remove(AR.outfile->name);
            AR.outfile->handle = -1;
        }
        AR.outfile->POfull =
        AR.outfile->POfill = AR.outfile->PObuffer;
        PUTZERO(AR.outfile->POposition);
        if ( AR.hidefile->handle >= 0 ) {
            CloseFile(AR.hidefile->handle);
            remove(AR.hidefile->name);
            AR.hidefile->handle = -1;
        }
        AR.hidefile->POfull =
        AR.hidefile->POfill = AR.hidefile->PObuffer;
        PUTZERO(AR.hidefile->POposition);
        AC.HideLevel = 0;
        if ( par == CLEARMODULE ) {
            if ( DeleteStore(0) < 0 ) {
                MesPrint("Cannot restart the storage file");
                RetCode = -1;
            }
            else RetCode = 0;
            CleanUp(1);
            ResetVariables(2);
            AM.gProcessBucketSize = AM.hProcessBucketSize;
            AM.gparallelflag = PARALLELFLAG;
            AM.gnumextrasym = AM.ggnumextrasym;
            PruneExtraSymbols(AM.ggnumextrasym);
            IniVars();
        }
        ClearSpectators(par);
    }
    else {
        if ( CleanExpr(0) ) RetCode = -1;
        if ( AC.DidClean ) CompactifyTree(AC.exprnames,EXPRNAMES);
        ResetVariables(0);
        CleanUpSort(-1);
    }
    clearcbuf(AC.cbufnum);
    if ( AC.MultiBracketBuf != 0 ) {
      for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
        if ( AC.MultiBracketBuf[i] ) {
            M_free(AC.MultiBracketBuf[i],"bracket buffer i");
            AC.MultiBracketBuf[i] = 0;
        }
      }
      AC.MultiBracketLevels = 0;
      M_free(AC.MultiBracketBuf,"multi bracket buffer");
      AC.MultiBracketBuf = 0;
    }

    return(RetCode);
}

/*
        #] DoExecute :
        #[ PutBracket :

    Routine uses the bracket info to split a term into two pieces:
    1: the part outside the bracket, and
    2: the part inside the bracket.
    These parts are separated by a subterm of type HAAKJE.
    This subterm looks like: HAAKJE,3,level
    The level is used for nestings of brackets. The print routines
    cannot handle this yet (31-Mar-1988).

    The Bracket selector is in AT.BrackBuf in the form of a regular term,
    but without coefficient.
    When AR.BracketOn < 0 we have a socalled antibracket. The main effect
    is an exchange of the inner and outer part and where the coefficient goes.

    Routine recoded to facilitate b p1,p2; etc for dotproducts and tensors
    15-oct-1991
*/

WORD PutBracket(PHEAD WORD *termin)
{
    GETBIDENTITY
    WORD *t, *t1, *b, i, j, *lastfun;
```

```
    WORD *t2, *s1, *s2;
    WORD *bStop, *bb, *bf, *tStop;
    WORD *term1,*term2, *m1, *m2, *tStopa;
    WORD *bbb = 0, *bind, *binst = 0, bwild = 0, *bss = 0, *bns = 0, bset = 0;
    term1 = AT.WorkPointer+1;
    term2 = (WORD *)(((UBYTE *)(term1)) + AM.MaxTer);
    if ( ( (WORD *)(((UBYTE *)(term2)) + AM.MaxTer) ) > AT.WorkTop ) return(MesWork());
    if ( AR.BracketOn < 0 ) {
        t2 = term1; t1 = term2;     /* AntiBracket */
    }
    else {
        t1 = term1; t2 = term2;     /* Regular bracket */
    }
    b = AT.BrackBuf; bStop = b+*b; b++;
    while ( b < bStop ) {
        if ( *b == INDEX ) { bwild = 1; bbb = b+2; binst = b + b[1]; }
        if ( *b == SETSET ) { bset = 1; bss = b+2; bns = b + b[1]; }
        b += b[1];
    }

    t = termin; tStopa = t + *t; i = *(t + *t -1); i = ABS(i);
    if ( AR.PolyFun && AT.PolyAct ) tStop = termin + AT.PolyAct;
    else tStop = tStopa - i;
    t++;
    if ( AR.BracketOn < 0 ) {
        lastfun = 0;
        while ( t < tStop && *t >= FUNCTION
            && functions[*t-FUNCTION].commute ) {
            b = AT.BrackBuf+1;
            while ( b < bStop ) {
                if ( *b == *t ) {
                    lastfun = t;
                    while ( t < tStop && *t >= FUNCTION
                        && functions[*t-FUNCTION].commute ) t += t[1];
                    goto NextNcom1;
                }
                b += b[1];
            }
            if ( bset ) {
                b = bss;
                while ( b < bns ) {
                    if ( b[1] == CFUNCTION ) { /* Set of functions */
                        SETS set = Sets+b[0]; WORD i;
                        for ( i = set->first; i < set->last; i++ ) {
                            if ( SetElements[i] == *t ) {
                                lastfun = t;
                                while ( t < tStop && *t >= FUNCTION
                                    && functions[*t-FUNCTION].commute ) t += t[1];
                                goto NextNcom1;
                            }
                        }
                    }
                    b += 2;
                }
            }
            if ( bwild && *t >= FUNCTION && functions[*t-FUNCTION].spec ) {
                s1 = t + t[1];
                s2 = t + FUNHEAD;
                while ( s2 < s1 ) {
                    bind = bbb;
                    while ( bind < binst ) {
                        if ( *bind == *s2 ) {
                            lastfun = t;
                            while ( t < tStop && *t >= FUNCTION
                                && functions[*t-FUNCTION].commute ) t += t[1];
                            goto NextNcom1;
                        }
                        bind++;
                    }
                    s2++;
                }
            }
            t += t[1];
        }
NextNcom1:
        s1 = termin + 1;
        if ( lastfun ) {
            while ( s1 < lastfun ) *t2++ = *s1++;
            while ( s1 < t ) *t1++ = *s1++;
        }
        else {
            while ( s1 < t ) *t2++ = *s1++;
        }

    }
    else {
        lastfun = t;
```

```
        while ( t < tStop && *t >= FUNCTION
            && functions[*t-FUNCTION].commute ) {
            b = AT.BrackBuf+1;
            while ( b < bStop ) {
                if ( *b == *t ) { lastfun = t + t[1]; goto NextNcom; }
                b += b[1];
            }
            if ( bset ) {
                b = bss;
                while ( b < bns ) {
                    if ( b[1] == CFUNCTION ) { /* Set of functions */
                        SETS set = Sets+b[0]; WORD i;
                        for ( i = set->first; i < set->last; i++ ) {
                            if ( SetElements[i] == *t ) {
                                lastfun = t + t[1];
                                goto NextNcom;
                            }
                        }
                    }
                    b += 2;
                }
            }
            if ( bwild && *t >= FUNCTION && functions[*t-FUNCTION].spec ) {
                s1 = t + t[1];
                s2 = t + FUNHEAD;
                while ( s2 < s1 ) {
                    bind = bbb;
                    while ( bind < binst ) {
                        if ( *bind == *s2 ) { lastfun = t + t[1]; goto NextNcom; }
                        bind++;
                    }
                    s2++;
                }
            }
NextNcom:
            t += t[1];
        }
        s1 = termin + 1;
        while ( s1 < lastfun ) *t1++ = *s1++;
        while ( s1 < t ) *t2++ = *s1++;
    }
/*
    Now we have only commuting functions left. Move the b pointer to them.
*/
    b = AT.BrackBuf + 1;
    while ( b < bStop && *b >= FUNCTION
        && ( *b < FUNCTION || functions[*b-FUNCTION].commute ) ) {
        b += b[1];
    }
    bf = b;

    while ( t < tStop && ( bf < bStop || bwild || bset ) ) {
        b = bf;
        while ( b < bStop && *b != *t ) { b += b[1]; }
        i = t[1];
        if ( *t >= FUNCTION ) { /* We are in function territory */
            if ( b < bStop && *b == *t ) goto FunBrac;
            if ( bset ) {
                b = bss;
                while ( b < bns ) {
                    if ( b[1] == CFUNCTION ) { /* Set of functions */
                        SETS set = Sets+b[0]; WORD i;
                        for ( i = set->first; i < set->last; i++ ) {
                            if ( SetElements[i] == *t ) goto FunBrac;
                        }
                    }
                    b += 2;
                }
            }
            if ( bwild && *t >= FUNCTION && functions[*t-FUNCTION].spec ) {
                s1 = t + t[1];
                s2 = t + FUNHEAD;
                while ( s2 < s1 ) {
                    bind = bbb;
                    while ( bind < binst ) {
                        if ( *bind == *s2 ) goto FunBrac;
                        bind++;
                    }
                    s2++;
                }
            }
            NCOPY(t2,t,i);
            continue;
FunBrac:    NCOPY(t1,t,i);
            continue;
        }
/*
```

```
    We have left: DELTA, INDEX, VECTOR, DOTPRODUCT, SYMBOL
*/
        if ( *t == DELTA ) {
            if ( b < bStop && *b == DELTA ) {
                b += b[1];
                NCOPY(t1,t,i);
            }
            else { NCOPY(t2,t,i); }
        }
        else if ( *t == INDEX ) {
            if ( bwild ) {
                m1 = t1; m2 = t2;
                *t1++ = *t; t1++; *t2++ = *t; t2++;
                bind = bbb;
                j = t[1] -2;
                t += 2;
                while ( --j >= 0 ) {
                    while ( *bind < *t && bind < binst ) bind++;
                    if ( *bind == *t && bind < binst ) {
                        *t1++ = *t++;
                    }
                    else if ( bset ) {
                        WORD *b3 = bss;
                        while ( b3 < bns ) {
                            if ( b3[1] == CVECTOR ) {
                                SETS set = Sets+b3[0]; WORD i;
                                for ( i = set->first; i < set->last; i++ ) {
                                    if ( SetElements[i] == *t ) {
                                        *t1++ = *t++;
                                        goto nextind;
                                    }
                                }
                            }
                            b3 += 2;
                        }
                        *t2++ = *t++;
                    }
                    else *t2++ = *t++;
nextind:;
                }
                m1[1] = WORDDIF(t1,m1);
                if ( m1[1] == 2 ) t1 = m1;
                m2[1] = WORDDIF(t2,m2);
                if ( m2[1] == 2 ) t2 = m2;
            }
            else if ( bset ) {
                m1 = t1; m2 = t2;
                *t1++ = *t; t1++; *t2++ = *t; t2++;
                j = t[1] -2;
                t += 2;
                while ( --j >= 0 ) {
                    WORD *b3 = bss;
                    while ( b3 < bns ) {
                        if ( b3[1] == CVECTOR ) {
                            SETS set = Sets+b3[0]; WORD i;
                            for ( i = set->first; i < set->last; i++ ) {
                                if ( SetElements[i] == *t ) {
                                    *t1++ = *t++;
                                    goto nextind2;
                                }
                            }
                        }
                        b3 += 2;
                    }
                    *t2++ = *t++;
nextind2:;
                }
                m1[1] = WORDDIF(t1,m1);
                if ( m1[1] == 2 ) t1 = m1;
                m2[1] = WORDDIF(t2,m2);
                if ( m2[1] == 2 ) t2 = m2;
            }
            else {
                NCOPY(t2,t,i);
            }
        }
        else if ( *t == VECTOR ) {
            if ( ( b < bStop && *b == VECTOR ) || bwild ) {
                if ( b < bStop && *b == VECTOR ) {
                    bb = b + b[1]; b += 2;
                }
                else bb = b;
                j = t[1] - 2;
                m1 = t1; m2 = t2; *t1++ = *t; *t2++ = *t; t1++; t2++; t += 2;
                while ( j > 0 ) {
                    j -= 2;
                    while ( b < bb && ( *b < *t ||
```

```
                    ( *b == *t && b[1] < t[1] ) ) ) b += 2;
                    if ( b < bb && ( *t == *b && t[1] == b[1] ) ) {
                        *t1++ = *t++; *t1++ = *t++; goto nextvec;
                    }
                    else if ( bwild ) {
                        bind = bbb;
                        while ( bind < binst ) {
                            if ( *t == *bind || t[1] == *bind ) {
                                *t1++ = *t++; *t1++ = *t++;
                                goto nextvec;
                            }
                            bind++;
                        }
                    }
                    if ( bset ) {
                        WORD *b3 = bss;
                        while ( b3 < bns ) {
                            if ( b3[1] == CVECTOR ) {
                                SETS set = Sets+b3[0]; WORD i;
                                for ( i = set->first; i < set->last; i++ ) {
                                    if ( SetElements[i] == *t ) {
                                        *t1++ = *t++; *t1++ = *t++;
                                        goto nextvec;
                                    }
                                }
                            }
                            b3 += 2;
                        }
                    }
                    *t2++ = *t++; *t2++ = *t++;
nextvec:;
                }
                m1[1] = WORDDIF(t1,m1);
                if ( m1[1] == 2 ) t1 = m1;
                m2[1] = WORDDIF(t2,m2);
                if ( m2[1] == 2 ) t2 = m2;
            }
            else if ( bset ) {
                m1 = t1; *t1++ = *t; t1++;
                m2 = t2; *t2++ = *t; t2++;
                s2 = t + i; t += 2;
                while ( t < s2 ) {
                    WORD *b3 = bss;
                    while ( b3 < bns ) {
                        if ( b3[1] == CVECTOR ) {
                            SETS set = Sets+b3[0]; WORD i;
                            for ( i = set->first; i < set->last; i++ ) {
                                if ( SetElements[i] == *t ) {
                                    *t1++ = *t++; *t1++ = *t++;
                                    goto nextvec2;
                                }
                            }
                        }
                        b3 += 2;
                    }
                    *t2++ = *t++; *t2++ = *t++;
nextvec2:;
                }
                m1[1] = WORDDIF(t1,m1);
                if ( m1[1] == 2 ) t1 = m1;
                m2[1] = WORDDIF(t2,m2);
                if ( m2[1] == 2 ) t2 = m2;
            }
            else {
                NCOPY(t2,t,i);
            }
        }
        else if ( *t == DOTPRODUCT ) {
            if ( ( b < bStop && *b == *t ) || bwild ) {
                m1 = t1; *t1++ = *t; t1++;
                m2 = t2; *t2++ = *t; t2++;
                if ( b >= bStop || *b != *t ) { bb = b; s1 = b; }
                else {
                    s1 = b + b[1]; bb = b + 2;
                }
                s2 = t + i; t += 2;
                while ( t < s2 && ( bb < s1 || bwild || bset ) ) {
                    while ( bb < s1 && ( *bb < *t ||
                    ( *bb == *t && bb[1] < t[1] ) ) ) bb += 3;
                    if ( bb < s1 && *bb == *t && bb[1] == t[1] ) {
                        *t1++ = *t++; *t1++ = *t++; *t1++ = *t++; bb += 3;
                        goto nextdot;
                    }
                    else if ( bwild ) {
                        bind = bbb;
                        while ( bind < binst ) {
                            if ( *bind == *t || *bind == t[1] ) {
```

```
                                *t1++ = *t++; *t1++ = *t++; *t1++ = *t++;
                                goto nextdot;
                            }
                            bind++;
                        }
                    }
                    if ( bset ) {
                        WORD *b3 = bss;
                        while ( b3 < bns ) {
                            if ( b3[1] == CVECTOR ) {
                                SETS set = Sets+b3[0]; WORD i;
                                for ( i = set->first; i < set->last; i++ ) {
                                    if ( SetElements[i] == *t || SetElements[i] == t[1] ) {
                                        *t1++ = *t++; *t1++ = *t++; *t1++ = *t++;
                                        goto nextdot;
                                    }
                                }
                            }
                            b3 += 2;
                        }
                    }
                    *t2++ = *t++; *t2++ = *t++; *t2++ = *t++;
nextdot:;
                }
                while ( t < s2 ) *t2++ = *t++;
                m1[1] = WORDDIF(t1,m1);
                if ( m1[1] == 2 ) t1 = m1;
                m2[1] = WORDDIF(t2,m2);
                if ( m2[1] == 2 ) t2 = m2;
            }
            else if ( bset ) {
                m1 = t1; *t1++ = *t; t1++;
                m2 = t2; *t2++ = *t; t2++;
                s2 = t + i; t += 2;
                while ( t < s2 ) {
                    WORD *b3 = bss;
                    while ( b3 < bns ) {
                        if ( b3[1] == CVECTOR ) {
                            SETS set = Sets+b3[0]; WORD i;
                            for ( i = set->first; i < set->last; i++ ) {
                                if ( SetElements[i] == *t || SetElements[i] == t[1] ) {
                                    *t1++ = *t++; *t1++ = *t++; *t1++ = *t++;
                                    goto nextdot2;
                                }
                            }
                        }
                        b3 += 2;
                    }
                    *t2++ = *t++; *t2++ = *t++; *t2++ = *t++;
nextdot2:;
                }
                m1[1] = WORDDIF(t1,m1);
                if ( m1[1] == 2 ) t1 = m1;
                m2[1] = WORDDIF(t2,m2);
                if ( m2[1] == 2 ) t2 = m2;
            }
            else { NCOPY(t2,t,i); }
        }
        else if ( *t == SYMBOL ) {
            if ( b < bStop && *b == *t ) {
                m1 = t1; *t1++ = *t; t1++;
                m2 = t2; *t2++ = *t; t2++;
                s1 = b + b[1]; bb = b+2;
                s2 = t + i; t += 2;
                while ( bb < s1 && t < s2 ) {
                    while ( bb < s1 && *bb < *t ) bb += 2;
                    if ( bb >= s1 ) {
                        if ( bset ) goto TrySymbolSet;
                        break;
                    }
                    if ( *bb == *t ) { *t1++ = *t++; *t1++ = *t++; }
                    else if ( bset ) {
                        WORD *bbb;
TrySymbolSet:
                        bbb = bss;
                        while ( bbb < bns ) {
                            if ( bbb[1] == CSYMBOL ) { /* Set of symbols */
                                SETS set = Sets+bbb[0]; WORD i;
                                for ( i = set->first; i < set->last; i++ ) {
                                    if ( SetElements[i] == *t ) {
                                        *t1++ = *t++; *t1++ = *t++;
                                        goto NextSymbol;
                                    }
                                }
                            }
                            bbb += 2;
                        }
```

```
                        *t2++ = *t++; *t2++ = *t++;
                    }
                    else { *t2++ = *t++; *t2++ = *t++; }
NextSymbol:;
                }
                while ( t < s2 ) *t2++ = *t++;
                m1[1] = WORDDIF(t1,m1);
                if ( m1[1] == 2 ) t1 = m1;
                m2[1] = WORDDIF(t2,m2);
                if ( m2[1] == 2 ) t2 = m2;
            }
            else if ( bset ) {
                WORD *bbb;
                m1 = t1; *t1++ = *t; t1++;
                m2 = t2; *t2++ = *t; t2++;
                s2 = t + i; t += 2;
                while ( t < s2 ) {
                    bbb = bss;
                    while ( bbb < bns ) {
                        if ( bbb[1] == CSYMBOL ) { /* Set of symbols */
                            SETS set = Sets+bbb[0]; WORD i;
                            for ( i = set->first; i < set->last; i++ ) {
                                if ( SetElements[i] == *t ) {
                                    *t1++ = *t++; *t1++ = *t++;
                                    goto NextSymbol2;
                                }
                            }
                        }
                        bbb += 2;
                    }
                    *t2++ = *t++; *t2++ = *t++;
NextSymbol2:;
                }
                m1[1] = WORDDIF(t1,m1);
                if ( m1[1] == 2 ) t1 = m1;
                m2[1] = WORDDIF(t2,m2);
                if ( m2[1] == 2 ) t2 = m2;
            }
            else { NCOPY(t2,t,i); }
        }
        else {
            NCOPY(t2,t,i);
        }
    }
    if ( ( i = WORDDIF(tStop,t) ) > 0 ) NCOPY(t2,t,i);
    if ( AR.BracketOn < 0 ) {
        s1 = t1; t1 = t2; t2 = s1;
    }
    do { *t2++ = *t++; } while ( t < (WORD *)tStopa );
    t = AT.WorkPointer;
    i = WORDDIF(t1,term1);
    *t++ = 4 + i + WORDDIF(t2,term2);
    t += i;
    *t++ = HAAKJE;
    *t++ = 3;
    *t++ = 0;           /* This feature won't be used for a while */
    i = WORDDIF(t2,term2);
    t1 = term2;
    if ( i > 0 ) NCOPY(t,t1,i);

    AT.WorkPointer = t;

    return(0);
}

/*
        #] PutBracket :
        #[ SpecialCleanup :
*/

VOID SpecialCleanup(PHEAD0)
{
    GETBIDENTITY
    if ( AT.previousEfactor ) M_free(AT.previousEfactor,"Efactor cache");
    AT.previousEfactor = 0;
}

/*
        #] SpecialCleanup :
        #[ SetMods :
*/

#ifndef WITHPTHREADS

void SetMods()
{
    int i, n;
```

```
    if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
    n = ABS(AN.ncmod);
    AN.cmod = (UWORD *)Malloc1(sizeof(WORD)*n,"AN.cmod");
    for ( i = 0; i < n; i++ ) AN.cmod[i] = AC.cmod[i];
}

#endif

/*
        #] SetMods :
        #[ UnSetMods :
*/

#ifndef WITHPTHREADS

void UnSetMods()
{
    if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
    AN.cmod = 0;
}

#endif

/*
        #] UnSetMods :
    #] DoExecute :
    #[ Expressions :
        #[ ExchangeExpressions :
*/

void ExchangeExpressions(int num1, int num2)
{
    GETIDENTITY
    WORD node1, node2, namesize, TMproto[SUBEXPSIZE];
    INDEXENTRY *ind;
    EXPRESSIONS e1, e2;
    LONG a;
    SBYTE *s1, *s2;
    int i;
    e1 = Expressions + num1;
    e2 = Expressions + num2;
    node1 = e1->node;
    node2 = e2->node;
    AC.exprnames->namenode[node1].number = num2;
    AC.exprnames->namenode[node2].number = num1;
    a = e1->name; e1->name = e2->name; e2->name = a;
    namesize = e1->namesize; e1->namesize = e2->namesize; e2->namesize = namesize;
    e1->node = node2;
    e2->node = node1;
    if ( e1->status == STOREDEXPRESSION ) {
/*
        Find the name in the index and replace by the new name
*/
        TMproto[0] = EXPRESSION;
        TMproto[1] = SUBEXPSIZE;
        TMproto[2] = num1;
        TMproto[3] = 1;
        { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
        AT.TMaddr = TMproto;
        ind = FindInIndex(num1,&AR.StoreData,0,0);
        s1 = (SBYTE *)(AC.exprnames->namebuffer+e1->name);
        i = e1->namesize;
        s2 = ind->name;
        NCOPY(s2,s1,i);
        *s2 = 0;
        SeekFile(AR.StoreData.Handle,&(e1->onfile),SEEK_SET);
        if ( WriteFile(AR.StoreData.Handle,(UBYTE *)ind,
        (LONG)(sizeof(INDEXENTRY))) != sizeof(INDEXENTRY) ) {
            MesPrint("File error while exchanging expressions");
            Terminate(-1);
        }
        FlushFile(AR.StoreData.Handle);
    }
    if ( e2->status == STOREDEXPRESSION ) {
/*
        Find the name in the index and replace by the new name
*/
        TMproto[0] = EXPRESSION;
        TMproto[1] = SUBEXPSIZE;
        TMproto[2] = num2;
        TMproto[3] = 1;
        { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
        AT.TMaddr = TMproto;
        ind = FindInIndex(num1,&AR.StoreData,0,0);
        s1 = (SBYTE *)(AC.exprnames->namebuffer+e2->name);
        i = e2->namesize;
        s2 = ind->name;
```

```
        NCOPY(s2,s1,i);
        *s2 = 0;
        SeekFile(AR.StoreData.Handle,&(e2->onfile),SEEK_SET);
        if ( WriteFile(AR.StoreData.Handle,(UBYTE *)ind,
        (LONG)(sizeof(INDEXENTRY))) != sizeof(INDEXENTRY) ) {
            MesPrint("File error while exchanging expressions");
            Terminate(-1);
        }
        FlushFile(AR.StoreData.Handle);
    }
}

/*
        #] ExchangeExpressions :
        #[ GetFirstBracket :
*/

int GetFirstBracket(WORD *term, int num)
{
/*
        Gets the first bracket of the expression 'num'
        Puts it in term. If no brackets the answer is one.
        Routine should be thread-safe
*/
    GETIDENTITY
    POSITION position, oldposition;
    RENUMBER renumber;
    FILEHANDLE *fi;
    WORD type, *oldcomppointer, oldonefile, numword;
    WORD *t, *tstop;

    oldcomppointer = AR.CompressPointer;
    type = Expressions[num].status;
    if ( type == STOREDEXPRESSION ) {
        WORD TMproto[SUBEXPSIZE];
        TMproto[0] = EXPRESSION;
        TMproto[1] = SUBEXPSIZE;
        TMproto[2] = num;
        TMproto[3] = 1;
        { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
        AT.TMaddr = TMproto;
        PUTZERO(position);
        if ( ( renumber = GetTable(num,&position,0) ) == 0 ) {
            MesCall("GetFirstBracket");
            SETERROR(-1)
        }
        if ( GetFromStore(term,&position,renumber,&numword,num) < 0 ) {
            MesCall("GetFirstBracket");
            SETERROR(-1)
        }
/*
#ifdef WITHPTHREADS
*/
        if ( renumber->symb.lo != AN.dummyrenumlist )
            M_free(renumber->symb.lo,"VarSpace");
        M_free(renumber,"Renumber");
/*
#endif
*/
    }
    else {          /* Active expression */
        oldonefile = AR.GetOneFile;
        if ( type == HIDDENLEXPRESSION || type == HIDDENGEXPRESSION ) {
            AR.GetOneFile = 2; fi = AR.hidefile;
        }
        else {
            AR.GetOneFile = 0; fi = AR.infile;
        }
        if ( fi->handle >= 0 ) {
            PUTZERO(oldposition);
/*
            SeekFile(fi->handle,&oldposition,SEEK_CUR);
*/
            }
        else {
            SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
        }
        position = AS.OldOnFile[num];
        if ( GetOneTerm(BHEAD term,fi,&position,1) < 0
        || ( GetOneTerm(BHEAD term,fi,&position,1) < 0 ) ) {
            MLOCK(ErrorMessageLock);
            MesCall("GetFirstBracket");
            MUNLOCK(ErrorMessageLock);
            SETERROR(-1)
        }
        if ( fi->handle >= 0 ) {
/*
```

```
            SeekFile(fi->handle,&oldposition,SEEK_SET);
            if ( ISNEGPOS(oldposition) ) {
                MLOCK(ErrorMessageLock);
                MesPrint("File error");
                MUNLOCK(ErrorMessageLock);
                SETERROR(-1)
            }
*/
        }
        else {
            fi->POfill = fi->PObuffer+BASEPOSITION(oldposition);
        }
        AR.GetOneFile = oldonefile;
    }
    AR.CompressPointer = oldcomppointer;
    if ( *term ) {
        tstop = term + *term; tstop -= ABS(tstop[-1]);
        t = term + 1;
        while ( t < tstop ) {
            if ( *t == HAAKJE ) break;
            t += t[1];
        }
        if ( t >= tstop ) {
            term[0] = 4; term[1] = 1; term[2] = 1; term[3] = 3;
        }
        else {
            *t++ = 1; *t++ = 1; *t++ = 3; *term = t - term;
        }
    }
    else {
        term[0] = 4; term[1] = 1; term[2] = 1; term[3] = 3;
    }
    return(*term);
}

/*
        #] GetFirstBracket :
        #[ GetFirstTerm :
*/

int GetFirstTerm(WORD *term, int num)
{
/*
        Gets the first term of the expression 'num'
        Puts it in term.
        Routine should be thread-safe
*/
    GETIDENTITY
    POSITION position, oldposition;
    RENUMBER renumber;
    FILEHANDLE *fi;
    WORD type, *oldcomppointer, oldonefile, numword;

    oldcomppointer = AR.CompressPointer;
    type = Expressions[num].status;
    if ( type == STOREDEXPRESSION ) {
        WORD TMproto[SUBEXPSIZE];
        TMproto[0] = EXPRESSION;
        TMproto[1] = SUBEXPSIZE;
        TMproto[2] = num;
        TMproto[3] = 1;
        { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
        AT.TMaddr = TMproto;
        PUTZERO(position);
        if ( ( renumber = GetTable(num,&position,0) ) == 0 ) {
            MesCall("GetFirstTerm");
            SETERROR(-1)
        }
        if ( GetFromStore(term,&position,renumber,&numword,num) < 0 ) {
            MesCall("GetFirstTerm");
            SETERROR(-1)
        }
/*
#ifdef WITHPTHREADS
*/
        if ( renumber->symb.lo != AN.dummyrenumlist )
            M_free(renumber->symb.lo,"VarSpace");
        M_free(renumber,"Renumber");
/*
#endif
*/
    }
    else {          /* Active expression */
        oldonefile = AR.GetOneFile;
        if ( type == HIDDENLEXPRESSION || type == HIDDENGEXPRESSION ) {
            AR.GetOneFile = 2; fi = AR.hidefile;
        }
```

```
        else {
            AR.GetOneFile = 0;
            if ( Expressions[num].replace == NEWLYDEFINEDEXPRESSION )
                 fi = AR.outfile;
            else fi = AR.infile;
        }
        if ( fi->handle >= 0 ) {
            PUTZERO(oldposition);
/*
            SeekFile(fi->handle,&oldposition,SEEK_CUR);
*/
            }
        else {
            SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
        }
        position = AS.OldOnFile[num];
        if ( GetOneTerm(BHEAD term,fi,&position,1) < 0
        || ( GetOneTerm(BHEAD term,fi,&position,1) < 0 ) ) {
            MLOCK(ErrorMessageLock);
            MesCall("GetFirstTerm");
            MUNLOCK(ErrorMessageLock);
            SETERROR(-1)
        }
        if ( fi->handle >= 0 ) {
/*
            SeekFile(fi->handle,&oldposition,SEEK_SET);
            if ( ISNEGPOS(oldposition) ) {
                MLOCK(ErrorMessageLock);
                MesPrint("File error");
                MUNLOCK(ErrorMessageLock);
                SETERROR(-1)
            }
*/
        }
        else {
            fi->POfill = fi->PObuffer+BASEPOSITION(oldposition);
        }
        AR.GetOneFile = oldonefile;
    }
    AR.CompressPointer = oldcomppointer;
    return(*term);
}

/*
        #] GetFirstTerm :
        #[ GetContent :
*/

int GetContent(WORD *content, int num)
{
/*
        Gets the content of the expression 'num'
        Puts it in content.
        Routine should be thread-safe
        The content is defined as the term that will make the expression 'num'
        with integer coefficients, no GCD and all common factors taken out,
        all negative powers removed when we divide the expression by this
        content.
*/
    GETIDENTITY
    POSITION position, oldposition;
    RENUMBER renumber;
    FILEHANDLE *fi;
    WORD type, *oldcomppointer, oldonefile, numword, *term, i;
    WORD *cbuffer = TermMalloc("GetContent");
    WORD *oldworkpointer = AT.WorkPointer;

    oldcomppointer = AR.CompressPointer;
    type = Expressions[num].status;
    if ( type == STOREDEXPRESSION ) {
        WORD TMproto[SUBEXPSIZE];
        TMproto[0] = EXPRESSION;
        TMproto[1] = SUBEXPSIZE;
        TMproto[2] = num;
        TMproto[3] = 1;
        { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
        AT.TMaddr = TMproto;
        PUTZERO(position);
        if ( ( renumber = GetTable(num,&position,0) ) == 0 ) goto CalledFrom;
        if ( GetFromStore(cbuffer,&position,renumber,&numword,num) < 0 ) goto CalledFrom;
        for(;;) {
            term = oldworkpointer;
            AR.CompressPointer = oldcomppointer;
            if ( GetFromStore(term,&position,renumber,&numword,num) < 0 ) goto CalledFrom;
            if ( *term == 0 ) break;
/*
            'merge' the two terms
```

```
*/
            if ( ContentMerge(BHEAD cbuffer,term) < 0 ) goto CalledFrom;
        }
/*
#ifdef WITHPTHREADS
*/
        if ( renumber->symb.lo != AN.dummyrenumlist )
            M_free(renumber->symb.lo,"VarSpace");
        M_free(renumber,"Renumber");
/*
#endif
*/
    }
    else {          /* Active expression */
        oldonefile = AR.GetOneFile;
        if ( type == HIDDENLEXPRESSION || type == HIDDENGEXPRESSION ) {
            AR.GetOneFile = 2; fi = AR.hidefile;
        }
        else {
            AR.GetOneFile = 0;
            if ( Expressions[num].replace == NEWLYDEFINEDEXPRESSION )
                fi = AR.outfile;
            else fi = AR.infile;
        }
        if ( fi->handle >= 0 ) {
            PUTZERO(oldposition);
/*
            SeekFile(fi->handle,&oldposition,SEEK_CUR);
*/
            }
        else {
            SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
        }
        position = AS.OldOnFile[num];
        if ( GetOneTerm(BHEAD cbuffer,fi,&position,1) < 0 ) goto CalledFrom;
        AR.CompressPointer = oldcomppointer;
        if ( GetOneTerm(BHEAD cbuffer,fi,&position,1) < 0 ) goto CalledFrom;
/*
        Now go through the terms. For each term we have to test whether
        what is in cbuffer is also in that term. If not, we have to remove
        it from cbuffer. Additionally we have to accumulate the GCD of the
        numerators and the LCM of the denominators. This is all done in the
        routine ContentMerge.
*/
        for(;;) {
            term = oldworkpointer;
            AR.CompressPointer = oldcomppointer;
            if ( GetOneTerm(BHEAD term,fi,&position,1) < 0 ) goto CalledFrom;
            if ( *term == 0 ) break;
/*
            'merge' the two terms
*/
            if ( ContentMerge(BHEAD cbuffer,term) < 0 ) goto CalledFrom;
        }
        if ( fi->handle < 0 ) {
            fi->POfill = fi->PObuffer+BASEPOSITION(oldposition);
        }
        AR.GetOneFile = oldonefile;
    }
    AR.CompressPointer = oldcomppointer;
    for ( i = 0; i < *cbuffer; i++ ) content[i] = cbuffer[i];
    TermFree(cbuffer,"GetContent");
    AT.WorkPointer = oldworkpointer;
    return(*content);
CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("GetContent");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] GetContent :
        #[ CleanupTerm :

        Removes noncommuting objects from the term
*/

int CleanupTerm(WORD *term)
{
    WORD *tstop, *t, *tfill, *tt;
    GETSTOP(term,tstop);
    t = term+1;
    while ( t < tstop ) {
        if ( *t >= FUNCTION && ( functions[*t-FUNCTION].commute || *t == DENOMINATOR ) ) {
            tfill = t; tt = t + t[1]; tstop = term + *term;
            while ( tt < tstop ) *tfill++ = *tt++;
```

```
            *term = tfill - term;
            tstop -= ABS(tfill[-1]);
        }
        else {
            t += t[1];
        }
    }
    return(0);
}

/*
        #] CleanupTerm :
        #[ ContentMerge :
*/

WORD ContentMerge(PHEAD WORD *content, WORD *term)
{
    GETBIDENTITY
    WORD *cstop, csize, crsize, sign = 1, numsize, densize, i, tnsize, tdsize;
    UWORD *num, *den, *tnum, *tden;
    WORD *outfill, *outb = TermMalloc("ContentMerge"), *ct;
    WORD *t, *tstop, tsize, trsize, *told;
    WORD *t1, *t2, *c1, *c2, i1, i2, *out1;
    cstop = content + *content;
    csize = cstop[-1];
    if ( csize < 0 ) { sign = -sign; csize = -csize; }
    cstop -= csize;
    numsize = densize = crsize = (csize-1)/2;
    num = NumberMalloc("ContentMerge");
    den = NumberMalloc("ContentMerge");
    for ( i = 0; i < numsize; i++ ) num[i] = (UWORD)(cstop[i]);
    for ( i = 0; i < densize; i++ ) den[i] = (UWORD)(cstop[i+crsize]);
    while ( num[numsize-1] == 0 ) numsize--;
    while ( den[densize-1] == 0 ) densize--;
/*
    First we do the coefficient
*/
    tstop = term + *term;
    tsize = tstop[-1];
    if ( tsize < 0 ) tsize = -tsize;
/*  else { sign = 1; } */
    tstop = tstop - tsize;
    tnsize = tdsize = trsize = (tsize-1)/2;
    tnum = (UWORD *)tstop; tden = (UWORD *)(tstop + trsize);
    while ( tnum[tnsize-1] == 0 ) tnsize--;
    while ( tden[tdsize-1] == 0 ) tdsize--;
    GcdLong(BHEAD num, numsize, tnum, tnsize, num, &numsize);
    if ( LcmLong(BHEAD den, densize, tden, tdsize, den, &densize) ) goto CalledFrom;
    outfill = outb + 1;
    ct = content + 1;
    t = term + 1;
    while ( ct < cstop ) {
        switch ( *ct ) {
            case SYMBOL:
                t = term+1;
                while ( t < tstop && *t != *ct ) t += t[1];
                if ( t >= tstop ) break;
                t1 = t+2; t2 = t+t[1];
                c1 = ct+2; c2 = ct+ct[1];
                out1 = outfill; *outfill++ = *ct; outfill++;
                while ( c1 < c2 && t1 < t2 ) {
                    if ( *c1 == *t1 ) {
                        if ( t1[1] <= c1[1] ) {
                            *outfill++ = *t1++; *outfill++ = *t1++;
                            c1 += 2;
                        }
                        else {
                            *outfill++ = *c1++; *outfill++ = *c1++;
                            t1 += 2;
                        }
                    }
                    else if ( *c1 < *t1 ) {
                        if ( c1[1] < 0 ) {
                            *outfill++ = *c1++; *outfill++ = *c1++;
                        }
                        else { c1 += 2; }
                    }
                    else {
                        if ( t1[1] < 0 ) {
                            *outfill++ = *t1++; *outfill++ = *t1++;
                        }
                        else t1 += 2;
                    }
                }
                while ( c1 < c2 ) {
                    if ( c1[1] < 0 ) { *outfill++ = c1[0]; *outfill++ = c1[1]; }
                    c1 += 2;
```

```
                }
                while ( t1 < t2 ) {
                    if ( t1[1] < 0 ) { *outfill++ = t1[0]; *outfill++ = t1[1]; }
                    t1 += 2;
                }
                out1[1] = outfill - out1;
                if ( out1[1] == 2 ) outfill = out1;
                break;
            case DOTPRODUCT:
                t = term+1;
                while ( t < tstop && *t != *ct ) t += t[1];
                if ( t >= tstop ) break;
                t1 = t+2; t2 = t+t[1];
                c1 = ct+2; c2 = ct+ct[1];
                out1 = outfill; *outfill++ = *ct; outfill++;
                while ( c1 < c2 && t1 < t2 ) {
                    if ( *c1 == *t1 && c1[1] == t1[1] ) {
                        if ( t1[2] <= c1[2] ) {
                            *outfill++ = *t1++; *outfill++ = *t1++; *outfill++ = *t1++;
                            c1 += 3;
                        }
                        else {
                            *outfill++ = *c1++; *outfill++ = *c1++; *outfill++ = *c1++;
                            t1 += 3;
                        }
                    }
                    else if ( *c1 < *t1 || ( *c1 == *t1 && c1[1] < t1[1] ) ) {
                        if ( c1[2] < 0 ) {
                            *outfill++ = *c1++; *outfill++ = *c1++; *outfill++ = *c1++;
                        }
                        else { c1 += 3; }
                    }
                    else {
                        if ( t1[2] < 0 ) {
                            *outfill++ = *t1++; *outfill++ = *t1++; *outfill++ = *t1++;
                        }
                        else t1 += 3;
                    }
                }
                while ( c1 < c2 ) {
                    if ( c1[2] < 0 ) { *outfill++ = c1[0]; *outfill++ = c1[1]; *outfill++ = c1[1]; }
                    c1 += 3;
                }
                while ( t1 < t2 ) {
                    if ( t1[2] < 0 ) { *outfill++ = t1[0]; *outfill++ = t1[1]; *outfill++ = t1[1]; }
                    t1 += 3;
                }
                out1[1] = outfill - out1;
                if ( out1[1] == 2 ) outfill = out1;
                break;
            case INDEX:
                t = term+1;
                while ( t < tstop && *t != *ct ) t += t[1];
                if ( t >= tstop ) break;
                t1 = t+2; t2 = t+t[1];
                c1 = ct+2; c2 = ct+ct[1];
                out1 = outfill; *outfill++ = *ct; outfill++;
                while ( c1 < c2 && t1 < t2 ) {
                    if ( *c1 == *t1 ) {
                        *outfill++ = *c1++;
                        t1 += 1;
                    }
                    else if ( *c1 < *t1 ) { c1 += 1; }
                    else { t1 += 1; }
                }
                out1[1] = outfill - out1;
                if ( out1[1] == 2 ) outfill = out1;
                break;
            case VECTOR:
            case DELTA:
                t = term+1;
                while ( t < tstop && *t != *ct ) t += t[1];
                if ( t >= tstop ) break;
                t1 = t+2; t2 = t+t[1];
                c1 = ct+2; c2 = ct+ct[1];
                out1 = outfill; *outfill++ = *ct; outfill++;
                while ( c1 < c2 && t1 < t2 ) {
                    if ( *c1 == *t1 && c1[1] && t1[1] ) {
                        *outfill++ = *c1++; *outfill++ = *c1++;
                        t1 += 2;
                    }
                    else if ( *c1 < *t1 || ( *c1 == *t1 && c1[1] < t1[1] ) ) {
                        c1 += 2;
                    }
                    else {
                        t1 += 2;
                    }
```

```
                }
                out1[1] = outfill - out1;
                if ( out1[1] == 2 ) outfill = out1;
                break;
            case GAMMA:
            default:            /* Functions */
                told = t;
                while ( *t < *ct && t < tstop ) t += t[1];
                if ( t >= tstop ) { t = told; }
                else {
                    t1 = t; t2 = ct; i1 = t1[1]; i2 = t2[1];
                    if ( i1 != i2 ) { t = told; }
                    else {
                        while ( i1 > 0 ) {
                            if ( *t1 != *t2 ) break;
                            t1++; t2++; i1--;
                        }
                        if ( i1 == 0 ) {
                            for ( i = 0; i < i2; i++ ) { *outfill++ = *t++; }
                        }
                        else { t = told; }
                    }
                }
                break;
        }
        ct += ct[1];
    }
/*
    Now put the coefficient back.
*/
    if ( numsize < densize ) {
        for ( i = numsize; i < densize; i++ ) num[i] = 0;
        numsize = densize;
    }
    else if ( densize < numsize ) {
        for ( i = densize; i < numsize; i++ ) den[i] = 0;
        densize = numsize;
    }
    for ( i = 0; i < numsize; i++ ) *outfill++ = num[i];
    for ( i = 0; i < densize; i++ ) *outfill++ = den[i];
    csize = numsize+densize+1;
    if ( sign < 0 ) csize = -csize;
    *outfill++ = csize;
    *outb = outfill-outb;
    NumberFree(den,"ContentMerge");
    NumberFree(num,"ContentMerge");
    for ( i = 0; i < *outb; i++ ) content[i] = outb[i];
    TermFree(outb,"ContentMerge");
    return(*content);
CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("GetContent");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] ContentMerge :
        #[ TermsInExpression :
*/

LONG TermsInExpression(WORD num)
{
    LONG x = Expressions[num].counter;
    if ( x >= 0 ) return(x);
    return(-1);
}

/*
        #] TermsInExpression :
        #[ UpdatePositions :
*/

void UpdatePositions()
{
    EXPRESSIONS e = Expressions;
    POSITION *old;
    WORD *oldw;
    int i;
    if ( NumExpressions > 0 &&
         ( AS.OldOnFile == 0 || AS.NumOldOnFile < NumExpressions ) ) {
        if ( AS.OldOnFile ) {
            old = AS.OldOnFile;
            AS.OldOnFile = (POSITION *)Malloc1(NumExpressions*sizeof(POSITION),"file pointers");
            for ( i = 0; i < AS.NumOldOnFile; i++ ) AS.OldOnFile[i] = old[i];
            AS.NumOldOnFile = NumExpressions;
            M_free(old,"process file pointers");
```

```
        }
        else {
            AS.OldOnFile = (POSITION *)Malloc1(NumExpressions*sizeof(POSITION),"file pointers");
            AS.NumOldOnFile = NumExpressions;
        }
    }
    if ( NumExpressions > 0 &&
         ( AS.OldNumFactors == 0 || AS.NumOldNumFactors < NumExpressions ) ) {
        if ( AS.OldNumFactors ) {
            oldw = AS.OldNumFactors;
            AS.OldNumFactors = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"numfactors pointers");
            for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.OldNumFactors[i] = oldw[i];
            M_free(oldw,"numfactors pointers");
            oldw = AS.Oldvflags;
            AS.Oldvflags = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"vflags pointers");
            for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Oldvflags[i] = oldw[i];
            AS.NumOldNumFactors = NumExpressions;
            M_free(oldw,"vflags pointers");
        }
        else {
            AS.OldNumFactors = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"numfactors pointers");
            AS.Oldvflags = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"vflags pointers");
            AS.NumOldNumFactors = NumExpressions;
        }
    }
    for ( i = 0; i < NumExpressions; i++ ) {
        AS.OldOnFile[i] = e[i].onfile;
        AS.OldNumFactors[i] = e[i].numfactors;
        AS.Oldvflags[i] = e[i].vflags;
    }
}

/*
        #] UpdatePositions :
        #[ CountTerms1 :        LONG CountTerms1()

        Counts the terms in the current deferred bracket
        Is mainly an adaptation of the routine Deferred in proces.c
*/

LONG CountTerms1(PHEAD0)
{
    GETBIDENTITY
    POSITION oldposition, startposition;
    WORD *t, *m, *mstop, decr, i, *oldwork, retval;
    WORD *oldipointer = AR.CompressPointer;
    WORD oldGetOneFile = AR.GetOneFile, olddeferflag = AR.DeferFlag;
    LONG numterms = 0;
    AR.GetOneFile = 1;
    oldwork = AT.WorkPointer;
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    AR.DeferFlag = 0;
    startposition = AR.DefPosition;
/*
        Store old position
*/
    if ( AR.infile->handle >= 0 ) {
        PUTZERO(oldposition);
/*
        SeekFile(AR.infile->handle,&oldposition,SEEK_CUR);
*/
    }
    else {
        SETBASEPOSITION(oldposition,AR.infile->POfill-AR.infile->PObuffer);
        AR.infile->POfill = (WORD *)((UBYTE *)(AR.infile->PObuffer)
                    +BASEPOSITION(startposition));
    }
/*
        Look in the CompressBuffer where the bracket contents start
*/
    t = m = AR.CompressBuffer;
    t += *t;
    mstop = t - ABS(t[-1]);
    m++;
    while ( *m != HAAKJE && m < mstop ) m += m[1];
    if ( m >= mstop ) { /* No deferred action! */
        numterms = 1;
        AR.DeferFlag = olddeferflag;
        AT.WorkPointer = oldwork;
        AR.GetOneFile = oldGetOneFile;
        return(numterms);
    }
    mstop = m + m[1];
    decr = WORDDIF(mstop,AR.CompressBuffer)-1;

    m = AR.CompressBuffer;
    t = AR.CompressPointer;
```

```
    i = *m;
    NCOPY(t,m,i);
    AR.TePos = 0;
    AN.TeSuOut = 0;
/*
        Status:
        First bracket content starts at mstop.
        Next term starts at startposition.
        Decompression information is in AR.CompressPointer.
        The outside of the bracket runs from AR.CompressBuffer+1 to mstop.
*/
    AR.CompressPointer = oldipointer;
    for(;;) {
        numterms++;
        retval = GetOneTerm(BHEAD AT.WorkPointer,AR.infile,&startposition,0);
        if ( retval >= 0 ) AR.CompressPointer = oldipointer;
        if ( retval <= 0 ) break;
        t = AR.CompressPointer;
        if ( *t < (1 + decr + ABS(*(t+*t-1))) ) break;
        t++;
        m = AR.CompressBuffer+1;
        while ( m < mstop ) {
            if ( *m != *t ) goto Thatsit;
            m++; t++;
        }
    }
Thatsit:;
/*
        Finished. Reposition the file, restore information and return.
*/
    AT.WorkPointer = oldwork;
    if ( AR.infile->handle >= 0 ) {
/*
        SeekFile(AR.infile->handle,&oldposition,SEEK_SET);
*/
    }
    else {
        AR.infile->POfill = AR.infile->PObuffer + BASEPOSITION(oldposition);
    }
    AR.DeferFlag = olddeferflag;
    AR.GetOneFile = oldGetOneFile;
    return(numterms);
}

/*
        #] CountTerms1 :
        #[ TermsInBracket :     LONG TermsInBracket(term,level)

    The function TermsInBracket_()
    Syntax:
        TermsInBracket_() : The current bracket in a Keep Brackets
        TermsInBracket_(bracket) : This bracket in the current expression
        TermsInBracket_(expression,bracket) : This bracket in the given expression
    All other specifications don't have any effect.
*/

#define CURRENTBRACKET 1
#define BRACKETCURRENTEXPR 2
#define BRACKETOTHEREXPR 3
#define NOBRACKETACTIVE 4

LONG TermsInBracket(PHEAD WORD *term, WORD level)
{
    WORD *t, *tstop, *b, *tt, *n1, *n2;
    int type = 0, i, num;
    LONG numterms = 0;
    WORD *bracketbuffer = AT.WorkPointer;
    t = term; GETSTOP(t,tstop);
    t++; b = bracketbuffer;
    while ( t < tstop ) {
        if ( *t != TERMSINBRACKET ) { t += t[1]; continue; }
        if ( t[1] == FUNHEAD || (
             t[1] == FUNHEAD+2
            && t[FUNHEAD] == -SNUMBER
            && t[FUNHEAD+1] == 0
         ) ) {
            if ( AC.ComDefer == 0 ) {
                type = NOBRACKETACTIVE;
            }
            else {
                type = CURRENTBRACKET;
            }
            *b = 0;
            break;
        }
        if ( t[FUNHEAD] == -EXPRESSION ) {
            if ( t[FUNHEAD+2] < 0 ) {
```

```
                if ( ( t[FUNHEAD+2] <= -FUNCTION ) && ( t[1] == FUNHEAD+3 ) ) {
                    type = BRACKETOTHEREXPR;
                    *b++ = FUNHEAD+4; *b++ = -t[FUNHEAD+2]; *b++ = FUNHEAD;
                    for ( i = 2; i < FUNHEAD; i++ ) *b++ = 0;
                    *b++ = 1; *b++ = 1; *b++ = 3;
                    break;
                }
                else if ( ( t[FUNHEAD+2] > -FUNCTION ) && ( t[1] == FUNHEAD+4 ) ) {
                    type = BRACKETOTHEREXPR;
                    tt = t + FUNHEAD+2;
                    switch ( *tt ) {
                        case -SYMBOL:
                            *b++ = 8; *b++ = SYMBOL; *b++ = 4; *b++ = tt[1];
                            *b++ = 1; *b++ = 1; *b++ = 1; *b++ = 3;
                            break;
                        case -SNUMBER:
                            if ( tt[1] == 1 ) {
                                *b++ = 4; *b++ = 1; *b++ = 1; *b++ = 3;
                            }
                            else goto IllBraReq;
                            break;
                        default:
                            goto IllBraReq;
                    }
                    break;
                }
            }
            else if ( ( t[FUNHEAD+2] == (t[1]-FUNHEAD-2) ) &&
                      ( t[FUNHEAD+2+ARGHEAD] == (t[FUNHEAD+2]-ARGHEAD) ) ) {
                type = BRACKETOTHEREXPR;
                tt = t + FUNHEAD + ARGHEAD; num = *tt;
                for ( i = 0; i < num; i++ ) *b++ = *tt++;
                break;
            }
        }
        else {
            if ( t[FUNHEAD] < 0 ) {
                if ( ( t[FUNHEAD] <= -FUNCTION ) && ( t[1] == FUNHEAD+1 ) ) {
                    type = BRACKETCURRENTEXPR;
                    *b++ = FUNHEAD+4; *b++ = -t[FUNHEAD+2]; *b++ = FUNHEAD;
                    for ( i = 2; i < FUNHEAD; i++ ) *b++ = 0;
                    *b++ = 1; *b++ = 1; *b++ = 3; *b = 0;
                    break;
                }
                else if ( ( t[FUNHEAD] > -FUNCTION ) && ( t[1] == FUNHEAD+2 ) ) {
                    type = BRACKETCURRENTEXPR;
                    tt = t + FUNHEAD+2;
                    switch ( *tt ) {
                        case -SYMBOL:
                            *b++ = 8; *b++ = SYMBOL; *b++ = 4; *b++ = tt[1];
                            *b++ = 1; *b++ = 1; *b++ = 1; *b++ = 3;
                            break;
                        case -SNUMBER:
                            if ( tt[1] == 1 ) {
                                *b++ = 4; *b++ = 1; *b++ = 1; *b++ = 3;
                            }
                            else goto IllBraReq;
                            break;
                        default:
                            goto IllBraReq;
                    }
                    break;
                }
            }
            else if ( ( t[FUNHEAD] == (t[1]-FUNHEAD) ) &&
                      ( t[FUNHEAD+ARGHEAD] == (t[FUNHEAD]-ARGHEAD) ) ) {
                type = BRACKETCURRENTEXPR;
                tt = t + FUNHEAD + ARGHEAD; num = *tt;
                for ( i = 0; i < num; i++ ) *b++ = *tt++;
                break;
            }
            else {
IllBraReq:;
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal bracket request in termsinbracket_ function.");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
        }
        t += t[1];
    }
    AT.WorkPointer = b;
    if ( AT.WorkPointer + *term +4 > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MesPrint("Called from termsinbracket_ function.");
        MUNLOCK(ErrorMessageLock);
```

```
        return(-1);
    }
/*
    We are now in the position to look for the bracket
*/
    switch ( type ) {
        case CURRENTBRACKET:
/*
            The code here should be rather similar to when we pick up
            the contents of the bracket. In our case we only count the
            terms though.
*/
            numterms = CountTerms1(BHEAD0);
            break;
        case BRACKETCURRENTEXPR:
/*
            Not implemented yet.
*/
            MLOCK(ErrorMessageLock);
            MesPrint("termsinbracket_ function currently only handles Keep Brackets.");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        case BRACKETOTHEREXPR:
            MLOCK(ErrorMessageLock);
            MesPrint("termsinbracket_ function currently only handles Keep Brackets.");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        case NOBRACKETACTIVE:
            numterms = 1;
            break;
    }
/*
    Now we have the number in numterms. We replace the function by it.
*/
    n1 = term; n2 = AT.WorkPointer; tstop = n1 + *n1;
    while ( n1 < t ) *n2++ = *n1++;
    i = numterms » BITSINWORD;
    if ( i == 0 ) {
        *n2++ = LNUMBER; *n2++ = 4; *n2++ = 1; *n2++ = (WORD)(numterms & WORDMASK);
    }
    else {
        *n2++ = LNUMBER; *n2++ = 5; *n2++ = 2;
        *n2++ = (WORD)(numterms & WORDMASK); *n2++ = i;
    }
    n1 += n1[1];
    while ( n1 < tstop ) *n2++ = *n1++;
    AT.WorkPointer[0] = n2 - AT.WorkPointer;
    AT.WorkPointer = n2;
    if ( Generator(BHEAD n1,level) < 0 ) {
        AT.WorkPointer = bracketbuffer;
        MLOCK(ErrorMessageLock);
        MesPrint("Called from termsinbracket_ function.");
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
/*
    Finished. Reset things and return.
*/
    AT.WorkPointer = bracketbuffer;
    return(numterms);
}
/*
        #] TermsInBracket :     LONG TermsInBracket(term,level)
    #] Expressions :
*/
```

4.27 extcmd.c File Reference

`#include "form3.h"`
Include dependency graph for extcmd.c:

Functions

- int openExternalChannel (UBYTE ∗cmd, int daemonize, UBYTE ∗shellname, UBYTE ∗stderrname)
- int initPresetExternalChannels (UBYTE ∗theline, int thetimeout)
- int closeExternalChannel (int n)
- int selectExternalChannel (int n)
- int getCurrentExternalChannel ()
- void closeAllExternalChannels ()

4.27.1 Detailed Description

The system that takes care of communication with external programs.

Definition in file extcmd.c.

4.27.2 Function Documentation

4.27.2.1 openExternalChannel()

```
int openExternalChannel (
            UBYTE * cmd,
            int daemonize,
            UBYTE * shellname,
            UBYTE * stderrname )
```

Definition at line 236 of file extcmd.c.

4.27.2.2 initPresetExternalChannels()

```
int initPresetExternalChannels (
            UBYTE * theline,
            int thetimeout )
```

Definition at line 238 of file extcmd.c.

4.27.2.3 closeExternalChannel()

```
int closeExternalChannel (
            int n )
```

Definition at line 239 of file extcmd.c.

4.27.2.4 selectExternalChannel()

```
int selectExternalChannel (
            int n )
```

Definition at line 240 of file extcmd.c.

4.27.2.5 getCurrentExternalChannel()

```
int getCurrentExternalChannel ( )
```

Definition at line 241 of file extcmd.c.

4.27.2.6 closeAllExternalChannels()

```
void closeAllExternalChannels ( )
```

Definition at line 242 of file extcmd.c.

4.28 extcmd.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Documentation :
00033
00034     This module is written by M.Tentyukov as a part of implementation of
00035     interaction between FORM and external processes, first release
```

```
09.04.2004. A part of this code is copyied from the DIANA project
written by M. Tentyukov and published under the GPL version 2 as
published by the Free Software Foundation.  The code of this module
is NOT covered by GPL; it can be used under the terms of the FORM
License http://www.nikhef.nl/~form/license.html

This file is completely re-written by M.Tentyukov in May 2006.
Since the interface was changed, the public function were changed,
also. A new publc functions were added: initPresetExternalChannels()
(see comments just before this function in the present file) and
setKillModeForExternalChannel (a pointer, not a function).

If a macro WITHEXTERNALCHANNEL is not defined, all public punctions
are stubs returning failure.

The idea is to start an external command  swallowing
its stdin and stdout. This can be done by means of the function
int openExternalChannel(cmd,daemonize,shellname,stderrname), where
cmd is a command to run,
daemonize: if !=0 then start the command in the "daemon" mode,
shellname: if !=NULL, execute the command in a subshell,
stderrname: if != NULL, redirect stderr of the command to this file.
The function returns some small positive integer number (the
descriptor of a newly created external channel), or -1 on failure.

After the command is started, it becomes a _current_ opened external
channel. The buffer can be sent to its stdin by a function
int writeBufToExtChannel(buf, n)
(here buf is a pointer to the buffer, n is the length in bytes; the
function returns 0 in success, or -1 on failure),
or one character can be read from its stdout by
means of the function
int getcFromExtChannel().

The latter returns the
character casted to integer, or something <0. This can be -2 (if there
is no current external channel) or EOF, if the external program closes
its stdout, or if the external program outputs a string coinciding
with a _terminator_.

By default, the terminator if an empty line. For the current external
channel it can be set by means of the function
int setTerminatorForExternalChannel(newterminaror).
The function returns 0 in success, or !0 if something is wrong (no
current channel, too long terminator).

After getcFromExtChannel() returns EOF, the current channel becomes
undefined. Any further attempts to read information by
getcFromExtChannel() result in -2. To set (re-set) a current channel,
the function
int selectExternalChannel(n)
can be used. This function accepts the valid external channel
descriptor (returned by openExternalChannel) and returns the
descriptor of a previous current channel (0, if there was no current
channel, or -1, if the external channel descriptor is invalid).
If n == 0, the function undefine the current external channel.

The function
int closeExternalChannel(n)
destroys the opened external channel with the descriptor n. It returns
0 in success, or -1 on failure. If the corresponding external channel
was the current one, the current channel becomes undefined. If n==0,
the function closes the current external channel.

The function
int getCurrentExternalChannel(void)
returns the descriptor if the current external channel, or 0 , if
there is no current external channel.

The function
void closeAllExternalChannels(void)

destroys all opened external channels.

List of all public functions:
int openExternalChannel(UBYTE *cmd,int daemonize,UBYTE *shellname, UBYTE * stderrname);
int initPresetExternalChannels(UBYTE *theline, int thetimeout);
int setTerminatorForExternalChannel(char *newterminaror);
int setKillModeForExternalChannel(int signum, int sentToWholeGroup);
int closeExternalChannel(int n);
int selectExternalChannel(int n);
int writeBufToExtChannel(char *buf,int n);
int getcFromExtChannel(void);
int getCurrentExternalChannel(void);
void closeAllExternalChannels(void);

ATTENTION!
```

```

    Four of them:
    1 setTerminatorForExternalChannel
    2 setKillModeForExternalChannel
    3 writeBufToExtChannel
    4 getcFromExtChannel

    are NOT functions, but variables (pointers) of a corrsponding type.
    They are initialised by proper values to avoid repeated error checking.

    All public functions are independent of realization hidden in this module.
    All other functions may have a returned type/parameters type local w.r.t.
    this module; they are not declared outside of this file.

    #] Documentation :
    #[ Selftest initializing:
*/

/*
Uncomment to get a self-consistent program:
#define SELFTEST 1
*/


#ifdef SELFTEST
#define WITHEXTERNALCHANNEL 1
#ifdef _MSC_VER
#define FORM_INLINE __inline
#else
#define FORM_INLINE inline
#endif
/*
    from declare.h:
*/
#define VOID void

/*
    From form3.h:
*/
typedef unsigned char UBYTE;

/*The following variables should be defined in variable.h:*/
extern int (*writeBufToExtChannel)(char *buffer, size_t n);
extern int (*getcFromExtChannel)();
extern int (*setTerminatorForExternalChannel)(char *buffer);
extern int (*setKillModeForExternalChannel)(int signum, int sentToWholeGroup);

#else /*ifdef SELFTEST*/
#include "form3.h"
#endif /*ifdef SELFTEST ... else*/
/*
pid_t  getExternalChannelPid(VOID);
*/
/*
    #] Selftest initializing:
    #[ Includes :
*/
#ifdef WITHEXTERNALCHANNEL
#include <stdio.h>
#ifndef _MSC_VER
#include <unistd.h>
#endif
#include <fcntl.h>
#include <sys/types.h>
#ifndef _MSC_VER
#include <sys/time.h>
#include <sys/wait.h>
#endif
#include <errno.h>
#include <signal.h>
#include <limits.h>
/*
    #] Includes :
    #[ FailureFunctions:
*/

/*Non-initialized variant of public functions:*/
int writeBufToExtChannelFailure(char *buf, size_t count)
{
    DUMMYUSE(buf); DUMMYUSE(count);
    return(-1);
}/*writeBufToExtChannelFailure*/

int setTerminatorForExternalChannelFailure(char *newTerminator)
{
    DUMMYUSE(newTerminator);
    return(-1);
```

```
}/*setTerminatorForExternalChannelFailure*/

int setKillModeForExternalChannelFailure(int signum, int sentToWholeGroup)
{
    DUMMYUSE(signum); DUMMYUSE(sentToWholeGroup);
    return(-1);
}/*setKillModeForExternalChannelFailure*/

int getcFromExtChannelFailure()
{
    return(-2);
}/*getcFromExtChannelFailure*/

int (*writeBufToExtChannel)(char *buffer, size_t n) = &writeBufToExtChannelFailure;
int (*setTerminatorForExternalChannel)(char *buffer) =
    &setTerminatorForExternalChannelFailure;
int (*setKillModeForExternalChannel)(int signum, int sentToWholeGroup) =
    &setKillModeForExternalChannelFailure;
int (*getcFromExtChannel)() = &getcFromExtChannelFailure;
#endif
/*
    #] FailureFunctions:
    #[ Stubs :
*/
#ifndef WITHEXTERNALCHANNEL
/*Stubs for public functions:*/
int openExternalChannel(UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)
{ DUMMYUSE(cmd); DUMMYUSE(daemonize); DUMMYUSE(shellname); DUMMYUSE(stderrname); return(-1); };
int initPresetExternalChannels(UBYTE *theline, int thetimeout) { DUMMYUSE(theline);
DUMMYUSE(thetimeout); return(-1); };
int closeExternalChannel(int n) { DUMMYUSE(n); return(-1); };
int selectExternalChannel(int n) { DUMMYUSE(n); return(-1); };
int getCurrentExternalChannel() { return(0); };
void closeAllExternalChannels() {};
#else /*ifndef WITHEXTERNALCHANNEL*/
/*
    #] Stubs :
    #[ Local types :
*/
/*First argument for the function signal:*/
#ifndef INTSIGHANDLER
typedef void (*mysighandler_t)(int);
#else
/* Sometimes, this nonsense may occurs:*/
/*typedef int (*mysighandler_t)(int);*/
#endif

/*Input IO buffer size increment -- each time the buffer
is expired it will be increased by this value (in bytes):*/
#define DELTA_EXT_BUF 128

/*Re-allocatable array containing External Channel
handlers increased each time by this value:*/
#define DELTA_EXT_LIST 8

/*How many times I/O routines may attempt to continue their work
in some failures:*/
#define MAX_FAILS_IO 2

/*The external channel handler structure:*/
typedef struct ExternalChannel {
    pid_t pid;       /*PID of the external process*/
    pid_t gpid;       /*process  group ID of the external process.
                            If <=0, not used, if >0, the kill signals
                            is sent to the whole group */
    FILE *frec;      /*stdout of the external process*/
    char *INbuf;     /*External channel buffer*/
    char *IBfill;    /*Position in INbuf from which the next input character will be read*/
    char *IBfull;    /*End of read INbuf*/
    char *IBstop;    /*end of allocated space for INbuf*/
    char *terminator;/* Terminator - when extern. program outputs ONLY this string,
                            it is assumed that the answer is ready, and getcFromExtChannel
                            returns EOF. Should not be longer then the minimal buffer!*/
    /*Info fields, not changable after creating a channel:*/
    char *cmd;       /*the command*/
    char *shellname;
    char *stderrname;/*filename to redirect stderr, or NULL*/
    int fsend;       /*stdin of the external process*/
    int killSignal; /*signal to kill*/
    int daemonize;/*0 --neither setsid nor daemonize, !=0 -- full daemonization*/
    PADPOINTER(0,3,0,0);
} EXTHANDLE;


static EXTHANDLE *externalChannelsList=0;
/*Here integers are better than pointers: */
static int externalChannelsListStop=0;
```

```c
static int externalChannelsListFill=0;

/*"current" external channel:*/
static EXTHANDLE *externalChannelsListTop=0;
/*
    #] Local types :
    #[ Selftest functions :
*/
#ifdef SELFTEST

/*For malloc prototype:*/
#include <stdlib.h>

/*StrLen, Malloc1, M_free and strDup1 are defined in tools.c -- here only emulation:*/
int StrLen(char *pattern)
{
register  char *p=(char*)pattern;
     while(*p)p++;
     return((int) ((p-(char*)pattern)) );
}/*StrLen*/
void *Malloc1(int l, char *c)
{
    return(malloc(l));
}
void M_free(void *p,char *c)
{
    return(free(p));
}

char *strDup1(UBYTE *instring, char *ifwrong)
{
    UBYTE *s = instring, *to;
    while ( *s ) s++;
    to = s = (UBYTE *)Malloc1((s-instring)+1,ifwrong);
    while ( *instring ) *to++ = *instring++;
    *to = 0;
    return(s);
}

/*PutPreVar from pre.c -- just ths stub:*/
int PutPreVar(UBYTE *a,UBYTE *b,UBYTE *c,int i)
{
    return(0);
}

#endif
/*
    #] Selftest functions :
    #[ Local functions :
*/

/*Initialize one cell of handler:*/
static FORM_INLINE VOID extHandlerInit(EXTHANDLE *h)
{
    h->pid=-1;
    h->gpid=-1;
    h->fsend=0;
    h->killSignal=SIGKILL;
    h->daemonize=1;
    h->frec=NULL;
    h->INbuf=h->IBfill=h->IBfull=h->IBstop=
    h->terminator=h->cmd=h->shellname=h->stderrname=NULL;
}/*extHandlerInit*/

/* Copies each field of handler:*/
static FORM_INLINE VOID extHandlerSwallowCopy(EXTHANDLE *to, EXTHANDLE *from)
{
    to->pid=from->pid;
    to->gpid=from->gpid;
    to->fsend=from->fsend;
    to->killSignal=from->killSignal;
    to->daemonize=from->daemonize;
    to->frec=from->frec;
    to->INbuf=from->INbuf;
    to->IBfill=from->IBfill;
    to->IBfull=from->IBfull;
    to->IBstop=from->IBstop;
    to->terminator=from->terminator;
    to->cmd=from->cmd;
    to->shellname=from->shellname;
    to->stderrname=from->stderrname;
}/*extHandlerSwallow*/

/*Allocates memory for fields of handler which have no fixed
 storage size and initializes some fields:*/
static FORM_INLINE VOID
extHandlerAlloc(EXTHANDLE *h, char *cmd, char *shellname, char *stderrname)
```

```
{
    h->IBfill=h->IBfull=h->INbuf=
                Malloc1(DELTA_EXT_BUF,"External channel buffer");
    h->IBstop=h->INbuf+DELTA_EXT_BUF;
    /*Initialize a terminator:*/
    *(h->terminator=Malloc1(DELTA_EXT_BUF,"External channel terminator"))='\n';
    (h->terminator)[1]='\0';/*By default the terminator is '\n'*/
    /*Deep copy all strings:*/
    if(cmd!=NULL)
        h->cmd=(char *)strDup1((UBYTE *)cmd,"External channel command");
    else/*cmd cannot be NULL! If this is NULL then force it to be something special*/
        h->cmd=(char *)strDup1((UBYTE *)"/","External channel command");
    if(shellname!=NULL)
        h->shellname=
             (char *)strDup1((UBYTE *)shellname,"External channel shell name");
    if(stderrname!=NULL)
        h->stderrname=
             (char *)strDup1((UBYTE *)stderrname,"External channel stderr name");
}/*extHandlerAlloc*/

/*Disallocates dynamically allocated fields of a handler:*/
static FORM_INLINE VOID extHandlerFree(EXTHANDLE *h)
{
    if(h->stderrname) M_free(h->stderrname,"External channel stderr name");
    if(h->shellname) M_free(h->shellname,"External channel shell name");
    if(h->cmd) M_free(h->cmd,"External channel command");
    if(h->terminator)M_free(h->terminator,"External channel terminator");
    if(h->INbuf)M_free(h->INbuf,"External channel buffer");
    extHandlerInit(h);
}/*extHandlerFree*/
/* Closes all descriptors, kills the external process, frees all internal fields,
BUT does NOT free the main container:*/
static VOID destroyExternalChannel(EXTHANDLE *h)
{
    /*Note, this function works in parallel mode correctly, see comments below.*/

    /*Note, for slaves in a parallel mode h->pid == 0:*/
    if( (h->pid > 0) && (h->killSignal > 0) ){
        int chstatus;
        if( h->gpid > 0)
            chstatus=kill(-h->gpid,h->killSignal);
        else
            chstatus=kill(h->pid,h->killSignal);
        if(chstatus==0)
        /*If the process will not be killed by this signal, FORM hangs up here!:*/
            waitpid(h->pid, &chstatus, 0);
    }/*if( (h->pid > 0) && (h->killSignal > 0) )*/

    /*Note, for slaves in a parallel mode h->frec == h->fsend == 0:*/
    if(h->frec) fclose(h->frec);
    if( h->fsend > 0) close(h->fsend);

    extHandlerFree(h);
    /*Does not do "free(h)"!*/
}/*destroyExternalChannel*/

/*Wrapper to the read() syscall, to handle possible interrupts by unblocked signals:*/
static FORM_INLINE ssize_t read2b(int fd, char *buf, size_t count)
{
ssize_t res;

    if( (res=read(fd,buf,count)) <1 )/*EOF or read is interrupted by a signal?:*/
         while( (errno == EINTR)&&(res <1) )
             /*The call was interrupted by  a  signal  before  any data was read, try again:*/
             res=read(fd,buf,count);
    return (res);
}/*read2b*/

/*Wrapper to the write() syscall, to handle possible interrupts by unblocked signals:*/
static FORM_INLINE ssize_t writeFromb(int fd, char *buf, size_t count)
{
ssize_t res;
    if( (res=write(fd,buf,count)) <1 )/*Is write interrupted by a signal?:*/
         while( (errno == EINTR)&&(res <1) )
             /*The call was interrupted by a signal before any data was written, try again:*/
             res=write(fd,buf,count);
    return (res);
}/*writeFromb*/

/* Read one (binary) PID from the file descriptor fd:*/
static FORM_INLINE pid_t readpid(int fd)
{
pid_t tmp;
    if(read2b(fd,(char*)&tmp,sizeof(pid_t))!=sizeof(pid_t))
      return (pid_t)-1;
    return tmp;
}/*readpid*/
```

```c

/* Writeone (binary) PID to the file descriptor fd:*/
static FORM_INLINE pid_t writepid(int fd, pid_t thepid)
{
    if(writeFromb(fd,(char*)&thepid,sizeof(pid_t))!=sizeof(pid_t))
      return (pid_t)-1;
    return (pid_t)0;
}/*readpid*/

/*Wrtites exactly count bytes from the  buffer buf into the descriptor fd, independently on
  nonblocked signals and the MPU/buffer hits. Returns 0 or -1:
*/
static FORM_INLINE int writexactly(int fd, char *buf, size_t count)
{
ssize_t i;
int j=0,n=0;

    for(;;){
        if(  (i=writeFromb(fd, buf+j, count-j)) < 0 ) return(-1);
        j+=i;
        if ( ((size_t)j) == count ) break;
        if(i==0)n++;
        else n=0;
        if(n>MAX_FAILS_IO)return (-1);
    }/*for(;;)*/
    return (0);
}/*writexactly*/

/* Set the FD_CLOEXEC  flag of desc if value is nonzero,
 or clear the flag if value is 0.
 Return 0 on success, or -1 on error with errno  set. */
static int set_cloexec_flag(int desc, int value)
{
int oldflags = fcntl (desc, F_GETFD, 0);
    /* If reading the flags failed, return error indication now.*/
    if (oldflags < 0)
        return (oldflags);
    /* Set just the flag we want to set. */
    if (value != 0)
        oldflags |= FD_CLOEXEC;
    else
      oldflags &= ~FD_CLOEXEC;
    /* Store modified flag word in the descriptor. */
    return (fcntl(desc, F_SETFD, oldflags));
}/*set_cloexec_flag*/

/* Adds the integer fd to the array fifo of length top+1 so that
the array is ascendantly ordered. It is supposed that all 0 -- top-1
elements in the array are already ordered:*/
static VOID pushDescriptor(int *fifo, int top, int fd)
{
    if ( top == 0 ) {
        fifo[top] = fd;
    } else {
        int ins=top-1;
        if( fifo[ins]<=fd )
            fifo[top]=fd;
        else{
            /*Find the position:*/
            while( (ins>=0)&&(fifo[ins]>fd) )ins--;
            /*Move all elements starting from the position to the right:*/
            for(ins++;top>ins; top--)
                fifo[top]=fifo[top-1];
            /*Put the element:*/
            fifo[ins]=fd;
        }
    }
}/*pushDescriptor*/

/*Close all descriptors greate or equal than startFrom except those
  listed in the ascendantly ordered array usedFd of length top:*/
static FORM_INLINE VOID closeAllDescriptors(int startFrom, int *usedFd, int top)
{
int n,maxfd;
    for(n=0;n<top; n++){
        maxfd=usedFd[n];
        for(;startFrom<maxfd;startFrom++)/*Close all less than maxfd*/
            close(startFrom);
        startFrom++;/*skip maxfd*/
    }/*for(;startFrom<maxfd;startFrom++)*/
    /*Close all the rest:*/
    maxfd=sysconf(_SC_OPEN_MAX);
    for(;startFrom<maxfd;startFrom++)
        close(startFrom);
}/*closeAllDescriptors*/

typedef int L_APIPE[2];
```

```
/*Closes both pipe descriptors if not -1:*/
static VOID closepipe(L_APIPE *thepipe)
{
  if( (*thepipe)[0] != -1) close ((*thepipe)[0]);
  if( (*thepipe)[1] != -1) close ((*thepipe)[1]);
}/*closepipe*/

/*Parses the cmd line like "sh -c myprg", passes each option to the
  correspondinig element of argv, ends agrv by NULL. Returns the
  number of stored argv elements, or -1 if fails:*/
static FORM_INLINE int parseline(char **argv, char *cmd)
{
int n=0;
    while(*cmd != '\0'){
        for(; (*cmd <= ' ') && (*cmd != '\0') ;cmd++);
        if(*cmd != '\0'){
            argv[n]=cmd;
            while(*++cmd > ' ');
            if(*cmd != '\0')
                *cmd++ = '\0';
            n++;
        }/*if(*cmd != '\0')*/
    }/*while(*cmd != '\0')*/
    argv[n]=NULL;
    if(n==0)return -1;
    return n;
}/*parseline*/

/*Reads positive decimal number (not bigger than maxnum)
  from the string and returns it;
  the pointer *b is set to the next non-converted character:*/
static LONG str2i(char *str, char **b, LONG maxnum)
{
    LONG n=0;
    /*Eat trailing spaces:*/
    while(*str<=' ')if(*str++ == '\0')return(-1);
    (*b)=str;
    while (*str>='0'&&*str<='9')
        if( (n=10*n + *str++ - '0')>maxnum )
            return(-1);
    if((*b)==str)/*No single number!*/
        return(-1);
    (*b)=str;
    return(n);
}

/*Converts long integer to a decimal representation.
  For portability reasons we cannot use LongCopy from tools.c
  since theoretically LONG may be smaller than pid_t:*/
static char *l2s(LONG x, char *to)
{
    char *s;
    int i = 0, j;
    s = to;
    do { *s++ = (x % 10)+'0'; i++; } while ( ( x /= 10 ) != 0 );
    *s-- = '\0';
    j = ( i - 1 ) » 1;
    while ( j >= 0 ) {
        i = to[j]; to[j] = s[-j]; s[-j] = (char)i; j--;
    }
    return(s+1);
}

/*like strcat() but returns the pointer to the end of the
resulting string:*/
static  FORM_INLINE char *addStr(char *to, char *from)
{
    while(  (*to++ = *from++)!='\0' );
    return(to-1);
}/*addStr*/


/*Try to write (atomically) short buffer (of length count) to fd.
  timeout is a timeout in millisecs. Returns number of writen bytes or -1:*/
static FORM_INLINE ssize_t writeSome(int fd, char *buf, size_t count, int timeout)
{
    ssize_t res = 0;
    fd_set wfds;
    struct timeval tv;
    int nrep=5;/*five attempts it interrupted by a non-blocking signal*/
    int flags = fcntl(fd, F_GETFL,0);

    /*Add O_NONBLOCK:*/
    fcntl(fd,F_SETFL, flags | O_NONBLOCK);
    /* important -- in order to avoid blocking of short rceiver buffer*/

    do{
```

```
        FD_ZERO(&wfds);
        FD_SET(fd, &wfds);
        /* Wait up to timeout. */

        tv.tv_sec =timeout /1000;
        tv.tv_usec = (timeout % 1000)*1000;
        nrep--;

        switch(select(fd+1, NULL, &wfds, NULL, &tv)){
            case -1:
                if((nrep == 0)||( errno != EINTR) ){
                    perror("select()");
                    res=-1;
                    nrep=0;
                }/*else -- A non blocked signal was caught, just repeat*/
                break;
            case 0:/*timeout*/
                res=-1;
                nrep=0;
                break;
            default:
                if( (res=write(fd,buf,count)) <0 )/*Signal?*/
                    while( (errno == EINTR)&&(res <0) )
                      res=write(fd,buf,count);
                nrep=0;
        }/*switch*/
    }while(nrep);

    /*restore the flags:*/
    fcntl(fd,F_SETFL, flags);
    return (res);
}/*writeSome*/

/*Try to read short buffer (of length not more than count)
  from fd. timeout is a timeout in millisecs. Returns number
  of writen bytes or -1: */
static FORM_INLINE ssize_t readSome(int fd, char *buf, size_t count, int timeout)
{
    ssize_t res = 0;
    fd_set rfds;
    struct timeval tv;
    int nrep=5;/*five attempts it interrupted by a non-blocking signal*/

    do{
        FD_ZERO(&rfds);
        FD_SET(fd, &rfds);
        /* Wait up to timeout. */

        tv.tv_sec = timeout/1000;
        tv.tv_usec = (timeout % 1000)*1000;
        nrep--;

        switch(select(fd+1, &rfds, NULL, NULL, &tv)){
            case -1:
                if((nrep == 0)||( errno != EINTR) ){
                    perror("select()");
                    res=-1;
                    nrep=0;
                }/*else -- A non blocked signal was caught, just repeat*/
                break;
            case 0:/*timeout*/
                res=-1;
                nrep=0;
                break;
            default:
                if( (res=read(fd,buf,count)) <0 )/*Signal?*/
                    while( (errno == EINTR)&&(res <0) )
                      res=read(fd,buf,count);
                nrep=0;
        }/*switch*/
    }while(nrep);
    return (res);
}/*readSome*/

/*
    #] Local functions :
    #[ Ok functions:
*/

/*Copies (deep copy) newTerminator to thehandler->terminator. Returns 0 if
  newTerminator fits to the buffer, or !0 if it does not fit. ATT! In the
  latter case thehandler->terminator is NOT '\0' terminated! */
int setTerminatorForExternalChannelOk(char *newTerminator)
{
int i=DELTA_EXT_BUF;
/*
    No problems with externalChannelsListTop are
```

```
    possible since this function may be invoked only
    when the current channel is defined and externalChannelsListTop
  is set properly
*/
char *t=externalChannelsListTop->terminator;

    for(; i>1; i--)
        if( (*t++ = *newTerminator++)=='\0' )
            break;
    /*Add trailing '\n', if absent:*/
    if(  (i == DELTA_EXT_BUF)/*newTerminator == '\0'*/
        ||(*(t-2)!='\n')       ){
      *(t-1)='\n';*t='\0';
    }
    return(i==1);
}/*setTerminatorForExternalChannelOk*/

/*Interface to change handler fields "killSignal" and "gpid"*/
int setKillModeForExternalChannelOk(int signum, int sentToWholeGroup)
{
    if(signum<0)
        return(-1);
    /*
        No problems with externalChannelsListTop are
        possible since this function may be invoked only
        when the current channel is defined and externalChannelsListTop
        is set properly
    */
    externalChannelsListTop->killSignal=signum;
    if(sentToWholeGroup){/*gpid must be >0*/
        if(externalChannelsListTop->gpid <= 0)
            externalChannelsListTop->gpid=-externalChannelsListTop->gpid;
    }else{/*gpid must be <=0*/
        if(externalChannelsListTop->gpid>0)
            externalChannelsListTop->gpid=-externalChannelsListTop->gpid;
    }
    return(0);
}/*setKillModeForExternalChannelOk*/

/*
        #[ getcFromExtChannelOk
*/
/*Returns one character from the external channel. It the input is expired,
returns EOF. If the external process is finished completely, the function closes
the channel (and returns EOF). If the external process was finished, the function
returns EOF:*/
int getcFromExtChannelOk()
{
mysighandler_t oldPIPE = 0;
EXTHANDLE *h;
int ret;

    if (externalChannelsListTop->IBfill < externalChannelsListTop->IBfull)
        /*in buffer*/
        return( *(externalChannelsListTop->IBfill++) );
    /*else -- the buffer is empty*/
    ret=EOF;
    h= externalChannelsListTop;
#ifdef WITHMPI
    if ( PF.me == MASTER ){
#endif
    /* Temporary ignore this signal:*/
    /* if compiler fails here, try to change the definition of
        mysighandler_t on the beginning of this file
        (just define INTSIGHANDLER).*/
    oldPIPE=signal(SIGPIPE,SIG_IGN);
#ifdef WITHMPI
    if(  fgets(h->INbuf,h->IBstop - h->INbuf, h->frec) == 0  )/*Fail! EOF?*/
        *(h->INbuf)='\0';/*Empty line may not appear here!*/
#else
    if(  (fgets(h->INbuf,h->IBstop - h->INbuf, h->frec) == 0)/*Fail! EOF?*/
        ||( *(h->INbuf) == '\0')/*Empty line? This shouldn't be!*/
      ){
        closeExternalChannel(externalChannelsListTop-externalChannelsList+1);
        /*Note, this code is only for the sequential mode! */
        goto getcFromExtChannelReady;
        /*Here we assume that fgets is never interrupted by singals*/
    }/*if( fgets(h->INbuf,h->IBstop - h->INbuf, h->frec) == 0 )*/
#endif
#ifdef WITHMPI
    }/*if ( PF.me == MASTER */

    /*Master broadcasts result to slaves, slaves read it from the master:*/
    if( PF_BroadcastString((UBYTE *)h->INbuf) ){/*Fail!*/
          MesPrint("Fail broadcasting external channel results");
          Terminate(-1);
    }/*if( PF_BroadcastString((UBYTE *)h->INbuf) )*/
```

```

    if( *(h->INbuf) == '\0'){/*Empty line? This shouldn't be!*/
        closeExternalChannel(externalChannelsListTop-externalChannelsList+1);
        goto getcFromExtChannelReady;
    }/*if( *(h->INbuf) == '\0')*/
#endif

    {/*Block*/
        char *t=h->terminator;
        /*Move IBfull to the end of read line and compare the line with the terminator.
          Note, by construction the terminator fits to the first read line, see
          the function setTerminatorForExternalChannel.*/
        for(h->IBfull=h->INbuf; *(h->IBfull)!='\0'; (h->IBfull)++)
            if( *t== *(h->IBfull) )
                t++;
            else
                break;/*not a terminator*/
        /*Continue moving IBfullto the end of read line:*/
        while(*(h->IBfull)!='\0')(h->IBfull)++;

        if( (t-h->terminator) == (h->IBfull-h->INbuf) ){
            /*Terminator!*/
            /*Reset the channel*/
            h->IBfull=h->IBfill=h->INbuf;
            externalChannelsListTop=0;/*Undefine the current channel*/
            writeBufToExtChannel=&writeBufToExtChannelFailure;
            getcFromExtChannel=&getcFromExtChannelFailure;
            setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
            setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
            goto getcFromExtChannelReady;
        }/*if(t == (h->IBfull-h->INbuf) )*/
    }/*Block*/
    /*Does the buffer have enough capacity?*/
    while( *(h->IBfull - 1) != '\n' ){/*Buffer is not enough!*/
        /*Extend the buffer:*/
        int l= (h->IBstop - h->INbuf)+DELTA_EXT_BUF;
        char *newbuf=Malloc1(l,"External channel buffer");
        /*We wouldn't like to use realloc.*/
        /*Copy the buffer:*/
        char *n=newbuf,*o=h->INbuf;
        while(  (*n++ = *o++)!='\0' );
        /*Att! The order of the following operators is important!:*/
        h->IBfull= newbuf+(h->IBfull-h->INbuf);
        M_free(h->INbuf,"External channel buffer");
        h->INbuf = newbuf;
        h->IBstop = h->INbuf+l;
#ifdef WITHMPI
        if ( PF.me == MASTER ){
            (h->IBfull)[1]='\0';/*Will mark (h->IBfull)[1] as '!' for failure*/
            if(  fgets(h->IBfull,h->IBstop - h->IBfull, h->frec) == 0 ){
                /*EOF! No trailing '\n'?*/
                /*Mark:*/
                (h->IBfull)[0]='\0';
                (h->IBfull)[1]='!';
                (h->IBfull)[2]='\0';
                 /*The string "\0!\0" is used as an image of NULL.*/
            }/*if(  fgets(h->IBfull,h->IBstop - h->IBfull, h->frec) == 0 )*/
        }/*if ( PF.me == MASTER )*/

        /*Master broadcasts results to slaves, slaves read it from the master:*/
        if( PF_BroadcastString((UBYTE *)h->IBfull) ){/*Fail!*/
          MesPrint("Fail broadcasting external channel results");
          Terminate(-1);
        }/*if( PF_BroadcastString(h->IBfull) )*/

        /*The string "\0!\0" is used as the image of NULL.*/
        if(
              ( (h->IBfull)[0]=='\0' )
            &&(  (h->IBfull)[1]=='!' )
            &&(  (h->IBfull)[2]=='\0' )
          )/*EOF! No trailing '\n'?*/
            break;
#else
        if(  fgets(h->IBfull,h->IBstop - h->IBfull, h->frec) == 0 )
            /*EOF! No trailing '\n'?*/
            break;
#endif
        while( *(h->IBfull)!='\0' )(h->IBfull)++;
    }/*while( *(h->IBfull - 1) != '\n' )*/
    /*In h->INbuf we have a fresh string.*/
    ret=*(h->IBfill=h->INbuf);
    h->IBfill++;/*Next time a new, isn't it?*/

    getcFromExtChannelReady:
#ifdef WITHMPI
    if ( PF.me == MASTER ){
#endif
```

```
            signal(SIGPIPE,oldPIPE);
#ifdef WITHMPI
    }/*if ( PF.me == MASTER )*/
#endif
            return(ret);
}/*getcFromExtChannelOk*/
/*
        #] getcFromExtChannelOk
*/
/*Writes exactly count bytes from the buffer buf to the external channel thehandler
  Returns 0 (on success) or -1:
*/
int writeBufToExtChannelOk(char *buf, size_t count)
{

int ret;
mysighandler_t oldPIPE;
#ifdef WITHMPI
    /*Only master communicates with the external program:*/
    if ( PF.me == MASTER ){
#endif
        /* Temporary ignore this signal:*/
        /* if compiler fails here, try to change the definition of
            mysighandler_t on the beginning of this file
            (just define INTSIGHANDLER)*/
        oldPIPE=signal(SIGPIPE,SIG_IGN);
        ret=writexactly( externalChannelsListTop->fsend, buf, count);
        signal(SIGPIPE,oldPIPE);
#ifdef WITHMPI
    }else{
        /*Do not wait the master status: this would be too slow!*/
        ret=0;
    }
#endif
    return(ret);
}/*writeBufToExtChannel*/
/*
    #] Ok functions:
    #[ do_run_cmd :
*/
/*The function returns PID of the started command*/
static FORM_INLINE pid_t do_run_cmd(
    int *fdsend,
    int *fdreceive,
    int *gpid, /*returns group process ID*/
    int ttymode,
                    /*
                     &8 - daemonizeing
                     &16 - setsid()*/
    char *cmd,
    char *argv[],
    char *stderrname
    )
{
int fdin[2]={-1,-1}, fdout[2]={-1,-1}, fdsig[2]={-1,-1};
/*initialised by -1 for possible rollback at failure, see closepipe() above*/

pid_t childpid,fatherchildpid = (pid_t)0;
mysighandler_t oldPIPE=NULL;

    if(
        (pipe(fdsig)!=0)/*This pipe will be used by a child to tell the father if fail.*/
        ||(pipe(fdin)!=0)
        ||(pipe(fdout)!=0)
    )goto fail_do_run_cmd;

    if((childpid = fork()) == -1){
         perror("fork");
         goto fail_do_run_cmd;
    }/*if((childpid = fork()) == -1)*/

    if(childpid == 0){/*Child.*/
        int fifo[3], top=0;
        /*
          To be thread safely we can't rely on ascendant order of opened
          file descriptors. So we put each of descriptor we have to
          preserve into the array fifo.
              Note, in _this_ process there are no any threads but descriptors
          were created in frame of the parent process which may have
          multiple threads.
        */
        /*Mark descriptors which will NOT be closed:*/
        pushDescriptor(fifo,top++,fdsig[1]);
        pushDescriptor(fifo,top++,fdin[0]);
        pushDescriptor(fifo,top++,fdout[1]);
        /*Close all except stdin, stdout, stderr and placed into fifo:*/
        closeAllDescriptors(3,fifo, top);
```

```

        /*Now reopen stdin and stdout.*/
        /*thread-safety is not a problem here since there are no any threads up to now:*/
        if(
            (close(0) == -1 )||/* Use fdin as stdin :*/
            (dup(fdin[0]) == -1 )||
            (close(1)==-1)||/* Use fdout as stdout:*/
            (dup(fdout[1]) == -1 )
          )
        {/*Fail!*/
            /*Signal to parent:*/
            writepid(fdsig[1],(pid_t)-2);
            _exit(1);
        }

        if(stderrname != NULL){
            if(
                (close(2) != 0 )||
                (open(stderrname,O_WRONLY)<0)
              )
            {/*Fail!*/
                writepid(fdsig[1],(pid_t)-2);
                _exit(1);
            }
        }/*if(stderrname != NULL)*/

        if( ttymode & 16 )/* create a session and sets the process group ID */
            setsid();

        /*   */
        if(set_cloexec_flag (fdsig[1], 1)!=0){/*Error?*/
            /*Signal to parent:*/
            writepid(fdsig[1],(pid_t)-2);
            _exit(1);
        }/*if(set_cloexec_flag (fdsig[1], 1)!=0)*/

        if( ttymode & 8 ){/*Daemonize*/
            int fdsig2[2];/*To check exec() success*/
            if(
                pipe(fdsig2)||
                (set_cloexec_flag (fdsig2[1], 1)!=0)
              )
            {/*Error?*/
                /*Signal to parent:*/
                writepid(fdsig[1],(pid_t)-2);
                _exit(1);
            }
            set_cloexec_flag (fdsig2[0], 1);
            switch(childpid=fork()){
                case 0:/*grandchild*/
                  /*Execute external command:*/
                  execvp(cmd, argv);
                  /* Control can  reach this point only on error!*/
                  writepid(fdsig2[1],(pid_t)-2);
                  break;
                case -1:
                  /* Control can  reach this point only on error!*/
                  /*Inform the father about the failure*/
                  writepid(fdsig[1],(pid_t)-2);
                  _exit(1);/*The child, just exit, not return*/
              default:/*Son of his father*/
                  close(fdsig2[1]);
                  /*Ignore SIGPIPE (up to the end of the process):*/
                  signal(SIGPIPE,SIG_IGN);

                  /*Wait on read() while the granchild close the pipe
                     (on success) or send -2 (if exec() fails).*/
                  /*There are two possibilities:
                      -1 -- this is ok, the pipe was closed on exec,
                      the program was successfully executed;
                      -2 -- something is wrong, exec failed since the
                      grandchild sends -2 after exec.
                  */
                  if( readpid(fdsig2[0]) != (pid_t)-1 )/*something is wrong*/
                      writepid(fdsig[1],(pid_t)-1);
                  else/*ok, send PID of the granchild to the father:*/
                      writepid(fdsig[1],childpid);
                  /*Die and free the life space for the grandchild:*/
                  _exit(0);/*The child, just exit, not return*/
            }/*switch(childpid=fork())*/
        }else{/*if( ttymode & 8 )*/
          execvp(cmd, argv);
          /* Control can  reach this point only on error!*/
          writepid(fdsig[1],(pid_t)-2);
          _exit(2);/*The child, just exit, not return*/
        }/*if( ttymode & 8 )...else*/
    }else{/* The (grand)father*/
```

```
        close(fdsig[1]);
        /*To prevent closing fdsig in rollback:*/
        fdsig[1]=-1;
        close(fdin[0]);
        close(fdout[1]);
        *fdsend    = fdin[1];
        *fdreceive = fdout[0];

        /*Get the process group ID.*/
        /*Avoid to use getpgid() which is non-standard.*/
        if( ttymode & 16)/*setsid() was invoked, the child is a group leader:*/
            *gpid=childpid;
        else/*the child belongs to the same process group as the this process:*/
            *gpid=getpgrp();/*if compiler fails here, try getpgrp(0) instead!*/
        /*
          Rationale: getpgrp  conform  to POSIX.1 while 4.3BSD provides a
          getpgrp() function that returns the process group ID for a
          specified process.
        */

        /* Temporary ignore this signal:*/
        /* if compiler fails here, try to change the definition of
          mysighandler_t on the beginning of this file
          (just define INTSIGHANDLER)*/
        oldPIPE=signal(SIGPIPE,SIG_IGN);

        if( ttymode & 8 ){/*Daemonize*/
            /*Read the grandchild PID from the son.*/
            fatherchildpid=childpid;
            if(  (childpid=readpid(fdsig[0]))<0  ){
                /*Daemonization process fails for some reasons!*/
                childpid=fatherchildpid;/*for rollback*/
                goto fail_do_run_cmd;
            }
        }else{
              /*fdsig[1] should be closed on exec and this read operation
                 must fail on success:*/
              if( readpid(fdsig[0])!= (pid_t)-1 )
                  goto fail_do_run_cmd;
        }/*if( ttymode & 8 ) ... else*/
    }/*if(childpid == 0)...else*/

    /*Here can be ONLY the father*/
    close(fdsig[0]);
    /*To prevent closing fdsig in rollback after goto fail_flnk_do_runcmd:*/
    fdsig[0]=-1;

    if( ttymode & 8 ){/*Daemonize*/
        int i;
        /*Wait while the father of a grandchild dies:*/
        waitpid(fatherchildpid,&i,0);
    }

    /*Restore the signal:*/
    signal(SIGPIPE,oldPIPE);

    return(childpid);

    fail_do_run_cmd:
        closepipe(&fdout);
        closepipe(&fdin);
        closepipe(&fdsig);
        return((pid_t)-1);
}/*do_run_cmd*/
/*
    #] do_run_cmd :
    #[ run_cmd :
*/
/*Starts the command cmd (directly, if shellpath is NULL, or in a subshell),
  swallowing its stdin and stdout;
  stderr will be re-directed to stderrname (if !=NULL). Returns PID of
  the started process. Stdin will be available as fdsend, and stdout
  will be available as fdreceive:*/
static FORM_INLINE pid_t run_cmd(char *cmd,
                                 int *fdsend,
                                 int *fdreceive,
                                 int *gpid,
                                 int daemonize,
                                 char *shellpath,
                                 char *stderrname )
{
char **argv;
pid_t thepid;

    cmd=(char*)strDup1((UBYTE*)cmd, "run_cmd: cmd");/*detouch cmd*/


```

```
    /* Prepare arguments for execvp:*/
    if(shellpath != NULL){/*Run in a subshell.*/
        int nopt;
        /*Allocate space which is definitely enough:*/
        argv=Malloc1(StrLen((UBYTE*)shellpath)*sizeof(char*)+2,"run_cmd:argv");
        shellpath=(char*)strDup1((UBYTE*)shellpath, "run_cmd: shellpath");/*detouch shellpath*/
        /*Parse a shell (e.g., "/bin/sh -c"):*/
        nopt=parseline(argv, shellpath);
        /* and add the command as a shell argument:*/
        argv[nopt]=cmd;
        argv[nopt+1]=NULL;
    }else{/*Run the command directly:*/
        /*Allocate space which is definitely enough:*/
        argv=Malloc1(StrLen((UBYTE*)cmd)*sizeof(char*)+1,"run_cmd:argv");
        parseline(argv, cmd);
    }

    thepid=do_run_cmd(
        fdsend,
        fdreceive,
        gpid,
        (daemonize)?(8|16):0,
        argv[0],
        argv,
        stderrname
    );

    M_free(argv,"run_cmd:argv");
    if(shellpath)
        M_free(shellpath,"run_cmd:argv");
    M_free(cmd, "run_cmd: cmd");

    return(thepid);
}/*run_cmd*/
/*
    #] run_cmd :
    #[ createExternalChannel :
*/
/*The structure to pass parameters to createExternalChannel
  and openExternalChannel in case of preset channel (instead of
  shellname):*/
typedef struct{
    int fdin;
    int fdout;
    pid_t theppid;
}ECINFOSTRUCT;

/* Creates a new external channel starting the command cmd (if cmd !=NULL)
    or using informaion from (ECINFOSTRUCT *)shellname, if cmd ==NULL:*/
static FORM_INLINE void *createExternalChannel(
                        EXTHANDLE *h,
                        char *cmd, /*Command to run or NULL*/
                        /*0 --neither setsid nor daemonize, !=0 -- full daemonization:*/
                        int daemonize,
                        char *shellname,/* The shell (like "/bin/sh -c") or NULL*/
                        char *stderrname/*filename to redirect stderr or NULL*/
                                                                    )
{
    int fdreceive=0;
    int gpid = 0;
    ECINFOSTRUCT *psetInfo;
#ifdef WITHMPI
    char statusbuf[2]={'\0','\0'};/*'\0' if run_cmd retuns ok, '!' othervise.*/
#endif
    extHandlerInit(h);

    h->pid=0;

    if( cmd==NULL ){/*Instead of strting a new command, use preset channel:*/
        psetInfo=(ECINFOSTRUCT *)shellname;
        shellname=NULL;
        h->killSignal=0;
        h->daemonize=0;
    }

    /*Create a channel:*/
#ifdef WITHMPI
    if ( PF.me == MASTER ){
#endif
        if(cmd!=NULL)
            h->pid=run_cmd (cmd, &(h->fsend),
                    &fdreceive,&gpid,daemonize,shellname,stderrname);
        else{
            gpid=-psetInfo->theppid;
            h->pid=psetInfo->theppid;
            h->fsend=psetInfo->fdout;
            fdreceive=psetInfo->fdin;
```

```
        }
#ifdef WITHMPI
        if(h->pid<0)
            statusbuf[0]='!';/*Brodcast fail to slaves*/
    }
     /*else: Keep h->pid = 0 and h->fsend = 0 for slaves in parallel mode!*/

    /*Master broadcasts status to slaves, slaves read it from the master:*/
    if( PF_BroadcastString((UBYTE *)statusbuf) ){/*Fail!*/
        h->pid=-1;
    }else if( statusbuf[0]=='!')/*Master fails*/
        h->pid=-1;
#endif

    if(h->pid<0)goto createExternalChannelFails;
#ifdef WITHMPI
    if ( PF.me == MASTER ){
#endif
        h->gpid=gpid;
    /*Open stdout of a newly created program as FILE* :*/
        if( (h->frec=fdopen(fdreceive,"r")) == 0 )goto createExternalChannelFails;

#ifdef WITHMPI
    }
#endif
    /*Initialize buffers:*/
    extHandlerAlloc(h,cmd,shellname,stderrname);
    return(h);
    /*Something is wrong?*/
    createExternalChannelFails:

        destroyExternalChannel(h);
        return(NULL);
}/*createExternalChannel*/

/*
    #] createExternalChannel :
    #[ openExternalChannel :
*/
int openExternalChannel(UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)
{
EXTHANDLE *h=externalChannelsListTop;
int i=0;

    for(externalChannelsListTop=0;i<externalChannelsListFill;i++)
        if(externalChannelsList[i].cmd==0){
            externalChannelsListTop=externalChannelsList+i;
            break;
        }/*if(externalChannelsList[i].cmd!=0)*/

    if(externalChannelsListTop==0){
      /*No free cells, create the new one:*/
      if(externalChannelsListFill == externalChannelsListStop){
            /*The list is full, increase and reallocate it:*/
            EXTHANDLE *newbuf=Malloc1(
                                        (externalChannelsListStop+=DELTA_EXT_LIST)*sizeof(EXTHANDLE),
                                      "External channel list");
            for(i=0;i<externalChannelsListFill;i++)
                 extHandlerSwallowCopy(newbuf+i,externalChannelsList+i);
            if(externalChannelsList!=0)
                M_free(externalChannelsList,"External channel list");
            for(;i<externalChannelsListStop;i++)
                extHandlerInit(newbuf+i);
            externalChannelsList=newbuf;
      }/*if(externalChannelsListFill == externalChannelsListStop)*/
      externalChannelsListTop=externalChannelsList+externalChannelsListFill;
      externalChannelsListFill++;
    }/*if(externalChannelsListTop==0)*/

    if(createExternalChannel(
                                      externalChannelsListTop,
                                      (char*) cmd,
                                      daemonize,
                                      (char*)shellname,
                                      (char*)stderrname            )!=NULL){
        writeBufToExtChannel=&writeBufToExtChannelOk;
        getcFromExtChannel=&getcFromExtChannelOk;
        setTerminatorForExternalChannel=&setTerminatorForExternalChannelOk;
        setKillModeForExternalChannel=&setKillModeForExternalChannelOk;
        return(externalChannelsListTop-externalChannelsList+1);
    }
    /*else - failure!*/
    externalChannelsListTop=h;
    return(-1);
}/*openExternalChannel*/
/*
    #] openExternalChannel :
```

```
    #[ initPresetExternalChannels :
*/
/*Just simpe wrapper to invoke  openExternalChannel()
    from initPresetExternalChannels():*/
static FORM_INLINE int openPresetExternalChannel(int fdin, int fdout, pid_t theppid)
{
ECINFOSTRUCT inf;
    inf.fdin=fdin; inf.fdout=fdout;inf.theppid=theppid;
    return( openExternalChannel(NULL,0,(UBYTE *)&inf,NULL) );
}/*openPresetExternalChannel*/

#define PIDTXTSIZE 23
#define BOTHPIDSIZE 45

#ifndef LONG_MAX
#define LONG_MAX 0x7FFFFFFFL
#endif

/*There is a possibility to start FORM from another program providing
one (or more) communication channels. These channels will be
visible from a FORM program as ``pre-opened" external channels existing
after FORM starts.
Before starting FORM, the parent application must create one or more
pairs of pipes The read-only descriptor of the first pipe in the pair
and the write-only descriptor of the second pipe must be passed to
FORM as an argument of a command line option -pipe in ASCII decimal
format. The argument of the option is a comma-separated list of pairs
r#,w# where r# is a read-only descriptor and w# is a write-only
descriptor. Alternatively, the environment variable FORM_PIPES can be
used.
    The following function expects as the first argument
this comma-separated list of the  desctiptor pairs and tries to
initialize each of channel during thetimeout milliseconds:*/

int initPresetExternalChannels(UBYTE *theline, int thetimeout)
{
    int i, nchannels = 0;
    int pidln;           /*The length of FORM PID in pidtxt*/
    char pidtxt[PIDTXTSIZE],     /*64/Log_2[10] = 19.3, this is enough for any integer*/
         chdescriptor[PIDTXTSIZE],
         bothpidtxt[BOTHPIDSIZE],   /*"#,#\n\0"*/
         *c,*b = 0;
    int pip[2];
    pid_t ppid;
    if ( theline == NULL ) return(-1);
    /*Put into pidtxt PID\n:*/
    c = l2s((LONG)getpid(),pidtxt);
    *c++='\n';
    *c = '\0';
    pidln = c-pidtxt;

    do {
        pip[0] = (int)str2i((char*)theline,&c,0xFFFF);
        if( ( pip[0] < 0 ) || ( *c != ',' ) ) goto presetFails;

        theline = (UBYTE*)c + 1;
        pip[1] = (int)str2i((char*)theline,&c,0xFFFF);
        if ( (pip[1] < 0 ) || ( ( *c != ',' ) && ( *c != '\0' ) ) ) goto presetFails;
        theline = (UBYTE *)c + 1;
        /*Now we have two descriptors.
          According to the protocol, FORM must send to external channel
          it's PID with added '\n' and read back two comma-separaetd
          decimals with added '\n'. The first must be repeated FORM PID,
          the second must be the parent PID
        */
        if ( writeSome(pip[1],pidtxt,pidln,thetimeout) != pidln ) goto presetFails;
        i = readSome(pip[0],bothpidtxt,BOTHPIDSIZE,thetimeout);
        if( ( i < 4 )                  /*at least 1,2\n*/
            || ( i == BOTHPIDSIZE )    /*No space for trailing '\0'*/
          ) goto presetFails;
        /*read the FORM PID:*/
        ppid = (pid_t)str2i(bothpidtxt,&b,getpid());
        if( ( *b != ',' ) || ( ppid != getpid() ) )goto presetFails;
        /*read the parent PID:*/
        /*The problem is that we do not know the the real type of pid_t.
          But long should be ehough. On obsolete systems (when LONG_MAX
          is not defined) we assume pid_t is 32-bit integer.
          This can lead to problem with portability: */
        ppid = (pid_t)str2i(b+1,&b,LONG_MAX);
        if ( (*b != '\n') || (ppid<2) ) goto presetFails;
        i = openPresetExternalChannel(pip[0],pip[1],ppid);
        if ( i < 0 ) goto presetFails;
        nchannels++;
        /*Now use bothpidtxt as a buffer for preprovar, the space is enough:*/
        /*"PIPE#_" where # is ne order number of the channel:*/
        b = l2s(nchannels,addStr(bothpidtxt,"PIPE"));
        *b = '_';
```

```
        b[1] = '\0';
        *l2s(i,chdescriptor) = '\0';
        PutPreVar((UBYTE*)bothpidtxt,(UBYTE*)chdescriptor,0,0);
    } while ( *c != '\0' );
    /*Export proprovar "PIPES_":*/
    *l2s(nchannels,chdescriptor)='\0';
    PutPreVar((UBYTE*)"PIPES_",(UBYTE*)chdescriptor,0,0);

    /*success:*/
    return (nchannels);

presetFails:
        /*Here we assume the descriptors the beginning of the list!*/
        for(i=0; i<nchannels; i++)
            destroyExternalChannel(externalChannelsList+i);
        return(-1);
} /*initPresetExternalChannels*/
/*
    #] initPresetExternalChannels :
    #[ selectExternalChannel :
*/
/*
Accepts the valid external channel descriptor (returned by
openExternalChannel) and returns the descriptor of a previous current
channel (0, if there was no current channel, or -1, if the external
channel descriptor is invalid).  If n == 0, the function undefine the
current external channel:
*/
int selectExternalChannel(int n)
{
int ret=0;
    if(externalChannelsListTop!=0)
        ret=externalChannelsListTop-externalChannelsList+1;

    if(--n<0){
        if(n!=-1)
            return(-1);
        externalChannelsListTop=0;
        writeBufToExtChannel=&writeBufToExtChannelFailure;
        getcFromExtChannel=&getcFromExtChannelFailure;
        setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
        setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
        return(ret);
    }
    if(
        (n>=externalChannelsListFill)||
        (externalChannelsList[n].cmd==0)
    )
        return(-1);

    externalChannelsListTop=externalChannelsList+n;
    writeBufToExtChannel=&writeBufToExtChannelOk;
    getcFromExtChannel=&getcFromExtChannelOk;
    setTerminatorForExternalChannel=&setTerminatorForExternalChannelOk;
    setKillModeForExternalChannel=&setKillModeForExternalChannelOk;
    return(ret);
}/*selectExternalChannel*/
/*
    #] selectExternalChannel :
    #[ closeExternalChannel :
*/

/*
Destroys the opened external channel with the descriptor n. It returns
0 in success, or -1 on failure. If the corresponding external channel
was the current one, the current channel becomes undefined. If n==0,
the function closes the current external channel.
*/
int closeExternalChannel(int n)
{
    if(n==0)
        n=externalChannelsListTop-externalChannelsList;
    else
        n--;/*Count from 0*/

    if(
        (n<0)||
        (n>=externalChannelsListFill)||
        (externalChannelsList[n].cmd==0)
    )/*No shuch a channel*/
        return(-1);

    destroyExternalChannel(externalChannelsList+n);
    /*If the current external channel was destroyed, undefine current channel:*/
    if(externalChannelsListTop==externalChannelsList+n){
        externalChannelsListTop=NULL;
        writeBufToExtChannel=&writeBufToExtChannelFailure;
```

```
        getcFromExtChannel=&getcFromExtChannelFailure;
        setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
        setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
    }/*if(externalChannelsListTop==externalChannelsList+n)*/
    return(0);
}/*closeExternalChannel*/
/*
    #] closeExternalChannel :
    #[ closeAllExternalChannels :
*/
void closeAllExternalChannels()
{
int i;
    for(i=0; i<externalChannelsListFill; i++)
        destroyExternalChannel(externalChannelsList+i);
    externalChannelsListFill=externalChannelsListStop=0;
    externalChannelsListTop=NULL;

    writeBufToExtChannel=&writeBufToExtChannelFailure;
    getcFromExtChannel=&getcFromExtChannelFailure;
    setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
    setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;

    if(externalChannelsList!=NULL){
        M_free(externalChannelsList,"External channel list");
        externalChannelsList=NULL;
    }
}/*closeAllExternalChannels*/
/*
    #] closeAllExternalChannels :
    #[ getExternalChannelPid :
*/
#ifdef SELFTEST
pid_t getExternalChannelPid()
{
  if(externalChannelsListTop!=0)
        return(externalChannelsListTop ->pid);
  return(-1);
}/*getExternalChannelPid*/
#endif
/*
    #] getExternalChannelPid :
    #[ getCurrentExternalChannel :
*/

int getCurrentExternalChannel()
{

    if ( externalChannelsListTop != 0 )
        return(externalChannelsListTop-externalChannelsList+1);
    return(0);
}/*getCurrentExternalChannel*/
/*
    #] getCurrentExternalChannel :
    #[ Selftest main :
*/

#ifdef SELFTEST

/*
    This is the example of how all these public functions may be used:
*/

char buf[1024];
char buf2[1024];

void help(void)
{
    printf("String started with a special letter is a command\n");
    printf("Known commands are:\n");
    printf("H or ? -- this help\n");
    printf("Nn<command> -- start a new command\n");
    printf("S<command> -- start a new command in a subshell,daemon,stderr>/dev/null\n");
    printf("C# -- destroy channel #\n");
    printf("R# -- set a new cahhel(number#) as a current one\n");
    printf("K#1 #2 --  set signal for kill and kill mode (0 or !=0)\n");
    printf("   ^d to quit\n");
}/*help*/

int main (void)
{
    int i, j, k,last;
    long long sum = 0;

    /*openExternalChannel(UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)*/

    help();
```

```

    printf("Initial channel:%d\n",last=openExternalChannel((UBYTE*)"cat",0,NULL,NULL));

    if( ( i = setTerminatorForExternalChannel("qu") ) != 0 ) return 1;
    printf("Terminaror is 'qu'\n");

    while ( fgets(buf, 1024, stdin) != NULL ) {
        if ( *buf == 'N' ) {
            printf("New channel:%d\n",j=openExternalChannel((UBYTE*)buf+1,0,NULL,NULL));
            continue;
        }
        else if ( *buf == 'C' ) {
            int n;
            sscanf(buf+1,"%d",&n);
            printf("Destroy last channel:%d\n",closeExternalChannel(n));
            continue;
        }
        else if ( *buf == 'R' ) {
            int n = 0;
            sscanf(buf+1,"%d",&n);
            printf("Reopen channel %d:%d\n",n,selectExternalChannel(n));
            continue;
        }else if( *buf == 'K' ) {
            int n=0,g = 0;
            sscanf(buf+1,"%d %d",&n,&g);
            printf("setKillMode %d\n",setKillModeForExternalChannel(n,g));
            continue;
        }else if( *buf == 'S' ) {
            printf("New channel with sh&d&stderr:%d\n",
                    j=openExternalChannel((UBYTE*)buf+1,1,(UBYTE*)"/bin/sh -c",(UBYTE*)"/dev/null"));
            continue;
        }
        else if( ( *buf == 'H' )|| ( *buf == '?' )  ){
            help();
            continue;
        }

        writeBufToExtChannel(buf,k=StrLen(buf));
        sum += k;
        for ( j = 0; ( i = getcFromExtChannel() ) != '\n'; j++) {
            if ( i == EOF ) {
                printf("EOF!\n");
                break;
            }
            buf2[j] = i;
        }
        buf2[j] = '\0';
        printf("I got:'%s'; pid=%d\n",buf2,getExternalChannelPid());
    }
    printf("Total:%lld bytes\n",sum);
    closeAllExternalChannels();
    return 0;
}
#endif /*ifdef SELFTEST*/
/*
    #] Selftest main :
*/

#endif /*ifndef WITHEXTERNALCHANNEL ... else*/
```

4.29 factor.c File Reference

`#include "form3.h"`

Include dependency graph for factor.c:

Functions

- int FactorIn (PHEAD WORD *term, WORD level)
- int FactorInExpr (PHEAD WORD *term, WORD level)

4.29.1 Detailed Description

The routines for finding (one term) factors in dollars and expressions.

Definition in file factor.c.

4.29.2 Function Documentation

4.29.2.1 FactorIn()

```
int FactorIn (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 46 of file factor.c.

4.29.2.2 FactorInExpr()

```
int FactorInExpr (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 421 of file factor.c.

4.30 factor.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
```

```
/*
    #[ Includes : factor.c
*/

#include "form3.h"

/*
    #] Includes :
    #[ FactorIn :

    This routine tests for a factor in a dollar expression.

    Note that unlike with regular active or hidden expressions we cannot
    add memory as easily as dollars are rather volatile.
*/
int FactorIn(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    WORD *t, *tstop, *m, *mm, *oldwork, *mstop, *n1, *n2, *n3, *n4, *n1stop, *n2stop;
    WORD *r1, *r2, *r3, *r4, j, k, kGCD, kGCD2, kLCM, jGCD, kkLCM, jLCM, size;
    UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
    int fromwhere = 0, i;
    DOLLARS d;
    t = term; GETSTOP(t,tstop); t++;
    while ( ( t < tstop ) && ( *t != FACTORIN || ( ( *t == FACTORIN )
     && ( t[FUNHEAD] != -DOLLAREXPRESSION || t[1] != FUNHEAD+2 ) ) ) ) t += t[1];
    if ( t >= tstop ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Internal error. Could not find proper factorin_ function.");
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    oldwork = AT.WorkPointer;
    d = Dollars + t[FUNHEAD+1];
#ifdef WITHPTHREADS
    {
        int nummodopt, dtype = -1;
        if ( AS.MultiThreaded ) {
            for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                if ( t[FUNHEAD+1] == ModOptdollars[nummodopt].number ) break;
            }
            if ( nummodopt < NumModOptdollars ) {
                dtype = ModOptdollars[nummodopt].type;
                if ( dtype == MODLOCAL ) {
                    d = ModOptdollars[nummodopt].dstruct+AT.identity;
                }
            }
        }
    }
#endif

    if ( d->type == DOLTERMS ) {
        fromwhere = 1;
    }
    else if ( ( d = DolToTerms(BHEAD t[FUNHEAD+1]) ) == 0 ) {
/*
        The variable cannot convert to an expression
        We replace the function by 1.
*/
        m = oldwork; n1 = term;
        while ( n1 < t ) *m++ = *n1++;
        n1 = t + t[1]; tstop = term + *term;
        while ( n1 < tstop ) *m++ = *n1++;
        *oldwork = m - oldwork;
        AT.WorkPointer = m;
        if ( Generator(BHEAD oldwork,level) ) return(-1);
        AT.WorkPointer = oldwork;
        return(0);
    }
    if ( d->where[0] == 0 ) {
        if ( fromwhere == 0 ) {
            if ( d->factors ) M_free(d->factors,"Dollar factors");
            M_free(d,"Dollar in FactorIn_");
        }
        return(0);
    }
/*
    Now we have an expression in d->where. Find the symbolic factor that
    divides the expression and the numerical factor that makes all
    coefficients integer.

    For the symbolic factor we make a copy of the first term, and then
    go through all terms, scratching in the copy the objects that do not
    occur in the terms.
*/
    m = oldwork;
    mm = d->where;
```

```
    k = *mm - ABS((mm[*mm-1]));
    for ( j = 0; j < k; j++ ) *m++ = *mm++;
    mstop = m;
    *oldwork = k;
/*
    The copy is in place. Now search through the terms. Start at the second term
*/
    mm = d->where + d->where[0];
    while ( *mm ) {
        m = oldwork+1;
        r2 = mm+*mm;
        r2 -= ABS(r2[-1]);
        r1 = mm+1;
        while ( m < mstop ) {
            while ( r1 < r2 ) {
                if ( *r1 != *m ) {
                    r1 += r1[1]; continue;
                }
/*
                Now the various cases
            #[ SYMBOL :
*/
                if ( *m == SYMBOL ) {
                    n1 = m+2; n1stop = m+m[1];
                    n2stop = r1+r1[1];
                    while ( n1 < n1stop ) {
                        n2 = r1+2;
                        while ( n2 < n2stop ) {
                            if ( *n1 != *n2 ) { n2 += 2; continue; }
                            if ( n1[1] > 0 ) {
                                if ( n2[1] < 0 ) { n2 += 2; continue; }
                                if ( n2[1] < n1[1] ) n1[1] = n2[1];
                            }
                            else {
                                if ( n2[1] > 0 ) { n2 += 2; continue; }
                                if ( n2[1] > n1[1] ) n1[1] = n2[1];
                            }
                            break;
                        }
                        if ( n2 >= n2stop ) {   /* scratch symbol */
                            if ( m[1] == 4 ) goto scratch;
                            m[1] -= 2;
                            n3 = n1; n4 = n1+2;
                            while ( n4 < mstop ) *n3++ = *n4++;
                            *oldwork = n3 - oldwork;
                            mstop -= 2; n1stop -= 2;
                            continue;
                        }
                        n1 += 2;
                    }
                    break;
                }
/*
            #] SYMBOL :
            #[ DOTPRODUCT :
*/
                else if ( *m == DOTPRODUCT ) {
                    n1 = m+2; n1stop = m+m[1];
                    n2stop = r1+r1[1];
                    while ( n1 < n1stop ) {
                        n2 = r1+2;
                        while ( n2 < n2stop ) {
                            if ( *n1 != *n2 || n1[1] != n2[1] ) { n2 += 3; continue; }
                            if ( n1[2] > 0 ) {
                                if ( n2[2] < 0 ) { n2 += 3; continue; }
                                if ( n2[2] < n1[2] ) n1[2] = n2[2];
                            }
                            else {
                                if ( n2[2] > 0 ) { n2 += 3; continue; }
                                if ( n2[2] > n1[2] ) n1[2] = n2[2];
                            }
                            break;
                        }
                        if ( n2 >= n2stop ) {   /* scratch symbol */
                            if ( m[1] == 5 ) goto scratch;
                            m[1] -= 3;
                            n3 = n1; n4 = n1+3;
                            while ( n4 < mstop ) *n3++ = *n4++;
                            *oldwork = n3 - oldwork;
                            mstop -= 3; n1stop -= 3;
                            continue;
                        }
                        n1 += 3;
                    }
                    break;
                }
/*
```

```
            #] DOTPRODUCT :
            #[ VECTOR :
*/
                else if ( *m == VECTOR ) {
/*
                    Here we have to be careful if there is more than
                    one of the same
*/
                    n1 = m+2; n1stop = m+m[1];
                    n2 = r1+2;n2stop = r1+r1[1];
                    while ( n1 < n1stop ) {
                        while ( n2 < n2stop ) {
                            if ( *n1 == *n2 && n1[1] == n2[1] ) {
                                n2 += 2; goto nextn1;
                            }
                            n2 += 2;
                        }
                        if ( n2 >= n2stop ) {   /* scratch symbol */
                            if ( m[1] == 4 ) goto scratch;
                            m[1] -= 2;
                            n3 = n1; n4 = n1+2;
                            while ( n4 < mstop ) *n3++ = *n4++;
                            *oldwork = n3 - oldwork;
                            mstop -= 2; n1stop -= 2;
                            continue;
                        }
                        n2 = r1+2;
                        n1 += 2;
nextn1:                 }
                    break;
                }
/*
            #] VECTOR :
            #[ REMAINDER :
*/
                else {
/*
                    Again: watch for multiple occurrences of the same object
*/
                    if ( m[1] != r1[1] ) { r1 += r1[1]; continue; }
                    for ( j = 2; j < m[1]; j++ ) {
                        if ( m[j] != r1[j] ) break;
                    }
                    if ( j < m[1] ) { r1 += r1[1]; continue; }
                    r1 += r1[1]; /* to restart at the next potential match */
                    goto nextm;  /* match */
                }
/*
            #] REMAINDER :
*/
            }
            if ( r1 >= r2 ) { /* no factor! */
scratch:;
                r3 = m + m[1]; r4 = m;
                while ( r3 < mstop ) *r4++ = *r3++;
                *oldwork = r4 - oldwork;
                if ( *oldwork == 1 ) goto nofactor;
                mstop = r4;
                r1 = mm + 1;
                continue;
            }
            r1 = mm + 1;
nextm:      m += m[1];
        }
        mm = mm + *mm;
    }

nofactor:;
/*
    For the coefficient we have to determine the LCM of the denominators
    and the GCD of the numerators.
*/
    GCDbuffer = NumberMalloc("FactorIn"); GCDbuffer2 = NumberMalloc("FactorIn");
    LCMbuffer = NumberMalloc("FactorIn"); LCMb = NumberMalloc("FactorIn"); LCMc =
NumberMalloc("FactorIn");
    r1 = d->where;
/*
    First take the first term to load up the LCM and the GCD
*/
    r2 = r1 + *r1;
    j = r2[-1];
    r3 = r2 - ABS(j);
    k = REDLENG(j);
    if ( k < 0 ) k = -k;
    while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
    for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
    k = REDLENG(j);
```

```
    if ( k < 0 ) k = -k;
    r3 += k;
    while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
    for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
    r1 = r2;
/*
    Now go through the rest of the terms in this dollar buffer.
*/
    while ( *r1 ) {
        r2 = r1 + *r1;
        j = r2[-1];
        r3 = r2 - ABS(j);
        k = REDLENG(j);
        if ( k < 0 ) k = -k;
        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
        if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
/*
            GCD is already 1
*/
        }
        else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
            if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
                goto onerror;
            }
            kGCD = kGCD2;
            for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
        }
        else {
            kGCD = 1; GCDbuffer[0] = 1;
        }
        k = REDLENG(j);
        if ( k < 0 ) k = -k;
        r3 += k;
        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
        if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
            for ( kLCM = 0; kLCM < k; kLCM++ )
                LCMbuffer[kLCM] = r3[kLCM];
        }
        else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
            if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
                goto onerror;
            }
            DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
            MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
            for ( kLCM = 0; kLCM < jLCM; kLCM++ )
                LCMbuffer[kLCM] = LCMc[kLCM];
        }
        else {} /* LCM doesn't change */
        r1 = r2;
    }
/*
    Now put the factor together: GCD/LCM
*/
    r3 = (WORD *)(GCDbuffer);
    if ( kGCD == kLCM ) {
        for ( jGCD = 0; jGCD < kGCD; jGCD++ )
            r3[jGCD+kGCD] = LCMbuffer[jGCD];
        k = kGCD;
    }
    else if ( kGCD > kLCM ) {
        for ( jGCD = 0; jGCD < kLCM; jGCD++ )
            r3[jGCD+kGCD] = LCMbuffer[jGCD];
        for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
            r3[jGCD+kGCD] = 0;
        k = kGCD;
    }
    else {
        for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
            r3[jGCD] = 0;
        for ( jGCD = 0; jGCD < kLCM; jGCD++ )
            r3[jGCD+kLCM] = LCMbuffer[jGCD];
        k = kLCM;
    }
    j = 2*k+1;
    mm = m = oldwork + oldwork[0];
/*
    Now compose the new term
*/
    n1 = term;
    while ( n1 < t ) *m++ = *n1++;
    n1 += n1[1];
    n2 = oldwork+1;
    while ( n2 < mm ) *m++ = *n2++;
    while ( n1 < tstop ) *m++ = *n1++;
/*
    And the coefficient
*/
```

```
    size = term[*term-1];
    size = REDLENG(size);
    if ( MulRat(BHEAD (UWORD *)tstop,size,(UWORD *)r3,k,
                                (UWORD *)m,&size) ) goto onerror;
    size = INCLENG(size);
    k = size < 0 ? -size: size;
    m[k-1] = size;
    m += k;
    *mm = (WORD)(m - mm);
    AT.WorkPointer = m;
    if ( Generator(BHEAD mm,level) ) goto onerror;
    AT.WorkPointer = oldwork;
    if ( fromwhere == 0 ) {
        if ( d->factors ) M_free(d->factors,"Dollar factors");
        M_free(d,"Dollar in FactorIn");
    }
    NumberFree(GCDbuffer,"FactorIn"); NumberFree(GCDbuffer2,"FactorIn");
    NumberFree(LCMbuffer,"FactorIn"); NumberFree(LCMb,"FactorIn"); NumberFree(LCMc,"FactorIn");
    return(0);
onerror:
    AT.WorkPointer = oldwork;
    MLOCK(ErrorMessageLock);
    MesCall("FactorIn");
    MUNLOCK(ErrorMessageLock);
    NumberFree(GCDbuffer,"FactorIn"); NumberFree(GCDbuffer2,"FactorIn");
    NumberFree(LCMbuffer,"FactorIn"); NumberFree(LCMb,"FactorIn"); NumberFree(LCMc,"FactorIn");
    return(-1);
}

/*
    #] FactorIn :
    #[ FactorInExpr :

    This routine tests for a factor in an active or hidden expression.

    The factor from the last call is stored in a cache.
    Main problem here is whether the cache is global or private to each thread.
    A global cache gives most likely the same traffic jam for the computation
    as a local cache. The local cache may stay valid longer.
    In the future we may make it such that threads can look at the cache
    of the others, or even whether a certain factor is under construction.
*/

int FactorInExpr(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    WORD *t, *tstop, *m, *oldwork, *mstop, *n1, *n2, *n3, *n4, *n1stop, *n2stop;
    WORD *r1, *r2, *r3, *r4, j, k, kGCD, kGCD2, kLCM, jGCD, kkLCM, jLCM, size, sign;
    WORD *newterm, expr = 0;
    WORD olddeferflag = AR.DeferFlag, oldgetfile = AR.GetFile, oldhold = AR.KeptInHold;
    WORD newgetfile, newhold;
    int i;
    EXPRESSIONS e;
    FILEHANDLE *file = 0;
    POSITION position, oldposition, startposition;
    WORD *oldcpointer = AR.CompressPointer;
    UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
    GCDbuffer = NumberMalloc("FactorInExpr"); GCDbuffer2 = NumberMalloc("FactorInExpr");
    LCMbuffer = NumberMalloc("FactorInExpr"); LCMb = NumberMalloc("FactorInExpr"); LCMc =
NumberMalloc("FactorInExpr");
    t = term; GETSTOP(t,tstop); t++;
    while ( t < tstop ) {
        if ( *t == FACTORIN && t[1] == FUNHEAD+2 && t[FUNHEAD] == -EXPRESSION ) {
            expr = t[FUNHEAD+1];
            break;
        }
        t += t[1];
    }
    if ( t >= tstop ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Internal error. Could not find proper factorin_ function.");
        MUNLOCK(ErrorMessageLock);
        NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
        NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
        return(-1);
    }
    oldwork = AT.WorkPointer;
    if ( AT.previousEfactor && ( expr == AT.previousEfactor[0] ) ) {
/*
        We have a hit in the cache. Construct the new term.
        At the moment AT.previousEfactor[1] is reserved for future flags
*/
        goto PutTheFactor;
    }
/*
    No hit. We have to do the work. We start with constructing the factor
```

```
    in the WorkSpace. Later we will move it to the cache.
    Finally we will jump to PutTheFactor.
*/
    e = Expressions + expr;
    switch ( e->status ) {
        case LOCALEXPRESSION:
        case SKIPLEXPRESSION:
        case DROPLEXPRESSION:
        case GLOBALEXPRESSION:
        case SKIPGEXPRESSION:
        case DROPGEXPRESSION:
/*
            Expression is to be found in the input Scratch file.
            Set the file handle and the position.
            The rest is done by GetTerm.
*/
            newhold = 0;
            newgetfile = 1;
            file = AR.infile;
            break;
        case HIDDENLEXPRESSION:
        case HIDDENGEXPRESSION:
        case HIDELEXPRESSION:
        case HIDEGEXPRESSION:
        case DROPHLEXPRESSION:
        case DROPHGEXPRESSION:
        case UNHIDELEXPRESSION:
        case UNHIDEGEXPRESSION:
/*
            Expression is to be found in the hide Scratch file.
            Set the file handle and the position.
            The rest is done by GetTerm.
*/
            newhold = 0;
            newgetfile = 2;
            file = AR.hidefile;
            break;
        case STOREDEXPRESSION:
/*
            This is an 'illegal' case
*/
            MLOCK(ErrorMessageLock);
            MesPrint("Error: factorin_ cannot determine factors in stored expressions.");
            MUNLOCK(ErrorMessageLock);
            NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
            NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
            return(-1);
        case DROPPEDEXPRESSION:
/*
            We replace the function by 1.
*/
            m = oldwork; n1 = term;
            while ( n1 < t ) *m++ = *n1++;
            n1 = t + t[1]; tstop = term + *term;
            while ( n1 < tstop ) *m++ = *n1++;
            *oldwork = m - oldwork;
            AT.WorkPointer = m;
            if ( Generator(BHEAD oldwork,level) ) {
                NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
                NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
                return(-1);
            }
            AT.WorkPointer = oldwork;
            NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
            NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
            return(0);
        default:
            MLOCK(ErrorMessageLock);
            MesPrint("Error: Illegal expression in factorinexpr.");
            MUNLOCK(ErrorMessageLock);
            NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
            NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
            return(-1);
    }
/*
    Before we start with the file we set the buffers for the coefficient
    For the coefficient we have to determine the LCM of the denominators
    and the GCD of the numerators.
*/
    position = AS.OldOnFile[expr];
    AR.DeferFlag = 0; AR.KeptInHold = newhold; AR.GetFile = newgetfile;
    SeekScratch(file,&oldposition);
    SetScratch(file,&position);
```

```
    if ( GetTerm(BHEAD oldwork) <= 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("(5) Expression %d has problems in scratchfile",expr);
        MUNLOCK(ErrorMessageLock);
        NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
        NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
        return(-1);
    }
    SeekScratch(file,&startposition);
    SeekScratch(file,&position);
/*
    Load the first term in the workspace
*/
    if ( GetTerm(BHEAD oldwork) == 0 ) {
        SetScratch(file,&oldposition); /* We still need this untill Processor is clean */
        AR.DeferFlag = olddeferflag;
        oldwork[0] = 4; oldwork[1] = 1; oldwork[2] = 1; oldwork[3] = 3;
        goto Complete;
    }
    SeekScratch(file,&position);
    AR.DeferFlag = olddeferflag; AR.KeptInHold = oldhold; AR.GetFile = oldgetfile;

    r2 = m = oldwork + *oldwork;
    j = m[-1];
    m -= ABS(j);
    *oldwork = (WORD)(m-oldwork);
    AT.WorkPointer = newterm = mstop = m;
/*
    Now take the coefficient of the first term to load up the LCM and the GCD
*/
    r3 = m;
    k = REDLENG(j);
    if ( k < 0 ) { k = -k; sign = -1; }
    else { sign = 1; }
    while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
    for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
    k = REDLENG(j);
    if ( k < 0 ) k = -k;
    r3 += k;
    while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
    for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
/*
    The copy and the coefficient are in place. Now search through the terms.
*/
    for (;;) {
        AR.DeferFlag = 0; AR.KeptInHold = newhold; AR.GetFile = newgetfile;
        SetScratch(file,&position);
        size = GetTerm(BHEAD newterm);
        SeekScratch(file,&position);
        AR.DeferFlag = olddeferflag; AR.KeptInHold = oldhold; AR.GetFile = oldgetfile;
        if ( size == 0 ) break;
        m = oldwork+1;
        r2 = newterm + *newterm;
        r2 -= ABS(r2[-1]);
        r1 = newterm+1;
        while ( m < mstop ) {
            while ( r1 < r2 ) {
                if ( *r1 != *m ) {
                    r1 += r1[1]; continue;
                }
/*
                Now the various cases
            #[ SYMBOL :
*/
                if ( *m == SYMBOL ) {
                    n1 = m+2; n1stop = m+m[1];
                    n2stop = r1+r1[1];
                    while ( n1 < n1stop ) {
                        n2 = r1+2;
                        while ( n2 < n2stop ) {
                            if ( *n1 != *n2 ) { n2 += 2; continue; }
                            if ( n1[1] > 0 ) {
                                if ( n2[1] < 0 ) { n2 += 2; continue; }
                                if ( n2[1] < n1[1] ) n1[1] = n2[1];
                            }
                            else {
                                if ( n2[1] > 0 ) { n2 += 2; continue; }
                                if ( n2[1] > n1[1] ) n1[1] = n2[1];
                            }
                            break;
                        }
                        if ( n2 >= n2stop ) {   /* scratch symbol */
                            if ( m[1] == 4 ) goto scratch;
                            m[1] -= 2;
                            n3 = n1; n4 = n1+2;
                            while ( n4 < mstop ) *n3++ = *n4++;
```

```
                            *oldwork = n3 - oldwork;
                            mstop -= 2; n1stop -= 2;
                            continue;
                        }
                        n1 += 2;
                    }
                    break;
                }
/*
            #] SYMBOL :
            #[ DOTPRODUCT :
*/
                else if ( *m == DOTPRODUCT ) {
                    n1 = m+2; n1stop = m+m[1];
                    n2stop = r1+r1[1];
                    while ( n1 < n1stop ) {
                        n2 = r1+2;
                        while ( n2 < n2stop ) {
                            if ( *n1 != *n2 || n1[1] != n2[1] ) { n2 += 3; continue; }
                            if ( n1[2] > 0 ) {
                                if ( n2[2] < 0 ) { n2 += 3; continue; }
                                if ( n2[2] < n1[2] ) n1[2] = n2[2];
                            }
                            else {
                                if ( n2[2] > 0 ) { n2 += 3; continue; }
                                if ( n2[2] > n1[2] ) n1[2] = n2[2];
                            }
                            break;
                        }
                        if ( n2 >= n2stop ) {   /* scratch dotproduct */
                            if ( m[1] == 5 ) goto scratch;
                            m[1] -= 3;
                            n3 = n1; n4 = n1+3;
                            while ( n4 < mstop ) *n3++ = *n4++;
                            *oldwork = n3 - oldwork;
                            mstop -= 3; n1stop -= 3;
                            continue;
                        }
                        n1 += 3;
                    }
                    break;
                }
/*
            #] DOTPRODUCT :
            #[ VECTOR :
*/
                else if ( *m == VECTOR ) {
/*
                    Here we have to be careful if there is more than
                    one of the same
*/
                    n1 = m+2; n1stop = m+m[1];
                    n2 = r1+2;n2stop = r1+r1[1];
                    while ( n1 < n1stop ) {
                        while ( n2 < n2stop ) {
                            if ( *n1 == *n2 && n1[1] == n2[1] ) {
                                n2 += 2; goto nextn1;
                            }
                            n2 += 2;
                        }
                        if ( n2 >= n2stop ) {   /* scratch vector */
                            if ( m[1] == 4 ) goto scratch;
                            m[1] -= 2;
                            n3 = n1; n4 = n1+2;
                            while ( n4 < mstop ) *n3++ = *n4++;
                            *oldwork = n3 - oldwork;
                            mstop -= 2; n1stop -= 2;
                            continue;
                        }
                        n2 = r1+2;
nextn1:                 n1 += 2;
                    }
                    break;
                }
/*
            #] VECTOR :
            #[ REMAINDER :
*/
                else {
/*
                    Again: watch for multiple occurrences of the same object
*/
                    if ( m[1] != r1[1] ) { r1 += r1[1]; continue; }
                    for ( j = 2; j < m[1]; j++ ) {
                        if ( m[j] != r1[j] ) break;
                    }
                    if ( j < m[1] ) { r1 += r1[1]; continue; }
```

```
                    r1 += r1[1]; /* to restart at the next potential match */
                    goto nextm;  /* match */
                }
/*
            #] REMAINDER :
*/
            }
            if ( r1 >= r2 ) { /* no factor! */
scratch:;
                r3 = m + m[1]; r4 = m;
                while ( r3 < mstop ) *r4++ = *r3++;
                *oldwork = r4 - oldwork;
                if ( *oldwork == 1 ) goto nofactor;
                mstop = r4;
                r1 = newterm + 1;
                continue;
            }
            r1 = newterm + 1;
nextm:      m += m[1];
        }
nofactor:;
/*
        Now the coefficient
*/
        r2 = newterm + *newterm;
        j = r2[-1];
        r3 = r2 - ABS(j);
        k = REDLENG(j);
        if ( k < 0 ) k = -k;
        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
        if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
/*
            GCD is already 1
*/
        }
        else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
            if ( GcdLong(BHEAD GCDbuffer,kGCD,(UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
                goto onerror;
            }
            kGCD = kGCD2;
            for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
        }
        else {
            kGCD = 1; GCDbuffer[0] = 1;
        }
        k = REDLENG(j);
        if ( k < 0 ) k = -k;
        r3 += k;
        while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
        if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
            for ( kLCM = 0; kLCM < k; kLCM++ )
                LCMbuffer[kLCM] = r3[kLCM];
        }
        else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
            if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kkLCM) ) {
                goto onerror;
            }
            DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
            MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
            for ( kLCM = 0; kLCM < jLCM; kLCM++ )
                LCMbuffer[kLCM] = LCMc[kLCM];
        }
        else {} /* LCM doesn't change */
    }
    SetScratch(file,&oldposition); /* Needed until Processor is thread safe */
    AR.DeferFlag = olddeferflag;
/*
    Now put the term together in oldwork: GCD/LCM
    We have already the algebraic contents there.
*/
    r3 = (WORD *)(GCDbuffer);
    r4 = (WORD *)(LCMbuffer);
    r1 = oldwork + *oldwork;
    if ( kGCD == kLCM ) {
        for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r3++;
        for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r4++;
        k = 2*kGCD+1;
    }
    else if ( kGCD > kLCM ) {
        for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r3++;
        for ( jGCD = 0; jGCD < kLCM; jGCD++ ) *r1++ = *r4++;
        for ( jGCD = kLCM; jGCD < kGCD; jGCD++ ) *r1++ = 0;
        k = 2*kGCD+1;
    }
    else {
        for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r3++;
        for ( jGCD = kGCD; jGCD < kLCM; jGCD++ ) *r1++ = 0;
```

```
        for ( jGCD = 0; jGCD < kLCM; jGCD++ ) *r1++ = *r4++;
        k = 2*kLCM+1;
    }
    if ( sign < 0 ) *r1++ = -k;
    else *r1++ = k;
    *oldwork = (WORD)(r1-oldwork);
/*
    Now put the new term in the cache
*/
Complete:;
    if ( AT.previousEfactor ) M_free(AT.previousEfactor,"Efactor cache");
    AT.previousEfactor = (WORD *)Malloc1((*oldwork+2)*sizeof(WORD),"Efactor cache");
    AT.previousEfactor[0] = expr;
    r1 = oldwork; r2 = AT.previousEfactor + 2; k = *oldwork;
    NCOPY(r2,r1,k)
    AT.previousEfactor[1] = 0;
/*
    Now we construct the new term in the workspace.
*/
PutTheFactor:;
    if ( AT.WorkPointer + AT.previousEfactor[2] >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MesPrint("Called from factorin_");
        MUNLOCK(ErrorMessageLock);
        NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
        NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
        return(-1);
    }
    n1 = oldwork; n2 = term; while ( n2 < t ) *n1++ = *n2++;
    n2 = AT.previousEfactor+2; GETSTOP(n2,n2stop); n3 = n2 + *n2;
    n2++; while ( n2 < n2stop ) *n1++ = *n2++;
    n2 = t + t[1]; while ( n2 < tstop ) *n1++ = *n2++;
    size = term[*term-1];
    size = REDLENG(size);
    k = n3[-1]; k = REDLENG(k);
    if ( MulRat(BHEAD (UWORD *)tstop,size,(UWORD *)n2stop,k,
                            (UWORD *)n1,&size) ) goto onerror;
    size = INCLENG(size);
    k = size < 0 ? -size: size;
       n1 += k; n1[-1] = size;
    *oldwork = n1 - oldwork;
    AT.WorkPointer = n1;
    if ( Generator(BHEAD oldwork,level) ) {
        NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
        NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
        return(-1);
    }
    AT.WorkPointer = oldwork;
    AR.CompressPointer = oldcpointer;
    NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
    NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
    return(0);
onerror:
    AT.WorkPointer = oldwork;
    AR.CompressPointer = oldcpointer;
    MLOCK(ErrorMessageLock);
    MesCall("FactorInExpr");
    MUNLOCK(ErrorMessageLock);
    NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
    NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
NumberFree(LCMc,"FactorInExpr");
    return(-1);
}

/*
    #] FactorInExpr :
*/
```

4.31 findpat.c File Reference

`#include "form3.h"`
Include dependency graph for findpat.c:

Functions

- WORD FindOnly (PHEAD WORD ∗term, WORD ∗pattern)
- WORD FindOnce (PHEAD WORD ∗term, WORD ∗pattern)
- WORD FindMulti (PHEAD WORD ∗term, WORD ∗pattern)
- WORD FindRest (PHEAD WORD ∗term, WORD ∗pattern)

4.31.1 Detailed Description

Pattern matching of symbols and dotproducts. There are various routines because of the options in the id-statements like once, only, multi and many. These are amoung the oldest routines in FORM and that can be noticed, because the interplay with the function matching is not complete. When we match functions and halfway we fail we can backtrack properly. With the symbols, the dotproducts and the vectors (in pattern.c) there is no proper backtracking. Hence the routines here need still quite some work or may even have to be rewritten.

Definition in file findpat.c.

4.31.2 Function Documentation

4.31.2.1 FindOnly()

```
WORD FindOnly (
            PHEAD WORD * term,
            WORD * pattern )
```

Definition at line 61 of file findpat.c.

4.31.2.2 FindOnce()

```
WORD FindOnce (
            PHEAD WORD * term,
            WORD * pattern )
```

Definition at line 419 of file findpat.c.

4.31.2.3 FindMulti()

```
WORD FindMulti (
            PHEAD WORD * term,
            WORD * pattern )
```

Definition at line 954 of file findpat.c.

4.31.2.4 FindRest()

```
WORD FindRest (
            PHEAD WORD * term,
            WORD * pattern )
```

Definition at line 1070 of file findpat.c.

4.32 findpat.c

Go to the documentation of this file.

```
00001
00013 /* #[ License : */
00014 /*
00015  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00016  *   When using this file you are requested to refer to the publication
00017  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00018  *   This is considered a matter of courtesy as the development was paid
00019  *   for by FOM the Dutch physics granting agency and we would like to
00020  *   be able to track its scientific use to convince FOM of its value
00021  *   for the community.
00022  *
00023  *   This file is part of FORM.
00024  *
00025  *   FORM is free software: you can redistribute it and/or modify it under the
00026  *   terms of the GNU General Public License as published by the Free Software
00027  *   Foundation, either version 3 of the License, or (at your option) any later
00028  *   version.
00029  *
00030  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00031  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00032  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00033  *   details.
00034  *
00035  *   You should have received a copy of the GNU General Public License along
00036  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00037  */
00038 /* #] License : */
00039 /*
00040     #[ Includes : findpat.c
00041 */
00042
00043 #include "form3.h"
00044
00045 /*
00046     #] Includes :
00047     #[ Patterns :
00048         #[ FindOnly :           WORD FindOnly(term,pattern)
00049
00050     The current version doesn't scan function arguments yet. 10-Apr-1988
00051
00052     This routine searches for an exact match. This means in particular:
00053     1:  x^# must match exactly.
00054     2:  x^n? must have a single value for n that cannot be addapted.
00055
00056     When setp != 0 it points to a collection of sets
00057     A match can occur only if no object will be left that belongs
00058     to any of these sets.
00059 */
00060
00061 WORD FindOnly(PHEAD WORD *term, WORD *pattern)
00062 {
00063     GETBIDENTITY
```

```
    WORD *t, *m;
    WORD *tstop, *mstop;
    WORD *xstop, *ystop, *setp = AN.ForFindOnly;
    WORD n, nt, *p, nq;
    WORD older[NORMSIZE], *q, newval1, newval2, newval3;
    AN.UsedOtherFind = 0;
    m = pattern;
    mstop = m + *m;
    m++;
    t = term;
    t += *term - 1;
    tstop = t - ABS(*t) + 1;
    t = term;
    t++;
    while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
    while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
    if ( m < mstop ) { do {
/*
            #[ SYMBOLS :
*/
        if ( *m == SYMBOL ) {
            ystop = m + m[1];
            m += 2;
            n = 0;
            p = older;
            if ( t < tstop ) while ( *t != SYMBOL ) {
                t += t[1];
                if ( t >= tstop ) {
OnlyZer1:
                    do {
                        if ( *m >= 2*MAXPOWER ) return(0);
                        if ( m[1] >= 2*MAXPOWER ) nt = m[1];
                        else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
                        else return(0);
                        nt -= 2*MAXPOWER;
                        if ( CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) return(0);
                        AddWild(BHEAD nt,SYMTONUM,0);
                        m += 2;
                    } while ( m < ystop );
                    goto EndLoop;
                }
            }
            else goto OnlyZer1;
            xstop = t + t[1];
            t += 2;
            do {
                if ( *m == *t && t < xstop ) {
                    if ( m[1] == t[1] ) { m += 2; t += 2; }
                    else if ( m[1] >= 2*MAXPOWER ) {
                        nt = t[1];
                        nq = m[1];
                        goto OnlyL2;
                    }
                    else if ( m[1] <= -2*MAXPOWER ) {
                        nt = -t[1];
                        nq = -m[1];
OnlyL2:                 nq -= 2*MAXPOWER;
                        if ( CheckWild(BHEAD nq,SYMTONUM,nt,&newval3) ) return(0);
                        AddWild(BHEAD nq,SYMTONUM,nt);
                        m += 2;
                        t += 2;
                    }
                    else {
                        *p++ = *t++; *p++ = *t++; n += 2;
                    }
                }
                else if ( *m >= 2*MAXPOWER ) {
                    while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
                    nq = n;
                    p = older;
                    while ( nq > 0 ) {
                        if ( !CheckWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,*p,&newval1) ) {
                            if ( m[1] == p[1] ) {
                                AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
                                break;
                            }
                            else if ( m[1] >= 2*MAXPOWER && m[1] != *m ) {
                                if ( !CheckWild(BHEAD m[1]-2*MAXPOWER,SYMTONUM,p[1],&newval3) ) {
                                    AddWild(BHEAD m[1]-2*MAXPOWER,SYMTONUM,p[1]);
                                    AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
                                    break;
                                }
                            }
                            else if ( m[1] <= -2*MAXPOWER && m[1] != -(*m) ) {
                                if ( !CheckWild(BHEAD -m[1]-2*MAXPOWER,SYMTONUM,-p[1],&newval3) ) {
                                    AddWild(BHEAD -m[1]-2*MAXPOWER,SYMTONUM,-p[1]);
                                    AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
```

```
                                    break;
                                }
                            }
                        }
                        nq -= 2;
                        p += 2;
                    }
                    if ( nq <= 0 ) return(0);
                    nq -= 2;
                    n -= 2;
                    q = p + 2;
                    while ( --nq >= 0 ) *p++ = *q++;
                    m += 2;
                }
                else {
                    if ( t >= xstop || *m < *t ) {
                        if ( m[1] >= 2*MAXPOWER ) nt = m[1];
                        else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
                        else return(0);
                        nt -= 2*MAXPOWER;
                        if ( CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) return(0);
                        AddWild(BHEAD nt,SYMTONUM,0);
                        m += 2;
                    }
                    else {
                        *p++ = *t++; *p++ = *t++; n += 2;
                    }
                }
            } while ( m < ystop );
            if ( setp ) {
                while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n+= 2; }
                p = older;
                while ( n > 0 ) {
                    nq = setp[1] - 2;
                    m = setp + 2;
                    while ( --nq >= 0 ) {
                        if ( Sets[*m].type != CSYMBOL ) { m++; continue; }
                        t = SetElements + Sets[*m].first;
                        tstop = SetElements + Sets[*m].last;
                        while ( t < tstop ) {
                            if ( *t++ == *p ) return(0);
                        }
                        m++;
                    }
                    n -= 2;
                    p += 2;
                }
            }
            return(1);
        }
/*
            #] SYMBOLS :
            #[ DOTPRODUCTS :
*/
        else if ( *m == DOTPRODUCT ) {
            ystop = m + m[1];
            m += 2;
            n = 0;
            p = older;
            if ( t < tstop ) {
                if ( *t < DOTPRODUCT ) goto OnlyZer2;
                while ( *t > DOTPRODUCT ) {
                t += t[1];
                if ( t >= tstop || *t < DOTPRODUCT ) {
OnlyZer2:
                    do {
                        if ( *m >= (AM.OffsetVector+WILDOFFSET)
                        || m[1] >= (AM.OffsetVector+WILDOFFSET) ) return(0);
                        if ( m[2] >= 2*MAXPOWER ) nq = m[2];
                        else if ( m[2] <= -2*MAXPOWER ) nq = -m[2];
                        else return(0);
                        nq -= 2*MAXPOWER;
                        if ( CheckWild(BHEAD nq,SYMTONUM,0,&newval3) ) return(0);
                        AddWild(BHEAD nq,SYMTONUM,0);
                        m += 3;
                    } while ( m < ystop );
                    goto EndLoop;
                }
                }
            }
            else goto OnlyZer2;
            xstop = t + t[1];
            t += 2;
            do {
                if ( *m == *t && m[1] == t[1] && t < xstop ) {
                    if ( t[2] != m[2] ) {
                        if ( m[2] >= 2*MAXPOWER ) {
```

```
                            nq = m[2];
                            nt = t[2];
                        }
                        else if ( m[2] <= -2*MAXPOWER ) {
                            nq = -m[2];
                            nt = -t[2];
                        }
                        else return(0);
                        nq -= 2*MAXPOWER;
                        if ( CheckWild(BHEAD nq,SYMTONUM,nt,&newval3) ) return(0);
                        AddWild(BHEAD nq,SYMTONUM,nt);
                    }
                    t += 3; m += 3;
                }
                else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
                    while ( t < xstop ) {
                        *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
                    }
                    nq = n;
                    p = older;
                    while ( nq > 0 ) {
                        if ( *m == m[1] ) {
                            if ( *p != p[1] ) goto NextInDot;
                        }
                        if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*p,&newval1) &&
                             !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,p[1],&newval2) ) {
                            if ( p[2] == m[2] ) {
OnlyL9:                         AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval2);
                                AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
                                break;
                            }
                            if ( m[2] >= 2*MAXPOWER ) {
                                if ( !CheckWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2],&newval3) ) {
                                    AddWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,newval3);
                                    goto OnlyL9;
                                }
                            }
                            else if ( m[2] <= -2*MAXPOWER ) {
                                if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2],&newval3) ) {
                                    AddWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2]);
                                    goto OnlyL9;
                                }
                            }
                        }
                        if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,p[1],&newval1) &&
                             !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*p,&newval2) ) {
                            if ( p[2] == m[2] ) {
OnlyL10:                        AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
                                AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval2);
                                break;
                            }
                            if ( m[2] >= 2*MAXPOWER ) {
                                if ( !CheckWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2],&newval3) ) {
                                    AddWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2]);
                                    goto OnlyL10;
                                }
                            }
                            else if ( m[2] <= -2*MAXPOWER ) {
                                if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2],&newval3) ) {
                                    AddWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2]);
                                    goto OnlyL10;
                                }
                            }
                        }
NextInDot:
                        p += 3; nq -= 3;
                    }
                    if ( nq <= 0 ) return(0);
                    q = p+3;
                    nq -= 3;
                    n -= 3;
                    while ( --nq >= 0 ) *p++ = *q++;
                    m += 3;
                }
                else if ( m[1] >= (AM.OffsetVector+WILDOFFSET) ) {
                    while ( *m >= *t && t < xstop ) {
                        *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
                    }
                    nq = n;
                    p = older;
                    while ( nq > 0 ) {
                        if ( *m == *p && !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,p[1],&newval1) ) {
                            if ( p[2] == m[2] ) {
                                AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                break;
                            }
                            else if ( m[2] >= 2*MAXPOWER ) {
```

```
                                if ( !CheckWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2],&newval3) ) {
                                    AddWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2]);
                                    AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                    break;
                                }
                            }
                            else if ( m[2] <= -2*MAXPOWER ) {
                                if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2],&newval3) ) {
                                    AddWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2]);
                                    AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                    break;
                                }
                            }
                        }
                        if ( *m == p[1] && !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*p,&newval1) ) {
                            if ( p[2] == m[2] ) {
                                AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                break;
                            }
                            if ( m[2] >= 2*MAXPOWER ) {
                                if ( !CheckWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2],&newval3) ) {
                                    AddWild(BHEAD m[2]-2*MAXPOWER,SYMTONUM,p[2]);
                                    AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                    break;
                                }
                            }
                            else if ( m[2] <= -2*MAXPOWER ) {
                                if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2],&newval3) ) {
                                    AddWild(BHEAD -m[2]-2*MAXPOWER,SYMTONUM,-p[2]);
                                    AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                    break;
                                }
                            }
                        }
                        p += 3; nq -= 3;
                    }
                    if ( nq <= 0 ) return(0);
                    q = p+3;
                    nq -= 3;
                    n -= 3;
                    while ( --nq >= 0 ) *p++ = *q++;
                    m += 3;
                }
                else {
                    if ( t >= xstop || *m < *t || ( *m == *t && m[1] < t[1] ) ) {
                        if ( m[2] > 2*MAXPOWER ) nt = m[2];
                        else if ( m[2] <= -2*MAXPOWER ) nt = -m[2];
                        else return(0);
                        nt -= 2*MAXPOWER;
                        if ( CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) return(0);
                        AddWild(BHEAD nt,SYMTONUM,0);
                        m += 3;
                    }
                    else {
                        *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
                    }
                }
            } while ( m < ystop );
            t = xstop;
        }
/*
            #] DOTPRODUCTS :
*/
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("Error in pattern");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
EndLoop:;
    } while ( m < mstop ); }
    if ( setp ) {
/*
        while ( t < tstop && *t > SYMBOL ) t += t[1];
        if ( t < tstop && setp[1] > 2 ) return(0);
*/
                        /* There were nonempty sets */
                        /* Empty sets are rejected by the compiler */
    }
    return(1);
}

/*
        #] FindOnly :
        #[ FindOnce :           WORD FindOnce(term,pattern)

    Searches for a single match in term. The difference with multi
```

```
    lies mainly in the fact that here functions may occur.
    The functions have not been implemented yet. (10-Apr-1988)
    Wildcard powers are adjustable. The value closer to zero is taken.
    Positive and negative gives (o surprise) zero.

*/

WORD FindOnce(PHEAD WORD *term, WORD *pattern)
{
    GETBIDENTITY
    WORD *t, *m;
    WORD *tstop, *mstop;
    WORD *xstop, *ystop;
    WORD n, nt, *p, nq, mt, ch;
    WORD older[2*NORMSIZE], *q, newval1, newval2, newval3;
    AN.UsedOtherFind = 0;
    m = pattern;
    mstop = m + *m;
    m++;
    t = term;
    t += *term - 1;
    tstop = t - ABS(*t) + 1;
    t = term;
    t++;
    while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
    while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
    if ( m < mstop ) { do {
/*
            #[ SYMBOLS :
*/
        if ( *m == SYMBOL ) {
            ystop = m + m[1];
            m += 2;
            n = 0;
            p = older;
            if ( t < tstop ) while ( *t != SYMBOL ) {
                t += t[1];
                if ( t >= tstop ) {
TryZero:
                    do {
                        if ( *m >= 2*MAXPOWER ) return(0);
                        if ( m[1] >= 2*MAXPOWER ) nt = m[1];
                        else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
                        else return(0);
                        nt -= 2*MAXPOWER;
                        if ( ( ch = CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) != 0 ) {
                            if ( ch > 1 ) return(0);
                            if ( AN.oldtype != SYMTONUM ) return(0);
                            if ( *AN.MaskPointer == 2 ) return(0);
                        }
                        AddWild(BHEAD nt,SYMTONUM,0);
                        m += 2;
                    } while ( m < ystop );
                    goto EndLoop;
                }
            }
            else goto TryZero;
            xstop = t + t[1];
            t += 2;
            do {
                if ( *m == *t && t < xstop ) {
                    nt = t[1];
                    mt = m[1];
                    if ( ( mt > 0 && mt <= nt ) ||
                         ( mt < 0 && mt >= nt ) ) { m += 2; t += 2; }
                    else if ( mt >= 2*MAXPOWER ) goto OnceL2;
                    else if ( mt <= -2*MAXPOWER ) {
                        nt = -nt;
                        mt = -mt;
OnceL2:                 mt -= 2*MAXPOWER;
                        if ( ( ch = CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) != 0 ) {
                            if ( ch > 1 ) return(0);
                            if ( AN.oldtype != SYMTONUM ) return(0);
                            if ( AN.oldvalue <= 0 ) {
                                if ( nt < AN.oldvalue ) nt = AN.oldvalue;
                                else {
                                    if ( *AN.MaskPointer == 2 ) return(0);
                                    if ( nt > 0 ) nt = 0;
                                }
                            }
                            if ( AN.oldvalue >= 0 ) {
                                if ( nt > AN.oldvalue ) nt = AN.oldvalue;
                                else {
                                    if ( *AN.MaskPointer == 2 ) return(0);
                                    if ( nt < 0 ) nt = 0;
                                }
                            }
```

```
                        }
                        AddWild(BHEAD mt,SYMTONUM,nt);
                        m += 2;
                        t += 2;
                    }
                    else {
                        *p++ = *t++; *p++ = *t++; n += 2;
                    }
                }
                else if ( *m >= 2*MAXPOWER ) {
                    while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
                    nq = n;
                    p = older;
                    while ( nq > 0 ) {
                        nt = p[1];
                        if ( !CheckWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,*p,&newval1) ) {
                            mt = m[1];
                            if ( ( mt > 0 && mt <= nt ) ||
                                 ( mt < 0 && mt >= nt ) ) {
                                AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
                                break;
                            }
                            else if ( mt >= 2*MAXPOWER && mt != *m ) {
OnceL4a:                        mt -= 2*MAXPOWER;
                                if ( ( ch = CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) != 0 ) {
                                    if ( ch > 1 ) return(0);
                                    if ( AN.oldtype == SYMTONUM ) {
                                        if ( AN.oldvalue >= 0 ) {
                                            if ( nt > AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt < 0 ) nt = 0;
                                            }
                                        }
                                        else {
                                            if ( nt < AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt > 0 ) nt = 0;
                                            }
                                        }
                                        AddWild(BHEAD mt,SYMTONUM,nt);
                                        AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
                                        break;
                                    }
                                }
                                else {
                                    AddWild(BHEAD mt,SYMTONUM,nt);
                                    AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
                                    break;
                                }
                            }
                            else if ( mt <= -2*MAXPOWER && mt != -(*m) ) {
                                nt = -nt;
                                mt = -mt;
                                goto OnceL4a;
                            }
                        }
                        nq -= 2;
                        p += 2;
                    }
                    if ( nq <= 0 ) return(0);
                    nq -= 2;
                    n -= 2;
                    q = p + 2;
                    while ( --nq >= 0 ) *p++ = *q++;
                    m += 2;
                }
                else {
                    if ( t >= xstop || *m < *t ) {
                        if ( m[1] >= 2*MAXPOWER ) nt = m[1];
                        else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
                        else return(0);
                        nt -= 2*MAXPOWER;
                        if ( ( ch = CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) != 0 ) {
                            if ( ch > 1 ) return(0);
                            if ( AN.oldtype != SYMTONUM ) return(0);
                            if ( *AN.MaskPointer == 2 ) return(0);
                        }
                        AddWild(BHEAD nt,SYMTONUM,0);
                        m += 2;
                    }
                    else {
                        *p++ = *t++; *p++ = *t++; n += 2;
                    }
                }
            } while ( m < ystop );
```

```
        }
/*
            #] SYMBOLS :
            #[ DOTPRODUCTS :
*/
        else if ( *m == DOTPRODUCT ) {
            ystop = m + m[1];
            m += 2;
            n = 0;
            p = older;
            if ( t < tstop ) {
                if ( *t < DOTPRODUCT ) goto OnceOp;
                while ( *t > DOTPRODUCT ) {
                t += t[1];
                if ( t >= tstop || *t < DOTPRODUCT ) {
OnceOp:
                    do {
                        if ( *m >= (AM.OffsetVector+WILDOFFSET)
                        || m[1] >= (AM.OffsetVector+WILDOFFSET) ) return(0);
                        if ( m[2] >= 2*MAXPOWER ) {
                            nq = m[2] - 2*MAXPOWER;
                        }
                        else if ( m[2] <= -2*MAXPOWER ) {
                            nq = -m[2] - 2*MAXPOWER;
                        }
                        else return(0);
                        if ( CheckWild(BHEAD nq,SYMTONUM,(WORD)0,&newval3) ) {
                            if ( AN.oldtype != SYMTONUM ) return(0);
                            if ( *AN.MaskPointer == 2 ) return(0);
                        }
                        AddWild(BHEAD nq,SYMTONUM,(WORD)0);
                        m += 3;
                    } while ( m < ystop );
                    goto EndLoop;
                }
                }
            }
            else goto OnceOp;
            xstop = t + t[1];
            t += 2;
            do {
                if ( *m == *t && m[1] == t[1] && t < xstop ) {
                    nt = t[2];
                    mt = m[2];
/*
                    if ( ( nt > 0 && nt < mt ) ||
                         ( nt < 0 && nt > mt ) ) {
                        if ( mt <= -2*MAXPOWER ) {
                            mt = -mt;
                            nt = -nt;
                        }
                        else if ( mt < 2*MAXPOWER ) return(0);
                        mt -= 2*MAXPOWER;
                        if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
                            if ( AN.oldtype != SYMTONUM ) return(0);
                            if ( AN.oldvalue <= 0 ) {
                                if ( nt < AN.oldvalue ) nt = AN.oldvalue;
                                else {
                                    if ( *AN.MaskPointer == 2 ) return(0);
                                    if ( nt > 0 ) nt = 0;
                                }
                            }
                            if ( AN.oldvalue >= 0 ) {
                                if ( nt > AN.oldvalue ) nt = AN.oldvalue;
                                else {
                                    if ( *AN.MaskPointer == 2 ) return(0);
                                    if ( nt < 0 ) nt = 0;
                                }
                            }
                        }
                        AddWild(BHEAD mt,SYMTONUM,nt);
                        m += 3; t += 3;
                    }
                    else if ( ( nt > 0 && nt >= mt && mt > -2*MAXPOWER )
                    || ( nt < 0 && nt <= mt && mt < 2*MAXPOWER ) ) {
                        m += 3; t += 3;
                    }
*/
                    if ( ( mt > 0 && mt <= nt ) ||
                         ( mt < 0 && mt >= nt ) ) { m += 3; t += 3; }
                    else if ( mt >= 2*MAXPOWER ) goto OnceL7;
                    else if ( mt <= -2*MAXPOWER ) {
                        nt = -nt;
                        mt = -mt;
OnceL7:                 mt -= 2*MAXPOWER;
                        if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
                            if ( AN.oldtype != SYMTONUM ) return(0);
```

```
                            if ( AN.oldvalue <= 0 ) {
                                if ( nt < AN.oldvalue ) nt = AN.oldvalue;
                                else {
                                    if ( *AN.MaskPointer == 2 ) return(0);
                                    if ( nt > 0 ) nt = 0;
                                }
                            }
                            if ( AN.oldvalue >= 0 ) {
                                if ( nt > AN.oldvalue ) nt = AN.oldvalue;
                                else {
                                    if ( *AN.MaskPointer == 2 ) return(0);
                                    if ( nt < 0 ) nt = 0;
                                }
                            }
                        }
                        AddWild(BHEAD mt,SYMTONUM,nt);
                        m += 3;
                        t += 3;
                    }
                    else {
                        *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
                    }
                }
                else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
                    while ( t < xstop ) {
                        *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
                    }
                    nq = n;
                    p = older;
                    while ( nq > 0 ) {
                        if ( *m == m[1] ) {
                            if ( *p != p[1] ) goto NextInDot;
                        }
                        if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*p,&newval1) &&
                             !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,p[1],&newval2) ) {
                            nt = p[2];
                            mt = m[2];
                            if ( ( mt > 0 && nt >= mt ) ||
                                 ( mt < 0 && nt <= mt ) ) {
OnceL9:                         AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
                                AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval2);
                                break;
                            }
                            if ( mt >= 2*MAXPOWER ) {
OnceL9a:                        mt -= 2*MAXPOWER;
                                if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
                                    if ( AN.oldtype == SYMTONUM ) {
                                        if ( AN.oldvalue >= 0 ) {
                                            if ( nt > AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt < 0 ) nt = 0;
                                            }
                                        }
                                        else {
                                            if ( nt < AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt > 0 ) nt = 0;
                                            }
                                        }
                                        AddWild(BHEAD mt,SYMTONUM,nt);
                                        goto OnceL9;
                                    }
                                }
                                else {
                                    AddWild(BHEAD mt,SYMTONUM,nt);
                                    goto OnceL9;
                                }
                            }
                            else if ( mt <= -2*MAXPOWER ) {
                                mt = -mt;
                                nt = -nt;
                                goto OnceL9a;
                            }
                        }
                        if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,p[1],&newval1) &&
                             !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*p,&newval2) ) {
                            nt = p[2];
                            mt = m[2];
                            if ( ( mt > 0 && nt >= mt ) ||
                                 ( mt < 0 && nt <= mt ) ) {
OnceL10:                        AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
                                AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval2);
                                break;
                            }
                            if ( mt >= 2*MAXPOWER ) {
```

```
OnceL10a:                       mt -= 2*MAXPOWER;
                                if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
                                    if ( AN.oldtype == SYMTONUM ) {
                                        if ( AN.oldvalue >= 0 ) {
                                            if ( nt > AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt < 0 ) nt = 0;
                                            }
                                        }
                                        else {
                                            if ( nt < AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt > 0 ) nt = 0;
                                            }
                                        }
                                        AddWild(BHEAD mt,SYMTONUM,nt);
                                        goto OnceL10;
                                    }
                                }
                                else {
                                    AddWild(BHEAD mt,SYMTONUM,nt);
                                    goto OnceL10;
                                }
                            }
                            else if ( mt <= -2*MAXPOWER ) {
                                mt = -mt;
                                nt = -nt;
                                goto OnceL10a;
                            }
                        }
NextInDot:
                        p += 3; nq -= 3;
                    }
                    if ( nq <= 0 ) return(0);
                    else {
                        q = p+3;
                        nq -= 3;
                        n -= 3;
                        while ( --nq >= 0 ) *p++ = *q++;
                    }
                    m += 3;
                }
                else if ( m[1] >= (AM.OffsetVector+WILDOFFSET) ) {
                    while ( *m >= *t && t < xstop ) {
                        *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
                    }
                    nq = n;
                    p = older;
                    while ( nq > 0 ) {
                        if ( *m == *p && !CheckWild(BHEAD m[1]-WILDOFFSET,
                        VECTOVEC,p[1],&newval1) ) {
                            nt = p[2];
                            mt = m[2];
                            if ( ( mt > 0 && nt >= mt ) ||
                                 ( mt < 0 && nt <= mt ) ) {
                                AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                break;
                            }
                            else if ( mt >= 2*MAXPOWER ) {
OnceL7a:                        mt -= 2*MAXPOWER;
                                if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
                                    if ( AN.oldtype == SYMTONUM ) {
                                        if ( AN.oldvalue >= 0 ) {
                                            if ( nt > AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt < 0 ) nt = 0;
                                            }
                                        }
                                        else {
                                            if ( nt < AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt > 0 ) nt = 0;
                                            }
                                        }
                                        AddWild(BHEAD mt,SYMTONUM,nt);
                                        AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                        break;
                                    }
                                }
                                else {
                                    AddWild(BHEAD mt,SYMTONUM,nt);
                                    AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                    break;
```

```
                                }
                            }
                            else if ( mt <= -2*MAXPOWER ) {
                                mt = -mt;
                                nt = -nt;
                                goto OnceL7a;
                            }
                        }
                        if ( *m == p[1] && !CheckWild(BHEAD m[1]-WILDOFFSET,
                        VECTOVEC,*p,&newval1) ) {
                            nt = p[2];
                            mt = m[2];
                            if ( ( mt > 0 && nt >= mt ) ||
                                 ( mt < 0 && nt <= mt ) ) {
                                AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                break;
                            }
                            if ( mt >= 2*MAXPOWER ) {
OnceL8a:                        mt -= 2*MAXPOWER;
                                if ( CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) {
                                    if ( AN.oldtype == SYMTONUM ) {
                                        if ( AN.oldvalue >= 0 ) {
                                            if ( nt > AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt < 0 ) nt = 0;
                                            }
                                        }
                                        else {
                                            if ( nt < AN.oldvalue ) nt = AN.oldvalue;
                                            else {
                                                if ( *AN.MaskPointer == 2 ) return(0);
                                                if ( nt > 0 ) nt = 0;
                                            }
                                        }
                                        AddWild(BHEAD mt,SYMTONUM,nt);
                                        AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                        break;
                                    }
                                }
                                else {
                                    AddWild(BHEAD mt,SYMTONUM,nt);
                                    AddWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,newval1);
                                    break;
                                }
                            }
                            else if ( mt < -2*MAXPOWER ) {
                                mt = -mt;
                                nt = -nt;
                                goto OnceL8a;
                            }
                        }
                        p += 3; nq -= 3;
                    }
                    if ( nq <= 0 ) return(0);
                    q = p+3;
                    nq -= 3;
                    n -= 3;
                    while ( --nq >= 0 ) *p++ = *q++;
                    m += 3;
                }
                else {
                    if ( t >= xstop || *m < *t || ( *m == *t && m[1] < t[1] ) ) {
                        if ( m[2] >= 2*MAXPOWER ) nt = m[2];
                        else if ( m[2] <= -2*MAXPOWER ) nt = -m[2];
                        else return(0);
                        nt -= 2*MAXPOWER;
                        if ( CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) {
                            if ( AN.oldtype != SYMTONUM ) return(0);
                            if ( *AN.MaskPointer == 2 ) return(0);
                        }
                        AddWild(BHEAD nt,SYMTONUM,0);
                        m += 3;
                    }
                    else {
                        *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
                    }
                }
            } while ( m < ystop );
            t = xstop;
        }
/*
            #] DOTPRODUCTS :
*/
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("Error in pattern");
```

```
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
EndLoop:;
    } while ( m < mstop ); }
    else {
        return(-1);
    }
    return(1);
}

/*
        #] FindOnce :
        #[ FindMulti :          WORD FindMulti(term,pattern)

    Note that multi cannot deal with wildcards. Those patterns revert
    to many which gives subsequent calls to once.

*/

WORD FindMulti(PHEAD WORD *term, WORD *pattern)
{
    GETBIDENTITY
    WORD *t, *m, *p;
    WORD *tstop, *mstop;
    WORD *xstop, *ystop;
    WORD mt, power, n, nq;
    WORD older[2*NORMSIZE], *q, newval1;
    AN.UsedOtherFind = 0;
    m = pattern;
    mstop = m + *m;
    m++;
    t = term;
    t += *term - 1;
    tstop = t - ABS(*t) + 1;
    t = term;
    t++;
    while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
    while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
    power = -1;             /* No power yet */
    if ( m < mstop ) { do {
/*
            #[ SYMBOLS :
*/
        if ( *m == SYMBOL ) {
            ystop = m + m[1];
            m += 2;
            if ( t >= tstop ) return(0);
            while ( *t != SYMBOL ) { t += t[1]; if ( t >= tstop ) return(0); }
            xstop = t + t[1];
            t += 2;
            p = older;
            n = 0;
            do {
                if ( *m >= 2*MAXPOWER ) {
                    while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
                    nq = n;
                    p = older;
                    while ( nq > 0 ) {
                        if ( !CheckWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,*p,&newval1) ) {
                            mt = p[1]/m[1];
                            if ( mt > 0 ) {
                                if ( power < 0 || mt < power ) power = mt;
                                AddWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,newval1);
                                break;
                            }
                        }
                        nq -= 2;
                        p += 2;
                    }
                    if ( nq <= 0 ) return(0);
                    nq -= 2;
                    n -= 2;
                    q = p + 2;
                    while ( --nq >= 0 ) *p++ = *q++;
                    m += 2;
                }
                else if ( t >= xstop ) return(0);
                else if ( *m == *t ) {
                    if ( ( mt = t[1]/m[1] ) <= 0 ) return(0);
                    if ( power < 0 || mt < power ) power = mt;
                    m += 2;
                    t += 2;
                }
                else if ( *m < *t ) return(0);
                else { *p++ = *t++; *p++ = *t++; n += 2; }
            } while ( m < ystop );
```

```
        }
/*
            #] SYMBOLS :
            #[ DOTPRODUCTS :
*/
        else if ( *m == DOTPRODUCT ) {
            ystop = m + m[1];
            m += 2;
            if ( t >= tstop ) return(0);
            while ( *t != DOTPRODUCT ) { t += t[1]; if ( t >= tstop ) return(0); }
            xstop = t + t[1];
            t += 2;
            do {
                if ( t >= xstop ) return(0);
                if ( *t == *m ) {
                    if ( t[1] == m[1] ) {
                        if ( ( mt = t[2]/m[2] ) <= 0 ) return(0);
                        if ( power < 0 || mt < power ) power = mt;
                        m += 3;
                    }
                    else if ( t[1] > m[1] ) return(0);
                }
                else if ( *t > *m ) return(0);
                t += 3;
            } while ( m < ystop );
            t = xstop;
        }
/*
            #] DOTPRODUCTS :
*/
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("Error in pattern");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
    } while ( m < mstop ); }
    if ( power < 0 ) power = 0;
    return(power);
}

/*
        #] FindMulti :
        #[ FindRest :           WORD FindRest(term,pattern)

    This routine scans for anything but dotproducts and symbols.

*/

WORD FindRest(PHEAD WORD *term, WORD *pattern)
{
    GETBIDENTITY
    WORD *t, *m, *tt, wild, regular;
    WORD *tstop, *mstop;
    WORD *xstop, *ystop;
    WORD n, *p, nq;
    WORD older[NORMSIZE], *q, newval1, newval2;
    int i, ntwa;
    AN.UsedOtherFind = 0;
    AN.findTerm = term; AN.findPattern = pattern;
    m = AN.WildValue;
    i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
    ntwa = 0;
    while ( i > 0 ) {
        if ( m[0] == ARGTOARG ) ntwa++;
        m += m[1];
        i--;
    }
    t = term;
    t += *term - 1;
    tstop = t - ABS(*t) + 1;
    t = term;
    t++; p = t;
    while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
    tstop = t;
    t = p;
    m = pattern;
    mstop = m + *m;
    m++;
    p = m;
    while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
    mstop = m;
    m = p;
    if ( m < mstop ) {
    do {
/*
            #[ FUNCTIONS :
```

```
*/
        if ( *m >= FUNCTION ) {
            if ( *mstop > 5 && !MatchIsPossible(pattern,term) ) return(0);
            ystop = m;
            n = 0;
            do {
                m += m[1]; n++;
            } while ( m < mstop && *m >= FUNCTION );
            AT.WorkPointer += n;
            while ( t < tstop && *t == SUBEXPRESSION ) t += t[1];
            tt = xstop = t;
            nq = 0;
            while ( t < tstop && ( *t >= FUNCTION || *t == SUBEXPRESSION ) ) {
                if ( *t != SUBEXPRESSION ) {
                    nq++;
                    if ( functions[*t-FUNCTION].commute ) tt = t + t[1];
                }
                t += t[1];
            }
            if ( nq < n ) return(0);
            AN.terstart = term;
            AN.terstop = t;
            AN.terfirstcomm = tt;
            AN.patstop = m;
            AN.NumTotWildArgs = ntwa;
            if ( !ScanFunctions(BHEAD ystop,xstop,0) ) return(0);
        }
/*
            #] FUNCTIONS :
            #[ VECTORS :
*/
        else if ( *m == VECTOR ) {
            while ( t < tstop && *t != VECTOR ) t += t[1];
            if ( t >= tstop ) return(0);
            xstop = t + t[1];
            ystop = m + m[1];
            t += 2;
            m += 2;
            n = 0;
            p = older;
            do {
                if ( *m == *t && m[1] == t[1] && t < xstop ) {
                    m += 2; t += 2;
                }
                else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
                    if ( t < xstop ) {
                        p = older + n;
                        do { *p++ = *t++; n++; } while ( t < xstop );
                    }
                    p = older;
                    nq = n;
                    if ( ( m[1] < (AM.OffsetIndex+WILDOFFSET) )
                    || ( m[1] >= (AM.OffsetIndex+2*WILDOFFSET) ) ) {
                        while ( nq > 0 ) {
                            if ( m[1] == p[1] ) {
                                if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*p,&newval1) ) {
RestL11:                            AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
                                    break;
                                }
                            }
                            p += 2;
                            nq -= 2;
                        }
                    }
                    else {      /* Double wildcard */
                        while ( nq > 0 ) {
                            if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*p,&newval1) &&
                                 !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,p[1],&newval2) ) {
                                AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval2);
                                goto RestL11;
                            }
                            p += 2;
                            nq -= 2;
                        }
                    }
                    if ( nq > 0 ) {
                        nq -= 2; q = p + 2; n -= 2;
                        while ( --nq >= 0 ) *p++ = *q++;
                    }
                    else return(0);
                    m += 2;
                }
                else if ( ( *m <= *t )
                && ( m[1] >= (AM.OffsetIndex + WILDOFFSET) )
                && ( m[1] <  (AM.OffsetIndex + 2*WILDOFFSET) ) ) {
                    if ( *m == *t && t < xstop ) {
                        p = older;
```

```
                        p += n;
                        *p++ = *t++;
                        *p++ = *t++;
                        n += 2;
                    }
                    p = older;
                    nq = n;
                    while ( nq > 0 ) {
                        if ( *m == *p ) {
                            if ( !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,p[1],&newval1) ) {
                                AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval1);
                                break;
                            }
                        }
                        p += 2;
                        nq -= 2;
                    }
                    if ( nq > 0 ) {
                        nq -= 2; q = p + 2; n -= 2;
                        while ( --nq >= 0 ) *p++ = *q++;
                    }
                    else return(0);
                    m += 2;
                }
                else {
                    if ( t >= xstop ) return(0);
                    *p++ = *t++; *p++ = *t++; n += 2;
                }
            } while ( m < ystop );
        }
/*
            #] VECTORS :
            #[ INDICES :
*/
        else if ( *m == INDEX ) {
/*
            This needs only to say that there is a match, after matching
            a 'wildcard'. This has to be prepared in TestMatch. The C->rhs
            should provide the replacement inside the prototype!
            Next question: id,p=q/2+r/2
*/
            while ( *t != INDEX ) { t += t[1]; if ( t >= tstop ) return(0); }
            xstop = t + t[1];
            ystop = m + m[1];
            t += 2;
            m += 2;
            n = 0;
            p = older;
            do {
                if ( *m == *t && t < xstop && m < ystop ) {
                    t++; m++;
                }
                else if ( ( *m >= (AM.OffsetIndex+WILDOFFSET) )
                && ( *m < (AM.OffsetIndex+2*WILDOFFSET) ) ) {
                    while ( t < xstop ) {
                        *p++ = *t++; n++;
                    }
                    if ( !n ) return(0);
                    nq = n;
                    q = older;
                    do {
                        if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*q,&newval1) ) {
                            AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newval1);
                            break;
                        }
                        q++;
                        nq--;
                    } while ( nq > 0 );
                    if ( nq <= 0 ) return (0);
                    n--;
                    nq--;
                    p = q + 1;
                    while ( nq > 0 ) { *q++ = *p++; nq--; }
                    p--;
                    m++;
                }
                else if ( ( *m >= (AM.OffsetVector+WILDOFFSET) )
                && ( *m < (AM.OffsetVector+2*WILDOFFSET) ) ) {
                    while ( t < xstop ) {
                        *p++ = *t++; n++;
                    }
                    if ( !n ) return(0);
                    nq = n;
                    q = older;
                    do {
                        if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*q,&newval1) ) {
                            AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
```

```
                            break;
                        }
                        q++;
                        nq--;
                    } while ( nq > 0 );
                    if ( nq <= 0 ) return (0);
                    n--;
                    nq--;
                    p = q + 1;
                    while ( nq > 0 ) { *q++ = *p++; nq--; }
                    p--;
                    m++;
                }
                else {
                    if ( t >= xstop ) return(0);
                    *p++ = *t++; n++;
                }
            } while ( m < ystop );

/*
            return(0);
*/
        }
/*
            #] INDICES :
            #[ DELTAS :
*/
        else if ( *m == DELTA ) {
            while ( *t != DELTA ) { t += t[1]; if ( t >= tstop ) return(0); }
            xstop = t + t[1];
            ystop = m + m[1];
            t += 2;
            m += 2;
            n = 0;
            p = older;
            do {
                if ( *t == *m && t[1] == m[1] && t < xstop ) {
                    m += 2;
                    t += 2;
                }
                else if ( ( *m >= (AM.OffsetIndex+WILDOFFSET) )
                && ( *m < (AM.OffsetIndex+2*WILDOFFSET) )
                && ( m[1] >= (AM.OffsetIndex+WILDOFFSET) )
                && ( m[1] < (AM.OffsetIndex+2*WILDOFFSET) ) ) { /* Two dummies */
                    while ( t < xstop ) {
                        *p++ = *t++; *p++ = *t++; n += 2;
                    }
                    if ( !n ) return(0);
                    nq = n;
                    q = older;
                    do {
                        if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*q,&newval1) &&
                             !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,q[1],&newval2) ) {
                            AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newval1);
                            AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval2);
                            break;
                        }
                        if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,q[1],&newval1) &&
                             !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,*q,&newval2) ) {
                            AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newval1);
                            AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval2);
                            break;
                        }
                        q += 2;
                        nq -= 2;
                    } while ( nq > 0 );
                    if ( nq <= 0 ) return(0);
                    n -= 2;
                    nq -= 2;
                    p = q + 2;
                    while ( nq > 0 ) { *q++ = *p++; nq--; }
                    p -= 2;
                    m += 2;
                }
                else if ( ( m[1] >= (AM.OffsetIndex+WILDOFFSET) )
                && ( m[1] < (AM.OffsetIndex+2*WILDOFFSET) ) ) {
                    wild = m[1]; regular = *m;
OneWild:
                    while ( ( regular == *t || regular == t[1] ) && t < xstop ) {
                        *p++ = *t++; *p++ = *t++; n += 2;
                    }
                    if ( !n ) return(0);
                    nq = n;
                    q = older;
                    do {
                        if ( regular == *q && !CheckWild(BHEAD wild-WILDOFFSET,INDTOIND,q[1],&newval1)
    ) {
```

```
                            AddWild(BHEAD wild-WILDOFFSET,INDTOIND,newval1);
                            break;
                        }
                        if ( regular == q[1] && !CheckWild(BHEAD wild-WILDOFFSET,INDTOIND,*q,&newval1) ) {
                            AddWild(BHEAD wild-WILDOFFSET,INDTOIND,newval1);
                            break;
                        }
                        q += 2;
                        nq -= 2;
                    } while ( nq > 0 );
                    if ( nq <= 0 ) return(0);
                    n -= 2;
                    nq -= 2;
                    p = q + 2;
                    while ( nq > 0 ) { *q++ = *p++; nq--; }
                    p -= 2;
                    m += 2;
                }
                else if ( ( *m >= (AM.OffsetIndex+WILDOFFSET) )
                && ( *m < (AM.OffsetIndex+2*WILDOFFSET) ) ) {
                    wild = *m; regular = m[1];
                    goto OneWild;
                }
                else {
                    if ( t >= tstop || *m < *t || ( *m == *t && m[1] < t[1] ) )
                        return(0);
                    *p++ = *t++; *p++ = *t++; n += 2;
                }
            } while ( m < ystop );
        }
/*
            #] DELTAS :
*/
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("Pattern not yet implemented");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
    } while ( m < mstop );
    return(1);
    }
    else return(-1);
}

/*
        #] FindRest :
    #] Patterns :
*/
```

4.33 form3.h File Reference

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "ftypes.h"
#include "fsizes.h"
#include "minos.h"
#include "structs.h"
#include "declare.h"
#include "variable.h"
```

Include dependency graph for form3.h:

This graph shows which files directly or indirectly include this file:

Macros

- #define MAJORVERSION 4
- #define MINORVERSION 2
- #define PRODUCTIONDATE "06-jul-2017"
- #define inline
- #define NDEBUG
- #define STATIC_ASSERT(condition) STATIC_ASSERT__1(condition,__LINE__)
- #define STATIC_ASSERT__1(X, L) STATIC_ASSERT__2(X,L)
- #define STATIC_ASSERT__2(X, L) STATIC_ASSERT__3(X,L)
- #define STATIC_ASSERT__3(X, L) typedef char static_assertion_failed_##L[(!!(X))∗2-1]
- #define TOPBITONLY ((ULONG)1 << (BITSINWORD - 1)) /∗ 0x00008000UL ∗/
- #define TOPLONGBITONLY ((ULONG)1 << (BITSINLONG - 1)) /∗ 0x80000000UL ∗/
- #define SPECMASK ((UWORD)1 << (BITSINWORD - 1)) /∗ 0x8000U ∗/
- #define WILDMASK ((UWORD)1 << (BITSINWORD - 2)) /∗ 0x4000U ∗/
- #define WORDMASK ((ULONG)FULLMAX - 1) /∗ 0x0000FFFFUL ∗/
- #define AWORDMASK (WORDMASK << BITSINWORD) /∗ 0xFFFF0000UL ∗/
- #define FULLMAX ((LONG)1 << BITSINWORD) /∗ 0x00010000L ∗/
- #define MAXPOSITIVE ((LONG)(TOPBITONLY - 1)) /∗ 0x00007FFFL ∗/
- #define MAXLONG ((LONG)(TOPLONGBITONLY - 1)) /∗ 0x7FFFFFFFL ∗/
- #define MAXPOSITIVE2 (MAXPOSITIVE / 2) /∗ 0x00003FFFL ∗/
- #define MAXPOSITIVE4 (MAXPOSITIVE / 4) /∗ 0x00001FFFL ∗/
- #define alignof(type) offsetof(struct { char c_; type x_; }, x_)
- #define PADDUMMY(type, size) UBYTE d_u_m_m_y[alignof(type) - ((size) & (alignof(type) - 1))]
- #define PADPOSITION(ptr_, long_, int_, word_, byte_)
- #define PADPOINTER(long_, int_, word_, byte_)
- #define PADLONG(int_, word_, byte_)
- #define PADINT(word_, byte_)
- #define PADWORD(byte_)
- #define WITHSORTBOTS
- #define FILES FILE
- #define Uopen(x, y) fopen(x,y)
- #define Uflush(x) fflush(x)
- #define Uclose(x) fclose(x)
- #define Uread(x, y, z, u) fread(x,y,z,u)
- #define Uwrite(x, y, z, u) fwrite(x,y,z,u)

- #define Usetbuf(x, y) setbuf(x,y)
- #define Useek(x, y, z) fseek(x,y,z)
- #define Utell(x) ftell(x)
- #define Ugetpos(x, y) fgetpos(x,y)
- #define Usetpos(x, y) fsetpos(x,y)
- #define Usync(x) fflush(x)
- #define Utruncate(x) _chsize(_fileno(x),0)
- #define Ustdout stdout
- #define MAX_OPEN_FILES FOPEN_MAX
- #define bzero(b, len) (memset((b), 0, (len)), (void)0)
- #define GetPID() ((LONG)GetCurrentProcessId())

Typedefs

- typedef void VOID
- typedef signed char SBYTE
- typedef unsigned char UBYTE
- typedef unsigned int UINT
- typedef ULONG RLONG
- typedef INT64 MLONG

Functions

- **STATIC_ASSERT** (sizeof(WORD) ∗8==BITSINWORD)
- **STATIC_ASSERT** (sizeof(LONG) ∗8==BITSINLONG)
- **STATIC_ASSERT** (sizeof(LONG) >=sizeof(int ∗))
- **STATIC_ASSERT** (sizeof(INT16)==2)
- **STATIC_ASSERT** (sizeof(INT32)==4)
- **STATIC_ASSERT** (sizeof(INT64)==8)

4.33.1 Detailed Description

Contains critical defines for the compilation process Also contains the inclusion of all necessary header files. There are also some system dependencies concerning file functions.

Definition in file form3.h.

4.33.2 Macro Definition Documentation

4.33.2.1 MAJORVERSION

```
#define MAJORVERSION 4
```

Definition at line 47 of file form3.h.

4.33.2.2 MINORVERSION

```
#define MINORVERSION 2
```

Definition at line 48 of file form3.h.

4.33.2.3 PRODUCTIONDATE

```
#define PRODUCTIONDATE "06-jul-2017"
```

Definition at line 53 of file form3.h.

4.33.2.4 inline

```
#define inline
```

Definition at line 129 of file form3.h.

4.33.2.5 NDEBUG

```
#define NDEBUG
```

Definition at line 165 of file form3.h.

4.33.2.6 STATIC_ASSERT

```
#define STATIC_ASSERT(
            condition ) STATIC_ASSERT__1(condition,__LINE__)
```

Definition at line 173 of file form3.h.

4.33.2.7 STATIC_ASSERT__1

```
#define STATIC_ASSERT__1(
            X,
            L ) STATIC_ASSERT__2(X,L)
```

Definition at line 174 of file form3.h.

4.33.2.8 STATIC_ASSERT__2

```
#define STATIC_ASSERT__2(
            X,
            L ) STATIC_ASSERT__3(X,L)
```

Definition at line 175 of file form3.h.

4.33.2.9 STATIC_ASSERT__3

```
#define STATIC_ASSERT__3(
            X,
            L )  typedef char static_assertion_failed_##L[(!!(X))*2-1]
```

Definition at line 176 of file form3.h.

4.33.2.10 TOPBITONLY

```
#define TOPBITONLY ((ULONG)1 << (BITSINWORD - 1)) /* 0x00008000UL */
```

Definition at line 291 of file form3.h.

4.33.2.11 TOPLONGBITONLY

```
#define TOPLONGBITONLY ((ULONG)1 << (BITSINLONG - 1)) /* 0x80000000UL */
```

Definition at line 292 of file form3.h.

4.33.2.12 SPECMASK

```
#define SPECMASK ((UWORD)1 << (BITSINWORD - 1)) /* 0x8000U */
```

Definition at line 293 of file form3.h.

4.33.2.13 WILDMASK

```
#define WILDMASK ((UWORD)1 << (BITSINWORD - 2)) /* 0x4000U */
```

Definition at line 294 of file form3.h.

4.33.2.14 WORDMASK

```
#define WORDMASK ((ULONG)FULLMAX - 1) /* 0x0000FFFFUL */
```

Definition at line 295 of file form3.h.

4.33.2.15 AWORDMASK

```
#define AWORDMASK (WORDMASK << BITSINWORD) /* 0xFFFF0000UL */
```

Definition at line 296 of file form3.h.

4.33.2.16 FULLMAX

```
#define FULLMAX ((LONG)1 << BITSINWORD) /* 0x00010000L */
```

Definition at line 297 of file form3.h.

4.33.2.17 MAXPOSITIVE

```
#define MAXPOSITIVE ((LONG)(TOPBITONLY - 1)) /* 0x00007FFFL */
```

Definition at line 298 of file form3.h.

4.33.2.18 MAXLONG

```
#define MAXLONG ((LONG)(TOPLONGBITONLY - 1)) /* 0x7FFFFFFFL */
```

Definition at line 299 of file form3.h.

4.33.2.19 MAXPOSITIVE2

```
#define MAXPOSITIVE2 (MAXPOSITIVE / 2) /* 0x00003FFFL */
```

Definition at line 300 of file form3.h.

4.33.2.20 MAXPOSITIVE4

```
#define MAXPOSITIVE4 (MAXPOSITIVE / 4) /* 0x00001FFFL */
```

Definition at line 301 of file form3.h.

4.33.2.21 alignof

```
#define alignof(
              type ) offsetof(struct { char c_; type x_; }, x_)
```

Definition at line 317 of file form3.h.

4.33.2.22 PADDUMMY

```
#define PADDUMMY(
              type,
              size )  UBYTE d_u_m_m_y[alignof(type) - ((size) & (alignof(type) - 1))]
```

Definition at line 362 of file form3.h.

4.33.2.23 PADPOSITION

```
#define PADPOSITION(
              ptr_,
              long_,
              int_,
              word_,
              byte_ )
```

Value:

```
PADDUMMY(off_t, \
    + sizeof(int *) * (ptr_) \
    + sizeof(LONG)  * (long_) \
    + sizeof(int)   * (int_) \
    + sizeof(WORD)  * (word_) \
    + sizeof(UBYTE) * (byte_) \
)
```

Definition at line 364 of file form3.h.

4.33.2.24 PADPOINTER

```
#define PADPOINTER(
                long_,
                int_,
                word_,
                byte_ )
```

Value:

```
PADDUMMY(int *, \
    + sizeof(LONG)  * (long_) \
    + sizeof(int)   * (int_) \
    + sizeof(WORD)  * (word_) \
    + sizeof(UBYTE) * (byte_) \
)
```

Definition at line 372 of file form3.h.

4.33.2.25 PADLONG

```
#define PADLONG(
                int_,
                word_,
                byte_ )
```

Value:

```
PADDUMMY(LONG, \
    + sizeof(int)   * (int_) \
    + sizeof(WORD)  * (word_) \
    + sizeof(UBYTE) * (byte_) \
)
```

Definition at line 379 of file form3.h.

4.33.2.26 PADINT

```
#define PADINT(
                word_,
                byte_ )
```

Value:

```
PADDUMMY(int, \
    + sizeof(WORD)  * (word_) \
    + sizeof(UBYTE) * (byte_) \
)
```

Definition at line 385 of file form3.h.

4.33.2.27 PADWORD

```
#define PADWORD(
                byte_ )
```

Value:

```
PADDUMMY(WORD, \
    + sizeof(UBYTE) * (byte_) \
)
```

Definition at line 390 of file form3.h.

4.33.2.28 WITHSORTBOTS

```
#define WITHSORTBOTS
```

Definition at line 400 of file form3.h.

4.33.2.29 FILES

```
#define FILES FILE
```

Definition at line 477 of file form3.h.

4.33.2.30 Uopen

```
#define Uopen(
            x,
            y ) fopen(x,y)
```

Definition at line 478 of file form3.h.

4.33.2.31 Uflush

```
#define Uflush(
            x ) fflush(x)
```

Definition at line 479 of file form3.h.

4.33.2.32 Uclose

```
#define Uclose(
            x ) fclose(x)
```

Definition at line 480 of file form3.h.

4.33.2.33 Uread

```
#define Uread(
            x,
            y,
            z,
            u ) fread(x,y,z,u)
```

Definition at line 481 of file form3.h.

4.33.2.34 Uwrite

```
#define Uwrite(
            x,
            y,
            z,
            u ) fwrite(x,y,z,u)
```

Definition at line 482 of file form3.h.

4.33.2.35 Usetbuf

```
#define Usetbuf(
            x,
            y ) setbuf(x,y)
```

Definition at line 483 of file form3.h.

4.33.2.36 Useek

```
#define Useek(
            x,
            y,
            z ) fseek(x,y,z)
```

Definition at line 484 of file form3.h.

4.33.2.37 Utell

```
#define Utell(
            x ) ftell(x)
```

Definition at line 485 of file form3.h.

4.33.2.38 Ugetpos

```
#define Ugetpos(
            x,
            y ) fgetpos(x,y)
```

Definition at line 486 of file form3.h.

4.33.2.39 Usetpos

```
#define Usetpos(
            x,
            y ) fsetpos(x,y)
```

Definition at line 487 of file form3.h.

4.33.2.40 Usync

```
#define Usync(
            x ) fflush(x)
```

Definition at line 488 of file form3.h.

4.33.2.41 Utruncate

```
#define Utruncate(
            x ) _chsize(_fileno(x),0)
```

Definition at line 489 of file form3.h.

4.33.2.42 Ustdout

```
#define Ustdout stdout
```

Definition at line 490 of file form3.h.

4.33.2.43 MAX_OPEN_FILES

```
#define MAX_OPEN_FILES FOPEN_MAX
```

Definition at line 491 of file form3.h.

4.33.2.44 bzero

```
#define bzero(
            b,
            len ) (memset((b), 0, (len)), (void)0)
```

Definition at line 492 of file form3.h.

4.33.2.45 GetPID

```
#define GetPID( ) ((LONG)GetCurrentProcessId())
```

Definition at line 493 of file form3.h.

4.33.3 Typedef Documentation

4.33.3.1 VOID

```
typedef void VOID
```

Definition at line 284 of file form3.h.

4.33.3.2 SBYTE

```
typedef signed char SBYTE
```

Definition at line 285 of file form3.h.

4.33.3.3 UBYTE

```
typedef unsigned char UBYTE
```

Definition at line 286 of file form3.h.

4.33.3.4 UINT

```
typedef unsigned int UINT
```

Definition at line 287 of file form3.h.

4.33.3.5 RLONG

```
typedef ULONG RLONG
```

Definition at line 288 of file form3.h.

4.33.3.6 MLONG

```
typedef INT64 MLONG
```

Definition at line 289 of file form3.h.

4.34 form3.h

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
```

```
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

#ifndef __FORM3H__
#define __FORM3H__

#ifdef HAVE_CONFIG_H

#ifndef CONFIG_H_INCLUDED
#define CONFIG_H_INCLUDED
#include <config.h>
#endif

#else  /* HAVE_CONFIG_H */

#define MAJORVERSION 4
#define MINORVERSION 2

#ifdef __DATE__
#define PRODUCTIONDATE __DATE__
#else
#define PRODUCTIONDATE "06-jul-2017"
#endif

#undef BETAVERSION

#ifdef LINUX32
#define UNIX
#define LINUX
#define ILP32
#define SIZEOF_LONG_LONG 8
#define _FILE_OFFSET_BITS 64
#define WITHZLIB
#define WITHGMP
#define WITHPOSIXCLOCK
#endif

#ifdef LINUX64
#define UNIX
#define LINUX
#define LP64
#define WITHZLIB
#define WITHGMP
#define WITHPOSIXCLOCK
#endif

#ifdef APPLE32
#define UNIX
#define ILP32
#define SIZEOF_LONG_LONG 8
#define _FILE_OFFSET_BITS 64
#define WITHZLIB
#endif

#ifdef APPLE64
#define UNIX
#define LP64
#define WITHZLIB
#define WITHGMP
#define WITHPOSIXCLOCK
#endif

#ifdef CYGWIN32
#define UNIX
#define ILP32
#define SIZEOF_LONG_LONG 8
#endif

#ifdef _MSC_VER
#define WINDOWS
#define _CRT_SECURE_NO_WARNINGS
#if defined(_WIN64)
#define LLP64
#elif defined(_WIN32)
#define ILP32
#define SIZEOF_LONG_LONG 8
#endif
#endif

/*
 * We must not define WITHPOSIXCLOCK in compiling the sequential FORM or ParFORM.
 */
#if !defined(WITHPTHREADS) && defined(WITHPOSIXCLOCK)
#undef WITHPOSIXCLOCK
#endif
```

```

#if !defined(__cplusplus) && !defined(inline)
#if (defined(__STDC_VERSION__) && __STDC_VERSION__ >= 199901L) || (defined(__GNUC__) &&
!defined(__STRICT_ANSI__))
/* "inline" is available. */
#elif defined(__GNUC__)
/* GNU C compiler has "__inline__". */
#define inline __inline__
#elif defined(_MSC_VER)
/* Microsoft C compiler has "__inline". */
#define inline __inline
#else
/* Inline functions may be not supported. Define "inline" to be empty. */
#define inline
#endif
#endif

#endif  /* HAVE_CONFIG_H */

/* Workaround for MSVC. */
#if defined(_MSC_VER)
/*
 * Recent versions of MSVC++ (>= 2012) don't like reserved keywords being
 * macroized even when they are not available. This is problematic for
 * `alignof`, which is used in legacy `PADXXX` macros. We disable tests in
 * xkeycheck.h.
 */
#if _MSC_VER >= 1700
#define _ALLOW_KEYWORD_MACROS
#endif
/*
 * Old versions of MSVC didn't support C99 function `snprintf`, which is used
 * in poly.cc. On the other hand, macroizing `snprintf` gives a fatal error
 * with MSVC >= 2015.
 */
#if _MSC_VER < 1900
#define snprintf _snprintf
#endif
#endif

/*
 * Translate our dialect "DEBUGGING" to the standard "NDEBUG".
 */
#ifdef DEBUGGING
#ifdef NDEBUG
#undef NDEBUG
#endif
#else
#ifndef NDEBUG
#define NDEBUG
#endif
#endif

/*
 * STATIC_ASSERT(condition) will fail to be compiled if the given
 * condition is false.
 */
#define STATIC_ASSERT(condition) STATIC_ASSERT__1(condition,__LINE__)
#define STATIC_ASSERT__1(X,L) STATIC_ASSERT__2(X,L)
#define STATIC_ASSERT__2(X,L) STATIC_ASSERT__3(X,L)
#define STATIC_ASSERT__3(X,L) \
    typedef char static_assertion_failed_##L[(!!(X))*2-1]

/*
 * UNIX or WINDOWS must be defined.
 */
#if defined(UNIX)
#define mBSD
#define ANSI
#elif defined(WINDOWS)
#define ANSI
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <io.h>
/* Undefine/rename conflicted symbols. */
#undef VOID  /* WinNT.h */
#undef MAXLONG  /* WinNT.h */
#define WORD FORM_WORD  /* WinDef.h */
#define LONG FORM_LONG /* WinNT.h */
#define ULONG FORM_ULONG  /* WinDef.h */
#undef CreateFile  /* WinBase.h */
#undef CopyFile  /* WinBase.h */
#define OpenFile FORM_OpenFile  /* WinBase.h */
#define ReOpenFile FORM_ReOpenFile  /* WinBase.h */
#define ReadFile FORM_ReadFile  /* WinBase.h */
#define WriteFile FORM_WriteFile  /* WinBase.h */
#define DeleteObject FORM_DeleteObject  /* WinGDI.h */
```

```
#else
#error UNIX or WINDOWS must be defined!
#endif

/*
 * Data model. ILP32 or LLP64 or LP64 must be defined.
 *
 * Here we define basic types WORD, LONG and their unsigned versions
 * UWORD and ULONG. LONG must be double size of WORD. Their actual types
 * are system-dependent. BITSINWORD and BITSINLONG are also defined.
 * INT16, INT32 (also INT64 and INT128 if available) are used for
 * system independent saved expressions (store.c).
 */
#if defined(ILP32)

typedef short WORD;
typedef long LONG;
typedef unsigned short UWORD;
typedef unsigned long ULONG;
#define BITSINWORD 16
#define BITSINLONG 32
#define INT16 short
#define INT32 int
#undef INT64
#undef INT128

#ifdef SIZEOF_LONG_LONG
#if SIZEOF_LONG_LONG == 8
#define INT64 long long
#endif
#endif

#ifndef INT64
#error INT64 is not available!
#endif

#elif defined(LLP64)

typedef int WORD;
typedef long long LONG;
typedef unsigned int UWORD;
typedef unsigned long long ULONG;
#define BITSINWORD 32
#define BITSINLONG 64
#define INT16 short
#define INT32 int
#define INT64 long long
#undef INT128

#elif defined(LP64)

typedef int WORD;
typedef long LONG;
typedef unsigned int UWORD;
typedef unsigned long ULONG;
#define BITSINWORD 32
#define BITSINLONG 64
#define INT16 short
#define INT32 int
#define INT64 long
#undef INT128

#else
#error ILP32 or LLP64 or LP64 must be defined!
#endif

STATIC_ASSERT(sizeof(WORD) * 8 == BITSINWORD);
STATIC_ASSERT(sizeof(LONG) * 8 == BITSINLONG);
STATIC_ASSERT(sizeof(WORD) * 2 == sizeof(LONG));
STATIC_ASSERT(sizeof(LONG) >= sizeof(int *));
STATIC_ASSERT(sizeof(INT16) == 2);
STATIC_ASSERT(sizeof(INT32) == 4);
STATIC_ASSERT(sizeof(INT64) == 8);
#ifdef INT128
STATIC_ASSERT(sizeof(INT128) == 16);
#endif

#if BITSINWORD == 32
#define WORDSIZE32 1
#endif

typedef void VOID;
typedef signed char SBYTE;
typedef unsigned char UBYTE;
typedef unsigned int UINT;
typedef ULONG RLONG;  /* Used in reken.c. */
typedef INT64 MLONG;  /* See commentary in minos.h. */
```

```
                                                        /* E.g. in 32-bits */
#define TOPBITONLY     ((ULONG)1 « (BITSINWORD - 1))  /* 0x00008000UL */
#define TOPLONGBITONLY ((ULONG)1 « (BITSINLONG - 1))  /* 0x80000000UL */
#define SPECMASK       ((UWORD)1 « (BITSINWORD - 1))  /*     0x8000U  */
#define WILDMASK       ((UWORD)1 « (BITSINWORD - 2))  /*     0x4000U  */
#define WORDMASK       ((ULONG)FULLMAX - 1)            /* 0x0000FFFFUL */
#define AWORDMASK      (WORDMASK « BITSINWORD)        /* 0xFFFF0000UL */
#define FULLMAX        ((LONG)1 « BITSINWORD)         /* 0x00010000L  */
#define MAXPOSITIVE    ((LONG)(TOPBITONLY - 1))        /* 0x00007FFFL  */
#define MAXLONG        ((LONG)(TOPLONGBITONLY - 1))    /* 0x7FFFFFFFL  */
#define MAXPOSITIVE2   (MAXPOSITIVE / 2)               /* 0x00003FFFL  */
#define MAXPOSITIVE4   (MAXPOSITIVE / 4)               /* 0x00001FFFL  */

/*
 * alignof(type) returns the number of bytes used in the alignment of
 * the type.
 */
#if !defined(alignof)
#if defined(__GNUC__)
/* GNU C compiler has "__alignof__". */
#define alignof(type) __alignof__(type)
#elif defined(_MSC_VER)
/* Microsoft C compiler has "__alignof". */
#define alignof(type) __alignof(type)
#elif !defined(__cplusplus)
/* Generic case in C. */
#include <stddef.h>
#define alignof(type) offsetof(struct { char c_; type x_; }, x_)
#else
/* Generic case in C++, at least works with a POD struct. */
#include <cstddef>
namespace alignof_impl_ {
template<typename T> struct calc {
    struct X { char c_; T x_; };
    enum { value = offsetof(X, x_) };
};
}
#define alignof(type) alignof_impl_::calc<type>::value
#endif
#endif

/*
 * Macros inserted to the end of a structure to align the whole structure.
 *
 * In the currently available systems,
 *   sizeof(POSITION) >= sizeof(pointers) == sizeof(LONG) >= sizeof(int)
 *                    >= sizeof(WORD) >= sizeof(UBYTE) = 1.
 * (POSITION is defined in struct.h and contains only an off_t variable.)
 * Thus, if we put members of a structure in this order and use those macros,
 * then we can align the data without relying on extra paddings added by
 * the compiler. For example,
 *   typedef struct {
 *     int *a;
 *     LONG b;
 *     WORD c[2];
 *     UBYTE d;
 *     PADPOINTER(1,0,2,1);
 *   } A;
 *   typedef struct {
 *     POSITION p;
 *     A a;  // aligned same as pointers
 *     int *b;
 *     LONG c;
 *     UBYTE d;
 *     PADPOSITION(1,1,0,0,1+sizeof(A));
 *   } B;
 * The cost for the use of those PADXXX functions is a padding (>= 1 byte) will
 * be always inserted even in the case that no padding is actually needed.
 *
 * Note that there is a 32-bit system in which off_t is aligned on 8-byte
 * boundary, (e.g., Cygwin).
 */
#define PADDUMMY(type, size) \
    UBYTE d_u_m_m_y[alignof(type) - ((size) & (alignof(type) - 1))]
#define PADPOSITION(ptr_,long_,int_,word_,byte_) \
    PADDUMMY(off_t, \
        + sizeof(int *) * (ptr_) \
        + sizeof(LONG)  * (long_) \
        + sizeof(int)   * (int_) \
        + sizeof(WORD)  * (word_) \
        + sizeof(UBYTE) * (byte_) \
    )
#define PADPOINTER(long_,int_,word_,byte_) \
    PADDUMMY(int *, \
        + sizeof(LONG)  * (long_) \
        + sizeof(int)   * (int_) \
        + sizeof(WORD)  * (word_) \
```

```
        + sizeof(UBYTE) * (byte_) \
    )
#define PADLONG(int_,word_,byte_) \
    PADDUMMY(LONG, \
        + sizeof(int)   * (int_) \
        + sizeof(WORD)  * (word_) \
        + sizeof(UBYTE) * (byte_) \
    )
#define PADINT(word_,byte_) \
    PADDUMMY(int, \
        + sizeof(WORD)  * (word_) \
        + sizeof(UBYTE) * (byte_) \
    )
#define PADWORD(byte_) \
    PADDUMMY(WORD, \
        + sizeof(UBYTE) * (byte_) \
    )

/*
#define WITHPCOUNTER
#define DEBUGGINGLOCKS
#define WITHSTATS
*/
#define WITHSORTBOTS

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#ifdef ANSI
#include <stdarg.h>
#include <time.h>
#endif
#ifdef WINDOWS
#include "fwin.h"
#endif
#ifdef UNIX
#include <unistd.h>
#include <time.h>
#include <fcntl.h>
#include <sys/file.h>
#include "unix.h"
#endif
#ifdef WITHZLIB
#include <zlib.h>
#endif
#ifdef WITHPTHREADS
#include <pthread.h>
#endif

/*
    PARALLELCODE indicates code that is common for TFORM and ParFORM but
    should not be there for sequential FORM.
*/
#if defined(WITHMPI) || defined(WITHPTHREADS)
#define PARALLELCODE
#endif

#include "ftypes.h"
#include "fsizes.h"
#include "minos.h"
#include "structs.h"
#include "declare.h"
#include "variable.h"

/*
 * The interface to file routines for UNIX or non-UNIX (Windows).
 */
#ifdef UNIX

#define UFILES
typedef struct FiLeS {
    int descriptor;
} FILES;
extern FILES *Uopen(char *,char *);
extern int    Uclose(FILES *);
extern size_t Uread(char *,size_t,size_t,FILES *);
extern size_t Uwrite(char *,size_t,size_t,FILES *);
extern int    Useek(FILES *,off_t,int);
extern off_t  Utell(FILES *);
extern void   Uflush(FILES *);
extern int    Ugetpos(FILES *,fpos_t *);
extern int    Usetpos(FILES *,fpos_t *);
extern void   Usetbuf(FILES *,char *);
#define Usync(f) fsync(f->descriptor)
#define Utruncate(f) { \
    if ( ftruncate(f->descriptor, 0) ) { \
```

```
        MLOCK(ErrorMessageLock); \
        MesPrint("Utruncate failed"); \
        MUNLOCK(ErrorMessageLock); \
        /* Calling Terminate() here may cause an infinite loop due to CleanUpSort(). */ \
        /* Terminate(-1); */ \
    } \
}
extern FILES *Ustdout;
#define MAX_OPEN_FILES getdtablesize()
#define GetPID() ((LONG)getpid())

#else  /* UNIX */

#define FILES FILE
#define Uopen(x,y) fopen(x,y)
#define Uflush(x) fflush(x)
#define Uclose(x) fclose(x)
#define Uread(x,y,z,u) fread(x,y,z,u)
#define Uwrite(x,y,z,u) fwrite(x,y,z,u)
#define Usetbuf(x,y) setbuf(x,y)
#define Useek(x,y,z) fseek(x,y,z)
#define Utell(x) ftell(x)
#define Ugetpos(x,y) fgetpos(x,y)
#define Usetpos(x,y) fsetpos(x,y)
#define Usync(x) fflush(x)
#define Utruncate(x) _chsize(_fileno(x),0)
#define Ustdout stdout
#define MAX_OPEN_FILES FOPEN_MAX
#define bzero(b,len) (memset((b), 0, (len)), (void)0)
#define GetPID() ((LONG)GetCurrentProcessId())

#endif  /* UNIX */

#ifdef WITHMPI
#include "parallel.h"
#endif

#endif  /*  __FORM3H__ */
```

4.35 fsizes.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define MAXPRENAMESIZE 128
- #define MAXPOWER 10000
- #define MAXVARIABLES 8050
- #define MAXDOLLARVARIABLES 32000
- #define WILDOFFSET 6100
- #define MAXINNAMETREE 32768
- #define MAXDUMMIES 1000
- #define WORKBUFFER 10000000
- #define MAXTER 10000
- #define HALFMAX 0x100
- #define MAXSUBEXPRESSIONS 0x3FFF
- #define MAXFILESTREAMSIZE 1048576
- #define MAXENAME 16
- #define MAXSAVEFUNCTION 16384
- #define MAXPARLEVEL 100
- #define MAXNUMBERSIZE 200

- #define MAXREPEAT 100
- #define NORMSIZE 1000
- #define INITNODESIZE 10
- #define INITNAMESIZE 100
- #define NUMFIXED 128
- #define MAXNEST 100
- #define MAXMATCH 30
- #define MAXIF 20
- #define SIZEFACS 640L
- #define NUMFACS 50
- #define MAXLOOPS 30
- #define MAXLABELS 20
- #define COMMERCIALSIZE 24
- #define MAXFLAGS 16
- #define COMPRESSBUFFER 90000
- #define FORTRANCONTINUATIONLINES 15
- #define MAXLEVELS 2000
- #define MAXLHS 400
- #define MAXWILDC 100
- #define NUMTABLEENTRIES 1000
- #define COMPILERBUFFER 20000
- #define SMALLBUFFER 10000000L
- #define SMALLOVERFLOW 20000000L
- #define TERMSSMALL 100000L
- #define LARGEBUFFER 50000000L
- #define MAXPATCHES 256
- #define MAXFPATCHES 256
- #define SORTIOSIZE 100000L
- #define SSMALLBUFFER 500000L
- #define SSMALLOVERFLOW 800000L
- #define STERMSSMALL 10000L
- #define SLARGEBUFFER 4000000L
- #define SMAXPATCHES 64
- #define SMAXFPATCHES 64
- #define SSORTIOSIZE 32768L
- #define SCRATCHSIZE 50000000L
- #define SPECTATORSIZE 1048576L
- #define MAXFLEVELS 30
- #define COMPINC 2
- #define MAXNUMSIZE 10
- #define MAXBRACKETBUFFERSIZE 200000
- #define SFHSIZE 40
- #define DEFAULTPROCESSBUCKETSIZE 1000
- #define SHMWINSIZE 65536L
- #define TABLEEXTENSION 6
- #define GZIPDEFAULT 6
- #define DEFAULTTHREADS 0
- #define DEFAULTTHREADBUCKETSIZE 500
- #define DEFAULTTHREADLOADBALANCING 1
- #define THREADSCRATCHSIZE 100000L
- #define THREADSCRATCHOUTSIZE 2500000L
- #define MAXTABLECOMBUF 1000000L
- #define MAXCOMBUFRHS 32500L
- #define NUMSTORECACHES 4
- #define SIZESTORECACHE 32768

- #define INDENTSPACE 3
- #define MULTIINDENTSPACE 1
- #define MAXMULTIBRACKETLEVELS 25
- #define FBUFFERSIZE 1026
- #define NPAIR1 38
- #define NPAIR2 89
- #define MAXLINELENGTH 256

4.35.1 Detailed Description

The definition the default values for certain buffer sizes etc.

Definition in file fsizes.h.

4.35.2 Macro Definition Documentation

4.35.2.1 MAXPRENAMESIZE

```
#define MAXPRENAMESIZE 128
```

Definition at line 36 of file fsizes.h.

4.35.2.2 MAXPOWER

```
#define MAXPOWER 10000
```

Definition at line 58 of file fsizes.h.

4.35.2.3 MAXVARIABLES

```
#define MAXVARIABLES 8050
```

Definition at line 59 of file fsizes.h.

4.35.2.4 MAXDOLLARVARIABLES

```
#define MAXDOLLARVARIABLES 32000
```

Definition at line 60 of file fsizes.h.

4.35.2.5 WILDOFFSET

```
#define WILDOFFSET 6100
```

Definition at line 61 of file fsizes.h.

4.35.2.6 MAXINNAMETREE

```
#define MAXINNAMETREE 32768
```

Definition at line 62 of file fsizes.h.

4.35.2.7 MAXDUMMIES

```
#define MAXDUMMIES 1000
```

Definition at line 63 of file fsizes.h.

4.35.2.8 WORKBUFFER

```
#define WORKBUFFER 10000000
```

Definition at line 64 of file fsizes.h.

4.35.2.9 MAXTER

```
#define MAXTER 10000
```

Definition at line 65 of file fsizes.h.

4.35.2.10 HALFMAX

```
#define HALFMAX 0x100
```

Definition at line 66 of file fsizes.h.

4.35.2.11 MAXSUBEXPRESSIONS

```
#define MAXSUBEXPRESSIONS 0x3FFF
```

Definition at line 67 of file fsizes.h.

4.35.2.12 MAXFILESTREAMSIZE

```
#define MAXFILESTREAMSIZE 1048576
```

Definition at line 68 of file fsizes.h.

4.35.2.13 MAXENAME

```
#define MAXENAME 16
```

Definition at line 70 of file fsizes.h.

4.35.2.14 MAXSAVEFUNCTION

```
#define MAXSAVEFUNCTION 16384
```

Definition at line 71 of file fsizes.h.

4.35.2.15 MAXPARLEVEL

```
#define MAXPARLEVEL 100
```

Definition at line 73 of file fsizes.h.

4.35.2.16 MAXNUMBERSIZE

```
#define MAXNUMBERSIZE 200
```

Definition at line 74 of file fsizes.h.

4.35.2.17 MAXREPEAT

```
#define MAXREPEAT 100
```

Definition at line 76 of file fsizes.h.

4.35.2.18 NORMSIZE

```
#define NORMSIZE 1000
```

Definition at line 77 of file fsizes.h.

4.35.2.19 INITNODESIZE

```
#define INITNODESIZE 10
```

Definition at line 79 of file fsizes.h.

4.35.2.20 INITNAMESIZE

```
#define INITNAMESIZE 100
```

Definition at line 80 of file fsizes.h.

4.35.2.21 NUMFIXED

```
#define NUMFIXED 128
```

Definition at line 82 of file fsizes.h.

4.35.2.22 MAXNEST

```
#define MAXNEST 100
```

Definition at line 83 of file fsizes.h.

4.35.2.23 MAXMATCH

```
#define MAXMATCH 30
```

Definition at line 84 of file fsizes.h.

4.35.2.24 MAXIF

```
#define MAXIF 20
```

Definition at line 85 of file fsizes.h.

4.35.2.25 SIZEFACS

```
#define SIZEFACS 640L
```

Definition at line 86 of file fsizes.h.

4.35.2.26 NUMFACS

```
#define NUMFACS 50
```

Definition at line 87 of file fsizes.h.

4.35.2.27 MAXLOOPS

```
#define MAXLOOPS 30
```

Definition at line 88 of file fsizes.h.

4.35.2.28 MAXLABELS

```
#define MAXLABELS 20
```

Definition at line 89 of file fsizes.h.

4.35.2.29 COMMERCIALSIZE

```
#define COMMERCIALSIZE 24
```

Definition at line 90 of file fsizes.h.

4.35.2.30 MAXFLAGS

```
#define MAXFLAGS 16
```

Definition at line 91 of file fsizes.h.

4.35.2.31 COMPRESSBUFFER

```
#define COMPRESSBUFFER 90000
```

Definition at line 96 of file fsizes.h.

4.35.2.32 FORTRANCONTINUATIONLINES

```
#define FORTRANCONTINUATIONLINES 15
```

Definition at line 97 of file fsizes.h.

4.35.2.33 MAXLEVELS

```
#define MAXLEVELS 2000
```

Definition at line 98 of file fsizes.h.

4.35.2.34 MAXLHS

```
#define MAXLHS 400
```

Definition at line 99 of file fsizes.h.

4.35.2.35 MAXWILDC

```
#define MAXWILDC 100
```

Definition at line 100 of file fsizes.h.

4.35.2.36 NUMTABLEENTRIES

```
#define NUMTABLEENTRIES 1000
```

Definition at line 101 of file fsizes.h.

4.35.2.37 COMPILERBUFFER

```
#define COMPILERBUFFER 20000
```

Definition at line 102 of file fsizes.h.

4.35.2.38 SMALLBUFFER

```
#define SMALLBUFFER 10000000L
```

Definition at line 104 of file fsizes.h.

4.35.2.39 SMALLOVERFLOW

```
#define SMALLOVERFLOW 20000000L
```

Definition at line 105 of file fsizes.h.

4.35.2.40 TERMSSMALL

```
#define TERMSSMALL 100000L
```

Definition at line 106 of file fsizes.h.

4.35.2.41 LARGEBUFFER

```
#define LARGEBUFFER 50000000L
```

Definition at line 107 of file fsizes.h.

4.35.2.42 MAXPATCHES

```
#define MAXPATCHES 256
```

Definition at line 108 of file fsizes.h.

4.35.2.43 MAXFPATCHES

```
#define MAXFPATCHES 256
```

Definition at line 109 of file fsizes.h.

4.35.2.44 SORTIOSIZE

```
#define SORTIOSIZE 100000L
```

Definition at line 110 of file fsizes.h.

4.35.2.45 SSMALLBUFFER

```
#define SSMALLBUFFER 500000L
```

Definition at line 112 of file fsizes.h.

4.35.2.46 SSMALLOVERFLOW

```
#define SSMALLOVERFLOW 800000L
```

Definition at line 113 of file fsizes.h.

4.35.2.47 STERMSSMALL

```
#define STERMSSMALL 10000L
```

Definition at line 114 of file fsizes.h.

4.35.2.48 SLARGEBUFFER

```
#define SLARGEBUFFER 4000000L
```

Definition at line 115 of file fsizes.h.

4.35.2.49 SMAXPATCHES

```
#define SMAXPATCHES 64
```

Definition at line 116 of file fsizes.h.

4.35.2.50 SMAXFPATCHES

```
#define SMAXFPATCHES 64
```

Definition at line 117 of file fsizes.h.

4.35.2.51 SSORTIOSIZE

```
#define SSORTIOSIZE 32768L
```

Definition at line 118 of file fsizes.h.

4.35.2.52 SCRATCHSIZE

```
#define SCRATCHSIZE 50000000L
```

Definition at line 120 of file fsizes.h.

4.35.2.53 SPECTATORSIZE

```
#define SPECTATORSIZE 1048576L
```

Definition at line 121 of file fsizes.h.

4.35.2.54 MAXFLEVELS

```
#define MAXFLEVELS 30
```

Definition at line 123 of file fsizes.h.

4.35.2.55 COMPINC

```
#define COMPINC 2
```

Definition at line 125 of file fsizes.h.

4.35.2.56 MAXNUMSIZE

```
#define MAXNUMSIZE 10
```

Definition at line 127 of file fsizes.h.

4.35.2.57 MAXBRACKETBUFFERSIZE

```
#define MAXBRACKETBUFFERSIZE 200000
```

Definition at line 129 of file fsizes.h.

4.35.2.58 SFHSIZE

```
#define SFHSIZE 40
```

Definition at line 131 of file fsizes.h.

4.35.2.59 DEFAULTPROCESSBUCKETSIZE

```
#define DEFAULTPROCESSBUCKETSIZE 1000
```

Definition at line 133 of file fsizes.h.

4.35.2.60 SHMWINSIZE

```
#define SHMWINSIZE 65536L
```

Definition at line 134 of file fsizes.h.

4.35.2.61 TABLEEXTENSION

```
#define TABLEEXTENSION 6
```

Definition at line 136 of file fsizes.h.

4.35.2.62 GZIPDEFAULT

```
#define GZIPDEFAULT 6
```

Definition at line 138 of file fsizes.h.

4.35.2.63 DEFAULTTHREADS

```
#define DEFAULTTHREADS 0
```

Definition at line 139 of file fsizes.h.

4.35.2.64 DEFAULTTHREADBUCKETSIZE

```
#define DEFAULTTHREADBUCKETSIZE 500
```

Definition at line 140 of file fsizes.h.

4.35.2.65 DEFAULTTHREADLOADBALANCING

```
#define DEFAULTTHREADLOADBALANCING 1
```

Definition at line 141 of file fsizes.h.

4.35.2.66 THREADSCRATCHSIZE

```
#define THREADSCRATCHSIZE 100000L
```

Definition at line 142 of file fsizes.h.

4.35.2.67 THREADSCRATCHOUTSIZE

```
#define THREADSCRATCHOUTSIZE 2500000L
```

Definition at line 143 of file fsizes.h.

4.35.2.68 MAXTABLECOMBUF

```
#define MAXTABLECOMBUF 1000000L
```

Definition at line 149 of file fsizes.h.

4.35.2.69 MAXCOMBUFRHS

```
#define MAXCOMBUFRHS 32500L
```

Definition at line 150 of file fsizes.h.

4.35.2.70 NUMSTORECACHES

```
#define NUMSTORECACHES 4
```

Definition at line 153 of file fsizes.h.

4.35.2.71 SIZESTORECACHE

```
#define SIZESTORECACHE 32768
```

Definition at line 154 of file fsizes.h.

4.35.2.72 INDENTSPACE

```
#define INDENTSPACE 3
```

Definition at line 156 of file fsizes.h.

4.35.2.73 MULTIINDENTSPACE

```
#define MULTIINDENTSPACE 1
```

Definition at line 158 of file fsizes.h.

4.35.2.74 MAXMULTIBRACKETLEVELS

```
#define MAXMULTIBRACKETLEVELS 25
```

Definition at line 159 of file fsizes.h.

4.35.2.75 FBUFFERSIZE

```
#define FBUFFERSIZE 1026
```

Definition at line 161 of file fsizes.h.

4.35.2.76 NPAIR1

```
#define NPAIR1 38
```

Definition at line 165 of file fsizes.h.

4.35.2.77 NPAIR2

```
#define NPAIR2 89
```

Definition at line 166 of file fsizes.h.

4.35.2.78 MAXLINELENGTH

```
#define MAXLINELENGTH 256
```

Definition at line 168 of file fsizes.h.

4.36 fsizes.h

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032
00033 /*
00034     First the fixed variables
00035 */
00036 #define MAXPRENAMESIZE 128
00037 /*
00038     The following variables are default sizes. They can be changed
00039     into values read from the setup file
00040
00041     Remark (21-dec-2008 JV): WILDOFFSET*3 should be larger than WILDMASK!!!!
00042         old value was WILDOFFSET 200000100
00043         be careful with old .sav files!!!
00044 */
00045 #ifdef WORDSIZE32
00046 #define MAXPOWER 500000000
00047 #define MAXVARIABLES 200000050
00048 #define MAXDOLLARVARIABLES 1000000000L
00049 #define WILDOFFSET 400000100
00050 #define MAXINNAMETREE 2000000000
00051 #define MAXDUMMIES 100000000
00052 #define WORKBUFFER 40000000
00053 #define MAXTER 40000
00054 #define HALFMAX 0x10000
00055 #define MAXSUBEXPRESSIONS 0x1FFFFFF
00056 #define MAXFILESTREAMSIZE 1024
00057 #else
00058 #define MAXPOWER 10000
00059 #define MAXVARIABLES 8050
00060 #define MAXDOLLARVARIABLES 32000
00061 #define WILDOFFSET 6100
00062 #define MAXINNAMETREE 32768
00063 #define MAXDUMMIES 1000
00064 #define WORKBUFFER 10000000
00065 #define MAXTER 10000
00066 #define HALFMAX 0x100
00067 #define MAXSUBEXPRESSIONS 0x3FFF
00068 #define MAXFILESTREAMSIZE 1048576
00069 #endif
00070 #define MAXENAME 16
00071 #define MAXSAVEFUNCTION 16384
00072
00073 #define MAXPARLEVEL 100
```

```
#define MAXNUMBERSIZE 200

#define MAXREPEAT 100
#define NORMSIZE 1000

#define INITNODESIZE 10
#define INITNAMESIZE 100

#define NUMFIXED 128
#define MAXNEST 100
#define MAXMATCH 30
#define MAXIF 20
#define SIZEFACS 640L
#define NUMFACS 50
#define MAXLOOPS 30
#define MAXLABELS 20
#define COMMERCIALSIZE 24
#define MAXFLAGS 16
/*
    The next quantities should still be eliminated from the program
    This should be together with changes in setfile!
*/
#define COMPRESSBUFFER 90000
#define FORTRANCONTINUATIONLINES 15
#define MAXLEVELS 2000
#define MAXLHS 400
#define MAXWILDC 100
#define NUMTABLEENTRIES 1000
#define COMPILERBUFFER 20000

#define SMALLBUFFER   10000000L
#define SMALLOVERFLOW 20000000L
#define TERMSSMALL      100000L
#define LARGEBUFFER   50000000L
#define MAXPATCHES 256
#define MAXFPATCHES 256
#define SORTIOSIZE 100000L

#define SSMALLBUFFER 500000L
#define SSMALLOVERFLOW 800000L
#define STERMSSMALL 10000L
#define SLARGEBUFFER 4000000L
#define SMAXPATCHES 64
#define SMAXFPATCHES 64
#define SSORTIOSIZE 32768L

#define SCRATCHSIZE 50000000L
#define SPECTATORSIZE 1048576L

#define MAXFLEVELS 30

#define COMPINC 2

#define MAXNUMSIZE 10

#define MAXBRACKETBUFFERSIZE 200000

#define SFHSIZE 40

#define DEFAULTPROCESSBUCKETSIZE 1000
#define SHMWINSIZE     65536L

#define TABLEEXTENSION 6

#define GZIPDEFAULT 6
#define DEFAULTTHREADS 0
#define DEFAULTTHREADBUCKETSIZE 500
#define DEFAULTTHREADLOADBALANCING 1
#define THREADSCRATCHSIZE 100000L
#define THREADSCRATCHOUTSIZE 2500000L

#ifdef WORDSIZE32
#define MAXTABLECOMBUF 100000000000L
#define MAXCOMBUFRHS 1000000000L
#else
#define MAXTABLECOMBUF 1000000L
#define MAXCOMBUFRHS 32500L
#endif

#define NUMSTORECACHES 4
#define SIZESTORECACHE 32768

#define INDENTSPACE 3

#define MULTIINDENTSPACE 1
#define MAXMULTIBRACKETLEVELS 25

```

```
#define FBUFFERSIZE 1026
/*
    For the random number generator (see commentary there)
*/
#define NPAIR1 38
#define NPAIR2 89

#define MAXLINELENGTH 256
```

4.37 ftypes.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define PHEAD
- #define PHEAD0 VOID
- #define BHEAD
- #define BHEAD0
- #define WITHOUTERROR 0
- #define WITHERROR 1
- #define FILESTREAM 0
- #define PREVARSTREAM 1
- #define PREREADSTREAM 2
- #define PIPESTREAM 3
- #define PRECALCSTREAM 4
- #define DOLLARSTREAM 5
- #define PREREADSTREAM2 6
- #define EXTERNALCHANNELSTREAM 7
- #define PREREADSTREAM3 8
- #define REVERSEFILESTREAM 9
- #define ENDOFSTREAM 0xFF
- #define ENDOFINPUT 0xFF
- #define SUBROUTINEFILE 0
- #define PROCEDUREFILE 1
- #define HEADERFILE 2
- #define SETUPFILE 3
- #define TABLEBASEFILE 4
- #define FIRSTMODULE -1
- #define GLOBALMODULE 0
- #define SORTMODULE 1
- #define STOREMODULE 2
- #define CLEARMODULE 3
- #define ENDMODULE 4
- #define POLYFUN 0
- #define NOPARALLEL_DOLLAR 0x0001
- #define NOPARALLEL_RHS 0x0002
- #define NOPARALLEL_CONVPOLY 0x0004
- #define NOPARALLEL_SPECTATOR 0x0008
- #define NOPARALLEL_USER 0x0010

- #define NOPARALLEL_TBLDOLLAR 0x0100
- #define NOPARALLEL_NPROC 0x0200
- #define PARALLELFLAG 0x0000
- #define PRENOACTION 0
- #define PRERAISEAFTER 1
- #define PRELOWERAFTER 2
- #define WITHSEMICOLON 0
- #define WITHOUTSEMICOLON 1
- #define MODULEINSTACK 8
- #define EXECUTINGIF 0
- #define LOOKINGFORELSE 1
- #define LOOKINGFORENDIF 2
- #define NEWSTATEMENT 1
- #define OLDSTATEMENT 0
- #define EXECUTINGPRESWITCH 0
- #define SEARCHINGPRECASE 1
- #define SEARCHINGPREENDSWITCH 2
- #define PREPROONLY 1
- #define DUMPTOCOMPILER 2
- #define DUMPOUTTERMS 4
- #define DUMPINTERMS 8
- #define DUMPTOSORT 16
- #define DUMPTOPARALLEL 32
- #define THREADSDEBUG 64
- #define ERROROUT 0
- #define INPUTOUT 1
- #define STATSOUT 2
- #define EXPRSOUT 3
- #define WRITEOUT 4
- #define EXTERNALCHANNELOUT 5
- #define NUMERICALLOOP 0
- #define LISTEDLOOP 1
- #define ONEEXPRESSION 2
- #define PRETYPENONE 0
- #define PRETYPEIF 1
- #define PRETYPEDO 2
- #define PRETYPEPROCEDURE 3
- #define PRETYPESWITCH 4
- #define PRETYPEINSIDE 5
- #define DECLARATION 1
- #define SPECIFICATION 2
- #define DEFINITION 3
- #define STATEMENT 4
- #define TOOUTPUT 5
- #define ATENDOFMODULE 6
- #define MIXED 9
- #define VOID void
- #define NOAUTO 0
- #define PARTEST 1
- #define WITHAUTO 2
- #define ALLVARIABLES -1
- #define SYMBOLONLY 1
- #define INDEXONLY 2
- #define VECTORONLY 4
- #define FUNCTIONONLY 8

- #define SETONLY 16
- #define EXPRESSIONONLY 32

Defines: compiler types

Type of variable found by the compiler.

- #define CDELETE -1
- #define ANYTYPE -1
- #define CSYMBOL 0
- #define CINDEX 1
- #define CVECTOR 2
- #define CFUNCTION 3
- #define CSET 4
- #define CEXPRESSION 5
- #define CDOTPRODUCT 6
- #define CNUMBER 7
- #define CSUBEXP 8
- #define CDELTA 9
- #define CDOLLAR 10
- #define CDUBIOUS 11
- #define CRANGE 12
- #define CVECTOR1 21
- #define CDOUBLEDOT 22
- #define TSYMBOL -1
- #define TINDEX -2
- #define TVECTOR -3
- #define TFUNCTION -4
- #define TSET -5
- #define TEXPRESSION -6
- #define TDOTPRODUCT -7
- #define TNUMBER -8
- #define TSUBEXP -9
- #define TDELTA -10
- #define TDOLLAR -11
- #define TDUBIOUS -12
- #define LPARENTHESIS -13
- #define RPARENTHESIS -14
- #define TWILDCARD -15
- #define TWILDARG -16
- #define TDOT -17
- #define LBRACE -18
- #define RBRACE -19
- #define TCOMMA -20
- #define TFUNOPEN -21
- #define TFUNCLOSE -22
- #define TMULTIPLY -23
- #define TDIVIDE -24
- #define TPOWER -25
- #define TPLUS -26
- #define TMINUS -27
- #define TNOT -28
- #define TENDOFIT -29
- #define TSETOPEN -30
- #define TSETCLOSE -31
- #define TGENINDEX -32
- #define TCONJUGATE -33
- #define LRPARENTHESES -34
- #define TNUMBER1 -35
- #define TPOWER1 -36
- #define TEMPTY -37
- #define TSETNUM -38
- #define TSGAMMA -39
- #define TSETDOL -40

- #define TYPEISFUN 0
- #define TYPEISSUB 1
- #define TYPEISMYSTERY -1
- #define LHSIDEX 2
- #define LHSIDE 1
- #define RHSIDE 0
- #define FORTRANMODE 1
- #define REDUCEMODE 2
- #define MAPLEMODE 3
- #define MATHEMATICAMODE 4
- #define CMODE 5
- #define VORTRANMODE 6
- #define PFORTRANMODE 7
- #define DOUBLEFORTRANMODE 33
- #define DOUBLEPRECISIONFLAG 32
- #define NODOUBLEMASK 31
- #define QUADRUPLEFORTRANMODE 65
- #define QUADRUPLEPRECISIONFLAG 64
- #define NOQUADMASK 63
- #define NORMALFORMAT 0
- #define NOSPACEFORMAT 1
- #define ISNOTFORTRAN90 0
- #define ISFORTRAN90 1
- #define ALSOREVERSE 1
- #define CHISHOLM 2
- #define NOTRICK 16
- #define SORTLOWFIRST 0
- #define SORTHIGHFIRST 1
- #define SORTPOWERFIRST 2
- #define SORTANTIPOWER 3
- #define NMIN4SHIFT 4
- #define SYMBOL 1
- #define DOTPRODUCT 2
- #define VECTOR 3
- #define INDEX 4
- #define EXPRESSION 5
- #define SUBEXPRESSION 6
- #define DOLLAREXPRESSION 7
- #define SETSET 8
- #define ARGWILD 9
- #define MINVECTOR 10
- #define SETEXP 11
- #define DOLLAREXPR2 12
- #define HAAKJE0 9
- #define FUNCTION 20
- #define TMPPOLYFUN 14
- #define ARGFIELD 15
- #define SNUMBER 16
- #define LNUMBER 17
- #define HAAKJE 18
- #define DELTA 19
- #define EXPONENT 20
- #define DENOMINATOR 21
- #define SETFUNCTION 22
- #define GAMMA 23
- #define GAMMAI 24
- #define GAMMAFIVE 25
- #define GAMMASIX 26
- #define GAMMASEVEN 27
- #define SUMF1 28
- #define SUMF2 29
- #define DUMFUN 30
- #define REPLACEMENT 31
- #define REVERSEFUNCTION 32

- #define DISTRIBUTION 33
- #define DELTA3 34
- #define DUMMYFUN 35
- #define DUMMYTEN 36
- #define LEVICIVITA 37
- #define FACTORIAL 38
- #define INVERSEFACTORIAL 39
- #define BINOMIAL 40
- #define NUMARGSFUN 41
- #define SIGNFUN 42
- #define MODFUNCTION 43
- #define MOD2FUNCTION 44
- #define MINFUNCTION 45
- #define MAXFUNCTION 46
- #define ABSFUNCTION 47
- #define SIGFUNCTION 48
- #define INTFUNCTION 49
- #define THETA 50
- #define THETA2 51
- #define DELTA2 52
- #define DELTAP 53
- #define BERNOULLIFUNCTION 54
- #define COUNTFUNCTION 55
- #define MATCHFUNCTION 56
- #define PATTERNFUNCTION 57
- #define TERMFUNCTION 58
- #define CONJUGATION 59
- #define ROOTFUNCTION 60
- #define TABLEFUNCTION 61
- #define FIRSTBRACKET 62
- #define TERMSINEXPR 63
- #define NUMTERMSFUN 64
- #define GCDFUNCTION 65
- #define DIVFUNCTION 66
- #define REMFUNCTION 67
- #define MAXPOWEROF 68
- #define MINPOWEROF 69
- #define TABLESTUB 70
- #define FACTORIN 71
- #define TERMSINBRACKET 72
- #define WILDARGFUN 73
- #define SQRTFUNCTION 74
- #define LNFUNCTION 75
- #define SINFUNCTION 76
- #define COSFUNCTION 77
- #define TANFUNCTION 78
- #define ASINFUNCTION 79
- #define ACOSFUNCTION 80
- #define ATANFUNCTION 81
- #define ATAN2FUNCTION 82
- #define SINHFUNCTION 83
- #define COSHFUNCTION 84
- #define TANHFUNCTION 85
- #define ASINHFUNCTION 86
- #define ACOSHFUNCTION 87
- #define ATANHFUNCTION 88
- #define LI2FUNCTION 89
- #define LINFUNCTION 90
- #define EXTRASYMFUN 91
- #define RANDOMFUNCTION 92
- #define RANPERM 93
- #define NUMFACTORS 94
- #define FIRSTTERM 95
- #define CONTENTTERM 96

- #define PRIMENUMBER 97
- #define EXTEUCLIDEAN 98
- #define MAKERATIONAL 99
- #define INVERSEFUNCTION 100
- #define IDFUNCTION 101
- #define PUTFIRST 102
- #define PERMUTATIONS 103
- #define PARTITIONS 104
- #define MULFUNCTION 105
- #define MAXBUILTINFUNCTION 105
- #define FIRSTUSERFUNCTION 150
- #define ISYMBOL 0
- #define PISYMBOL 1
- #define COEFFSYMBOL 2
- #define NUMERATORSYMBOL 3
- #define DENOMINATORSYMBOL 4
- #define WILDARGSYMBOL 5
- #define DIMENSIONSYMBOL 6
- #define FACTORSYMBOL 7
- #define SEPARATESYMBOL 8
- #define BUILTINSYMBOLS 9
- #define FIRSTUSERSYMBOL 20
- #define BUILTININDICES 1
- #define BUILTINVECTORS 1
- #define BUILTINDOLLARS 1
- #define WILDARGVECTOR 0
- #define WILDARGINDEX 0
- #define TYPEEXPRESSION 0
- #define TYPEIDNEW 1
- #define TYPEIDOLD 2
- #define TYPEOPERATION 3
- #define TYPEREPEAT 4
- #define TYPEENDREPEAT 5
- #define TYPECOUNT 20
- #define TYPEMULT 21
- #define TYPEGOTO 22
- #define TYPEDISCARD 23
- #define TYPEIF 24
- #define TYPEELSE 25
- #define TYPEELIF 26
- #define TYPEENDIF 27
- #define TYPESUM 28
- #define TYPECHISHOLM 29
- #define TYPEREVERSE 30
- #define TYPEARG 31
- #define TYPENORM 32
- #define TYPENORM2 33
- #define TYPENORM3 34
- #define TYPEEXIT 35
- #define TYPESETEXIT 36
- #define TYPEPRINT 37
- #define TYPEFPRINT 38
- #define TYPEREDEFPRE 39
- #define TYPESPLITARG 40
- #define TYPESPLITARG2 41
- #define TYPEFACTARG 42
- #define TYPEFACTARG2 43
- #define TYPETRY 44
- #define TYPEASSIGN 45
- #define TYPERENUMBER 46
- #define TYPESUMFIX 47
- #define TYPEFINDLOOP 48
- #define TYPEUNRAVEL 49
- #define TYPEADJUSTBOUNDS 50

- #define TYPEINSIDE 51
- #define TYPETERM 52
- #define TYPESORT 53
- #define TYPEDETCURDUM 54
- #define TYPEINEXPRESSION 55
- #define TYPESPLITFIRSTARG 56
- #define TYPESPLITLASTARG 57
- #define TYPEMERGE 58
- #define TYPETESTUSE 59
- #define TYPEAPPLY 60
- #define TYPEAPPLYRESET 61
- #define TYPECHAININ 62
- #define TYPECHAINOUT 63
- #define TYPENORM4 64
- #define TYPEFACTOR 65
- #define TYPEARGIMPLODE 66
- #define TYPEARGEXPLODE 67
- #define TYPEDENOMINATORS 68
- #define TYPESTUFFLE 69
- #define TYPEDROPCOEFFICIENT 70
- #define TYPETRANSFORM 71
- #define TYPETOPOLYNOMIAL 72
- #define TYPEFROMPOLYNOMIAL 73
- #define TYPEDOLOOP 74
- #define TYPEENDDOLOOP 75
- #define TYPEDROPSYMBOLS 76
- #define TYPEPUTINSIDE 77
- #define TYPETOSPECTATOR 78
- #define TYPEARGTOEXTRASYMBOL 79
- #define TAKETRACE 1
- #define CONTRACT 2
- #define RATIO 3
- #define SYMMETRIZE 4
- #define TENVEC 5
- #define SUMNUM1 6
- #define SUMNUM2 7
- #define WILDDUMMY 0
- #define SYMTONUM 1
- #define SYMTOSYM 2
- #define SYMTOSUB 3
- #define VECTOMIN 4
- #define VECTOVEC 5
- #define VECTOSUB 6
- #define INDTOIND 7
- #define INDTOSUB 8
- #define FUNTOFUN 9
- #define ARGTOARG 10
- #define ARLTOARL 11
- #define EXPTOEXP 12
- #define FROMBRAC 13
- #define FROMSET 14
- #define SETTONUM 15
- #define WILDCARDS 16
- #define SETNUMBER 17
- #define LOADDOLLAR 18
- #define NUMTONUM 20
- #define NUMTOSYM 21
- #define NUMTOIND 22
- #define NUMTOSUB 23
- #define CLEANFLAG 0
- #define DIRTYFLAG 1
- #define DIRTYSYMFLAG 2
- #define MUSTCLEANPRF 4
- #define ALLDIRTY (DIRTYFLAG|DIRTYSYMFLAG)

- #define ARGHEAD 2
- #define FUNHEAD 3
- #define SUBEXPSIZE 5
- #define EXPRHEAD 5
- #define TYPEARGHEADSIZE 6
- #define SKIP 1
- #define DROP 2
- #define HIDE 3
- #define UNHIDE 4
- #define INTOHIDE 5
- #define LOCALEXPRESSION 0
- #define SKIPLEXPRESSION 1
- #define DROPLEXPRESSION 2
- #define DROPPEDEXPRESSION 3
- #define GLOBALEXPRESSION 4
- #define SKIPGEXPRESSION 5
- #define DROPGEXPRESSION 6
- #define STOREDEXPRESSION 8
- #define HIDDENLEXPRESSION 9
- #define HIDDENGEXPRESSION 13
- #define INCEXPRESSION 9
- #define HIDELEXPRESSION 10
- #define HIDEGEXPRESSION 14
- #define DROPHLEXPRESSION 11
- #define DROPHGEXPRESSION 15
- #define UNHIDELEXPRESSION 12
- #define UNHIDEGEXPRESSION 16
- #define INTOHIDELEXPRESSION 17
- #define INTOHIDEGEXPRESSION 18
- #define SPECTATOREXPRESSION 19
- #define DROPSPECTATOREXPRESSION 20
- #define SKIPUNHIDELEXPRESSION 21
- #define SKIPUNHIDEGEXPRESSION 22
- #define PRINTOFF 0
- #define PRINTON 1
- #define PRINTCONTENTS 2
- #define PRINTCONTENT 3
- #define PRINTLFILE 4
- #define PRINTONETERM 8
- #define PRINTONEFUNCTION 16
- #define PRINTALL 32
- #define REGULAREXPRESSION -1
- #define REDEFINEDEXPRESSION -2
- #define NEWLYDEFINEDEXPRESSION -3

Defines: function specs

Function specifications.

- #define GENERALFUNCTION 0
- #define FASTFUNCTION 1
- #define TENSORFUNCTION 2
- #define GAMMAFUNCTION 4
- #define POS_ 0 /* integer > 0 */
- #define POS0_ 1 /* integer >= 0 */
- #define NEG_ 2 /* integer < 0 */
- #define NEG0_ 3 /* integer <= 0 */
- #define EVEN_ 4 /* integer (even) */
- #define ODD_ 5 /* integer (odd) */
- #define Z_ 6 /* integer */
- #define SYMBOL_ 7 /* symbol only */
- #define FIXED_ 8 /* fixed index */
- #define INDEX_ 9 /* index only */
- #define Q_ 10 /* rational */

- #define DUMMYINDEX_ 11 /∗ dummy index only ∗/
- #define VECTOR_ 12 /∗ vector only ∗/
- #define GAMMA1 0
- #define GAMMA5 -1
- #define GAMMA6 -2
- #define GAMMA7 -3
- #define FUNNYVEC -4
- #define FUNNYWILD -5
- #define SUMMEDIND -6
- #define NOINDEX -7
- #define FUNNYDOLLAR -8
- #define EMPTYINDEX -9
- #define MINSPEC -10
- #define USEDFLAG 2
- #define DUMMYFLAG 1
- #define MAINSORT 0
- #define FUNCTIONSORT 1
- #define SUBSORT 2
- #define FLOATMODE 1
- #define RATIONALMODE 0
- #define NUMSPECSETS 10
- #define EATTENSOR 0x2000
- #define ISZERO 1
- #define ISUNMODIFIED 2
- #define ISCOMPRESSED 4
- #define ISINRHS 8
- #define ISFACTORIZED 16
- #define TOBEFACTORED 32
- #define TOBEUNFACTORED 64
- #define KEEPZERO 128
- #define VARTYPENONE 0
- #define VARTYPECOMPLEX 1
- #define VARTYPEIMAGINARY 2
- #define VARTYPEROOTOFUNITY 4
- #define VARTYPEMINUS 8
- #define CYCLESYMMETRIC 1
- #define RCYCLESYMMETRIC 2
- #define SYMMETRIC 3
- #define ANTISYMMETRIC 4
- #define REVERSEORDER 256
- #define SUBMULTI 1
- #define SUBONCE 2
- #define SUBONLY 3
- #define SUBMANY 4
- #define SUBVECTOR 5
- #define SUBSELECT 6
- #define SUBALL 7
- #define SUBMASK 15
- #define SUBDISORDER 16
- #define SUBAFTER 32
- #define SUBAFTERNOT 64
- #define IDHEAD 6
- #define DOLLARFLAG 1
- #define NORMALIZEFLAG 2
- #define GIDENT 1
- #define GFIVE 4
- #define GPLUS 3
- #define GMINUS 2
- #define LONGNUMBER 1
- #define MATCH 2
- #define COEFFI 3
- #define SUBEXPR 4
- #define MULTIPLEOF 5
- #define IFDOLLAR 6

- #define IFEXPRESSION 7
- #define IFDOLLAREXTRA 8
- #define IFISFACTORIZED 9
- #define IFOCCURS 10
- #define GREATER 0
- #define GREATEREQUAL 1
- #define LESS 2
- #define LESSEQUAL 3
- #define EQUAL 4
- #define NOTEQUAL 5
- #define ORCOND 6
- #define ANDCOND 7
- #define DUMMY 1
- #define SORT 1
- #define STORE 2
- #define END 3
- #define GLOBAL 4
- #define CLEAR 5
- #define VECTBIT 1
- #define DOTPBIT 2
- #define FUNBIT 4
- #define SETBIT 8
- #define EXTRAPARAMETER 0x4000
- #define GENCOMMUTE 0
- #define GENNONCOMMUTE 0x2000
- #define NAMENOTFOUND -9
- #define DOLUNDEFINED 0
- #define DOLNUMBER 1
- #define DOLARGUMENT 2
- #define DOLSUBTERM 3
- #define DOLTERMS 4
- #define DOLWILDARGS 5
- #define DOLINDEX 6
- #define DOLZERO 7
- #define FINDLOOP 0
- #define REPLACELOOP 1
- #define NOFUNPOWERS 0
- #define COMFUNPOWERS 1
- #define ALLFUNPOWERS 2
- #define PROPERORDERFLAG 0
- #define REGULAR 0
- #define FINISH 1
- #define POLYADD 1
- #define POLYSUB 2
- #define POLYMUL 3
- #define POLYDIV 4
- #define POLYREM 5
- #define POLYGCD 6
- #define POLYINTFAC 7
- #define POLYNORM 8
- #define MODNONE 0
- #define MODSUM 1
- #define MODMAX 2
- #define MODMIN 3
- #define MODLOCAL 4
- #define ELEMENTUSED 1
- #define ELEMENTLOADED 2
- #define POSNEG 0x1
- #define INVERSETABLE 0x2
- #define COEFFICIENTSONLY 0x4
- #define ALSOPOWERS 0x8
- #define ALSOFUNARGS 0x10
- #define ALSODOLLARS 0x20
- #define NOINVERSES 0x40

- #define POSITIVEONLY 0
- #define UNPACK 0x80
- #define NOUNPACK 0
- #define FROMFUNCTION 0x100
- #define VARNAMES 0
- #define AUTONAMES 1
- #define EXPRNAMES 2
- #define DOLLARNAMES 3
- #define DUMMYBUFFER 1
- #define ALLARGS 1
- #define NUMARG 2
- #define ARGRANGE 3
- #define MAKEARGS 4
- #define MAXRANGEINDICATOR 4
- #define REPLACEARG 5
- #define ENCODEARG 6
- #define DECODEARG 7
- #define IMPLODEARG 8
- #define EXPLODEARG 9
- #define PERMUTEARG 10
- #define REVERSEARG 11
- #define CYCLEARG 12
- #define ISLYNDON 13
- #define ISLYNDONR 14
- #define TOLYNDON 15
- #define TOLYNDONR 16
- #define ADDARG 17
- #define MULTIPLYARG 18
- #define DROPARG 19
- #define SELECTARG 20
- #define DEDUPARG 21
- #define BASECODE 1
- #define YESLYNDON 1
- #define NOLYNDON 2
- #define TOPOLYNOMIALFLAG 1
- #define FACTARGFLAG 2
- #define OLDFACTARG 1
- #define NEWFACTARG 0
- #define FROMMODULEOPTION 0
- #define FROMPOINTINSTRUCTION 1
- #define EXTRASYMBOL 0
- #define REGULARSYMBOL 1
- #define EXPRESSIONNUMBER 2
- #define O_NONE 0
- #define O_CSE 1
- #define O_CSEGREEDY 2
- #define O_GREEDY 3
- #define O_OCCURRENCE 0
- #define O_MCTS 1
- #define O_SIMULATED_ANNEALING 2
- #define O_FORWARD 0
- #define O_BACKWARD 1
- #define O_FORWARDORBACKWARD 2
- #define O_FORWARDANDBACKWARD 3
- #define OPTHEAD 3
- #define DOALL 1
- #define ONLYFUNCTIONS 2
- #define INUSE 1
- #define COULDCOMMUTE 2
- #define DOESNOTCOMMUTE 4
- #define DICT_NONUMBERS 0
- #define DICT_INTEGERONLY 1
- #define DICT_RATIONALONLY 2
- #define DICT_ALLNUMBERS 3

- #define DICT_NOVARIABLES 0
- #define DICT_DOVARIABLES 1
- #define DICT_NOSPECIALS 0
- #define DICT_DOSPECIALS 1
- #define DICT_NOFUNWITHARGS 0
- #define DICT_DOFUNWITHARGS 1
- #define DICT_NOTINDOLLARS 0
- #define DICT_INDOLLARS 1
- #define DICT_INTEGERNUMBER 1
- #define DICT_RATIONALNUMBER 2
- #define DICT_SYMBOL 3
- #define DICT_VECTOR 4
- #define DICT_INDEX 5
- #define DICT_FUNCTION 6
- #define DICT_FUNCTION_WITH_ARGUMENTS 7
- #define DICT_SPECIALCHARACTER 8
- #define DICT_RANGE 9
- #define READSPECTATORFLAG 3
- #define GLOBALSPECTATORFLAG 1

Typedefs

- typedef VOID(∗ PVFUNWP) ()
- typedef VOID(∗ PVFUNV) ()
- typedef int(∗ CFUN) ()
- typedef int(∗ TFUN) ()
- typedef int(∗ TFUN1) ()

4.37.1 Detailed Description

Contains the definitions of many internal codes Rather than using numbers directly we do this by defines, making it much easier to change things. Changing things is sometimes also a good way of testing the code.

Definition in file ftypes.h.

4.37.2 Macro Definition Documentation

4.37.2.1 PHEAD

```
#define PHEAD
```

The next macros were introduced when TFORM was programmed. In the case of workers, each worker may need some private data. These can in principle be accessed by some posix calls but that is unnecessarily slow. The passing of a pointer to the complete data struct with private data will be much faster. And anyway, there would have to be a macro that either makes the posix call (TFORM) or doesn't (FORM). The solution by having macro's that either pass the pointer (TFORM) or don't pass it (FORM) is seen as the best solution.

In the declarations and the calling of the functions we have to use the PHEAD or the BHEAD macro, respectively, if the pointer is to be passed. These macro's contain the comma as well. Hence we need special macro's if there are no other arguments. These are called PHEAD0 and BHEAD0.

Definition at line 56 of file ftypes.h.

4.37.2.2 PHEAD0

```
#define PHEAD0 VOID
```

Definition at line 57 of file ftypes.h.

4.37.2.3 BHEAD

```
#define BHEAD
```

Definition at line 58 of file ftypes.h.

4.37.2.4 BHEAD0

```
#define BHEAD0
```

Definition at line 59 of file ftypes.h.

4.37.2.5 WITHOUTERROR

```
#define WITHOUTERROR 0
```

Definition at line 62 of file ftypes.h.

4.37.2.6 WITHERROR

```
#define WITHERROR 1
```

Definition at line 63 of file ftypes.h.

4.37.2.7 FILESTREAM

```
#define FILESTREAM 0
```

Definition at line 69 of file ftypes.h.

4.37.2.8 PREVARSTREAM

```
#define PREVARSTREAM 1
```

Definition at line 70 of file ftypes.h.

4.37.2.9 PREREADSTREAM

```
#define PREREADSTREAM 2
```

Definition at line 71 of file ftypes.h.

4.37.2.10 PIPESTREAM

```
#define PIPESTREAM 3
```

Definition at line 72 of file ftypes.h.

4.37.2.11 PRECALCSTREAM

```
#define PRECALCSTREAM 4
```

Definition at line 73 of file ftypes.h.

4.37.2.12 DOLLARSTREAM

```
#define DOLLARSTREAM 5
```

Definition at line 74 of file ftypes.h.

4.37.2.13 PREREADSTREAM2

```
#define PREREADSTREAM2 6
```

Definition at line 75 of file ftypes.h.

4.37.2.14 EXTERNALCHANNELSTREAM

```
#define EXTERNALCHANNELSTREAM 7
```

Definition at line 76 of file ftypes.h.

4.37.2.15 PREREADSTREAM3

```
#define PREREADSTREAM3 8
```

Definition at line 77 of file ftypes.h.

4.37.2.16 REVERSEFILESTREAM

```
#define REVERSEFILESTREAM 9
```

Definition at line 78 of file ftypes.h.

4.37.2.17 ENDOFSTREAM

```
#define ENDOFSTREAM 0xFF
```

Definition at line 80 of file ftypes.h.

4.37.2.18 ENDOFINPUT

```
#define ENDOFINPUT 0xFF
```

Definition at line 81 of file ftypes.h.

4.37.2.19 SUBROUTINEFILE

```
#define SUBROUTINEFILE 0
```

Definition at line 87 of file ftypes.h.

4.37.2.20 PROCEDUREFILE

```
#define PROCEDUREFILE 1
```

Definition at line 88 of file ftypes.h.

4.37.2.21 HEADERFILE

```
#define HEADERFILE 2
```

Definition at line 89 of file ftypes.h.

4.37.2.22 SETUPFILE

```
#define SETUPFILE 3
```

Definition at line 90 of file ftypes.h.

4.37.2.23 TABLEBASEFILE

```
#define TABLEBASEFILE 4
```

Definition at line 91 of file ftypes.h.

4.37.2.24 FIRSTMODULE

```
#define FIRSTMODULE -1
```

Definition at line 97 of file ftypes.h.

4.37.2.25 GLOBALMODULE

```
#define GLOBALMODULE 0
```

Definition at line 98 of file ftypes.h.

4.37.2.26 SORTMODULE

```
#define SORTMODULE 1
```

Definition at line 99 of file ftypes.h.

4.37.2.27 STOREMODULE

```
#define STOREMODULE 2
```

Definition at line 100 of file ftypes.h.

4.37.2.28 CLEARMODULE

```
#define CLEARMODULE 3
```

Definition at line 101 of file ftypes.h.

4.37.2.29 ENDMODULE

```
#define ENDMODULE 4
```

Definition at line 102 of file ftypes.h.

4.37.2.30 POLYFUN

```
#define POLYFUN 0
```

Definition at line 104 of file ftypes.h.

4.37.2.31 NOPARALLEL_DOLLAR

```
#define NOPARALLEL_DOLLAR 0x0001
```

Definition at line 106 of file ftypes.h.

4.37.2.32 NOPARALLEL_RHS

```
#define NOPARALLEL_RHS 0x0002
```

Definition at line 107 of file ftypes.h.

4.37.2.33 NOPARALLEL_CONVPOLY

```
#define NOPARALLEL_CONVPOLY 0x0004
```

Definition at line 108 of file ftypes.h.

4.37.2.34 NOPARALLEL_SPECTATOR

```
#define NOPARALLEL_SPECTATOR 0x0008
```

Definition at line 109 of file ftypes.h.

4.37.2.35 NOPARALLEL_USER

```
#define NOPARALLEL_USER 0x0010
```

Definition at line 110 of file ftypes.h.

4.37.2.36 NOPARALLEL_TBLDOLLAR

```
#define NOPARALLEL_TBLDOLLAR 0x0100
```

Definition at line 111 of file ftypes.h.

4.37.2.37 NOPARALLEL_NPROC

```
#define NOPARALLEL_NPROC 0x0200
```

Definition at line 112 of file ftypes.h.

4.37.2.38 PARALLELFLAG

```
#define PARALLELFLAG 0x0000
```

Definition at line 113 of file ftypes.h.

4.37.2.39 PRENOACTION

```
#define PRENOACTION 0
```

Definition at line 115 of file ftypes.h.

4.37.2.40 PRERAISEAFTER

```
#define PRERAISEAFTER 1
```

Definition at line 116 of file ftypes.h.

4.37.2.41 PRELOWERAFTER

```
#define PRELOWERAFTER 2
```

Definition at line 117 of file ftypes.h.

4.37.2.42 WITHSEMICOLON

```
#define WITHSEMICOLON 0
```

Definition at line 124 of file ftypes.h.

4.37.2.43 WITHOUTSEMICOLON

```
#define WITHOUTSEMICOLON 1
```

Definition at line 125 of file ftypes.h.

4.37.2.44 MODULEINSTACK

```
#define MODULEINSTACK 8
```

Definition at line 126 of file ftypes.h.

4.37.2.45 EXECUTINGIF

```
#define EXECUTINGIF 0
```

Definition at line 127 of file ftypes.h.

4.37.2.46 LOOKINGFORELSE

```
#define LOOKINGFORELSE 1
```

Definition at line 128 of file ftypes.h.

4.37.2.47 LOOKINGFORENDIF

```
#define LOOKINGFORENDIF 2
```

Definition at line 129 of file ftypes.h.

4.37.2.48 NEWSTATEMENT

```
#define NEWSTATEMENT 1
```

Definition at line 130 of file ftypes.h.

4.37.2.49 OLDSTATEMENT

```
#define OLDSTATEMENT 0
```

Definition at line 131 of file ftypes.h.

4.37.2.50 EXECUTINGPRESWITCH

```
#define EXECUTINGPRESWITCH 0
```

Definition at line 133 of file ftypes.h.

4.37.2.51 SEARCHINGPRECASE

```
#define SEARCHINGPRECASE 1
```

Definition at line 134 of file ftypes.h.

4.37.2.52 SEARCHINGPREENDSWITCH

```
#define SEARCHINGPREENDSWITCH 2
```

Definition at line 135 of file ftypes.h.

4.37.2.53 PREPROONLY

```
#define PREPROONLY 1
```

Definition at line 137 of file ftypes.h.

4.37.2.54 DUMPTOCOMPILER

```
#define DUMPTOCOMPILER 2
```

Definition at line 138 of file ftypes.h.

4.37.2.55 DUMPOUTTERMS

```
#define DUMPOUTTERMS 4
```

Definition at line 139 of file ftypes.h.

4.37.2.56 DUMPINTERMS

```
#define DUMPINTERMS 8
```

Definition at line 140 of file ftypes.h.

4.37.2.57 DUMPTOSORT

```
#define DUMPTOSORT 16
```

Definition at line 141 of file ftypes.h.

4.37.2.58 DUMPTOPARALLEL

```
#define DUMPTOPARALLEL 32
```

Definition at line 142 of file ftypes.h.

4.37.2.59 THREADSDEBUG

```
#define THREADSDEBUG 64
```

Definition at line 143 of file ftypes.h.

4.37.2.60 ERROROUT

```
#define ERROROUT 0
```

Definition at line 145 of file ftypes.h.

4.37.2.61 INPUTOUT

```
#define INPUTOUT 1
```

Definition at line 146 of file ftypes.h.

4.37.2.62 STATSOUT

```
#define STATSOUT 2
```

Definition at line 147 of file ftypes.h.

4.37.2.63 EXPRSOUT

```
#define EXPRSOUT 3
```

Definition at line 148 of file ftypes.h.

4.37.2.64 WRITEOUT

```
#define WRITEOUT 4
```

Definition at line 149 of file ftypes.h.

4.37.2.65 EXTERNALCHANNELOUT

```
#define EXTERNALCHANNELOUT 5
```

Definition at line 151 of file ftypes.h.

4.37.2.66 NUMERICALLOOP

```
#define NUMERICALLOOP 0
```

Definition at line 153 of file ftypes.h.

4.37.2.67 LISTEDLOOP

```
#define LISTEDLOOP 1
```

Definition at line 154 of file ftypes.h.

4.37.2.68 ONEEXPRESSION

```
#define ONEEXPRESSION 2
```

Definition at line 155 of file ftypes.h.

4.37.2.69 PRETYPENONE

```
#define PRETYPENONE 0
```

Definition at line 157 of file ftypes.h.

4.37.2.70 PRETYPEIF

```
#define PRETYPEIF 1
```

Definition at line 158 of file ftypes.h.

4.37.2.71 PRETYPEDO

```
#define PRETYPEDO 2
```

Definition at line 159 of file ftypes.h.

4.37.2.72 PRETYPEPROCEDURE

```
#define PRETYPEPROCEDURE 3
```

Definition at line 160 of file ftypes.h.

4.37.2.73 PRETYPESWITCH

```
#define PRETYPESWITCH 4
```

Definition at line 161 of file ftypes.h.

4.37.2.74 PRETYPEINSIDE

```
#define PRETYPEINSIDE 5
```

Definition at line 162 of file ftypes.h.

4.37.2.75 DECLARATION

```
#define DECLARATION 1
```

Definition at line 168 of file ftypes.h.

4.37.2.76 SPECIFICATION

```
#define SPECIFICATION 2
```

Definition at line 169 of file ftypes.h.

4.37.2.77 DEFINITION

```
#define DEFINITION 3
```

Definition at line 170 of file ftypes.h.

4.37.2.78 STATEMENT

```
#define STATEMENT 4
```

Definition at line 171 of file ftypes.h.

4.37.2.79 TOOUTPUT

```
#define TOOUTPUT 5
```

Definition at line 172 of file ftypes.h.

4.37.2.80 ATENDOFMODULE

```
#define ATENDOFMODULE 6
```

Definition at line 173 of file ftypes.h.

4.37.2.81 MIXED

```
#define MIXED 9
```

Definition at line 174 of file ftypes.h.

4.37.2.82 VOID

```
#define VOID void
```

Definition at line 184 of file ftypes.h.

4.37.2.83 NOAUTO

```
#define NOAUTO 0
```

Definition at line 206 of file ftypes.h.

4.37.2.84 PARTEST

```
#define PARTEST 1
```

Definition at line 207 of file ftypes.h.

4.37.2.85 WITHAUTO

```
#define WITHAUTO 2
```

Definition at line 208 of file ftypes.h.

4.37.2.86 ALLVARIABLES

```
#define ALLVARIABLES -1
```

Definition at line 210 of file ftypes.h.

4.37.2.87 SYMBOLONLY

```
#define SYMBOLONLY 1
```

Definition at line 211 of file ftypes.h.

4.37.2.88 INDEXONLY

```
#define INDEXONLY 2
```

Definition at line 212 of file ftypes.h.

4.37.2.89 VECTORONLY

```
#define VECTORONLY 4
```

Definition at line 213 of file ftypes.h.

4.37.2.90 FUNCTIONONLY

```
#define FUNCTIONONLY 8
```

Definition at line 214 of file ftypes.h.

4.37.2.91 SETONLY

```
#define SETONLY 16
```

Definition at line 215 of file ftypes.h.

4.37.2.92 EXPRESSIONONLY

```
#define EXPRESSIONONLY 32
```

Definition at line 216 of file ftypes.h.

4.37.2.93 CDELETE

```
#define CDELETE -1
```

Definition at line 227 of file ftypes.h.

4.37.2.94 ANYTYPE

```
#define ANYTYPE -1
```

Definition at line 228 of file ftypes.h.

4.37.2.95 CSYMBOL

```
#define CSYMBOL 0
```

Definition at line 229 of file ftypes.h.

4.37.2.96 CINDEX

```
#define CINDEX 1
```

Definition at line 230 of file ftypes.h.

4.37.2.97 CVECTOR

```
#define CVECTOR 2
```

Definition at line 231 of file ftypes.h.

4.37.2.98 CFUNCTION

```
#define CFUNCTION 3
```

Definition at line 232 of file ftypes.h.

4.37.2.99 CSET

```
#define CSET 4
```

Definition at line 233 of file ftypes.h.

4.37.2.100 CEXPRESSION

```
#define CEXPRESSION 5
```

Definition at line 234 of file ftypes.h.

4.37.2.101 CDOTPRODUCT

```
#define CDOTPRODUCT 6
```

Definition at line 235 of file ftypes.h.

4.37.2.102 CNUMBER

```
#define CNUMBER 7
```

Definition at line 236 of file ftypes.h.

4.37.2.103 CSUBEXP

```
#define CSUBEXP 8
```

Definition at line 237 of file ftypes.h.

4.37.2.104 CDELTA

```
#define CDELTA 9
```

Definition at line 238 of file ftypes.h.

4.37.2.105 CDOLLAR

```
#define CDOLLAR 10
```

Definition at line 239 of file ftypes.h.

4.37.2.106 CDUBIOUS

```
#define CDUBIOUS 11
```

Definition at line 240 of file ftypes.h.

4.37.2.107 CRANGE

```
#define CRANGE 12
```

Definition at line 241 of file ftypes.h.

4.37.2.108 CVECTOR1

```
#define CVECTOR1 21
```

Definition at line 242 of file ftypes.h.

4.37.2.109 CDOUBLEDOT

```
#define CDOUBLEDOT 22
```

Definition at line 243 of file ftypes.h.

4.37.2.110 TSYMBOL

```
#define TSYMBOL -1
```

Definition at line 251 of file ftypes.h.

4.37.2.111 TINDEX

```
#define TINDEX -2
```

Definition at line 252 of file ftypes.h.

4.37.2.112 TVECTOR

```
#define TVECTOR -3
```

Definition at line 253 of file ftypes.h.

4.37.2.113 TFUNCTION

```
#define TFUNCTION -4
```

Definition at line 254 of file ftypes.h.

4.37.2.114 TSET

```
#define TSET -5
```

Definition at line 255 of file ftypes.h.

4.37.2.115 TEXPRESSION

```
#define TEXPRESSION -6
```

Definition at line 256 of file ftypes.h.

4.37.2.116 TDOTPRODUCT

```
#define TDOTPRODUCT -7
```

Definition at line 257 of file ftypes.h.

4.37.2.117 TNUMBER

```
#define TNUMBER -8
```

Definition at line 258 of file ftypes.h.

4.37.2.118 TSUBEXP

```
#define TSUBEXP -9
```

Definition at line 259 of file ftypes.h.

4.37.2.119 TDELTA

```
#define TDELTA -10
```

Definition at line 260 of file ftypes.h.

4.37.2.120 TDOLLAR

```
#define TDOLLAR -11
```

Definition at line 261 of file ftypes.h.

4.37.2.121 TDUBIOUS

```
#define TDUBIOUS -12
```

Definition at line 262 of file ftypes.h.

4.37.2.122 LPARENTHESIS

```
#define LPARENTHESIS -13
```

Definition at line 263 of file ftypes.h.

4.37.2.123 RPARENTHESIS

```
#define RPARENTHESIS -14
```

Definition at line 264 of file ftypes.h.

4.37.2.124 TWILDCARD

```
#define TWILDCARD -15
```

Definition at line 265 of file ftypes.h.

4.37.2.125 TWILDARG

```
#define TWILDARG -16
```

Definition at line 266 of file ftypes.h.

4.37.2.126 TDOT

```
#define TDOT -17
```

Definition at line 267 of file ftypes.h.

4.37.2.127 LBRACE

```
#define LBRACE -18
```

Definition at line 268 of file ftypes.h.

4.37.2.128 RBRACE

```
#define RBRACE -19
```

Definition at line 269 of file ftypes.h.

4.37.2.129 TCOMMA

```
#define TCOMMA -20
```

Definition at line 270 of file ftypes.h.

4.37.2.130 TFUNOPEN

```
#define TFUNOPEN -21
```

Definition at line 271 of file ftypes.h.

4.37.2.131 TFUNCLOSE

```
#define TFUNCLOSE -22
```

Definition at line 272 of file ftypes.h.

4.37.2.132 TMULTIPLY

```
#define TMULTIPLY -23
```

Definition at line 273 of file ftypes.h.

4.37.2.133 TDIVIDE

```
#define TDIVIDE -24
```

Definition at line 274 of file ftypes.h.

4.37.2.134 TPOWER

```
#define TPOWER -25
```

Definition at line 275 of file ftypes.h.

4.37.2.135 TPLUS

```
#define TPLUS -26
```

Definition at line 276 of file ftypes.h.

4.37.2.136 TMINUS

```
#define TMINUS -27
```

Definition at line 277 of file ftypes.h.

4.37.2.137 TNOT

```
#define TNOT -28
```

Definition at line 278 of file ftypes.h.

4.37.2.138 TENDOFIT

```
#define TENDOFIT -29
```

Definition at line 279 of file ftypes.h.

4.37.2.139 TSETOPEN

```
#define TSETOPEN -30
```

Definition at line 280 of file ftypes.h.

4.37.2.140 TSETCLOSE

```
#define TSETCLOSE -31
```

Definition at line 281 of file ftypes.h.

4.37.2.141 TGENINDEX

```
#define TGENINDEX -32
```

Definition at line 282 of file ftypes.h.

4.37.2.142 TCONJUGATE

```
#define TCONJUGATE -33
```

Definition at line 283 of file ftypes.h.

4.37.2.143 LRPARENTHESES

```
#define LRPARENTHESES -34
```

Definition at line 284 of file ftypes.h.

4.37.2.144 TNUMBER1

```
#define TNUMBER1 -35
```

Definition at line 285 of file ftypes.h.

4.37.2.145 TPOWER1

```
#define TPOWER1 -36
```

Definition at line 286 of file ftypes.h.

4.37.2.146 TEMPTY

```
#define TEMPTY -37
```

Definition at line 287 of file ftypes.h.

4.37.2.147 TSETNUM

```
#define TSETNUM -38
```

Definition at line 288 of file ftypes.h.

4.37.2.148 TSGAMMA

```
#define TSGAMMA -39
```

Definition at line 289 of file ftypes.h.

4.37.2.149 TSETDOL

```
#define TSETDOL -40
```

Definition at line 290 of file ftypes.h.

4.37.2.150 TYPEISFUN

```
#define TYPEISFUN 0
```

Definition at line 292 of file ftypes.h.

4.37.2.151 TYPEISSUB

```
#define TYPEISSUB 1
```

Definition at line 293 of file ftypes.h.

4.37.2.152 TYPEISMYSTERY

```
#define TYPEISMYSTERY -1
```

Definition at line 294 of file ftypes.h.

4.37.2.153 LHSIDEX

```
#define LHSIDEX 2
```

Definition at line 296 of file ftypes.h.

4.37.2.154 LHSIDE

```
#define LHSIDE 1
```

Definition at line 297 of file ftypes.h.

4.37.2.155 RHSIDE

```
#define RHSIDE 0
```

Definition at line 298 of file ftypes.h.

4.37.2.156 FORTRANMODE

```
#define FORTRANMODE 1
```

Definition at line 304 of file ftypes.h.

4.37.2.157 REDUCEMODE

```
#define REDUCEMODE 2
```

Definition at line 305 of file ftypes.h.

4.37.2.158 MAPLEMODE

```
#define MAPLEMODE 3
```

Definition at line 306 of file ftypes.h.

4.37.2.159 MATHEMATICAMODE

```
#define MATHEMATICAMODE 4
```

Definition at line 307 of file ftypes.h.

4.37.2.160 CMODE

```
#define CMODE 5
```

Definition at line 308 of file ftypes.h.

4.37.2.161 VORTRANMODE

```
#define VORTRANMODE 6
```

Definition at line 309 of file ftypes.h.

4.37.2.162 PFORTRANMODE

```
#define PFORTRANMODE 7
```

Definition at line 310 of file ftypes.h.

4.37.2.163 DOUBLEFORTRANMODE

```
#define DOUBLEFORTRANMODE 33
```

Definition at line 311 of file ftypes.h.

4.37.2.164 DOUBLEPRECISIONFLAG

```
#define DOUBLEPRECISIONFLAG 32
```

Definition at line 312 of file ftypes.h.

4.37.2.165 NODOUBLEMASK

```
#define NODOUBLEMASK 31
```

Definition at line 313 of file ftypes.h.

4.37.2.166 QUADRUPLEFORTRANMODE

```
#define QUADRUPLEFORTRANMODE 65
```

Definition at line 314 of file ftypes.h.

4.37.2.167 QUADRUPLEPRECISIONFLAG

```
#define QUADRUPLEPRECISIONFLAG 64
```

Definition at line 315 of file ftypes.h.

4.37.2.168 NOQUADMASK

```
#define NOQUADMASK 63
```

Definition at line 316 of file ftypes.h.

4.37.2.169 NORMALFORMAT

```
#define NORMALFORMAT 0
```

Definition at line 317 of file ftypes.h.

4.37.2.170 NOSPACEFORMAT

```
#define NOSPACEFORMAT 1
```

Definition at line 318 of file ftypes.h.

4.37.2.171 ISNOTFORTRAN90

```
#define ISNOTFORTRAN90 0
```

Definition at line 320 of file ftypes.h.

4.37.2.172 ISFORTRAN90

```
#define ISFORTRAN90 1
```

Definition at line 321 of file ftypes.h.

4.37.2.173 ALSOREVERSE

```
#define ALSOREVERSE 1
```

Definition at line 323 of file ftypes.h.

4.37.2.174 CHISHOLM

```
#define CHISHOLM 2
```

Definition at line 324 of file ftypes.h.

4.37.2.175 NOTRICK

```
#define NOTRICK 16
```

Definition at line 325 of file ftypes.h.

4.37.2.176 SORTLOWFIRST

```
#define SORTLOWFIRST 0
```

Definition at line 327 of file ftypes.h.

4.37.2.177 SORTHIGHFIRST

```
#define SORTHIGHFIRST 1
```

Definition at line 328 of file ftypes.h.

4.37.2.178 SORTPOWERFIRST

```
#define SORTPOWERFIRST 2
```

Definition at line 329 of file ftypes.h.

4.37.2.179 SORTANTIPOWER

```
#define SORTANTIPOWER 3
```

Definition at line 330 of file ftypes.h.

4.37.2.180 NMIN4SHIFT

```
#define NMIN4SHIFT 4
```

Definition at line 332 of file ftypes.h.

4.37.2.181 SYMBOL

```
#define SYMBOL 1
```

Definition at line 346 of file ftypes.h.

4.37.2.182 DOTPRODUCT

```
#define DOTPRODUCT 2
```

Definition at line 347 of file ftypes.h.

4.37.2.183 VECTOR

```
#define VECTOR 3
```

Definition at line 348 of file ftypes.h.

4.37.2.184 INDEX

```
#define INDEX 4
```

Definition at line 349 of file ftypes.h.

4.37.2.185 EXPRESSION

```
#define EXPRESSION 5
```

Definition at line 350 of file ftypes.h.

4.37.2.186 SUBEXPRESSION

```
#define SUBEXPRESSION 6
```

Definition at line 351 of file ftypes.h.

4.37.2.187 DOLLAREXPRESSION

```
#define DOLLAREXPRESSION 7
```

Definition at line 352 of file ftypes.h.

4.37.2.188 SETSET

```
#define SETSET 8
```

Definition at line 353 of file ftypes.h.

4.37.2.189 ARGWILD

```
#define ARGWILD 9
```

Definition at line 354 of file ftypes.h.

4.37.2.190 MINVECTOR

```
#define MINVECTOR 10
```

Definition at line 355 of file ftypes.h.

4.37.2.191 SETEXP

```
#define SETEXP 11
```

Definition at line 356 of file ftypes.h.

4.37.2.192 DOLLAREXPR2

```
#define DOLLAREXPR2 12
```

Definition at line 357 of file ftypes.h.

4.37.2.193 HAAKJE0

```
#define HAAKJE0 9
```

Definition at line 358 of file ftypes.h.

4.37.2.194 FUNCTION

```
#define FUNCTION 20
```

Definition at line 359 of file ftypes.h.

4.37.2.195 TMPPOLYFUN

```
#define TMPPOLYFUN 14
```

Definition at line 361 of file ftypes.h.

4.37.2.196 ARGFIELD

```
#define ARGFIELD 15
```

Definition at line 362 of file ftypes.h.

4.37.2.197 SNUMBER

```
#define SNUMBER 16
```

Definition at line 363 of file ftypes.h.

4.37.2.198 LNUMBER

```
#define LNUMBER 17
```

Definition at line 364 of file ftypes.h.

4.37.2.199 HAAKJE

```
#define HAAKJE 18
```

Definition at line 365 of file ftypes.h.

4.37.2.200 DELTA

```
#define DELTA 19
```

Definition at line 366 of file ftypes.h.

4.37.2.201 EXPONENT

```
#define EXPONENT 20
```

Definition at line 367 of file ftypes.h.

4.37.2.202 DENOMINATOR

```
#define DENOMINATOR 21
```

Definition at line 368 of file ftypes.h.

4.37.2.203 SETFUNCTION

```
#define SETFUNCTION 22
```

Definition at line 369 of file ftypes.h.

4.37.2.204 GAMMA

```
#define GAMMA 23
```

Definition at line 370 of file ftypes.h.

4.37.2.205 GAMMAI

```
#define GAMMAI 24
```

Definition at line 371 of file ftypes.h.

4.37.2.206 GAMMAFIVE

```
#define GAMMAFIVE 25
```

Definition at line 372 of file ftypes.h.

4.37.2.207 GAMMASIX

```
#define GAMMASIX 26
```

Definition at line 373 of file ftypes.h.

4.37.2.208 GAMMASEVEN

```
#define GAMMASEVEN 27
```

Definition at line 374 of file ftypes.h.

4.37.2.209 SUMF1

```
#define SUMF1 28
```

Definition at line 375 of file ftypes.h.

4.37.2.210 SUMF2

```
#define SUMF2 29
```

Definition at line 376 of file ftypes.h.

4.37.2.211 DUMFUN

```
#define DUMFUN 30
```

Definition at line 377 of file ftypes.h.

4.37.2.212 REPLACEMENT

```
#define REPLACEMENT 31
```

Definition at line 378 of file ftypes.h.

4.37.2.213 REVERSEFUNCTION

```
#define REVERSEFUNCTION 32
```

Definition at line 379 of file ftypes.h.

4.37.2.214 DISTRIBUTION

```
#define DISTRIBUTION 33
```

Definition at line 380 of file ftypes.h.

4.37.2.215 DELTA3

```
#define DELTA3 34
```

Definition at line 381 of file ftypes.h.

4.37.2.216 DUMMYFUN

```
#define DUMMYFUN 35
```

Definition at line 382 of file ftypes.h.

4.37.2.217 DUMMYTEN

```
#define DUMMYTEN 36
```

Definition at line 383 of file ftypes.h.

4.37.2.218 LEVICIVITA

```
#define LEVICIVITA 37
```

Definition at line 384 of file ftypes.h.

4.37.2.219 FACTORIAL

```
#define FACTORIAL 38
```

Definition at line 385 of file ftypes.h.

4.37.2.220 INVERSEFACTORIAL

```
#define INVERSEFACTORIAL 39
```

Definition at line 386 of file ftypes.h.

4.37.2.221 BINOMIAL

```
#define BINOMIAL 40
```

Definition at line 387 of file ftypes.h.

4.37.2.222 NUMARGSFUN

```
#define NUMARGSFUN 41
```

Definition at line 388 of file ftypes.h.

4.37.2.223 SIGNFUN

```
#define SIGNFUN 42
```

Definition at line 389 of file ftypes.h.

4.37.2.224 MODFUNCTION

```
#define MODFUNCTION 43
```

Definition at line 390 of file ftypes.h.

4.37.2.225 MOD2FUNCTION

```
#define MOD2FUNCTION 44
```

Definition at line 391 of file ftypes.h.

4.37.2.226 MINFUNCTION

```
#define MINFUNCTION 45
```

Definition at line 392 of file ftypes.h.

4.37.2.227 MAXFUNCTION

```
#define MAXFUNCTION 46
```

Definition at line 393 of file ftypes.h.

4.37.2.228 ABSFUNCTION

```
#define ABSFUNCTION 47
```

Definition at line 394 of file ftypes.h.

4.37.2.229 SIGFUNCTION

```
#define SIGFUNCTION 48
```

Definition at line 395 of file ftypes.h.

4.37.2.230 INTFUNCTION

```
#define INTFUNCTION 49
```

Definition at line 396 of file ftypes.h.

4.37.2.231 THETA

```
#define THETA 50
```

Definition at line 397 of file ftypes.h.

4.37.2.232 THETA2

```
#define THETA2 51
```

Definition at line 398 of file ftypes.h.

4.37.2.233 DELTA2

```
#define DELTA2 52
```

Definition at line 399 of file ftypes.h.

4.37.2.234 DELTAP

```
#define DELTAP 53
```

Definition at line 400 of file ftypes.h.

4.37.2.235 BERNOULLIFUNCTION

```
#define BERNOULLIFUNCTION 54
```

Definition at line 401 of file ftypes.h.

4.37.2.236 COUNTFUNCTION

```
#define COUNTFUNCTION 55
```

Definition at line 402 of file ftypes.h.

4.37.2.237 MATCHFUNCTION

```
#define MATCHFUNCTION 56
```

Definition at line 403 of file ftypes.h.

4.37.2.238 PATTERNFUNCTION

```
#define PATTERNFUNCTION 57
```

Definition at line 404 of file ftypes.h.

4.37.2.239 TERMFUNCTION

```
#define TERMFUNCTION 58
```

Definition at line 405 of file ftypes.h.

4.37.2.240 CONJUGATION

```
#define CONJUGATION 59
```

Definition at line 406 of file ftypes.h.

4.37.2.241 ROOTFUNCTION

```
#define ROOTFUNCTION 60
```

Definition at line 407 of file ftypes.h.

4.37.2.242 TABLEFUNCTION

```
#define TABLEFUNCTION 61
```

Definition at line 408 of file ftypes.h.

4.37.2.243 FIRSTBRACKET

```
#define FIRSTBRACKET 62
```

Definition at line 409 of file ftypes.h.

4.37.2.244 TERMSINEXPR

```
#define TERMSINEXPR 63
```

Definition at line 410 of file ftypes.h.

4.37.2.245 NUMTERMSFUN

```
#define NUMTERMSFUN 64
```

Definition at line 411 of file ftypes.h.

4.37.2.246 GCDFUNCTION

```
#define GCDFUNCTION 65
```

Definition at line 412 of file ftypes.h.

4.37.2.247 DIVFUNCTION

```
#define DIVFUNCTION 66
```

Definition at line 413 of file ftypes.h.

4.37.2.248 REMFUNCTION

```
#define REMFUNCTION 67
```

Definition at line 414 of file ftypes.h.

4.37.2.249 MAXPOWEROF

```
#define MAXPOWEROF 68
```

Definition at line 415 of file ftypes.h.

4.37.2.250 MINPOWEROF

```
#define MINPOWEROF 69
```

Definition at line 416 of file ftypes.h.

4.37.2.251 TABLESTUB

```
#define TABLESTUB 70
```

Definition at line 417 of file ftypes.h.

4.37.2.252 FACTORIN

```
#define FACTORIN 71
```

Definition at line 418 of file ftypes.h.

4.37.2.253 TERMSINBRACKET

```
#define TERMSINBRACKET 72
```

Definition at line 419 of file ftypes.h.

4.37.2.254 WILDARGFUN

```
#define WILDARGFUN 73
```

Definition at line 420 of file ftypes.h.

4.37.2.255 SQRTFUNCTION

```
#define SQRTFUNCTION 74
```

Definition at line 428 of file ftypes.h.

4.37.2.256 LNFUNCTION

```
#define LNFUNCTION 75
```

Definition at line 429 of file ftypes.h.

4.37.2.257 SINFUNCTION

```
#define SINFUNCTION 76
```

Definition at line 430 of file ftypes.h.

4.37.2.258 COSFUNCTION

```
#define COSFUNCTION 77
```

Definition at line 431 of file ftypes.h.

4.37.2.259 TANFUNCTION

```
#define TANFUNCTION 78
```

Definition at line 432 of file ftypes.h.

4.37.2.260 ASINFUNCTION

```
#define ASINFUNCTION 79
```

Definition at line 433 of file ftypes.h.

4.37.2.261 ACOSFUNCTION

```
#define ACOSFUNCTION 80
```

Definition at line 434 of file ftypes.h.

4.37.2.262 ATANFUNCTION

```
#define ATANFUNCTION 81
```

Definition at line 435 of file ftypes.h.

4.37.2.263 ATAN2FUNCTION

```
#define ATAN2FUNCTION 82
```

Definition at line 436 of file ftypes.h.

4.37.2.264 SINHFUNCTION

```
#define SINHFUNCTION 83
```

Definition at line 437 of file ftypes.h.

4.37.2.265 COSHFUNCTION

```
#define COSHFUNCTION 84
```

Definition at line 438 of file ftypes.h.

4.37.2.266 TANHFUNCTION

```
#define TANHFUNCTION 85
```

Definition at line 439 of file ftypes.h.

4.37.2.267 ASINHFUNCTION

```
#define ASINHFUNCTION 86
```

Definition at line 440 of file ftypes.h.

4.37.2.268 ACOSHFUNCTION

```
#define ACOSHFUNCTION 87
```

Definition at line 441 of file ftypes.h.

4.37.2.269 ATANHFUNCTION

```
#define ATANHFUNCTION 88
```

Definition at line 442 of file ftypes.h.

4.37.2.270 LI2FUNCTION

```
#define LI2FUNCTION 89
```

Definition at line 443 of file ftypes.h.

4.37.2.271 LINFUNCTION

```
#define LINFUNCTION 90
```

Definition at line 444 of file ftypes.h.

4.37.2.272 EXTRASYMFUN

```
#define EXTRASYMFUN 91
```

Definition at line 446 of file ftypes.h.

4.37.2.273 RANDOMFUNCTION

```
#define RANDOMFUNCTION 92
```

Definition at line 447 of file ftypes.h.

4.37.2.274 RANPERM

```
#define RANPERM 93
```

Definition at line 448 of file ftypes.h.

4.37.2.275 NUMFACTORS

```
#define NUMFACTORS 94
```

Definition at line 449 of file ftypes.h.

4.37.2.276 FIRSTTERM

```
#define FIRSTTERM 95
```

Definition at line 450 of file ftypes.h.

4.37.2.277 CONTENTTERM

```
#define CONTENTTERM 96
```

Definition at line 451 of file ftypes.h.

4.37.2.278 PRIMENUMBER

```
#define PRIMENUMBER 97
```

Definition at line 452 of file ftypes.h.

4.37.2.279 EXTEUCLIDEAN

```
#define EXTEUCLIDEAN 98
```

Definition at line 453 of file ftypes.h.

4.37.2.280 MAKERATIONAL

```
#define MAKERATIONAL 99
```

Definition at line 454 of file ftypes.h.

4.37.2.281 INVERSEFUNCTION

```
#define INVERSEFUNCTION 100
```

Definition at line 455 of file ftypes.h.

4.37.2.282 IDFUNCTION

```
#define IDFUNCTION 101
```

Definition at line 456 of file ftypes.h.

4.37.2.283 PUTFIRST

```
#define PUTFIRST 102
```

Definition at line 457 of file ftypes.h.

4.37.2.284 PERMUTATIONS

```
#define PERMUTATIONS 103
```

Definition at line 458 of file ftypes.h.

4.37.2.285 PARTITIONS

```
#define PARTITIONS 104
```

Definition at line 459 of file ftypes.h.

4.37.2.286 MULFUNCTION

```
#define MULFUNCTION 105
```

Definition at line 460 of file ftypes.h.

4.37.2.287 MAXBUILTINFUNCTION

```
#define MAXBUILTINFUNCTION 105
```

Definition at line 462 of file ftypes.h.

4.37.2.288 FIRSTUSERFUNCTION

```
#define FIRSTUSERFUNCTION 150
```

Definition at line 463 of file ftypes.h.

4.37.2.289 ISYMBOL

```
#define ISYMBOL 0
```

Definition at line 471 of file ftypes.h.

4.37.2.290 PISYMBOL

```
#define PISYMBOL 1
```

Definition at line 472 of file ftypes.h.

4.37.2.291 COEFFSYMBOL

```
#define COEFFSYMBOL 2
```

Definition at line 473 of file ftypes.h.

4.37.2.292 NUMERATORSYMBOL

```
#define NUMERATORSYMBOL 3
```

Definition at line 474 of file ftypes.h.

4.37.2.293 DENOMINATORSYMBOL

```
#define DENOMINATORSYMBOL 4
```

Definition at line 475 of file ftypes.h.

4.37.2.294 WILDARGSYMBOL

```
#define WILDARGSYMBOL 5
```

Definition at line 476 of file ftypes.h.

4.37.2.295 DIMENSIONSYMBOL

```
#define DIMENSIONSYMBOL 6
```

Definition at line 477 of file ftypes.h.

4.37.2.296 FACTORSYMBOL

```
#define FACTORSYMBOL 7
```

Definition at line 478 of file ftypes.h.

4.37.2.297 SEPARATESYMBOL

```
#define SEPARATESYMBOL 8
```

Definition at line 479 of file ftypes.h.

4.37.2.298 BUILTINSYMBOLS

```
#define BUILTINSYMBOLS 9
```

Definition at line 481 of file ftypes.h.

4.37.2.299 FIRSTUSERSYMBOL

```
#define FIRSTUSERSYMBOL 20
```

Definition at line 482 of file ftypes.h.

4.37.2.300 BUILTININDICES

```
#define BUILTININDICES 1
```

Definition at line 484 of file ftypes.h.

4.37.2.301 BUILTINVECTORS

```
#define BUILTINVECTORS 1
```

Definition at line 485 of file ftypes.h.

4.37.2.302 BUILTINDOLLARS

```
#define BUILTINDOLLARS 1
```

Definition at line 486 of file ftypes.h.

4.37.2.303 WILDARGVECTOR

```
#define WILDARGVECTOR 0
```

Definition at line 488 of file ftypes.h.

4.37.2.304 WILDARGINDEX

```
#define WILDARGINDEX 0
```

Definition at line 489 of file ftypes.h.

4.37.2.305 TYPEEXPRESSION

```
#define TYPEEXPRESSION 0
```

Definition at line 500 of file ftypes.h.

4.37.2.306 TYPEIDNEW

```
#define TYPEIDNEW 1
```

Definition at line 501 of file ftypes.h.

4.37.2.307 TYPEIDOLD

```
#define TYPEIDOLD 2
```

Definition at line 502 of file ftypes.h.

4.37.2.308 TYPEOPERATION

```
#define TYPEOPERATION 3
```

Definition at line 503 of file ftypes.h.

4.37.2.309 TYPEREPEAT

```
#define TYPEREPEAT 4
```

Definition at line 504 of file ftypes.h.

4.37.2.310 TYPEENDREPEAT

```
#define TYPEENDREPEAT 5
```

Definition at line 505 of file ftypes.h.

4.37.2.311 TYPECOUNT

```
#define TYPECOUNT 20
```

Definition at line 509 of file ftypes.h.

4.37.2.312 TYPEMULT

```
#define TYPEMULT 21
```

Definition at line 510 of file ftypes.h.

4.37.2.313 TYPEGOTO

```
#define TYPEGOTO 22
```

Definition at line 511 of file ftypes.h.

4.37.2.314 TYPEDISCARD

```
#define TYPEDISCARD 23
```

Definition at line 512 of file ftypes.h.

4.37.2.315 TYPEIF

```
#define TYPEIF 24
```

Definition at line 513 of file ftypes.h.

4.37.2.316 TYPEELSE

```
#define TYPEELSE 25
```

Definition at line 514 of file ftypes.h.

4.37.2.317 TYPEELIF

```
#define TYPEELIF 26
```

Definition at line 515 of file ftypes.h.

4.37.2.318 TYPEENDIF

```
#define TYPEENDIF 27
```

Definition at line 516 of file ftypes.h.

4.37.2.319 TYPESUM

```
#define TYPESUM 28
```

Definition at line 517 of file ftypes.h.

4.37.2.320 TYPECHISHOLM

```
#define TYPECHISHOLM 29
```

Definition at line 518 of file ftypes.h.

4.37.2.321 TYPEREVERSE

```
#define TYPEREVERSE 30
```

Definition at line 519 of file ftypes.h.

4.37.2.322 TYPEARG

```
#define TYPEARG 31
```

Definition at line 520 of file ftypes.h.

4.37.2.323 TYPENORM

```
#define TYPENORM 32
```

Definition at line 521 of file ftypes.h.

4.37.2.324 TYPENORM2

```
#define TYPENORM2 33
```

Definition at line 522 of file ftypes.h.

4.37.2.325 TYPENORM3

```
#define TYPENORM3 34
```

Definition at line 523 of file ftypes.h.

4.37.2.326 TYPEEXIT

```
#define TYPEEXIT 35
```

Definition at line 524 of file ftypes.h.

4.37.2.327 TYPESETEXIT

```
#define TYPESETEXIT 36
```

Definition at line 525 of file ftypes.h.

4.37.2.328 TYPEPRINT

```
#define TYPEPRINT 37
```

Definition at line 526 of file ftypes.h.

4.37.2.329 TYPEFPRINT

```
#define TYPEFPRINT 38
```

Definition at line 527 of file ftypes.h.

4.37.2.330 TYPEREDEFPRE

```
#define TYPEREDEFPRE 39
```

Definition at line 528 of file ftypes.h.

4.37.2.331 TYPESPLITARG

```
#define TYPESPLITARG 40
```

Definition at line 529 of file ftypes.h.

4.37.2.332 TYPESPLITARG2

```
#define TYPESPLITARG2 41
```

Definition at line 530 of file ftypes.h.

4.37.2.333 TYPEFACTARG

```
#define TYPEFACTARG 42
```

Definition at line 531 of file ftypes.h.

4.37.2.334 TYPEFACTARG2

```
#define TYPEFACTARG2 43
```

Definition at line 532 of file ftypes.h.

4.37.2.335 TYPETRY

```
#define TYPETRY 44
```

Definition at line 533 of file ftypes.h.

4.37.2.336 TYPEASSIGN

```
#define TYPEASSIGN 45
```

Definition at line 534 of file ftypes.h.

4.37.2.337 TYPERENUMBER

```
#define TYPERENUMBER 46
```

Definition at line 535 of file ftypes.h.

4.37.2.338 TYPESUMFIX

```
#define TYPESUMFIX 47
```

Definition at line 536 of file ftypes.h.

4.37.2.339 TYPEFINDLOOP

```
#define TYPEFINDLOOP 48
```

Definition at line 537 of file ftypes.h.

4.37.2.340 TYPEUNRAVEL

```
#define TYPEUNRAVEL 49
```

Definition at line 538 of file ftypes.h.

4.37.2.341 TYPEADJUSTBOUNDS

```
#define TYPEADJUSTBOUNDS 50
```

Definition at line 539 of file ftypes.h.

4.37.2.342 TYPEINSIDE

```
#define TYPEINSIDE 51
```

Definition at line 540 of file ftypes.h.

4.37.2.343 TYPETERM

```
#define TYPETERM 52
```

Definition at line 541 of file ftypes.h.

4.37.2.344 TYPESORT

```
#define TYPESORT 53
```

Definition at line 542 of file ftypes.h.

4.37.2.345 TYPEDETCURDUM

```
#define TYPEDETCURDUM 54
```

Definition at line 543 of file ftypes.h.

4.37.2.346 TYPEINEXPRESSION

```
#define TYPEINEXPRESSION 55
```

Definition at line 544 of file ftypes.h.

4.37.2.347 TYPESPLITFIRSTARG

```
#define TYPESPLITFIRSTARG 56
```

Definition at line 545 of file ftypes.h.

4.37.2.348 TYPESPLITLASTARG

```
#define TYPESPLITLASTARG 57
```

Definition at line 546 of file ftypes.h.

4.37.2.349 TYPEMERGE

```
#define TYPEMERGE 58
```

Definition at line 547 of file ftypes.h.

4.37.2.350 TYPETESTUSE

```
#define TYPETESTUSE 59
```

Definition at line 548 of file ftypes.h.

4.37.2.351 TYPEAPPLY

```
#define TYPEAPPLY 60
```

Definition at line 549 of file ftypes.h.

4.37.2.352 TYPEAPPLYRESET

```
#define TYPEAPPLYRESET 61
```

Definition at line 550 of file ftypes.h.

4.37.2.353 TYPECHAININ

```
#define TYPECHAININ 62
```

Definition at line 551 of file ftypes.h.

4.37.2.354 TYPECHAINOUT

```
#define TYPECHAINOUT 63
```

Definition at line 552 of file ftypes.h.

4.37.2.355 TYPENORM4

```
#define TYPENORM4 64
```

Definition at line 553 of file ftypes.h.

4.37.2.356 TYPEFACTOR

```
#define TYPEFACTOR 65
```

Definition at line 554 of file ftypes.h.

4.37.2.357 TYPEARGIMPLODE

```
#define TYPEARGIMPLODE 66
```

Definition at line 555 of file ftypes.h.

4.37.2.358 TYPEARGEXPLODE

```
#define TYPEARGEXPLODE 67
```

Definition at line 556 of file ftypes.h.

4.37.2.359 TYPEDENOMINATORS

```
#define TYPEDENOMINATORS 68
```

Definition at line 557 of file ftypes.h.

4.37.2.360 TYPESTUFFLE

```
#define TYPESTUFFLE 69
```

Definition at line 558 of file ftypes.h.

4.37.2.361 TYPEDROPCOEFFICIENT

```
#define TYPEDROPCOEFFICIENT 70
```

Definition at line 559 of file ftypes.h.

4.37.2.362 TYPETRANSFORM

```
#define TYPETRANSFORM 71
```

Definition at line 560 of file ftypes.h.

4.37.2.363 TYPETOPOLYNOMIAL

```
#define TYPETOPOLYNOMIAL 72
```

Definition at line 561 of file ftypes.h.

4.37.2.364 TYPEFROMPOLYNOMIAL

```
#define TYPEFROMPOLYNOMIAL 73
```

Definition at line 562 of file ftypes.h.

4.37.2.365 TYPEDOLOOP

```
#define TYPEDOLOOP 74
```

Definition at line 563 of file ftypes.h.

4.37.2.366 TYPEENDDOLOOP

```
#define TYPEENDDOLOOP 75
```

Definition at line 564 of file ftypes.h.

4.37.2.367 TYPEDROPSYMBOLS

```
#define TYPEDROPSYMBOLS 76
```

Definition at line 565 of file ftypes.h.

4.37.2.368 TYPEPUTINSIDE

```
#define TYPEPUTINSIDE 77
```

Definition at line 566 of file ftypes.h.

4.37.2.369 TYPETOSPECTATOR

```
#define TYPETOSPECTATOR 78
```

Definition at line 567 of file ftypes.h.

4.37.2.370 TYPEARGTOEXTRASYMBOL

```
#define TYPEARGTOEXTRASYMBOL 79
```

Definition at line 568 of file ftypes.h.

4.37.2.371 TAKETRACE

```
#define TAKETRACE 1
```

Definition at line 573 of file ftypes.h.

4.37.2.372 CONTRACT

```
#define CONTRACT 2
```

Definition at line 574 of file ftypes.h.

4.37.2.373 RATIO

```
#define RATIO 3
```

Definition at line 575 of file ftypes.h.

4.37.2.374 SYMMETRIZE

```
#define SYMMETRIZE 4
```

Definition at line 576 of file ftypes.h.

4.37.2.375 TENVEC

```
#define TENVEC 5
```

Definition at line 577 of file ftypes.h.

4.37.2.376 SUMNUM1

```
#define SUMNUM1 6
```

Definition at line 578 of file ftypes.h.

4.37.2.377 SUMNUM2

```
#define SUMNUM2 7
```

Definition at line 579 of file ftypes.h.

4.37.2.378 WILDDUMMY

```
#define WILDDUMMY 0
```

Definition at line 585 of file ftypes.h.

4.37.2.379 SYMTONUM

```
#define SYMTONUM 1
```

Definition at line 586 of file ftypes.h.

4.37.2.380 SYMTOSYM

```
#define SYMTOSYM 2
```

Definition at line 587 of file ftypes.h.

4.37.2.381 SYMTOSUB

```
#define SYMTOSUB 3
```

Definition at line 588 of file ftypes.h.

4.37.2.382 VECTOMIN

```
#define VECTOMIN 4
```

Definition at line 589 of file ftypes.h.

4.37.2.383 VECTOVEC

```
#define VECTOVEC 5
```

Definition at line 590 of file ftypes.h.

4.37.2.384 VECTOSUB

```
#define VECTOSUB 6
```

Definition at line 591 of file ftypes.h.

4.37.2.385 INDTOIND

```
#define INDTOIND 7
```

Definition at line 592 of file ftypes.h.

4.37.2.386 INDTOSUB

```
#define INDTOSUB 8
```

Definition at line 593 of file ftypes.h.

4.37.2.387 FUNTOFUN

```
#define FUNTOFUN 9
```

Definition at line 594 of file ftypes.h.

4.37.2.388 ARGTOARG

```
#define ARGTOARG 10
```

Definition at line 595 of file ftypes.h.

4.37.2.389 ARLTOARL

```
#define ARLTOARL 11
```

Definition at line 596 of file ftypes.h.

4.37.2.390 EXPTOEXP

```
#define EXPTOEXP 12
```

Definition at line 597 of file ftypes.h.

4.37.2.391 FROMBRAC

```
#define FROMBRAC 13
```

Definition at line 598 of file ftypes.h.

4.37.2.392 FROMSET

```
#define FROMSET 14
```

Definition at line 599 of file ftypes.h.

4.37.2.393 SETTONUM

```
#define SETTONUM 15
```

Definition at line 600 of file ftypes.h.

4.37.2.394 WILDCARDS

```
#define WILDCARDS 16
```

Definition at line 601 of file ftypes.h.

4.37.2.395 SETNUMBER

```
#define SETNUMBER 17
```

Definition at line 602 of file ftypes.h.

4.37.2.396 LOADDOLLAR

```
#define LOADDOLLAR 18
```

Definition at line 603 of file ftypes.h.

4.37.2.397 NUMTONUM

```
#define NUMTONUM 20
```

Definition at line 607 of file ftypes.h.

4.37.2.398 NUMTOSYM

```
#define NUMTOSYM 21
```

Definition at line 608 of file ftypes.h.

4.37.2.399 NUMTOIND

```
#define NUMTOIND 22
```

Definition at line 609 of file ftypes.h.

4.37.2.400 NUMTOSUB

```
#define NUMTOSUB 23
```

Definition at line 610 of file ftypes.h.

4.37.2.401 CLEANFLAG

```
#define CLEANFLAG 0
```

Definition at line 616 of file ftypes.h.

4.37.2.402 DIRTYFLAG

```
#define DIRTYFLAG 1
```

Definition at line 617 of file ftypes.h.

4.37.2.403 DIRTYSYMFLAG

```
#define DIRTYSYMFLAG 2
```

Definition at line 618 of file ftypes.h.

4.37.2.404 MUSTCLEANPRF

```
#define MUSTCLEANPRF 4
```

Definition at line 619 of file ftypes.h.

4.37.2.405 ALLDIRTY

```
#define ALLDIRTY (DIRTYFLAG|DIRTYSYMFLAG)
```

Definition at line 620 of file ftypes.h.

4.37.2.406 ARGHEAD

```
#define ARGHEAD 2
```

Definition at line 622 of file ftypes.h.

4.37.2.407 FUNHEAD

```
#define FUNHEAD 3
```

Definition at line 623 of file ftypes.h.

4.37.2.408 SUBEXPSIZE

```
#define SUBEXPSIZE 5
```

Definition at line 624 of file ftypes.h.

4.37.2.409 EXPRHEAD

```
#define EXPRHEAD 5
```

Definition at line 625 of file ftypes.h.

4.37.2.410 TYPEARGHEADSIZE

```
#define TYPEARGHEADSIZE 6
```

Definition at line 626 of file ftypes.h.

4.37.2.411 SKIP

```
#define SKIP 1
```

Definition at line 633 of file ftypes.h.

4.37.2.412 DROP

```
#define DROP 2
```

Definition at line 634 of file ftypes.h.

4.37.2.413 HIDE

```
#define HIDE 3
```

Definition at line 635 of file ftypes.h.

4.37.2.414 UNHIDE

```
#define UNHIDE 4
```

Definition at line 636 of file ftypes.h.

4.37.2.415 INTOHIDE

```
#define INTOHIDE 5
```

Definition at line 637 of file ftypes.h.

4.37.2.416 LOCALEXPRESSION

```
#define LOCALEXPRESSION 0
```

Definition at line 643 of file ftypes.h.

4.37.2.417 SKIPLEXPRESSION

```
#define SKIPLEXPRESSION 1
```

Definition at line 644 of file ftypes.h.

4.37.2.418 DROPLEXPRESSION

```
#define DROPLEXPRESSION 2
```

Definition at line 645 of file ftypes.h.

4.37.2.419 DROPPEDEXPRESSION

```
#define DROPPEDEXPRESSION 3
```

Definition at line 646 of file ftypes.h.

4.37.2.420 GLOBALEXPRESSION

```
#define GLOBALEXPRESSION 4
```

Definition at line 647 of file ftypes.h.

4.37.2.421 SKIPGEXPRESSION

```
#define SKIPGEXPRESSION 5
```

Definition at line 648 of file ftypes.h.

4.37.2.422 DROPGEXPRESSION

```
#define DROPGEXPRESSION 6
```

Definition at line 649 of file ftypes.h.

4.37.2.423 STOREDEXPRESSION

```
#define STOREDEXPRESSION 8
```

Definition at line 650 of file ftypes.h.

4.37.2.424 HIDDENLEXPRESSION

```
#define HIDDENLEXPRESSION 9
```

Definition at line 651 of file ftypes.h.

4.37.2.425 HIDDENGEXPRESSION

```
#define HIDDENGEXPRESSION 13
```

Definition at line 652 of file ftypes.h.

4.37.2.426 INCEXPRESSION

```
#define INCEXPRESSION 9
```

Definition at line 653 of file ftypes.h.

4.37.2.427 HIDELEXPRESSION

```
#define HIDELEXPRESSION 10
```

Definition at line 654 of file ftypes.h.

4.37.2.428 HIDEGEXPRESSION

```
#define HIDEGEXPRESSION 14
```

Definition at line 655 of file ftypes.h.

4.37.2.429 DROPHLEXPRESSION

```
#define DROPHLEXPRESSION 11
```

Definition at line 656 of file ftypes.h.

4.37.2.430 DROPHGEXPRESSION

```
#define DROPHGEXPRESSION 15
```

Definition at line 657 of file ftypes.h.

4.37.2.431 UNHIDELEXPRESSION

```
#define UNHIDELEXPRESSION 12
```

Definition at line 658 of file ftypes.h.

4.37.2.432 UNHIDEGEXPRESSION

```
#define UNHIDEGEXPRESSION 16
```

Definition at line 659 of file ftypes.h.

4.37.2.433 INTOHIDELEXPRESSION

```
#define INTOHIDELEXPRESSION 17
```

Definition at line 660 of file ftypes.h.

4.37.2.434 INTOHIDEGEXPRESSION

```
#define INTOHIDEGEXPRESSION 18
```

Definition at line 661 of file ftypes.h.

4.37.2.435 SPECTATOREXPRESSION

```
#define SPECTATOREXPRESSION 19
```

Definition at line 662 of file ftypes.h.

4.37.2.436 DROPSPECTATOREXPRESSION

```
#define DROPSPECTATOREXPRESSION 20
```

Definition at line 663 of file ftypes.h.

4.37.2.437 SKIPUNHIDELEXPRESSION

```
#define SKIPUNHIDELEXPRESSION 21
```

Definition at line 664 of file ftypes.h.

4.37.2.438 SKIPUNHIDEGEXPRESSION

```
#define SKIPUNHIDEGEXPRESSION 22
```

Definition at line 665 of file ftypes.h.

4.37.2.439 PRINTOFF

```
#define PRINTOFF 0
```

Definition at line 667 of file ftypes.h.

4.37.2.440 PRINTON

```
#define PRINTON 1
```

Definition at line 668 of file ftypes.h.

4.37.2.441 PRINTCONTENTS

```
#define PRINTCONTENTS 2
```

Definition at line 669 of file ftypes.h.

4.37.2.442 PRINTCONTENT

```
#define PRINTCONTENT 3
```

Definition at line 670 of file ftypes.h.

4.37.2.443 PRINTLFILE

```
#define PRINTLFILE 4
```

Definition at line 671 of file ftypes.h.

4.37.2.444 PRINTONETERM

```
#define PRINTONETERM 8
```

Definition at line 672 of file ftypes.h.

4.37.2.445 PRINTONEFUNCTION

```
#define PRINTONEFUNCTION 16
```

Definition at line 673 of file ftypes.h.

4.37.2.446 PRINTALL

```
#define PRINTALL 32
```

Definition at line 674 of file ftypes.h.

4.37.2.447 REGULAREXPRESSION

```
#define REGULAREXPRESSION -1
```

Definition at line 680 of file ftypes.h.

4.37.2.448 REDEFINEDEXPRESSION

```
#define REDEFINEDEXPRESSION -2
```

Definition at line 681 of file ftypes.h.

4.37.2.449 NEWLYDEFINEDEXPRESSION

```
#define NEWLYDEFINEDEXPRESSION -3
```

Definition at line 682 of file ftypes.h.

4.37.2.450 GENERALFUNCTION

```
#define GENERALFUNCTION 0
```

Definition at line 691 of file ftypes.h.

4.37.2.451 FASTFUNCTION

```
#define FASTFUNCTION 1
```

Definition at line 692 of file ftypes.h.

4.37.2.452 TENSORFUNCTION

```
#define TENSORFUNCTION 2
```

Definition at line 693 of file ftypes.h.

4.37.2.453 GAMMAFUNCTION

```
#define GAMMAFUNCTION 4
```

Definition at line 694 of file ftypes.h.

4.37.2.454 POS_

```
#define POS_ 0 /* integer > 0 */
```

Definition at line 701 of file ftypes.h.

4.37.2.455 POS0_

```
#define POS0_ 1 /* integer >= 0 */
```

Definition at line 702 of file ftypes.h.

4.37.2.456 NEG_

```
#define NEG_ 2 /* integer < 0 */
```

Definition at line 703 of file ftypes.h.

4.37.2.457 NEG0_

```
#define NEG0_ 3 /* integer <= 0 */
```

Definition at line 704 of file ftypes.h.

4.37.2.458 EVEN_

```
#define EVEN_ 4 /* integer (even) */
```

Definition at line 705 of file ftypes.h.

4.37.2.459 ODD_

```
#define ODD_ 5 /* integer (odd) */
```

Definition at line 706 of file ftypes.h.

4.37.2.460 Z_

```
#define Z_ 6 /* integer */
```

Definition at line 707 of file ftypes.h.

4.37.2.461 SYMBOL_

```
#define SYMBOL_ 7 /* symbol only */
```

Definition at line 708 of file ftypes.h.

4.37.2.462 FIXED_

```
#define FIXED_ 8 /* fixed index */
```

Definition at line 709 of file ftypes.h.

4.37.2.463 INDEX_

```
#define INDEX_ 9 /* index only */
```

Definition at line 710 of file ftypes.h.

4.37.2.464 Q_

```
#define Q_ 10 /* rational */
```

Definition at line 711 of file ftypes.h.

4.37.2.465 DUMMYINDEX_

```
#define DUMMYINDEX_ 11 /* dummy index only */
```

Definition at line 712 of file ftypes.h.

4.37.2.466 VECTOR_

```
#define VECTOR_ 12 /* vector only */
```

Definition at line 713 of file ftypes.h.

4.37.2.467 GAMMA1

```
#define GAMMA1 0
```

Definition at line 719 of file ftypes.h.

4.37.2.468 GAMMA5

```
#define GAMMA5 -1
```

Definition at line 720 of file ftypes.h.

4.37.2.469 GAMMA6

```
#define GAMMA6 -2
```

Definition at line 721 of file ftypes.h.

4.37.2.470 GAMMA7

```
#define GAMMA7 -3
```

Definition at line 722 of file ftypes.h.

4.37.2.471 FUNNYVEC

```
#define FUNNYVEC -4
```

Definition at line 723 of file ftypes.h.

4.37.2.472 FUNNYWILD

```
#define FUNNYWILD -5
```

Definition at line 724 of file ftypes.h.

4.37.2.473 SUMMEDIND

```
#define SUMMEDIND -6
```

Definition at line 725 of file ftypes.h.

4.37.2.474 NOINDEX

```
#define NOINDEX -7
```

Definition at line 726 of file ftypes.h.

4.37.2.475 FUNNYDOLLAR

```
#define FUNNYDOLLAR -8
```

Definition at line 727 of file ftypes.h.

4.37.2.476 EMPTYINDEX

```
#define EMPTYINDEX -9
```

Definition at line 728 of file ftypes.h.

4.37.2.477 MINSPEC

```
#define MINSPEC -10
```

Definition at line 734 of file ftypes.h.

4.37.2.478 USEDFLAG

```
#define USEDFLAG 2
```

Definition at line 736 of file ftypes.h.

4.37.2.479 DUMMYFLAG

```
#define DUMMYFLAG 1
```

Definition at line 737 of file ftypes.h.

4.37.2.480 MAINSORT

```
#define MAINSORT 0
```

Definition at line 739 of file ftypes.h.

4.37.2.481 FUNCTIONSORT

```
#define FUNCTIONSORT 1
```

Definition at line 740 of file ftypes.h.

4.37.2.482 SUBSORT

```
#define SUBSORT 2
```

Definition at line 741 of file ftypes.h.

4.37.2.483 FLOATMODE

```
#define FLOATMODE 1
```

Definition at line 743 of file ftypes.h.

4.37.2.484 RATIONALMODE

```
#define RATIONALMODE 0
```

Definition at line 744 of file ftypes.h.

4.37.2.485 NUMSPECSETS

```
#define NUMSPECSETS 10
```

Definition at line 746 of file ftypes.h.

4.37.2.486 EATTENSOR

```
#define EATTENSOR 0x2000
```

Definition at line 747 of file ftypes.h.

4.37.2.487 ISZERO

```
#define ISZERO 1
```

Definition at line 749 of file ftypes.h.

4.37.2.488 ISUNMODIFIED

```
#define ISUNMODIFIED 2
```

Definition at line 750 of file ftypes.h.

4.37.2.489 ISCOMPRESSED

```
#define ISCOMPRESSED 4
```

Definition at line 751 of file ftypes.h.

4.37.2.490 ISINRHS

```
#define ISINRHS 8
```

Definition at line 752 of file ftypes.h.

4.37.2.491 ISFACTORIZED

```
#define ISFACTORIZED 16
```

Definition at line 753 of file ftypes.h.

4.37.2.492 TOBEFACTORED

```
#define TOBEFACTORED 32
```

Definition at line 754 of file ftypes.h.

4.37.2.493 TOBEUNFACTORED

```
#define TOBEUNFACTORED 64
```

Definition at line 755 of file ftypes.h.

4.37.2.494 KEEPZERO

```
#define KEEPZERO 128
```

Definition at line 756 of file ftypes.h.

4.37.2.495 VARTYPENONE

```
#define VARTYPENONE 0
```

Definition at line 758 of file ftypes.h.

4.37.2.496 VARTYPECOMPLEX

```
#define VARTYPECOMPLEX 1
```

Definition at line 759 of file ftypes.h.

4.37.2.497 VARTYPEIMAGINARY

```
#define VARTYPEIMAGINARY 2
```

Definition at line 760 of file ftypes.h.

4.37.2.498 VARTYPEROOTOFUNITY

```
#define VARTYPEROOTOFUNITY 4
```

Definition at line 761 of file ftypes.h.

4.37.2.499 VARTYPEMINUS

```
#define VARTYPEMINUS 8
```

Definition at line 762 of file ftypes.h.

4.37.2.500 CYCLESYMMETRIC

```
#define CYCLESYMMETRIC 1
```

Definition at line 763 of file ftypes.h.

4.37.2.501 RCYCLESYMMETRIC

```
#define RCYCLESYMMETRIC 2
```

Definition at line 764 of file ftypes.h.

4.37.2.502 SYMMETRIC

```
#define SYMMETRIC 3
```

Definition at line 765 of file ftypes.h.

4.37.2.503 ANTISYMMETRIC

```
#define ANTISYMMETRIC 4
```

Definition at line 766 of file ftypes.h.

4.37.2.504 REVERSEORDER

```
#define REVERSEORDER 256
```

Definition at line 767 of file ftypes.h.

4.37.2.505 SUBMULTI

```
#define SUBMULTI 1
```

Definition at line 773 of file ftypes.h.

4.37.2.506 SUBONCE

```
#define SUBONCE 2
```

Definition at line 774 of file ftypes.h.

4.37.2.507 SUBONLY

```
#define SUBONLY 3
```

Definition at line 775 of file ftypes.h.

4.37.2.508 SUBMANY

```
#define SUBMANY 4
```

Definition at line 776 of file ftypes.h.

4.37.2.509 SUBVECTOR

```
#define SUBVECTOR 5
```

Definition at line 777 of file ftypes.h.

4.37.2.510 SUBSELECT

```
#define SUBSELECT 6
```

Definition at line 778 of file ftypes.h.

4.37.2.511 SUBALL

```
#define SUBALL 7
```

Definition at line 779 of file ftypes.h.

4.37.2.512 SUBMASK

```
#define SUBMASK 15
```

Definition at line 780 of file ftypes.h.

4.37.2.513 SUBDISORDER

```
#define SUBDISORDER 16
```

Definition at line 781 of file ftypes.h.

4.37.2.514 SUBAFTER

```
#define SUBAFTER 32
```

Definition at line 782 of file ftypes.h.

4.37.2.515 SUBAFTERNOT

```
#define SUBAFTERNOT 64
```

Definition at line 783 of file ftypes.h.

4.37.2.516 IDHEAD

```
#define IDHEAD 6
```

Definition at line 785 of file ftypes.h.

4.37.2.517 DOLLARFLAG

```
#define DOLLARFLAG 1
```

Definition at line 787 of file ftypes.h.

4.37.2.518 NORMALIZEFLAG

```
#define NORMALIZEFLAG 2
```

Definition at line 788 of file ftypes.h.

4.37.2.519 GIDENT

```
#define GIDENT 1
```

Definition at line 790 of file ftypes.h.

4.37.2.520 GFIVE

```
#define GFIVE 4
```

Definition at line 791 of file ftypes.h.

4.37.2.521 GPLUS

```
#define GPLUS 3
```

Definition at line 792 of file ftypes.h.

4.37.2.522 GMINUS

```
#define GMINUS 2
```

Definition at line 793 of file ftypes.h.

4.37.2.523 LONGNUMBER

```
#define LONGNUMBER 1
```

Definition at line 799 of file ftypes.h.

4.37.2.524 MATCH

```
#define MATCH 2
```

Definition at line 800 of file ftypes.h.

4.37.2.525 COEFFI

```
#define COEFFI 3
```

Definition at line 801 of file ftypes.h.

4.37.2.526 SUBEXPR

```
#define SUBEXPR 4
```

Definition at line 802 of file ftypes.h.

4.37.2.527 MULTIPLEOF

```
#define MULTIPLEOF 5
```

Definition at line 803 of file ftypes.h.

4.37.2.528 IFDOLLAR

```
#define IFDOLLAR 6
```

Definition at line 804 of file ftypes.h.

4.37.2.529 IFEXPRESSION

```
#define IFEXPRESSION 7
```

Definition at line 805 of file ftypes.h.

4.37.2.530 IFDOLLAREXTRA

```
#define IFDOLLAREXTRA 8
```

Definition at line 806 of file ftypes.h.

4.37.2.531 IFISFACTORIZED

```
#define IFISFACTORIZED 9
```

Definition at line 807 of file ftypes.h.

4.37.2.532 IFOCCURS

```
#define IFOCCURS 10
```

Definition at line 808 of file ftypes.h.

4.37.2.533 GREATER

```
#define GREATER 0
```

Definition at line 809 of file ftypes.h.

4.37.2.534 GREATEREQUAL

```
#define GREATEREQUAL 1
```

Definition at line 810 of file ftypes.h.

4.37.2.535 LESS

```
#define LESS 2
```

Definition at line 811 of file ftypes.h.

4.37.2.536 LESSEQUAL

```
#define LESSEQUAL 3
```

Definition at line 812 of file ftypes.h.

4.37.2.537 EQUAL

```
#define EQUAL 4
```

Definition at line 813 of file ftypes.h.

4.37.2.538 NOTEQUAL

```
#define NOTEQUAL 5
```

Definition at line 814 of file ftypes.h.

4.37.2.539 ORCOND

```
#define ORCOND 6
```

Definition at line 815 of file ftypes.h.

4.37.2.540 ANDCOND

```
#define ANDCOND 7
```

Definition at line 816 of file ftypes.h.

4.37.2.541 DUMMY

```
#define DUMMY 1
```

Definition at line 817 of file ftypes.h.

4.37.2.542 SORT

```
#define SORT 1
```

Definition at line 818 of file ftypes.h.

4.37.2.543 STORE

```
#define STORE 2
```

Definition at line 819 of file ftypes.h.

4.37.2.544 END

```
#define END 3
```

Definition at line 820 of file ftypes.h.

4.37.2.545 GLOBAL

```
#define GLOBAL 4
```

Definition at line 821 of file ftypes.h.

4.37.2.546 CLEAR

```
#define CLEAR 5
```

Definition at line 822 of file ftypes.h.

4.37.2.547 VECTBIT

```
#define VECTBIT 1
```

Definition at line 824 of file ftypes.h.

4.37.2.548 DOTPBIT

```
#define DOTPBIT 2
```

Definition at line 825 of file ftypes.h.

4.37.2.549 FUNBIT

```
#define FUNBIT 4
```

Definition at line 826 of file ftypes.h.

4.37.2.550 SETBIT

```
#define SETBIT 8
```

Definition at line 827 of file ftypes.h.

4.37.2.551 EXTRAPARAMETER

```
#define EXTRAPARAMETER 0x4000
```

Definition at line 829 of file ftypes.h.

4.37.2.552 GENCOMMUTE

```
#define GENCOMMUTE 0
```

Definition at line 830 of file ftypes.h.

4.37.2.553 GENNONCOMMUTE

```
#define GENNONCOMMUTE 0x2000
```

Definition at line 831 of file ftypes.h.

4.37.2.554 NAMENOTFOUND

```
#define NAMENOTFOUND -9
```

Definition at line 833 of file ftypes.h.

4.37.2.555 DOLUNDEFINED

```
#define DOLUNDEFINED 0
```

Definition at line 839 of file ftypes.h.

4.37.2.556 DOLNUMBER

```
#define DOLNUMBER 1
```

Definition at line 840 of file ftypes.h.

4.37.2.557 DOLARGUMENT

```
#define DOLARGUMENT 2
```

Definition at line 841 of file ftypes.h.

4.37.2.558 DOLSUBTERM

```
#define DOLSUBTERM 3
```

Definition at line 842 of file ftypes.h.

4.37.2.559 DOLTERMS

```
#define DOLTERMS 4
```

Definition at line 843 of file ftypes.h.

4.37.2.560 DOLWILDARGS

```
#define DOLWILDARGS 5
```

Definition at line 844 of file ftypes.h.

4.37.2.561 DOLINDEX

```
#define DOLINDEX 6
```

Definition at line 845 of file ftypes.h.

4.37.2.562 DOLZERO

```
#define DOLZERO 7
```

Definition at line 846 of file ftypes.h.

4.37.2.563 FINDLOOP

```
#define FINDLOOP 0
```

Definition at line 848 of file ftypes.h.

4.37.2.564 REPLACELOOP

```
#define REPLACELOOP 1
```

Definition at line 849 of file ftypes.h.

4.37.2.565 NOFUNPOWERS

```
#define NOFUNPOWERS 0
```

Definition at line 851 of file ftypes.h.

4.37.2.566 COMFUNPOWERS

```
#define COMFUNPOWERS 1
```

Definition at line 852 of file ftypes.h.

4.37.2.567 ALLFUNPOWERS

```
#define ALLFUNPOWERS 2
```

Definition at line 853 of file ftypes.h.

4.37.2.568 PROPERORDERFLAG

```
#define PROPERORDERFLAG 0
```

Definition at line 855 of file ftypes.h.

4.37.2.569 REGULAR

```
#define REGULAR 0
```

Definition at line 857 of file ftypes.h.

4.37.2.570 FINISH

```
#define FINISH 1
```

Definition at line 858 of file ftypes.h.

4.37.2.571 POLYADD

```
#define POLYADD 1
```

Definition at line 860 of file ftypes.h.

4.37.2.572 POLYSUB

```
#define POLYSUB 2
```

Definition at line 861 of file ftypes.h.

4.37.2.573 POLYMUL

```
#define POLYMUL 3
```

Definition at line 862 of file ftypes.h.

4.37.2.574 POLYDIV

```
#define POLYDIV 4
```

Definition at line 863 of file ftypes.h.

4.37.2.575 POLYREM

```
#define POLYREM 5
```

Definition at line 864 of file ftypes.h.

4.37.2.576 POLYGCD

```
#define POLYGCD 6
```

Definition at line 865 of file ftypes.h.

4.37.2.577 POLYINTFAC

```
#define POLYINTFAC 7
```

Definition at line 866 of file ftypes.h.

4.37.2.578 POLYNORM

```
#define POLYNORM 8
```

Definition at line 867 of file ftypes.h.

4.37.2.579 MODNONE

```
#define MODNONE 0
```

Definition at line 869 of file ftypes.h.

4.37.2.580 MODSUM

```
#define MODSUM 1
```

Definition at line 870 of file ftypes.h.

4.37.2.581 MODMAX

```
#define MODMAX 2
```

Definition at line 871 of file ftypes.h.

4.37.2.582 MODMIN

```
#define MODMIN 3
```

Definition at line 872 of file ftypes.h.

4.37.2.583 MODLOCAL

```
#define MODLOCAL 4
```

Definition at line 873 of file ftypes.h.

4.37.2.584 ELEMENTUSED

```
#define ELEMENTUSED 1
```

Definition at line 875 of file ftypes.h.

4.37.2.585 ELEMENTLOADED

```
#define ELEMENTLOADED 2
```

Definition at line 876 of file ftypes.h.

4.37.2.586 POSNEG

```
#define POSNEG 0x1
```

Definition at line 881 of file ftypes.h.

4.37.2.587 INVERSETABLE

```
#define INVERSETABLE 0x2
```

Definition at line 882 of file ftypes.h.

4.37.2.588 COEFFICIENTSONLY

```
#define COEFFICIENTSONLY 0x4
```

Definition at line 883 of file ftypes.h.

4.37.2.589 ALSOPOWERS

```
#define ALSOPOWERS 0x8
```

Definition at line 884 of file ftypes.h.

4.37.2.590 ALSOFUNARGS

```
#define ALSOFUNARGS 0x10
```

Definition at line 885 of file ftypes.h.

4.37.2.591 ALSODOLLARS

```
#define ALSODOLLARS 0x20
```

Definition at line 886 of file ftypes.h.

4.37.2.592 NOINVERSES

```
#define NOINVERSES 0x40
```

Definition at line 887 of file ftypes.h.

4.37.2.593 POSITIVEONLY

```
#define POSITIVEONLY 0
```

Definition at line 889 of file ftypes.h.

4.37.2.594 UNPACK

```
#define UNPACK 0x80
```

Definition at line 890 of file ftypes.h.

4.37.2.595 NOUNPACK

```
#define NOUNPACK 0
```

Definition at line 891 of file ftypes.h.

4.37.2.596 FROMFUNCTION

```
#define FROMFUNCTION 0x100
```

Definition at line 892 of file ftypes.h.

4.37.2.597 VARNAMES

```
#define VARNAMES 0
```

Definition at line 894 of file ftypes.h.

4.37.2.598 AUTONAMES

```
#define AUTONAMES 1
```

Definition at line 895 of file ftypes.h.

4.37.2.599 EXPRNAMES

```
#define EXPRNAMES 2
```

Definition at line 896 of file ftypes.h.

4.37.2.600 DOLLARNAMES

```
#define DOLLARNAMES 3
```

Definition at line 897 of file ftypes.h.

4.37.2.601 DUMMYBUFFER

```
#define DUMMYBUFFER 1
```

Definition at line 952 of file ftypes.h.

4.37.2.602 ALLARGS

```
#define ALLARGS 1
```

Definition at line 954 of file ftypes.h.

4.37.2.603 NUMARG

```
#define NUMARG 2
```

Definition at line 955 of file ftypes.h.

4.37.2.604 ARGRANGE

```
#define ARGRANGE 3
```

Definition at line 956 of file ftypes.h.

4.37.2.605 MAKEARGS

```
#define MAKEARGS 4
```

Definition at line 957 of file ftypes.h.

4.37.2.606 MAXRANGEINDICATOR

```
#define MAXRANGEINDICATOR 4
```

Definition at line 958 of file ftypes.h.

4.37.2.607 REPLACEARG

```
#define REPLACEARG 5
```

Definition at line 959 of file ftypes.h.

4.37.2.608 ENCODEARG

```
#define ENCODEARG 6
```

Definition at line 960 of file ftypes.h.

4.37.2.609 DECODEARG

```
#define DECODEARG 7
```

Definition at line 961 of file ftypes.h.

4.37.2.610 IMPLODEARG

```
#define IMPLODEARG 8
```

Definition at line 962 of file ftypes.h.

4.37.2.611 EXPLODEARG

```
#define EXPLODEARG 9
```

Definition at line 963 of file ftypes.h.

4.37.2.612 PERMUTEARG

```
#define PERMUTEARG 10
```

Definition at line 964 of file ftypes.h.

4.37.2.613 REVERSEARG

```
#define REVERSEARG 11
```

Definition at line 965 of file ftypes.h.

4.37.2.614 CYCLEARG

```
#define CYCLEARG 12
```

Definition at line 966 of file ftypes.h.

4.37.2.615 ISLYNDON

```
#define ISLYNDON 13
```

Definition at line 967 of file ftypes.h.

4.37.2.616 ISLYNDONR

```
#define ISLYNDONR 14
```

Definition at line 968 of file ftypes.h.

4.37.2.617 TOLYNDON

```
#define TOLYNDON 15
```

Definition at line 969 of file ftypes.h.

4.37.2.618 TOLYNDONR

```
#define TOLYNDONR 16
```

Definition at line 970 of file ftypes.h.

4.37.2.619 ADDARG

```
#define ADDARG 17
```

Definition at line 971 of file ftypes.h.

4.37.2.620 MULTIPLYARG

```
#define MULTIPLYARG 18
```

Definition at line 972 of file ftypes.h.

4.37.2.621 DROPARG

```
#define DROPARG 19
```

Definition at line 973 of file ftypes.h.

4.37.2.622 SELECTARG

```
#define SELECTARG 20
```

Definition at line 974 of file ftypes.h.

4.37.2.623 DEDUPARG

```
#define DEDUPARG 21
```

Definition at line 975 of file ftypes.h.

4.37.2.624 BASECODE

```
#define BASECODE 1
```

Definition at line 977 of file ftypes.h.

4.37.2.625 YESLYNDON

```
#define YESLYNDON 1
```

Definition at line 978 of file ftypes.h.

4.37.2.626 NOLYNDON

```
#define NOLYNDON 2
```

Definition at line 979 of file ftypes.h.

4.37.2.627 TOPOLYNOMIALFLAG

```
#define TOPOLYNOMIALFLAG 1
```

Definition at line 981 of file ftypes.h.

4.37.2.628 FACTARGFLAG

```
#define FACTARGFLAG 2
```

Definition at line 982 of file ftypes.h.

4.37.2.629 OLDFACTARG

```
#define OLDFACTARG 1
```

Definition at line 984 of file ftypes.h.

4.37.2.630 NEWFACTARG

```
#define NEWFACTARG 0
```

Definition at line 985 of file ftypes.h.

4.37.2.631 FROMMODULEOPTION

```
#define FROMMODULEOPTION 0
```

Definition at line 987 of file ftypes.h.

4.37.2.632 FROMPOINTINSTRUCTION

```
#define FROMPOINTINSTRUCTION 1
```

Definition at line 988 of file ftypes.h.

4.37.2.633 EXTRASYMBOL

```
#define EXTRASYMBOL 0
```

Definition at line 990 of file ftypes.h.

4.37.2.634 REGULARSYMBOL

```
#define REGULARSYMBOL 1
```

Definition at line 991 of file ftypes.h.

4.37.2.635 EXPRESSIONNUMBER

```
#define EXPRESSIONNUMBER 2
```

Definition at line 992 of file ftypes.h.

4.37.2.636 O_NONE

```
#define O_NONE 0
```

Definition at line 994 of file ftypes.h.

4.37.2.637 O_CSE

```
#define O_CSE 1
```

Definition at line 995 of file ftypes.h.

4.37.2.638 O_CSEGREEDY

```
#define O_CSEGREEDY 2
```

Definition at line 996 of file ftypes.h.

4.37.2.639 O_GREEDY

```
#define O_GREEDY 3
```

Definition at line 997 of file ftypes.h.

4.37.2.640 O_OCCURRENCE

```
#define O_OCCURRENCE 0
```

Definition at line 999 of file ftypes.h.

4.37.2.641 O_MCTS

```
#define O_MCTS 1
```

Definition at line 1000 of file ftypes.h.

4.37.2.642 O_SIMULATED_ANNEALING

```
#define O_SIMULATED_ANNEALING 2
```

Definition at line 1001 of file ftypes.h.

4.37.2.643 O_FORWARD

```
#define O_FORWARD 0
```

Definition at line 1003 of file ftypes.h.

4.37.2.644 O_BACKWARD

```
#define O_BACKWARD 1
```

Definition at line 1004 of file ftypes.h.

4.37.2.645 O_FORWARDORBACKWARD

```
#define O_FORWARDORBACKWARD 2
```

Definition at line 1005 of file ftypes.h.

4.37.2.646 O_FORWARDANDBACKWARD

```
#define O_FORWARDANDBACKWARD 3
```

Definition at line 1006 of file ftypes.h.

4.37.2.647 OPTHEAD

```
#define OPTHEAD 3
```

Definition at line 1008 of file ftypes.h.

4.37.2.648 DOALL

```
#define DOALL 1
```

Definition at line 1009 of file ftypes.h.

4.37.2.649 ONLYFUNCTIONS

```
#define ONLYFUNCTIONS 2
```

Definition at line 1010 of file ftypes.h.

4.37.2.650 INUSE

```
#define INUSE 1
```

Definition at line 1012 of file ftypes.h.

4.37.2.651 COULDCOMMUTE

```
#define COULDCOMMUTE 2
```

Definition at line 1013 of file ftypes.h.

4.37.2.652 DOESNOTCOMMUTE

```
#define DOESNOTCOMMUTE 4
```

Definition at line 1014 of file ftypes.h.

4.37.2.653 DICT_NONUMBERS

```
#define DICT_NONUMBERS 0
```

Definition at line 1016 of file ftypes.h.

4.37.2.654 DICT_INTEGERONLY

```
#define DICT_INTEGERONLY 1
```

Definition at line 1017 of file ftypes.h.

4.37.2.655 DICT_RATIONALONLY

```
#define DICT_RATIONALONLY 2
```

Definition at line 1018 of file ftypes.h.

4.37.2.656 DICT_ALLNUMBERS

```
#define DICT_ALLNUMBERS 3
```

Definition at line 1019 of file ftypes.h.

4.37.2.657 DICT_NOVARIABLES

```
#define DICT_NOVARIABLES 0
```

Definition at line 1020 of file ftypes.h.

4.37.2.658 DICT_DOVARIABLES

```
#define DICT_DOVARIABLES 1
```

Definition at line 1021 of file ftypes.h.

4.37.2.659 DICT_NOSPECIALS

```
#define DICT_NOSPECIALS 0
```

Definition at line 1022 of file ftypes.h.

4.37.2.660 DICT_DOSPECIALS

```
#define DICT_DOSPECIALS 1
```

Definition at line 1023 of file ftypes.h.

4.37.2.661 DICT_NOFUNWITHARGS

```
#define DICT_NOFUNWITHARGS 0
```

Definition at line 1024 of file ftypes.h.

4.37.2.662 DICT_DOFUNWITHARGS

```
#define DICT_DOFUNWITHARGS 1
```

Definition at line 1025 of file ftypes.h.

4.37.2.663 DICT_NOTINDOLLARS

```
#define DICT_NOTINDOLLARS 0
```

Definition at line 1026 of file ftypes.h.

4.37.2.664 DICT_INDOLLARS

```
#define DICT_INDOLLARS 1
```

Definition at line 1027 of file ftypes.h.

4.37.2.665 DICT_INTEGERNUMBER

```
#define DICT_INTEGERNUMBER 1
```

Definition at line 1029 of file ftypes.h.

4.37.2.666 DICT_RATIONALNUMBER

```
#define DICT_RATIONALNUMBER 2
```

Definition at line 1030 of file ftypes.h.

4.37.2.667 DICT_SYMBOL

```
#define DICT_SYMBOL 3
```

Definition at line 1031 of file ftypes.h.

4.37.2.668 DICT_VECTOR

```
#define DICT_VECTOR 4
```

Definition at line 1032 of file ftypes.h.

4.37.2.669 DICT_INDEX

```
#define DICT_INDEX 5
```

Definition at line 1033 of file ftypes.h.

4.37.2.670 DICT_FUNCTION

```
#define DICT_FUNCTION 6
```

Definition at line 1034 of file ftypes.h.

4.37.2.671 DICT_FUNCTION_WITH_ARGUMENTS

```
#define DICT_FUNCTION_WITH_ARGUMENTS 7
```

Definition at line 1035 of file ftypes.h.

4.37.2.672 DICT_SPECIALCHARACTER

```
#define DICT_SPECIALCHARACTER 8
```

Definition at line 1036 of file ftypes.h.

4.37.2.673 DICT_RANGE

```
#define DICT_RANGE 9
```

Definition at line 1037 of file ftypes.h.

4.37.2.674 READSPECTATORFLAG

```
#define READSPECTATORFLAG 3
```

Definition at line 1039 of file ftypes.h.

4.37.2.675 GLOBALSPECTATORFLAG

```
#define GLOBALSPECTATORFLAG 1
```

Definition at line 1040 of file ftypes.h.

4.37.3 Typedef Documentation

4.37.3.1 PVFUNWP

```
typedef VOID(* PVFUNWP) ()
```

Definition at line 198 of file ftypes.h.

4.37.3.2 PVFUNV

```
typedef VOID(* PVFUNV) ()
```

Definition at line 199 of file ftypes.h.

4.37.3.3 CFUN

```
typedef int(* CFUN) ()
```

Definition at line 200 of file ftypes.h.

4.37.3.4 TFUN

```
typedef int(* TFUN) ()
```

Definition at line 201 of file ftypes.h.

4.37.3.5 TFUN1

```
typedef int(* TFUN1) ()
```

Definition at line 202 of file ftypes.h.

4.38 ftypes.h

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

#ifdef WITHPTHREADS
#define PHEAD  ALLPRIVATES *B,
#define PHEAD0 ALLPRIVATES *B
#define BHEAD  B,
#define BHEAD0 B
#else
#define PHEAD
#define PHEAD0 VOID
#define BHEAD
#define BHEAD0
#endif

#define WITHOUTERROR 0
#define WITHERROR 1

/*
    The various streams. (look also in tools.c)
*/

#define FILESTREAM 0
#define PREVARSTREAM 1
#define PREREADSTREAM 2
#define PIPESTREAM 3
#define PRECALCSTREAM 4
#define DOLLARSTREAM 5
#define PREREADSTREAM2 6
#define EXTERNALCHANNELSTREAM 7
#define PREREADSTREAM3 8
#define REVERSEFILESTREAM 9

#define ENDOFSTREAM 0xFF
#define ENDOFINPUT 0xFF

/*
    Types of files
*/

#define SUBROUTINEFILE 0
#define PROCEDUREFILE 1
#define HEADERFILE 2
#define SETUPFILE 3
#define TABLEBASEFILE 4

/*
    Types of modules
*/

#define FIRSTMODULE -1
#define GLOBALMODULE 0
#define SORTMODULE 1
#define STOREMODULE 2
#define CLEARMODULE 3
#define ENDMODULE 4
```

```
#define POLYFUN 0

#define NOPARALLEL_DOLLAR       0x0001
#define NOPARALLEL_RHS          0x0002
#define NOPARALLEL_CONVPOLY     0x0004
#define NOPARALLEL_SPECTATOR    0x0008
#define NOPARALLEL_USER         0x0010
#define NOPARALLEL_TBLDOLLAR    0x0100
#define NOPARALLEL_NPROC        0x0200
#define PARALLELFLAG            0x0000

#define PRENOACTION 0
#define PRERAISEAFTER 1
#define PRELOWERAFTER 2
/*
#define ELIUMOD 1
#define ELIZMOD 2
#define SKIUMOD 3
#define SKIZMOD 4
*/
#define WITHSEMICOLON 0
#define WITHOUTSEMICOLON 1
#define MODULEINSTACK 8
#define EXECUTINGIF 0
#define LOOKINGFORELSE 1
#define LOOKINGFORENDIF 2
#define NEWSTATEMENT 1
#define OLDSTATEMENT 0

#define EXECUTINGPRESWITCH 0
#define SEARCHINGPRECASE 1
#define SEARCHINGPREENDSWITCH 2

#define PREPROONLY 1
#define DUMPTOCOMPILER 2
#define DUMPOUTTERMS 4
#define DUMPINTERMS 8
#define DUMPTOSORT 16
#define DUMPTOPARALLEL 32
#define THREADSDEBUG 64

#define ERROROUT 0
#define INPUTOUT 1
#define STATSOUT 2
#define EXPRSOUT 3
#define WRITEOUT 4

#define EXTERNALCHANNELOUT 5

#define NUMERICALLOOP 0
#define LISTEDLOOP 1
#define ONEEXPRESSION 2

#define PRETYPENONE 0
#define PRETYPEIF 1
#define PRETYPEDO 2
#define PRETYPEPROCEDURE 3
#define PRETYPESWITCH 4
#define PRETYPEINSIDE 5

/*
    Type of statement. Used to make sure that the statements are in proper order
*/

#define DECLARATION   1
#define SPECIFICATION 2
#define DEFINITION    3
#define STATEMENT     4
#define TOOUTPUT      5
#define ATENDOFMODULE 6
#define MIXED         9

/*
    The typedefs are to allow the compilers to do better error checking.
*/

/*
    icc doesn't like the typedef void VOID; Neither does g++ on the apple
    Hence we work the old fashioned way:
*/
#define VOID void

#ifdef ANSI
typedef VOID (*PVFUNWP)(WORD *);
#ifdef INTELCOMPILER
typedef VOID (*PVFUNV)();
typedef int (*CFUN)();
```

```
#else
typedef VOID (*PVFUNV)(VOID);
typedef int (*CFUN)(VOID);
#endif
typedef int (*TFUN)(UBYTE *);
typedef int (*TFUN1)(UBYTE *,int);
#else
typedef VOID (*PVFUNWP)();
typedef VOID (*PVFUNV)();
typedef int (*CFUN)();
typedef int (*TFUN)();
typedef int (*TFUN1)();
#endif


#define NOAUTO 0
#define PARTEST 1
#define WITHAUTO 2

#define ALLVARIABLES -1
#define SYMBOLONLY 1
#define INDEXONLY 2
#define VECTORONLY 4
#define FUNCTIONONLY 8
#define SETONLY 16
#define EXPRESSIONONLY 32


#define CDELETE -1
#define ANYTYPE -1
#define CSYMBOL 0
#define CINDEX 1
#define CVECTOR 2
#define CFUNCTION 3
#define CSET 4
#define CEXPRESSION 5
#define CDOTPRODUCT 6
#define CNUMBER 7
#define CSUBEXP 8
#define CDELTA 9
#define CDOLLAR 10
#define CDUBIOUS 11
#define CRANGE 12
#define CVECTOR1 21
#define CDOUBLEDOT 22

/*
    Types of tokens in the tokenizer.
*/

#define TSYMBOL -1
#define TINDEX -2
#define TVECTOR -3
#define TFUNCTION -4
#define TSET -5
#define TEXPRESSION -6
#define TDOTPRODUCT -7
#define TNUMBER -8
#define TSUBEXP -9
#define TDELTA -10
#define TDOLLAR -11
#define TDUBIOUS -12
#define LPARENTHESIS -13
#define RPARENTHESIS -14
#define TWILDCARD -15
#define TWILDARG -16
#define TDOT -17
#define LBRACE -18
#define RBRACE -19
#define TCOMMA -20
#define TFUNOPEN -21
#define TFUNCLOSE -22
#define TMULTIPLY -23
#define TDIVIDE -24
#define TPOWER -25
#define TPLUS -26
#define TMINUS -27
#define TNOT -28
#define TENDOFIT -29
#define TSETOPEN -30
#define TSETCLOSE -31
#define TGENINDEX -32
#define TCONJUGATE -33
#define LRPARENTHESES -34
#define TNUMBER1 -35
#define TPOWER1 -36
```

```
#define TEMPTY -37
#define TSETNUM -38
#define TSGAMMA -39
#define TSETDOL -40

#define TYPEISFUN 0
#define TYPEISSUB 1
#define TYPEISMYSTERY -1

#define LHSIDEX 2
#define LHSIDE 1
#define RHSIDE 0

/*
    Output modes
*/

#define FORTRANMODE 1
#define REDUCEMODE 2
#define MAPLEMODE 3
#define MATHEMATICAMODE 4
#define CMODE 5
#define VORTRANMODE 6
#define PFORTRANMODE 7
#define DOUBLEFORTRANMODE 33
#define DOUBLEPRECISIONFLAG 32
#define NODOUBLEMASK 31
#define QUADRUPLEFORTRANMODE 65
#define QUADRUPLEPRECISIONFLAG 64
#define NOQUADMASK 63
#define NORMALFORMAT 0
#define NOSPACEFORMAT 1

#define ISNOTFORTRAN90 0
#define ISFORTRAN90 1

#define ALSOREVERSE 1
#define CHISHOLM 2
#define NOTRICK 16

#define SORTLOWFIRST 0
#define SORTHIGHFIRST 1
#define SORTPOWERFIRST 2
#define SORTANTIPOWER 3

#define NMIN4SHIFT 4
/*
    The next are the main codes.
    Note: SETSET is not allowed to be 4*n+1
    We use those codes in CoIdExpression for function information
    after the pattern. Because SETSET also stands there we have to
    be careful!!
    Don't forget MAXBUILTINFUNCTION when adding codes!
    The object FUNCTION is at the start of the functions that are in regular
    notation. Anything below it is in special notation.

    Remark: HAAKJE0 is for compression purposes and should only occur
    at moments that ARGWILD cannot occur.
*/
#define SYMBOL 1
#define DOTPRODUCT 2
#define VECTOR 3
#define INDEX 4
#define EXPRESSION 5
#define SUBEXPRESSION 6
#define DOLLAREXPRESSION 7
#define SETSET 8
#define ARGWILD 9
#define MINVECTOR 10
#define SETEXP 11
#define DOLLAREXPR2 12
#define HAAKJE0 9
#define FUNCTION 20

#define TMPPOLYFUN 14
#define ARGFIELD 15
#define SNUMBER 16
#define LNUMBER 17
#define HAAKJE 18
#define DELTA 19
#define EXPONENT 20
#define DENOMINATOR 21
#define SETFUNCTION 22
#define GAMMA 23
#define GAMMAI 24
#define GAMMAFIVE 25
#define GAMMASIX 26
```

```
#define GAMMASEVEN 27
#define SUMF1 28
#define SUMF2 29
#define DUMFUN 30
#define REPLACEMENT 31
#define REVERSEFUNCTION 32
#define DISTRIBUTION 33
#define DELTA3 34
#define DUMMYFUN 35
#define DUMMYTEN 36
#define LEVICIVITA 37
#define FACTORIAL 38
#define INVERSEFACTORIAL 39
#define BINOMIAL 40
#define NUMARGSFUN 41
#define SIGNFUN 42
#define MODFUNCTION 43
#define MOD2FUNCTION 44
#define MINFUNCTION 45
#define MAXFUNCTION 46
#define ABSFUNCTION 47
#define SIGFUNCTION 48
#define INTFUNCTION 49
#define THETA 50
#define THETA2 51
#define DELTA2 52
#define DELTAP 53
#define BERNOULLIFUNCTION 54
#define COUNTFUNCTION 55
#define MATCHFUNCTION 56
#define PATTERNFUNCTION 57
#define TERMFUNCTION 58
#define CONJUGATION 59
#define ROOTFUNCTION 60
#define TABLEFUNCTION 61
#define FIRSTBRACKET 62
#define TERMSINEXPR 63
#define NUMTERMSFUN 64
#define GCDFUNCTION 65
#define DIVFUNCTION 66
#define REMFUNCTION 67
#define MAXPOWEROF 68
#define MINPOWEROF 69
#define TABLESTUB 70
#define FACTORIN 71
#define TERMSINBRACKET 72
#define WILDARGFUN 73
/*
    In the past we would add new functions here and raise the numbers
    on the special reserved names. This is impractical in the light of
    the .sav files. The .sav files need a mechanism that contains the
    value of MAXBUILTINFUNCTION at the moment of writing. This allows
    form corrections if this value has changed in the mean time.
*/
#define SQRTFUNCTION 74
#define LNFUNCTION 75
#define SINFUNCTION 76
#define COSFUNCTION 77
#define TANFUNCTION 78
#define ASINFUNCTION 79
#define ACOSFUNCTION 80
#define ATANFUNCTION 81
#define ATAN2FUNCTION 82
#define SINHFUNCTION 83
#define COSHFUNCTION 84
#define TANHFUNCTION 85
#define ASINHFUNCTION 86
#define ACOSHFUNCTION 87
#define ATANHFUNCTION 88
#define LI2FUNCTION 89
#define LINFUNCTION 90

#define EXTRASYMFUN 91
#define RANDOMFUNCTION 92
#define RANPERM 93
#define NUMFACTORS 94
#define FIRSTTERM 95
#define CONTENTTERM 96
#define PRIMENUMBER 97
#define EXTEUCLIDEAN 98
#define MAKERATIONAL 99
#define INVERSEFUNCTION 100
#define IDFUNCTION 101
#define PUTFIRST 102
#define PERMUTATIONS 103
#define PARTITIONS 104
#define MULFUNCTION 105
```

```

#define MAXBUILTINFUNCTION 105
#define FIRSTUSERFUNCTION 150

/*
    Note: if we add a builtin table we have to look also inside names.c
    in the routine Globalize because there we assume there does not exist
    such an object
*/

#define ISYMBOL 0
#define PISYMBOL 1
#define COEFFSYMBOL 2
#define NUMERATORSYMBOL 3
#define DENOMINATORSYMBOL 4
#define WILDARGSYMBOL 5
#define DIMENSIONSYMBOL 6
#define FACTORSYMBOL 7
#define SEPARATESYMBOL 8

#define BUILTINSYMBOLS 9
#define FIRSTUSERSYMBOL 20

#define BUILTININDICES 1
#define BUILTINVECTORS 1
#define BUILTINDOLLARS 1

#define WILDARGVECTOR 0
#define WILDARGINDEX 0

/*
    The objects that have a name that starts with TYPE are codes of statements
    made by the compiler. Each statement starts with such a code, followed by
    its size. For how most of these statements are used can be seen in the
    Generator function in the file proces.c
    TYPEOPERATION is an anachronism that remains used only for the statements
    that are executed in the file opera.c (like traces and contractions).
*/

#define TYPEEXPRESSION 0
#define TYPEIDNEW 1
#define TYPEIDOLD 2
#define TYPEOPERATION 3
#define TYPEREPEAT 4
#define TYPEENDREPEAT 5
/*
    The next counts must be higher than the ones before
*/
#define TYPECOUNT 20
#define TYPEMULT 21
#define TYPEGOTO 22
#define TYPEDISCARD 23
#define TYPEIF 24
#define TYPEELSE 25
#define TYPEELIF 26
#define TYPEENDIF 27
#define TYPESUM 28
#define TYPECHISHOLM 29
#define TYPEREVERSE 30
#define TYPEARG 31
#define TYPENORM 32
#define TYPENORM2 33
#define TYPENORM3 34
#define TYPEEXIT 35
#define TYPESETEXIT 36
#define TYPEPRINT 37
#define TYPEFPRINT 38
#define TYPEREDEFPRE 39
#define TYPESPLITARG 40
#define TYPESPLITARG2 41
#define TYPEFACTARG 42
#define TYPEFACTARG2 43
#define TYPETRY 44
#define TYPEASSIGN 45
#define TYPERENUMBER 46
#define TYPESUMFIX 47
#define TYPEFINDLOOP 48
#define TYPEUNRAVEL 49
#define TYPEADJUSTBOUNDS 50
#define TYPEINSIDE 51
#define TYPETERM 52
#define TYPESORT 53
#define TYPEDETCURDUM 54
#define TYPEINEXPRESSION 55
#define TYPESPLITFIRSTARG 56
#define TYPESPLITLASTARG 57
#define TYPEMERGE 58
```

```
#define TYPETESTUSE 59
#define TYPEAPPLY 60
#define TYPEAPPLYRESET 61
#define TYPECHAININ 62
#define TYPECHAINOUT 63
#define TYPENORM4 64
#define TYPEFACTOR 65
#define TYPEARGIMPLODE 66
#define TYPEARGEXPLODE 67
#define TYPEDENOMINATORS 68
#define TYPESTUFFLE 69
#define TYPEDROPCOEFFICIENT 70
#define TYPETRANSFORM 71
#define TYPETOPOLYNOMIAL 72
#define TYPEFROMPOLYNOMIAL 73
#define TYPEDOLOOP 74
#define TYPEENDDOLOOP 75
#define TYPEDROPSYMBOLS 76
#define TYPEPUTINSIDE 77
#define TYPETOSPECTATOR 78
#define TYPEARGTOEXTRASYMBOL 79
/*
    The codes for the 'operations' that are part of TYPEOPERATION.
*/

#define TAKETRACE 1
#define CONTRACT 2
#define RATIO 3
#define SYMMETRIZE 4
#define TENVEC 5
#define SUMNUM1 6
#define SUMNUM2 7

/*
    The various types of wildcards.
*/

#define WILDDUMMY 0
#define SYMTONUM 1
#define SYMTOSYM 2
#define SYMTOSUB 3
#define VECTOMIN 4
#define VECTOVEC 5
#define VECTOSUB 6
#define INDTOIND 7
#define INDTOSUB 8
#define FUNTOFUN 9
#define ARGTOARG 10
#define ARLTOARL 11
#define EXPTOEXP 12
#define FROMBRAC 13
#define FROMSET 14
#define SETTONUM 15
#define WILDCARDS 16
#define SETNUMBER 17
#define LOADDOLLAR 18
/*
    Some new types of wildcards that hold only for function arguments.
*/
#define NUMTONUM 20
#define NUMTOSYM 21
#define NUMTOIND 22
#define NUMTOSUB 23

/*
    Dirty flags (introduced when functions got a field with a dirty flag)
*/

#define CLEANFLAG 0
#define DIRTYFLAG 1
#define DIRTYSYMFLAG 2
#define MUSTCLEANPRF 4
#define ALLDIRTY (DIRTYFLAG|DIRTYSYMFLAG)

#define ARGHEAD 2
#define FUNHEAD 3
#define SUBEXPSIZE 5
#define EXPRHEAD 5
#define TYPEARGHEADSIZE 6

/*
    Actions to be taken with expressions. They are marked with these objects
    during compilation.
*/

#define SKIP 1
#define DROP 2
```

```
#define HIDE 3
#define UNHIDE 4
#define INTOHIDE 5

/*
    Types of expressions
*/

#define LOCALEXPRESSION 0
#define SKIPLEXPRESSION 1
#define DROPLEXPRESSION 2
#define DROPPEDEXPRESSION 3
#define GLOBALEXPRESSION 4
#define SKIPGEXPRESSION 5
#define DROPGEXPRESSION 6
#define STOREDEXPRESSION 8
#define HIDDENLEXPRESSION 9
#define HIDDENGEXPRESSION 13
#define INCEXPRESSION 9
#define HIDELEXPRESSION 10
#define HIDEGEXPRESSION 14
#define DROPHLEXPRESSION 11
#define DROPHGEXPRESSION 15
#define UNHIDELEXPRESSION 12
#define UNHIDEGEXPRESSION 16
#define INTOHIDELEXPRESSION 17
#define INTOHIDEGEXPRESSION 18
#define SPECTATOREXPRESSION 19
#define DROPSPECTATOREXPRESSION 20
#define SKIPUNHIDELEXPRESSION 21
#define SKIPUNHIDEGEXPRESSION 22

#define PRINTOFF 0
#define PRINTON 1
#define PRINTCONTENTS 2
#define PRINTCONTENT 3
#define PRINTLFILE 4
#define PRINTONETERM 8
#define PRINTONEFUNCTION 16
#define PRINTALL 32

/*
    Special codes for the replace variable in the EXPRESSIONS struct
*/

#define REGULAREXPRESSION -1
#define REDEFINEDEXPRESSION -2
#define NEWLYDEFINEDEXPRESSION -3

#define GENERALFUNCTION 0
#define FASTFUNCTION 1
#define TENSORFUNCTION 2
#define GAMMAFUNCTION 4
/*
    Special sets
*/

#define POS_         0   /* integer > 0 */
#define POS0_        1   /* integer >= 0 */
#define NEG_         2   /* integer < 0 */
#define NEG0_        3   /* integer <= 0 */
#define EVEN_        4   /* integer (even) */
#define ODD_         5   /* integer (odd) */
#define Z_           6   /* integer */
#define SYMBOL_      7   /* symbol only */
#define FIXED_       8   /* fixed index */
#define INDEX_       9   /* index only */
#define Q_          10   /* rational */
#define DUMMYINDEX_ 11   /* dummy index only */
#define VECTOR_     12   /* vector only */

/*
    Special indices.
*/

#define GAMMA1 0
#define GAMMA5 -1
#define GAMMA6 -2
#define GAMMA7 -3
#define FUNNYVEC -4
#define FUNNYWILD -5
#define SUMMEDIND -6
#define NOINDEX -7
#define FUNNYDOLLAR -8
#define EMPTYINDEX -9

/*
```

```
    The next one should be less than all of the above special indices.
*/

#define MINSPEC -10

#define USEDFLAG 2
#define DUMMYFLAG 1

#define MAINSORT 0
#define FUNCTIONSORT 1
#define SUBSORT 2

#define FLOATMODE 1
#define RATIONALMODE 0

#define NUMSPECSETS 10
#define EATTENSOR 0x2000

#define ISZERO 1
#define ISUNMODIFIED 2
#define ISCOMPRESSED 4
#define ISINRHS 8
#define ISFACTORIZED 16
#define TOBEFACTORED 32
#define TOBEUNFACTORED 64
#define KEEPZERO 128

#define VARTYPENONE 0
#define VARTYPECOMPLEX 1
#define VARTYPEIMAGINARY 2
#define VARTYPEROOTOFUNITY 4
#define VARTYPEMINUS 8
#define CYCLESYMMETRIC 1
#define RCYCLESYMMETRIC 2
#define SYMMETRIC 3
#define ANTISYMMETRIC 4
#define REVERSEORDER 256

/*
    Types of id statements (substitutions)
*/

#define SUBMULTI 1
#define SUBONCE 2
#define SUBONLY 3
#define SUBMANY 4
#define SUBVECTOR 5
#define SUBSELECT 6
#define SUBALL 7
#define SUBMASK 15
#define SUBDISORDER 16
#define SUBAFTER 32
#define SUBAFTERNOT 64

#define IDHEAD 6

#define DOLLARFLAG 1
#define NORMALIZEFLAG 2

#define GIDENT 1
#define GFIVE 4
#define GPLUS 3
#define GMINUS 2

/*
    Types of objects inside an if clause.
*/

#define LONGNUMBER 1
#define MATCH 2
#define COEFFI 3
#define SUBEXPR 4
#define MULTIPLEOF 5
#define IFDOLLAR 6
#define IFEXPRESSION 7
#define IFDOLLAREXTRA 8
#define IFISFACTORIZED 9
#define IFOCCURS 10
#define GREATER 0
#define GREATEREQUAL 1
#define LESS 2
#define LESSEQUAL 3
#define EQUAL 4
#define NOTEQUAL 5
#define ORCOND 6
#define ANDCOND 7
#define DUMMY 1
```

```
#define SORT 1
#define STORE 2
#define END 3
#define GLOBAL 4
#define CLEAR 5

#define VECTBIT 1
#define DOTPBIT 2
#define FUNBIT  4
#define SETBIT  8

#define EXTRAPARAMETER 0x4000
#define GENCOMMUTE 0
#define GENNONCOMMUTE 0x2000

#define NAMENOTFOUND -9

/*
    Types of dollar expressions.
*/

#define DOLUNDEFINED 0
#define DOLNUMBER 1
#define DOLARGUMENT 2
#define DOLSUBTERM 3
#define DOLTERMS 4
#define DOLWILDARGS 5
#define DOLINDEX 6
#define DOLZERO 7

#define FINDLOOP 0
#define REPLACELOOP 1

#define NOFUNPOWERS 0
#define COMFUNPOWERS 1
#define ALLFUNPOWERS 2

#define PROPERORDERFLAG 0

#define REGULAR 0
#define FINISH 1

#define POLYADD 1
#define POLYSUB 2
#define POLYMUL 3
#define POLYDIV 4
#define POLYREM 5
#define POLYGCD 6
#define POLYINTFAC 7
#define POLYNORM 8

#define MODNONE 0
#define MODSUM 1
#define MODMAX 2
#define MODMIN 3
#define MODLOCAL 4

#define ELEMENTUSED 1
#define ELEMENTLOADED 2
/*
    Variables for the modulus statement, flags in AC.modmode
    For explanation, see CoModulus
*/
#define POSNEG 0x1
#define INVERSETABLE 0x2
#define COEFFICIENTSONLY 0x4
#define ALSOPOWERS 0x8
#define ALSOFUNARGS 0x10
#define ALSODOLLARS 0x20
#define NOINVERSES 0x40

#define POSITIVEONLY 0
#define UNPACK 0x80
#define NOUNPACK 0
#define FROMFUNCTION 0x100

#define VARNAMES 0
#define AUTONAMES 1
#define EXPRNAMES 2
#define DOLLARNAMES 3

#ifdef WITHPTHREADS
/*
    Signals that the workers have to react to
*/

#define TERMINATETHREAD -1
```

```
#define STARTNEWEXPRESSION 1
#define LOWESTLEVELGENERATION 2
#define FINISHEXPRESSION 3
#define CLEANUPEXPRESSION 4
#define HIGHERLEVELGENERATION 5
#define STARTNEWMODULE 6
#define CLAIMOUTPUT 7
#define FINISHEXPRESSION2 8
#define INISORTBOT 7
#define RUNSORTBOT 8
#define DOONEEXPRESSION 9
#define DOBRACKETS 10
#define CLEARCLOCK 11
#define MCTSEXPANDTREE 12
#define OPTIMIZEEXPRESSION 13

#define MASTERBUFFERISFULL 1

/*
    Bucket states
*/

#define BUCKETFREE 1
#define BUCKETINUSE 0
#define BUCKETCOMINGFREE 2
#define BUCKETFILLED -1
#define BUCKETATEND -2
#define BUCKETTERMINATED 3
#define BUCKETRELEASED 4

#define NUMBEROFBLOCKSINSORT 10
#define MINIMUMNUMBEROFTERMS 10

#define BUCKETDOINGTERM 1
#define BUCKETASSIGNED -1
#define BUCKETTOBERELEASED -2
#define BUCKETPREPARINGTERM 0

#define BUCKETDOINGTERMS 0
#define BUCKETDOINGBRACKET 1
#endif

/*
    The next variable is because there is some use of cbufnum that is
    probably irrelevant. We use here DUMMYBUFNUM instead of AC.cbufnum
    just in case we run into trouble later.
*/
#define DUMMYBUFFER 1

#define ALLARGS      1
#define NUMARG       2
#define ARGRANGE     3
#define MAKEARGS     4
#define MAXRANGEINDICATOR 4
#define REPLACEARG   5
#define ENCODEARG    6
#define DECODEARG    7
#define IMPLODEARG   8
#define EXPLODEARG   9
#define PERMUTEARG  10
#define REVERSEARG  11
#define CYCLEARG    12
#define ISLYNDON    13
#define ISLYNDONR   14
#define TOLYNDON    15
#define TOLYNDONR   16
#define ADDARG      17
#define MULTIPLYARG 18
#define DROPARG     19
#define SELECTARG   20
#define DEDUPARG    21

#define BASECODE 1
#define YESLYNDON 1
#define NOLYNDON 2

#define TOPOLYNOMIALFLAG 1
#define FACTARGFLAG 2

#define OLDFACTARG 1
#define NEWFACTARG 0

#define FROMMODULEOPTION 0
#define FROMPOINTINSTRUCTION 1

#define EXTRASYMBOL 0
#define REGULARSYMBOL 1
```

```
#define EXPRESSIONNUMBER 2

#define O_NONE 0
#define O_CSE 1
#define O_CSEGREEDY 2
#define O_GREEDY 3

#define O_OCCURRENCE 0
#define O_MCTS 1
#define O_SIMULATED_ANNEALING 2

#define O_FORWARD 0
#define O_BACKWARD 1
#define O_FORWARDORBACKWARD 2
#define O_FORWARDANDBACKWARD 3

#define OPTHEAD 3
#define DOALL 1
#define ONLYFUNCTIONS 2

#define INUSE 1
#define COULDCOMMUTE 2
#define DOESNOTCOMMUTE 4

#define DICT_NONUMBERS 0
#define DICT_INTEGERONLY 1
#define DICT_RATIONALONLY 2
#define DICT_ALLNUMBERS 3
#define DICT_NOVARIABLES 0
#define DICT_DOVARIABLES 1
#define DICT_NOSPECIALS 0
#define DICT_DOSPECIALS 1
#define DICT_NOFUNWITHARGS 0
#define DICT_DOFUNWITHARGS 1
#define DICT_NOTINDOLLARS 0
#define DICT_INDOLLARS 1

#define DICT_INTEGERNUMBER 1
#define DICT_RATIONALNUMBER 2
#define DICT_SYMBOL 3
#define DICT_VECTOR 4
#define DICT_INDEX 5
#define DICT_FUNCTION 6
#define DICT_FUNCTION_WITH_ARGUMENTS 7
#define DICT_SPECIALCHARACTER 8
#define DICT_RANGE 9

#define READSPECTATORFLAG 3
#define GLOBALSPECTATORFLAG 1
```

4.39 function.c File Reference

`#include "form3.h"`

Include dependency graph for function.c:

Functions

- WORD MakeDirty (WORD *term, WORD *x, WORD par)

- void MarkDirty (WORD ∗term, WORD flags)
- void PolyFunDirty (PHEAD WORD ∗term)
- void PolyFunClean (PHEAD WORD ∗term)
- WORD Symmetrize (PHEAD WORD ∗func, WORD ∗Lijst, WORD ngroups, WORD gsize, WORD type)
- WORD CompGroup (PHEAD WORD type, WORD ∗∗args, WORD ∗a1, WORD ∗a2, WORD num)
- int FullSymmetrize (PHEAD WORD ∗fun, int type)
- WORD SymGen (PHEAD WORD ∗term, WORD ∗params, WORD num, WORD level)
- WORD SymFind (PHEAD WORD ∗term, WORD ∗params)
- int ChainIn (PHEAD WORD ∗term, WORD funnum)
- int ChainOut (PHEAD WORD ∗term, WORD funnum)
- WORD MatchFunction (PHEAD WORD ∗pattern, WORD ∗interm, WORD ∗wilds)
- WORD ScanFunctions (PHEAD WORD ∗inpat, WORD ∗inter, WORD par)

4.39.1 Detailed Description

The file with the central routines for the pattern matching of functions and their arguments. The file also contains the routines for the execution of the Symmetrize statement and its variations (like antisymmetrize etc).

Definition in file function.c.

4.39.2 Function Documentation

4.39.2.1 MakeDirty()

```
WORD MakeDirty (
            WORD * term,
            WORD * x,
            WORD par )
```

Definition at line 51 of file function.c.

4.39.2.2 MarkDirty()

```
void MarkDirty (
            WORD * term,
            WORD flags )
```

Definition at line 97 of file function.c.

4.39.2.3 PolyFunDirty()

```
void PolyFunDirty (
            PHEAD WORD * term )
```

Definition at line 139 of file function.c.

4.39.2.4 PolyFunClean()

```
void PolyFunClean (
            PHEAD WORD * term )
```

Definition at line 174 of file function.c.

4.39.2.5 Symmetrize()

```
WORD Symmetrize (
            PHEAD WORD * func,
            WORD * Lijst,
            WORD ngroups,
            WORD gsize,
            WORD type )
```

Definition at line 213 of file function.c.

4.39.2.6 CompGroup()

```
WORD CompGroup (
            PHEAD WORD type,
            WORD ** args,
            WORD * a1,
            WORD * a2,
            WORD num )
```

Definition at line 373 of file function.c.

4.39.2.7 FullSymmetrize()

```
int FullSymmetrize (
            PHEAD WORD * fun,
            int type )
```

Definition at line 473 of file function.c.

4.39.2.8 SymGen()

```
WORD SymGen (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )
```

Definition at line 518 of file function.c.

4.39.2.9 SymFind()

```
WORD SymFind (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 633 of file function.c.

4.39.2.10 ChainIn()

```
int ChainIn (
            PHEAD WORD * term,
            WORD funnum )
```

Definition at line 691 of file function.c.

4.39.2.11 ChainOut()

```
int ChainOut (
            PHEAD WORD * term,
            WORD funnum )
```

Definition at line 739 of file function.c.

4.39.2.12 MatchFunction()

```
WORD MatchFunction (
            PHEAD WORD * pattern,
            WORD * interm,
            WORD * wilds )
```

Definition at line 817 of file function.c.

4.39.2.13 ScanFunctions()

```
WORD ScanFunctions (
            PHEAD WORD * inpat,
            WORD * inter,
            WORD par )
```

Definition at line 1604 of file function.c.

4.40 function.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes : function.c
*/

#include "form3.h"

/*
    #] Includes :
    #[ Utilities :
        #[ MakeDirty :

        Routine finds the function with the address x in it
        and mark all arguments that contain x as dirty.
        if par == 0 term is a full term, else term is the start of a
        function
*/

WORD MakeDirty(WORD *term, WORD *x, WORD par)
{
    WORD *next, *n;
    if ( !par ) {
        next = term; next += *term;
        next -= ABS(next[-1]);
        term++;
        if ( x < term ) return(0);
        if ( x >= next ) return(0);
        while ( term < next ) {
            n = term + term[1];
            if ( x < n ) break;
            term = n;
        }
/*      next = n; */
    }
    else {
        next = term + term[1];
        if ( x < term || x >= next ) return(0);
    }
    if ( *term < FUNCTION ) return(0);
    if ( functions[*term-FUNCTION].spec >= TENSORFUNCTION ) return(0);
    term += FUNHEAD;
    if ( x < term ) return(0);
    next = term; NEXTARG(next)
    while ( x >= next ) { term = next; NEXTARG(next) }
    if ( *term < 0 ) return(0);
    term[1] = 1;
    term += ARGHEAD;
    if ( x < term ) return(1);
    next = term + *term;
    while ( x >= next ) { term = next; next += *next; }
    MakeDirty(term,x,0);
    return(1);
}

/*
        #] MakeDirty :
```

```
        #[ MarkDirty :

        Routine marks all functions dirty with the given flags.
        Is to be used when there is a possibility that symmetrization
        properties of functions may have changed. In that case we play
        it safe.
*/

void MarkDirty(WORD *term, WORD flags)
{
    WORD *t, *r, *m, *tstop;
    GETSTOP(term,tstop);
    t = term+1;
    while ( t < tstop ) {
        if ( *t < FUNCTION ) { t += t[1]; continue; }
        t[2] |= flags;
        if ( *t < FUNCTION+WILDOFFSET && functions[*t-FUNCTION].spec > 0 ) {
            t += t[1]; continue;
        }
        if ( *t >= FUNCTION+WILDOFFSET && functions[*t-FUNCTION-WILDOFFSET].spec > 0 ) {
            t += t[1]; continue;
        }
        r = t + FUNHEAD;
        t += t[1];
        while ( r < t ) {
            if ( *r <= 0 ) {
                if ( *r <= -FUNCTION ) r++;
                else r += 2;
                continue;
            }
            r[1] |= DIRTYFLAG;
            m = r + ARGHEAD;
            r += *r;
            while ( m < r ) {
                MarkDirty(m,flags);
                m += *m;
            }
        }
    }
}

/*
        #] MarkDirty :
        #[ PolyFunDirty :

        Routine marks the PolyFun or the PolyRatFun dirty.
        This is used when there is modular calculus and the modulus
        has changed for the current module.
*/

void PolyFunDirty(PHEAD WORD *term)
{
    GETBIDENTITY
    WORD *t, *tstop, *endarg;
    tstop = term + *term;
    tstop -= ABS(tstop[-1]);
    t = term+1;
    while ( t < tstop ) {
        if ( *t == AR.PolyFun ) {
            if ( AR.PolyFunType == 2 ) t[2] |= MUSTCLEANPRF;
            endarg = t + t[1];
            t[2] |= DIRTYFLAG;
            t += FUNHEAD;
            while ( t < endarg ) {
                if ( *t > 0 ) {
                    t[1] |= DIRTYFLAG;
                }
                NEXTARG(t);
            }
        }
        else {
            t += t[1];
        }
    }
}

/*
        #] PolyFunDirty :
        #[ PolyFunClean :

        Routine marks the PolyFun or the PolyRatFun clean.
        This is used when there is modular calculus and the modulus
        has changed for the current module.
*/

void PolyFunClean(PHEAD WORD *term)
{
```

```
    GETBIDENTITY
    WORD *t, *tstop;
    tstop = term + *term;
    tstop -= ABS(tstop[-1]);
    t = term+1;
    while ( t < tstop ) {
        if ( *t == AR.PolyFun ) {
            t[2] &= ~MUSTCLEANPRF;
        }
        t += t[1];
    }
}

/*
        #] PolyFunClean :
        #[ Symmetrize :

        (Anti)Symmetrizes the arguments of a function.
        Nlist tells of how many arguments are involved.
        Nlist == 0      All arguments must be sorted.
        Nlist > 0       Arguments mentioned are to be sorted, rest skipped.
        type = SYMMETRIC       Full symmetrization
        type = ANTISYMMETRIC:  Full symmetrization
        type = CYCLESYMMETRIC: Cyclic
        type = RCYCLESYMMETRIC:Cyclic or reverse
        Return value: OR of:
            0 even, 1 odd
            2 equal groups
            4 there was a permutation.

        The information in Lijst tells what grouping is to be applied.
        The information is:
        ngroups number of groups
        gsize size of groups
        Lijst[0]....  The groups.
*/

WORD Symmetrize(PHEAD WORD *func, WORD *Lijst, WORD ngroups, WORD gsize,
                WORD type)
{
    GETBIDENTITY
    WORD **args,**arg,nargs;
    WORD *to, *r, *fstop;
    WORD i, j, k, ff, exch, nexch, neq;
    WORD *a1, *a2, *a3;
    WORD reverseorder;
    if ( ( type & REVERSEORDER ) != 0 ) reverseorder = -1;
    else                                reverseorder = 1;
    type &= ~REVERSEORDER;

    ff = ( *func > FUNCTION ) ? functions[*func-FUNCTION].spec: 0;

    if ( 2*func[1] > AN.arglistsize ) {
        if ( AN.arglist ) M_free(AN.arglist,"Symmetrize");
        AN.arglistsize = 2*func[1] + 8;
        AN.arglist = (WORD **)Malloc1(AN.arglistsize*sizeof(WORD *),"Symmetrize");
    }
    arg = args = AN.arglist;
    to = AT.WorkPointer;
    r = func;
    fstop = r + r[1];
    r += FUNHEAD;
    nargs = 0;
    while ( r < fstop ) {   /* Make list of arguments */
        *arg++ = r;
        nargs++;
        if ( ff ) {
            if ( *r == FUNNYWILD ) r++;
            r++;
        }
        else { NEXTARG(r); }
    }
    exch = 0;
    nexch = 0;
    neq = 0;
    a1 = Lijst;
    if ( type == SYMMETRIC || type == ANTISYMMETRIC ) {
    for ( i = 1; i < ngroups; i++ ) {
        a3 = a2 = a1 + gsize;
        k = reverseorder*CompGroup(BHEAD ff,args,a1,a2,gsize);
        if ( k < 0 ) {
            j = i-1;
            for(;;) {
                for ( k = 0; k < gsize; k++ ) {
                    r = args[a1[k]]; args[a1[k]] = args[a2[k]]; args[a2[k]] = r;
                }
                exch ^= 1;
```

```
                nexch = 4;
                if ( j <= 0 ) break;
                a1 -= gsize;
                a2 -= gsize;
                k = reverseorder*CompGroup(BHEAD ff,args,a1,a2,gsize);
                if ( k == 0 ) neq = 2;
                if ( k >= 0 ) break;
                j--;
            }
        }
        else if ( k == 0 ) neq = 2;
        a1 = a3;
    }
    }
    else if ( type == CYCLESYMMETRIC || type == RCYCLESYMMETRIC ) {
        WORD rev = 0, jmin = 0, ii, iimin;
recycle:
        for ( j = 1; j < ngroups; j++ ) {
            for ( i = 0; i < ngroups; i++ ) {
                iimin = jmin + i;
                if ( iimin >= ngroups ) iimin -= ngroups;
                ii = j + i;
                if ( ii >= ngroups ) ii -= ngroups;
                k = reverseorder*CompGroup(BHEAD ff,args,Lijst+gsize*iimin,Lijst+gsize*ii,gsize);
                if ( k > 0 ) break;
                if ( k < 0 ) { jmin = j; nexch = 4; break; }
            }
        }
        if ( type == RCYCLESYMMETRIC && rev == 0 && ngroups > 1 ) {
            for ( j = 0; j < ngroups; j++ ) {
                for ( i = 0; i < ngroups; i++ ) {
                    iimin = jmin + i;
                    if ( iimin >= ngroups ) iimin -= ngroups;
                    ii = j - i;
                    if ( ii < 0 ) ii += ngroups;
                    k = reverseorder*CompGroup(BHEAD ff,args,Lijst+gsize*iimin,Lijst+gsize*ii,gsize);
                    if ( k > 0 ) break;
                    if ( k < 0 ) {
                        nexch = 4;
                        jmin = 0;
                        a1 = Lijst;
                        a2 = Lijst + gsize * (ngroups-1);
                        while ( a2 > a1 ) {
                            for ( k = 0; k < gsize; k++ ) {
                                r = args[a1[k]];
                                args[a1[k]] = args[a2[k]];
                                args[a2[k]] = r;
                            }
                            a1 += gsize; a2 -= gsize;
                        }
                        rev = 1;
                        goto recycle;
                    }
                }
            }
        }
        if ( jmin != 0 ) {
            arg = AN.arglist + func[1];
            a1 = Lijst + gsize * jmin;
            k = gsize * ngroups;
            a2 = Lijst + k;
            for ( i = 0; i < k; i++ ) {
                if ( a1 >= a2 ) a1 = Lijst;
                *arg++ = args[*a1++];
            }
            arg = AN.arglist + func[1];
            a1 = Lijst;
            for ( i = 0; i < k; i++ ) args[*a1++] = *arg++;
        }
    }
    r = func;
    i = FUNHEAD;
    NCOPY(to,r,i);
    for ( i = 0; i < nargs; i++ ) {
        if ( ff ) {
            if ( *(args[i]) == FUNNYWILD ) {
                *to++ = *(args[i]);
                *to++ = args[i][1];
            }
            else *to++ = *(args[i]);
        }
        else if ( ( j = *args[i] ) < 0 ) {
            *to++ = j;
            if ( j > -FUNCTION ) *to++ = args[i][1];
        }
        else {
            r = args[i];
```

```
            NCOPY(to,r,j);
        }
    }
    i = func[1];
    to = func;
    r = AT.WorkPointer;
    NCOPY(to,r,i);
    return ( exch | nexch | neq );
}

/*
        #] Symmetrize :
        #[ CompGroup :

            Routine compares two groups of arguments
            The arguments are in args[a1[i]] and args[a2[i]]
            for i = 0 to num
            type indicates the type of function.
            return value: -1 if there should be an exchange
            0 if they are equal
            1 if they are OK.
*/

WORD CompGroup(PHEAD WORD type, WORD **args, WORD *a1, WORD *a2, WORD num)
{
    GETBIDENTITY
    WORD *t1, *t2, i1, i2, n, k;

    for ( n = 0; n < num; n++ ) {
        t1 = args[a1[n]]; t2 = args[a2[n]];
        if ( type >= TENSORFUNCTION ) {
            if ( AR.Eside == LHSIDE || AR.Eside == LHSIDEX ) {
                if ( *t1 == FUNNYWILD ) {
                    if ( *t2 == FUNNYWILD ) {
                        if ( t1[1] < t2[1] ) return(1);
                        if ( t1[1] > t2[1] ) return(-1);
                    }
                    return(-1);
                }
                else if ( *t2 == FUNNYWILD ) {
                    return(1);
                }
                else {
                    if ( *t1 < *t2 ) return(1);
                    if ( *t1 > *t2 ) return(-1);
                }
            }
            else {
                if ( *t1 < *t2 ) return(1);
                if ( *t1 > *t2 ) return(-1);
            }
        }
        else if ( type == 0 ) {
            if ( AC.properorderflag ) {
                k = CompArg(t1,t2);
                if ( k < 0 ) return(1);
                if ( k > 0 ) return(-1);
                NEXTARG(t1)
                NEXTARG(t2)
            }
            else {
                if ( *t1 > 0 ) {
                    i1 = *t1 - ARGHEAD - 1;
                    t1 += ARGHEAD + 1;
                    if ( *t2 > 0 ) {
                        i2 = *t2 - ARGHEAD - 1;
                        t2 += ARGHEAD + 1;
                        while ( i1 > 0 && i2 > 0 ) {
                            if ( *t1 > *t2 ) return(-1);
                            else if ( *t1 < *t2 ) return(1);
                            i1--; i2--; t1++; t2++;
                        }
                        if ( i1 > 0 ) return(-1);
                        else if ( i2 > 0 ) return(1);
                    }
/*
                    This seems to be a bug. Reported by Aneesh Monahar, 28-sep-2005
                    else return(1);
*/
                    else return(-1);
                }
                else if ( *t2 > 0 ) return(1);
                else {
                    if ( *t1 != *t2 ) {
                        if ( *t1 <= -FUNCTION && *t2 <= -FUNCTION ) {
                            if ( *t1 < *t2 ) return(-1);
                            return(1);
```

```
                        }
                        else {
                            if ( *t1 < *t2 ) return(1);
                            return(-1);
                        }
                    }
                    if ( *t1 > -FUNCTION ) {
                        if ( t1[1] != t2[1] ) {
                            if ( t1[1] < t2[1] ) return(1);
                            return(-1);
                        }
                    }
                }
            }
        }
    }
    return(0);
}

/*
        #] CompGroup :
        #[ FullSymmetrize :

        Relay function for Normalize to execute a full symmetrization
        of a function fun. It hooks into Symmetrize according to the
        calling conventions for it.
        type = 0: Symmetrize
        type = 1: AntiSymmetrize
        type = 2: CycleSymmetrize
        type = 3: RCycleSymmetrize
        Return values:
        bit 0: odd permutation
        bit 1: identical arguments
        bit 2: there was a permutation.
*/

int FullSymmetrize(PHEAD WORD *fun, int type)
{
    GETBIDENTITY
    WORD *Lijst, count = 0;
    WORD *t, *funstop, i;
    int retval;

    if ( functions[*fun-FUNCTION].spec > 0 ) {
        count = fun[1] - FUNHEAD;
        for ( i = fun[1]-1; i >= FUNHEAD; i-- ) {
            if ( fun[i] == FUNNYWILD ) count--;
        }
    }
    else {
        funstop = fun + fun[1];
        t = fun + FUNHEAD;
        while ( t < funstop ) { count++; NEXTARG(t) }
    }
    if ( count < 2 ) {
        fun[2] &= ~DIRTYSYMFLAG;
        return(0);
    }
    Lijst = AT.WorkPointer;
    for ( i = 0; i < count; i++ ) Lijst[i] = i;
    AT.WorkPointer += count;
    retval = Symmetrize(BHEAD fun,Lijst,count,1,type);
    fun[2] &= ~DIRTYSYMFLAG;
    AT.WorkPointer = Lijst;
    return(retval);
}

/*
        #] FullSymmetrize :
        #[ SymGen :

        Routine does the outer work in the symmetrization.
        It locates the function(s) and loads up the parameters.
        It also studies the result.

        if params[4] = -1 and no extra -> all
                                extra -> strip groups with elements too large
                        0  -> if group with element too large: nofun
                        >0 -> must have right number of arguments
*/

WORD SymGen(PHEAD WORD *term, WORD *params, WORD num, WORD level)
{
    GETBIDENTITY
    WORD *t, *r, *m;
    WORD i, j, k, c1, c2, ngroup;
    WORD *rstop, Nlist, *inLijst, *Lijst, sign = 1, sumch = 0, count;
```

```
    DUMMYUSE(num);
    c1 = params[3];     /* function number */
    c2 = FUNCTION + WILDOFFSET;
    Nlist = params[4];
    if ( Nlist < 0 ) Nlist = 0;
    else Nlist = params[0] - 7;
    t = term;
    m = t + *t;
    m -= ABS(m[-1]);
    t++;
    while ( t < m ) {
        if ( *t == c1 || c1 > c2 ) {    /* Candidate function */
            if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
            >= TENSORFUNCTION ) {
                count = t[1] - FUNHEAD;
            }
            else {
                count = 0;
                r = t;
                rstop = t + t[1];
                r += FUNHEAD;
                while ( r < rstop ) { count++; NEXTARG(r) }
            }
            if ( ( j = params[4] ) > 0 && j != count ) goto NextFun;
            if ( j == 0 ) {
                inLijst = params+7;
                for ( i = 0; i < Nlist; i++ )
                    if ( inLijst[i] > count-1 ) goto NextFun;
            }

            if ( Nlist > (params[0] - 7) ) Nlist = params[0] - 7;
            Lijst = AT.WorkPointer;
            inLijst = params + 7;
            ngroup = params[5];
            if ( Nlist > 0 && j < 0 ) {
                k = 0;
                for ( i = 0; i < ngroup; i++ ) {
                    for ( j = 0; j < params[6]; j++ ) {
                        if ( inLijst[j] > count+1 ) {
                            inLijst += params[6];
                            goto NextGroup;
                        }
                    }
                    j = params[6];
                    NCOPY(Lijst,inLijst,j);
                    k++;
NextGroup:;
                }
                if ( k <= 1 ) goto NextFun;
                ngroup = k;
                inLijst = AT.WorkPointer;
                AT.WorkPointer = Lijst;
                Lijst = inLijst;
            }
            else if ( Nlist == 0 ) {
                for ( i = 0; i < count; i++ ) Lijst[i] = i;
                AT.WorkPointer += count;
                ngroup = count;
            }
            else {
                for ( i = 0; i < Nlist; i++ ) Lijst[i] = inLijst[i];
                AT.WorkPointer += Nlist;
            }
            j = Symmetrize(BHEAD t,Lijst,ngroup,params[6],params[2]);
            AT.WorkPointer = Lijst;
            if ( params[2] == 4 ) { /* antisymmetric */
                if ( ( j & 1 ) != 0 ) sign = -sign;
                if ( ( j & 2 ) != 0 ) return(0); /* equal arguments */
            }
            if ( ( j & 4 ) != 0 ) sumch++;
            t[2] &= ~DIRTYSYMFLAG;
        }
NextFun:
        t += t[1];
    }
    if ( sign < 0 ) {
        t = term;
        t += *t - 1;
        *t = -*t;
    }
    if ( sumch ) {
        if ( Normalize(BHEAD term) ) {
            MLOCK(ErrorMessageLock);
            MesCall("SymGen");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
```

```
        if ( !*term ) return(0);
        *AN.RepPoint = 1;
        AR.expchanged = 1;
        if ( AR.CurDum > AM.IndDum && AR.sLevel <= 0 ) ReNumber(BHEAD term);
    }
    return(Generator(BHEAD term,level));
}

/*
        #] SymGen :
        #[ SymFind :

        There is a certain amount of double work here, as this routine
        finds the function to be treated, while the SymGen routine has
        to find it again. Note however that this way things remain
        uniform and simple. Moreover this avoids problems with actions
        on more than one function simultaneously.
        Output in AT.TMout:
        Number,sym/anti,fun,lenpar,ngroups,gsize,fields

*/

WORD SymFind(PHEAD WORD *term, WORD *params)
{
    GETBIDENTITY
    WORD *t, *r, *m;
    WORD j, c1, c2, count;
    WORD *rstop;
    c1 = params[4];     /* function number */
    c2 = FUNCTION + WILDOFFSET;
    t = term;
    m = t + *t;
    m -= ABS(m[-1]);
    t++;
    while ( t < m ) {
        if ( *t == c1 || c1 > c2 ) {    /* Candidate function */
            if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
                >= TENSORFUNCTION ) { count = t[1] - FUNHEAD; }
            else {
                count = 0;
                r = t;
                rstop = t + t[1];
                r += FUNHEAD;
                while ( r < rstop ) { count++; NEXTARG(r) }
            }
            if ( ( j = params[5] ) > 0 && j != count ) goto NextFun;
            if ( j == 0 ) {
                r = params + 8;
                rstop = params + params[1];
                while ( r < rstop ) {
                    if ( *r > count + 1 ) goto NextFun;
                    r++;
                }
            }

            t = AT.TMout;
            r = params;
            j = r[1] - 1;
            *t++ = j;
            *t++ = SYMMETRIZE;
            r += 3;
            j--;
            NCOPY(t,r,j);
            return(1);
        }
NextFun:
        t += t[1];
    }
    return(0);
}

/*
        #] SymFind :
        #[ ChainIn :

        Equivalent to repeat id f(?a)*f(?b) = f(?a,?b);

        This one always takes less space.
*/

int ChainIn(PHEAD WORD *term, WORD funnum)
{
    GETBIDENTITY
    WORD *t, *tend, *m, *tt, *ts;
    int action;
    if ( funnum < 0 ) { /* Dollar to be expanded */
        funnum = DolToFunction(BHEAD -funnum);
```

```
        if ( AN.ErrorInDollar || funnum <= 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Dollar variable does not evaluate to function in ChainIn statement");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
    }
    do {
        action = 0;
        tend = term+*term;
        tend -= ABS(tend[-1]);
        t = term+1;
        while ( t < tend ) {
            if ( *t != funnum ) { t += t[1]; continue; }
            m = t;
            t += t[1];
            tt = t;
            if ( t >= tend || *t != funnum ) continue;
            action = 1;
            while ( t < tend && *t == funnum ) {
                ts = t + t[1];
                t += FUNHEAD;
                while ( t < ts ) *tt++ = *t++;
            }
            m[1] = tt - m;
            ts = term + *term;
            while ( t < ts ) *tt++ = *t++;
            *term = tt - term;
            break;
        }
    } while ( action );
    return(0);
}

/*
        #] ChainIn :
        #[ ChainOut :

        Equivalent to repeat id f(x1?,x2?,?a) = f(x1)*f(x2,?a);
*/

int ChainOut(PHEAD WORD *term, WORD funnum)
{
    GETBIDENTITY
    WORD *t, *tend, *tt, *ts, *w, *ws;
    int flag = 0, i;
    if ( funnum < 0 ) { /* Dollar to be expanded */
        funnum = DolToFunction(BHEAD -funnum);
        if ( AN.ErrorInDollar || funnum <= 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Dollar variable does not evaluate to function in ChainOut statement");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
    }
    tend = term+*term;
    if ( AT.WorkPointer < tend ) AT.WorkPointer = tend;
    tend -= ABS(tend[-1]);
    t = term+1; tt = term; w = AT.WorkPointer;
    while ( t < tend ) {
        if ( *t != funnum || t[1] == FUNHEAD ) { t += t[1]; continue; }
        flag = 1;
        while ( tt < t ) *w++ = *tt++;
        ts = t + t[1];
        t += FUNHEAD;
        while ( t < ts ) {
            ws = w;
            for ( i = 0; i < FUNHEAD; i++ ) *w++ = tt[i];
            if ( functions[*tt-FUNCTION].spec >= TENSORFUNCTION ) {
                *w++ = *t++;
            }
            else if ( *t < 0 ) {
                if ( *t <= -FUNCTION ) *w++ = *t++;
                else { *w++ = *t++; *w++ = *t++; }
            }
            else {
                i = *t; NCOPY(w,t,i);
            }
            ws[1] = w - ws;
        }
        tt = t;
    }
    if ( flag == 1 ) {
        ts = term + *term;
        while ( tt < ts ) *w++ = *tt++;
        *AT.WorkPointer = w - AT.WorkPointer;
        t = term; w = AT.WorkPointer; i = *w;
```

```
        NCOPY(t,w,i)
        AT.WorkPointer = term + *term;
        Normalize(BHEAD term);
    }
    return(0);
}

/*
        #] ChainOut :
    #] Utilities :
    #[ Patterns :
        #[ MatchFunction :          WORD MatchFunction(pattern,interm,wilds)

        The routine assumes that the function numbers are the same.
        The contents are compared and a possible wildcard assignment
        is made. Note that it may be necessary to use a wildcard
        assignment stack to do things right.
        The routine can become arbitrarily complicated as there is
        no end to the possible wildcarding.
        Examples:
        -   a:  No wildcarding -> straight match
        -   b:  Individual arguments (object -> object)
        -   c:  whole arguments (object to subexpression)
        -   d:  any argumentlist
            e:  part of an argument (object inside subexpression)

        The ones with a minus sign in front have been implemented.

        Note: the argument wilds allows backtracking when multiple
        ?a,?b give a match that later turns out to be useless.
*/

WORD MatchFunction(PHEAD WORD *pattern, WORD *interm, WORD *wilds)
{
    GETBIDENTITY
    WORD *m, *t, *r, i;
    WORD *mstop = 0, *tstop = 0;
    WORD *argmstop, *argtstop;
    WORD *mtrmstop, *ttrmstop;
    WORD *msubstop, *mnextsub;
    WORD msizcoef, mcount, tcount, newvalue, j;
    WORD *oldm, *oldt;
    WORD *OldWork, numofwildarg;
    WORD nwstore, tobeeaten, reservevalue = 0, resernum = 0, withwild;
    WORD *wildargtaken;
    CBUF *C = cbuf+AT.ebufnum;
    int ntwa = AN.NumTotWildArgs;
    LONG oldcpointer = C->Pointer - C->Buffer;
/*
    Test first for a straight match
*/
    AN.RepFunList[AN.RepFunNum+1] = 0;
    if ( *wilds == 0 ) {
        m = pattern; t = interm;

        if ( *m != *t ) {
            if ( *m < (FUNCTION + WILDOFFSET) ) return(0);
            if ( *t < FUNCTION ) return(0);
            if ( functions[*t-FUNCTION].spec !=
            functions[*m-FUNCTION-WILDOFFSET].spec ) return(0);
        }
        i = m[1];
        if ( *m >= (FUNCTION + WILDOFFSET) ) { i--; m++; t++; }
        do { if ( *m++ != *t++ ) break; } while ( --i > 0 );
        if ( i <= 0 ) {         /* Arguments match */
            if ( AN.SignCheck && AN.ExpectedSign ) return(0);
            i = *pattern - WILDOFFSET;
            if ( i >= FUNCTION ) {
                if ( *interm != GAMMA
                && !CheckWild(BHEAD i,FUNTOFUN,*interm,&newvalue) ) {
                    AddWild(BHEAD i,FUNTOFUN,newvalue);
                    return(1);
                }
                return(0);
            }
            else return(1);
        }
    }
/*
    Store the current Wildcard assignments
*/
    t = wildargtaken = OldWork = AT.WorkPointer;
    t += ntwa;
    m = AN.WildValue;
    nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
    if ( i > 0 ) {
        r = AT.WildMask;
```

```
        do {
            *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
        } while ( --i > 0 );
        *t++ = C->numrhs;
    }
    if ( t >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    AT.WorkPointer = t;

    if ( *wilds ) {
        if ( *wilds == 1 ) goto endoloop;
        else               goto enloop;         /* tensors = 2 */
    }
    m = pattern; t = interm;
/*
    Single out the specials
*/
    if ( *t == GAMMA ) {
/*
        #[ GAMMA :

        For the gamma's we need to do two things:
        a:  Find that there is a match
        b:  Find where the match occurs in the string
        This last thing cannot be stored in the current conventions,
        but once the wildcard assignments have been made it is much
        easier to find it back.
        Alternative: replace the function number in the term temporarily
        by the offset inside the string. This makes things maybe easier.
*/
        if ( *m != GAMMA ) goto NoCaseB;
        i = t[1] - m[1];
        if ( m[1] == FUNHEAD+1 ) {
            if ( i ) goto NoCaseB;
            if ( m[FUNHEAD] < (AM.OffsetIndex+WILDOFFSET) ||
            t[FUNHEAD] >= (AM.OffsetIndex+WILDOFFSET) ) goto NoCaseB;

            if ( CheckWild(BHEAD m[FUNHEAD]-WILDOFFSET,INDTOIND,t[FUNHEAD],&newvalue) ) goto NoCaseB;
            AddWild(BHEAD m[FUNHEAD]-WILDOFFSET,INDTOIND,newvalue);

            AT.WorkPointer = OldWork;
            if ( AN.SignCheck && AN.ExpectedSign ) return(0);
            return(1);      /* m was eaten. we have a match! */
        }
        if ( i < 0 ) goto NoCaseB;  /* Pattern longer than target */
        mstop = m + m[1];
        tstop = t + t[1];
        m += FUNHEAD; t += FUNHEAD;
        if ( *m >= (AM.OffsetIndex+WILDOFFSET) && *t < (AM.OffsetIndex+WILDOFFSET) ) {
            if ( CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*t,&newvalue) ) goto NoCaseB;
            reservevalue = newvalue;
            withwild = 1;
            resernum = *m-WILDOFFSET;
            AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newvalue);
        }
        else if ( *m != *t ) goto NoCaseB;
        else withwild = 0;
        m++; t++;
        oldm = m; argtstop = oldt = t;
        j = 0;                  /* No wildcard assignments yet */
        while ( i >= 0 ) {
            if ( *m == *t ) {
WithGamma:      m++; t++;
                if ( m >= mstop ) {
                    if ( t < tstop && mstop < AN.patstop ) {
                        WORD k;
                        mnextsub = pattern + pattern[1];
                        k = *mnextsub;
                        while ( k == GAMMA && mnextsub[FUNHEAD]
                        != pattern[FUNHEAD] ) {
                            mnextsub += mnextsub[1];
                            if ( mnextsub >= AN.patstop ) goto FullOK;
                            k = *mnextsub;
                        }
                        if ( k >= FUNCTION ) {
                            if ( k > (FUNCTION + WILDOFFSET) ) k -= WILDOFFSET;
                            if ( functions[k-FUNCTION].commute ) goto NoGamma;
                        }
                    }
FullOK:             if ( AN.SignCheck && AN.ExpectedSign ) goto NoGamma;
                    AN.RepFunList[AN.RepFunNum+1] = WORDDIF(oldt,argtstop);
                    return(1);
                }
```

```
                if ( t >= tstop ) goto NoCaseB;
            }
            else if ( *m >= (AM.OffsetIndex+WILDOFFSET)
            && *m < (AM.OffsetIndex + (WILDOFFSET«1)) && ( *t >= 0 ||
            *t < MINSPEC ) ) {          /* Wildcard index */
                if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*t,&newvalue) ) {
                    AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newvalue);
                    j = 1;
                    goto WithGamma;
                }
                else goto NoGamma;
            }
            else if ( *m < MINSPEC && *m >= (AM.OffsetVector+WILDOFFSET)
            && *t < MINSPEC ) {         /* Wildcard vecor */
                if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*t,&newvalue) ) {
                    AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newvalue);
                    j = 1;
                    goto WithGamma;
                }
                else goto NoGamma;
            }
            else {
NoGamma:
                if ( j ) {      /* Undo wildcards */
                    m = AN.WildValue;
                    t = OldWork + AN.NumTotWildArgs; r = AT.WildMask; j = nwstore;
                    if ( j > 0 ) {
                        do {
                            *m++ = *t++; *m++ = *t++;
                            *m++ = *t++; *m++ = *t++; *r++ = *t++;
                        } while ( --j > 0 );
                        C->numrhs = *t++;
                        C->Pointer = C->Buffer + oldcpointer;
                    }
                    j = 0;
                }
                m = oldm; t = ++oldt; i--;
                if ( withwild ) {
                    AddWild(BHEAD resernum,INDTOIND,reservevalue);
                }
            }
        }
        goto NoCaseB;
/*
        #] GAMMA :
        #[ Tensors :
*/
    }
    else if ( *t >= FUNCTION && functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
        mstop = m + m[1];
        tstop = t + t[1];
        mcount = 0;
        m += FUNHEAD;
        t += FUNHEAD;
        AN.WildArgs = 0;
        tcount = WORDDIF(tstop,t);
        while ( m < mstop ) {
            if ( *m == FUNNYWILD ) { m++; AN.WildArgs++; }
            m++; mcount++;
        }
        tobeeaten = tcount - mcount + AN.WildArgs;
        if ( tobeeaten ) {
            if ( tobeeaten < 0 || AN.WildArgs == 0 ) {
                AT.WorkPointer = OldWork;
                return(0);  /* Cannot match */
            }
        }
        AT.WildArgTaken[0] = AN.WildEat = tobeeaten;
        for ( i = 1; i < AN.WildArgs; i++ ) AT.WildArgTaken[i] = 0;
toploop:
        numofwildarg = 0;

        m = pattern; t = interm;
        mstop = m + m[1];
        if ( *m != *t ) {
            i = *m - WILDOFFSET;
            if ( CheckWild(BHEAD i,FUNTOFUN,*t,&newvalue) ) goto NoCaseB;
            AddWild(BHEAD i,FUNTOFUN,newvalue);
        }
        m += FUNHEAD;
        t += FUNHEAD;
        while ( m < mstop ) {
/*
            First test for an exact match
*/
            if ( *m == *t ) { m++; t++; continue; }
/*
```

```
            No exact match. Try ARGWILD
*/
            AN.argaddress = t;
            if ( *m == FUNNYWILD ) {
                tobeeaten = AT.WildArgTaken[numofwildarg++];
                i = tobeeaten | EATTENSOR;
                if ( CheckWild(BHEAD m[1],ARGTOARG,i,t) ) goto endloop;
                AddWild(BHEAD m[1],ARGTOARG,i);
                m += 2;
                t += tobeeaten;
                continue;
            }
/*
            Now the various cases:
*/
            i = *m;
            if ( i < MINSPEC ) {
                if ( *t != i ) {
                    if ( *t >= MINSPEC ) goto endloop;
                    i -= WILDOFFSET;
                    if ( i < AM.OffsetVector ) goto endloop;
                    if ( CheckWild(BHEAD i,VECTOVEC,*t,&newvalue) )
                        goto endloop;
                    AddWild(BHEAD i,VECTOVEC,newvalue);
                }
            }
            else if ( i >= AM.OffsetIndex ) {           /* Index */
                if ( i < ( AM.OffsetIndex + WILDOFFSET ) ) goto endloop;
                if ( i >= ( AM.OffsetIndex + (WILDOFFSET«1) ) ) {
                                                /* Summed over index */
                    goto endloop;               /* For the moment */
                }
                i -= WILDOFFSET;
                if ( CheckWild(BHEAD i,INDTOIND,*t,&newvalue) )
                    goto endloop;       /* Assignment not allowed */
                AddWild(BHEAD i,INDTOIND,newvalue);
            }
            else goto endloop;
            m++; t++;
        }
        if ( AN.SignCheck && AN.ExpectedSign ) goto endloop;
        AT.WorkPointer = OldWork;
        if ( AN.WildArgs > 1 ) *wilds = 2;
        return(1);      /* m was eaten. we have a match! */

endloop:;
/*
    restore the current Wildcard assignments
*/
        i = nwstore;
        if ( i > 0 ) {
            m = AN.WildValue;
            t = OldWork + ntwa; r = AT.WildMask;
            do {
                *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
            } while ( --i > 0 );
            C->numrhs = *t++;
            C->Pointer = C->Buffer + oldcpointer;
        }
enloop:;
        i = AN.WildArgs - 1;
        if ( i <= 0 ) {
            AT.WorkPointer = OldWork;
            return(0);
        }
        while ( --i >= 0 ) {
            if ( AT.WildArgTaken[i] == 0 ) {
                if ( i == 0 ) {
                    AT.WorkPointer = OldWork;
                    *wilds = 0;
                    return(0);
                }
            }
            else {
                (AT.WildArgTaken[i])--;
                numofwildarg = 0;
                for ( j = 0; j <= i; j++ ) {
                    numofwildarg += AT.WildArgTaken[j];
                }
                AT.WildArgTaken[j] = AN.WildEat-numofwildarg;
                for ( j++; j < AN.WildArgs; j++ ) AT.WildArgTaken[j] = 0;
                break;
            }
        }
        goto toploop;
/*
        #] Tensors :
```

```
*/
    }
/*
    Count the number of arguments. Either equal or an argument wildcard.
*/
    mstop = m + m[1];
    tstop = t + t[1];
    mcount = 0; tcount = 0;
    m += FUNHEAD; t += FUNHEAD;
    while ( t < tstop ) { tcount++; NEXTARG(t) }
    AN.WildArgs = 0;
    while ( m < mstop ) {
        mcount++;
        if ( *m == -ARGWILD ) AN.WildArgs++;
        NEXTARG(m)
    }
    tobeeaten = tcount - mcount + AN.WildArgs;
    if ( tobeeaten ) {
        if ( tobeeaten < 0 || AN.WildArgs == 0 ) {
            AT.WorkPointer = OldWork;
            return(0);  /* Cannot match */
        }
    }
/*
    Set up the array AT.WildArgTaken for the number of arguments that each
    wildarg eats.
*/
    AT.WildArgTaken[0] = AN.WildEat = tobeeaten;
    for ( i = 1; i < AN.WildArgs; i++ ) AT.WildArgTaken[i] = 0;
topofloop:
    numofwildarg = 0;
/*
    Test for single wildcard object/argument
*/
    m = pattern; t = interm;
    if ( *m != *t ) {
        i = *m - WILDOFFSET;
        if ( CheckWild(BHEAD i,FUNTOFUN,*t,&newvalue) ) goto NoCaseB;
        AddWild(BHEAD i,FUNTOFUN,newvalue);
    }
    mstop = m + m[1];
/*  tstop = t + t[1];  */
    m += FUNHEAD;
    t += FUNHEAD;
    while ( m < mstop ) {
        argmstop = oldm = m;
        argtstop = oldt = t;
        NEXTARG(argmstop)
        NEXTARG(argtstop)
        if ( t == tstop ) { /* This concerns a very rare bug */
            if ( *m == -ARGWILD ) goto ArgAll;
            goto endofloop;
        }
        if ( *m < 0 && *t < 0 ) {
            if ( *t <= -FUNCTION ) {
                if ( *t == *m ) {}
                else if ( *m <= -FUNCTION-WILDOFFSET
                && functions[-*t-FUNCTION].spec
                == functions[-*m-FUNCTION-WILDOFFSET].spec ) {
                    i = -*m - WILDOFFSET;
                    if ( CheckWild(BHEAD i,FUNTOFUN,-*t,&newvalue) ) goto endofloop;
                    AddWild(BHEAD i,FUNTOFUN,newvalue);
                }
                else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER ) {
                    i = m[1] - 2*MAXPOWER;
                    AN.argaddress = AT.FunArg;
                    AT.FunArg[ARGHEAD+1] = -*t;
                    if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) goto endofloop;
                    AddWild(BHEAD i,SYMTOSUB,0);
                }
                else if ( *m == -ARGWILD ) {
ArgAll:             i = AT.WildArgTaken[numofwildarg++];
                    AN.argaddress = t;
                    if ( CheckWild(BHEAD m[1],ARGTOARG,i,t) ) goto endofloop;
                    AddWild(BHEAD m[1],ARGTOARG,i);
/*                  m += 2; */
                    while ( --i >= 0 ) { NEXTARG(t) }
                    argtstop = t;
                }
                else goto endofloop;
            }
            else if ( *t == *m ) {
                if ( t[1] == m[1] ) {}
                else if ( *t == -SYMBOL ) {
                    j = SYMTOSYM;
SymAll:
                    if ( ( i = m[1] - 2*MAXPOWER ) < 0 ) goto endofloop;
```

```
                    if ( CheckWild(BHEAD i,j,t[1],&newvalue) ) goto endofloop;
                    AddWild(BHEAD i,j,newvalue);
                }
                else if ( *t == -INDEX ) {
IndAll:             i = m[1] - WILDOFFSET;
                    if ( i < AM.OffsetIndex || i >= WILDOFFSET+AM.OffsetIndex )
                                                    goto endofloop;
                                /* We kill the summed over indices here */
                    if ( CheckWild(BHEAD i,INDTOIND,t[1],&newvalue) ) goto endofloop;
                    AddWild(BHEAD i,INDTOIND,newvalue);
                }
                else if ( *t == -VECTOR || *t == -MINVECTOR ) {
                    i = m[1] - WILDOFFSET;
                    if ( i < AM.OffsetVector ) goto endofloop;
                    if ( CheckWild(BHEAD i,VECTOVEC,t[1],&newvalue) ) goto endofloop;
                    AddWild(BHEAD i,VECTOVEC,newvalue);
                }
                else goto endofloop;
            }
            else if ( *m == -ARGWILD ) goto ArgAll;
            else if ( *m == -INDEX && m[1] >= AM.OffsetIndex+WILDOFFSET
            && m[1] < AM.OffsetIndex+(WILDOFFSET«1) ) {
                if ( *t == -VECTOR ) goto IndAll;
                if ( *t == -SNUMBER && t[1] >= 0 && t[1] < AM.OffsetIndex ) goto IndAll;
                if ( *t == -MINVECTOR ) {
                    i = m[1] - WILDOFFSET;
                    AN.argaddress = AT.MinVecArg;
                    AT.MinVecArg[ARGHEAD+3] = t[1];
                    if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) goto endofloop;
                    AddWild(BHEAD i,INDTOSUB,(WORD)0);
                }
                else goto endofloop;
            }
            else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER && *t == -SNUMBER ) {
                j = SYMTONUM;
                goto SymAll;
            }
            else if ( *m == -VECTOR && *t == -MINVECTOR &&
            ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
/*
================================
                AN.argaddress = AT.MinVecArg;
                AT.MinVecArg[ARGHEAD+3] = t[1];
                if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) goto endofloop;
                AddWild(BHEAD i,VECTOSUB,(WORD)0);
================================
*/
                if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) goto endofloop;
                AddWild(BHEAD i,VECTOMIN,newvalue);

            }
            else if ( *m == -MINVECTOR && *t == -VECTOR &&
            ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
/*
================================
                AN.argaddress = AT.MinVecArg;
                AT.MinVecArg[ARGHEAD+3] = t[1];
                if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) goto endofloop;
                AddWild(BHEAD i,VECTOSUB,(WORD)0);
================================
*/
                if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) goto endofloop;
                AddWild(BHEAD i,VECTOMIN,newvalue);
            }
            else goto endofloop;
        }
        else if ( *t <= -FUNCTION && *m > 0 ) {
            if ( ( m[ARGHEAD]+ARGHEAD == *m ) && m[*m-1] == 3
            && m[*m-2] == 1 && m[*m-3] == 1 && m[ARGHEAD+1] >= FUNCTION
            && m[ARGHEAD+2] == *m-ARGHEAD-4 ) { /* Check for f(?a) etc */
                WORD *mmmst, *mmm;
                if ( m[ARGHEAD+1] >= FUNCTION+WILDOFFSET ) {
/*                  i = *m - WILDOFFSET; */
                    i = m[ARGHEAD+1] - WILDOFFSET;
                    if ( CheckWild(BHEAD i,FUNTOFUN,-*t,&newvalue) ) goto endofloop;
                    AddWild(BHEAD i,FUNTOFUN,newvalue);
                }
                else if ( m[ARGHEAD+1] != -*t ) goto endofloop;
/*
                    Only arguments allowed are ?a etc.
*/
                mmmst = m+*m-3;
                mmm = m + ARGHEAD + FUNHEAD + 1;
                while ( mmm < mmmst ) {
                    if ( *mmm != -ARGWILD ) goto endofloop;
                    i = 0;
                    AN.argaddress = t;
```

```
                    if ( CheckWild(BHEAD mmm[1],ARGTOARG,i,t) ) goto endofloop;
                    AddWild(BHEAD mmm[1],ARGTOARG,i);
                    mmm += 2;
                }
            }
            else goto endofloop;
        }
        else if ( *m < 0 && *t > 0 ) {
            if ( *m == -SYMBOL ) {          /* SYMTOSUB */
                if ( m[1] < 2*MAXPOWER ) goto endofloop;
                i = m[1] - 2*MAXPOWER;
                AN.argaddress = t;
                if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) goto endofloop;
                AddWild(BHEAD i,SYMTOSUB,0);
            }
            else if ( *m == -VECTOR ) {
                if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetVector )
                                                        goto endofloop;
                AN.argaddress = t;
                if ( CheckWild(BHEAD i,VECTOSUB,1,t) ) goto endofloop;
                AddWild(BHEAD i,VECTOSUB,(WORD)0);
            }
            else if ( *m == -INDEX ) {
                if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetIndex ) goto endofloop;
                if ( i >= AM.OffsetIndex + WILDOFFSET ) goto endofloop;
                AN.argaddress = t;
                if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) goto endofloop;
                AddWild(BHEAD i,INDTOSUB,(WORD)0);
            }
            else if ( *m == -ARGWILD ) goto ArgAll;
            else goto endofloop;
        }
        else if ( *m > 0 && *t > 0 ) {
            WORD ii = *t-*m;
            i = *m;
            do { if ( *m++ != *t++ ) break; } while ( --i > 0 );
            if ( i == 1 && ii == 0 ) {  /* sign difference */
                goto endofloop;
            }
            else if ( i > 0 ) {
                WORD *cto, *cfrom, *csav, ci;
                WORD oRepFunNum;
                WORD *oRepFunList;
                WORD *oterstart,*oterstop,*opatstop;
                WORD oExpectedSign;
                WORD wildargs, wildeat;
/*
                Not an exact match here.
                We have to hope that the pattern contains a composite wildcard.
*/
                m = oldm; t = oldt;
                m += ARGHEAD; t += ARGHEAD;         /* Point at (first?) term */
                mtrmstop = m + *m;
                ttrmstop = t + *t;
                if ( mtrmstop < argmstop ) goto endofloop;/* More than one term */
                msizcoef = mtrmstop[-1];
                if ( msizcoef < 0 ) msizcoef = -msizcoef;
                msubstop = mtrmstop - msizcoef;
                m++;
                if ( m >= msubstop ) goto endofloop;    /* Only coefficient */
/*
                Here we have a composite term. It can match provided it
                matches the entire argument. This argument must be a
                single term also and the coefficients should match
                (more or less).
                The matching takes:
                1:  Match the functions etc. Nothing can be left.
                2:  Match dotproducts and symbols. ONLY must match
                    and nothing may be left.
                For safety it is best to take the term out and put it
                in workspace.
*/

                if ( argtstop > ttrmstop ) goto endofloop;
                m--;
                oterstart = AN.terstart;
                oterstop = AN.terstop;
                opatstop = AN.patstop;
                oRepFunList = AN.RepFunList;
                oRepFunNum = AN.RepFunNum;
                AN.RepFunNum = 0;
                AN.RepFunList = AT.WorkPointer;
                AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
                if ( AT.WorkPointer+*t+5 > AT.WorkTop ) {
                    MLOCK(ErrorMessageLock);
                    MesWork();
                    MUNLOCK(ErrorMessageLock);
```

```
                    return(-1);
                }
                csav = cto = AT.WorkPointer;
                cfrom = t;
                ci = *t;
                while ( --ci >= 0 ) *cto++ = *cfrom++;
                AT.WorkPointer = cto;
                ci = msizcoef;
                cfrom = mtrmstop;
                --ci;
                if ( abs(*--cfrom) != abs(*--cto) ) {
                    AT.WorkPointer = csav;
                    AN.RepFunList = oRepFunList;
                    AN.RepFunNum = oRepFunNum;
                    AN.terstart = oterstart;
                    AN.terstop = oterstop;
                    AN.patstop = opatstop;
                    goto endofloop;
                }
                i = (*cfrom != *cto) ? 1 : 0; /* buffer AN.ExpectedSign until we are beyond the goto */
                while ( --ci >= 0 ) {
                    if ( *--cfrom != *--cto ) {
                        AT.WorkPointer = csav;
                        AN.RepFunList = oRepFunList;
                        AN.RepFunNum = oRepFunNum;
                        AN.terstart = oterstart;
                        AN.terstop = oterstop;
                        AN.patstop = opatstop;
                        goto endofloop;
                    }
                }
                oExpectedSign =  AN.ExpectedSign; /* buffer AN.ExpectedSign until we are beyond FindRest/FindOnly */
                AN.ExpectedSign = i;
                *m -= msizcoef;
                wildargs = AN.WildArgs;
                wildeat = AN.WildEat;
                for ( i = 0; i < wildargs; i++ ) wildargtaken[i] = AT.WildArgTaken[i];
                AN.ForFindOnly = 0; AN.UseFindOnly = 1;
                AN.nogroundlevel++;
                if ( FindRest(BHEAD csav,m) && ( AN.UsedOtherFind || FindOnly(BHEAD csav,m) ) ) {}
                else {
nomatch:
                    *m += msizcoef;
                    AT.WorkPointer = csav;
                    AN.RepFunList = oRepFunList;
                    AN.RepFunNum = oRepFunNum;
                    AN.terstart = oterstart;
                    AN.terstop = oterstop;
                    AN.patstop = opatstop;
                    AN.WildArgs = wildargs;
                    AN.WildEat = wildeat;
                    AN.ExpectedSign = oExpectedSign;
                    AN.nogroundlevel--;
                    for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
                    goto endofloop;
                }
/*              if ( *m == 1 || m[1] < FUNCTION || functions[m[1]-FUNCTION].spec >= TENSORFUNCTION ) { */
                if ( *m == 1 || m[1] < FUNCTION ) {
                    if ( AN.ExpectedSign ) goto nomatch;
                }
                else {
                    if ( m[1] > FUNCTION + WILDOFFSET ) {
                        if ( functions[m[1]-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION ) {
                            if ( AN.ExpectedSign != AN.RepFunList[AN.RepFunNum-1] ) goto nomatch;
                        }
                    }
                    else {
                        if ( AN.ExpectedSign != AN.RepFunList[AN.RepFunNum-1] ) goto nomatch;
/*
                        if ( functions[m[1]-FUNCTION].spec >= TENSORFUNCTION ) {
                            if ( AN.ExpectedSign != AN.RepFunList[AN.RepFunNum-1] ) goto nomatch;
                        }
*/
                    }
                }
                AN.nogroundlevel--;
                AN.ExpectedSign = oExpectedSign;
                AN.WildArgs = wildargs;
                AN.WildEat = wildeat;
                for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
                Substitute(BHEAD csav,m,1);
                cto = csav;
                cfrom = cto + *cto - msizcoef;
                cto++;
```

```
                *m += msizcoef;
                AT.WorkPointer = csav;
                AN.RepFunList = oRepFunList;
                AN.RepFunNum = oRepFunNum;
                AN.terstart = oterstart;
                AN.terstop = oterstop;
                AN.patstop = opatstop;
                if ( *cto != SUBEXPRESSION ) goto endofloop;
                cto += cto[1];
                if ( cto < cfrom ) goto endofloop;
            }
        }
        else goto endofloop;

        t = argtstop;                       /* Next argument */
        m = argmstop;
    }
    if ( AN.SignCheck && AN.ExpectedSign ) goto endofloop;
    AT.WorkPointer = OldWork;
    if ( AN.WildArgs > 1 ) *wilds = 1;
    if ( AN.SignCheck && AN.ExpectedSign ) return(0);
    return(1);      /* m was eaten. we have a match! */

endofloop:;
/*
    restore the current Wildcard assignments
*/
    i = nwstore;
    if ( i > 0 ) {
        m = AN.WildValue;
        t = OldWork + ntwa; r = AT.WildMask;
        do {
            *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
        } while ( --i > 0 );
        C->numrhs = *t++;
        C->Pointer = C->Buffer + oldcpointer;
    }

endoloop:;
    i = AN.WildArgs-1;
    if ( i <= 0 ) {
        AT.WorkPointer = OldWork;
        return(0);
    }
    while ( --i >= 0 ) {
        if ( AT.WildArgTaken[i] == 0 ) {
            if ( i == 0 ) {
                AT.WorkPointer = OldWork;
                return(0);
            }
        }
        else {
            (AT.WildArgTaken[i])--;
            numofwildarg = 0;
            for ( j = 0; j <= i; j++ ) {
                numofwildarg += AT.WildArgTaken[j];
            }
            AT.WildArgTaken[j] = AN.WildEat-numofwildarg;
/* ----> bug to be replaced in other source code */
            for ( j++; j < AN.WildArgs; j++ ) AT.WildArgTaken[j] = 0;
            break;
        }
    }
    goto topofloop;
NoCaseB:
/*
    Restore the old Wildcard assignments
*/
    i = nwstore;
    if ( i > 0 ) {
        m = AN.WildValue;
        t = OldWork + ntwa; r = AT.WildMask;
        do {
            *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
        } while ( --i > 0 );
        C->numrhs = *t++;
        C->Pointer = C->Buffer + oldcpointer;
    }
    AT.WorkPointer = OldWork;
    return(0);      /* no match */
}

/*
        #] MatchFunction :
        #[ ScanFunctions :          WORD ScanFunctions(inpat,inter,par)

        Finds in which functions to look for a match.
```

```
        inpat is the start of the pattern still to be matched.
        inter is the start of the term still to be matched.
        par gives information about commutativity.
            par = 0: nothing special
            par = 1: regular noncommuting function
            par = 2: GAMMA function

        AN.patstop: end of the functions field in the search pattern
        AN.terstop: end of the functions field in the target pattern
        AN.terstart: address of entire term;

        The actual matching of the functions and their arguments is done
        in a number of different routines. Mainly MatchFunction when there
        are no symmetry properties.
        Also: MatchE
              MatchCy
              FunMatchSy
              FunMatchCy

        The main problem here is backtracking, ie continuing with wildcard
        possibilities when a first assignment doesn't work.
        Important note: this was completely forgotten in the symmetric
        functions till 6-jan-2009. As of the moment this still has to
        be fixed.

        Functions inside functions can cause problems when antisymmetric
        functions are involved. The sign of the term may be at stake.
        At the lowest level this is no problem but in f(-fas(n2,n1)) this
        plays a role. Next is when we have a product of functions inside
        an argument. The strategy must be that we test the sign only at the
        last function. Hence, when inpat+inpat[1] >= AN.patstop.
        We might relax that to the last antisymmetric function at a later stage.

    New scheme to be implemented for non-commuting objects:
    When we are matching a second (or higher) function, any match can only
    be directly after the last matched non-commuting function or a commuting
    function. This will take care of whatever happens in MatchE etc.
*/

WORD ScanFunctions(PHEAD WORD *inpat, WORD *inter, WORD par)
{
    GETBIDENTITY
    WORD i, *m, *t, *r, sym, psym;
    WORD *newpat, *newter, *instart, *oinpat = 0, *ointer = 0;
    WORD nwstore, offset, *OldWork, SetStop = 0, oRepFunNum = AN.RepFunNum;
    WORD wilds, wildargs = 0, wildeat = 0, *wildargtaken;
    WORD *Oterfirstcomm = AN.terfirstcomm;
    CBUF *C = cbuf+AT.ebufnum;
    int ntwa = AN.NumTotWildArgs;
    LONG oldcpointer = C->Pointer - C->Buffer;
    WORD oldSignCheck = AN.SignCheck;
    instart = inter;
/*
    Only active for the last function in the pattern.
    The actual test on the sign is in MatchFunction or the symmetric functions
*/
    if ( AN.nogroundlevel ) {
        AN.SignCheck = ( inpat + inpat[1] >= AN.patstop ) ? 1 : 0;
    }
    else {
        AN.SignCheck = 0;
    }
/*
            Store the current Wildcard assignments
*/
    t = wildargtaken = OldWork = AT.WorkPointer;
    t += ntwa;
    m = AN.WildValue;
    nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
    if ( i > 0 ) {
        r = AT.WildMask;
        do {
            *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
        } while ( --i > 0 );
        *t++ = C->numrhs;
    }
    if ( t >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    AT.WorkPointer = t;
    do {
#ifndef NEWCOMMUTE
/*
        Find an eligible unsubstituted function
```

```
*/
        if ( AN.RepFunNum > 0 ) {
/*
            First try a non-commuting function, just after the last
            substituted non-commuting function.
*/
            if ( *inter >= FUNCTION && functions[*inter-FUNCTION].commute ) {
                do {
                    offset = WORDDIF(inter,AN.terstart);
                    for ( i = 0; i < AN.RepFunNum; i += 2 ) {
                        if ( AN.RepFunList[i] >= offset ) break;
                    }
                    if ( i >= AN.RepFunNum ) break;
                    inter += inter[1];
                } while ( inter < AN.terfirstcomm );
                if ( inter < AN.terfirstcomm ) { /* Check that it is directly after */
                    for ( i = 0; i < AN.RepFunNum; i += 2 ) {
                        if ( functions[AN.terstart[AN.RepFunList[i]]-FUNCTION].commute
                        && AN.RepFunList[i]+AN.terstart[AN.RepFunList[i]+1] == offset ) break;
                    }
                    if ( i < AN.RepFunNum ) goto trythis;
                }
                inter = AN.terfirstcomm;
            }
/*
            Now try one of the commuting functions
*/
            while ( inter < AN.terstop ) {
                offset = WORDDIF(inter,AN.terstart);
                for ( i = 0; i < AN.RepFunNum; i += 2 ) {
                    if ( AN.RepFunList[i] == offset ) break;
                }
                if ( i >= AN.RepFunNum ) break;
                inter += inter[1];
            }
            if ( inter >= AN.terstop ) goto Failure;
trythis:;
        }
        else {
/*
            The first function can be anywhere. We have no problems.
*/
            offset = WORDDIF(inter,AN.terstart);
        }
#else
        /* first find an unsubstituted function */
        do {
            offset = WORDDIF(inter,AN.terstart);
            for ( i = 0; i < AN.RepFunNum; i += 2 ) {
                if ( AN.RepFunList[i] == offset ) break;
            }
            if ( i >= AN.RepFunNum ) break;
            inter += inter[1];
        } while ( inter < AN.terstop );
        if ( inter >= AN.terstop ) goto Failure;
#endif
        wilds = 0;
        /* We found one */
        if ( *inter >= FUNCTION && *inpat >= FUNCTION ) {
            if ( *inpat == *inter || *inpat >= FUNCTION + WILDOFFSET ) {
/*
                if ( inter[1] == FUNHEAD ) goto rewild;
*/
                if ( functions[*inter-FUNCTION].spec >= TENSORFUNCTION
                && ( *inter == *inpat ||
                functions[*inpat-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION ) ) {
                    sym = functions[*inter-FUNCTION].symmetric & ~REVERSEORDER;
                    if ( *inpat == *inter ) psym = sym;
                    else psym = functions[*inpat-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER;
                    if ( sym == ANTISYMMETRIC || sym == SYMMETRIC
                    || psym == SYMMETRIC || psym == ANTISYMMETRIC ) {
                        if ( sym == ANTISYMMETRIC && psym == SYMMETRIC ) goto rewild;
                        if ( sym == SYMMETRIC && psym == ANTISYMMETRIC ) goto rewild;
/*
                        Special function call for (anti)symmetric tensors
*/
                        if ( MatchE(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
                    }
                    else if ( sym == CYCLESYMMETRIC || sym == RCYCLESYMMETRIC
                    || psym == CYCLESYMMETRIC || psym == RCYCLESYMMETRIC ) {
/*
                        Special function call for (r)cyclic tensors
*/
                        if ( MatchCy(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
                    }
                    else goto rewild;
                }
```

```
                else if ( functions[*inter-FUNCTION].spec == 0
                && ( *inter == *inpat ||
                functions[*inpat-FUNCTION-WILDOFFSET].spec == 0 ) ) {
                    sym = functions[*inter-FUNCTION].symmetric & ~REVERSEORDER;
                    if ( *inpat == *inter ) psym = sym;
                    else psym = functions[*inpat-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER;
                    if ( psym == SYMMETRIC || sym == SYMMETRIC
/*
                    The next statement was commented out. Why????
                    Werkt nog niet. Teken wordt nog niet bijgehouden.
                    5-nov-2001
*/
                    || psym == ANTISYMMETRIC || sym == ANTISYMMETRIC
                    ) {
                        if ( sym == ANTISYMMETRIC && psym == SYMMETRIC ) goto rewild;
                        if ( sym == SYMMETRIC && psym == ANTISYMMETRIC ) goto rewild;
                        if ( FunMatchSy(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
                    }
                    else
                        if ( sym == CYCLESYMMETRIC || sym == RCYCLESYMMETRIC
                    || psym == CYCLESYMMETRIC || psym == RCYCLESYMMETRIC ) {
                        if ( FunMatchCy(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
                    }
                    else goto rewild;
                }
                else goto rewild;
                AN.terfirstcomm = Oterfirstcomm;
            }
            else if ( par > 0 ) { SetStop = 1; goto maybenext; }
        }
        else {
rewild:
        AN.terfirstcomm = Oterfirstcomm;
        if ( *inter != SUBEXPRESSION && MatchFunction(BHEAD inpat,inter,&wilds) ) {
            AN.terfirstcomm = Oterfirstcomm;
            if ( wilds ) {
/*
                Store wildcards to continue in MatchFunction if the current
                wildcards do not work out.
*/
                wildargs = AN.WildArgs;
                wildeat = AN.WildEat;
                for ( i = 0; i < wildargs; i++ ) wildargtaken[i] = AT.WildArgTaken[i];
                oinpat = inpat; ointer = inter;
            }
            if ( par && *inter == GAMMA && AN.RepFunList[AN.RepFunNum+1] ) {
                SetStop = 1; goto NoMat;
            }
            if ( par == 2 ) {
                if ( *inter < FUNCTION || functions[*inter-FUNCTION].commute ) {
                    goto NoMat;
                }
                par = 1;
            }
            AN.RepFunList[AN.RepFunNum] = offset;
            AN.RepFunNum += 2;
            newpat = inpat + inpat[1];
            if ( newpat >= AN.patstop ) {
                if ( AN.UseFindOnly == 0 ) {
                    if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                        AN.UsedOtherFind = 1;
                        goto OnSuccess;
                    }
                    AN.RepFunNum -= 2;
                    goto NoMat;
                }
                goto OnSuccess;
            }
            if ( *inter < FUNCTION || functions[*inter-FUNCTION].commute ) {
                newter = inter + inter[1];
                if ( newter >= AN.terstop ) goto Failure;
                if ( *inter == GAMMA && inpat[1] <
                inter[1] - AN.RepFunList[AN.RepFunNum-1] ) {
                    if ( ScanFunctions(BHEAD newpat,newter,2) ) goto OnSuccess;
                    AN.terfirstcomm = Oterfirstcomm;
                }
                else if ( *newter ==  SUBEXPRESSION ) {}
                else if ( functions[*inter-FUNCTION].commute ) {
                    if ( ScanFunctions(BHEAD newpat,newter,1) ) goto OnSuccess;
                    AN.terfirstcomm = Oterfirstcomm;
                    if ( ( *newpat < (FUNCTION+WILDOFFSET)
                        && ( functions[*newpat-FUNCTION].commute == 0 ) ) ||
                        ( *newpat >= (FUNCTION+WILDOFFSET)
                        && ( functions[*newpat-FUNCTION-WILDOFFSET].commute == 0 ) ) ) {
                        newter = AN.terfirstcomm;
                        if ( newter < AN.terstop && ScanFunctions(BHEAD newpat,newter,1) ) goto
    OnSuccess;
```

```
                    }
                }
                else {
                    if ( ScanFunctions(BHEAD newpat,instart,1) ) goto OnSuccess;
                    AN.terfirstcomm = Oterfirstcomm;
                }
                SetStop = par;
            }
            else {
/*
                Shouldn't this be newpat instead of inpat?????
*/
                if ( par && inter > instart && ( ( *newpat < (FUNCTION+WILDOFFSET)
                && functions[*newpat-FUNCTION].commute ) ||
                ( *newpat >= (FUNCTION+WILDOFFSET)
                && functions[*newpat-FUNCTION-WILDOFFSET].commute ) ) ) {
                    SetStop = 1;
                }
                else {
                    newter = instart;
                    if ( ScanFunctions(BHEAD newpat,newter,par) ) goto OnSuccess;
                    AN.terfirstcomm = Oterfirstcomm;
                }
            }
/*
            Restore the old Wildcard assignments
*/
NoMat:
            i = nwstore;
            if ( i > 0 ) {
                m = AN.WildValue;
                t = OldWork + ntwa; r = AT.WildMask;
                do {
                    *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                } while ( --i > 0 );
                C->numrhs = *t++;
                C->Pointer = C->Buffer + oldcpointer;
            }
/*          AN.RepFunNum -= 2; */
            AN.RepFunNum = oRepFunNum;
            if ( wilds ) {
                inter = ointer; inpat = oinpat;
                AN.WildArgs = wildargs;
                AN.WildEat = wildeat;
                for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
                goto rewild;
            }
            if ( SetStop ) break;
        }
        else if ( par ) {
maybenext:
            if ( *inpat < (FUNCTION+WILDOFFSET) ) {
                if ( *inpat < FUNCTION ||
                functions[*inpat-FUNCTION].commute ) break;
            }
            else {
                if ( functions[*inpat-FUNCTION-WILDOFFSET].commute ) break;
            }
        }}
        inter += inter[1];
    } while ( inter < AN.terstop );
Failure:
    AN.SignCheck = oldSignCheck;
    AT.WorkPointer = OldWork;
    return(0);
OnSuccess:
    if ( AT.idallflag && AN.nogroundlevel <= 0 ) {
        if ( AT.idallmaxnum > 0 && AT.idallnum >= AT.idallmaxnum ) {
            AN.terfirstcomm = Oterfirstcomm;
            AN.SignCheck = oldSignCheck;
            AT.WorkPointer = OldWork;
            return(0);
        }
        SubsInAll(BHEAD0);
        AT.idallnum++;
        if ( AT.idallmaxnum == 0 || AT.idallnum < AT.idallmaxnum ) goto NoMat;
    }
    AN.terfirstcomm = Oterfirstcomm;
    AN.SignCheck = oldSignCheck;
/*
    Now the disorder test
*/
    if ( AN.DisOrderFlag && AN.RepFunNum >= 4 ) {
        WORD k, kk;
        for ( i = 2; i < AN.RepFunNum; i += 2 ) {
/*
------------> We still have to copy the code from Normalize wrt properorderflag
```

```
*/
            m = AN.terstart + AN.RepFunList[i-2];
            t = AN.terstart + AN.RepFunList[i];
            if ( *m != *t ) {
                if ( *m > *t ) continue;
                goto doesmatch;
            }
            if ( *m >= FUNCTION && functions[*m-FUNCTION].spec >=
                TENSORFUNCTION ) {
                k = m[1] - FUNHEAD;
                kk = t[1] - FUNHEAD;
                m += FUNHEAD;
                t += FUNHEAD;
            }
            else {
                k = m[1] - FUNHEAD;
                kk = t[1] - FUNHEAD;
                m += FUNHEAD;
                t += FUNHEAD;
            }
            while ( k > 0 && kk > 0 ) {
                if ( *m < *t ) goto NextFor;
                else if ( *m++ > *t++ ) goto doesmatch;
                k--; kk--;
            }
            if ( k > 0 ) goto doesmatch;
NextFor:;
        }
        SetStop = 1;
        goto NoMat;
    }
doesmatch:
    AT.WorkPointer = OldWork;
    return(1);
}

/*
        #] ScanFunctions :
    #] Patterns :
*/
```

4.41 fwin.h File Reference

Macros

- #define LINEFEED '\n'
- #define CARRIAGERETURN 0x0D
- #define WITHRETURN
- #define WITHSYSTEM
- #define P_term(code) exit((int)(code<0?-code:code))
- #define SEPARATOR '\\'
- #define ALTSEPARATOR '/'
- #define PATHSEPARATOR ';'
- #define WITH_ENV

4.41.1 Detailed Description

Settings for Windows computers.

Definition in file fwin.h.

4.41.2 Macro Definition Documentation

4.41.2.1 LINEFEED

```
#define LINEFEED '\n'
```

Definition at line 33 of file fwin.h.

4.41.2.2 CARRIAGERETURN

```
#define CARRIAGERETURN 0x0D
```

Definition at line 34 of file fwin.h.

4.41.2.3 WITHRETURN

```
#define WITHRETURN
```

Definition at line 35 of file fwin.h.

4.41.2.4 WITHSYSTEM

```
#define WITHSYSTEM
```

Definition at line 37 of file fwin.h.

4.41.2.5 P_term

```
#define P_term(
              code ) exit((int)(code<0?-code:code))
```

Definition at line 39 of file fwin.h.

4.41.2.6 SEPARATOR

```
#define SEPARATOR '\\'
```

Definition at line 41 of file fwin.h.

4.41.2.7 ALTSEPARATOR

```
#define ALTSEPARATOR '/'
```

Definition at line 42 of file fwin.h.

4.41.2.8 PATHSEPARATOR

```
#define PATHSEPARATOR ';'
```

Definition at line 43 of file fwin.h.

4.41.2.9 WITH_ENV

```
#define WITH_ENV
```

Definition at line 44 of file fwin.h.

4.42 fwin.h

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032
00033 #define LINEFEED '\n'
00034 #define CARRIAGERETURN 0x0D
00035 #define WITHRETURN
00036
00037 #define WITHSYSTEM
00038
00039 #define P_term(code)    exit((int)(code<0?-code:code))
00040
00041 #define SEPARATOR '\\'
00042 #define ALTSEPARATOR '/'
00043 #define PATHSEPARATOR ';'
00044 #define WITH_ENV
```

4.43 if.c File Reference

`#include "form3.h"`

Include dependency graph for if.c:

Functions

- WORD GetIfDollarNum (WORD ∗ifp, WORD ∗ifstop)
- int FindVar (WORD ∗v, WORD ∗term)
- WORD DoIfStatement (PHEAD WORD ∗ifcode, WORD ∗term)
- WORD HowMany (PHEAD WORD ∗ifcode, WORD ∗term)
- VOID DoubleIfBuffers ()

4.43.1 Detailed Description

Routines for the dealing with if statements.

Definition in file if.c.

4.43.2 Function Documentation

4.43.2.1 GetIfDollarNum()

```
WORD GetIfDollarNum (
            WORD * ifp,
            WORD * ifstop )
```

Definition at line 106 of file if.c.

4.43.2.2 FindVar()

```
int FindVar (
            WORD * v,
            WORD * term )
```

Definition at line 173 of file if.c.

4.43.2.3 DoIfStatement()

```
WORD DoIfStatement (
            PHEAD WORD * ifcode,
            WORD * term )
```

Definition at line 280 of file if.c.

4.43.2.4 HowMany()

```
WORD HowMany (
            PHEAD WORD * ifcode,
            WORD * term )
```

Definition at line 816 of file if.c.

4.43.2.5 DoubleIfBuffers()

```
VOID DoubleIfBuffers ( )
```

Definition at line 1009 of file if.c.

4.44 if.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes : if.c
00033 */
00034
00035 #include "form3.h"
00036
00037 /*
00038     #] Includes :
00039     #[ If statement :
00040         #[ Syntax :
00041
00042         The `if' is a conglomerate of statements: if,else,endif
00043
00044         The if consists in principle of:
00045
00046         if ( number );
00047             statements
00048         else;
00049             statements
00050         endif;
00051
00052         The first set is taken when number != 0.
00053         The else is not mandatory.
00054         TRUE = 1 and FALSE = 0
00055
00056         The number can be built up via a logical expression:
00057
00058                     expr1 condition expr2
00059
00060         each expression can be a subexpression again. It has to be
00061         enclosed in parentheses in that case.
00062         Conditions are:
00063             >, >=, <, <=, ==, !=, ||, &&
00064
00065         When Expressions are chained evaluation is from left to right,
00066         independent of whether this indicates nonsense.
00067         if ( a || b || c || d ); is a perfectly normal statement.
00068         if ( a >= b || c == d ); would be messed up. This should be:
00069         if ( ( a >= b ) || ( c == d ) );
00070
00071         The building blocks of the Expressions are:
00072
```

```
            Match(option,pattern)   The number of times pattern fits in term_
            Count(....)             The count value of term_
            Coeff[icient]           The coefficient of term_
            FindLoop(options)       Are there loops (as in ReplaceLoop).

        Implementation for internal notation:

        TYPEIF,length,gotolevel(if fail),EXPRTYPE,length,......

        EXPRTYPE can be:
            SHORTNUMBER         ->,4,sign,size
            LONGNUMBER          ->,|ncoef+2|,ncoef,numer,denom
            MATCH               ->,patternsiz+3,keyword,pattern
            MULTIPLEOF          ->,3,thenumber
            COUNT               ->,countsiz+2,countinfo
            TYPEFINDLOOP        ->,7 (findloop info)
            COEFFICIENT         ->,2
            IFDOLLAR            ->,3,dollarnumber
            SUBEXPR             ->,size,dummy,size1,EXPRTYPE,length,...
                                  ,2,condition1,size2,...
                                This is like functions.

        Note that there must be a restriction to the number of nestings
        of parentheses in an if statement. It has been set to 10.

        The syntax of match corresponds to the syntax of the left side
        of an id statement. The only difference is the keyword
        MATCH vs TYPEIDNEW.

        #] Syntax :
        #[ GetIfDollarNum :
*/

WORD GetIfDollarNum(WORD *ifp, WORD *ifstop)
{
    DOLLARS d;
    WORD num, *w;
    if ( ifp[2] < 0 ) { return(-ifp[2]-1); }
    d = Dollars+ifp[2];
    if ( ifp+3 < ifstop && ifp[3] == IFDOLLAREXTRA ) {
        if ( d->nfactors == 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Attempt to use a factor of an unfactored $-variable");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        num = GetIfDollarNum(ifp+3,ifstop);
        if ( num > d->nfactors ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Dollar factor number %s out of range",num);
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        if ( num == 0 ) {
            return(d->nfactors);
        }
        w = d->factors[num-1].where;
        if ( w == 0 ) return(d->factors[num].value);
getnumber:;
        if ( *w == 0 ) return(0);
        if ( *w == 4 && w[3] == 3 && w[2] == 1 && w[1] < MAXPOSITIVE && w[4] == 0 ) {
            return(w[1]);
        }
        if ( ( w[w[0]] != 0 ) || ( ABS(w[w[0]-1]) != w[0]-1 ) ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Dollar factor number expected but found expression");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("Dollar factor number out of range");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        return(0);
    }
/*
    Now we have just a dollar and should evaluate that into a short number
*/
    if ( d->type == DOLZERO ) {
        return(0);
    }
    else if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
        w = d->where; goto getnumber;
    }
    else {
```

```
        MLOCK(ErrorMessageLock);
        MesPrint("Dollar factor number is wrong type");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
        return(0);
    }
}

/*
        #] GetIfDollarNum :
        #[ FindVar :
*/

int FindVar(WORD *v, WORD *term)
{
    WORD *t, *tstop, *m, *mstop, *f, *fstop, *a, *astop;
    GETSTOP(term,tstop);
    t = term+1;
    while ( t < tstop ) {
        if ( *v == *t && *v < FUNCTION ) {  /* VECTOR, INDEX, SYMBOL, DOTPRODUCT */
            switch ( *v ) {
                case SYMBOL:
                    m = t+2; mstop = t+t[1];
                    while ( m < mstop ) {
                        if ( *m == v[1] ) return(1);
                        m += 2;
                    }
                    break;
                case INDEX:
                case VECTOR:
InVe:
                    m = t+2; mstop = t+t[1];
                    while ( m < mstop ) {
                        if ( *m == v[1] ) return(1);
                        m++;
                    }
                    break;
                case DOTPRODUCT:
                    m = t+2; mstop = t+t[1];
                    while ( m < mstop ) {
                        if ( *m == v[1] && m[1] == v[2] ) return(1);
                        if ( *m == v[2] && m[1] == v[1] ) return(1);
                        m += 3;
                    }
                    break;
            }
        }
        else if ( *v == VECTOR && *t == INDEX ) goto InVe;
        else if ( *v == INDEX && *t == VECTOR ) goto InVe;
        else if ( ( *v == VECTOR || *v == INDEX ) && *t == DOTPRODUCT ) {
            m = t+2; mstop = t+t[1];
            while ( m < mstop ) {
                if ( v[1] == m[0] || v[1] == m[1] ) return(1);
                m += 3;
            }
        }
        else if ( *t >= FUNCTION ) {
            if ( *v == FUNCTION && v[1] == *t ) return(1);
            if ( functions[*t-FUNCTION].spec > 0 ) {
              if ( *v == VECTOR || *v == INDEX ) { /* we need to check arguments */
                int i;
                for ( i = FUNHEAD; i < t[1]; i++ ) {
                    if ( v[1] == t[i] ) return(1);
                }
              }
            }
            else {
              fstop = t + t[1]; f = t + FUNHEAD;
              while ( f < fstop ) { /* Do the arguments one by one */
                if ( *f <= 0 ) {
                    switch ( *f ) {
                        case -SYMBOL:
                            if ( *v == SYMBOL && v[1] == f[1] ) return(1);
                            f += 2;
                            break;
                        case -VECTOR:
                        case -MINVECTOR:
                        case -INDEX:
                            if ( ( *v == VECTOR || *v == INDEX )
                             && ( v[1] == f[1] ) ) return(1);
                            f += 2;
                            break;
                        case -SNUMBER:
                            f += 2;
                            break;
                        default:
                            if ( *v == FUNCTION && v[1] == -*f && *f <= -FUNCTION ) return(1);
```

```
                            if ( *f <= -FUNCTION ) f++;
                            else f += 2;
                            break;
                        }
                    }
                    else {
                        a = f + ARGHEAD; astop = f + *f;
                        while ( a < astop ) {
                            if ( FindVar(v,a) == 1 ) return(1);
                            a += *a;
                        }
                        f = astop;
                    }
                }
            }
        }
        t += t[1];
    }
    return(0);
}

/*
        #] FindVar :
        #[ DoIfStatement :              WORD DoIfStatement(PHEAD ifcode,term)

        The execution time part of the if-statement.
        The arguments are a pointer to the TYPEIF and a pointer to the term.
        The answer is either 1 (success) or 0 (fail).
        The calling routine can figure out where to go in case of failure
        by picking up gotolevel.
        Note that the whole setup asks for recursions.
*/

WORD DoIfStatement(PHEAD WORD *ifcode, WORD *term)
{
    GETBIDENTITY
    WORD *ifstop, *ifp;
    UWORD *coef1 = 0, *coef2, *coef3, *cc;
    WORD ncoef1, ncoef2, ncoef3, i = 0, first, *r, acoef, ismul1, ismul2, j;
    UWORD *Spac1, *Spac2;
    ifstop = ifcode + ifcode[1];
    ifp = ifcode + 3;
    if ( ifp >= ifstop ) return(1);
    if ( ( ifp + ifp[1] ) >= ifstop ) {
        switch ( *ifp ) {
            case LONGNUMBER:
                if ( ifp[2] ) return(1);
                else return(0);
            case MATCH:
            case TYPEIF:
                if ( HowMany(BHEAD ifp,term) ) return(1);
                else return(0);
            case TYPEFINDLOOP:
                if ( Lus(term,ifp[3],ifp[4],ifp[5],ifp[6],ifp[2]) ) return(1);
                else return(0);
            case TYPECOUNT:
                if ( CountDo(term,ifp) ) return(1);
                else return(0);
            case COEFFI:
            case MULTIPLEOF:
                return(1);
            case IFDOLLAR:
                {
                    DOLLARS d = Dollars + ifp[2];
#ifdef WITHPTHREADS
                    int nummodopt, dtype = -1;
                    if ( AS.MultiThreaded ) {
                        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                            if ( ifp[2] == ModOptdollars[nummodopt].number ) break;
                        }
                        if ( nummodopt < NumModOptdollars ) {
                            dtype = ModOptdollars[nummodopt].type;
                            if ( dtype == MODLOCAL ) {
                                d = ModOptdollars[nummodopt].dstruct+AT.identity;
                            }
                        }
                    }
                    dtype = d->type;
#else
                    int dtype = d->type;  /* We use dtype to make the operation atomic */
#endif
                    if ( dtype == DOLZERO ) return(0);
                    if ( dtype == DOLUNDEFINED ) {
                        if ( AC.UnsureDollarMode == 0 ) {
                            MesPrint("$%s is undefined",AC.dollarnames->namebuffer+d->name);
                            Terminate(-1);
                        }
```

```
                    }
                }
                return(1);
            case IFEXPRESSION:
                r = ifp+2; j = ifp[1] - 2;
                while ( --j >= 0 ) {
                    if ( *r == AR.CurExpr ) return(1);
                    r++;
                }
                return(0);
            case IFISFACTORIZED:
                r = ifp+2; j = ifp[1] - 2;
                if ( j == 0 ) {
                    if ( ( Expressions[AR.CurExpr].vflags & ISFACTORIZED ) != 0 )
                        return(1);
                    else
                        return(0);
                }
                while ( --j >= 0 ) {
                    if ( ( Expressions[*r].vflags & ISFACTORIZED ) == 0 ) return(0);
                    r++;
                }
                return(1);
            case IFOCCURS:
                {
                    WORD *OccStop = ifp + ifp[1];
                    ifp += 2;
                    while ( ifp < OccStop ) {
                        if ( FindVar(ifp,term) == 1 ) return(1);
                        if ( *ifp == DOTPRODUCT ) ifp += 3;
                        else ifp += 2;
                    }
                }
                return(0);
            default:
/*
                Now we have a subexpression. Test first for one with a single item.
*/
                if ( ifp[3] == ( ifp[1] + 3 ) ) return(DoIfStatement(BHEAD ifp,term));
                ifstop = ifp + ifp[1];
                ifp += 3;
                break;
        }
    }
/*
    Here is the composite condition.
*/
    coef3 = NumberMalloc("DoIfStatement");
    Spac1 = NumberMalloc("DoIfStatement");
    Spac2 = (UWORD *)(TermMalloc("DoIfStatement"));
    ncoef1 = 0; first = 1; ismul1 = 0;
    do {
        if ( !first ) {
            ifp += 2;
            if ( ifp[-2] == ORCOND && ncoef1 ) {
                coef1 = Spac1;
                ncoef1 = 1; coef1[0] = coef1[1] = 1;
                goto SkipCond;
            }
            if ( ifp[-2] == ANDCOND && !ncoef1 ) goto SkipCond;
        }
        coef2 = Spac2;
        ncoef2 = 1;
        ismul2 = 0;
        switch ( *ifp ) {
            case LONGNUMBER:
                ncoef2 = ifp[2];
                j = 2*(ABS(ncoef2));
                cc = (UWORD *)(ifp + 3);
                for ( i = 0; i < j; i++ ) coef2[i] = cc[i];
                break;
            case MATCH:
            case TYPEIF:
                coef2[0] = HowMany(BHEAD ifp,term);
                coef2[1] = 1;
                if ( coef2[0] == 0 ) ncoef2 = 0;
                break;
            case TYPECOUNT:
                acoef = CountDo(term,ifp);
                coef2[0] = ABS(acoef);
                coef2[1] = 1;
                if ( acoef == 0 ) ncoef2 = 0;
                else if ( acoef < 0 ) ncoef2 = -1;
                break;
            case TYPEFINDLOOP:
                acoef = Lus(term,ifp[3],ifp[4],ifp[5],ifp[6],ifp[2]);
                coef2[0] = ABS(acoef);
```

```
                coef2[1] = 1;
                if ( acoef == 0 ) ncoef2 = 0;
                else if ( acoef < 0 ) ncoef2 = -1;
                break;
            case COEFFI:
                r = term + *term;
                ncoef2 = r[-1];
                i = ABS(ncoef2);
                cc = (UWORD *)(r - i);
                if ( ncoef2 < 0 ) ncoef2 = (ncoef2+1)>>1;
                else ncoef2 = (ncoef2-1)>>1;
                i--; for ( j = 0; j < i; j++ ) coef2[j] = cc[j];
                break;
            case SUBEXPR:
                ncoef2 = coef2[0] = DoIfStatement(BHEAD ifp,term);
                coef2[1] = 1;
                break;
            case MULTIPLEOF:
                ncoef2 = 1;
                coef2[0] = ifp[2];
                coef2[1] = 1;
                ismul2 = 1;
                break;
            case IFDOLLAREXTRA:
                break;
            case IFDOLLAR:
                {
/*
                    We need to abstract a long rational in coef2
                    with length ncoef2. What if that cannot be done?
*/
                    DOLLARS d = Dollars + ifp[2];
#ifdef WITHPTHREADS
                    int nummodopt, dtype = -1;
                    if ( AS.MultiThreaded ) {
                        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                            if ( ifp[2] == ModOptdollars[nummodopt].number ) break;
                        }
                        if ( nummodopt < NumModOptdollars ) {
                            dtype = ModOptdollars[nummodopt].type;
                            if ( dtype == MODLOCAL ) {
                                d = ModOptdollars[nummodopt].dstruct+AT.identity;
                            }
                            else {
                                LOCK(d->pthreadslockread);
                            }
                        }
                    }
#endif
/*
                    We have to pick up the IFDOLLAREXTRA pieces for [1], [$y] etc.
*/
                    if ( ifp+3 < ifstop && ifp[3] == IFDOLLAREXTRA ) {
                        if ( d->nfactors == 0 ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Attempt to use a factor of an unfactored $-variable");
                            MUNLOCK(ErrorMessageLock);
                            Terminate(-1);
                        } {
                        WORD num = GetIfDollarNum(ifp+3,ifstop);
                        WORD *w;
                        while ( ifp+3 < ifstop && ifp[3] == IFDOLLAREXTRA ) ifp += 3;
                        if ( num > d->nfactors ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Dollar factor number %s out of range",num);
                            MUNLOCK(ErrorMessageLock);
                            Terminate(-1);
                        }
                        if ( num == 0 ) {
                            ncoef2 = 1; coef2[0] = d->nfactors; coef2[1] = 1;
                            break;
                        }
                        w = d->factors[num-1].where;
                        if ( w == 0 ) {
                            if ( d->factors[num-1].value < 0 ) {
                                ncoef2 = -1; coef2[0] = -d->factors[num-1].value; coef2[1] = 1;
                            }
                            else {
                                ncoef2 = 1; coef2[0] = d->factors[num-1].value; coef2[1] = 1;
                            }
                            break;
                        }
                        if ( w[*w] == 0 ) {
                            r = w + *w - 1;
                            i = ABS(*r);
                            if ( i == ( *w-1 ) ) {
                                ncoef2 = (i-1)/2;
```

```
                                if ( *r < 0 ) ncoef2 = -ncoef2;
                                i--; cc = coef2; r = w + 1;
                                while ( --i >= 0 ) *cc++ = (UWORD)(*r++);
                                break;
                            }
                        }
                        goto generic;
                        }
                    }
                    else {
                      switch ( d->type ) {
                        case DOLUNDEFINED:
                            if ( AC.UnsureDollarMode == 0 ) {
#ifdef WITHPTHREADS
                                if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                MLOCK(ErrorMessageLock);
                                MesPrint("$%s is undefined",AC.dollarnames->namebuffer+d->name);
                                MUNLOCK(ErrorMessageLock);
                                Terminate(-1);
                            }
                            ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
                            break;
                        case DOLZERO:
                            ncoef2 = coef2[0] = 0; coef2[1] = 1;
                            break;
                        case DOLSUBTERM:
                            if ( d->where[0] != INDEX || d->where[1] != 3
                            || d->where[2] < 0 || d->where[2] >= AM.OffsetIndex ) {
                                if ( AC.UnsureDollarMode == 0 ) {
#ifdef WITHPTHREADS
                                    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                    MLOCK(ErrorMessageLock);
                                    MesPrint("$%s is of wrong type",AC.dollarnames->namebuffer+d->name);
                                    MUNLOCK(ErrorMessageLock);
                                    Terminate(-1);
                                }
                                ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
                                break;
                            }
                            d->index = d->where[2];
                        case DOLINDEX:
                            if ( d->index == 0 ) {
                                ncoef2 = coef2[0] = 0; coef2[1] = 1;
                            }
                            else if ( d->index > 0 && d->index < AM.OffsetIndex ) {
                                ncoef2 = 1; coef2[0] = d->index; coef2[1] = 1;
                            }
                            else if ( AC.UnsureDollarMode == 0 ) {
#ifdef WITHPTHREADS
                                if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                MLOCK(ErrorMessageLock);
                                MesPrint("$%s is of wrong type",AC.dollarnames->namebuffer+d->name);
                                MUNLOCK(ErrorMessageLock);
                                Terminate(-1);
                            }
                            ncoef2 = coef2[0] = 0; coef2[1] = 1;
                            break;
                        case DOLWILDARGS:
                            if ( d->where[0] <= -FUNCTION ||
                            ( d->where[0] < 0 && d->where[2] != 0 )
                            || ( d->where[0] > 0 && d->where[d->where[0]] != 0 )
                            ) {
                                if ( AC.UnsureDollarMode == 0 ) {
#ifdef WITHPTHREADS
                                    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                    MLOCK(ErrorMessageLock);
                                    MesPrint("$%s is of wrong type",AC.dollarnames->namebuffer+d->name);
                                    MUNLOCK(ErrorMessageLock);
                                    Terminate(-1);
                                }
                                ncoef2 = coef2[0] = 0; coef2[1] = 1;
                                break;
                            }
                        case DOLARGUMENT:
                            if ( d->where[0] == -SNUMBER ) {
                                if ( d->where[1] == 0 ) {
                                    ncoef2 = coef2[0] = 0;
                                }
                                else if ( d->where[1] < 0 ) {
```

```
                                    ncoef2 = -1;
                                    coef2[0] = -d->where[1];
                                }
                                else {
                                    ncoef2 = 1;
                                    coef2[0] = d->where[1];
                                }
                                coef2[1] = 1;
                            }
                            else if ( d->where[0] == -INDEX
                            && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
                                if ( d->where[1] == 0 ) {
                                    ncoef2 = coef2[0] = 0; coef2[1] = 1;
                                }
                                else {
                                    ncoef2 = 1; coef2[0] = d->where[1];
                                    coef2[1] = 1;
                                }
                            }
                            else if ( d->where[0] > 0
                            && d->where[ARGHEAD] == (d->where[0]-ARGHEAD)
                            && ABS(d->where[d->where[0]-1]) ==
                                        (d->where[0] - ARGHEAD-1) ) {
                                i = d->where[d->where[0]-1];
                                ncoef2 = (ABS(i)-1)/2;
                                if ( i < 0 ) { ncoef2 = -ncoef2; i = -i; }
                                i--; cc = coef2; r = d->where + ARGHEAD+1;
                                while ( --i >= 0 ) *cc++ = (UWORD)(*r++);
                            }
                            else {
                                if ( AC.UnsureDollarMode == 0 ) {
#ifdef WITHPTHREADS
                                    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                    MLOCK(ErrorMessageLock);
                                    MesPrint("$%s is of wrong type",AC.dollarnames->namebuffer+d->name);
                                    MUNLOCK(ErrorMessageLock);
                                    Terminate(-1);
                                }
                                ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
                            }
                            break;
                        case DOLNUMBER:
                        case DOLTERMS:
                            if ( d->where[d->where[0]] == 0 ) {
                                r = d->where + d->where[0]-1;
                                i = ABS(*r);
                                if ( i == ( d->where[0]-1 ) ) {
                                    ncoef2 = (i-1)/2;
                                    if ( *r < 0 ) ncoef2 = -ncoef2;
                                    i--; cc = coef2; r = d->where + 1;
                                    while ( --i >= 0 ) *cc++ = (UWORD)(*r++);
                                    break;
                                }
                            }
generic:;
                            if ( AC.UnsureDollarMode == 0 ) {
#ifdef WITHPTHREADS
                                if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                MLOCK(ErrorMessageLock);
                                MesPrint("$%s is of wrong type",AC.dollarnames->namebuffer+d->name);
                                MUNLOCK(ErrorMessageLock);
                                Terminate(-1);
                            }
                            ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
                            break;
                        }
                    }
#ifdef WITHPTHREADS
                    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                }
                break;
            case IFEXPRESSION:
                r = ifp+2; j = ifp[1] - 2; ncoef2 = 0;
                while ( --j >= 0 ) {
                    if ( *r == AR.CurExpr ) { ncoef2 = 1; break; }
                    r++;
                }
                coef2[0] = ncoef2;
                coef2[1] = 1;
                break;
            case IFISFACTORIZED:
                r = ifp+2; j = ifp[1] - 2;
```

```
                if ( j == 0 ) {
                    ncoef2 = 0;
                    if ( ( Expressions[AR.CurExpr].vflags & ISFACTORIZED ) != 0 ) {
                        ncoef2 = 1;
                    }
                }
                else {
                    ncoef2 = 1;
                    while ( --j >= 0 ) {
                        if ( ( Expressions[*r].vflags & ISFACTORIZED ) == 0 ) {
                            ncoef2 = 0;
                            break;
                        }
                        r++;
                    }
                }
                coef2[0] = ncoef2;
                coef2[1] = 1;
                break;
            case IFOCCURS:
                {
                    WORD *OccStop = ifp + ifp[1], *ifpp = ifp+2;
                    ncoef2 = 0;
                    while ( ifpp < OccStop ) {
                        if ( FindVar(ifpp,term) == 1 ) {
                            ncoef2 = 1; break;
                        }
                        if ( *ifpp == DOTPRODUCT ) ifp += 3;
                        else ifpp += 2;
                    }
                    coef2[0] = ncoef2;
                    coef2[1] = 1;
                }
                break;
            default:
                break;
        }
        if ( !first ) {
            if ( ifp[-2] != ORCOND && ifp[-2] != ANDCOND ) {
                if ( ( ifp[-2] == EQUAL || ifp[-2] == NOTEQUAL ) &&
                ( ismul2 || ismul1 ) ) {
                    if ( ismul1 && ismul2 ) {
                        if ( coef1[0] == coef2[0] ) i = 1;
                        else i = 0;
                    }
                    else {
                        if ( ismul1 ) {
                            if ( ncoef2 )
                                Divvy(BHEAD coef2,&ncoef2,coef1,ncoef1);
                            cc = coef2; ncoef3 = ncoef2;
                        }
                        else {
                            if ( ncoef1 )
                                Divvy(BHEAD coef1,&ncoef1,coef2,ncoef2);
                            cc = coef1; ncoef3 = ncoef1;
                        }
                        if ( ncoef3 < 0 ) ncoef3 = -ncoef3;
                        if ( ncoef3 == 0 ) {
                            if ( ifp[-2] == EQUAL ) i = 1;
                            else i = 0;
                        }
                        else if ( cc[ncoef3] != 1 ) {
                            if ( ifp[-2] == EQUAL ) i = 0;
                            else i = 1;
                        }
                        else {
                            for ( j = 1; j < ncoef3; j++ ) {
                                if ( cc[ncoef3+j] != 0 ) break;
                            }
                            if ( j < ncoef3 ) {
                                if ( ifp[-2] == EQUAL ) i = 0;
                                else i = 1;
                            }
                            else if ( ifp[-2] == EQUAL ) i = 1;
                            else i = 0;
                        }
                    }
                    goto donemul;
                }
                else if ( AddRat(BHEAD coef1,ncoef1,coef2,-ncoef2,coef3,&ncoef3) ) {
                    NumberFree(coef3,"DoIfStatement"); NumberFree(Spac1,"DoIfStatement");
    TermFree(Spac2,"DoIfStatement");
                    MesCall("DoIfStatement"); return(-1);
                }
                switch ( ifp[-2] ) {
                    case GREATER:
                        if ( ncoef3 > 0 ) i = 1;
```

```
                        else i = 0;
                        break;
                    case GREATEREQUAL:
                        if ( ncoef3 >= 0 ) i = 1;
                        else i = 0;
                        break;
                    case LESS:
                        if ( ncoef3 < 0 ) i = 1;
                        else i = 0;
                        break;
                    case LESSEQUAL:
                        if ( ncoef3 <= 0 ) i = 1;
                        else i = 0;
                        break;
                    case EQUAL:
                        if ( ncoef3 == 0 ) i = 1;
                        else i = 0;
                        break;
                    case NOTEQUAL:
                        if ( ncoef3 != 0 ) i = 1;
                        else i = 0;
                        break;
                }
donemul:        if ( i ) { ncoef2 = 1; coef2 = Spac2; coef2[0] = coef2[1] = 1; }
                else ncoef2 = 0;
                ismul1 = ismul2 = 0;
            }
        }
        else {
            first = 0;
        }
        coef1 = Spac1;
        i = 2*ABS(ncoef2);
        for ( j = 0; j < i; j++ ) coef1[j] = coef2[j];
        ncoef1 = ncoef2;
SkipCond:
        ifp += ifp[1];
    } while ( ifp < ifstop );

    NumberFree(coef3,"DoIfStatement"); NumberFree(Spac1,"DoIfStatement");
TermFree(Spac2,"DoIfStatement");
    if ( ncoef1 ) return(1);
    else return(0);
}

/*
        #] DoIfStatement :
        #[ HowMany :                    WORD HowMany(ifcode,term)

        Returns the number of times that the pattern in ifcode
        can be taken out from term. There is a subkey in ifcode[2];
        The notation is identical to the lhs of an id statement.
        Most of the code comes from TestMatch.
*/

WORD HowMany(PHEAD WORD *ifcode, WORD *term)
{
    GETBIDENTITY
    WORD *m, *t, *r, *w, power, RetVal, i, topje, *newterm;
    WORD *OldWork, *ww, *mm;
    int *RepSto, RepVal;
    int numdollars = 0;
    m = ifcode + IDHEAD;
    AN.FullProto = m;
    AN.WildValue = w = m + SUBEXPSIZE;
    m += m[1];
    AN.WildStop = m;
    OldWork = AT.WorkPointer;
    if ( ( ifcode[4] & 1 ) != 0 ) { /* We have at least one dollar in the pattern */
        AR.Eside = LHSIDEX;
        ww = AT.WorkPointer; i = m[0]; mm = m;
        NCOPY(ww,mm,i);
        *OldWork += 3;
        *ww++ = 1; *ww++ = 1; *ww++ = 3;
        AT.WorkPointer = ww;
        RepSto = AN.RepPoint;
        RepVal = *RepSto;
        NewSort(BHEAD0);
        if ( Generator(BHEAD OldWork,AR.Cnumlhs) ) {
            LowerSortLevel();
            *RepSto = RepVal;
            AN.RepPoint = RepSto;
            AT.WorkPointer = OldWork;
            return(-1);
        }
        AT.WorkPointer = ww;
        if ( EndSort(BHEAD ww,0) < 0 ) {}
```

```
        *RepSto = RepVal;
        AN.RepPoint = RepSto;
        if ( *ww == 0 || *(ww+*ww) != 0 ) {
            if ( AP.lhdollarerror == 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("&LHS must be one term");
                MUNLOCK(ErrorMessageLock);
                AP.lhdollarerror = 1;
            }
            AT.WorkPointer = OldWork;
            return(-1);
        }
        m = ww; AT.WorkPointer = ww = m + *m;
        if ( m[*m-1] < 0 ) { m[*m-1] = -m[*m-1]; }
        *m -= m[*m-1];
        AR.Eside = RHSIDE;
    }
    else {
        ww = term + *term;
        if ( AT.WorkPointer < ww ) AT.WorkPointer = ww;
    }
    ClearWild(BHEAD0);
    while ( w < AN.WildStop ) {
        if ( *w == LOADDOLLAR ) numdollars++;
        w += w[1];
    }
    AN.RepFunNum = 0;
    AN.RepFunList = AT.WorkPointer;
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    topje = cbuf[AT.ebufnum].numrhs;
    if ( AT.WorkPointer >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    AN.DisOrderFlag = ifcode[2] & SUBDISORDER;
    switch ( ifcode[2] & (~SUBDISORDER) ) {
        case SUBONLY :
            /* Must be an exact match */
            AN.UseFindOnly = 1; AN.ForFindOnly = 0;
/*
        Copy the term first to scratchterm. This is needed
        because of the Substitute.
*/
            i = *term;
            t = term; newterm = r = AT.WorkPointer;
            NCOPY(r,t,i); AT.WorkPointer = r;
            RetVal = 0;
            if ( FindRest(BHEAD newterm,m) && ( AN.UsedOtherFind ||
                FindOnly(BHEAD newterm,m) ) ) {
                Substitute(BHEAD newterm,m,1);
                if ( numdollars ) {
                    WildDollars(BHEAD (WORD *)0);
                    numdollars = 0;
                }
                ClearWild(BHEAD0);
                RetVal = 1;
            }
            else RetVal = 0;
            break;
        case SUBMANY :
/*
        Copy the term first to scratchterm. This is needed
        because of the Substitute.
*/
            i = *term;
            t = term; newterm = r = AT.WorkPointer;
            NCOPY(r,t,i); AT.WorkPointer = r;
            RetVal = 0;
            AN.UseFindOnly = 0;
            if ( ( power = FindRest(BHEAD newterm,m) ) > 0 ) {
                if ( ( power = FindOnce(BHEAD newterm,m) ) > 0 ) {
                    AN.UseFindOnly = 0;
                    do {
                        Substitute(BHEAD newterm,m,1);
                        if ( numdollars ) {
                            WildDollars(BHEAD (WORD *)0);
                            numdollars = 0;
                        }
                        ClearWild(BHEAD0);
                        RetVal++;
                    } while ( FindRest(BHEAD newterm,m) && (
                        AN.UsedOtherFind || FindOnce(BHEAD newterm,m) ) );
                }
                else if ( power < 0 ) {
                    do {
```

```
                        Substitute(BHEAD newterm,m,1);
                        if ( numdollars ) {
                            WildDollars(BHEAD (WORD *)0);
                            numdollars = 0;
                        }
                        ClearWild(BHEAD0);
                        RetVal++;
                    } while ( FindRest(BHEAD newterm,m) );
                }
            }
            else if ( power < 0 ) {
                if ( FindOnce(BHEAD newterm,m) ) {
                    do {
                        Substitute(BHEAD newterm,m,1);
                        if ( numdollars ) {
                            WildDollars(BHEAD (WORD *)0);
                            numdollars = 0;
                        }
                        ClearWild(BHEAD0);
                    } while ( FindOnce(BHEAD newterm,m) );
                    RetVal = 1;
                }
            }
            break;
        case SUBONCE :
/*
        Copy the term first to scratchterm. This is needed
        because of the Substitute.
*/
            i = *term;
            t = term; newterm = r = AT.WorkPointer;
            NCOPY(r,t,i); AT.WorkPointer = r;
            RetVal = 0;
            AN.UseFindOnly = 0;
            if ( FindRest(BHEAD newterm,m) && ( AN.UsedOtherFind || FindOnce(BHEAD newterm,m) ) ) {
                Substitute(BHEAD newterm,m,1);
                if ( numdollars ) {
                    WildDollars(BHEAD (WORD *)0);
                    numdollars = 0;
                }
                ClearWild(BHEAD0);
                RetVal = 1;
            }
            else RetVal = 0;
            break;
        case SUBMULTI :
            RetVal = FindMulti(BHEAD term,m);
            break;
        case SUBVECTOR :
            RetVal = 0;
            for ( i = 0; i < *term; i++ ) ww[i] = term[i];
            while ( ( power = FindAll(BHEAD ww,m,AR.Cnumlhs,ifcode) ) != 0 ) { RetVal += power; }
            break;
        case SUBSELECT :
            ifcode += IDHEAD;   ifcode += ifcode[1];    ifcode += *ifcode;
            AN.UseFindOnly = 1; AN.ForFindOnly = ifcode;
            if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind ||
                    FindOnly(BHEAD term,m) ) ) RetVal = 1;
            else RetVal = 0;
            break;
        default :
            RetVal = 0;
            break;
    }
    AT.WorkPointer = AN.RepFunList;
    cbuf[AT.ebufnum].numrhs = topje;
    return(RetVal);
}

/*
        #] HowMany :
        #[ DoubleIfBuffers :
*/

VOID DoubleIfBuffers()
{
    int newmax, i;
    WORD *newsumcheck;
    LONG *newheap, *newifcount;
    if ( AC.MaxIf == 0 ) newmax = 10;
    else                 newmax = 2*AC.MaxIf;
    newheap = (LONG *)Malloc1(sizeof(LONG)*(newmax+1),"IfHeap");
    newsumcheck = (WORD *)Malloc1(sizeof(WORD)*(newmax+1),"IfSumCheck");
    newifcount = (LONG *)Malloc1(sizeof(LONG)*(newmax+1),"IfCount");
    if ( AC.MaxIf ) {
        for ( i = 0; i < AC.MaxIf; i++ ) {
            newheap[i] = AC.IfHeap[i];
```

```
            newsumcheck[i] = AC.IfSumCheck[i];
            newifcount[i] = AC.IfCount[i];
        }
        AC.IfStack = (AC.IfStack-AC.IfHeap) + newheap;
        M_free(AC.IfHeap,"AC.IfHeap");
        M_free(AC.IfCount,"AC.IfCount");
        M_free(AC.IfSumCheck,"AC.IfSumCheck");
    }
    else {
        AC.IfStack = newheap;
    }
    AC.IfHeap = newheap;
    AC.IfSumCheck = newsumcheck;
    AC.IfCount = newifcount;
    AC.MaxIf = newmax;
}

/*
        #] DoubleIfBuffers :
    #] If statement :
*/
```

4.45 index.c File Reference

`#include "form3.h"`

Include dependency graph for index.c:

Functions

- POSITION * FindBracket (WORD nexp, WORD *bracket)
- VOID PutBracketInIndex (PHEAD WORD *term, POSITION *newpos)
- void ClearBracketIndex (WORD numexp)
- VOID OpenBracketIndex (WORD nexpr)
- int PutInside (PHEAD WORD *term, WORD *code)

4.45.1 Detailed Description

The routines that deal with bracket indexing It creates the bracket index and it can find the brackets using the index.

Definition in file index.c.

4.45.2 Function Documentation

4.45.2.1 FindBracket()

```
POSITION * FindBracket (
            WORD nexp,
            WORD * bracket )
```

Definition at line 65 of file index.c.

4.45.2.2 PutBracketInIndex()

```
VOID PutBracketInIndex (
            PHEAD WORD * term,
            POSITION * newpos )
```

Definition at line 331 of file index.c.

4.45.2.3 ClearBracketIndex()

```
void ClearBracketIndex (
            WORD numexp )
```

Definition at line 546 of file index.c.

4.45.2.4 OpenBracketIndex()

```
VOID OpenBracketIndex (
            WORD nexpr )
```

Definition at line 578 of file index.c.

4.45.2.5 PutInside()

```
int PutInside (
            PHEAD WORD * term,
            WORD * code )
```

Definition at line 609 of file index.c.

4.46 index.c

Go to the documentation of this file.

```
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
```

```c
/* #] License : */

/*
    #[ Includes : index.c
*/

#include "form3.h"

/*
    #] Includes :
    #[ syntax and use :

    The indexing of brackets is not automatic! It should only be used
    when one intends to use the contents of individual brackets.
    This is done with the addition of a + sign to the bracket statement:
    B+ a,b,c; or AB+ a,b,c;
    It does require resources! The index is kept in memory and is removed
    once the expression is treated and passed on to the output with different
    or no brackets.
    The index is limited to a given amount of space. Hence if there are too
    many brackets we will skip some in the index. Skipping goes by space
    occupied by the contents. We take the two adjacent bracket(s) with the
    least space together and represent them by the first one only. This gives
    a new spot.
    The expression struct has two pointers:
    bracketinfo      for using.
    newbracketinfo   for making new index.

    #] syntax and use :
    #[ FindBracket :
*/

POSITION *FindBracket(WORD nexp, WORD *bracket)
{
    GETIDENTITY
    BRACKETINDEX *bi;
    BRACKETINFO *bracketinfo;
    EXPRESSIONS e = &(Expressions[nexp]);
    LONG hi, low, med;
    int i;
    WORD oldsorttype = AR.SortType, *t1, *t2, j, bsize, *term, *p, *pstop, *pp;
    WORD *tstop, *cp, a[4], *bracketh;
    FILEHANDLE *fi;
    POSITION auxpos, toppos;
    switch ( e->status ) {
        case UNHIDELEXPRESSION:
        case UNHIDEGEXPRESSION:
        case DROPHLEXPRESSION:
        case DROPHGEXPRESSION:
        case HIDDENLEXPRESSION:
        case HIDDENGEXPRESSION:
            fi = AR.hidefile;
            break;
        default:
            fi = AR.infile;
            break;
    }
    if ( AT.bracketinfo ) bracketinfo = AT.bracketinfo;
    else                  bracketinfo = e->bracketinfo;
    hi = bracketinfo->indexfill; low = 0;
    if ( hi <= 0 ) return(0);
/*
    The next code is needed for a problem with sorting when there are
    only functions outside the bracket. This gives ordinarily the wrong
    sorting. We solve that by taking HAAKJE along with the outside while
    running the Compare. But this means that we need to copy bracket.
*/
    bracketh = TermMalloc("FindBracket");
    i = *bracket; p = bracket; pp = bracketh; NCOPY(pp,p,i)
    pp -= 3; *pp++ = HAAKJE; *pp++ = 3; *pp++ = 0; *pp++ = 1; *pp++ = 1; *pp++ = 3;
    *bracketh += 3;

    AT.fromindex = 1;
    AR.SortType = bracketinfo->SortType;
    bi = bracketinfo->indexbuffer + hi - 1;
    if ( *bracketh == 7 ) {
        if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 0;
        else i = -1;
    }
    else if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 1;
    else i = CompareTerms(BHEAD bracketh,bracketinfo->bracketbuffer+bi->bracket,0);
    if ( i < 0 ) {
        AR.SortType = oldsorttype;
        AT.fromindex = 0;
        TermFree(bracketh,"FindBracket");
        return(0);
    }
```

```
    else if ( i == 0 ) med = hi-1;
    else {
        for (;;) {
        med = (hi+low)/2;
        bi = bracketinfo->indexbuffer + med;
        if ( *bracketh == 7 ) {
            if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 0;
            else i = -1;
        }
        else if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 1;
        else i = CompareTerms(BHEAD bracketh,bracketinfo->bracketbuffer+bi->bracket,0);
        if ( i == 0 ) { break; }
        if ( i > 0 ) {
            if ( low == med ) { /* no occurrence */
                AR.SortType = oldsorttype;
                AT.fromindex = 0;
                TermFree(bracketh,"FindBracket");
                return(0);
            }
            hi = med;
        }
        else if ( i < 0 ) {
            if ( low == med ) break;
            low = med;
        }
    }}
/*
    The bracket is now either bi itself or between bi and the next one
    or it is not present at all.
*/
    AN.theposition = AS.OldOnFile[nexp];
    ADD2POS(AN.theposition,bi->start);
/*
    The seek will have to move closer to the actual read so that we
    can place a lock around the two.
    if ( fi->handle >= 0 ) SeekFile(fi->handle,&AN.theposition,SEEK_SET);
    else SetScratch(fi,&AN.theposition);
*/
/*
    Put the bracket in the compress buffer as if it were the last term read.
    Have a look at AR.CompressPointer. (set it right)
*/
    term = AT.WorkPointer;
    t1 = bracketinfo->bracketbuffer+bi->bracket;
    j = *t1;
/*
    Note that in the bracketbuffer, the bracket sits with HAAKJE.

    The next is (hopefully) a bug fix. Originally the code read bsize = j
    but that overcounts one. We have the part outside the bracket and the
    coefficient which is 1,1,3. But we also have the length indicator.
    Where we use the variable bsize we do not include the length indicator,
    and we have the part outside plus 7,3,0 which is also three words.
*/
    bsize = j-1;
    t2 = AR.CompressPointer;
    NCOPY(t2,t1,j)
    if ( i == 0 ) { /* We found the proper bracket already */
        AR.SortType = oldsorttype;
        AT.fromindex = 0;
        TermFree(bracketh,"FindBracket");
        return(&AN.theposition);
    }
/*
    Here we have to skip to the bracket if it exists (!)
    Let us first look whether the expression is in memory.
    If not we have to make a buffer to increase speed..
*/
    if ( fi->handle < 0 ) {
        p = (WORD *)((UBYTE *)(fi->PObuffer)
             + BASEPOSITION(AS.OldOnFile[nexp])
             + BASEPOSITION(bi->start));
        pstop = (WORD *)((UBYTE *)(fi->PObuffer)
             + BASEPOSITION(AS.OldOnFile[nexp])
             + BASEPOSITION(bi->next));
        while ( p < pstop ) {
/*
            Check now: if size or part from previous term < size of bracket
                    we have to setup the bracket again and test.
                    Otherwise, skip immediately to the next term.
*/
            if ( *p <= -bsize ) {   /* no change of bracket */
                p++; p += *p + 1;
            }
            else if ( *p < 0 ) {    /* changes bracket */
                pp = p;
                t2 = AR.CompressPointer;
```

```
                t1 = t2 - *p++ + 1;
                j = *p++;
                NCOPY(t1,p,j)
                t2++; while ( *t2 != HAAKJE ) t2 += t2[1];
                t2 += t2[1];
                a[1] = t2[0]; a[2] = t2[1]; a[3] = t2[2];
                *t2++ = 1; *t2++ = 1; *t2++ = 3;
                *AR.CompressPointer = t2 - AR.CompressPointer;
                bsize = *AR.CompressPointer - 1;
                if ( *bracketh == 7 ) {
                    if ( AR.CompressPointer[0] == 7 ) i = 0;
                    else i = -1;
                }
                else if ( AR.CompressPointer[0] == 7 ) i = 1;
                else i = CompareTerms(BHEAD bracketh,AR.CompressPointer,0);
                t2[-3] = a[1]; t2[-2] = a[2]; t2[-1] = a[3];
                if ( i == 0 ) {
                    SETBASEPOSITION(AN.theposition,(pp-fi->PObuffer)*sizeof(WORD));
                    fi->POfill = pp;
                    goto found;
                }
                if ( i > 0 ) break; /* passed what was possible */
            }
            else {  /* no compression. We have to check! */
                WORD *oldworkpointer = AT.WorkPointer, *t3, *t4;
                t2 = p + 1; while ( *t2 != HAAKJE ) t2 += t2[1];
                t2 += t2[1];
/*
                Here we need to copy the term. Modifying has proven to
                be NOT threadsafe.
*/
                t3 = oldworkpointer; t4 = p;
                while ( t4 < t2 ) *t3++ = *t4++;
                *t3++ = 1; *t3++ = 1; *t3++ = 3;
                *oldworkpointer = t3 - oldworkpointer;
                bsize = *oldworkpointer - 1;
                AT.WorkPointer = t3;
                t3 = oldworkpointer;
                if ( *bracketh == 7 ) {
                    if ( t3[0] == 7 ) i = 0;
                    else i = -1;
                }
                else if ( t3[0] == 7 ) i = 1;
                else {
                    i = CompareTerms(BHEAD bracketh,t3,0);
                }
                AT.WorkPointer = oldworkpointer;
                if ( i == 0 ) {
                    SETBASEPOSITION(AN.theposition,(p-fi->PObuffer)*sizeof(WORD));
                    fi->POfill = p;
                    goto found;
                }
                if ( i > 0 ) break; /* passed what was possible */
                p += *p;
            }
        }
        AR.SortType = oldsorttype;
        AT.fromindex = 0;
        TermFree(bracketh,"FindBracket");
        return(0);  /* Bracket does not exist */
    }
    else {
/*
        In this case we can work with the old representation without HAAKJE.
        We stop searching when we reach toppos and we do not call Compare.
*/
        toppos = AS.OldOnFile[nexp];
        ADD2POS(toppos,bi->next);
        cp = AR.CompressPointer;
        for(;;) {
            auxpos = AN.theposition;
            GetOneTerm(BHEAD term,fi,&auxpos,0);
            if ( *term == 0 ) {
                AR.SortType = oldsorttype;
                AT.fromindex = 0;
                return(0);  /* Bracket does not exist */
            }
            tstop = term + *term;
            tstop -= ABS(tstop[-1]);
            t1 = term + 1;
            while ( *t1 != HAAKJE && t1 < tstop ) t1 += t1[1];
            i = *bracket-4;
            if ( t1-term == *bracket-3 ) {
                t1 = term + 1; t2 = bracket+1;
                while ( i > 0 && *t1 == *t2 ) { t1++; t2++; i--; }
                if ( i <= 0 ) {
                    AR.CompressPointer = cp;
```

```
                    goto found;
                }
            }
            AR.CompressPointer = cp;
            AN.theposition = auxpos;
/*
            Now check whether we passed the 'point'
*/
            if ( ISGEPOS(AN.theposition,toppos) ) {
                AR.SortType = oldsorttype;
                AR.CompressPointer = cp;
                AT.fromindex = 0;
                TermFree(bracketh,"FindBracket");
                return(0);  /* Bracket does not exist */
            }
        }
    }
found:
    AR.SortType = oldsorttype;
    AT.fromindex = 0;
    TermFree(bracketh,"FindBracket");
    return(&AN.theposition);
}

/*
    #] FindBracket :
    #[ PutBracketInIndex :

    Call via
    if ( AR.BracketOn ) PutBracketInIndex(BHEAD term);

    This means that there should be a bracket somewhere
    Note that the brackets come in in proper order.

    DON'T forget AR.SortType to be put into e->bracketinfo->SortType
*/

VOID PutBracketInIndex(PHEAD WORD *term, POSITION *newpos)
{
    GETBIDENTITY
    BRACKETINDEX *bi, *b1, *b2, *b3;
    BRACKETINFO *b;
    POSITION thepos;
    EXPRESSIONS e = Expressions + AR.CurExpr;
    LONG hi, i, average;
    WORD *t, *tstop, *t1, *t2, *oldt, oldsize, a[4];
    if ( ( b = e->newbracketinfo ) == 0 ) return;
    DIFPOS(thepos,*newpos,e->onfile);
    tstop = term + *term;
    tstop -= ABS(tstop[-1]);
    t = term+1;
    while ( *t != HAAKJE && t < tstop ) t += t[1];
    if ( t >= tstop ) return; /* no ticket, no laundry */
    t += t[1];  /* include HAAKJE for the sorting */
    a[0] = t[0]; a[1] = t[1]; a[2] = t[2];
    oldt = t; oldsize = *term; *t++ = 1; *t++ = 1; *t++ = 3;
    *term = t - term;
    AT.fromindex = 1;
/*
    Check now with the last bracket in the buffer.
    If it is the same we can abort.
*/
    hi = b->indexfill;
    if ( hi > 0 ) {
        bi = b->indexbuffer + hi - 1;
        bi->next = thepos;
        if ( *term == 7 ) {
            if ( b->bracketbuffer[bi->bracket] == 7 ) i = 0;
            else i = -1;
        }
        else if ( b->bracketbuffer[bi->bracket] == 7 ) i = 1;
        else i = CompareTerms(BHEAD term,b->bracketbuffer+bi->bracket,0);
        if ( i == 0 ) { /* still the same bracket */
            bi->termsinbracket++;
            goto bracketdone;
        }
        if ( i > 0 ) { /* We have a problem */
/*
            There is a special case in which we have only functions and
            term is contained completely in the bracket
*/
/*
            t = term + 1;
            tstop = term + *term - 3;
            while ( t < tstop && *t > HAAKJE ) t += t[1];
            if ( t < tstop ) goto problems;
*/
```

```
            for ( i = 1; i < *term - 3; i++ ) {
                if ( term[i] != b->bracketbuffer[bi->bracket+i] ) break;
            }
            if ( i < *term - 3 ) {
/*
problems:;
*/
                *term = oldsize; oldt[0] = a[0]; oldt[1] = a[1]; oldt[2] = a[2];
                MLOCK(ErrorMessageLock);
                MesPrint("Error!!!! Illegal bracket sequence detected in PutBracketInIndex");
#ifdef WITHPTHREADS
                MesPrint("Worker = %w");
#endif
                PrintTerm(term,"term into index");
                PrintTerm(b->bracketbuffer+bi->bracket,"Last in index");
                MUNLOCK(ErrorMessageLock);
                AT.fromindex = 0;
                Terminate(-1);
            }
            i = -1;
        }
    }
/*
    If there is room for more brackets, we add this one.
*/
    if ( b->bracketfill+*term >= b->bracketbuffersize
    && ( b->bracketbuffersize < AM.MaxBracketBufferSize
    || ( e->vflags & ISFACTORIZED ) != 0 ) ) {
/*
        Enlarge bracket buffer
*/
        WORD *oldbracketbuffer = b->bracketbuffer;
        i = MaX(b->bracketbuffersize * 2, b->bracketfill+*term+1);
        if ( i > AM.MaxBracketBufferSize && ( e->vflags & ISFACTORIZED ) == 0 )
                i = AM.MaxBracketBufferSize;
        if ( i > b->bracketfill+*term ) {
            b->bracketbuffersize = i;
            b->bracketbuffer = (WORD *)Malloc1(b->bracketbuffersize*sizeof(WORD),
                                "new bracket buffer");
            t1 = b->bracketbuffer; t2 = oldbracketbuffer;
            i = b->bracketfill;
            NCOPY(t1,t2,i)
            if ( oldbracketbuffer ) M_free(oldbracketbuffer,"old bracket buffer");
        }
    }
    if ( b->bracketfill+*term < b->bracketbuffersize ) {
        if ( b->indexfill >= b->indexbuffersize ) {
/*
            Enlarge index
*/
            BRACKETINDEX *oldindexbuffer = b->indexbuffer;
            b->indexbuffersize *= 2;
            b->indexbuffer = (BRACKETINDEX *)
                Malloc1(b->indexbuffersize*sizeof(BRACKETINDEX),"new bracket index");
            b1 = b->indexbuffer; b2 = oldindexbuffer;
            i = b->indexfill;
            NCOPY(b1,b2,i)
            if ( oldindexbuffer ) M_free(oldindexbuffer,"old bracket index");
        }
    }
    else {
/*
        We have too many brackets in the buffer. Try to improve.
        This is the interesting algorithm. We try to eliminate about 1/4 to
        1/2 of the brackets from the index. This should be done by size of
        the bracket contents to make the searching as fast as possible.
        But! Do not touch the last bracket.
        Note that we are always filling from the back.
        Algorithm: Throw away every second bracket, unless b1+b2 is much longer
        than average. How much is something we can tune.
*/
        average = DIVPOS(thepos,b->indexfill+1);
        if ( average <= 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Problems with bracket buffer. Increase MaxBracketBufferSize in form.set");
            MesPrint("Current size is %l",AM.MaxBracketBufferSize*sizeof(WORD));
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        average *= 4;  /* 2*2: one 2 for much longer, one 2 because we have pairs */
        t2 = b->bracketbuffer;
        b3 = b1 = b->indexbuffer;
        bi = b->indexbuffer + b->indexfill;
        b2 = b1+1;
        while ( b2+2 < bi ) {
            if ( DIFBASE(b2->next,b1->start) > average ) {
                t1 = b->bracketbuffer + b1->bracket;
```

```
                b1->bracket = t2 - b->bracketbuffer;
                i = *t1; NCOPY(t2,t1,i)
                *b3++ = *b1;
                t1 = b->bracketbuffer + b2->bracket;
                b2->bracket = t2 - b->bracketbuffer;
                i = *t1; NCOPY(t2,t1,i)
                *b3++ = *b2;
                if ( b3 <= b1 ) {
                    PUTZERO(b1->start);
                    PUTZERO(b1->next);
                    b1->bracket = 0;
                    b1->termsinbracket = 0;
                }
                if ( b3 <= b2 ) {
                    PUTZERO(b2->start);
                    PUTZERO(b2->next);
                    b2->bracket = 0;
                    b2->termsinbracket = 0;
                }
            }
            else {
                t1 = b->bracketbuffer + b1->bracket;
                b1->bracket = t2 - b->bracketbuffer;
                i = *t1; NCOPY(t2,t1,i)
                b1->next = b2->next;
                b1->termsinbracket += b2->termsinbracket;
                *b3++ = *b1;
                if ( b3 <= b1 ) {
                    PUTZERO(b1->start);
                    PUTZERO(b1->next);
                    b1->bracket = 0;
                    b1->termsinbracket = 0;
                }
                PUTZERO(b2->start);
                PUTZERO(b2->next);
                b2->bracket = 0;
                b2->termsinbracket = 0;
            }
            b1 += 2; b2 += 2;
        }
        while ( b1 < bi ) {
            t1 = b->bracketbuffer + b1->bracket;
            b1->bracket = t2 - b->bracketbuffer;
            i = *t1; NCOPY(t2,t1,i)
            *b3++ = *b1;
            if ( b3 <= b1 ) {
                PUTZERO(b1->start);
                PUTZERO(b1->next);
                b1->bracket = 0;
                b1->termsinbracket = 0;
            }
            b1++;
        }
        b->indexfill = b3 - b->indexbuffer;
        b->bracketfill = t2 - b->bracketbuffer;
    }
    bi = b->indexbuffer + b->indexfill;
    b->indexfill++;
    bi->bracket = b->bracketfill;
    bi->start = thepos;
    bi->next = thepos;
    bi->termsinbracket = 1;
/*
    Copy the bracket into the buffer
*/
    t1 = term; t2 = b->bracketbuffer + bi->bracket; i = *t1;
    b->bracketfill += i;
    NCOPY(t2,t1,i)
bracketdone:
    *term = oldsize; oldt[0] = a[0]; oldt[1] = a[1]; oldt[2] = a[2];
    AT.fromindex = 0;
}

/*
    #] PutBracketInIndex :
    #[ ClearBracketIndex :
*/

void ClearBracketIndex(WORD numexp)
{
    BRACKETINFO *b;
    if ( numexp >= 0 ) {
        b = Expressions[numexp].bracketinfo;
        Expressions[numexp].bracketinfo = 0;
    }
    else if ( numexp == -1 ) {
        GETIDENTITY
```

```
        b = AT.bracketinfo;
        AT.bracketinfo = 0;
    }
    else {
        numexp = -numexp-2;
        b = Expressions[numexp].newbracketinfo;
        Expressions[numexp].newbracketinfo = 0;
    }
    if ( b == 0 ) return;
    b->indexfill = b->indexbuffersize = 0;
    b->bracketfill = b->bracketbuffersize = 0;
    M_free(b->bracketbuffer,"ClearBracketBuffer");
    M_free(b->indexbuffer,"ClearIndexBuffer");
    M_free(b,"BracketInfo");
}

/*
    #] ClearBracketIndex :
    #[ OpenBracketIndex :

    Note: This routine is thread-safe
*/

VOID OpenBracketIndex(WORD nexpr)
{
    EXPRESSIONS e = Expressions + nexpr;
    BRACKETINFO *bi;
    LONG i;
    bi = (BRACKETINFO *)Malloc1(sizeof(BRACKETINFO),"BracketInfo");
    e->newbracketinfo = bi;
/*
    i = 20*AM.MaxTer/sizeof(WORD);
    if ( i < 1000 ) i = 1000;
*/
    i = 2000;
    bi->bracketbuffer = (WORD *)Malloc1(i*sizeof(WORD),"Bracket Buffer");
    bi->bracketbuffersize = i;
    bi->bracketfill = 0;
    i = 50;
    bi->indexbuffer = (BRACKETINDEX *)Malloc1(i*sizeof(BRACKETINDEX),"Bracket Index");
    bi->indexbuffersize = i;
    bi->indexfill = 0;
    bi->SortType = AC.SortType;
}

/*
    #] OpenBracketIndex :
    #[ PutInside :

    Puts a term, or a bracket determined part of a term inside a function.

    AT.WorkPointer points at term+*term
*/

int PutInside(PHEAD WORD *term, WORD *code)
{
    WORD *from, *to, *oldbuf, *tStop, *t, *tt, oldon, oldact, inc, argsize, *termout;
    int i, ii, error;

    if ( code[1] == 4 && ( code[2] == 0 || code[2] == 1 ) ) {
/*
        Put all inside. Move the term by 1+FUNHEAD+ARGHEAD
*/
        from = term+*term; to = from+1+ARGHEAD+FUNHEAD; i = ii = *term;
        to[0] = 1; to[1] = 1; to[2] = 3;
        while ( --i >= 0 ) *--to = *--from;
        to = term;
        *to++ = term[0]+4+ARGHEAD+FUNHEAD;
        *to++ = code[3];
        *to++ = ii+FUNHEAD+ARGHEAD;
        *to++ = 1;  /* set dirty flags, because there could be a fast notation */
        FILLFUN3(to)
        *to++ = ii+ARGHEAD;
        *to++ = 1;
        FILLARG(to)
        return(0);
    }
/*
    First we save the old bracket variables. Then we set variables to
    influence the PutBracket routine and call it.
    After that we set the values back and sort out the results by placing the
    inside of the bracket inside the function.
*/
    termout = AT.WorkPointer;
    oldbuf = AT.BrackBuf;
    oldon  = AR.BracketOn;
    oldact = AT.PolyAct;
```

```
    AR.BracketOn = -code[2];
    AT.BrackBuf  = code+4;
    AT.PolyAct = 0;
    error = PutBracket(BHEAD term);
    AT.PolyAct = oldact;
    AT.BrackBuf = oldbuf;
    AR.BracketOn = oldon;
    if ( error ) return(error);
    i = *termout; from = termout; to = term;
    NCOPY(to,from,i);
    tStop = term +*term; tStop -= tStop[-1];
    t = term+1;
    while ( t < tStop && *t != HAAKJE ) t += t[1];
    from = term + *term;
    inc = FUNHEAD+ARGHEAD-t[1]+1;
    tt = t + t[1];
    argsize = from-tt+1;
    to = from + inc;
    to[0] = 1;
    to[1] = 1;
    to[2] = 3;
    while ( from > tt ) *--to = *--from;
    *--to = argsize;
    *t++ = code[3];
    *t++ = argsize+FUNHEAD+ARGHEAD;
    *t++ = 1;
    FILLFUN3(t);
    *t++ = argsize+ARGHEAD;
    *t++ = 1;
    FILLARG(t);
    *term += inc+3;
    AT.WorkPointer = term+*term;
    if ( Normalize(BHEAD term) ) error = 1;
    return(error);
}

/*
    #] PutInside :
*/

/*
    The next routines are for indexing the local output files in a parallel
    sort. This indexing is needed to get a fast determination of the
    splitting terms needed to divide the terms evenly over the processors.
    Actually this method works well for ParFORM, but may not work well
    for TFORM.

    #[ PutTermInIndex :

    Puts a term in the term index.
    Action:
        if the index hasn't reached its full size
            if there is room, put the term
            if there is no room: extend the buffer, put the term
        else
            check if the last term has a number of the type skip*m+1
                if no, overwrite the last term
                if yes, check whether there is room for one more term
                    yes: add the term
                    no: drop all even terms, compress the list,
                        multiply skip by 2, and add this term.

int PutTermInIndex(WORD *term,POSITION *position)
{
    return(0);
}

    #] PutTermInIndex :
*/
```

4.47 inivar.h File Reference

This graph shows which files directly or indirectly include this file:

Data Structures

- struct **fixedfun**

Variables

- FIXEDGLOBALS FG
- ALLGLOBALS A
- FIXEDSET fixedsets [ ]
- UBYTE BufferForOutput [MAXLINELENGTH+14]
- char ∗ setupfilename = "form.set"

4.47.1 Detailed Description

Contains the initialization of a number of structs at compile time This file should only be included in the file startup.c !!!

Definition in file inivar.h.

4.47.2 Variable Documentation

4.47.2.1 FG

```
FIXEDGLOBALS FG
```

Definition at line 34 of file inivar.h.

4.47.2.2 A

```
ALLGLOBALS A
```

Definition at line 131 of file inivar.h.

4.47.2.3 fixedsets

```
FIXEDSET fixedsets[]
```

Initial value:

```
= {
     {"pos_",   "integers > 0",  CSYMBOL, 0}
    ,{"pos0_",  "integers >= 0", CSYMBOL, 0}
    ,{"neg_",   "integers < 0",  CSYMBOL, 0}
    ,{"neg0_",  "integers <= 0", CSYMBOL, 0}
    ,{"even_",  "even integers", CSYMBOL, 0}
    ,{"odd_",   "odd integers",  CSYMBOL, 0}
    ,{"int_",   "all integers",  CSYMBOL, 0}
    ,{"symbol_","only symbols",  CSYMBOL, MAXPOSITIVE}
    ,{"fixed_", "fixed indices", CINDEX, 0}
    ,{"index_", "all indices",   CINDEX, 0}
    ,{"number_","all rationals", CSYMBOL, 0}
    ,{"dummyindices_", "dummy indices", CINDEX, 0}
    ,{"vector_","only vectors",  CVECTOR, 0}
}
```

Definition at line 236 of file inivar.h.

4.47.2.4 BufferForOutput

```
UBYTE BufferForOutput[MAXLINELENGTH+14]
```

Definition at line 252 of file inivar.h.

4.47.2.5 setupfilename

```
char* setupfilename = "form.set"
```

Definition at line 254 of file inivar.h.

4.48 inivar.h

Go to the documentation of this file.

```
00001
00007 /* #[ License : */
00008 /*
00009  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00010  *   When using this file you are requested to refer to the publication
00011  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  *   This is considered a matter of courtesy as the development was paid
00013  *   for by FOM the Dutch physics granting agency and we would like to
00014  *   be able to track its scientific use to convince FOM of its value
00015  *   for the community.
00016  *
00017  *   This file is part of FORM.
00018  *
00019  *   FORM is free software: you can redistribute it and/or modify it under the
00020  *   terms of the GNU General Public License as published by the Free Software
00021  *   Foundation, either version 3 of the License, or (at your option) any later
00022  *   version.
00023  *
00024  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00027  *   details.
00028  *
00029  *   You should have received a copy of the GNU General Public License along
00030  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #] License : */
00033
```

```
FIXEDGLOBALS FG = {
         {Traces                /* Dummy as zeroeth argument */
        ,Traces
        ,EpfCon
        ,RatioGen
        ,SymGen
        ,TenVec
        ,DoSumF1
        ,DoSumF2}

        ,{TraceFind             /* Dummy as zeroeth argument */
        ,TraceFind
        ,EpfFind
        ,RatioFind
        ,SymFind
        ,TenVecFind}

        ,{"symbol"
        ,"index"
        ,"vector"
        ,"function"
        ,"set"
        ,"expression"
        ,"dotproduct"
        ,"number"
        ,"subexp"
        ,"delta"}

        ,{"(local)"
        ,"(skip/local)"
        ,"(drop/local)"
        ,"(dropped)"
        ,"(global)"
        ,"(skip/global)"
        ,"(drop/global)"
        ,"(dropped)"
        ,"(stored)"
        ,"(local-hidden)"
        ,"(local-to be hidden)"
        ,"(local-hidden-dropped)"
        ,"(local-to be unhidden)"
        ,"(global-hidden)"
        ,"(global-to be hidden)"
        ,"(global-hidden-dropped)"
        ,"(global-to be unhidden)"
        ,"(into-hide-local)"
        ,"(into-hide-global)"
        ,"(spectator)"
        ,"(drop/spectator)"}

        ,{" Functions"
        ," Commuting Functions"}

        ,{"left     "
        ,"active   "
        ,"in output"}

    ,(char *)0
    ,(char *)0
    ,(UBYTE *)"1"
    ,(WORD)0
    ,(WORD)0

/*  ASCII table of character types. Note that on some computers this
    table may be different from the ASCII table.

    -1  Illegal character
     0  Alphabetic character
     1  Digit
     2  . $ _ ? # or '
     3  [,]
     4  ( ) = ; or ,
     5  + - * % / ^ :
     6  blank, tab, linefeed
     7  {,|,}
     8  ! & < >
     9  "
    10  The ultimate end.
*/
    ,{    10,255,255,255,255,255,255,255,255,  6,  6,255,255,  6,255,255,
         255,255,255,255,255,255,255,255,255,255, 10,255,255,255,255,255,
           6,  8,  9,  2,  2,  5,  8,  2,  4,  4,  5,  5,  4,  5,  2,  5,
           1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  5,  4,  8,  4,  8,  2,
         255,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
           0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  3,255,  3,  5,  2,
         255,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
           0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  7,  7,  7,255,255,
```

```
        255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
        255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
        255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
        255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
        255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
        255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
        255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
        255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255 }
};

ALLGLOBALS A;
#ifdef WITHPTHREADS
ALLPRIVATES **AB;
#endif

static struct fixedfun {
    char *name;
    int      commu ,tensor           ,complx          ,symmetric;
} fixedfunctions[] = {
     {"exp_"     ,1 ,0                ,0               ,0} /* EXPONENT */
    ,{"denom_"   ,1 ,0                ,0               ,0} /* DENOMINATOR */
    ,{"setfun_"  ,1 ,0                ,0               ,0} /* SETFUNCTION */
    ,{"g_"       ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMA */
    ,{"gi_"      ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMAI */
    ,{"g5_"      ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMAFIVE */
    ,{"g6_"      ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMASIX */
    ,{"g7_"      ,1 ,GAMMAFUNCTION    ,VARTYPEIMAGINARY,0} /* GAMMASEVEN */
    ,{"sum_"     ,1 ,0                ,0               ,0} /* SUMF1 */
    ,{"sump_"    ,1 ,0                ,0               ,0} /* SUMF2 */
    ,{"dum_"     ,1 ,0                ,0               ,0} /* DUMFUN */
    ,{"replace_" ,0 ,0                ,0               ,0} /* REPLACEMENT */
    ,{"reverse_" ,1 ,0                ,0               ,0} /* REVERSEFUNCTION */
    ,{"distrib_" ,1 ,0                ,0               ,0} /* DISTRIBUTION */
    ,{"dd_"      ,0 ,TENSORFUNCTION   ,0               ,0} /* DELTA3 */
    ,{"dummy_"   ,0 ,0                ,0               ,0} /* DUMMYFUN */
    ,{"dummyten_",0 ,TENSORFUNCTION   ,0               ,0} /* DUMMYTEN */
    ,{"e_"       ,0 ,TENSORFUNCTION|1 ,VARTYPEIMAGINARY,ANTISYMMETRIC}
                                                          /* LEVICIVITA */
    ,{"fac_"     ,0 ,0                ,0               ,0} /* FACTORIAL */
    ,{"invfac_"  ,0 ,0                ,0               ,0} /* INVERSEFACTORIAL */
    ,{"binom_"   ,0 ,0                ,0               ,0} /* BINOMIAL */
    ,{"nargs_"   ,0 ,0                ,0               ,0} /* NUMARGSFUN */
    ,{"sign_"    ,0 ,0                ,0               ,0} /* SIGNFUN */
    ,{"mod_"     ,0 ,0                ,0               ,0} /* MODFUNCTION */
    ,{"mod2_"    ,0 ,0                ,0               ,0} /* MOD2FUNCTION */
    ,{"min_"     ,0 ,0                ,0               ,0} /* MINFUNCTION */
    ,{"max_"     ,0 ,0                ,0               ,0} /* MAXFUNCTION */
    ,{"abs_"     ,0 ,0                ,0               ,0} /* ABSFUNCTION */
    ,{"sig_"     ,0 ,0                ,0               ,0} /* SIGFUNCTION */
    ,{"integer_" ,0 ,0                ,0               ,0} /* INTFUNCTION */
    ,{"theta_"   ,0 ,0                ,0               ,0} /* THETA */
    ,{"thetap_"  ,0 ,0                ,0               ,0} /* THETA2 */
    ,{"delta_"   ,0 ,0                ,0               ,0} /* DELTA2 */
    ,{"deltap_"  ,0 ,0                ,0               ,0} /* DELTAP */
    ,{"bernoulli_",0,0                ,0               ,0} /* BERNOULLIFUNCTION */
    ,{"count_"   ,0 ,0                ,0               ,0} /* COUNTFUNCTION */
    ,{"match_"   ,0 ,0                ,0               ,0} /* MATCHFUNCTION */
    ,{"pattern_" ,0 ,0                ,VARTYPECOMPLEX  ,0} /* PATTERNFUNCTION */
    ,{"term_"    ,1 ,0                ,0               ,0} /* TERMFUNCTION */
    ,{"conjg_"   ,1 ,0                ,VARTYPECOMPLEX  ,0} /* CONJUGATEFUNCTION */
    ,{"root_"    ,0 ,0                ,0               ,0} /* ROOTFUNCTION */
    ,{"table_"   ,1 ,0                ,0               ,0} /* TABLEFUNCTION */
    ,{"firstbracket_",0 ,0            ,0               ,0} /* FIRSTBRACKET */
    ,{"termsin_" ,0 ,0                ,0               ,0} /* TERMSINEXPR */
    ,{"nterms_"  ,0 ,0                ,0               ,0} /* NUMTERMSFUN */
    ,{"gcd_"     ,0 ,0                ,0               ,0} /* GCDFUNCTION */
    ,{"div_"        ,0 ,0             ,0               ,0} /* DIVFUNCTION */
    ,{"rem_"        ,0 ,0             ,0               ,0} /* REMFUNCTION */
    ,{"maxpowerof_",0 ,0              ,0               ,0} /* MAXPOWEROF */
    ,{"minpowerof_",0 ,0              ,0               ,0} /* MINPOWEROF */
    ,{"tbl_"     ,0 ,0                ,0               ,0} /* TABLESTUB */
    ,{"factorin_",0 ,0                ,0               ,0} /* FACTORIN */
    ,{"termsinbracket_",0 ,0          ,0               ,0} /* TERMSINBRACKET */
    ,{"farg_"    ,0 ,0                ,0               ,0} /* WILDARGFUN */
/*
     The following names are reserved for the floating point library.
     As long as we have no floating point numbers they do not do anything.
*/
    ,{"sqrt_"    ,0 ,0                ,0               ,0} /* SQRTFUNCTION */
    ,{"ln_"      ,0 ,0                ,0               ,0} /* LNFUNCTION */
    ,{"sin_"     ,0 ,0                ,0               ,0} /* SINFUNCTION */
    ,{"cos_"     ,0 ,0                ,0               ,0} /* COSFUNCTION */
    ,{"tan_"     ,0 ,0                ,0               ,0} /* TANFUNCTION */
    ,{"asin_"    ,0 ,0                ,0               ,0} /* ASINFUNCTION */
    ,{"acos_"    ,0 ,0                ,0               ,0} /* ACOSFUNCTION */
    ,{"atan_"    ,0 ,0                ,0               ,0} /* ATANFUNCTION */
    ,{"atan2_"   ,0 ,0                ,0               ,0} /* ATAN2FUNCTION */
```

```
    ,{"sinh_"    ,0 ,0                ,0                ,0} /* SINHFUNCTION */
    ,{"cosh_"    ,0 ,0                ,0                ,0} /* COSHFUNCTION */
    ,{"tanh_"    ,0 ,0                ,0                ,0} /* TANHFUNCTION */
    ,{"asinh_"   ,0 ,0                ,0                ,0} /* ASINHFUNCTION */
    ,{"acosh_"   ,0 ,0                ,0                ,0} /* ACOSHFUNCTION */
    ,{"atanh_"   ,0 ,0                ,0                ,0} /* ATANHFUNCTION */
    ,{"li2_"     ,0 ,0                ,0                ,0} /* LI2FUNCTION */
    ,{"lin_"     ,0 ,0                ,0                ,0} /* LINFUNCTION */
/*
     From here on we continue with new functions (26-sep-2010)
*/
    ,{"extrasymbol_",0 ,0             ,0                ,0} /* EXTRASYMFUN */
    ,{"random_"     ,0 ,0             ,0                ,0} /* RANDOMFUNCTION */
    ,{"ranperm_"    ,1 ,0             ,0                ,0} /* RANPERM */
    ,{"numfactors_" ,0 ,0             ,0                ,0} /* NUMFACTORS */
    ,{"firstterm_"  ,0 ,0             ,0                ,0} /* FIRSTTERM */
    ,{"content_"    ,0 ,0             ,0                ,0} /* CONTENTTERM */
    ,{"prime_"      ,0 ,0             ,0                ,0} /* PRIMENUMBER */
    ,{"exteuclidean_",0 ,0            ,0                ,0} /* EXTEUCLIDEAN */
    ,{"makerational_",0 ,0            ,0                ,0} /* MAKERATIONAL */
    ,{"inverse_"    ,0 ,0             ,0                ,0} /* INVERSEFUNCTION */
    ,{"id_"         ,1 ,0             ,0                ,0} /* IDFUNCTION */
    ,{"putfirst_"   ,1 ,0             ,0                ,0} /* PUTFIRST */
    ,{"perm_"       ,1 ,0             ,0                ,0} /* PERMUTATIONS */
    ,{"partitions_" ,1 ,0             ,0                ,0} /* PARTITIONS */
    ,{"mul_"        ,0 ,0             ,0                ,0} /* MULFUNCTION */
};

FIXEDSET fixedsets[] = {
     {"pos_",    "integers > 0",  CSYMBOL, 0}  /* POS_    0  */
    ,{"pos0_",   "integers >= 0", CSYMBOL, 0}  /* POS0_   1  */
    ,{"neg_",    "integers < 0",  CSYMBOL, 0}  /* NEG_    2  */
    ,{"neg0_",   "integers <= 0", CSYMBOL, 0}  /* NEG0_   3  */
    ,{"even_",   "even integers", CSYMBOL, 0}  /* EVEN_   4  */
    ,{"odd_",    "odd integers",  CSYMBOL, 0}  /* ODD_    5  */
    ,{"int_",    "all integers",  CSYMBOL, 0}  /* Z_      6  */
    ,{"symbol_","only symbols",  CSYMBOL, MAXPOSITIVE}  /* SYMBOL_ 7  */
    ,{"fixed_", "fixed indices", CINDEX, 0}   /* FIXED_  8  */
    ,{"index_", "all indices",   CINDEX, 0}   /* INDEX_  9  */
    ,{"number_","all rationals", CSYMBOL, 0}  /* Q_     10  */
    ,{"dummyindices_", "dummy indices", CINDEX, 0}   /* DUMMYINDEX_ 11  */
    ,{"vector_","only vectors",  CVECTOR, 0}  /* VECTOR_ 12  */
};

UBYTE BufferForOutput[MAXLINELENGTH+14];

char *setupfilename = "form.set";

#ifdef WITHPTHREADS
INILOCK(ErrorMessageLock);
INILOCK(FileReadLock);
#endif
```

4.49 lus.c File Reference

`#include "form3.h"`

Include dependency graph for lus.c:

Functions

- int Lus (WORD *term, WORD funnum, WORD loopsize, WORD numargs, WORD outfun, WORD mode)

- int FindLus (int from, int level, int openindex)
- int SortTheList (int *slist, int num)

4.49.1 Detailed Description

Routines to find loops in index contractions. These routines allow for a category of topological statements. They were originally developed for the color library.

Definition in file lus.c.

4.49.2 Function Documentation

4.49.2.1 Lus()

```
int Lus (
            WORD * term,
            WORD funnum,
            WORD loopsize,
            WORD numargs,
            WORD outfun,
            WORD mode )
```

Definition at line 57 of file lus.c.

4.49.2.2 FindLus()

```
int FindLus (
            int from,
            int level,
            int openindex )
```

Definition at line 515 of file lus.c.

4.49.2.3 SortTheList()

```
int SortTheList (
            int * slist,
            int num )
```

Definition at line 578 of file lus.c.

4.50 lus.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes : lus.c
*/

#include "form3.h"

/*
    #] Includes :
    #[ Lus :

    Routine to find loops.
    Mode: 0: Just tell whether there is such a loop.
          1: Take out the functions and replace by outfun with the
             remaining arguments of the loop function
          > AM.OffsetIndex: This index must be included in the loop.
          < -AM.OffsetIndex: This index must be included in the loop. Replace.
    Return value: 0: no loop. 1: there is/was such a loop.
    funnum: the function(s) in which we look for a loop.
    numargs: the number of arguments admissible in the function.
    outfun: the output function in case of a substitution.
    loopsize: the size of the loop we are looking for.
              if < 0 we look for all loops.
*/

int Lus(WORD *term, WORD funnum, WORD loopsize, WORD numargs, WORD outfun, WORD mode)
{
    GETIDENTITY
    WORD *w, *t, *tt, *m, *r, **loc, *tstop, minloopsize;
    int nfun, i, j, jj, k, n, sign = 0, action = 0, L, ten, ten2, totnum,
    sign2, *alist, *wi, mini, maxi, medi = 0;
    if ( numargs <= 1 ) return(0);
    GETSTOP(term,tstop);
/*
    First count the number of functions with the proper number of arguments.
*/
    t = term+1; nfun = 0;
    if ( ( ten = functions[funnum-FUNCTION].spec ) >= TENSORFUNCTION ) {
        while ( t < tstop ) {
            if ( *t == funnum && t[1] == FUNHEAD+numargs ) { nfun++; }
            t += t[1];
        }
    }
    else {
        while ( t < tstop ) {
            if ( *t == funnum ) {
                i = 0; m = t+FUNHEAD; t += t[1];
                while ( m < t ) { i++; NEXTARG(m) }
                if ( i == numargs ) nfun++;
            }
            else t += t[1];
        }
    }
    if ( loopsize < 0 ) minloopsize = 2;
    else                minloopsize = loopsize;
    if ( funnum < minloopsize ) return(0); /* quick abort */
```

```
    if ( ((functions[funnum-FUNCTION].symmetric) & ~REVERSEORDER) == ANTISYMMETRIC ) sign = 1;
    if ( mode == 1 || mode < 0 ) {
        ten2 = functions[outfun-FUNCTION].spec >= TENSORFUNCTION;
    }
    else ten2 = -1;
/*
    Allocations:
*/
    if ( AN.numflocs < funnum ) {
        if ( AN.funlocs ) M_free(AN.funlocs,"Lus: AN.funlocs");
        AN.numflocs = funnum;
        AN.funlocs = (WORD **)Malloc1(sizeof(WORD *)*AN.numflocs,"Lus: AN.funlocs");
    }
    if ( AN.numfargs < funnum*numargs ) {
        if ( AN.funargs ) M_free(AN.funargs,"Lus: AN.funargs");
        AN.numfargs = funnum*numargs;
        AN.funargs = (int *)Malloc1(sizeof(int *)*AN.numfargs,"Lus: AN.funargs");
    }
/*
    Make a list of relevant indices
*/
    alist = AN.funargs; loc = AN.funlocs;
    t = term+1;
    if ( ten >= TENSORFUNCTION ) {
        while ( t < tstop ) {
            if ( *t == funnum && t[1] == FUNHEAD+numargs ) {
                *loc++ = t;
                t += FUNHEAD;
                j = i = numargs; while ( --i >= 0 ) {
                    if ( *t >= AM.OffsetIndex &&
                        ( *t >= AM.OffsetIndex+WILDOFFSET ||
                        indices[*t-AM.OffsetIndex].dimension != 0 ) ) {
                        *alist++ = *t++; j--;
                    }
                    else t++;
                }
                while ( --j >= 0 ) *alist++ = -1;
            }
            else t += t[1];
        }
    }
    else {
        nfun = 0;
        while ( t < tstop ) {
            if ( *t == funnum ) {
                w = t;
                i = 0; m = t+FUNHEAD; t += t[1];
                while ( m < t ) { i++; NEXTARG(m) }
                if ( i == numargs ) {
                    m = w + FUNHEAD;
                    while ( m < t ) {
                        if ( *m == -INDEX && m[1] >= AM.OffsetIndex &&
                        ( m[1] >= AM.OffsetIndex+WILDOFFSET ||
                        indices[m[1]-AM.OffsetIndex].dimension != 0 ) ) {
                            *alist++ = m[1]; m += 2; i--;
                        }
                        else if ( ten2 >= TENSORFUNCTION && *m != -INDEX
                         && *m != -VECTOR && *m != -MINVECTOR &&
                        ( *m != -SNUMBER || *m < 0 || *m >= AM.OffsetIndex ) ) {
                            i = numargs; break;
                        }
                        else { NEXTARG(m) }
                    }
                    if ( i < numargs ) {
                        *loc++ = w;
                        nfun++;
                        while ( --i >= 0 ) *alist++ = -1;
                    }
                }
            }
            else t += t[1];
        }
        if ( nfun < minloopsize ) return(0);
    }
/*
    We have now nfun objects. Not all indices may be usable though.
    If the list is not long, we use a quadratic algorithm to remove
    indices and vertices that cannot be used. If it becomes large we
    sort the list of available indices (and their multiplicity) and
    work with binary searches.
*/
    alist = AN.funargs; totnum = numargs*nfun;
    if ( nfun > 7 ) {
        if ( AN.funisize < totnum ) {
            if ( AN.funinds ) M_free(AN.funinds,"AN.funinds");
            AN.funisize = (totnum*3)/2;
            AN.funinds = (int *)Malloc1(AN.funisize*2*sizeof(int),"AN.funinds");
```

```
        }
        i = totnum; n = 0; wi = AN.funinds;
        while ( --i >= 0 ) {
            if ( *alist >= 0 ) { n++; *wi++ = *alist; *wi++ = 1; }
            alist++;
        }
        n = SortTheList(AN.funinds,n);
        do {
            action = 0;
            for ( i = 0; i < nfun; i++ ) {
                alist = AN.funargs + i*numargs;
                jj = numargs;
                for ( j = 0; j < jj; j++ ) {
                    if ( alist[j] < 0 ) break;
                    mini = 0; maxi = n-1;
                    while ( mini <= maxi ) {
                        medi = (mini + maxi) / 2; k = AN.funinds[2*medi];
                        if ( alist[j] > k ) mini = medi + 1;
                        else if ( alist[j] < k ) maxi = medi - 1;
                        else break;
                    }
                    if ( AN.funinds[2*medi+1] <= 1 ) {
                        (AN.funinds[2*medi+1])--;
                        jj--; k = j; while ( k < jj ) { alist[k] = alist[k+1]; k++; }
                        alist[jj] = -1; j--;
                    }
                }
                if ( jj < 2 ) {
                    if ( jj == 1 ) {
                        mini = 0; maxi = n-1;
                        while ( mini <= maxi ) {
                            medi = (mini + maxi) / 2; k = AN.funinds[2*medi];
                            if ( alist[0] > k ) mini = medi + 1;
                            else if ( alist[0] < k ) maxi = medi - 1;
                            else break;
                        }
                        (AN.funinds[2*medi+1])--;
                        if ( AN.funinds[2*medi+1] == 1 ) action++;
                    }
                    nfun--; totnum -= numargs; AN.funlocs[i] = AN.funlocs[nfun];
                    wi = AN.funargs + nfun*numargs;
                    for ( j = 0; j < numargs; j++ ) alist[j] = *wi++;
                    i--;
                }
            }
        } while ( action );
    }
    else {
        for ( i = 0; i < totnum; i++ ) {
            if ( alist[i] == -1 ) continue;
            for ( j = 0; j < totnum; j++ ) {
                if ( alist[j] == alist[i] && j != i ) break;
            }
            if ( j >= totnum ) alist[i] = -1;
        }
        do {
            action = 0;
            for ( i = 0; i < nfun; i++ ) {
                alist = AN.funargs + i*numargs;
                n = numargs;
                for ( k = 0; k < n; k++ ) {
                    if ( alist[k] < 0 ) { alist[k--] = alist[--n]; alist[n] = -1; }
                }
                if ( n <= 1 ) {
                    if ( n == 1 ) { j = alist[0]; }
                    else j = -1;
                    nfun--; totnum -= numargs; AN.funlocs[i] = AN.funlocs[nfun];
                    wi = AN.funargs + nfun * numargs;
                    for ( k = 0; k < numargs; k++ ) alist[k] = wi[k];
                    i--;
                    if ( j >= 0 ) {
                        for ( k = 0, jj = 0, wi = AN.funargs; k < totnum; k++, wi++ ) {
                            if ( *wi == j ) { jj++; if ( jj > 1 ) break; }
                        }
                        if ( jj <= 1 ) {
                            for ( k = 0, wi = AN.funargs; k < totnum; k++, wi++ ) {
                                if ( *wi == j ) { *wi = -1; action = 1; }
                            }
                        }
                    }
                }
            }
        } while ( action );
    }
    if ( nfun < minloopsize ) return(0);
/*
    Now we have nfun objects, each with at least 2 indices, each of which
```

```
    occurs at least twice in our list. There will be a loop!
*/
    if ( mode != 0 && mode != 1 ) {
        if ( mode > 0 ) AN.tohunt =  mode - 5;
        else            AN.tohunt = -mode - 5;
        AN.nargs = numargs; AN.numoffuns = nfun;
        i = 0;
        if ( loopsize < 0 ) {
            if ( loopsize == -1 ) k = nfun;
            else { k = -loopsize-1; if ( k > nfun ) k = nfun; }
            for ( L = 2; L <= k; L++ ) {
                if ( FindLus(0,L,AN.tohunt) ) goto Success;
            }
        }
        else if ( FindLus(0,loopsize,AN.tohunt) ) { L = loopsize; goto Success; }
    }
    else {
        AN.nargs = numargs; AN.numoffuns = nfun;
        if ( loopsize < 0 ) {
            jj = 2; k = nfun;
            if ( loopsize < -1 ) { k = -loopsize-1; if ( k > nfun ) k = nfun; }
        }
        else { jj = k = loopsize; }
        for ( L = jj; L <= k; L++ ) {
            for ( i = 0; i <= nfun-L; i++ ) {
                alist = AN.funargs + i * numargs;
                for ( jj = 0; jj < numargs; jj++ ) {
                    if ( alist[jj] < 0 ) continue;
                    AN.tohunt = alist[jj];
                    for ( j = jj+1; j < numargs; j++ ) {
                        if ( alist[j] < 0 ) continue;
                        if ( FindLus(i+1,L-1,alist[j]) ) {
                            alist[0] = alist[jj];
                            alist[1] = alist[j];
                            goto Success;
                        }
                    }
                }
            }
        }
    }
    return(0);
Success:;
    if ( mode == 0 || mode > 1 ) return(1);
/*
    Now we have to make the replacement and fix the potential sign
*/
    sign2 = 1;
    wi = AN.funargs + i*numargs; loc = AN.funlocs + i;
    for ( k = 0; k < L; k++ ) *(loc[k]) = -1;
    if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
    w = AT.WorkPointer + 1;
    m = t = term + 1;
    while ( t < tstop ) {
        if ( *t == -1 ) break;
        t += t[1];
    }
    while ( m < t ) *w++ = *m++;
    r = w;
    *w++ = outfun;
    w++;
    *w++ = DIRTYFLAG;
    FILLFUN3(w)
    if ( functions[outfun-FUNCTION].spec >= TENSORFUNCTION ) {
        if ( ten >= TENSORFUNCTION ) {
            for ( i = 0; i < L; i++ ) {
                alist = wi + i*numargs;
                m = loc[i] + FUNHEAD;
                for ( k = 0; k < numargs; k++ ) {
                    if ( m[k] == alist[0] ) {
                        if ( k != 0 ) {
                            jj = m[k]; m[k] = m[0]; m[0] = jj;
                            sign = -sign;
                        }
                        break;
                    }
                }
                for ( k = 1; k < numargs; k++ ) {
                    if ( m[k] == alist[1] ) {
                        if ( k != 1 ) {
                            jj = m[k]; m[k] = m[1]; m[1] = jj;
                            sign = -sign;
                        }
                        break;
                    }
                }
                m += 2;
```

```
                for ( k = 2; k < numargs; k++ ) *w++ = *m++;
            }
        }
        else {
            WORD *t1, *t2, *t3;
            for ( i = 0; i < L; i++ ) {
                alist = wi + i*numargs;
                tt = loc[i];
                m = tt + FUNHEAD;
                for ( k = 0; k < numargs; k++ ) {
                    if ( *m == -INDEX && m[1] == alist[0] ) {
                        if ( k != 0 ) {
                            if ( ( k & 1 ) != 0 ) sign = -sign;
/*
                            now move to position 0
*/
                            t2 = m+2; t1 = m; t3 = tt+FUNHEAD;
                            while ( t1 > t3 ) { *--t2 = *--t1; }
                            t3[0] = -INDEX; t3[1] = alist[0];
                        }
                        break;
                    }
                    NEXTARG(m)
                }
                m = tt + FUNHEAD + 2;
                for ( k = 1; k < numargs; k++ ) {
                    if ( *m == -INDEX && m[1] == alist[1] ) {
                        if ( k != 1 ) {
                            if ( ( k & 1 ) == 0 ) sign = -sign;
/*
                            now move to position 1
*/
                            t2 = m+2; t1 = m; t3 = tt+FUNHEAD+2;
                            while ( t1 > t3 ) { *--t2 = *--t1; }
                            t3[0] = -INDEX; t3[1] = alist[1];
                        }
                        break;
                    }
                    NEXTARG(m)
                }
/*
                now copy the remaining arguments to w
                keep in mind that the output function is a tensor!
*/
                t1 = tt + FUNHEAD + 4;
                t2 = tt + tt[1];
                while ( t1 < t2 ) {
                    if ( *t1 == -INDEX || *t1 == -VECTOR ) {
                        *w++ = t1[1]; t1 += 2;
                    }
                    else if ( *t1 == -MINVECTOR ) {
                        *w++ = t1[1]; t1 += 2; sign2 = -sign2;
                    }
                    else if ( ( *t1 == -SNUMBER ) && ( t1[1] >= 0 ) && ( t1[1] < AM.OffsetIndex ) ) {
                        *w++ = t1[1]; t1 += 2; sign2 = -sign2;
                    }
                    else {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Illegal attempt to use a non-index-like argument in a tensor in
    ReplaceLoop statement");
                        MUNLOCK(ErrorMessageLock);
                        Terminate(-1);
                    }
                }
            }
        }
    }
    else {
        if ( ten >= TENSORFUNCTION ) {
            for ( i = 0; i < L; i++ ) {
                alist = wi + i*numargs;
                m = loc[i] + FUNHEAD;
                for ( k = 0; k < numargs; k++ ) {
                    if ( m[k] == alist[0] ) {
                        if ( k != 0 ) {
                            jj = m[k]; m[k] = m[0]; m[0] = jj;
                            sign = -sign;
                            break;
                        }
                    }
                }
                for ( k = 1; k < numargs; k++ ) {
                    if ( m[k] == alist[1] ) {
                        if ( k != 1 ) {
                            jj = m[k]; m[k] = m[1]; m[1] = jj;
                            sign = -sign;
                            break;
```

```
                        }
                    }
                }
                m += 2;
                for ( k = 2; k < numargs; k++ ) {
                    if ( *m >= AM.OffsetIndex ) { *w++ = -INDEX; }
                    else if ( *m < 0 ) { *w++ = -VECTOR; }
                    else { *w = -SNUMBER; }
                    *w++ = *m++;
                }
            }
        }
        else {
            WORD *t1, *t2, *t3;
            for ( i = 0; i < L; i++ ) {
                alist = wi + i*numargs;
                tt = loc[i];
                m = tt + FUNHEAD;
                for ( k = 0; k < numargs; k++ ) {
                    if ( *m == -INDEX && m[1] == alist[0] ) {
                        if ( k != 0 ) {
                            if ( ( k & 1 ) != 0 ) sign = -sign;
/*
                            now move to position 0
*/
                            t2 = m+2; t1 = m; t3 = tt+FUNHEAD;
                            while ( t1 > t3 ) { *--t2 = *--t1; }
                            t3[0] = -INDEX; t3[1] = alist[0];
                        }
                        break;
                    }
                    NEXTARG(m)
                }
                m = tt + FUNHEAD + 2;
                for ( k = 1; k < numargs; k++ ) {
                    if ( *m == -INDEX && m[1] == alist[1] ) {
                        if ( k != 1 ) {
                            if ( ( k & 1 ) == 0 ) sign = -sign;
/*
                            now move to position 1
*/
                            t2 = m+2; t1 = m; t3 = tt+FUNHEAD+2;
                            while ( t1 > t3 ) { *--t2 = *--t1; }
                            t3[0] = -INDEX; t3[1] = alist[1];
                        }
                        break;
                    }
                    NEXTARG(m)
                }
/*
                now copy the remaining arguments to w
*/
                t1 = tt + FUNHEAD + 4;
                t2 = tt + tt[1];
                while ( t1 < t2 ) *w++ = *t1++;
            }
        }
    }
    r[1] = w-r;
    while ( t < tstop ) {
        if ( *t == -1 ) { t += t[1]; continue; }
        i = t[1];
        NCOPY(w,t,i)
    }
    tstop = term + *term;
    while ( t < tstop ) *w++ = *t++;
    if ( sign < 0 ) w[-1] = -w[-1];
    i = w - AT.WorkPointer;
    *AT.WorkPointer = i;
    t = term; w = AT.WorkPointer;
    NCOPY(t,w,i)
    *AN.RepPoint = 1;   /* For Repeat */
    return(1);
}

/*
    #] Lus :
    #[ FindLus :
*/

int FindLus(int from, int level, int openindex)
{
    GETIDENTITY
    int i, j, k, jj, *alist, *blist, *w, *m, partner;
    WORD **loc = AN.funlocs, *wor;
    if ( level == 1 ) {
        for ( i = from; i < AN.numoffuns; i++ ) {
```

```
            alist = AN.funargs + i*AN.nargs;
            for ( j = 0; j < AN.nargs; j++ ) {
                if ( alist[j] == openindex ) {
                    for ( k = 0; k < AN.nargs; k++ ) {
                        if ( k == j ) continue;
                        if ( alist[k] == AN.tohunt ) {
                            loc[from] = loc[i];
                            alist = AN.funargs + from*AN.nargs;
                            alist[0] = openindex; alist[1] = AN.tohunt;
                            return(1);
                        }
                    }
                }
            }
        }
    }
    else {
        for ( i = from; i < AN.numoffuns; i++ ) {
            alist = AN.funargs + i*AN.nargs;
            for ( j = 0; j < AN.nargs; j++ ) {
                if ( alist[j] == openindex ) {
                    if ( from != i ) {
                        wor = loc[i]; loc[i] = loc[from]; loc[from] = wor;
                        blist = w = AN.funargs + from*AN.nargs;
                        m = alist;
                        k = AN.nargs;
                        while ( --k >= 0 ) { jj = *m; *m++ = *w; *w++ = jj; }
                    }
                    else blist = alist;
                    for ( k = 0; k < AN.nargs; k++ ) {
                        if ( k == j || blist[k] < 0 ) continue;
                        partner = blist[k];
                        if ( FindLus(from+1,level-1,partner) ) {
                            blist[0] = openindex; blist[1] = partner;
                            return(1);
                        }
                    }
                    if ( from != i ) {
                        wor = loc[i]; loc[i] = loc[from]; loc[from] = wor;
                        w = AN.funargs + from*AN.nargs;
                        m = alist;
                        k = AN.nargs;
                        while ( --k >= 0 ) { jj = *m; *m++ = *w; *w++ = jj; }
                    }
                }
            }
        }
    }
    return(0);
}

/*
    #] FindLus :
    #[ SortTheList :
*/

int SortTheList(int *slist, int num)
{
    GETIDENTITY
    int i, nleft, nright, *t1, *t2, *t3, *rlist;
    if ( num <= 2 ) {
        if ( num <=  1 ) return(num);
        if ( slist[0] < slist[2] ) return(2);
        if ( slist[0] > slist[2] ) {
            i = slist[0]; slist[0] = slist[2]; slist[2] = i;
            i = slist[1]; slist[1] = slist[3]; slist[3] = i;
            return(2);
        }
        slist[1] += slist[3];
        return(1);
    }
    else {
        nleft = num/2; rlist = slist + 2*nleft;
        nright = SortTheList(rlist,num-nleft);
        nleft = SortTheList(slist,nleft);
        if ( AN.tlistsize < nleft ) {
            if ( AN.tlistbuf ) M_free(AN.tlistbuf,"AN.tlistbuf");
            AN.tlistsize = (nleft*3)/2;
            AN.tlistbuf = (int *)Malloc1(AN.tlistsize*2*sizeof(int),"AN.tlistbuf");
        }
        i = nleft; t1 = slist; t2 = AN.tlistbuf;
        while ( --i >= 0 ) { *t2++ = *t1++; *t2++ = *t1++; }
        i = nleft+nright; t1 = AN.tlistbuf; t2 = rlist; t3 = slist;
        while ( nleft > 0 && nright > 0 ) {
            if ( *t1 < *t2 ) {
                *t3++ = *t1++; *t3++ = *t1++; nleft--;
            }
```

```
            else if ( *t1 > *t2 ) {
                *t3++ = *t2++; *t3++ = *t2++; nright--;
            }
            else {
                *t3++ = *t1++; t2++; *t3++ = (*t1++) + (*t2++); i--;
                nleft--; nright--;
            }
        }
        while ( --nleft >= 0 ) { *t3++ = *t1++; *t3++ = *t1++; }
        while ( --nright >= 0 ) { *t3++ = *t2++; *t3++ = *t2++; }
        return(i);
    }
}

/*
    #] SortTheList :
*/
```

4.51 mallocprotect.h

```
#ifndef MALLOCPROTECT_H
#define MALLOCPROTECT_H

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

/*
    #[ Documentation :

The enhanced malloc debugger. It intercepts access (both write AND
read) to the memory outside the allocated chunk in the following
manner: it prints out (to the stderr) the information about the event
and goes to the spilock. The user is able to attach the debugger to
the running process, investigate the problem and continue evaluation.

In case of error, the debugger will print to the stderr the following
information:
"***** PID: ###, Addr: ###, signal ### (code ###)"
"Attach gdb -p ### and set var loopForever=0 in a corresponding frame to continue"
or for TFORM:
"Attach gdb -p ### and set var loopForever=0 in a corresponding frame
   of a thread with LPW id ### to continue"

and go to the spilock. The user may attach gdb by the command
gdb -p ### and type "where" (in case of TFORM, the user should first
switch to the proper thread).  Then investigation of the corresponding
frames should clarify the situation. In order to continue the program,
the user might set (in the corresponding frame of the corresponding
thread) the variable loopForever to 0. The debugger will try to remove
local problem lead to the exception.

To activate the debugger, the macro MALLOCPROTECT should be
defined. There are three possible values:
#define MALLOCPROTECT -1 (any integer <0)
#define MALLOCPROTECT 0
#define MALLOCPROTECT 1 (any integer>0)

If MALLOCPROTECT < 0, the debugger will intercept any access to the
memory before the allocated chunk, even one byte.
```

```

If MALLOCPROTECT == 0, the debugger will intercept any access to the
memory before the allocated chunk, even one byte, and access to the
memory page next to the allocated one.

If MALLOCPROTECT > 0, the debugger will intercept any access to the
memory after the allocated chunk, even one byte, and access to the
memory page before the allocated one.

The original FORM malloc debugger is able to catch errors when the
allocated memory is freed improper, or if some small portion OUT of
the allocated chunk is written. This debugger is a complementary
one. It permits to catch situation when some small portion of memory
before or after the allocated chunk is written or even just read.

The idea is to protect the beginning or the end of the allocated chunk
for any kind of access and install the SIGSEGV handler in order to
intercept the access to this memory. Moreover, the allocated memory is
always immediately returned to the system so if the user tries to
access the memory after it is freed then the handler is triggered,
also.

The problem here is that we are able to allocate / protec only the
whole page. So the user has to run the debugger twice: one time with
the left alignment (#define MALLOCPROTECT <0), and then with the rigth
alignment (#define MALLOCPROTECT >0). During the first run the possible
errors like x[-1] will be catched, and the second run will manifest
reading over the allocated ares.

The leftmost extra page is always allocated and mprotected. The size
of the allocated chunk is stored in the beginning of this page.

    #] Documentation :
    #[ Includes :
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/mman.h>

#ifdef WITHPTHREADS
#ifdef LINUX
#include <sys/syscall.h>
#endif
#endif

/*
    #] Includes :
*/

static int pageSize=4096;/*the default value*/


/*
    #[ segv_handler :
*/

/*The handler will be invoked on some signal (usually SIGSEGV). It
will print some diagnostics to stderr and hands up in infinite loop
waiting the user for debugging:*/
static void segv_handler(int sig, siginfo_t *sip, void *xxx) {
    char *vadr;
    char *errStr;
    int actionBeforeExit=0;/* < 0 - unprotect, > 0 - map */
    switch(sip->si_signo){/* All POSIX signals for which the field siginfo_t.si_addr is defined:*/
        case SIGILL:
            switch(sip->si_code){
            case ILL_ILLOPC:
                errStr="SIGILL: Illegal opcode";
                break;
            case ILL_ILLOPN:
                errStr="SIGILL: Illegal operand";
                break;
            case ILL_ILLADR:
                errStr="SIGILL: Illegal addressing mode";
                break;
            case ILL_ILLTRP:
                errStr="SIGILL: Illegal trap";
                break;
            case ILL_PRVOPC:
                errStr="SIGILL: Privileged opcode";
                break;
            case ILL_PRVREG:
                errStr="SIGILL: Privileged register";
                break;
```

```
            case ILL_COPROC:
                errStr="SIGILL: Coprocessor error";
                break;
            case ILL_BADSTK:
                errStr="SIGILL: Internal stack error";
                break;
            default:
                errStr="SIGILL: Unknown signal code";
        }/*case SIGILL:*/
        break;
        case SIGFPE:
            switch(sip->si_code){
                case FPE_INTDIV:
                    errStr="SIGFPE: Integer divide-by-zero";
                    break;
                case FPE_INTOVF:
                    errStr="SIGFPE: Integer overflow";
                    break;
                case FPE_FLTDIV:
                    errStr="SIGFPE: Floating point divide-by-zero";
                    break;
                case FPE_FLTOVF:
                    errStr="SIGFPE: Floating point overflow";
                    break;
                case FPE_FLTUND:
                    errStr="SIGFPE: Floating point underflow";
                    break;
                case FPE_FLTRES:
                    errStr="SIGFPE: Floating point inexact result";
                    break;
                case FPE_FLTINV:
                    errStr="SIGFPE: Invalid floating point operation";
                    break;
                case FPE_FLTSUB:
                    errStr="SIGFPE: Subscript out of range";
                    break;
                default:
                    errStr="SIGFPE: Unknown signal code";
            }/*switch(sip->si_code)*/
            break;
        case SIGSEGV:
            switch(sip->si_code){
                case SEGV_MAPERR:
                    errStr="SIGSEGV: Address not mapped";
                    actionBeforeExit = 1;
                    break;
                case SEGV_ACCERR:
                    errStr="SIGSEGV: Invalid permissions";
                    actionBeforeExit = -1;
                    break;
                default:
                    errStr="SIGSEGV: Unknown signal code";
            }/*switch(sip->si_code)*/
            break;
        case SIGBUS:
            switch(sip->si_code){
                case BUS_ADRALN:
                    errStr="SIGBUS: Invalid address alignment";
                    break;
                case BUS_ADRERR:
                    errStr="SIGBUS: Non-existent physical address";
                    break;
                case BUS_OBJERR:
                    errStr="SIGBUS: Object-specific hardware error";
                    break;
                default:
                    errStr="SIGBUS: Unknown signal code";
            }/*switch(sip->si_code)*/
            break;
        default:
            errStr="Unknown signal";
    }/*switch(sip->si_signo)*/

    vadr = (caddr_t)sip->si_addr;
    fprintf(stderr, "\n***** PID: %ld, Addr: %p, signal %s (code %d)\n",
        (long)getpid(), vadr, errStr,sip->si_code);

    {/*Block*/
        /*The process hangs up at this block.
        Attach gdb to investigate and continue:
        */
        volatile int loopForever=1;
        size_t alignedAdr=((size_t)vadr) & (~(pageSize-1));
        fprintf(stderr, "   Attach gdb -p %ld and set var loopForever=0 in a corresponding frame",
            (long)getpid());
#ifdef CORRECTCODE
#ifdef WITHPTHREADS
```

```
#ifdef LINUX
        fprintf(stderr, "\n   of a thread with LPW id %ld",(long)syscall(SYS_gettid));
#else
        /*If the compiler fails here, just remove the next line:*/
        fprintf(stderr, "\n   of thread with LPW id %ld",(long)gettid());
#endif
#endif
#endif
        fprintf(stderr, " to continue\n");


        while(loopForever)
            sleep(1);

        if(actionBeforeExit<0)/*After changing loopForever=0, unprotect the page to continue:*/
            mprotect((char*)alignedAdr, pageSize, PROT_READ | PROT_WRITE);
        if(actionBeforeExit>0)/*After changing loopForever=0, map the page to continue:*/
/*          mmap((void*)alignedAdr,pageSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0); */
            mmap((void*)alignedAdr,pageSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
    }/*Block*/
}/*segv_handler*/

/*
    #] segv_handler :
*/

/*
    #[ mprotectInit :
*/

static FORM_INLINE int mprotectInit(void)
{
    struct sigaction sa;
    pageSize = getpagesize();
    sa.sa_sigaction = &segv_handler;

    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGIO);
    sigaddset(&sa.sa_mask, SIGALRM);

    sa.sa_flags = SA_SIGINFO;

    if (sigaction(SIGILL, &sa, NULL)) {
        fprintf(stderr,"Error on assigning %s.\n","SIGILL");
        return SIGILL;
    }
    if (sigaction(SIGFPE, &sa, NULL)) {
        fprintf(stderr,"Error on assigning %s.\n","SIGFPE");
        return SIGFPE;
    }
    if (sigaction(SIGSEGV, &sa, NULL)) {
        fprintf(stderr,"Error on assigning %s.\n","SIGSEGV");
        return SIGSEGV;
    }
    if (sigaction(SIGBUS, &sa, NULL)) {
        fprintf(stderr,"Error on assigning %s.\n","SIGBUS");
        return SIGBUS;
    }
    return 0;
}/*mprotectInit*/

/*
    #] mprotectInit :
*/


/*
    #[ mprotectMalloc :
*/

static void *mprotectMalloc(size_t theSize)
{
#if MALLOCPROTECT < 0
    /*Only one side is protected*/
    size_t nPages=1;
#else
    /*Both sides are protected*/
    size_t nPages=2;
#if MALLOCPROTECT > 0
    /*will need the original theSize*/
    size_t requestedSize=theSize;
#endif
#endif

    char *ret=NULL;
    size_t *ptr;

```

```


    /*Align required size to the pagesize:*/
    if(theSize % pageSize)
        nPages++;
    theSize= (theSize/pageSize+nPages)*pageSize;

/*  ret=(char*)mmap(0,theSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0); */
    ret=(char*)mmap(0,theSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
    if(ret == MAP_FAILED)
    return NULL;
    ptr=(size_t *)ret;
    *ptr=theSize;
    if(mprotect(ret, pageSize, PROT_NONE))return NULL;
#if MALLOCPROTECT < 0
    return ret+pageSize;
#else
    if(mprotect(ret+(theSize-pageSize), pageSize, PROT_NONE))return NULL;
#if MALLOCPROTECT ==0
    return ret+pageSize;
#endif
#endif
    /*MALLOCPROTECT > 0, but we need conditional compilation
    since the variable requestedSize:*/
#if MALLOCPROTECT > 0

    /*Potential problems with alignment if the requested size is not
    a multiple of items. But no poblems on x86-64:*/
    return ret+ (theSize-pageSize-requestedSize);
#endif
}/*mprotectMalloc*/
/*
    #] mprotectMalloc :
*/

/*
    #[ mprotectFree :
*/

static void mprotectFree(char *ptr)
{
    size_t theSize;
    if(ptr==NULL) return;
#if MALLOCPROTECT > 0
    /*The memory block was moved to the right, find the left page boundary:*/
    {/*Block*/
        size_t alignedPtr=((size_t)ptr) & (~(pageSize-1));
        ptr=(char*)alignedPtr;
    }/*Block*/
#endif

    ptr-=pageSize;
    mprotect(ptr, pageSize, PROT_READ);
    theSize=*((size_t*)ptr);
    munmap(ptr,theSize);
}/*mprotectFree*/

/*
    #] mprotectFree :
*/

#endif
```

4.52 message.c File Reference

`#include "form3.h"`
Include dependency graph for message.c:

Functions

- VOID Error0 (char ∗s)
- VOID Error1 (char ∗s, UBYTE ∗t)
- VOID Error2 (char ∗s1, char ∗s2, UBYTE ∗t)
- int MesWork ()
- int MesPrint (va_alist)
- VOID Warning (char ∗s)
- VOID HighWarning (char ∗s)
- int MesCall (char ∗s)
- WORD MesCerr (char ∗s, UBYTE ∗t)
- WORD MesComp (char ∗s, UBYTE ∗p, UBYTE ∗q)
- VOID PrintTerm (WORD ∗term, char ∗where)
- VOID PrintTermC (WORD ∗term, char ∗where)
- VOID PrintSubTerm (WORD ∗term, char ∗where)
- VOID PrintWords (WORD ∗buffer, LONG number)
- void PrintSeq (WORD ∗a, char ∗text)

4.52.1 Detailed Description

Contains the routines that write messages. This includes the very important routine MesPrint which is the FORM equivalent of printf but then with escape sequences that are relevant for symbolic manipulation. The FORM statement Print "...." is passed almost literally to MesPrint.

Definition in file message.c.

4.52.2 Function Documentation

4.52.2.1 Error0()

```
VOID Error0 (
            char * s )
```

Definition at line 56 of file message.c.

4.52.2.2 Error1()

```
VOID Error1 (
            char * s,
            UBYTE * t )
```

Definition at line 67 of file message.c.

4.52.2.3 Error2()

```
VOID Error2 (
            char * s1,
            char * s2,
            UBYTE * t )
```

Definition at line 78 of file message.c.

4.52.2.4 MesWork()

```
int MesWork ( )
```

Definition at line 89 of file message.c.

4.52.2.5 MesPrint()

```
int MesPrint (
            va_alist  )
```

Definition at line 140 of file message.c.

4.52.2.6 Warning()

```
VOID Warning (
            char * s )
```

Definition at line 790 of file message.c.

4.52.2.7 HighWarning()

```
VOID HighWarning (
            char * s )
```

Definition at line 802 of file message.c.

4.52.2.8 MesCall()

```
int MesCall (
            char * s )
```

Definition at line 814 of file message.c.

4.52.2.9 MesCerr()

```
WORD MesCerr (
            char * s,
            UBYTE * t )
```

Definition at line 824 of file message.c.

4.52.2.10 MesComp()

```
WORD MesComp (
            char * s,
            UBYTE * p,
            UBYTE * q )
```

Definition at line 843 of file message.c.

4.52.2.11 PrintTerm()

```
VOID PrintTerm (
            WORD * term,
            char * where )
```

Definition at line 857 of file message.c.

4.52.2.12 PrintTermC()

```
VOID PrintTermC (
            WORD * term,
            char * where )
```

Definition at line 887 of file message.c.

4.52.2.13 PrintSubTerm()

```
VOID PrintSubTerm (
            WORD * term,
            char * where )
```

Definition at line 921 of file message.c.

4.52.2.14 PrintWords()

```
VOID PrintWords (
            WORD * buffer,
            LONG number )
```

Definition at line 943 of file message.c.

4.52.2.15 PrintSeq()

```
void PrintSeq (
            WORD * a,
            char * text )
```

Definition at line 961 of file message.c.

4.53 message.c

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035 /*
00036     #[ Includes :
00037
00038     The static variables for the messages can remain as such also for
00039     the parallel version as messages are to be locked to avoid problems
00040     with simultaneous messages.
00041 */
00042
00043 #include "form3.h"
00044
00045 static int iswarning = 0;
00046
00047 static char hex[] = {'0','1','2','3','4','5','6','7','8','9',
00048                      'A','B','C','D','E','F'};
00049
00050 /*
00051     #] Includes :
00052     #[ exit :
00053         #[ Error0 :
00054 */
00055
00056 VOID Error0(char *s)
00057 {
00058     MesPrint("=== %s",s);
00059     Terminate(-1);
00060 }
00061
00062 /*
00063         #] Error0 :
00064         #[ Error1 :
00065 */
00066
00067 VOID Error1(char *s, UBYTE *t)
00068 {
00069     MesPrint("@%s %s",s,t);
00070     Terminate(-1);
00071 }
00072
00073 /*
```

```
        #] Error1 :
        #[ Error2 :
*/

VOID Error2(char *s1, char *s2, UBYTE *t)
{
    MesPrint("@%s%s %s",s1,s2,t);
    Terminate(-1);
}

/*
        #] Error2 :
        #[ MesWork :
*/

int MesWork()
{
    MesPrint("=== Workspace overflow. %l bytes is not enough.",AM.WorkSize);
    MesPrint("=== Change parameter WorkSpace in %s",setupfilename);
    Terminate(-1);
    return(-1);
}

/*
        #] MesWork :
        #[ MesPrint :

    Kind of a printf function for simple messages.
    The main concern is getting the arguments in a portable way.
    Note: many compilers have errors when sizeof(WORD) < sizeof(int)
    %a  array of size n WORDs (two parameters, first is int, second WORD *)
    %b  array of size n UBYTEs (two parameters, first is int, second UBYTE *)
    %C  array of size n chars (two parameters, first is int, second char *)
    %d  word;
    %l  long;
    %L  long long *;
    %s  string;
    %#i unsigned word filled
    %#d word positioned
    %#l long word positioned.
    %#L long long word * positioned.
    %#s string positioned.
    %#p position in file.
    %r  The current term in raw format (internal representation)
    %t  The current term (AN.currentTerm)
    %T  The current term (AN.currentTerm) with its sign
    %w  Number of the thread(worker)
    %$  The next $ in AN.listinprint
    %x  hexadecimal. Takes 8 places. Mainly for debugging.
    %%  %
    %#  #
    #   " ==> "
    @   " ==> "   Preprocessor error
    &   ' --> '   Regular compiler error
    Each call is terminated with a new line.
    Put a % at the end of the string to suppress the new line.

    New feature (7-dec-2011): The & will only work when we do not block it
    from the execution of the print statement because we need the & also for
    the tabulator in the print "" statement.
*/

int
#ifdef ANSI
MesPrint(const char *fmt, ... )
#else
MesPrint(va_alist)
va_dcl
#endif
{
    GETIDENTITY
    char Out[MAXLINELENGTH+14], *stopper, *t, *s, *u, c, *carray;
    UBYTE extrabuffer[MAXLINELENGTH+14];
    int w, x, i, specialerror = 0;
    LONG num, y;
    WORD *array;
    UBYTE *oldoutfill = AO.OutputLine, *barray;
    /*[19apr2004 mt]:*/
    LONG (*OldWrite)(int handle, UBYTE *buffer, LONG size) = WriteFile;
    /*:[19apr2004 mt]*/
    va_list ap;
#ifdef ANSI
    va_start(ap,fmt);
    s = (char *)fmt;
#else
    va_start(ap);
    s = va_arg(ap,char *);
```

```
#endif
#ifdef WITHMPI
    /*
     * On slaves, if AS.printflag is
     *   = 0 : print nothing.
     *   > 0 : synchronized output. All text will be sent to the master
     *         in the next MUNLOCK().
     *   < 0 : normal output.
     */
    if ( PF.me != MASTER && AS.printflag == 0 ) return(0);
    if ( PF.me == MASTER || AS.printflag < 0 )
#endif
    FLUSHCONSOLE;
    /*
     * MesPrints() never prints a message to an external channel even if
     * WriteFile is set to &WriteToExternalChannel.
     */
#ifdef WITHMPI
    WriteFile = PF.me == MASTER || AS.printflag > 0 ? &PF_WriteFileToFile : &WriteFileToFile;
#else
    WriteFile = &WriteFileToFile;
#endif
    AO.OutputLine = extrabuffer;
    t = Out;
    stopper = Out + AC.LineLength;
    while ( *s ) {
        if ( ( ( *s == '&' && AO.ErrorBlock == 0 ) || *s == '@' || *s == '#' ) && AC.CurrentStream != 0 ) {
            u = (char *)AC.CurrentStream->name;
            while ( *u ) {
                *t++ = *u++;
                if ( t >= stopper ) {
                    num = t - Out;
                    WriteString(ERROROUT,(UBYTE *)Out,num);
                    num = 0; t = Out;
                }
            }
            *t++ = ' ';
            if ( t+20 >= stopper ) {
                num = t - Out;
                WriteString(ERROROUT,(UBYTE *)Out,num);
                num = 0; t = Out;
            }
            *t++ = 'L'; *t++ = 'i'; *t++ = 'n'; *t++ = 'e'; *t++ = ' ';
            if ( *s == '&' ) y = AC.CurrentStream->prevline;
            else             y = AC.CurrentStream->linenumber;
            t = LongCopy(y,t);
            if ( !iswarning && ( *s == '&' || *s == '@' ) ) {
                for ( i = 0; i < NumDoLoops; i++ ) DoLoops[i].errorsinloop = 1;
            }
        }
        if ( ( *s == '&' && AO.ErrorBlock == 0 ) ) {
            *t++ = ' '; *t++ = '-'; *t++ = '-'; *t++ = '>'; *t++ = ' '; s++;
        }
        else if ( *s == '@' || *s == '#' ) {
            *t++ = ' '; *t++ = '='; *t++ = '='; *t++ = '>'; *t++ = ' '; s++;
        }
/*
        else if ( *s == '&' && AO.ErrorBlock == 1 ) {

        }
*/
        else if ( *s != '%' ) {
            *t++ = *s++;
            if ( t >= stopper ) {
                num = t - Out;
                WriteString(ERROROUT,(UBYTE *)Out,num);
                num = 0; t = Out;
            }
        }
        else {
            s++;
            if ( *s == 'd' ) {
                if ( ( w = va_arg(ap, int) ) < 0 ) { *t++ = '-'; w = -w; }
                t = (char *)NumCopy(w,(UBYTE *)t);
            }
            else if ( *s == 'l' ) {
                if ( ( y = va_arg(ap, LONG) ) < 0 ) { *t++ = '-'; y = -y; }
                t = LongCopy(y,t);
            }
/*  #ifdef __GLIBC_HAVE_LONG_LONG */
            else if ( *s == 'p' ) {
                POSITION *pp;
                off_t ly;
                pp = va_arg(ap, POSITION *);
                ly = BASEPOSITION(*pp);
                if ( ly < 0 ) { *t++ = '-'; ly = -ly; }
```

```
/*----change 10-feb-2003 did not have & */
                t = LongLongCopy(&(ly),t);
            }
/*  #endif  */
            else if ( *s == 'c' ) {
                c = (char)(va_arg(ap, int));
                *t++ = c; *t = 0;
            }
            else if ( *s == 'a' ) {
                w = va_arg(ap, int);
                array = va_arg(ap,WORD *);
                while ( w > 0 ) {
                    t = (char *)NumCopy(*array,(UBYTE *)t);
                    if ( t >= stopper ) {
                        num = t - Out;
                        WriteString(ERROROUT,(UBYTE *)Out,num);
                        t = Out;
                        *t++ = ' ';
                    }
                    *t++ = ' ';
                    w--; array++;
                }
            }
            else if ( *s == 'b' ) {
                w = va_arg(ap, int);
                barray = va_arg(ap,UBYTE *);
                while ( w > 0 ) {
                    *t++ = hex[((*barray)»4)&0xF];
                    *t++ = hex[(*barray)&0xF];
                    *t = 0;
                    if ( t >= stopper ) {
                        num = t - Out;
                        WriteString(ERROROUT,(UBYTE *)Out,num);
                        t = Out;
                        *t++ = ' ';
                    }
                    *t++ = ' ';
                    w--; barray++;
                }
            }
            else if ( *s == 'C' ) {
                w = va_arg(ap, int);
                carray = va_arg(ap,char *);
                while ( w > 0 ) {
                    if ( *carray < 32 ) *t++ = '^';
                    else *t++ = *carray;
                    *t = 0;
                    if ( t >= stopper ) {
                        num = t - Out;
                        WriteString(ERROROUT,(UBYTE *)Out,num);
                        t = Out;
                        *t++ = ' ';
                    }
                    w--; carray++;
                }
            }
            else if ( *s == 'I' ) {
                int *iarray;
                w = va_arg(ap, int);
                iarray = va_arg(ap,int *);
                while ( w > 0 ) {
                    t = (char *)LongCopy((LONG)(*iarray),(char *)t);
                    if ( t >= stopper ) {
                        num = t - Out;
                        WriteString(ERROROUT,(UBYTE *)Out,num);
                        t = Out;
                        *t++ = ' ';
                    }
                    *t++ = ' ';
                    w--; array++;
                }
            }
            else if ( *s == 'E' ) {
                LONG *larray;
                w = va_arg(ap, int);
                larray = va_arg(ap,LONG *);
                while ( w > 0 ) {
                    t = (char *)LongCopy(*larray,(char *)t);
                    if ( t >= stopper ) {
                        num = t - Out;
                        WriteString(ERROROUT,(UBYTE *)Out,num);
                        t = Out;
                        *t++ = ' ';
                    }
                    *t++ = ' ';
                    w--; array++;
                }
```

```
            }
            else if ( *s == 's' ) {
                u = va_arg(ap,char *);
                while ( *u ) {
                    if ( t >= stopper ) {
                        num = t - Out;
                        WriteString(ERROROUT,(UBYTE *)Out,num);
                        t = Out;
                    }
                    *t++ = *u++;
                }
                *t = 0;
            }
            else if ( *s == 't' || *s == 'T' ) {
                WORD oldskip = AO.OutSkip, noleadsign;
                WORD oldmode = AC.OutputMode;
                WORD oldbracket = AO.IsBracket;
                WORD oldlength = AC.LineLength;
                UBYTE *oldStop = AO.OutStop;
                if ( AN.currentTerm ) {
                    if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
                    AO.IsBracket = 0;
                    AO.OutSkip = 1;
                    AC.OutputMode = 0;
                    AO.OutFill = AO.OutputLine;
                    AO.OutStop = AO.OutputLine + AC.LineLength;
                    *t = 0;
                    AddToLine((UBYTE *)Out);
                    if ( *s == 'T' ) noleadsign = 1;
                    else noleadsign = 0;
                    if ( WriteInnerTerm(AN.currentTerm,noleadsign) ) Terminate(-1);
                    t = Out;
                    u = (char *)AO.OutputLine;
                    *(AO.OutFill) = 0;
                    while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
                    *t = 0;
                    AO.OutSkip = oldskip;
                    AC.OutputMode = oldmode;
                    AO.IsBracket = oldbracket;
                    AC.LineLength = oldlength;
                    AO.OutStop = oldStop;
                }
            }
            else if ( *s == 'r' ) {
                WORD oldskip = AO.OutSkip;
                WORD oldmode = AC.OutputMode;
                WORD oldbracket = AO.IsBracket;
                WORD oldlength = AC.LineLength;
                UBYTE *oldStop = AO.OutStop;
                if ( AN.currentTerm ) {
                    WORD *tt = AN.currentTerm;
                    if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
                    AO.IsBracket = 0;
                    AO.OutSkip = 1;
                    AC.OutputMode = 0;
                    AO.OutFill = AO.OutputLine;
                    AO.OutStop = AO.OutputLine + AC.LineLength;
                    *t = 0;
                    i = *tt;
                    while ( --i >= 0 ) {
                        t = (char *)NumCopy(*tt,(UBYTE *)t);
                        tt++;
                        if ( t >= stopper ) {
                            num = t - Out;
                            WriteString(ERROROUT,(UBYTE *)Out,num);
                            num = 0; t = Out;
                        }
                        *t++ = ' '; *t++ = ' ';
                    }
                    *t = 0;
                    AO.OutSkip = oldskip;
                    AC.OutputMode = oldmode;
                    AO.IsBracket = oldbracket;
                    AC.LineLength = oldlength;
                    AO.OutStop = oldStop;
                }
            }
            else if ( *s == '$' ) {
/*
            #[ dollars :
*/
                WORD oldskip = AO.OutSkip;
                WORD oldmode = AC.OutputMode;
                WORD oldbracket = AO.IsBracket;
                WORD oldlength = AC.LineLength;
                UBYTE *oldStop = AO.OutStop;
                WORD *term, indsubterm[3], *tt;
```

```
                WORD value[5], first, num;
                if ( *AN.listinprint != DOLLAREXPRESSION ) {
                    specialerror = 1;
                }
                else {
                    DOLLARS d = Dollars + AN.listinprint[1];
#ifdef WITHPTHREADS
                    int nummodopt, dtype;
                    dtype = -1;
                    if ( AS.MultiThreaded ) {
                        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                            if ( AN.listinprint[1] == ModOptdollars[nummodopt].number ) break;
                        }
                        if ( nummodopt < NumModOptdollars ) {
                            dtype = ModOptdollars[nummodopt].type;
                            if ( dtype == MODLOCAL ) {
                                d = ModOptdollars[nummodopt].dstruct+AT.identity;
                            }
                            else {
                                LOCK(d->pthreadslockread);
                            }
                        }
                    }
#endif
                    AO.IsBracket = 0;
                    AO.OutSkip = 0;
                    AC.OutputMode = 0;
                    AO.OutFill = AO.OutputLine;
                    AO.OutStop = AO.OutputLine + AC.LineLength;
                    *t = 0;
                    AddToLine((UBYTE *)Out);
                    if ( d->nfactors >= 1 && AN.listinprint[2] == DOLLAREXPR2 ) {
                        if ( d->type == 0 ||
                         ( d->factors == 0 && d->nfactors != 1 ) ) goto dollarzero;
                        num = EvalDoLoopArg(BHEAD AN.listinprint+2,-1);
                        if ( num == 0 ) {
                            value[0] = 4; value[1] = d->nfactors; value[2] = 1; value[3] = 3; value[4] = 0;
                            term = value; goto printterms;
                        }
                        if ( num == 1 && d->nfactors == 1 ) {
                            term = d->where;
                            if ( *term == 0 ) goto dollarzero;
                            goto printterms;
                        }
                        if ( num > d->nfactors ) {
                            MesPrint("\nFactor number for dollar is too large.");
                            Terminate(-1);
                        }
                        term = d->factors[num-1].where;
                        if ( term == 0 ) {
                            if ( d->factors[num-1].value < 0 ) {
                                value[0] = 4; value[1] = -d->factors[num-1].value; value[2] = 1; value[3] = -3; value[4] = 0;
                            }
                            else {
                                value[0] = 4; value[1] = d->factors[num-1].value; value[2] = 1; value[3] = 3; value[4] = 0;
                            }
                            term = value;
                        }
                        goto printterms;
                    }
                    if ( d->type == DOLTERMS || d->type == DOLNUMBER ) {
                        term = d->where;
printterms:             first = 1;
                        do {
                            if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
                            AO.IsBracket = 0;
                            AO.OutSkip = 1;
                            AC.OutputMode = 0;
                            AO.OutFill = AO.OutputLine;
                            AO.OutStop = AO.OutputLine + AC.LineLength;
                            *t = 0;
                            AddToLine((UBYTE *)Out);
                            if ( WriteInnerTerm(term,first) ) {
#ifdef WITHPTHREADS
                                if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                Terminate(-1);
                            }
                            first = 0;
                            t = Out;
                            u = (char *)AO.OutputLine;
                            *(AO.OutFill) = 0;
                            while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
                            *t = 0;
```

```
                            AO.OutSkip = oldskip;
                            AC.OutputMode = oldmode;
                            AO.IsBracket = oldbracket;
                            AC.LineLength = oldlength;
                            AO.OutStop = oldStop;
                            term += *term;
                        } while ( *term );
                        AO.OutSkip = oldskip;
                    }
                    else if ( d->type == DOLSUBTERM ) {
                        tt = d->where;
dosubterm:              if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
                        AO.IsBracket = 0;
                        AO.OutSkip = 1;
                        AC.OutputMode = 0;
                        AO.OutFill = AO.OutputLine;
                        AO.OutStop = AO.OutputLine + AC.LineLength;
                        *t = 0;
                        AddToLine((UBYTE *)Out);
                        if ( WriteSubTerm(tt,1) ) {
#ifdef WITHPTHREADS
                            if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                            Terminate(-1);
                        }
                        t = Out;
                        u = (char *)AO.OutputLine;
                        *(AO.OutFill) = 0;
                        while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
                        *t = 0;
                        AO.OutSkip = oldskip;
                        AC.OutputMode = oldmode;
                        AO.IsBracket = oldbracket;
                        AC.LineLength = oldlength;
                        AO.OutStop = oldStop;
                    }
                    else if ( d->type == DOLUNDEFINED ) {
                        *t++ = '*'; *t++ = '*'; *t++ = '*'; *t = 0;
                    }
                    else if ( d->type == DOLZERO ) {
dollarzero:             *t++ = '0'; *t = 0;
                    }
                    else if ( d->type == DOLINDEX ) {
                        tt = indsubterm; *tt = INDEX;
                        tt[1] = 3; tt[2] = d->index;
                        goto dosubterm;
                    }
                    else if ( d->type == DOLARGUMENT ) {
                        if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
                        AO.IsBracket = 0;
                        AO.OutSkip = 1;
                        AC.OutputMode = 0;
                        AO.OutFill = AO.OutputLine;
                        AO.OutStop = AO.OutputLine + AC.LineLength;
                        *t = 0;
                        AddToLine((UBYTE *)Out);
                        WriteArgument(d->where);
                        t = Out;
                        u = (char *)AO.OutputLine;
                        *(AO.OutFill) = 0;
                        while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
                        *t = 0;
                        AO.OutSkip = oldskip;
                        AC.OutputMode = oldmode;
                        AO.IsBracket = oldbracket;
                        AC.LineLength = oldlength;
                        AO.OutStop = oldStop;
                    }
                    else if ( d->type == DOLWILDARGS ) {
                        tt = d->where;
                        if ( *tt == 0 ) { tt++;
                         while ( *tt ) {
                          if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
                          AO.IsBracket = 0;
                          AO.OutSkip = 1;
                          AC.OutputMode = 0;
                          AO.OutFill = AO.OutputLine;
                          AO.OutStop = AO.OutputLine + AC.LineLength;
                          *t = 0;
                          AddToLine((UBYTE *)Out);
                          WriteArgument(tt);
                          NEXTARG(tt);
                          if ( *tt ) TokenToLine((UBYTE *)",");
                          t = Out;
                          u = (char *)AO.OutputLine;
                          *(AO.OutFill) = 0;
                          while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
```

```
                        *t = 0;
                        AO.OutSkip = oldskip;
                        AC.OutputMode = oldmode;
                        AO.IsBracket = oldbracket;
                        AC.LineLength = oldlength;
                        AO.OutStop = oldStop;
                       }
                      }
                      else if ( *tt > 0 ) {   /* Tensor arguments */
                          i = *tt++;
                          while ( --i >= 0 ) {
                              indsubterm[0] = INDEX;
                              indsubterm[1] = 3;
                              indsubterm[2] = *tt++;
                              if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
                              AO.IsBracket = 0;
                              AO.OutSkip = 1;
                              AC.OutputMode = 0;
                              AO.OutFill = AO.OutputLine;
                              AO.OutStop = AO.OutputLine + AC.LineLength;
                              *t = 0;
                              AddToLine((UBYTE *)Out);
                              if ( WriteSubTerm(indsubterm,1) ) Terminate(-1);
                              if ( i > 0 ) TokenToLine((UBYTE *)",");
                              t = Out;
                              u = (char *)AO.OutputLine;
                              *(AO.OutFill) = 0;
                              while ( u < (char *)(AO.OutFill) ) *t++ = *u++;
                              *t = 0;
                              AO.OutSkip = oldskip;
                              AC.OutputMode = oldmode;
                              AO.IsBracket = oldbracket;
                              AC.LineLength = oldlength;
                              AO.OutStop = oldStop;
                          }
                      }
                  }
#ifdef WITHPTHREADS
                  if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                  AN.listinprint += 2;
                  while ( AN.listinprint[0] == DOLLAREXPR2 ) AN.listinprint += 2;
              }
/*
          #] dollars :
*/
          }
#ifdef WITHPTHREADS
          else if ( *s == 'W' ) { /* number of the thread with time */
              LONG millitime;
              WORD timepart;
              t = (char *)NumCopy(identity,(UBYTE *)t);
              millitime = TimeCPU(1);
              timepart = (WORD)(millitime%1000);
              millitime /= 1000;
              timepart /= 10;
              *t++ = '('; *t = 0;
              t = (char *)LongCopy(millitime,(char *)t);
              *t++ = '.'; *t = 0;
              t = (char *)NumCopy(timepart,(UBYTE *)t);
              *t++ = ')'; *t = 0;
              if ( t >= stopper ) {
                  num = t - Out;
                  WriteString(ERROROUT,(UBYTE *)Out,num);
                  num = 0; t = Out;
              }
          }
          else if ( *s == 'w' ) { /* number of the thread */
              t = (char *)NumCopy(identity,(UBYTE *)t);
          }
#elif defined(WITHMPI)
          else if ( *s == 'W' ) { /* number of the thread with time */
              LONG millitime;
              WORD timepart;
              t = (char *)NumCopy(PF.me,(UBYTE *)t);
              millitime = TimeCPU(1);
              timepart = (WORD)(millitime%1000);
              millitime /= 1000;
              timepart /= 10;
              *t++ = '('; *t = 0;
              t = (char *)LongCopy(millitime,(char *)t);
              *t++ = '.'; *t = 0;
              t = (char *)NumCopy(timepart,(UBYTE *)t);
              *t++ = ')'; *t = 0;
              if ( t >= stopper ) {
                  num = t - Out;
                  WriteString(ERROROUT,(UBYTE *)Out,num);
```

```
                    num = 0; t = Out;
                }
            }
            else if ( *s == 'w' ) { /* number of the thread */
                t = (char *)NumCopy(PF.me,(UBYTE *)t);
            }
#else
            else if ( *s == 'w' ) { }
            else if ( *s == 'W' ) { }
#endif
            else if ( FG.cTable[(int)*s] == 1 ) {
                x = *s++ - '0';
                while ( FG.cTable[(int)*s] == 1 )
                    x = 10 * x + *s++ - '0';

                if ( *s == 'l' || *s == 'd' ) {
                    if ( *s == 'l' ) { y = va_arg(ap,LONG); }
                    else { y = va_arg(ap,int); }
                    if ( y < 0 ) { y = -y; w = 1; }
                    else w = 0;
                    u = t + x;
                    do { *--u = y%10+'0'; y /= 10; } while ( y && u > t );
                    if ( w && u > t ) *--u = '-';
                    while ( --u >= t ) *u = ' ';
                    t += x;
                }
                else if ( *s == 's' ) {
                    u = va_arg(ap,char *);
                    i = 0;
                    while ( *u ) { i++; u++; }
                    if ( i > x ) i = x;
                    while ( x > i ) { *t++ = ' '; x--; }
                    t += x;
                    while ( --i >= 0 ) { *--t = *--u; }
                    t += x;
                }
                else if ( *s == 'p' ) {
                    POSITION *pp;
/*#ifdef __GLIBC_HAVE_LONG_LONG */
                    off_t ly;
/*
#else
                    LONG ly;
#endif
*/
                    pp = va_arg(ap,POSITION *);
                    ly = BASEPOSITION(*pp);
                    u = t + x;
                    do { *--u = ly%10+'0'; ly /= 10; } while ( ly && u > t );
                    while ( --u >= t ) *u = ' ';
                    t += x;
                }
                else if ( *s == 'i' ) {
                    w = va_arg(ap, int);
                    u = t + x;
                    do { *--u = (char)(w%10+'0'); w /= 10; } while ( u > t );
                    t += x;
                }
                else {
                    w = va_arg(ap, int);
                    u = t + x;
                    do { *--u = (char )(w%10+'0'); w /= 10; } while ( w && u > t );
                    while ( --u >= t ) *u = ' ';
                    t += x;
                }
            }
            else if ( *s == 'x' ) {
                char ccc;
                y = va_arg(ap, LONG);
                i = 2*sizeof(LONG);
                while ( --i > 0 ) {
                    ccc = ( y » (i*4) ) & 0xF;
                    if ( ccc ) break;
                }
                do {
                    ccc = ( y » (i*4) ) & 0xF;
                    *t++ = hex[(int)ccc];
                } while ( --i >= 0 );
            }
            else if ( *s == '#' ) *t++ = *s;
            else if ( *s == '%' ) *t++ = *s;
            else if ( *s == 0 ) { *t++ = 0; break; }
            else if ( *s == '&' ) {
                *t++ = *s;
            }
            else {
                *t++ = '%';
```

```
                s--;
            }
            s++;
        }
    }
    num = t - Out;
    WriteString(ERROROUT,(UBYTE *)Out,num);
    va_end(ap);
    if ( specialerror == 1 ) {
        MesPrint("!!!Wrong object in Print statement!!!");
        MesPrint("!!!Object encountered is of a different type as in the format specifier");
    }
    AO.OutputLine = oldoutfill;
    /*[19apr2004 mt]:*/
    WriteFile=OldWrite;
    /*:[19apr2004 mt]*/
    return(-1);
}

/*
        #] MesPrint :
        #[ Warning :
*/

VOID Warning(char *s)
{
    iswarning = 1;
    if ( AC.WarnFlag ) MesPrint("&Warning: %s",s);
    iswarning = 0;
}

/*
        #] Warning :
        #[ HighWarning :
*/

VOID HighWarning(char *s)
{
    iswarning = 1;
    if ( AC.WarnFlag >= 2 ) MesPrint("&Warning: %s",s);
    iswarning = 0;
}

/*
        #] HighWarning :
        #[ MesCall :
*/

int MesCall(char *s)
{
    return(MesPrint((char *)"Called from %s",s));
}

/*
        #] MesCall :
        #[ MesCerr :
*/

WORD MesCerr(char *s, UBYTE *t)
{
    UBYTE *u, c;
    WORD i = 11;
    u = t;
    while ( *u && --i >= 0 ) u--;
    u++;
    c = *++t;
    *t = 0;
    MesPrint("&Illegal %s: %s",s,u);
    *t = c;
    return(-1);
}

/*
        #] MesCerr :
        #[ MesComp :
*/

WORD MesComp(char *s, UBYTE *p, UBYTE *q)
{
    UBYTE c;
    c = *++q; *q = 0;
    MesPrint("&%s: %s",s,p);
    *q = c;
    return(-1);
}

/*
```

```
        #] MesComp :
        #[ PrintTerm :
*/

VOID PrintTerm(WORD *term, char *where)
{
    UBYTE OutBuf[140];
    WORD *t, x;
    int i;
    AO.OutFill = AO.OutputLine = OutBuf;
    t = term;
    AO.OutSkip = 3;
    FiniLine();
    TokenToLine((UBYTE *)where);
    TokenToLine((UBYTE *)": ");
    i = *t;
    while ( --i >= 0 ) {
        x = *t++;
        if ( x < 0 ) {
            x = -x;
            TokenToLine((UBYTE *)"-");
        }
        TalToLine((UWORD)(x));
        TokenToLine((UBYTE *)"  ");
    }
    AO.OutSkip = 0;
    FiniLine();
}

/*
        #] PrintTerm :
        #[ PrintTermC :
*/

VOID PrintTermC(WORD *term, char *where)
{
    UBYTE OutBuf[140];
    WORD *t, x;
    int i;
    if ( *term >= 0 ) {
        PrintTerm(term,where);
        return;
    }
    AO.OutFill = AO.OutputLine = OutBuf;
    t = term;
    AO.OutSkip = 3;
    FiniLine();
    TokenToLine((UBYTE *)where);
    TokenToLine((UBYTE *)": ");
    i = t[1]+2;
    while ( --i >= 0 ) {
        x = *t++;
        if ( x < 0 ) {
            x = -x;
            TokenToLine((UBYTE *)"-");
        }
        TalToLine((UWORD)(x));
        TokenToLine((UBYTE *)"  ");
    }
    AO.OutSkip = 0;
    FiniLine();
}

/*
        #] PrintTermC :
        #[ PrintSubTerm :
*/

VOID PrintSubTerm(WORD *term, char *where)
{
    UBYTE OutBuf[140];
    WORD *t;
    int i;
    AO.OutFill = AO.OutputLine = OutBuf;
    t = term;
    AO.OutSkip = 3;
    FiniLine();
    TokenToLine((UBYTE *)where);
    TokenToLine((UBYTE *)": ");
    i = t[1];
    while ( --i >= 0 ) { TalToLine((UWORD)(*t++)); TokenToLine((UBYTE *)"  "); }
    AO.OutSkip = 0;
    FiniLine();
}

/*
        #] PrintSubTerm :
```

```
        #[ PrintWords :
*/

VOID PrintWords(WORD *buffer, LONG number)
{
    UBYTE OutBuf[140];
    WORD *t;
    AO.OutFill = AO.OutputLine = OutBuf;
    t = buffer;
    AO.OutSkip = 3;
    FiniLine();
    while ( --number >= 0 ) { TalToLine((UWORD)(*t++)); TokenToLine((UBYTE *)"  "); }
    AO.OutSkip = 0;
    FiniLine();
}

/*
        #] PrintWords :
        #[ PrintSeq :
*/

void PrintSeq(WORD *a,char *text)
{
    MesPrint(" %s:",text);
    while ( *a ) {
        MesPrint("     %a",a[0],a);
        a += *a;
    }
}

/*
        #] PrintSeq :
    #] exit :
*/
```

4.54 minos.c File Reference

```
#include "form3.h"
#include "minos.h"
```

Include dependency graph for minos.c:

Macros

- #define CFD(y, s, type, x, j)
- #define CTD(y, s, type, x, j)

Functions

- int minosread (FILE *f, char *buffer, MLONG size)
- int minoswrite (FILE *f, char *buffer, MLONG size)
- char * str_dup (char *str)

- void convertblock (INDEXBLOCK *in, INDEXBLOCK *out, int mode)
- void convertnamesblock (NAMESBLOCK *in, NAMESBLOCK *out, int mode)
- void convertiniinfo (INIIINFO *in, INIIINFO *out, int mode)
- FILE * LocateBase (char **name, char **newname)
- int ReadIndex (DBASE *d)
- int WriteIndexBlock (DBASE *d, MLONG num)
- int WriteNamesBlock (DBASE *d, MLONG num)
- int WriteIndex (DBASE *d)
- int WriteIniInfo (DBASE *d)
- int ReadIniInfo (DBASE *d)
- DBASE * GetDbase (char *filename)
- DBASE * NewDbase (char *name, MLONG number)
- void FreeTableBase (DBASE *d)
- int ComposeTableNames (DBASE *d)
- DBASE * OpenDbase (char *filename)
- MLONG AddTableName (DBASE *d, char *name, TABLES T)
- MLONG GetTableName (DBASE *d, char *name)
- int PutTableNames (DBASE *d)
- int AddToIndex (DBASE *d, MLONG number)
- MLONG AddObject (DBASE *d, MLONG tablenumber, char *arguments, char *rhs)
- MLONG FindTableNumber (DBASE *d, char *name)
- int WriteObject (DBASE *d, MLONG tablenumber, char *arguments, char *rhs, MLONG number)
- char * ReadObject (DBASE *d, MLONG tablenumber, char *arguments)
- char * ReadijObject (DBASE *d, MLONG i, MLONG j, char *arguments)
- int ExistsObject (DBASE *d, MLONG tablenumber, char *arguments)
- int DeleteObject (DBASE *d, MLONG tablenumber, char *arguments)

Variables

- int withoutflush = 0

4.54.1 Detailed Description

These are the low level functions for the database part of the tablebases. These routines have been copied (and then adapted) from the minos database program. This file goes together with minos.h

Definition in file minos.c.

4.54.2 Macro Definition Documentation

4.54.2.1 CFD

```
#define CFD(
            y,
            s,
            type,
            x,
            j )
```

Value:

```
for(x=0,j=0;j<((int)sizeof(type));j++) \
x=(x«8)+((*s++)&0x00FF); y=x;
```

Definition at line 55 of file minos.c.

4.54.2.2 CTD

```
#define CTD(
            y,
            s,
            type,
            x,
            j )
```

Value:

```
x=y;for(j=sizeof(type)-1;j>=0;j--){s[j]=x&0xFF; \
x»=8;} s += sizeof(type);
```

Definition at line 57 of file minos.c.

4.54.3 Function Documentation

4.54.3.1 minosread()

```
int minosread (
            FILE * f,
            char * buffer,
            MLONG size )
```

Definition at line 66 of file minos.c.

4.54.3.2 minoswrite()

```
int minoswrite (
            FILE * f,
            char * buffer,
            MLONG size )
```

Definition at line 83 of file minos.c.

4.54.3.3 str_dup()

```
char * str_dup (
            char * str )
```

Definition at line 101 of file minos.c.

4.54.3.4 convertblock()

```
void convertblock (
            INDEXBLOCK * in,
            INDEXBLOCK * out,
            int mode )
```

Definition at line 120 of file minos.c.

4.54.3.5 convertnamesblock()

```
void convertnamesblock (
            NAMESBLOCK * in,
            NAMESBLOCK * out,
            int mode )
```

Definition at line 171 of file minos.c.

4.54.3.6 convertiniinfo()

```
void convertiniinfo (
            INIINFO * in,
            INIINFO * out,
            int mode )
```

Definition at line 197 of file minos.c.

4.54.3.7 LocateBase()

```
FILE * LocateBase (
            char ** name,
            char ** newname )
```

Definition at line 225 of file minos.c.

4.54.3.8 ReadIndex()

```
int ReadIndex (
            DBASE * d )
```

Definition at line 287 of file minos.c.

4.54.3.9 WriteIndexBlock()

```
int WriteIndexBlock (
            DBASE * d,
            MLONG num )
```

Definition at line 400 of file minos.c.

4.54.3.10 WriteNamesBlock()

```
int WriteNamesBlock (
            DBASE * d,
            MLONG num )
```

Definition at line 421 of file minos.c.

4.54.3.11 WriteIndex()

```
int WriteIndex (
            DBASE * d )
```

Definition at line 444 of file minos.c.

4.54.3.12 WriteIniInfo()

```
int WriteIniInfo (
            DBASE * d )
```

Definition at line 507 of file minos.c.

4.54.3.13 ReadIniInfo()

```
int ReadIniInfo (
            DBASE * d )
```

Definition at line 524 of file minos.c.

4.54.3.14 GetDbase()

```
DBASE * GetDbase (
            char * filename )
```

Definition at line 547 of file minos.c.

4.54.3.15 NewDbase()

```
DBASE * NewDbase (
            char * name,
            MLONG number )
```

Definition at line 591 of file minos.c.

4.54.3.16 FreeTableBase()

```
void FreeTableBase (
            DBASE * d )
```

Definition at line 725 of file minos.c.

4.54.3.17 ComposeTableNames()

```
int ComposeTableNames (
            DBASE * d )
```

Definition at line 757 of file minos.c.

4.54.3.18 OpenDbase()

```
DBASE * OpenDbase (
            char * filename )
```

Definition at line 806 of file minos.c.

4.54.3.19 AddTableName()

```
MLONG AddTableName (
            DBASE * d,
            char * name,
            TABLES T )
```

Definition at line 840 of file minos.c.

4.54.3.20 GetTableName()

```
MLONG GetTableName (
            DBASE * d,
            char * name )
```

Definition at line 923 of file minos.c.

4.54.3.21 PutTableNames()

```
int PutTableNames (
            DBASE * d )
```

Definition at line 955 of file minos.c.

4.54.3.22 AddToIndex()

```
int AddToIndex (
            DBASE * d,
            MLONG number )
```

Definition at line 1049 of file minos.c.

4.54.3.23 AddObject()

```
MLONG AddObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments,
            char * rhs )
```

Definition at line 1136 of file minos.c.

4.54.3.24 FindTableNumber()

```
MLONG FindTableNumber (
            DBASE * d,
            char * name )
```

Definition at line 1150 of file minos.c.

4.54.3.25 WriteObject()

```
int WriteObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments,
            char * rhs,
            MLONG number )
```

Definition at line 1178 of file minos.c.

4.54.3.26 ReadObject()

```
char * ReadObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments )
```

Definition at line 1269 of file minos.c.

4.54.3.27 ReadijObject()

```
char * ReadijObject (
            DBASE * d,
            MLONG i,
            MLONG j,
            char * arguments )
```

Definition at line 1348 of file minos.c.

4.54.3.28 ExistsObject()

```
int ExistsObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments )
```

Definition at line 1400 of file minos.c.

4.54.3.29 DeleteObject()

```
int DeleteObject (
            DBASE * d,
            MLONG tablenumber,
            char * arguments )
```

Definition at line 1432 of file minos.c.

4.54.4 Variable Documentation

4.54.4.1 withoutflush

```
int withoutflush = 0
```

Definition at line 45 of file minos.c.

4.55 minos.c

Go to the documentation of this file.

```
00001
00007 /* #[ License : */
00008 /*
00009  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00010  *   When using this file you are requested to refer to the publication
00011  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  *   This is considered a matter of courtesy as the development was paid
00013  *   for by FOM the Dutch physics granting agency and we would like to
00014  *   be able to track its scientific use to convince FOM of its value
00015  *   for the community.
00016  *
00017  *   This file is part of FORM.
00018  *
00019  *   FORM is free software: you can redistribute it and/or modify it under the
00020  *   terms of the GNU General Public License as published by the Free Software
00021  *   Foundation, either version 3 of the License, or (at your option) any later
00022  *   version.
00023  *
00024  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00027  *   details.
00028  *
00029  *   You should have received a copy of the GNU General Public License along
00030  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #] License : */
00033 /*
00034     #[ Includes :
00035
00036     File contains the lowlevel routines for the management of primitive
00037     database-like files. The structures are explained in the file manage.h
00038     Original file for minos made by J.Vermaseren, april-1994.
00039
00040     Note: the minos primitives for writing are never invoked in parallel.
00041 */
00042
00043 #include "form3.h"
00044 #include "minos.h"
00045 int withoutflush = 0;
00046
00047 /*
00048     #] Includes :
00049     #[ Variables :
00050 */
00051
00052 static INDEXBLOCK scratchblock;
00053 static NAMESBLOCK scratchnamesblock;
00054
```

```c
#define CFD(y,s,type,x,j) for(x=0,j=0;j<((int)sizeof(type));j++) \
    x=(x<<8)+((*s++)&0x00FF); y=x;
#define CTD(y,s,type,x,j) x=y;for(j=sizeof(type)-1;j>=0;j--){s[j]=x&0xFF; \
                        x>>=8;} s += sizeof(type);

/*
    #] Variables :
    #[ Utilities :
        #[ minosread :
*/

int minosread(FILE *f,char *buffer,MLONG size)
{
    MLONG x;
    while ( size > 0 ) {
        x = fread(buffer,sizeof(char),size,f);
        if ( x <= 0 ) return(-1);
        buffer += x;
        size -= x;
    }
    return(0);
}

/*
        #] minosread :
        #[ minoswrite :
*/

int minoswrite(FILE *f,char *buffer,MLONG size)
{
    MLONG x;
    while ( size > 0 ) {
        x = fwrite(buffer,sizeof(char),size,f);
        if ( x <= 0 ) return(-1);
        buffer += x;
        size -= x;
    }
    if ( withoutflush == 0 ) fflush(f);
    return(0);
}

/*
        #] minoswrite :
        #[ str_dup :
*/

char *str_dup(char *str)
{
    char *s, *t;
    int i;
    s = str;
    while ( *s ) s++;
    i = s - str + 1;
    if ( ( s = (char *)Malloc1((size_t)i,"a string copy") ) == 0 ) return(0);
    t = s;
    while ( *str ) *t++ = *str++;
    *t = 0;
    return(s);
}

/*
        #] str_dup :
        #[ convertblock :
*/

void convertblock(INDEXBLOCK *in,INDEXBLOCK *out,int mode)
{
    char *s,*t;
    MLONG i, x;
    int j;
    OBJECTS *obj;
    switch ( mode ) {
    case TODISK:
        s = (char *)out;
        CTD(in->flags,s,MLONG,x,j)
        CTD(in->previousblock,s,MLONG,x,j)
        CTD(in->position,s,MLONG,x,j)
        for ( i = 0, obj = in->objects; i < NUMOBJECTS; i++, obj++ ) {
            CTD(obj->position,s,MLONG,x,j)
            CTD(obj->size,s,MLONG,x,j)
            CTD(obj->date,s,MLONG,x,j)
            CTD(obj->tablenumber,s,MLONG,x,j)
            CTD(obj->uncompressed,s,MLONG,x,j)
            CTD(obj->spare1,s,MLONG,x,j)
            CTD(obj->spare2,s,MLONG,x,j)
            CTD(obj->spare3,s,MLONG,x,j)
            t = obj->element;
```

```
            for ( j = 0; j < ELEMENTSIZE; j++ ) *s++ = *t++;
        }
        break;
    case FROMDISK:
        s = (char *)in;
        CFD(out->flags,s,MLONG,x,j)
        CFD(out->previousblock,s,MLONG,x,j)
        CFD(out->position,s,MLONG,x,j)
        for ( i = 0, obj = out->objects; i < NUMOBJECTS; i++, obj++ ) {
            CFD(obj->position,s,MLONG,x,j)
            CFD(obj->size,s,MLONG,x,j)
            CFD(obj->date,s,MLONG,x,j)
            CFD(obj->tablenumber,s,MLONG,x,j)
            CFD(obj->uncompressed,s,MLONG,x,j)
            CFD(obj->spare1,s,MLONG,x,j)
            CFD(obj->spare2,s,MLONG,x,j)
            CFD(obj->spare3,s,MLONG,x,j)
            t = obj->element;
            for ( j = 0; j < ELEMENTSIZE; j++ ) *t++ = *s++;
        }
        break;
    }
}

/*
        #] convertblock :
        #[ convertnamesblock :
*/

void convertnamesblock(NAMESBLOCK *in,NAMESBLOCK *out,int mode)
{
    char *s;
    MLONG x;
    int j;
    switch ( mode ) {
    case TODISK:
        s = (char *)out;
        CTD(in->previousblock,s,MLONG,x,j)
        CTD(in->position,s,MLONG,x,j)
        for ( j = 0; j < NAMETABLESIZE; j++ ) out->names[j] = in->names[j];
        break;
    case FROMDISK:
        s = (char *)in;
        CFD(out->previousblock,s,MLONG,x,j)
        CFD(out->position,s,MLONG,x,j)
        for ( j = 0; j < NAMETABLESIZE; j++ ) out->names[j] = in->names[j];
        break;
    }
}

/*
        #] convertnamesblock :
        #[ convertiniinfo :
*/

void convertiniinfo(INIINFO *in,INIINFO *out,int mode)
{
    char *s;
    MLONG i, x, *y;
    int j;
    switch ( mode ) {
    case TODISK:
        s = (char *)out; y = (MLONG *)in;
        for ( i = sizeof(INIINFO)/sizeof(MLONG); i > 0; i-- ) {
            CTD(*y,s,MLONG,x,j)
            y++;
        }
        break;
    case FROMDISK:
        s = (char *)in; y = (MLONG *)out;
        for ( i = sizeof(INIINFO)/sizeof(MLONG); i > 0; i-- ) {
            CFD(*y,s,MLONG,x,j)
            y++;
        }
        break;
    }
}

/*
        #] convertiniinfo :
        #[ LocateBase :
*/

FILE *LocateBase(char **name, char **newname)
{
    FILE *handle;
    int  namesize, i;
```

```
    UBYTE *s, *to, *u1, *u2, *indir;
    if ( ( handle = fopen(*name,"r+b") ) != 0 ) {
        *newname = (char *)strDup1((UBYTE *)(*name),"LocateBase");
        return(handle);
    }
    namesize = 2; s = (UBYTE *)(*name);
    while ( *s ) { s++; namesize++; }
    indir = AM.IncDir;
    if ( indir ) {
        s = indir; i = 0;
        while ( *s ) { s++; i++; }
        *newname = (char *)Malloc1(namesize+i,"LocateBase");
        s = indir; to = (UBYTE *)(*newname);
        while ( *s ) *to++ = *s++;
        if ( to > (UBYTE *)(*newname) && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
        s = (UBYTE *)(*name);
        while ( *s ) *to++ = *s++;
        *to = 0;
        if ( ( handle = fopen(*newname,"r+b") ) != 0 ) {
            return(handle);
        }
        M_free(*newname,"LocateBase, incdir/file");
    }
    if ( AM.Path ) {
        u1 = AM.Path;
        while ( *u1 ) {
            u2 = u1; i = 0;
            while ( *u1 && *u1 != ':' ) {
                if ( *u1 == '\\' ) u1++;
                u1++; i++;
            }
            *newname = (char *)Malloc1(namesize+i,"LocateBase");
            s = u2; to = (UBYTE *)(*newname);
            while ( s < u1 ) {
                if ( *s == '\\' ) s++;
                *to++ = *s++;
            }
            if ( to > (UBYTE *)(*newname) && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
            s = (UBYTE *)(*name);
            while ( *s ) *to++ = *s++;
            *to = 0;
            if ( ( handle = fopen(*newname,"r+b") ) != 0 ) {
                return(handle);
            }
            M_free(*newname,"LocateBase Path/file");
            if ( *u1 ) u1++;
        }
    }
/*  Error1("LocateBase: Cannot find file",*name); */
    return(0);
}

/*
        #] LocateBase :
    #] Utilities :
    #[ ReadIndex :
*/

int ReadIndex(DBASE *d)
{
    MLONG i;
    INDEXBLOCK **ib;
    NAMESBLOCK **ina;
    MLONG position, size;
/*
    Allocate the pieces one by one (makes it easier to give it back)
*/
    if ( d->info.numberofindexblocks <= 0 ) return(0);
#ifndef WORDSIZE32
    if ( sizeof(INDEXBLOCK)*d->info.numberofindexblocks > MAXINDEXSIZE ) {
        MesPrint("We need more than %ld bytes for the index.\n",MAXINDEXSIZE);
        MesPrint("The file %s may not be a proper database\n",d->name);
        return(-1);
    }
#endif
    size = sizeof(INDEXBLOCK *)*d->info.numberofindexblocks;
    if ( ( ib = (INDEXBLOCK **)Malloc1(size,"tb,index") ) == 0 ) return(-1);
    for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
        if ( ( ib[i] = (INDEXBLOCK *)Malloc1(sizeof(INDEXBLOCK),"index block") ) == 0 ) {
            for ( --i; i >= 0; i-- ) M_free(ib[i],"tb,indexblock");
            M_free(ib,"tb,index");
            return(-1);
        }
    }
    size = sizeof(NAMESBLOCK *)*d->info.numberofnamesblocks;
    if ( ( ina = (NAMESBLOCK **)Malloc1(size,"tb,indexnames") ) == 0 ) return(-1);
    for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
```

```
        if ( ( ina[i] = (NAMESBLOCK *)Malloc1(sizeof(NAMESBLOCK),"index names block") ) == 0 ) {
            for ( --i; i >= 0; i-- ) M_free(ina[i],"index names block");
            M_free(ina,"tb,indexnames");
            for ( i = 0; i < d->info.numberofindexblocks; i++ ) M_free(ib[i],"tb,indexblock");
            M_free(ib,"tb,index");
            return(-1);
        }
    }
/*
    Read the index blocks, from the back to the front. The links are only
    reliable that way.
*/
    position = d->info.lastindexblock;
    for ( i = d->info.numberofindexblocks - 1; i >= 0; i-- ) {
        fseek(d->handle,position,SEEK_SET);
        if ( minosread(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
            MesPrint("Error while reading file %s\n",d->name);
thisiswrong:
            for ( i = 0; i < d->info.numberofnamesblocks; i++ ) M_free(ina[i],"index names block");
            M_free(ina,"tb,indexnames");
            for ( i = 0; i < d->info.numberofindexblocks; i++ ) M_free(ib[i],"tb,indexblock");
            M_free(ib,"tb,index");
            return(-1);
        }
        convertblock(&scratchblock,ib[i],FROMDISK);
        if ( ib[i]->position != position ||
        ( ib[i]->previousblock <= 0 && i > 0 ) ) {
            MesPrint("File %s has inconsistent contents\n",d->name);
            goto thisiswrong;
        }
        position = ib[i]->previousblock;
    }
    d->info.firstindexblock = ib[0]->position;
    for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
        ib[i]->flags &= MCLEANFLAG;
    }
/*
    Read the names blocks, from the back to the front. The links are only
    reliable that way.
*/
    position = d->info.lastnameblock;
    for ( i = d->info.numberofnamesblocks - 1; i >= 0; i-- ) {
        fseek(d->handle,position,SEEK_SET);
        if ( minosread(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
            MesPrint("Error while reading file %s\n",d->name);
            goto thisiswrong;
        }
        convertnamesblock(&scratchnamesblock,ina[i],FROMDISK);
        if ( ina[i]->position != position ||
        ( ina[i]->previousblock <= 0 && i > 0 ) ) {
            MesPrint("File %s has inconsistent contents\n",d->name);
            goto thisiswrong;
        }
        position = ina[i]->previousblock;
    }
    d->info.firstnameblock = ina[0]->position;
/*
    Give the old info back to the system.
*/
    if ( d->iblocks ) {
        for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
            if ( d->iblocks[i] ) M_free(d->iblocks[i],"d->iblocks[i]");
        }
        M_free(d->iblocks,"d->iblocks");
    }
    if ( d->nblocks ) {
        for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
            if ( d->nblocks[i] ) M_free(d->nblocks[i],"d->nblocks[i]");
        }
        M_free(d->nblocks,"d->nblocks");
    }
/*
    And substitute the new blocks
*/
    d->iblocks = ib;
    d->nblocks = ina;
    return(0);
}

/*
    #] ReadIndex :
    #[ WriteIndexBlock :
*/

int WriteIndexBlock(DBASE *d,MLONG num)
{
    if ( num >= d->info.numberofindexblocks ) {
```

```
        MesPrint("Illegal number specified for number of index blocks\n");
        return(-1);
    }
    fseek(d->handle,d->iblocks[num]->position,SEEK_SET);
    convertblock(d->iblocks[num],&scratchblock,TODISK);
    if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
        MesPrint("Error while writing an index block in file %s\n",d->name);
        MesPrint("File may be unreliable now\n");
        return(-1);
    }
    return(0);
}

/*
    #] WriteIndexBlock :
    #[ WriteNamesBlock :
*/

int WriteNamesBlock(DBASE *d,MLONG num)
{
    if ( num >= d->info.numberofnamesblocks ) {
        MesPrint("Illegal number specified for number of names blocks\n");
        return(-1);
    }
    fseek(d->handle,d->nblocks[num]->position,SEEK_SET);
    convertnamesblock(d->nblocks[num],&scratchnamesblock,TODISK);
    if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
        MesPrint("Error while writing a names block in file %s\n",d->name);
        MesPrint("File may be unreliable now\n");
        return(-1);
    }
    return(0);
}

/*
    #] WriteNamesBlock :
    #[ WriteIndex :

    Problem here is to get the links right.
*/

int WriteIndex(DBASE *d)
{
    MLONG i, position;
    if ( d->iblocks == 0 ) return(0);
    if ( d->nblocks == 0 ) return(0);
    for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
        if ( d->iblocks[i] == 0 ) {
            MesPrint("Error: unassigned index blocks. Cannot write\n");
            return(-1);
        }
    }
    for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
        if ( d->nblocks[i] == 0 ) {
            MesPrint("Error: unassigned names blocks. Cannot write\n");
            return(-1);
        }
    }
    d->info.lastindexblock = -1;
    for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
        position = d->iblocks[i]->position;
        if ( position <= 0 ) {
            fseek(d->handle,0,SEEK_END);
            position = ftell(d->handle);
            d->iblocks[i]->position = position;
            if ( i <= 0 ) d->iblocks[i]->previousblock = -1;
            else d->iblocks[i]->previousblock = d->iblocks[i-1]->position;
        }
        else fseek(d->handle,position,SEEK_SET);
        convertblock(d->iblocks[i],&scratchblock,TODISK);
        if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
            MesPrint("Error while writing index of file %s",d->name);
            d->iblocks[i]->position = -1;
            return(-1);
        }
        d->info.lastindexblock = position;
    }
    d->info.lastnameblock = -1;
    for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
        position = d->nblocks[i]->position;
        if ( position <= 0 ) {
            fseek(d->handle,0,SEEK_END);
            position = ftell(d->handle);
            d->nblocks[i]->position = position;
            if ( i <= 0 ) d->nblocks[i]->previousblock = -1;
            else d->nblocks[i]->previousblock = d->nblocks[i-1]->position;
        }
```

```
        else fseek(d->handle,position,SEEK_SET);
        convertnamesblock(d->nblocks[i],&scratchnamesblock,TODISK);
        if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
            MesPrint("Error while writing index of file %s",d->name);
            d->nblocks[i]->position = -1;
            return(-1);
        }
        d->info.lastnameblock = position;
    }
    return(0);
}

/*
    #] WriteIndex :
    #[ WriteIniInfo :
*/

int WriteIniInfo(DBASE *d)
{
    INIINFO inf;
    fseek(d->handle,0,SEEK_SET);
    convertiniinfo(&(d->info),&inf,TODISK);
    if ( minoswrite(d->handle,(char *)(&inf),sizeof(INIINFO)) ) {
        MesPrint("Error while writing masterindex of file %s",d->name);
        return(-1);
    }
    return(0);
}

/*
    #] WriteIniInfo :
    #[ ReadIniInfo :
*/

int ReadIniInfo(DBASE *d)
{
    INIINFO inf;
    fseek(d->handle,0,SEEK_SET);
    if ( minosread(d->handle,(char *)(&inf),sizeof(INIINFO)) ) {
        MesPrint("Error while reading masterindex of file %s",d->name);
        return(-1);
    }
    convertiniinfo(&inf,&(d->info),FROMDISK);
    if ( d->info.entriesinindex < 0
    || d->info.numberofindexblocks < 0
    || d->info.lastindexblock < 0 ) {
        MesPrint("The file %s is not a proper database\n",d->name);
        return(-1);
    }
    return(0);
}

/*
    #] ReadIniInfo :
    #[ GetDbase :
*/

DBASE *GetDbase(char *filename)
{
    FILE *f;
    DBASE *d;
    char *newname;
    if ( ( f = LocateBase(&filename,&newname) ) == 0 ) {

        return(NewDbase(filename,0));
    }
/*  setbuf(f,0); */
    d = (DBASE *)From0List(&(AC.TableBaseList));
    d->mode = 0;
    d->tablenamessize = 0;
    d->topnumber = 0;
    d->tablenamefill = 0;
    d->iblocks = 0;
    d->nblocks = 0;
    d->tablenames = 0;

    d->info.entriesinindex = 0;
    d->info.numberofindexblocks = 0;
    d->info.firstindexblock = 0;
    d->info.lastindexblock = 0;
    d->info.numberoftables = 0;
    d->info.numberofnamesblocks = 0;
    d->info.firstnameblock = 0;
    d->info.lastnameblock = 0;

    d->name = str_dup(filename); /* For the moment just for the error messages */
    d->handle = f;
```

```
    if ( ReadIniInfo(d) || ReadIndex(d) ) { M_free(d,"index-d"); fclose(f); return(0); }
    if ( ComposeTableNames(d) < 0 ) { FreeTableBase(d); fclose(f); return(0); }
    d->name = str_dup(filename);
    d->fullname = newname;
    return(d);
}

/*
    #] GetDbase :
    #[ NewDbase :

    Creates a new database with 'number' entries in the index.
*/

DBASE *NewDbase(char *name,MLONG number)
{
    FILE *f;
    DBASE *d;
    MLONG numblocks, numnameblocks, i;
    char *s;
/*----------change 10-feb-2003 */
    int j, jj;
    MLONG t = (MLONG)(time(0));
/*-----------------------------*/
    if ( number < 0 ) number = 0;
    if ( ( f = fopen(name,"w+b") ) == 0 ) {
        MesPrint("Could not create a new file with name %s\n",name);
        return(0);
    }
    numblocks = (number+NUMOBJECTS-1)/NUMOBJECTS;
    numnameblocks = 1;
    if ( numblocks <= 0 ) numblocks = 1;
    if ( numnameblocks <= 0 ) numnameblocks = 1;
    d = (DBASE *)From0List(&(AC.TableBaseList));
    if ( ( d->iblocks = (INDEXBLOCK **)Malloc1(numblocks*sizeof(INDEXBLOCK *),
    "new database") ) == 0 ) {
        NumTableBases--;
        return(0);
    }
    d->tablenames = 0;
    d->tablenamessize = 0;
    d->topnumber = 0;
    d->tablenamefill = 0;

    d->mode = 0;
    if ( ( d->nblocks = (NAMESBLOCK **)Malloc1(sizeof(NAMESBLOCK *)*numnameblocks,
    "new database") ) == 0 ) {
        M_free(d->iblocks,"new database");
        NumTableBases--;
        return(0);
    }
    if ( ( f = fopen(name,"w+b") ) == 0 ) {
        MesPrint("Could not create new file %s\n",name);
        NumTableBases--;
        return(0);
    }
/*  setbuf(f,0); */
    d->name = str_dup(name);
    d->fullname = str_dup(name);
    d->handle = f;

    d->info.entriesinindex = number;
    d->info.numberofindexblocks = numblocks;
    d->info.numberofnamesblocks = numnameblocks;
    d->info.firstindexblock = 0;
    d->info.lastindexblock = 0;
    d->info.numberoftables = 0;
    d->info.firstnameblock = 0;
    d->info.lastnameblock = 0;

    if ( WriteIniInfo(d) ) {
getout:
        fclose(f);
        remove(d->fullname);
        if ( d->name ) { M_free(d->name,"name tablebase"); d->name = 0; }
        if ( d->fullname ) { M_free(d->fullname,"fullname tablebase"); d->fullname = 0; }
        M_free(d->nblocks,"new database");
        M_free(d->iblocks,"new database");
        NumTableBases--;
        return(0);
    }
    for ( i = 0; i < numblocks; i++ ) {
        if ( ( d->iblocks[i] = (INDEXBLOCK *)Malloc1(sizeof(INDEXBLOCK),
        "index blocks of new database") ) == 0 ) {
            while ( --i >= 0 ) M_free(d->iblocks[i],"index blocks of new database");
            goto getout;
        }
```

```
        if ( i > 0 ) d->iblocks[i]->previousblock = d->iblocks[i-1]->position;
        else d->iblocks[i]->previousblock = -1;
        d->iblocks[i]->position = ftell(f);
/*----------change 10-feb-2003 */
/*
            Zero things properly. We don't want garbage in the file.
*/
            for ( j = 0; j < NUMOBJECTS; j++ ) {
                d->iblocks[i]->objects[j].date = t;
                d->iblocks[i]->objects[j].size = 0;
                d->iblocks[i]->objects[j].position = -1;
                d->iblocks[i]->objects[j].tablenumber = 0;
                d->iblocks[i]->objects[j].uncompressed = 0;
                d->iblocks[i]->objects[j].spare1 = 0;
                d->iblocks[i]->objects[j].spare2 = 0;
                d->iblocks[i]->objects[j].spare3 = 0;
                for ( jj = 0; jj < ELEMENTSIZE; jj++ ) d->iblocks[i]->objects[j].element[jj] = 0;
            }
        convertblock(d->iblocks[i],&scratchblock,TODISK);
        if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
            MesPrint("Error while writing new index blocks\n");
            goto getout;
        }
    }
    for ( i = 0; i < numnameblocks; i++ ) {
        if ( ( d->nblocks[i] = (NAMESBLOCK *)Malloc1(sizeof(NAMESBLOCK),
        "names blocks of new database") ) == 0 ) {
            while ( --i >= 0 ) { M_free(d->nblocks[i],"names blocks of new database"); }
            for ( i = 0; i < numblocks; i++ ) M_free(d->iblocks[i],"index blocks of new database");
            goto getout;
        }
        if ( i > 0 ) d->nblocks[i]->previousblock = d->nblocks[i-1]->position;
        else d->nblocks[i]->previousblock = -1;
        d->nblocks[i]->position = ftell(f);
        s = d->nblocks[i]->names;
        for ( j = 0; j < NAMETABLESIZE; j++ ) *s++ = 0;
        convertnamesblock(d->nblocks[i],&scratchnamesblock,TODISK);
        if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
            MesPrint("Error while writing new names blocks\n");
            for ( i = 0; i < numnameblocks; i++ ) M_free(d->nblocks[i],"names blocks of new
database");
            for ( i = 0; i < numblocks; i++ ) M_free(d->iblocks[i],"index blocks of new database");
            goto getout;
        }
    }
    d->info.firstindexblock = d->iblocks[0]->position;
    d->info.lastindexblock = d->iblocks[numblocks-1]->position;
    d->info.firstnameblock = d->nblocks[0]->position;
    d->info.lastnameblock = d->nblocks[numnameblocks-1]->position;
    if ( WriteIniInfo(d) ) {
        for ( i = 0; i < numnameblocks; i++ ) M_free(d->nblocks[i],"names blocks of new database");
        for ( i = 0; i < numblocks; i++ ) M_free(d->iblocks[i],"index blocks of new database");
        goto getout;
    }
    return(d);
}

/*
    #] NewDbase :
    #[ FreeTableBase :
*/

void FreeTableBase(DBASE *d)
{
    int i, j, *old, *newL;
    LIST *L;
    for ( i = 0; i < d->info.numberofnamesblocks; i++ ) M_free(d->nblocks[i],"nblocks[i]");
    for ( i = 0; i < d->info.numberofindexblocks; i++ ) M_free(d->iblocks[i],"iblocks[i]");
    M_free(d->nblocks,"nblocks");
    M_free(d->iblocks,"iblocks");
    if ( d->tablenames ) M_free(d->tablenames,"d->tablenames");
    if ( d->name ) M_free(d->name,"d->name");
    if ( d->fullname ) M_free(d->fullname,"d->fullname");
    i = d - tablebases;
    if ( i < ( NumTableBases - 1 ) ) {
        L = &(AC.TableBaseList);
        j = ( ( NumTableBases - i - 1 ) * L->size ) / sizeof(int);
        old = (int *)d; newL = (int *)(d+1);
        while ( --j >= 0 ) *newL++ = *old++;
        j = L->size / sizeof(int);
        while ( --j >= 0 ) *newL++ = 0;
    }
    NumTableBases--;
    M_free(d,"tb,d");
}

/*
```

```
    #] FreeTableBase :
    #[ ComposeTableNames :

        The nameblocks are supposed to be in memory.
        Hence we have to go through them
*/

int ComposeTableNames(DBASE *d)
{
    MLONG nsize = 0;
    int i, j, k;
    char *s, *t, *ss;
    d->topnumber = 0;
    i = 0; s = d->nblocks[i]->names; j = NAMETABLESIZE;
    while ( *s ) {
      if ( *s ) d->topnumber++;
      for ( k = 0; k < 2; k++ ) { /* name and argtail */
        while ( *s ) {
            j--;
            if ( j <= 0 ) {
                i++; if ( i >= d->info.numberofnamesblocks ) goto gotall;
                s = d->nblocks[i]->names; j = NAMETABLESIZE;
            }
            else s++;
        }
        j--;
        if ( j <= 0 ) {
            i++; if ( i >= d->info.numberofnamesblocks ) goto gotall;
            s = d->nblocks[i]->names; j = NAMETABLESIZE;
        }
        else s++;
      }
    }
gotall:;
    nsize = (d->info.numberofnamesblocks-1)*NAMETABLESIZE +
                (s-d->nblocks[i]->names)+1;
    if ( ( d->tablenames = (char *)Malloc1((2*nsize+30)*sizeof(char),"tablenames") )
        == 0 ) { return(-1); }
    t = d->tablenames;
    d->tablenamessize = 2*nsize+30;
    d->tablenamefill = nsize-1;
    for ( k = 0; k < i; k++ ) {
        ss = d->nblocks[k]->names;
        for ( j = 0; j < NAMETABLESIZE; j++ ) *t++ = *ss++;
    }
    ss = d->nblocks[i]->names;
    while ( ss < s ) *t++ = *ss++;
    *t = 0;
    return(0);
}

/*
    #] ComposeTableNames :
    #[ OpenDbase :
*/

DBASE *OpenDbase(char *filename)
{
    FILE *f;
    DBASE *d;
    char *newname;
    if ( ( f = LocateBase(&filename,&newname) ) == 0 ) {
        MesPrint("Cannot open file %s\n",filename);
        return(0);
    }
/*  setbuf(f,0); */
    d = (DBASE *)From0List(&(AC.TableBaseList));
    d->name = filename; /* For the moment just for the error messages */
    d->handle = f;
    if ( ReadIniInfo(d) || ReadIndex(d) ) { M_free(d,"OpenDbase"); fclose(f); return(0); }
    if ( ComposeTableNames(d) ) {
        FreeTableBase(d);
        fclose(f);
        return(0);
    }
    d->name = str_dup(filename);
    d->fullname = newname;
    return(d);
}

/*
    #] OpenDbase :
    #[ AddTableName :

    Adds a name of a table. Writes the namelist to disk.
    Returns the number of this tablename in the database.
    If the name was already in the table we return its value in negative.
```

```
    Zero is an error!
*/

MLONG AddTableName(DBASE *d,char *name,TABLES T)
{
    char *s, *t, *tt;
    int namesize, tailsize;
    MLONG newsize, i, num;
/*
    First search for the name in what we have already
*/
    if ( d->tablenames ) {
        num = 0;
        s = d->tablenames;
        while ( *s ) {
            num++;
            t = name;
            while ( ( *s == *t ) && *t ) { s++; t++; }
            if ( *s == *t ) { return(-num); }
            while ( *s ) s++;
            s++;
            while ( *s ) s++;
            s++;
        }
    }
/*
    This name has to be added
*/
    MesPrint("We add the name %s\n",name);
    t = name;
    while ( *t ) { t++; }
    namesize = t-name;
    if ( ( t = (char *)(T->argtail) ) != 0 ) {
        while ( *t ) { t++; }
        tailsize = t - (char *)(T->argtail);
    }
    else { tailsize = 0; }
    if ( d->tablenames == 0 ) {
        if ( ComposeTableNames(d) ) {
            FreeTableBase(d);
            M_free(d,"AddTableName");
            return(0);
        }
    }
    d->info.numberoftables++;
    while ( ( d->tablenamefill+namesize+tailsize+3 > d->tablenamessize )
                    || ( d->tablenames == 0 ) ) {
        newsize = 2*d->tablenamessize + 2*namesize + 2*tailsize + 6;
        if ( ( t = (char *)Malloc1(newsize*sizeof(char),"AddTableName") ) == 0 )
            return(0);
        tt = t;
        if ( d->tablenames ) {
            s = d->tablenames;
            for ( i = 0; i < d->tablenamefill; i++ ) *t++ = *s++;
            *t = 0;
            M_free(d->tablenames,"d->tablenames");
        }
        d->tablenames = tt;
        d->tablenamessize = newsize;
    }
    s = d->tablenames + d->tablenamefill;
    t = name;
    while ( *t ) *s++ = *t++;
    *s++ = 0;
    t = (char *)(T->argtail);
    while ( *t ) *s++ = *t++;
    *s++ = 0;
    *s = 0;
    d->tablenamefill = s - d->tablenames;
    d->topnumber++;
/*
    Now we have to synchronize
*/
    if ( PutTableNames(d) ) return(0);
    return(d->topnumber);
}

/*
    #] AddTableName :
    #[ GetTableName :

    Gets a name of a table.
    Returns the number of this tablename in the database.
    Zero -> error
*/

MLONG GetTableName(DBASE *d,char *name)
```

```
{
    char *s, *t;
    MLONG num;
/*
    search for the name in what we have
*/
    if ( d->tablenames ) {
        num = 0;
        s = d->tablenames;
        while ( *s ) {
            num++;
            t = name;
            while ( ( *s == *t ) && *t ) { s++; t++; }
            if ( *s == *t ) { return(num); }
            while ( *s ) s++;
            s++;
            while ( *s ) s++;
            s++;
        }
    }
    return(0);
}

/*
    #] GetTableName :
    #[ PutTableNames :

    Takes the names string in d->tablenames and puts it in the nblocks
    pieces. Writes what has been changed to disk.
*/

int PutTableNames(DBASE *d)
{
    NAMESBLOCK **nnew;
    int i, j, firstdif;
    MLONG m;
    char *s, *t;
/*
    Determine how many blocks are needed.
*/
    MLONG numblocks = d->tablenamefill/NAMETABLESIZE + 1;
    if ( d->info.numberofnamesblocks < numblocks ) {
/*
        We need more blocks. First make sure of the space for nblocks.
*/
        if ( ( nnew = (NAMESBLOCK **)Malloc1(sizeof(NAMESBLOCK *)*numblocks,
            "new names block") ) == 0 ) {
            return(-1);
        }
        for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
            nnew[i] = d->nblocks[i];
        }
        for ( ; i < numblocks; i++ ) {
            if ( ( d->nblocks[i] = (NAMESBLOCK *)Malloc1(sizeof(NAMESBLOCK),
            "additional names blocks ") ) == 0 ) {
                FreeTableBase(d);
                return(-1);
            }
            d->nblocks[i]->previousblock = -1;
            d->nblocks[i]->position = -1;
            s = d->nblocks[i]->names;
            for ( j = 0; j < NAMETABLESIZE; j++ ) *s++ = 0;
        }
        d->info.numberofnamesblocks = numblocks;
    }
/*
    Now look till where the new contents agree with the old.
*/
    firstdif = 0;
    i = 0; t = d->nblocks[i]->names; j = 0; s = d->tablenames;
    for ( m = 0; m < d->tablenamefill; m++ ) {
        if ( *s == *t ) {
            s++; t++; j++;
            if ( j >= NAMETABLESIZE ) {
                i++;
                t = d->nblocks[i]->names;
                j = 0;
            }
        }
        else {
            firstdif = i;
            for ( ; m < d->tablenamefill; m++ ) {
                *t++ = *s++; j++;
            }
            if ( j >= NAMETABLESIZE ) {
                i++;
                t = d->nblocks[i]->names;
```

```
                j = 0;
            }
            *t = 0;
            break;
        }
    }
    for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
        if ( i == firstdif ) break;
        if ( d->nblocks[i]->position < 0 ) { firstdif = i; break; }
    }
/*
    Now we have to (re)write the blocks, starting at firstdif.
*/
    for ( i = firstdif; i < d->info.numberofnamesblocks; i++ ) {
        if ( i > 0 ) d->nblocks[i]->previousblock = d->nblocks[i-1]->position;
        else d->nblocks[i]->previousblock = -1;
        if ( d->nblocks[i]->position < 0 ) {
            fseek(d->handle,0,SEEK_END);
            d->nblocks[i]->position = ftell(d->handle);
        }
        else fseek(d->handle,d->nblocks[i]->position,SEEK_SET);
        convertnamesblock(d->nblocks[i],&scratchnamesblock,TODISK);
        if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
            MesPrint("Error while writing names blocks\n");
            FreeTableBase(d);
            return(-1);
        }
    }
    d->info.lastnameblock = d->nblocks[d->info.numberofnamesblocks-1]->position;
    d->info.firstnameblock = d->nblocks[0]->position;
    return(WriteIniInfo(d));
}

/*
    #] PutTableNames :
    #[ AddToIndex :
*/

int AddToIndex(DBASE *d,MLONG number)
{
    MLONG i, oldnumofindexblocks = d->info.numberofindexblocks;
    MLONG j, newnumofindexblocks, jj;
    INDEXBLOCK **ib;
    MLONG t = (MLONG)(time(0));
    if ( number == 0 ) return(0);
    else if ( number < 0 ) {
        if ( d->info.entriesinindex < -number ) {
            MesPrint("There are only %ld entries in the index of file %s\n",
            d->info.entriesinindex,d->name);
            return(-1);
        }
        d->info.entriesinindex += number;
dowrite:
        if ( WriteIniInfo(d) ) {
            d->info.entriesinindex -= number;
            MesPrint("File may be corrupted\n");
            return(-1);
        }
    }
    else if ( d->info.entriesinindex+number <=
    NUMOBJECTS*d->info.numberofindexblocks ) {
        d->info.entriesinindex += number;
        goto dowrite;
    }
    else {
        d->info.entriesinindex += number;
        newnumofindexblocks = d->info.numberofindexblocks + ((number -
            (NUMOBJECTS*d->info.numberofindexblocks - d->info.entriesinindex))
            +NUMOBJECTS-1)/NUMOBJECTS;
        if ( ( ib = (INDEXBLOCK **)Malloc1(sizeof(INDEXBLOCK *)*newnumofindexblocks,
        "index") ) == 0 ) return(-1);
        for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
            ib[i] = d->iblocks[i];
        }
        for ( i = d->info.numberofindexblocks; i < newnumofindexblocks; i++ ) {
            if ( ( ib[i] = (INDEXBLOCK *)Malloc1(sizeof(INDEXBLOCK),"index block") ) == 0 ) {
                FreeTableBase(d);
                return(-1);
            }
            if ( i > 0 ) ib[i]->previousblock = ib[i-1]->position;
            else ib[i]->previousblock = -1;
/*
            Zero things properly. We don't want garbage in the file.
*/
            for ( j = 0; j < NUMOBJECTS; j++ ) {
                ib[i]->objects[j].date = t;
                ib[i]->objects[j].size = 0;
```

```
                ib[i]->objects[j].position = -1;
                ib[i]->objects[j].tablenumber = 0;
                ib[i]->objects[j].uncompressed = 0;
                ib[i]->objects[j].spare1 = 0;
                ib[i]->objects[j].spare2 = 0;
                ib[i]->objects[j].spare3 = 0;
                for ( jj = 0; jj < ELEMENTSIZE; jj++ ) ib[i]->objects[j].element[jj] = 0;
            }
            fseek(d->handle,0,SEEK_END);
            ib[i]->position = ftell(d->handle);
            convertblock(ib[i],&scratchblock,TODISK);
            if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
                MesPrint("Error while writing new index of file %s",d->name);
                FreeTableBase(d);
                return(-1);
            }
        }
        d->info.lastindexblock = ib[newnumofindexblocks-1]->position;
        d->info.firstindexblock = ib[0]->position;
        d->info.numberofindexblocks = newnumofindexblocks;
        if ( WriteIniInfo(d) ) {
            d->info.numberofindexblocks = oldnumofindexblocks;
            d->info.entriesinindex -= number;
            MesPrint("File may be corrupted\n");
            FreeTableBase(d);
            return(-1);
        }
        M_free(d->iblocks,"AddToIndex");
        d->iblocks = ib;
    }
    return(0);
}

/*
    #] AddToIndex :
    #[ AddObject :
*/

MLONG AddObject(DBASE *d,MLONG tablenumber,char *arguments,char *rhs)
{
    MLONG number;
    number = d->info.entriesinindex;
    if ( AddToIndex(d,1) ) return(-1);
    if ( WriteObject(d,tablenumber,arguments,rhs,number) ) return(-1);
    return(number);
}

/*
    #] AddObject :
    #[ FindTableNumber :
*/

MLONG FindTableNumber(DBASE *d,char *name)
{
    char *s = d->tablenames, *t, *ss;
    MLONG num = 0;
    ss = d->tablenames + d->tablenamefill;
    while ( s < ss ) {
        num++;
        t = name;
        while ( *s == *t && *t ) {
            s++; t++;
        }
        if ( *s == 0 && *t == 0 ) return(num);
        while ( *s ) s++;
        s++;
/*
        Skip also the argument tail
*/
        while ( *s ) s++;
        s++;
    }
    return(-1);     /* Name not found */
}

/*
    #] FindTableNumber :
    #[ WriteObject :
*/

int WriteObject(DBASE *d,MLONG tablenumber,char *arguments,char *rhs,MLONG number)
{
    char *s, *a;
#ifdef WITHZLIB
    char *buffer = 0;
    uLongf newsize = 0, oldsize = 0;
    uLong ssize;
```

```
    int error = 0;
#endif
    MLONG i, j, position, size, n;
    OBJECTS *obj;
    if ( ( d->mode & INPUTONLY ) == INPUTONLY ) {
        MesPrint("Not allowed to write to input\n");
        return(-1);
    }
    if ( number >= d->info.entriesinindex ) {
        MesPrint("Reference to non-existing object number %ld\n",number+1);
        return(0);
    }
    j = number/NUMOBJECTS;
    i = number%NUMOBJECTS;
    obj = &(d->iblocks[j]->objects[i]);
    a = arguments;
    while ( *a ) a++;
    a++; n = a - arguments;
    if ( n > ELEMENTSIZE ) {
        MesPrint("Table element %s has more than %ld characters.\n",arguments,
            (MLONG)ELEMENTSIZE);
        return(-1);
    }
    s = obj->element;
    a = arguments;
    while ( *a ) *s++ = *a++;
    *s++ = 0;
    while ( n < ELEMENTSIZE ) { *s++ = 0; n++; }
    obj->spare1 = obj->spare2 = obj->spare3 = 0;

    fseek(d->handle,0,SEEK_END);
    position = ftell(d->handle);
    s = rhs;
    while ( *s ) s++;
    s++;
    size = s - rhs;
#ifdef WITHZLIB
    if ( ( d->mode & NOCOMPRESS ) == 0 ) {
        newsize = size + size/1000 + 20;
        if ( ( buffer = (char *)Malloc1(newsize*sizeof(char),"compress buffer") )
            == 0 ) {
            MesPrint("No compress used for element %s in file %s\n",arguments,d->name);
        }
    }
    else buffer = 0;
    if ( buffer ) {
        ssize = size;
        if ( ( error = compress((Bytef *)buffer,&newsize,(Bytef *)rhs,ssize) ) != Z_OK ) {
            MesPrint("Error = %d\n",error);
            MesPrint("Due to error no compress used for element %s in file %s\n",arguments,d->name);
            M_free(buffer,"tb,WriteObject");
            buffer = 0;
        }
    }
    if ( buffer ) {
        rhs = buffer;
        oldsize = size;
        size = newsize;
    }
#endif
    if ( minoswrite(d->handle,rhs,size) ) {
        MesPrint("Error while writing rhs\n");
        return(-1);
    }
    obj->position = position;
    obj->size = size;
    obj->date = (MLONG)(time(0));
    obj->tablenumber = tablenumber;
#ifdef WITHZLIB
    obj->uncompressed = oldsize;
    if ( buffer ) M_free(buffer,"tb,WriteObject");
#else
    obj->uncompressed = 0;
#endif
    return(WriteIndexBlock(d,j));
}

/*
    #] WriteObject :
    #[ ReadObject :

    Returns a pointer to the proper rhs
*/

char *ReadObject(DBASE *d,MLONG tablenumber,char *arguments)
{
    OBJECTS *obj;
```

```
    MLONG i, j;
    char *buffer1, *s, *t;
#ifdef WITHZLIB
    char *buffer2 = 0;
    uLongf finallength = 0;
#endif
    if ( tablenumber > d->topnumber ) {
        MesPrint("Reference to non-existing table number in tablebase %s: %ld\n",
                d->name,tablenumber);
        return(0);
    }
/*
    Start looking for the object
*/
    for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
        for ( j = 0; j < NUMOBJECTS; j++ ) {
            if ( d->iblocks[i]->objects[j].tablenumber != tablenumber ) continue;
            s = arguments; t = d->iblocks[i]->objects[j].element;
            while ( *s == *t && *s ) { s++; t++; }
            if ( *t == 0 && *s == 0 ) goto foundelement;
        }
    }
    s = d->tablenames; i = 1;
    while ( *s ) {
        if ( i == tablenumber ) break;
        while ( *s ) s++; s++;
        while ( *s ) s++; s++;
        i++;
    }
    MesPrint("%s(%s) not found in tablebase %s\n",s,arguments,d->name);
    return(0);

foundelement:;
    obj = &(d->iblocks[i]->objects[j]);
    fseek(d->handle,obj->position,SEEK_SET);
    if ( ( buffer1 = (char *)Malloc1(obj->size,"reading rhs buffer1") ) == 0 ) {
        return(0);
    }
#ifdef WITHZLIB
    if ( obj->uncompressed > 0 ) {
        if ( ( buffer2 = (char *)Malloc1(obj->uncompressed,"reading rhs buffer2") ) == 0 ) {
            return(0);
        }
    }
    else buffer2 = 0;
#endif
    if ( minosread(d->handle,buffer1,obj->size) ) {
        MesPrint("Could not read rhs %s in file %s\n",arguments,d->name);
        M_free(buffer1,"tb,ReadObject");
#ifdef WITHZLIB
        if ( buffer2 ) M_free(buffer2,"tb,ReadObject");
#endif
        return(0);
    }
#ifdef WITHZLIB
    if ( buffer2 == 0 ) return(buffer1);
    finallength = obj->uncompressed;
    if ( uncompress((Bytef *)buffer2,&finallength,(Bytef *)buffer1,obj->size) != Z_OK ) {
        MesPrint("Cannot uncompress element %s in file %s\n",arguments,d->name);
        M_free(buffer1,"tb,ReadObject"); M_free(buffer2,"tb,ReadObject");
        return(0);
    }
    M_free(buffer1,"tb,ReadObject");
    return(buffer2);
#else
    return(buffer1);
#endif
}

/*
    #] ReadObject :
    #[ ReadijObject :

    Returns a pointer to the proper rhs
*/

char *ReadijObject(DBASE *d,MLONG i,MLONG j,char *arguments)
{
    OBJECTS *obj;
    char *buffer1;
#ifdef WITHZLIB
    char *buffer2 = 0;
    uLongf finallength = 0;
#endif
    obj = &(d->iblocks[i]->objects[j]);
    fseek(d->handle,obj->position,SEEK_SET);
    if ( ( buffer1 = (char *)Malloc1(obj->size,"reading rhs buffer1") ) == 0 ) {
```

```
        return(0);
    }
#ifdef WITHZLIB
    if ( obj->uncompressed > 0 ) {
        if ( ( buffer2 = (char *)Malloc1(obj->uncompressed,"reading rhs buffer2") ) == 0 ) {
            return(0);
        }
    }
    else buffer2 = 0;
#endif
    if ( minosread(d->handle,buffer1,obj->size) ) {
        MesPrint("Could not read rhs %s in file %s\n",arguments,d->name);
        if ( buffer1 ) M_free(buffer1,"rhs buffer1");
#ifdef WITHZLIB
        if ( buffer2 ) M_free(buffer2,"rhs buffer2");
#endif
        return(0);
    }
#ifdef WITHZLIB
    if ( buffer2 == 0 ) return(buffer1);
    finallength = obj->uncompressed;
    if ( uncompress((Bytef *)buffer2,&finallength,(Bytef *)buffer1,obj->size) != Z_OK ) {
        MesPrint("Cannot uncompress element %s in file %s\n",arguments,d->name);
        if ( buffer1 ) M_free(buffer1,"rhs buffer1");
        if ( buffer2 ) M_free(buffer2,"rhs buffer2");
        return(0);
    }
    M_free(buffer1,"rhs buffer1");
    return(buffer2);
#else
    return(buffer1);
#endif
}

/*
    #] ReadijObject :
    #[ ExistsObject :

    Returns 1 if Object exists
*/

int ExistsObject(DBASE *d,MLONG tablenumber,char *arguments)
{
    MLONG i, j;
    char *s, *t;
    if ( tablenumber > d->topnumber ) {
        MesPrint("Reference to non-existing table number in tablebase %s: %ld\n",
                d->name,tablenumber);
        return(0);
    }
/*
    Start looking for the object
*/
    for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
        for ( j = 0; j < NUMOBJECTS; j++ ) {
            if ( d->iblocks[i]->objects[j].tablenumber != tablenumber ) continue;
            s = arguments; t = d->iblocks[i]->objects[j].element;
            while ( *s == *t && *s ) { s++; t++; }
            if ( *t == 0 && *s == 0 ) return(1);
        }
    }
    return(0);
}

/*
    #] ExistsObject :
    #[ DeleteObject :

    Returns 1 if Object has been deleteted.
    We leave a hole. Actually the object is still there but has been
    inactivated. It can be reactivated by calling this routine again.
*/

int DeleteObject(DBASE *d,MLONG tablenumber,char *arguments)
{
    MLONG i, j;
    char *s, *t;
    if ( tablenumber > d->topnumber ) {
        MesPrint("Reference to non-existing table number in tablebase %s: %ld\n",
                d->name,tablenumber);
        return(0);
    }
/*
    Start looking for the object
*/
    for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
        for ( j = 0; j < NUMOBJECTS; j++ ) {
```

```
            if ( d->iblocks[i]->objects[j].tablenumber != tablenumber ) continue;
            s = arguments; t = d->iblocks[i]->objects[j].element;
            while ( *s == *t && *s ) { s++; t++; }
            if ( *t == 0 && *s == 0 ) {
                d->iblocks[i]->objects[j].tablenumber =
                    -d->iblocks[i]->objects[j].tablenumber - 1;
                return(1);
            }
        }
    }
    return(0);
}

/*
    #] DeleteObject :
*/
```

4.56 minos.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <time.h>
```

Include dependency graph for minos.h:

This graph shows which files directly or indirectly include this file:

Data Structures

- struct iniinfo
- struct objects
- struct indexblock
- struct nameblock
- struct dbase

Macros

- #define MAXBASES 16
- #define NUMOBJECTS 100
- #define MAXINDEXSIZE 33000000L
- #define NAMETABLESIZE 1008
- #define ELEMENTSIZE 128
- #define TODISK 0
- #define FROMDISK 1
- #define MCLEANFLAG -2L
- #define DIRTYFLAG 1
- #define INANDOUT 0
- #define INPUTONLY 1
- #define OUTPUTONLY 2
- #define NOCOMPRESS 4

Typedefs

- typedef struct iniinfo **INIINFO**
- typedef struct objects **OBJECTS**
- typedef struct indexblock **INDEXBLOCK**
- typedef struct nameblock **NAMESBLOCK**
- typedef struct dbase **DBASE**

Functions

- int minosread (FILE ∗f, char ∗buffer, MLONG size)
- int minoswrite (FILE ∗f, char ∗buffer, MLONG size)

Variables

- int withoutflush

4.56.1 Detailed Description

Contains all needed declarations and definitions for the tablebase low level file routines. These have been taken from the minos database system and modified somewhat.

!!!CAUTION!!! Changes in this file will most likely have consequences for the recovery mechanism (see checkpoint.c). You need to care for the code in checkpoint.c as well and modify the code there accordingly!

Definition in file minos.h.

4.56.2 Macro Definition Documentation

4.56.2.1 MAXBASES

```
#define MAXBASES 16
```

Definition at line 61 of file minos.h.

4.56.2.2 NUMOBJECTS

```
#define NUMOBJECTS 100
```

Definition at line 68 of file minos.h.

4.56.2.3 MAXINDEXSIZE

```
#define MAXINDEXSIZE 33000000L
```

Definition at line 69 of file minos.h.

4.56.2.4 NAMETABLESIZE

```
#define NAMETABLESIZE 1008
```

Definition at line 70 of file minos.h.

4.56.2.5 ELEMENTSIZE

```
#define ELEMENTSIZE 128
```

Definition at line 71 of file minos.h.

4.56.2.6 TODISK

```
#define TODISK 0
```

Definition at line 141 of file minos.h.

4.56.2.7 FROMDISK

```
#define FROMDISK 1
```

Definition at line 142 of file minos.h.

4.56.2.8 MCLEANFLAG

```
#define MCLEANFLAG -2L
```

Definition at line 143 of file minos.h.

4.56.2.9 DIRTYFLAG

```
#define DIRTYFLAG 1
```

Definition at line 144 of file minos.h.

4.56.2.10 INANDOUT

```
#define INANDOUT 0
```

Definition at line 145 of file minos.h.

4.56.2.11 INPUTONLY

```
#define INPUTONLY 1
```

Definition at line 146 of file minos.h.

4.56.2.12 OUTPUTONLY

```
#define OUTPUTONLY 2
```

Definition at line 147 of file minos.h.

4.56.2.13 NOCOMPRESS

```
#define NOCOMPRESS 4
```

Definition at line 148 of file minos.h.

4.56.3 Function Documentation

4.56.3.1 minosread()

```
int minosread (
            FILE * f,
            char * buffer,
            MLONG size )
```

Definition at line 66 of file minos.c.

4.56.3.2 minoswrite()

```
int minoswrite (
            FILE * f,
            char * buffer,
            MLONG size )
```

Definition at line 83 of file minos.c.

4.56.4 Variable Documentation

4.56.4.1 withoutflush

```
int withoutflush  [extern]
```

Definition at line 45 of file minos.c.

4.57 minos.h

Go to the documentation of this file.

```
#ifndef __MANAGE_H__

#define __MANAGE_H__

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <time.h>

/*
    The following typedef has been moved to form3.h where all the sizes
    are defined for the various memory models.
    We want MLONG to have a more or less fixed size.
    In form3.h we try to fix it at 8 bytes. This should make files exchangable
    between various 32-bits and 64-bits systems. At 4 bytes it might have
    problems with files of more than 2 Gbytes.

typedef long MLONG;
*/

#define MAXBASES 16
#ifdef WORDSIZE32
#define NUMOBJECTS 1024
#define MAXINDEXSIZE 1000000000L
#define NAMETABLESIZE 1008
#define ELEMENTSIZE 256
#else
#define NUMOBJECTS 100
#define MAXINDEXSIZE 33000000L
#define NAMETABLESIZE 1008
#define ELEMENTSIZE 128
#endif

int minosread(FILE *f,char *buffer,MLONG size);
int minoswrite(FILE *f,char *buffer,MLONG size);

/*
    ELEMENTSIZE should make a nice number of sizeof(OBJECTS)
```

```
    Usually this will be much too much, but there are cases.....
*/

typedef struct iniinfo {
/* should contains only MLONG variables or convertiniinfo should be modified */
    MLONG   entriesinindex;
    MLONG   numberofindexblocks;
    MLONG   firstindexblock;
    MLONG   lastindexblock;
    MLONG   numberoftables;
    MLONG   numberofnamesblocks;
    MLONG   firstnameblock;
    MLONG   lastnameblock;
} INIINFO;

typedef struct objects {
/* if any changes, convertblock should be adapted too!!!! */
    MLONG position;             /* position of RHS= */
    MLONG size;                 /* size on disk (could be compressed) */
    MLONG date;                 /* Time stamp */
    MLONG tablenumber;          /* Number of table. Refers to name in special index */
    MLONG uncompressed;         /* uncompressed size if compressed. If not: 0 */
    MLONG spare1;
    MLONG spare2;
    MLONG spare3;
    char element[ELEMENTSIZE];  /* table element in character form */
} OBJECTS;

typedef struct indexblock {
    MLONG   flags;
    MLONG   previousblock;
    MLONG   position;
    OBJECTS  objects[NUMOBJECTS];
} INDEXBLOCK;

typedef struct nameblock {
    MLONG   previousblock;
    MLONG   position;
    char    names[NAMETABLESIZE];
} NAMESBLOCK;

typedef struct dbase {
    INIINFO      info;
    MLONG       mode;
    MLONG       tablenamessize;
    MLONG       topnumber;
    MLONG       tablenamefill;
    INDEXBLOCK    **iblocks;
    NAMESBLOCK  **nblocks;
    FILE        *handle;
    char        *name;
    char        *fullname;
    char        *tablenames;
} DBASE;

/*
typedef int (*SFUN)(char *);
typedef struct compile {
    char *keyword;
    SFUN func;
} MCFUNCTION;
 */
#define TODISK 0
#define FROMDISK 1
#define MCLEANFLAG -2L
#define DIRTYFLAG 1
#define INANDOUT 0
#define INPUTONLY 1
#define OUTPUTONLY 2
#define NOCOMPRESS 4

extern int withoutflush;

#endif
```

4.58 module.c File Reference

`#include "form3.h"`
Include dependency graph for module.c:

Functions

- int ModuleInstruction (int ∗moduletype, int ∗specialtype)
- int CoModuleOption (UBYTE ∗s)
- int CoModOption (UBYTE ∗s)
- VOID SetSpecialMode (int moduletype, int specialtype)
- int ExecModule (int moduletype)
- int ExecStore ()
- VOID FullCleanUp ()
- int DoPolyfun (UBYTE ∗s)
- int DoPolyratfun (UBYTE ∗s)
- int DoNoParallel (UBYTE ∗s)
- int DoParallel (UBYTE ∗s)
- int DoModSum (UBYTE ∗s)
- int DoModMax (UBYTE ∗s)
- int DoModMin (UBYTE ∗s)
- int DoModLocal (UBYTE ∗s)
- int DoProcessBucket (UBYTE ∗s)
- UBYTE ∗ DoModDollar (UBYTE ∗s, int type)
- int DoinParallel (UBYTE ∗s)
- int DonotinParallel (UBYTE ∗s)
- int DoExecStatement ()
- int DoPipeStatement ()

4.58.1 Detailed Description

A number of routines that deal with the moduleoption statement and the execution of modules. Additionally there are the execution of the exec and pipe instructions.

Definition in file module.c.

4.58.2 Function Documentation

4.58.2.1 ModuleInstruction()

```
int ModuleInstruction (
            int * moduletype,
            int * specialtype )
```

Definition at line 76 of file module.c.

4.58.2.2 CoModuleOption()

```
int CoModuleOption (
            UBYTE * s )
```

Definition at line 143 of file module.c.

4.58.2.3 CoModOption()

```
int CoModOption (
            UBYTE * s )
```

Definition at line 210 of file module.c.

4.58.2.4 SetSpecialMode()

```
VOID SetSpecialMode (
            int moduletype,
            int specialtype )
```

Definition at line 255 of file module.c.

4.58.2.5 ExecModule()

```
int ExecModule (
            int moduletype )
```

Definition at line 272 of file module.c.

4.58.2.6 ExecStore()

```
int ExecStore ( )
```

Definition at line 282 of file module.c.

4.58.2.7 FullCleanUp()

```
VOID FullCleanUp ( )
```

Definition at line 297 of file module.c.

4.58.2.8 DoPolyfun()

```
int DoPolyfun (
            UBYTE * s )
```

Definition at line 374 of file module.c.

4.58.2.9 DoPolyratfun()

```
int DoPolyratfun (
            UBYTE * s )
```

Definition at line 437 of file module.c.

4.58.2.10 DoNoParallel()

```
int DoNoParallel (
            UBYTE * s )
```

Definition at line 508 of file module.c.

4.58.2.11 DoParallel()

```
int DoParallel (
            UBYTE * s )
```

Definition at line 523 of file module.c.

4.58.2.12 DoModSum()

```
int DoModSum (
            UBYTE * s )
```

Definition at line 538 of file module.c.

4.58.2.13 DoModMax()

```
int DoModMax (
            UBYTE * s )
```

Definition at line 558 of file module.c.

4.58.2.14 DoModMin()

```
int DoModMin (
            UBYTE * s )
```

Definition at line 578 of file module.c.

4.58.2.15 DoModLocal()

```
int DoModLocal (
            UBYTE * s )
```

Definition at line 598 of file module.c.

4.58.2.16 DoProcessBucket()

```
int DoProcessBucket (
            UBYTE * s )
```

Definition at line 618 of file module.c.

4.58.2.17 DoModDollar()

```
UBYTE * DoModDollar (
            UBYTE * s,
            int type )
```

Definition at line 636 of file module.c.

4.58.2.18 DoinParallel()

```
int DoinParallel (
            UBYTE * s )
```

Definition at line 737 of file module.c.

4.58.2.19 DonotinParallel()

```
int DonotinParallel (
            UBYTE * s )
```

Definition at line 747 of file module.c.

4.58.2.20 DoExecStatement()

```
int DoExecStatement ( )
```

Definition at line 759 of file module.c.

4.58.2.21 DoPipeStatement()

```
int DoPipeStatement ( )
```

Definition at line 776 of file module.c.

4.59 module.c

Go to the documentation of this file.

```
00001
00007 /* #[ License : */
00008 /*
00009  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00010  *   When using this file you are requested to refer to the publication
00011  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  *   This is considered a matter of courtesy as the development was paid
00013  *   for by FOM the Dutch physics granting agency and we would like to
00014  *   be able to track its scientific use to convince FOM of its value
00015  *   for the community.
00016  *
00017  *   This file is part of FORM.
00018  *
00019  *   FORM is free software: you can redistribute it and/or modify it under the
00020  *   terms of the GNU General Public License as published by the Free Software
00021  *   Foundation, either version 3 of the License, or (at your option) any later
00022  *   version.
00023  *
00024  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00027  *   details.
00028  *
00029  *   You should have received a copy of the GNU General Public License along
00030  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #] License : */
00033 /*
00034     #[ Includes :
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040     #] Includes :
00041     #[ Modules :
00042         #[ ModuleInstruction :
00043
00044         Enters after the . of .sort etc
00045         We have word[[(options)][:commentary];]
00046         Word is one of 'clear end global sort store'
00047         Options are 'polyfun, endloopifunchanged, endloopifzero'
00048         Additions for solving equations can be added to 'word'
00049
00050         The flag in the moduleoptions is for telling whether the current
00051         option can be followed by others. If it is > 0 it cannot.
00052 */
00053
00054 static KEYWORD ModuleWords[] = {
00055      {"clear",  (TFUN)0,    CLEARMODULE,    0}
00056     ,{"end",    (TFUN)0,    ENDMODULE,      0}
00057     ,{"global", (TFUN)0,    GLOBALMODULE,   0}
00058     ,{"sort",   (TFUN)0,    SORTMODULE,     0}
00059     ,{"store",  (TFUN)0,    STOREMODULE,    0}
00060 };
00061
00062 static KEYWORD ModuleOptions[] = {
00063      {"inparallel",         DoinParallel,   1,              1}
00064     ,{"local",              DoModLocal,     MODLOCAL,       0}
00065     ,{"maximum",            DoModMax,       MODMAX,         0}
00066     ,{"minimum",            DoModMin,       MODMIN,         0}
00067     ,{"noparallel",         DoNoParallel,   NOPARALLEL_USER,0}
00068     ,{"notinparallel",      DonotinParallel,0,              1}
00069     ,{"parallel",           DoParallel,     PARALLELFLAG,   0}
00070     ,{"polyfun",            DoPolyfun,      POLYFUN,        0}
00071     ,{"polyratfun",         DoPolyratfun,   POLYFUN,        0}
00072     ,{"processbucketsize",  DoProcessBucket,0,              0}
00073     ,{"sum",                DoModSum,       MODSUM,         0}
00074 };
```

```

int ModuleInstruction(int *moduletype, int *specialtype)
{
    UBYTE *t, *s, *u, c;
    KEYWORD *key;
    int addit = 0, error = 0, i, j;
    DUMMYUSE(specialtype);
    LoadInstruction(0);
    AC.firstctypemessage = 0;
    s = AP.preStart; SKIPBLANKS(s)
    t = EndOfToken(s); c = *t; *t = 0;
    AC.origin = FROMPOINTINSTRUCTION;
    key = FindKeyWord(AP.preStart,ModuleWords,sizeof(ModuleWords)/sizeof(KEYWORD));
    if ( key == 0 ) {
        MesPrint("@Unrecognized module terminator: %s",s);
        error = 1;
        key = ModuleWords;
        while ( StrCmp((UBYTE *)key->name,(UBYTE *)"end") ) key++;
    }
    *t = c;
    *moduletype = key->type;
    SKIPBLANKS(t);
    while ( *t == '(' ) {   /* There are options */
        s = t+1; SKIPBRA3(t)
        if ( *t == 0 ) {
            MesPrint("@Improper options field in . instruction");
            error = 1;
        }
        else {
            *t = 0;
            if ( CoModOption(s) ) error = 1;
            *t++ = ')';
        }
    }
    if ( *t == ':' ) {  /* There is an 'advertisement' */
        t++;
        SKIPBLANKS(t)
        s = t; i = 0;
        while ( *t && *t != ';' ) {
            if ( *t == '\\' ) t++;
            t++; i++;
        }
        u = t;
        while ( u > s && u[-1] == ' ' ) { u--; i--; }
        if ( *u == '\\' ) { u++; i++; }
        for ( j = COMMERCIALSIZE-1; j >= 0; j-- ) {
            if ( i <= 0 ) break;
            AC.Commercial[j] = *--u; i--;
            if ( u > s && u[-1] == '\\' ) u--;
        }
        for ( ; j >= 0; j-- ) AC.Commercial[j] = ' ';
        AC.Commercial[COMMERCIALSIZE] = 0;
        addit += 2;
    }
    if ( addit && *t != ';' ) {
        MesPrint("@Improper ending of . instruction");
        error = -1;
    }
    return(error);
}

/*
        #] ModuleInstruction :
        #[ CoModuleOption :

    ModuleOption, options;
*/

int CoModuleOption(UBYTE *s)
{
    UBYTE *t,*tt,c;
    KEYWORD *option;
    int error = 0, polyflag = 0;
    AC.origin = FROMMODULEOPTION;
    if ( *s ) do {
        s = ToToken(s);
        t = EndOfToken(s);
        c = *t; *t = 0;
        option = FindKeyWord(s,ModuleOptions,
            sizeof(ModuleOptions)/sizeof(KEYWORD));
        if ( option == 0 ) {
            if ( polyflag ) {
                *t = c; t++; s = SkipAName(t);
                polyflag = 0;
                continue;
            }
            else {
```

```
                MesPrint("@Unrecognized module option: %s",s);
                error = 1;
                polyflag = 0;
                *t = c;
            }
        }
        else {
            *t = c;
            SKIPBLANKS(t)
            if ( (option->func)(t) ) error = 1;
        }
        if ( StrCmp((UBYTE *)(option->name),(UBYTE *)("polyfun")) == 0
         || StrCmp((UBYTE *)(option->name),(UBYTE *)("polyratfun")) == 0 ) {
            polyflag = 1;
        }
        else polyflag = 0;
        if ( option->flags > 0 ) return(error);
        while ( *t ) {
            if ( *t == ',' ) {
                tt = t+1;
                while ( *tt == ',' ) tt++;
                if ( *tt != '$' ) break;
                t = tt+1;
            }
            if ( *t == ')' ) break;
            if ( *t == '(' ) SKIPBRA3(t)
            else if ( *t == '{' ) SKIPBRA2(t)
            else if ( *t == '[' ) SKIPBRA1(t)
            t++;
        }
        s = t;
    } while ( *s == ',' );
    if ( *s ) {
        MesPrint("@Unrecognized module option: %s",s);
        error = 1;
    }
    return(error);
}

/*
        #] CoModuleOption :
        #[ CoModOption :

    To be called from a .instruction.
    Only recognizes polyfun. The newer ones should be via the
    ModuleOption statement.
*/

int CoModOption(UBYTE *s)
{
    UBYTE *t,c;
    int error = 0;
    AC.origin = FROMPOINTINSTRUCTION;
    if ( *s ) do {
        s = ToToken(s);
        t = EndOfToken(s);
        c = *t; *t = 0;
        if ( StrICmp(s,(UBYTE *)"polyfun") == 0 ) {
            *t = c;
            SKIPBLANKS(t)
            if ( DoPolyfun(t) ) error = 1;
        }
        else if ( StrICmp(s,(UBYTE *)"polyratfun") == 0 ) {
            *t = c;
            SKIPBLANKS(t)
            if ( DoPolyratfun(t) ) error = 1;
        }
        else {
            MesPrint("@Unrecognized module option in .instruction: %s",s);
            error = 1;
            *t = c;
        }
        while ( *t ) {
            if ( *t == ',' || *t == ')' ) break;
            if ( *t == '(' ) SKIPBRA3(t)
            else if ( *t == '{' ) SKIPBRA2(t)
            else if ( *t == '[' ) SKIPBRA1(t)
            t++;
        }
        s = t;
    } while ( *s == ',' );
    if ( *s ) {
        MesPrint("@Unrecognized module option in .instruction: %s",s);
        error = 1;
    }
    return(error);
}
```

```
/*
        #] CoModOption :
        #[ SetSpecialMode :
*/

VOID SetSpecialMode(int moduletype, int specialtype)
{
    DUMMYUSE(moduletype); DUMMYUSE(specialtype);
}

/*
        #] SetSpecialMode :
        #[ MakeGlobal :

VOID MakeGlobal()
{
}

        #] MakeGlobal :
        #[ ExecModule :
*/

int ExecModule(int moduletype)
{
    return(DoExecute(moduletype,0));
}

/*
        #] ExecModule :
        #[ ExecStore :
*/

int ExecStore()
{
    return(0);
}

/*
        #] ExecStore :
        #[ FullCleanUp :

        Remark 27-oct-2005 by JV
        This routine (and CleanUp in startup.c) may still need some work:
            What to do with preprocessor variables
            What to do with files we write to
*/

VOID FullCleanUp()
{
    int j;

    while ( AC.CurrentStream->previous >= 0 )
        AC.CurrentStream = CloseStream(AC.CurrentStream);
    AP.PreSwitchLevel = AP.PreIfLevel = 0;

    for ( j = NumProcedures-1; j >= 0; j-- ) {
        if ( Procedures[j].name ) M_free(Procedures[j].name,"name of procedure");
        if ( Procedures[j].p.buffer ) M_free(Procedures[j].p.buffer,"buffer of procedure");
    }
    NumProcedures = 0;

    while ( NumPre > AP.gNumPre ) {
        NumPre--;
        M_free(PreVar[NumPre].name,"PreVar[NumPre].name");
        PreVar[NumPre].name = PreVar[NumPre].value = 0;
    }

    AC.DidClean = 0;
    for ( j = 0; j < NumExpressions; j++ ) {
        AC.exprnames->namenode[Expressions[j].node].type = CDELETE;
        AC.DidClean = 1;
    }

    CompactifyTree(AC.exprnames,EXPRNAMES);

    for ( j = AO.NumDictionaries-1; j >= 0; j-- ) {
        RemoveDictionary(AO.Dictionaries[j]);
        M_free(AO.Dictionaries[j],"Dictionary");
    }
    AO.NumDictionaries = AO.gNumDictionaries = 0;
    M_free(AO.Dictionaries,"Dictionaries");
    AO.Dictionaries = 0;
    AO.SizeDictionaries = 0;
    AP.OpenDictionary = 0;
    AO.CurrentDictionary = 0;

```

```
    AP.ComChar = AP.cComChar;
    if ( AP.procedureExtension ) M_free(AP.procedureExtension,"procedureextension");
    AP.procedureExtension = strDup1(AP.cprocedureExtension,"procedureextension");

    AC.StatsFlag = AM.gStatsFlag = AM.ggStatsFlag;
    AC.extrasymbols = AM.gextrasymbols = AM.ggextrasymbols;
    AC.extrasym[0] = AM.gextrasym[0] = AM.ggextrasym[0] = 'Z';
    AC.extrasym[1] = AM.gextrasym[1] = AM.ggextrasym[1] = 0;
    AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers = AM.ggNoSpacesInNumbers;
    AO.IndentSpace = AM.gIndentSpace = AM.ggIndentSpace;
    AC.ThreadStats = AM.gThreadStats = AM.ggThreadStats;
    AC.OldFactArgFlag = AM.gOldFactArgFlag = AM.ggOldFactArgFlag;
    AC.FinalStats = AM.gFinalStats = AM.ggFinalStats;
    AC.OldGCDflag = AM.gOldGCDflag = AM.ggOldGCDflag;
    AC.ThreadsFlag = AM.gThreadsFlag = AM.ggThreadsFlag;
    if ( AC.ThreadsFlag && AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
    AC.ThreadBucketSize = AM.gThreadBucketSize = AM.ggThreadBucketSize;
    AC.ThreadBalancing = AM.gThreadBalancing = AM.ggThreadBalancing;
    AC.ThreadSortFileSynch = AM.gThreadSortFileSynch = AM.ggThreadSortFileSynch;
    AC.ShortStatsMax = AM.gShortStatsMax = AM.ggShortStatsMax;
    AC.SizeCommuteInSet = AM.gSizeCommuteInSet = 0;

    NumExpressions = 0;
    if ( DeleteStore(0) < 0 ) {
        MesPrint("@Cannot restart the storage file");
        Terminate(-1);
    }
    RemoveDollars();
    CleanUp(1);
    ResetVariables(2);
    IniVars();
}

/*
        #] FullCleanUp :
        #[ DoPolyfun :
*/

int DoPolyfun(UBYTE *s)
{
    GETIDENTITY
    UBYTE *t, c;
    WORD funnum, eqsign = 0;
    if ( AC.origin == FROMPOINTINSTRUCTION ) {
        if ( *s == 0 || *s == ',' || *s == ')' ) {
            AR.PolyFun = 0; AR.PolyFunType = 0;
            return(0);
        }
        if ( *s != '=' ) {
            MesPrint("@Proper use in point instructions is: PolyFun[=functionname]");
            return(-1);
        }
        eqsign = 1;
    }
    else {
        if ( *s == 0 ) {
            AR.PolyFun = 0; AR.PolyFunType = 0;
            return(0);
        }
        if ( *s != '=' && *s != ',' ) {
            MesPrint("@Proper use is: PolyFun[{ ,=}functionname]");
            return(-1);
        }
        if ( *s == '=' ) eqsign = 1;
    }
    s++;
    SKIPBLANKS(s)
    t = EndOfToken(s);
    c = *t; *t = 0;

    if ( GetName(AC.varnames,s,&funnum,WITHAUTO) != CFUNCTION ) {
        if ( AC.origin != FROMPOINTINSTRUCTION && eqsign == 0 ) {
            AR.PolyFun = 0; AR.PolyFunType = 0;
            return(0);
        }
        MesPrint("@ %s is not a properly declared function",s);
        *t = c;
        return(-1);
    }
    if ( functions[funnum].spec != 0 || functions[funnum].commute != 0 ) {
        MesPrint("@The PolyFun must be a regular commuting function!");
        *t = c;
        return(-1);
    }
    AR.PolyFun = funnum+FUNCTION; AR.PolyFunType = 1;
    *t = c;
    SKIPBLANKS(t)
```

```
    if ( *t && *t != ',' && *t != ')' ) {
        t++; c = *t; *t = 0;
        MesPrint("@Improper ending of end-of-module instruction: %s",s);
        *t = c;
        return(-1);
    }
    return(0);
}

/*
        #] DoPolyfun :
        #[ DoPolyratfun :
*/

int DoPolyratfun(UBYTE *s)
{
    GETIDENTITY
    UBYTE *t, c;
    WORD funnum;
    if ( AC.origin == FROMPOINTINSTRUCTION ) {
        if ( *s == 0 || *s == ',' || *s == ')' ) {
            AR.PolyFun = 0; AR.PolyFunType = 0; AR.PolyFunInv = 0; AR.PolyFunExp = 0;
            return(0);
        }
        if ( *s != '=' ) {
            MesPrint("@Proper use in point instructions is: PolyRatFun[=functionname[+functionname]]");
            return(-1);
        }
    }
    else {
        if ( *s == 0 ) {
            AR.PolyFun = 0; AR.PolyFunType = 0; AR.PolyFunInv = 0; AR.PolyFunExp = 0;
            return(0);
        }
        if ( *s != '=' && *s != ',' ) {
            MesPrint("@Proper use is: PolyRatFun[{ ,=}functionname[+functionname]]");
            return(-1);
        }
    }
    s++;
    SKIPBLANKS(s)
    t = EndOfToken(s);
    c = *t; *t = 0;

    if ( GetName(AC.varnames,s,&funnum,WITHAUTO) != CFUNCTION ) {
Error1:;
        MesPrint("@ %s is not a properly declared function",s);
        *t = c;
        return(-1);
    }
    if ( functions[funnum].spec != 0 || functions[funnum].commute != 0 ) {
Error2:;
        MesPrint("@The PolyRatFun must be a regular commuting function!");
        *t = c;
        return(-1);
    }
    AR.PolyFun = funnum+FUNCTION; AR.PolyFunType = 2;
    AR.PolyFunInv = 0;
    AR.PolyFunExp = 0;
    AC.PolyRatFunChanged = 1;
    *t = c;
    if ( *t == '+' ) {
        t++; s = t;
        t = EndOfToken(s);
        c = *t; *t = 0;
        if ( GetName(AC.varnames,s,&funnum,WITHAUTO) != CFUNCTION ) goto Error1;
        if ( functions[funnum].spec != 0 || functions[funnum].commute != 0 ) goto Error2;
        AR.PolyFunInv = funnum+FUNCTION;
        *t = c;
    }
    SKIPBLANKS(t)
    if ( *t && *t != ',' && *t != ')' ) {
        t++; c = *t; *t = 0;
        MesPrint("@Improper ending of end-of-module instruction: %s",s);
        *t = c;
        return(-1);
    }
    return(0);
}

/*
        #] DoPolyratfun :
        #[ DoNoParallel :
*/

int DoNoParallel(UBYTE *s)
```

```
{
    if ( *s == 0 || *s == ',' || *s == ')' ) {
        AC.mparallelflag |= NOPARALLEL_USER;
        return(0);
    }
    MesPrint("@NoParallel should not have extra parameters");
    return(-1);
}

/*
        #] DoNoParallel :
        #[ DoParallel :
*/

int DoParallel(UBYTE *s)
{
    if ( *s == 0 || *s == ',' || *s == ')' ) {
        AC.mparallelflag &= ~NOPARALLEL_USER;
        return(0);
    }
    MesPrint("@Parallel should not have extra parameters");
    return(-1);
}

/*
        #] DoParallel :
        #[ DoModSum :
*/

int DoModSum(UBYTE *s)
{
    while ( *s == ',' ) s++;
    if ( *s != '$' ) {
        MesPrint("@Module Sum should mention which $-variables");
        return(-1);
    }
    s = DoModDollar(s,MODSUM);
    if ( s && *s != 0 && *s != ')' ) {
        MesPrint("@Irregular end of Sum option of Module statement");
        return(-1);
    }
    return(0);
}

/*
        #] DoModSum :
        #[ DoModMax :
*/

int DoModMax(UBYTE *s)
{
    while ( *s == ',' ) s++;
    if ( *s != '$' ) {
        MesPrint("@Module Maximum should mention which $-variables");
        return(-1);
    }
    s = DoModDollar(s,MODMAX);
    if ( s && *s != 0 ) {
        MesPrint("@Irregular end of Maximum option of Module statement");
        return(-1);
    }
    return(0);
}

/*
        #] DoModMax :
        #[ DoModMin :
*/

int DoModMin(UBYTE *s)
{
    while ( *s == ',' ) s++;
    if ( *s != '$' ) {
        MesPrint("@Module Minimum should mention which $-variables");
        return(-1);
    }
    s = DoModDollar(s,MODMIN);
    if ( s && *s != 0 ) {
        MesPrint("@Irregular end of Minimum option of Module statement");
        return(-1);
    }
    return(0);
}

/*
        #] DoModMin :
        #[ DoModLocal :
```

```
*/

int DoModLocal(UBYTE *s)
{
    while ( *s == ',' ) s++;
    if ( *s != '$' ) {
        MesPrint("@ModuleOption Local should mention which $-variables");
        return(-1);
    }
    s = DoModDollar(s,MODLOCAL);
    if ( s && *s != 0 ) {
        MesPrint("@Irregular end of Local option of ModuleOption statement");
        return(-1);
    }
    return(0);
}

/*
        #] DoModLocal :
        #[ DoProcessBucket :
*/

int DoProcessBucket(UBYTE *s)
{
    LONG x;
    while ( *s == ',' || *s == '=' ) s++;
    ParseNumber(x,s)
    if ( *s && *s != ' ' && *s != '\t' ) {
        MesPrint("&Numerical value expected for ProcessBucketSize");
        return(1);
    }
    AC.mProcessBucketSize = x;
    return(0);
}

/*
        #] DoProcessBucket :
        #[ DoModDollar :
*/

UBYTE * DoModDollar(UBYTE *s, int type)
{
    UBYTE *name, c;
    WORD number;
    MODOPTDOLLAR *md;
    while ( *s == '$' ) {
/*
        Read the name of the dollar
        Mark the type
*/
        s++;
        name = s;
        if ( FG.cTable[*s] == 0 ) {
            while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
            c = *s; *s = 0;
            number = GetDollar(name);
            if ( number < 0 ) {
                number = AddDollar(s,0,0,0);
                Warning("&Undefined $-variable in module statement");
            }
            md = (MODOPTDOLLAR *)FromList(&AC.ModOptDolList);
            md->number = number;
            md->type = type;
#ifdef WITHPTHREADS
            if ( type == MODLOCAL ) {
                int j, i;
                DOLLARS dglobal, dlocal;
                md->dstruct = (DOLLARS)Malloc1(
                    sizeof(struct DoLlArS)*AM.totalnumberofthreads,"Local DOLLARS");
/*
                Now copy the global dollar into the local copies.
                This can be nontrivial if the value needs an allocation.
                We don't really need the locks.
*/
                dglobal = Dollars + number;
                for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
                    dlocal = md->dstruct + j;
                    dlocal->index = dglobal->index;
                    dlocal->node = dglobal->node;
                    dlocal->type = dglobal->type;
                    dlocal->name = dglobal->name;
                    dlocal->size = dglobal->size;
                    dlocal->where = dglobal->where;
                    if ( dlocal->size > 0 ) {
                        dlocal->where = (WORD *)Malloc1((dlocal->size+1)*sizeof(WORD),"Local dollar value");
                        for ( i = 0; i < dlocal->size; i++ )
```

```
                            dlocal->where[i] = dglobal->where[i];
                        dlocal->where[dlocal->size] = 0;
                    }
                    dlocal->pthreadslockread = dummylock;
                    dlocal->pthreadslockwrite = dummylock;
                    dlocal->nfactors = dglobal->nfactors;
                    if ( dglobal->nfactors > 1 ) {
                        int nsize;
                        WORD *t, *m;
                        dlocal->factors = (FACDOLLAR *)Malloc1(dglobal->nfactors*sizeof(FACDOLLAR),"Dollar factors");
                        for ( i = 0; i < dglobal->nfactors; i++ ) {
                            nsize = dglobal->factors[i].size;
                            dlocal->factors[i].type = DOLUNDEFINED;
                            dlocal->factors[i].value = dglobal->factors[i].value;
                            if ( ( dlocal->factors[i].size = nsize ) > 0 ) {
                                dlocal->factors[i].where = t = (WORD *)Malloc1(sizeof(WORD)*(nsize+1),"DollarCopyFactor");
                                m = dglobal->factors[i].where;
                                NCOPY(t,m,nsize);
                                *t = 0;
                            }
                            else {
                                dlocal->factors[i].where = 0;
                            }
                        }
                    }
                    else { dlocal->factors = 0; }
                }
            }
            else {
                md->dstruct = 0;
            }
#endif
            *s = c;
        }
        else {
            MesPrint("&Illegal name for $-variable in module option");
            while ( *s != ',' && *s != 0 && *s != ')' ) s++;
        }
        while ( *s == ',' ) s++;
    }
    return(s);
}

/*
        #] DoModDollar :
        #[ DoinParallel :

        The idea is that we should have the commands
        ModuleOption,InParallel;
        ModuleOption,InParallel,name1,name2,...,namen;
        ModuleOption,NotInParallel,name1,name2,...,namen;
        The advantage over the InParallel statement is that this statement
        comes after the definition of the expressions.
*/

int DoinParallel(UBYTE *s)
{
    return(DoInParallel(s,1));
}

/*
        #] DoinParallel :
        #[ DonotinParallel :
*/

int DonotinParallel(UBYTE *s)
{
    return(DoInParallel(s,0));
}

/*
        #] DonotinParallel :
    #] Modules :
    #[ External :
        #[ DoExecStatement :
*/

int DoExecStatement()
{
#ifdef WITHSYSTEM
    FLUSHCONSOLE;
    if ( system((char *)(AP.preStart)) ) return(-1);
    return(0);
#else
    Error0("External programs not implemented on this computer/system");
```

```
    return(-1);
#endif
}

/*
        #] DoExecStatement :
        #[ DoPipeStatement :
*/

int DoPipeStatement()
{
#ifdef WITHPIPE
    FLUSHCONSOLE;
    if ( OpenStream(AP.preStart,PIPESTREAM,0,PRENOACTION) == 0 ) return(-1);
    return(0);
#else
    Error0("Pipes not implemented on this computer/system");
    return(-1);
#endif
}

/*
        #] DoPipeStatement :
    #] External :
*/
```

4.60 mpi.c File Reference

```
#include <limits.h>
#include "form3.h"
```

Include dependency graph for mpi.c:

Data Structures

- struct longMultiStruct

Macros

- #define PF_PACKSIZE 1600
- #define MPI_ERRCODE_CHECK(err)

Typedefs

- typedef struct longMultiStruct **PF_LONGMULTI**

Functions

- LONG PF_RealTime (int i)
- int PF_LibInit (int ∗argcp, char ∗∗∗argvp)
- int PF_LibTerminate (int error)
- int PF_Probe (int ∗src)
- int PF_ISendSbuf (int to, int tag)
- int PF_RecvWbuf (WORD ∗b, LONG ∗s, int ∗src)
- int PF_IRecvRbuf (PF_BUFFER ∗r, int bn, int from)
- int PF_WaitRbuf (PF_BUFFER ∗r, int bn, LONG ∗size)
- int PF_Bcast (void ∗buffer, int count)
- int PF_RawSend (int dest, void ∗buf, LONG l, int tag)
- LONG PF_RawRecv (int ∗src, void ∗buf, LONG thesize, int ∗tag)
- int PF_RawProbe (int ∗src, int ∗tag, int ∗bytesize)
- int PF_PrintPackBuf (char ∗s, int size)
- int PF_PreparePack (void)
- int PF_Pack (const void ∗buffer, size_t count, MPI_Datatype type)
- int PF_Unpack (void ∗buffer, size_t count, MPI_Datatype type)
- int PF_PackString (const UBYTE ∗str)
- int PF_UnpackString (UBYTE ∗str)
- int PF_Send (int to, int tag)
- int PF_Receive (int src, int tag, int ∗psrc, int ∗ptag)
- int PF_Broadcast (void)
- int PF_PrepareLongSinglePack (void)
- int PF_LongSinglePack (const void ∗buffer, size_t count, MPI_Datatype type)
- int PF_LongSingleUnpack (void ∗buffer, size_t count, MPI_Datatype type)
- int PF_LongSingleSend (int to, int tag)
- int PF_LongSingleReceive (int src, int tag, int ∗psrc, int ∗ptag)
- int PF_PrepareLongMultiPack (void)
- int PF_LongMultiPackImpl (const void ∗buffer, size_t count, size_t eSize, MPI_Datatype type)
- int PF_LongMultiUnpackImpl (void ∗buffer, size_t count, size_t eSize, MPI_Datatype type)
- int PF_LongMultiBroadcast (void)

Variables

- LONG PF_maxDollarChunkSize = 0

4.60.1 Detailed Description

MPI dependent functions of parform

This file contains all the functions for the parallel version of form3 that explicitly need to call mpi routines. This is the only file that really needs to be linked to the mpi-library!

Definition in file mpi.c.

4.60.2 Macro Definition Documentation

4.60.2.1 PF_PACKSIZE

```
#define PF_PACKSIZE 1600
```

Definition at line 58 of file mpi.c.

4.60.2.2 MPI_ERRCODE_CHECK

```
#define MPI_ERRCODE_CHECK(
                err )
```

Value:

```
do { \
    int _tmp_err = (err); \
    if ( _tmp_err != MPI_SUCCESS ) return _tmp_err != 0 ? _tmp_err : -1; \
} while (0)
```

A macro which exits the caller with a non-zero return value if *err* is not MPI_SUCCESS.

Parameters

err	The return value of a MPI function to be checked.

Remarks

The MPI standard defines MPI_SUCCESS == 0. Then (_tmp_err == 0) appears twice and we can expect the second evaluation will be eliminated by the compiler optimization.

Definition at line 84 of file mpi.c.

4.60.3 Function Documentation

4.60.3.1 PF_RealTime()

```
LONG PF_RealTime (
            int i )
```

Returns the realtime in 1/100 sec. as a LONG.

Parameters

i	the timer will be reset if i == 0.

Returns

the real elapsed time in 1/100 second.

Definition at line 101 of file mpi.c.

Referenced by PF_Init().

4.60.3.2 PF_LibInit()

```
int PF_LibInit (
            int * argcp,
            char *** argvp )
```

Performs all library dependent initializations.

Parameters

argcp	pointer to the number of arguments.
argvp	pointer to the arguments.

Returns

0 if OK, nonzero on error.

Definition at line 123 of file mpi.c.

Referenced by PF_Init().

4.60.3.3 PF_LibTerminate()

```
int PF_LibTerminate (
            int error )
```

Exits mpi, when there is an error either indicated or happening, returnvalue is 1, else returnvalue is 0.

Parameters

error	an error code.

Returns

0 if OK, nonzero on error.

Definition at line 209 of file mpi.c.

Referenced by PF_Terminate().

4.60.3.4 PF_Probe()

```
int PF_Probe (
            int * src )
```

Probes the next incoming message. If src == PF_ANY_SOURCE this operation is blocking, otherwise nonbloking.

Parameters

in,out	*src*	the source process number. In output, the process number of actual found source.

Returns

the tag value of the next incoming message if found, 0 if a nonbloking probe (input src != PF_ANY_SOURCE) did not find any messages. A negative returned value indicates an error.

Definition at line 230 of file mpi.c.

4.60.3.5 PF_ISendSbuf()

```
int PF_ISendSbuf (
            int to,
            int tag )
```

Nonblocking send operation. It sends everything from `buff` to `fill` of the active buffer. Depending on *tag* it also can do waiting for other sends to finish or set the active buffer to the next one.

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 261 of file mpi.c.

Referenced by FlushOut(), PF_Processor(), and PutOut().

4.60.3.6 PF_RecvWbuf()

```
int PF_RecvWbuf (
            WORD * b,
            LONG * s,
            int * src )
```

Blocking receive of a `WORD` buffer.

Parameters

`out`	*b*	the buffer to store the received data.
`in,out`	*s*	the size of the buffer. The output value is the actual size of the received data.
`in,out`	*src*	the source process number. The output value is the process number of actual source.

Returns

the received message tag. A negative value indicates an error.

Definition at line 337 of file mpi.c.

4.60.3.7 PF_IRecvRbuf()

```
int PF_IRecvRbuf (
            PF_BUFFER * r,
            int bn,
            int from )
```

Posts nonblocking receive for the active receive buffer. The buffer is filled from `full` to `stop`.

Parameters

r	the PF_BUFFER struct for the nonblocking receive.
bn	the index of the cyclic buffer.
from	the source process number.

Returns

0 if OK, nonzero on error.

Definition at line 366 of file mpi.c.

4.60.3.8 PF_WaitRbuf()

```
int PF_WaitRbuf (
            PF_BUFFER * r,
            int bn,
            LONG * size )
```

Waits for the buffer *bn* to finish a pending nonblocking receive. It returns the received tag and in `*size` the number of field received. If the receive is already finished, just return the flag and size, else wait for it to finish, but also check for other pending receives.

Parameters

	r	the PF_BUFFER struct for the pending nonblocking receive.
	bn	the index of the cyclic buffer.
out	*size*	the actual size of received data.

Returns

the received message tag. A negative value indicates an error.

Definition at line 400 of file mpi.c.

4.60.3.9 PF_Bcast()

```
int PF_Bcast (
            void * buffer,
            int count )
```

Broadcasts a message from the master to slaves.

Parameters

in,out	*buffer*	the starting address of buffer. The contents in this buffer on the master will be transferred to those on the slaves.
	count	the length of the buffer in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 440 of file mpi.c.

Referenced by PF_BroadcastBuffer(), and PF_BroadcastNumber().

4.60.3.10 PF_RawSend()

```
int PF_RawSend (
            int dest,
            void * buf,
            LONG l,
            int tag )
```

Sends *l* bytes from *buf* to *dest*. Returns 0 on success, or -1.

Parameters

dest	the destination process number.
buf	the send buffer.
l	the size of data in the send buffer in bytes.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 463 of file mpi.c.

Referenced by PF_MUnlock(), and PF_SendFile().

4.60.3.11 PF_RawRecv()

```
LONG PF_RawRecv (
            int * src,
            void * buf,
            LONG thesize,
            int * tag )
```

Receives not more than *thesize* bytes from *src*, returns the actual number of received bytes, or -1 on failure.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
out	*buf*	the receive buffer.
	thesize	the size of the receive buffer in bytes.
out	*tag*	the message tag of the actual received message.

Returns

the actual sizeof received data in bytes, or -1 on failure.

Definition at line 484 of file mpi.c.

Referenced by PF_RecvFile().

4.60.3.12 PF_RawProbe()

```
int PF_RawProbe (
            int * src,
            int * tag,
            int * bytesize )
```

Probes an incoming message.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
in,out	*tag*	the message tag. In output, that of the actual received message.
out	*bytesize*	the size of incoming data in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 508 of file mpi.c.

4.60.3.13 PF_PrintPackBuf()

```
int PF_PrintPackBuf (
            char * s,
            int size )
```

Prints the contents in the pack buffer.

Parameters

s	a message to be shown.
size	the length of the buffer to be shown.

Returns

0 if OK, nonzero on error.

Definition at line 594 of file mpi.c.

4.60.3.14 PF_PreparePack()

```
int PF_PreparePack (
            void  )
```

Prepares for the next pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 624 of file mpi.c.

Referenced by PF_BroadcastPreDollar(), PF_BroadcastString(), and PF_Init().

4.60.3.15 PF_Pack()

```
int PF_Pack (
            const void * buffer,
            size_t count,
            MPI_Datatype type )
```

Adds data into the pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 642 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by PF_BroadcastPreDollar(), and PF_Init().

4.60.3.16 PF_Unpack()

```
int PF_Unpack (
            void * buffer,
            size_t count,
            MPI_Datatype type )
```

Retrieves the next data in the pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 671 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by PF_BroadcastPreDollar(), and PF_Init().

4.60.3.17 PF_PackString()

```
int PF_PackString (
            const UBYTE * str )
```

Packs a string *str* into the packed buffer PF_packbuf, including the trailing zero.

The first element (PF_INT) is the length of the packed portion of the string. If the string does not fit to the buffer PF↩_packbuf, the function packs only the initial portion. It returns the number of packed bytes, so if (str[length-1]=='\0') then the whole string fits to the buffer, if not, then the rest (str+length) bust be packed and send again. On error, the function returns the negative error code.

One exception: the string "\0!\0" is used as an image of the NULL, so all 3 characters will be packed.

Parameters

str	a string to be packed.

Returns

the number of packed bytes, or a negative value on failure.

Definition at line 706 of file mpi.c.

Referenced by PF_BroadcastString().

4.60.3.18 PF_UnpackString()

```
int PF_UnpackString (
            UBYTE * str )
```

Unpacks a string to *str* from the packed buffer PF_packbuf, including the trailing zero.

It returns the number of unpacked bytes, so if (str[length-1]=='\0') then the whole string was unpacked, if not, then the rest must be appended to (str+length). On error, the function returns the negative error code.

Parameters

out	*str*	the buffer to store the unpacked string

Returns

the number of unpacked bytes, or a negative value on failure.

Definition at line 774 of file mpi.c.

Referenced by PF_BroadcastString().

4.60.3.19 PF_Send()

```
int PF_Send (
            int to,
            int tag )
```

Sends the contents in the pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
  PF_PreparePack();
  // Packing operations here...
  PF_Send(DEST, TAG);
}
else if ( PF.me == DEST ) {
  PF_Receive(SRC, TAG, &actual_src, &actual_tag);
  // Unpacking operations here...
}
```

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 822 of file mpi.c.

References MPI_ERRCODE_CHECK.

4.60.3.20 PF_Receive()

```
int PF_Receive (
            int src,
            int tag,
            int * psrc,
            int * ptag )
```

Receives data into the pack buffer from the process specified by *src*. This function allows &src == psrc or &tag == ptag. Either *psrc* or *ptag* can be NULL.

See the example of PF_Send().

Parameters

	src	the source process number (can be PF_ANY_SOURCE).
	tag	the source message tag (can be PF_ANY_TAG).
out	*psrc*	the actual source process number of received message.
out	*ptag*	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 848 of file mpi.c.

References MPI_ERRCODE_CHECK.

4.60.3.21 PF_Broadcast()

```
int PF_Broadcast (
            void  )
```

Broadcasts the contents in the pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
  PF_PreparePack();
  // Packing operations here...
}
PF_Broadcast();
if ( PF.me != MASTER ) {
  // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 883 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by PF_BroadcastPreDollar(), PF_BroadcastString(), and PF_Init().

4.60.3.22 PF_PrepareLongSinglePack()

```
int PF_PrepareLongSinglePack (
            void  )
```

Prepares for the next long-single-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1451 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.60.3.23 PF_LongSinglePack()

```
int PF_LongSinglePack (
            const void * buffer,
            size_t count,
            MPI_Datatype type )
```

Adds data into the "long single" pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1469 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.60.3.24 PF_LongSingleUnpack()

```
int PF_LongSingleUnpack (
            void * buffer,
            size_t count,
            MPI_Datatype type )
```

Retrieves the next data in the "long single" pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1503 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.60.3.25 PF_LongSingleSend()

```
int PF_LongSingleSend (
            int to,
            int tag )
```

Sends the contents in the "long single" pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
  PF_PrepareLongSinglePack();
  // Packing operations here...
  PF_LongSingleSend(DEST, TAG);
}
else if ( PF.me == DEST ) {
  PF_LongSingleReceive(SRC, TAG, &actual_src, &actual_tag);
  // Unpacking operations here...
}
```

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1540 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.60.3.26 PF_LongSingleReceive()

```
int PF_LongSingleReceive (
            int src,
            int tag,
            int * psrc,
            int * ptag )
```

Receives data into the "long single" pack buffer from the process specified by *src*. This function allows &src == psrc or &tag == ptag. Either *psrc* or *ptag* can be NULL.

See the example of PF_LongSingleSend().

Parameters

	src	the source process number (can be PF_ANY_SOURCE).
	tag	the source message tag (can be PF_ANY_TAG).
out	*psrc*	the actual source process number of received message.
out	*ptag*	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1583 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.60.3.27 PF_PrepareLongMultiPack()

```
int PF_PrepareLongMultiPack (
            void  )
```

Prepares for the next long-multi-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1643 of file mpi.c.

Referenced by PF_BroadcastCBuf(), PF_BroadcastExpFlags(), PF_BroadcastModifiedDollars(), and PF_BroadcastRedefinedPreVars

4.60.3.28 PF_LongMultiPackImpl()

```
int PF_LongMultiPackImpl (
            const void * buffer,
            size_t count,
            size_t eSize,
            MPI_Datatype type )
```

Adds data into the "long multi" pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
eSize	the byte size of each element of data.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1662 of file mpi.c.

References PF_LongMultiPackImpl().

Referenced by PF_LongMultiPackImpl().

Here is the call graph for this function:

4.60.3.29 PF_LongMultiUnpackImpl()

```
int PF_LongMultiUnpackImpl (
            void * buffer,
            size_t count,
            size_t eSize,
            MPI_Datatype type )
```

Retrieves the next data in the "long multi" pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	eSize	the byte size of each element of data.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1721 of file mpi.c.

References PF_LongMultiUnpackImpl().

Referenced by PF_LongMultiUnpackImpl().

Here is the call graph for this function:

4.60.3.30 PF_LongMultiBroadcast()

```
int PF_LongMultiBroadcast (
            void  )
```

Broadcasts the contents in the "long multi" pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
  PF_PrepareLongMultiPack();
  // Packing operations here...
}
PF_LongMultiBroadcast();
if ( PF.me != MASTER ) {
  // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 1807 of file mpi.c.

Referenced by PF_BroadcastCBuf(), PF_BroadcastExpFlags(), PF_BroadcastModifiedDollars(), and PF_BroadcastRedefinedPreVars

4.60.4 Variable Documentation

4.60.4.1 PF_maxDollarChunkSize

```
LONG PF_maxDollarChunkSize = 0
```

Definition at line 69 of file mpi.c.

4.61 mpi.c

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035 /*
00036     #[ Includes and variables :
00037 */
00038
00039 #include <limits.h>
00040 #include "form3.h"
00041
00042 #ifdef MPICH_PROFILING
00043 # include "mpe.h"
00044 #endif
00045
00046 #ifdef MPIDEBUGGING
00047 #include "mpidbg.h"
00048 #endif
00049
00050 /*[12oct2005 mt]:*/
00051 /*
00052     Today there was some cleanup, some stuff is moved into another place
00053     in this file, and PF.packsize is removed and PF_packsize is used
00054     instead. It is rather difficult to proper comment it, so not all these
00055     changing are marked by "[12oct2005 mt]"
00056 */
00057
00058 #define PF_PACKSIZE 1600
00059
00060 /*
00061     Size in bytes, will be initialized soon as
00062     PF_packsize=PF_PACKSIZE/sizeof(int)*sizeof(int); for possible
00063     future developing we prefer to do this initialization not here,
00064     but in PF_LibInit:
00065 */
00066
00067 static int PF_packsize = 0;
00068 static MPI_Status PF_status;
00069 LONG PF_maxDollarChunkSize = 0;      /*:[04oct2005 mt]*/
00070
00071 static int PF_ShortPackInit(void);
00072 static int PF_longPackInit(void);      /*:[12oct2005 mt]*/
00073
```

```
#define MPI_ERRCODE_CHECK(err) \
    do { \
        int _tmp_err = (err); \
        if ( _tmp_err != MPI_SUCCESS ) return _tmp_err != 0 ? _tmp_err : -1; \
    } while (0)

/*
    #] Includes and variables :
    #[ PF_RealTime :
*/

LONG PF_RealTime(int i)
{
    static double starttime;
    if ( i == PF_RESET ) {
        starttime = MPI_Wtime();
        return((LONG)0);
    }
    return((LONG)( 100. * (MPI_Wtime() - starttime) ) );
}

/*
    #] PF_RealTime :
    #[ PF_LibInit :
*/

int PF_LibInit(int *argcp, char ***argvp)
{
    int ret;
    ret = MPI_Init(argcp,argvp);
    if ( ret != MPI_SUCCESS ) return(ret);
    ret = MPI_Comm_rank(PF_COMM,&PF.me);
    if ( ret != MPI_SUCCESS ) return(ret);
    ret = MPI_Comm_size(PF_COMM,&PF.numtasks);
    if ( ret != MPI_SUCCESS ) return(ret);

    /* Initialization of packed communications. */
    PF_packsize = PF_PACKSIZE/sizeof(int)*sizeof(int);
    if ( PF_ShortPackInit() ) return -1;
    if ( PF_longPackInit() ) return -1;

    {/*Block*/
        int bytes, totalbytes=0;
/*
            There is one problem with maximal possible packing: there is no API to
            convert bytes to the record number. So, here we calculate the buffer
            size needed for storing dollarvars:

            LONG PF_maxDollarChunkSize is the size for the portion of the dollar
            variable buffer suitable for broadcasting. This variable should be
            visible from parallel.c

            Evaluate PF_Pack(numterms,1,PF_INT):
*/
        if ( ( ret = MPI_Pack_size(1,PF_INT,PF_COMM,&bytes) )!=MPI_SUCCESS )
            return(ret);

        totalbytes+=bytes;
/*
            Evaluate PF_Pack( newsize,1,PF_LONG):
*/
        if ( ( ret = MPI_Pack_size(1,PF_LONG,PF_COMM,&bytes) )!=MPI_SUCCESS )
            return(ret);

        totalbytes += bytes;
/*
            Now available room is PF_packsize-totalbytes
*/
        totalbytes = PF_packsize-totalbytes;
/*
            Now totalbytes is the size of chunk in bytes.
            Evaluate this size in number of records:

            Rough estimate:
*/
        PF_maxDollarChunkSize=totalbytes/sizeof(WORD);
/*
            Go to the up limit:
*/
        do {
            if ( ( ret = MPI_Pack_size(
                        ++PF_maxDollarChunkSize,PF_WORD,PF_COMM,&bytes) )!=MPI_SUCCESS )
                return(ret);
        } while ( bytes<totalbytes );
/*
            Now the chunk size is too large
            And now evaluate the exact value:
```

```
*/
        do {
            if ( ( ret = MPI_Pack_size(
                        --PF_maxDollarChunkSize,PF_WORD,PF_COMM,&bytes) )!=MPI_SUCCESS )
                return(ret);
        } while ( bytes>totalbytes );
/*
            Now PF_maxDollarChunkSize is the size of chunk of PF_WORD fitting the
            buffer <= (PF_packsize-PF_INT-PF_LONG)
*/
    }/*Block*/
    return(0);
}
/*
    #] PF_LibInit :
    #[ PF_LibTerminate :
*/

int PF_LibTerminate(int error)
{
    DUMMYUSE(error);
    return(MPI_Finalize());
}

/*
    #] PF_LibTerminate :
    #[ PF_Probe :
*/

int PF_Probe(int *src)
{
    int ret, flag;
    if ( *src == PF_ANY_SOURCE ) { /*Blocking call*/
        ret = MPI_Probe(*src,MPI_ANY_TAG,PF_COMM,&PF_status);
        flag = 1;
    }
    else { /*Non-blocking call*/
        ret = MPI_Iprobe(*src,MPI_ANY_TAG,PF_COMM,&flag,&PF_status);
    }
    *src = PF_status.MPI_SOURCE;
    if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
    if ( !flag ) return(0);
    return(PF_status.MPI_TAG);
}

/*
    #] PF_Probe :
    #[ PF_ISendSbuf :
*/

int PF_ISendSbuf(int to, int tag)
{
    PF_BUFFER *s = PF.sbuf;
    int a = s->active;
    int size = s->fill[a] - s->buff[a];
    int r = 0;

    static int finished;

    s->fill[a] = s->buff[a];
    if ( s->numbufs == 1 ) {
        r = MPI_Ssend(s->buff[a],size,PF_WORD,MASTER,tag,PF_COMM);
        if ( r != MPI_SUCCESS ) {
            fprintf(stderr,"[%d|%d] PF_ISendSbuf: MPI_Ssend returns: %d \n",
                    PF.me,(int)AC.CModule,r);
            fflush(stderr);
            return(r);
        }
        return(0);
    }

    switch ( tag ) { /* things to do before sending */
        case PF_TERM_MSGTAG:
            if ( PF.sbuf->request[to] != MPI_REQUEST_NULL)
                r = MPI_Wait(&PF.sbuf->request[to],&PF.sbuf->retstat[to]);
            if ( r != MPI_SUCCESS ) return(r);
            break;
        default:
            break;
    }

    r = MPI_Isend(s->buff[a],size,PF_WORD,to,tag,PF_COMM,&s->request[a]);

    if ( r != MPI_SUCCESS ) return(r);

    switch ( tag ) { /* things to do after initialising sending */
        case PF_TERM_MSGTAG:
```

```
            finished = 0;
            break;
        case PF_ENDSORT_MSGTAG:
            if ( ++finished == PF.numtasks - 1 )
                r = MPI_Waitall(s->numbufs,s->request,s->status);
            if ( r != MPI_SUCCESS ) return(r);
            break;
        case PF_BUFFER_MSGTAG:
            if ( ++s->active >= s->numbufs ) s->active = 0;
            while ( s->request[s->active] != MPI_REQUEST_NULL ) {
                r = MPI_Waitsome(s->numbufs,s->request,&size,s->index,s->retstat);
                if ( r != MPI_SUCCESS ) return(r);
            }
            break;
        case PF_ENDBUFFER_MSGTAG:
            if ( ++s->active >= s->numbufs ) s->active = 0;
            r = MPI_Waitall(s->numbufs,s->request,s->status);
            if ( r != MPI_SUCCESS ) return(r);
            break;
        default:
            return(-99);
            break;
    }
    return(0);
}

/*
    #] PF_ISendSbuf :
    #[ PF_RecvWbuf :
*/

int PF_RecvWbuf(WORD *b, LONG *s, int *src)
{
    int i, r = 0;

    r = MPI_Recv(b,(int)*s,PF_WORD,*src,PF_ANY_MSGTAG,PF_COMM,&PF_status);
    if ( r != MPI_SUCCESS ) { if ( r > 0 ) r *= -1; return(r); }

    r = MPI_Get_count(&PF_status,PF_WORD,&i);
    if ( r != MPI_SUCCESS ) { if ( r > 0 ) r *= -1; return(r); }

    *s = (LONG)i;
    *src = PF_status.MPI_SOURCE;
    return(PF_status.MPI_TAG);
}

/*
    #] PF_RecvWbuf :
    #[ PF_IRecvRbuf :
*/

int PF_IRecvRbuf(PF_BUFFER *r, int bn, int from)
{
    int ret;
    r->type[bn] = PF_WORD;

    if ( r->numbufs == 1 ) {
        r->tag[bn] = MPI_ANY_TAG;
        r->from[bn] = from;
    }
    else {
        ret = MPI_Irecv(r->full[bn],(int)(r->stop[bn] - r->full[bn]),PF_WORD,from,
                        MPI_ANY_TAG,PF_COMM,&r->request[bn]);
        if (ret != MPI_SUCCESS) { if(ret > 0) ret *= -1; return(ret); }
    }
    return(0);
}

/*
    #] PF_IRecvRbuf :
    #[ PF_WaitRbuf :
*/

int PF_WaitRbuf(PF_BUFFER *r, int bn, LONG *size)
{
    int ret, rsize;

    if ( r->numbufs == 1 ) {
        *size = r->stop[bn] - r->full[bn];
        ret = MPI_Recv(r->full[bn],(int)*size,r->type[bn],r->from[bn],r->tag[bn],
                       PF_COMM,&(r->status[bn]));
        if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
        ret = MPI_Get_count(&(r->status[bn]),r->type[bn],&rsize);
        if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
        if ( rsize > *size ) return(-99);
        *size = (LONG)rsize;
    }
```

```
    else {
        while ( r->request[bn] != MPI_REQUEST_NULL ) {
            ret = MPI_Waitsome(r->numbufs,r->request,&rsize,r->index,r->retstat);
            if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
            while ( --rsize >= 0 ) r->status[r->index[rsize]] = r->retstat[rsize];
        }
        ret = MPI_Get_count(&(r->status[bn]),r->type[bn],&rsize);
        if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
        *size = (LONG)rsize;
    }
    return(r->status[bn].MPI_TAG);
}

/*
    #] PF_WaitRbuf :
    #[ PF_Bcast :
*/

int PF_Bcast(void *buffer, int count)
{
    if ( MPI_Bcast(buffer,count,MPI_BYTE,MASTER,PF_COMM) != MPI_SUCCESS )
        return(-1);
    return(0);
}

/*
    #] PF_Bcast :
    #[ PF_RawSend :
*/

int PF_RawSend(int dest, void *buf, LONG l, int tag)
{
    int ret=MPI_Ssend(buf,(int)l,MPI_BYTE,dest,tag,PF_COMM);
    if ( ret != MPI_SUCCESS ) return(-1);
    return(0);
}
/*
    #] PF_RawSend :
    #[ PF_RawRecv :
*/

LONG PF_RawRecv(int *src,void *buf,LONG thesize,int *tag)
{
    MPI_Status stat;
    int ret=MPI_Recv(buf,(int)thesize,MPI_BYTE,*src,MPI_ANY_TAG,PF_COMM,&stat);
    if ( ret != MPI_SUCCESS ) return(-1);
    if ( MPI_Get_count(&stat,MPI_BYTE,&ret) != MPI_SUCCESS ) return(-1);
    *tag = stat.MPI_TAG;
    *src = stat.MPI_SOURCE;
    return(ret);
}

/*
    #] PF_RawRecv :
    #[ PF_RawProbe :
*/

int PF_RawProbe(int *src, int *tag, int *bytesize)
{
    MPI_Status stat;
    int srcval = src != NULL ? *src : PF_ANY_SOURCE;
    int tagval = tag != NULL ? *tag : PF_ANY_MSGTAG;
    int ret = MPI_Probe(srcval, tagval, PF_COMM, &stat);
    if ( ret != MPI_SUCCESS ) return -1;
    if ( src != NULL ) *src = stat.MPI_SOURCE;
    if ( tag != NULL ) *tag = stat.MPI_TAG;
    if ( bytesize != NULL ) {
        ret = MPI_Get_count(&stat, MPI_BYTE, bytesize);
        if ( ret != MPI_SUCCESS ) return -1;
    }
    return 0;
}

/*
    #] PF_RawProbe :
    #[ The pack buffer :
        #[ Variables :
*/

/*
 * The pack buffer with the fixed size (= PF_packsize).
 */
static UBYTE *PF_packbuf  = NULL;
static UBYTE *PF_packstop = NULL;
static int    PF_packpos  = 0;

/*
```

```
        #] Variables :
        #[ PF_ShortPackInit :
*/

static int PF_ShortPackInit(void)
{
    PF_packbuf = (UBYTE *)Malloc1(sizeof(UBYTE) * PF_packsize, "PF_ShortPackInit");
    if ( PF_packbuf == NULL ) return -1;
    PF_packstop = PF_packbuf + PF_packsize;
    return 0;
}

/*
        #] PF_ShortPackInit :
        #[ PF_InitPackBuf :
*/

static inline int PF_InitPackBuf(void)
{
/*
        This is definitely not the best place for allocating the
        buffer! Moved to PF_LibInit():

    if ( PF_packbuf == 0 ) {
        PF_packbuf = (UBYTE *)Malloc1(sizeof(UBYTE)*PF.packsize,"PF_InitPackBuf");
        if ( PF_packbuf == 0 ) return(-1);
        PF_packstop = PF_packbuf + PF.packsize;
    }
*/
    PF_packpos = 0;
    return(0);
}

/*
        #] PF_InitPackBuf :
        #[ PF_PrintPackBuf :
*/

int PF_PrintPackBuf(char *s, int size)
{
#ifdef NOMESPRINTYET
/*
        The use of printf should be discouraged. The results are flushed to
        the output at unpredictable moments. We should use printf only
        during startup when MesPrint doesn't have its buffers and output
        channels initialized.
*/
    int i;
    printf("[%d] %s: ",PF.me,s);
    for(i=0;i<size;i++) printf("%d ",PF_packbuf[i]);
    printf("\n");
#else
    MesPrint("[%d] %s: %a",PF.me,s,size,(WORD *)(PF_packbuf));
#endif
    return(0);
}

/*
        #] PF_PrintPackBuf :
        #[ PF_PreparePack :
*/

int PF_PreparePack(void)
{
    return PF_InitPackBuf();
}

/*
        #] PF_PreparePack :
        #[ PF_Pack :
*/

int PF_Pack(const void *buffer, size_t count, MPI_Datatype type)
{
    int err, bytes;

    if ( count > INT_MAX ) return -99;

    err = MPI_Pack_size((int)count, type, PF_COMM, &bytes);
    MPI_ERRCODE_CHECK(err);
    if ( PF_packpos + bytes > PF_packstop - PF_packbuf ) return -99;

    err = MPI_Pack((void *)buffer, (int)count, type, PF_packbuf, PF_packsize, &PF_packpos, PF_COMM);
    MPI_ERRCODE_CHECK(err);

    return 0;
}
```

```
/*
        #] PF_Pack :
        #[ PF_Unpack :
*/

int PF_Unpack(void *buffer, size_t count, MPI_Datatype type)
{
    int err;

    if ( count > INT_MAX ) return -99;

    err = MPI_Unpack(PF_packbuf, PF_packsize, &PF_packpos, buffer, (int)count, type, PF_COMM);
    MPI_ERRCODE_CHECK(err);

    return 0;
}

/*
        #] PF_Unpack :
        #[ PF_PackString :
*/

int PF_PackString(const UBYTE *str)
{
    int ret,buflength,bytes,length;
/*
        length will be packed in the beginning.
        Decrement buffer size by the length of the field "length":
*/
    if ( ( ret = MPI_Pack_size(1,PF_INT,PF_COMM,&bytes) ) != MPI_SUCCESS )
        return(ret);
    buflength = PF_packsize - bytes;
/*
        Calculate the string length (INCLUDING the trailing zero!):
*/
    for ( length = 0; length < buflength; length++ ) {
        if ( str[length] == '\0' ) {
            length++; /* since the trailing zero must be accounted */
            break;
        }
    }
/*
        The string "\0!\0" is used as an image of the NULL.
*/
    if ( ( str[0] == '\0' ) /* empty string */
      && ( str[1] == '!' )  /* Special case? */
      && ( str[2] == '\0' ) /* Yes, pass 3 initial symbols */
            ) length += 2;  /* all 3 characters will be packed */
    length++;               /* Will be decremented in the following loop */
/*
        The problem: packed size of byte may be not equal 1! So first, suppose
        it is 1, and if this is not the case decrease the length of the string
        until it fits the buffer:
*/
    do {
        if ( ( ret = MPI_Pack_size(--length,PF_BYTE,PF_COMM,&bytes) )
            != MPI_SUCCESS ) return(ret);
    } while ( bytes > buflength );
/*
        Note, now if str[length-1] == '\0' then the string fits to the buffer
        (INCLUDING the trailing zero!);if not, the rest must be packed further!

        Pack the length to PF_packbuf:
*/
    if ( ( ret = MPI_Pack(&length,1,PF_INT,PF_packbuf,PF_packsize,
            &PF_packpos,PF_COMM) ) != MPI_SUCCESS ) return(ret);
/*
        Pack the string to PF_packbuf:
*/
    if ( ( ret = MPI_Pack((UBYTE *)str,length,PF_BYTE,PF_packbuf,PF_packsize,
            &PF_packpos,PF_COMM) ) != MPI_SUCCESS ) return(ret);
    return(length);
}

/*
        #] PF_PackString :
        #[ PF_UnpackString :
*/

int PF_UnpackString(UBYTE *str)
{
    int ret,length;
/*
        Unpack the length:
*/
    if(  (ret = MPI_Unpack(PF_packbuf,PF_packsize,&PF_packpos,
```

```
            &length,1,PF_INT,PF_COMM))!= MPI_SUCCESS )
                return(ret);
/*
        Unpack the string:
*/
    if ( ( ret = MPI_Unpack(PF_packbuf,PF_packsize,&PF_packpos,
            str,length,PF_BYTE,PF_COMM) ) != MPI_SUCCESS ) return(ret);
/*
        Now if str[length-1]=='\0' then the whole string
        (INCLUDING the trailing zero!) was unpacked ;if not, the rest
        must be unpacked to str+length.
*/
    return(length);
}

/*
        #] PF_UnpackString :
        #[ PF_Send :
*/

int PF_Send(int to, int tag)
{
    int err;
    err = MPI_Ssend(PF_packbuf, PF_packpos, MPI_PACKED, to, tag, PF_COMM);
    MPI_ERRCODE_CHECK(err);
    return 0;
}

/*
        #] PF_Send :
        #[ PF_Receive :
*/

int PF_Receive(int src, int tag, int *psrc, int *ptag)
{
    int err;
    MPI_Status status;
    PF_InitPackBuf();
    err = MPI_Recv(PF_packbuf, PF_packsize, MPI_PACKED, src, tag, PF_COMM, &status);
    MPI_ERRCODE_CHECK(err);
    if ( psrc ) *psrc = status.MPI_SOURCE;
    if ( ptag ) *ptag = status.MPI_TAG;
    return 0;
}

/*
        #] PF_Receive :
        #[ PF_Broadcast :
*/

int PF_Broadcast(void)
{
    int err;
/*
 * If PF_SHORTBROADCAST is defined, then the broadcasting will be performed in
 * 2 steps. First, the size of the buffer will be broadcast, then the buffer of
 * exactly used size. This should be faster with slow connections, but slower on
 * SMP shmem MPI because of the latency.
 */
#ifdef PF_SHORTBROADCAST
    int pos = PF_packpos;
#endif
    if ( PF.me != MASTER ) {
        err = PF_InitPackBuf();
        if ( err ) return err;
    }
#ifdef PF_SHORTBROADCAST
    err = MPI_Bcast(&pos, 1, MPI_INT, MASTER, PF_COMM);
    MPI_ERRCODE_CHECK(err);
    err = MPI_Bcast(PF_packbuf, pos, MPI_PACKED, MASTER, PF_COMM);
#else
    err = MPI_Bcast(PF_packbuf, PF_packsize, MPI_PACKED, MASTER, PF_COMM);
#endif
    MPI_ERRCODE_CHECK(err);
    return 0;
}

/*
        #] PF_Broadcast :
    #] The pack buffer :
    #[ Long pack stuff :
        #[ Explanations :

    The problems here are:
        1. We need to send/receive long dollar variables. For
    preprocessor-defined dollarvars we used multiply
    packing/broadcasting  (see parallel.c:PF_BroadcastPreDollar())
```

```
since each variable must be broadcast immediately. For run-time
the changed dollar variables, collecting and broadcasting are
performed at the end of the module and all modified dollarvars
are transferred "at once", that is why the size of packed and
transferred buffers may be really very large.
    2. There is some strange feature of MPI_Bcast() on Altix MPI
implementation, namely, sometimes it silently fails with big
buffers. For better performance, it would be useful to send one
big buffer instead of several small ones (since the latency is more
important than the bandwidth). That is why we need two different
sets of routines: for long point-to-point communication we collect
big re-allocatable buffer, the corresponding routines have the
prefix PF_longSingle, and for broadcasting we pack data into
several smaller buffers, the corresponding routines have the
prefix PF_longMulti.
    Note, from portability reasons we cannot split large packed
buffer into small chunks, send them and collect back on the other
side, see "Advice to users" on page 180 MPI--The Complete Reference
Volume1, second edition.
    OPTIMIZING:
    We assume, for most communications, the single buffer of size
PF_packsize is enough.

    How does it work:
    For point-to-point, there is one big re-allocatable
buffer PF_longPackBuf with two integer positions: PF_longPackPos
and PF_longPackTop (due to re-allocatable character of the buffer,
it is better to use integers rather than pointers).
    Each time of re-allocation, the size of the buffer
PF_longPackBuf is incremented by the same size of a "standard" chunk
PF_packsize.
    For broadcasting there is one linked list (PF_longMultiRoot),
which contains either positions of a chunk of PF_longPackBuf, or
it's own buffer. This is done for better memory utilisation:
longSingle and longMulti are never used simultaneously.
    When a new cell is needed for LongMulti packing, we increment
the counter PF_longPackN and just follow the list. If it is not
possible, we allocate the cell's own buffer and link it to the end
of the list PF_longMultiRoot.
    When PF_longPackPos is reallocated, we link new chunks into
existing PF_longMultiRoot list before the first longMulti allocated
cell's own buffer. The pointer PF_longMultiLastChunk points to the last
cell of PF_longMultiRoot containing the pointer to the chunk of
PF_longPackBuf.
    Initialization PF_longPackBuf is made by the function
PF_longSingleReset(). In the begin of the PF_longPackBuf it packs
the size of the last sent buffer. Upon sending, the program checks,
whether there was at list one re-allocation (PF_longPackN>1) .
If so, the sender first packs and sends small buffer
(PF_longPackSmallBuf) containing one integer number -- the
_negative_ new size of the send buffer. Getting the buffer, a
receiver unpacks one integer and checks whether it is <0 . If so,
the receiver will repeat receiving, but first it checks whether
it has enough buffer and increase it, if necessary.
    Initialization PF_longMultiRoot is made by the function
PF_longMultiReset(). In the begin of the first chunk it packs
one integer -- the number 1. Upon sending, the program checks,
how many cells were packed (PF_longPackN). If more than 1, the
sender packs to the next cell the integer PF_longPackN, than
packs PF_longPackN pairs of integers -- the information about how many
times chunk on each cell was accessed by the packing procedure,
this information is contained by the nPacks field of the cell
structure, and how many non-complete items was at the end of this
chunk the structure field lastLen. Then the sender sends first
this auxiliary chunk.
The receiver unpacks the integer from obtained chunk and, if this
integer is more than 1, it gets more chunks, unpacking information
from the first auxiliary chunk into the corresponding nPacks
fields. Unpacking information from multiple chunks, the receiver
knows, when the chunk is expired and it must switch to the next cell,
successively decrementing corresponding nPacks field.

XXX: There are still some flaws:
PF_LongSingleSend/PF_LongSingleReceive may fail, for example, for data
transfers from the master to many slaves. Suppose that the master sends big
data to slaves, which needs an increase of the buffer of the receivers. For
the first data transfer, the master sends the new buffer size as the first
message, and then sends the data as the second message, because
PF_LongSinglePack records the increase of the buffer size on the master. For
the next time, however, the master sends the data without sending the new
buffer size, and then MPI_Recv fails due to the data overflow.
In parallel.c, they are used for the communication from slaves to the
master. In this case, this problem does not occur because the master always
has enough buffer.
The maximum size that PF_LongMultiBroadcast can broadcast is limited to
around 320kB because the current implementation tries to pack all
information of chained buffers into one buffer, whose size is PF_packsize
```

```
    = 1600B.

        #] Explanations :
        #[ Variables :
*/

typedef struct longMultiStruct {
    UBYTE *buffer; /* NULL if */
    int bufpos;    /* if >=0, PF_longPackBuf+bufpos is the chunk start */
    int packpos;   /* the current position */
    int nPacks;    /* How many times PF_longPack operates on this cell */
    int lastLen;   /* if > 0, the last packing didn't fit completely to this
                      chunk, only lastLen items was packed, the rest is in
                      the next cell. */
    struct longMultiStruct *next;  /* next linked cell, or NULL */
} PF_LONGMULTI;

static UBYTE *PF_longPackBuf = NULL;
static VOID *PF_longPackSmallBuf = NULL;
static int PF_longPackPos = 0;
static int PF_longPackTop = 0;
static PF_LONGMULTI *PF_longMultiRoot = NULL;
static PF_LONGMULTI *PF_longMultiTop = NULL;
static PF_LONGMULTI *PF_longMultiLastChunk = NULL;
static int PF_longPackN = 0;

/*
        #] Variables :
        #[ Long pack private functions :
        #[ PF_longMultiNewCell :
*/

static inline int PF_longMultiNewCell(void)
{
/*
        Allocate a new cell:
*/
    PF_longMultiTop->next = (PF_LONGMULTI *)
            Malloc1(sizeof(PF_LONGMULTI),"PF_longMultiCell");
    if ( PF_longMultiTop->next == NULL ) return(-1);
/*
        Allocate a private buffer:
*/
    PF_longMultiTop->next->buffer=(UBYTE*)
            Malloc1(sizeof(UBYTE)*PF_packsize,"PF_longMultiChunk");
    if ( PF_longMultiTop->next->buffer == NULL ) return(-1);
/*
        For the private buffer position is -1:
*/
    PF_longMultiTop->next->bufpos = -1;
/*
        This is the last cell in the chain:
*/
    PF_longMultiTop->next->next = NULL;
/*
        packpos and nPacks are not initialized!
*/
    return(0);
}

/*
        #] PF_longMultiNewCell :
        #[ PF_longMultiPack2NextCell :
*/
static inline int PF_longMultiPack2NextCell(void)
{
/*
        Is there a free cell in the chain?
*/
    if ( PF_longMultiTop->next == NULL ) {
/*
            No, allocate the new cell with a private buffer:
*/
        if ( PF_longMultiNewCell() ) return(-1);
    }
/*
        Move to the next cell in the chain:
*/
    PF_longMultiTop = PF_longMultiTop->next;
/*
        if >=0, the cell buffer is the chunk of PF_longPackBuf, initialize it:
*/
    if ( PF_longMultiTop->bufpos > -1 )
        PF_longMultiTop->buffer = PF_longPackBuf+PF_longMultiTop->bufpos;
/*
    else -- the cell has it's own private buffer.
    Initialize the cell fields:
```

```
*/
    PF_longMultiTop->nPacks  = 0;
    PF_longMultiTop->lastLen = 0;
    PF_longMultiTop->packpos = 0;
    return(0);
}

/*
        #] PF_longMultiPack2NextCell :
        #[ PF_longMultiNewChunkAdded :
*/

static inline int PF_longMultiNewChunkAdded(int n)
{
/*
        Store the list tail:
*/
    PF_LONGMULTI *MemCell = PF_longMultiLastChunk->next;
    int pos = PF_longPackTop;

    while ( n-- > 0 ) {
/*
            Allocate a new cell:
*/
        PF_longMultiLastChunk->next = (PF_LONGMULTI *)
                Malloc1(sizeof(PF_LONGMULTI),"PF_longMultiCell");
        if ( PF_longMultiLastChunk->next == NULL ) return(-1);
/*
            Update the Last Chunk Pointer:
*/
        PF_longMultiLastChunk = PF_longMultiLastChunk->next;
/*
            Initialize the new cell:
*/
        PF_longMultiLastChunk->bufpos = pos;
        pos += PF_packsize;
        PF_longMultiLastChunk->buffer = NULL;
        PF_longMultiLastChunk->packpos = 0;
        PF_longMultiLastChunk->nPacks  = 0;
        PF_longMultiLastChunk->lastLen = 0;
    }
/*
        Hitch the tail:
*/
    PF_longMultiLastChunk->next = MemCell;
    return(0);
}

/*
        #] PF_longMultiNewChunkAdded :
        #[ PF_longCopyChunk :
*/

static inline void PF_longCopyChunk(int *to, int *from, int n)
{
    NCOPYI(to,from,n)
/*  for ( ; n > 0; n-- ) *to++ = *from++; */
}

/*
        #] PF_longCopyChunk :
        #[ PF_longAddChunk :

    The chunk must be increased by n*PF_packsize.
*/

static int PF_longAddChunk(int n, int mustRealloc)
{
    UBYTE *newbuf;
    if ( ( newbuf = (UBYTE*)Malloc1(sizeof(UBYTE)*(PF_longPackTop+n*PF_packsize),
                "PF_longPackBuf") ) == NULL ) return(-1);
/*
        Allocate and chain a new cell for longMulti:
*/
    if ( PF_longMultiNewChunkAdded(n) ) return(-1);
/*
        Copy the content to the new buffer:
*/
    if ( mustRealloc ) {
        PF_longCopyChunk((int*)newbuf,(int*)PF_longPackBuf,PF_longPackTop/sizeof(int));
    }
/*
        Note, PF_packsize is multiple by sizeof(int) by construction!
*/
    PF_longPackTop += (n*PF_packsize);
/*
        Free the old buffer and store the new one:
```

```
*/
    M_free(PF_longPackBuf,"PF_longPackBuf");
    PF_longPackBuf = newbuf;
/*
        Count number of re-allocs:
*/
    PF_longPackN += n;
    return(0);
}

/*
        #] PF_longAddChunk :
        #[ PF_longMultiHowSplit :

    "count" of "type" elements in an input buffer occupy "bytes" bytes.
    We know from the algorithm, that it is too many. How to split
    the buffer so that the head fits to rest of a storage buffer?*/
static inline int PF_longMultiHowSplit(int count, MPI_Datatype type, int bytes)
{
    int ret, items, totalbytes;

    if ( count < 2 ) return(0); /* Nothing to split */
/*
        A rest of a storage buffer:
*/
    totalbytes = PF_packsize - PF_longMultiTop->packpos;
/*
    Rough estimate:
*/
    items = (int)((double)totalbytes*count/bytes);
/*
        Go to the up limit:
*/
    do {
        if ( ( ret = MPI_Pack_size(++items,type,PF_COMM,&bytes) )
            !=MPI_SUCCESS ) return(ret);
    } while ( bytes < totalbytes );
/*
        Now the value of "items" is too large
        And now evaluate the exact value:
*/
    do {
        if ( ( ret = MPI_Pack_size(--items,type,PF_COMM,&bytes) )
            !=MPI_SUCCESS ) return(ret);
        if ( items == 0 ) /* Nothing about MPI_Pack_size(0) == 0 in standards! */
            return(0);
    } while ( bytes > totalbytes );
    return(items);
}
/*
        #] PF_longMultiHowSplit :
        #[ PF_longPackInit :
*/

static int PF_longPackInit(void)
{
    int ret;
    PF_longPackBuf = (UBYTE*)Malloc1(sizeof(UBYTE)*PF_packsize,"PF_longPackBuf");
    if ( PF_longPackBuf == NULL ) return(-1);
/*
        PF_longPackTop is not initialized yet, use in as a return value:
*/
    ret = MPI_Pack_size(1,MPI_INT,PF_COMM,&PF_longPackTop);
    if ( ret != MPI_SUCCESS ) return(ret);

    PF_longPackSmallBuf =
            (VOID*)Malloc1(sizeof(UBYTE)*PF_longPackTop,"PF_longPackSmallBuf");

    PF_longPackTop = PF_packsize;
    PF_longMultiRoot =
            (PF_LONGMULTI *)Malloc1(sizeof(PF_LONGMULTI),"PF_longMultiRoot");
    if ( PF_longMultiRoot == NULL ) return(-1);
    PF_longMultiRoot->bufpos = 0;
    PF_longMultiRoot->buffer = NULL;
    PF_longMultiRoot->next   = NULL;
    PF_longMultiLastChunk    = PF_longMultiRoot;

    PF_longPackPos = 0;
    PF_longMultiRoot->packpos = 0;
    PF_longMultiTop = PF_longMultiRoot;
    PF_longPackN = 1;
    return(0);
}

/*
        #] PF_longPackInit :
        #[ PF_longMultiPreparePrefix :
```

```
*/

static inline int PF_longMultiPreparePrefix(void)
{
    int ret;
    PF_LONGMULTI *thePrefix;
    int i = PF_longPackN;
/*
        Here we have PF_longPackN>1!
        New cell (at the list end) to create the auxiliary chunk:
*/
    if ( PF_longMultiPack2NextCell() ) return(-1);
/*
        Store the pointer to the chunk we will proceed:
*/
    thePrefix = PF_longMultiTop;
/*
        Pack PF_longPackN:
*/
    ret = MPI_Pack(&(PF_longPackN),
                   1,
                   MPI_INT,
                   thePrefix->buffer,
                   PF_packsize,
                   &(thePrefix->packpos),
                   PF_COMM);
    if ( ret != MPI_SUCCESS ) return(ret);
/*
        And start from the beginning:
*/
    for ( PF_longMultiTop = PF_longMultiRoot; i > 0; i-- ) {
/*
            Pack number of Pack hits:
*/
        ret = MPI_Pack(&(PF_longMultiTop->nPacks),
                       1,
                       MPI_INT,
                       thePrefix->buffer,
                       PF_packsize,
                       &(thePrefix->packpos),
                       PF_COMM);
/*
                Pack the length of the last fit portion:
*/
        ret |= MPI_Pack(&(PF_longMultiTop->lastLen),
                        1,
                        MPI_INT,
                        thePrefix->buffer,
                        PF_packsize,
                        &(thePrefix->packpos),
                        PF_COMM);
/*
                Check the size -- not necessary, MPI_Pack did it.
*/
        if (  ret != MPI_SUCCESS ) return(ret);
/*
            Go to the next cell:
*/
        PF_longMultiTop = PF_longMultiTop->next;
    }

    PF_longMultiTop = thePrefix;
/*
        PF_longMultiTop is ready!
*/
    return(0);
}

/*
        #] PF_longMultiPreparePrefix :
        #[ PF_longMultiProcessPrefix :
*/

static inline int PF_longMultiProcessPrefix(void)
{
    int ret,i;
/*
        We have PF_longPackN records packed in PF_longMultiRoot->buffer,
        pairs nPacks and lastLen. Loop through PF_longPackN cells,
        unpacking these integers into proper fields:
*/
    for ( PF_longMultiTop = PF_longMultiRoot, i = 0; i < PF_longPackN; i++ ) {
/*
            Go to th next cell, allocating, when necessary:
*/
        if ( PF_longMultiPack2NextCell() ) return(-1);
/*
```

```
            Unpack the number of Pack hits:
*/
        ret = MPI_Unpack(PF_longMultiRoot->buffer,
                         PF_packsize,
                         &(  PF_longMultiRoot->packpos),
                         &(PF_longMultiTop->nPacks),
                         1,
                         MPI_INT,
                         PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
/*
            Unpack the length of the last fit portion:
*/
        ret = MPI_Unpack(PF_longMultiRoot->buffer,
                         PF_packsize,
                         &(  PF_longMultiRoot->packpos),
                         &(PF_longMultiTop->lastLen),
                         1,
                         MPI_INT,
                         PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
    }
    return(0);
}

/*
        #] PF_longMultiProcessPrefix :
        #[ PF_longSingleReset :
*/

static inline int PF_longSingleReset(int is_sender)
{
    int ret;
    PF_longPackPos=0;
    if ( is_sender ) {
        ret = MPI_Pack(&PF_longPackTop,1,MPI_INT,
            PF_longPackBuf,PF_longPackTop,&PF_longPackPos,PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
        PF_longPackN = 1;
    }
    else {
        PF_longPackN=0;
    }
    return(0);
}

/*
        #] PF_longSingleReset :
        #[ PF_longMultiReset :
*/

static inline int PF_longMultiReset(int is_sender)
{
    int ret = 0, theone = 1;
    PF_longMultiRoot->packpos = 0;
    if ( is_sender ) {
        ret = MPI_Pack(&theone,1,MPI_INT,
            PF_longPackBuf,PF_longPackTop,&(PF_longMultiRoot->packpos),PF_COMM);
        PF_longPackN = 1;
    }
    else {
        PF_longPackN = 0;
    }
    PF_longMultiRoot->nPacks = 0;   /* The auxiliary field is not counted */
    PF_longMultiRoot->lastLen = 0;
    PF_longMultiTop = PF_longMultiRoot;
    PF_longMultiRoot->buffer = PF_longPackBuf;
    return ret;
}

/*
        #] PF_longMultiReset :
        #] Long pack private functions :
        #[ PF_PrepareLongSinglePack :
*/

int PF_PrepareLongSinglePack(void)
{
    return PF_longSingleReset(1);
}

/*
        #] PF_PrepareLongSinglePack :
        #[ PF_LongSinglePack :
*/

int PF_LongSinglePack(const void *buffer, size_t count, MPI_Datatype type)
```

```
{
    int ret, bytes;
    /* XXX: Limited by int size. */
    if ( count > INT_MAX ) return -99;
    ret = MPI_Pack_size((int)count,type,PF_COMM,&bytes);
    if ( ret != MPI_SUCCESS ) return(ret);

    while ( PF_longPackPos+bytes > PF_longPackTop ) {
        if ( PF_longAddChunk(1, 1) ) return(-1);
    }
/*
        PF_longAddChunk(1, 1) means, the chunk must
        be increased by 1 and re-allocated
*/
    ret = MPI_Pack((void *)buffer,(int)count,type,
                    PF_longPackBuf,PF_longPackTop,&PF_longPackPos,PF_COMM);
    if ( ret != MPI_SUCCESS ) return(ret);
    return(0);
}


/*
        #] PF_LongSinglePack :
        #[ PF_LongSingleUnpack :
*/

int PF_LongSingleUnpack(void *buffer, size_t count, MPI_Datatype type)
{
    int ret;
    /* XXX: Limited by int size. */
    if ( count > INT_MAX ) return -99;
    ret = MPI_Unpack(PF_longPackBuf,PF_longPackTop,&PF_longPackPos,
                     buffer,(int)count,type,PF_COMM);
    if ( ret != MPI_SUCCESS ) return(ret);
    return(0);
}

/*
        #] PF_LongSingleUnpack :
        #[ PF_LongSingleSend :
*/

int PF_LongSingleSend(int to, int tag)
{
    int ret, pos = 0;
/*
        Note, here we assume that this function couldn't be used
        with to == PF_ANY_SOURCE!
*/
    if ( PF_longPackN > 1 ) {
        /* The buffer was incremented, pack send the new size first: */
        int tmp = -PF_longPackTop;
/*
            Negative value means there will be the second buffer
*/
        ret = MPI_Pack(&tmp, 1,PF_INT,
                       PF_longPackSmallBuf,PF_longPackTop,&pos,PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
        ret = MPI_Ssend(PF_longPackSmallBuf,pos,MPI_PACKED,to,tag,PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
    }
    ret = MPI_Ssend(PF_longPackBuf,PF_longPackPos,MPI_PACKED,to,tag,PF_COMM);
    if ( ret != MPI_SUCCESS ) return(ret);
    return(0);
}

/*
        #] PF_LongSingleSend :
        #[ PF_LongSingleReceive :
*/

int PF_LongSingleReceive(int src, int tag, int *psrc, int *ptag)
{
    int ret, missed, oncemore;
    MPI_Status status;
    PF_longSingleReset(0);
    do {
        ret = MPI_Recv(PF_longPackBuf,PF_longPackTop,MPI_PACKED,src,tag,
                       PF_COMM,&status);
        if ( ret != MPI_SUCCESS ) return(ret);
/*
            The source and tag must be specified here for the case if
            MPI_Recv is performed more than once:
*/
        src = status.MPI_SOURCE;
        tag = status.MPI_TAG;
        if ( psrc ) *psrc = status.MPI_SOURCE;
        if ( ptag ) *ptag = status.MPI_TAG;
```

```
/*
            Now we got either small buffer with the new PF_longPackTop,
            or just a regular chunk.
*/
        ret = MPI_Unpack(PF_longPackBuf,PF_longPackTop,&PF_longPackPos,
                         &missed,1,MPI_INT,PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);

        if ( missed < 0 ) { /* The small buffer was received. */
            oncemore = 1; /* repeat receiving afterwards */
                          /* Reallocate the buffer and get the data */
            missed = -missed;
/*
                restore after unpacking small from buffer:
*/
            PF_longPackPos = 0;
        }
        else {
            oncemore = 0;  /* That's all, no repetition */
        }
        if ( missed > PF_longPackTop ) {
            /*
             * The room must be increased. We need a re-allocation for the
             * case that there is no repetition.
             */
            if ( PF_longAddChunk( (missed-PF_longPackTop)/PF_packsize, !oncemore ) )
                return(-1);
        }
    } while ( oncemore );
    return(0);
}

/*
        #] PF_LongSingleReceive :
        #[ PF_PrepareLongMultiPack :
*/

int PF_PrepareLongMultiPack(void)
{
    return PF_longMultiReset(1);
}

/*
        #] PF_PrepareLongMultiPack :
        #[ PF_LongMultiPackImpl :
*/

int PF_LongMultiPackImpl(const void*buffer, size_t count, size_t eSize, MPI_Datatype type)
{
    int ret, items;

    /* XXX: Limited by int size. */
    if ( count > INT_MAX ) return -99;

    ret = MPI_Pack_size((int)count,type,PF_COMM,&items);
    if ( ret != MPI_SUCCESS ) return(ret);

    if ( PF_longMultiTop->packpos + items <= PF_packsize ) {
        ret = MPI_Pack((void *)buffer,(int)count,type,PF_longMultiTop->buffer,
                       PF_packsize,&(PF_longMultiTop->packpos),PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
        PF_longMultiTop->nPacks++;
        return(0);
    }
/*
        The data do not fit to the rest of the buffer.
        There are two possibilities here: go to the next cell
        immediately, or first try to pack some portion. The function
        PF_longMultiHowSplit() returns the number of items could be
        packed in the end of the current cell:
*/
    if ( ( items = PF_longMultiHowSplit((int)count,type,items) ) < 0 ) return(items);

    if ( items > 0 ) {   /* store the head */
        ret = MPI_Pack((void *)buffer,items,type,PF_longMultiTop->buffer,
                       PF_packsize,&(PF_longMultiTop->packpos),PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
        PF_longMultiTop->nPacks++;
        PF_longMultiTop->lastLen = items;
    }
/*
        Now the rest should be packed to the new cell.
        Slide to the new cell:
*/
    if ( PF_longMultiPack2NextCell() ) return(-1);
    PF_longPackN++;
/*
```

```
        Pack the rest to the next cell:
*/
    return(PF_LongMultiPackImpl((char *)buffer+items*eSize,count-items,eSize,type));
}

/*
        #] PF_LongMultiPackImpl :
        #[ PF_LongMultiUnpackImpl :
*/

int PF_LongMultiUnpackImpl(void *buffer, size_t count, size_t eSize, MPI_Datatype type)
{
    int ret;

    /* XXX: Limited by int size. */
    if ( count > INT_MAX ) return -99;

    if ( PF_longPackN < 2 ) { /* Just unpack the buffer from the single cell */
        ret = MPI_Unpack(
                    PF_longMultiTop->buffer,
                    PF_packsize,
                    &(PF_longMultiTop->packpos),
                    buffer,
                    count,type,PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
        return(0);
    }
/*
        More than one cell is in use.
*/
    if ( ( PF_longMultiTop->nPacks > 1 )     /* the cell is not expired */
        ||          /* The last cell contains exactly required portion: */
        ( ( PF_longMultiTop->nPacks == 1 ) && ( PF_longMultiTop->lastLen == 0 ) )
    ) {    /* Just unpack the buffer from the current cell */
        ret = MPI_Unpack(
                    PF_longMultiTop->buffer,
                    PF_packsize,
                    &(PF_longMultiTop->packpos),
                    buffer,
                    count,type,PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
        (PF_longMultiTop->nPacks)--;
        return(0);
    }
    if ( ( PF_longMultiTop->nPacks == 1 ) && ( PF_longMultiTop->lastLen != 0 ) ) {
/*
            Unpack the head:
*/
        ret = MPI_Unpack(
                    PF_longMultiTop->buffer,
                    PF_packsize,
                    &(PF_longMultiTop->packpos),
                    buffer,
                    PF_longMultiTop->lastLen,type,PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
/*
            Decrement the counter by read items:
*/
        count -= PF_longMultiTop->lastLen;
        if ( count <= 0 ) return(-1);  /*Something is wrong! */
/*
            Shift the output buffer position:
*/
        buffer = (char *)buffer + PF_longMultiTop->lastLen * eSize;
        (PF_longMultiTop->nPacks)--;
    }
/*
        Here PF_longMultiTop->nPacks == 0
*/
    if ( ( PF_longMultiTop = PF_longMultiTop->next ) == NULL ) return(-1);
    return(PF_LongMultiUnpackImpl(buffer,count,eSize,type));
}

/*
        #] PF_LongMultiUnpackImpl :
        #[ PF_LongMultiBroadcast :
*/

int PF_LongMultiBroadcast(void)
{
    int ret, i;

    if ( PF.me == MASTER ) {
/*
            PF_longPackN is the number of packed chunks. If it is more
            than 1, we have to pack a new one and send it first
*/
```

```
        if ( PF_longPackN > 1 ) {
            if ( PF_longMultiPreparePrefix() ) return(-1);
            ret = MPI_Bcast((VOID*)PF_longMultiTop->buffer,
                            PF_packsize,MPI_PACKED,MASTER,PF_COMM);
            if ( ret != MPI_SUCCESS ) return(ret);
/*
                PF_longPackN was not incremented by PF_longMultiPreparePrefix()!
*/
        }
/*
            Now we start from the beginning:
*/
        PF_longMultiTop = PF_longMultiRoot;
/*
            Just broadcast all the chunks:
*/
        for ( i = 0; i < PF_longPackN; i++ ) {
            ret = MPI_Bcast((VOID*)PF_longMultiTop->buffer,
                            PF_packsize,MPI_PACKED,MASTER,PF_COMM);
            if ( ret != MPI_SUCCESS ) return(ret);
            PF_longMultiTop = PF_longMultiTop->next;
        }
        return(0);
    }
/*
        else - the slave
*/
    PF_longMultiReset(0);
/*
        Get the first chunk; it can be either the only data chunk, or
        an auxiliary chunk, if the data do not fit the single chunk:
*/
    ret = MPI_Bcast((VOID*)PF_longMultiRoot->buffer,
                    PF_packsize,MPI_PACKED,MASTER,PF_COMM);
    if ( ret != MPI_SUCCESS ) return(ret);

    ret = MPI_Unpack((VOID*)PF_longMultiRoot->buffer,
                     PF_packsize,
                     &(PF_longMultiRoot->packpos),
                     &PF_longPackN,1,MPI_INT,PF_COMM);
    if ( ret != MPI_SUCCESS ) return(ret);
/*
        Now in PF_longPackN we have the number of cells used
        for broadcasting. If it is >1, then we have to allocate
        enough cells, initialize them and receive all the chunks.
*/
    if ( PF_longPackN < 2 ) /* That's all, the single chunk is received. */
        return(0);
/*
        Here we have to get PF_longPackN chunks. But, first,
        initialize cells by info from the received auxiliary chunk.
*/
    if ( PF_longMultiProcessPrefix() ) return(-1);
/*
        Now we have free PF_longPackN cells, starting
        from PF_longMultiRoot->next,  with properly initialized
        nPacks and lastLen fields. Get chunks:
*/
    for ( PF_longMultiTop = PF_longMultiRoot->next, i = 0; i < PF_longPackN; i++ ) {
        ret = MPI_Bcast((VOID*)PF_longMultiTop->buffer,
                        PF_packsize,MPI_PACKED,MASTER,PF_COMM);
        if ( ret != MPI_SUCCESS ) return(ret);
        if ( i == 0 ) {   /* The first chunk, it contains extra "1". */
            int tmp;
/*
                Extract this 1 into tmp and forget about it.
*/
            ret = MPI_Unpack((VOID*)PF_longMultiTop->buffer,
                             PF_packsize,
                             &(PF_longMultiTop->packpos),
                             &tmp,1,MPI_INT,PF_COMM);
            if ( ret != MPI_SUCCESS ) return(ret);
        }
        PF_longMultiTop = PF_longMultiTop->next;
    }
/*
        multiUnPack starts with PF_longMultiTop, skip auxiliary chunk in
        PF_longMultiRoot:
*/
    PF_longMultiTop = PF_longMultiRoot->next;
    return(0);
}

/*
        #] PF_LongMultiBroadcast :
    #] Long pack stuff :
*/
```

4.62 mpidbg.h File Reference

```
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <mpi.h>
```

Include dependency graph for mpidbg.h:

Macros

- #define MPIDBG_RANK MPIDBG_Get_rank()
- #define MPIDBG_Out(...) MPIDBG_Out(file, line, func, __VA_ARGS__)
- #define MPIDBG_EXTARG const char *file, int line, const char *func
- #define MPI_Send(...) MPIDBG_Send(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Recv(...) MPIDBG_Recv(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Bsend(...) MPIDBG_Bsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Ssend(...) MPIDBG_Ssend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Rsend(...) MPIDBG_Rsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Isend(...) MPIDBG_Isend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Ibsend(...) MPIDBG_Ibsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Issend(...) MPIDBG_Issend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Irsend(...) MPIDBG_Irsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Irecv(...) MPIDBG_Irecv(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Wait(...) MPIDBG_Wait(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Test(...) MPIDBG_Test(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Waitany(...) MPIDBG_Waitany(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Testany(...) MPIDBG_Testany(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Waitall(...) MPIDBG_Waitall(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Testall(...) MPIDBG_Testall(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Waitsome(...) MPIDBG_Waitsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Testsome(...) MPIDBG_Testsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Iprobe(...) MPIDBG_Iprobe(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Probe(...) MPIDBG_Probe(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Cancel(...) MPIDBG_Cancel(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Test_cancelled(...) MPIDBG_Test_cancelled(__VA_ARGS__, __FILE__, __LINE__, __func_↩ _)
- #define MPI_Barrier(...) MPIDBG_Barrier(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define MPI_Bcast(...) MPIDBG_Bcast(__VA_ARGS__, __FILE__, __LINE__, __func__)

4.62.1 Detailed Description

MPI APIs with the logging feature. NOTE: This file needs C99.

Definition in file mpidbg.h.

4.62.2 Macro Definition Documentation

4.62.2.1 MPIDBG_RANK

```
#define MPIDBG_RANK MPIDBG_Get_rank()
```

Definition at line 55 of file mpidbg.h.

4.62.2.2 MPIDBG_Out

```
#define MPIDBG_Out(
              ...  ) MPIDBG_Out(file, line, func, __VA_ARGS__)
```

Definition at line 72 of file mpidbg.h.

4.62.2.3 MPIDBG_EXTARG

```
#define MPIDBG_EXTARG const char *file, int line, const char *func
```

Definition at line 139 of file mpidbg.h.

4.62.2.4 MPI_Send

```
#define MPI_Send(
              ...  ) MPIDBG_Send(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 157 of file mpidbg.h.

4.62.2.5 MPI_Recv

```
#define MPI_Recv(
              ...  ) MPIDBG_Recv(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 178 of file mpidbg.h.

4.62.2.6 MPI_Bsend

```
#define MPI_Bsend(
              ...  ) MPIDBG_Bsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 197 of file mpidbg.h.

4.62.2.7 MPI_Ssend

```
#define MPI_Ssend(
            ...  ) MPIDBG_Ssend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 216 of file mpidbg.h.

4.62.2.8 MPI_Rsend

```
#define MPI_Rsend(
            ...  ) MPIDBG_Rsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 235 of file mpidbg.h.

4.62.2.9 MPI_Isend

```
#define MPI_Isend(
            ...  ) MPIDBG_Isend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 254 of file mpidbg.h.

4.62.2.10 MPI_Ibsend

```
#define MPI_Ibsend(
            ...  ) MPIDBG_Ibsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 273 of file mpidbg.h.

4.62.2.11 MPI_Issend

```
#define MPI_Issend(
            ...  ) MPIDBG_Issend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 292 of file mpidbg.h.

4.62.2.12 MPI_Irsend

```
#define MPI_Irsend(
            ...  ) MPIDBG_Irsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 311 of file mpidbg.h.

4.62.2.13 MPI_Irecv

```
#define MPI_Irecv(
            ...  ) MPIDBG_Irecv(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 330 of file mpidbg.h.

4.62.2.14 MPI_Wait

```
#define MPI_Wait(
            ...  ) MPIDBG_Wait(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 353 of file mpidbg.h.

4.62.2.15 MPI_Test

```
#define MPI_Test(
            ...  ) MPIDBG_Test(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 381 of file mpidbg.h.

4.62.2.16 MPI_Waitany

```
#define MPI_Waitany(
            ...  ) MPIDBG_Waitany(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 407 of file mpidbg.h.

4.62.2.17 MPI_Testany

```
#define MPI_Testany(
            ...  ) MPIDBG_Testany(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 438 of file mpidbg.h.

4.62.2.18 MPI_Waitall

```
#define MPI_Waitall(
            ...  ) MPIDBG_Waitall(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 462 of file mpidbg.h.

4.62.2.19 MPI_Testall

```
#define MPI_Testall(
            ...  ) MPIDBG_Testall(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 491 of file mpidbg.h.

4.62.2.20 MPI_Waitsome

```
#define MPI_Waitsome(
            ...  ) MPIDBG_Waitsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 515 of file mpidbg.h.

4.62.2.21 MPI_Testsome

```
#define MPI_Testsome(
            ...  ) MPIDBG_Testsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 539 of file mpidbg.h.

4.62.2.22 MPI_Iprobe

```
#define MPI_Iprobe(
            ...  ) MPIDBG_Iprobe(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 565 of file mpidbg.h.

4.62.2.23 MPI_Probe

```
#define MPI_Probe(
            ...  ) MPIDBG_Probe(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 586 of file mpidbg.h.

4.62.2.24 MPI_Cancel

```
#define MPI_Cancel(
            ...  ) MPIDBG_Cancel(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 605 of file mpidbg.h.

4.62.2.25 MPI_Test_cancelled

```
#define MPI_Test_cancelled(
            ...  ) MPIDBG_Test_cancelled(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 629 of file mpidbg.h.

4.62.2.26 MPI_Barrier

```
#define MPI_Barrier(
            ...  ) MPIDBG_Barrier(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 648 of file mpidbg.h.

4.62.2.27 MPI_Bcast

```
#define MPI_Bcast(
            ...  ) MPIDBG_Bcast(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 667 of file mpidbg.h.

4.63 mpidbg.h

Go to the documentation of this file.

```
#ifndef MPIDBG_H
#define MPIDBG_H

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

/*
    #[ Includes :
*/

#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <mpi.h>

/*
    #] Includes :
    #[ Utilities :
        #[ MPIDBG_RANK :
*/

static inline int MPIDBG_Get_rank(void) {
    return PF.me;  /* Assume we are working with ParFORM. */
}
#define MPIDBG_RANK MPIDBG_Get_rank()

/*
        #] MPIDBG_RANK :
        #[ MPIDBG_Out :
*/

static inline void MPIDBG_Out(const char *file, int line, const char *func, const char *fmt, ...) {
    char buf[1024];  /* Enough. */
    va_list ap;
    va_start(ap, fmt);
    sprintf(buf, "*** [%d] %10s %4d @ %-16s: ", MPIDBG_RANK, file, line, func);
    vsprintf(buf + strlen(buf), fmt, ap);
    va_end(ap);
    /* Assume fprintf with a line will work well even in multi-processes. */
    fprintf(stderr, "%s\n", buf);
}
#define MPIDBG_Out(...) MPIDBG_Out(file, line, func, __VA_ARGS__)

/*
        #] MPIDBG_Out :
        #[ MPIDBG_sprint_requests :
*/

static inline void MPIDBG_sprint_requests(char *buf, int count, const MPI_Request *requests)
{
    /* Assume sprintf never fail and returns >= 0... */
    buf += sprintf(buf, "(");
    int i, first = 1;
    for ( i = 0; i < count; i++ ) {
        if ( first ) {
            first = 0;
        }
        else {
```

```
            buf += sprintf(buf, ",");
        }
        if ( requests[i] != MPI_REQUEST_NULL ) {
            buf += sprintf(buf, "%d", i);
        }
        else {
            buf += sprintf(buf, "*");
        }
    }
    buf += sprintf(buf, ")");
}

/*
        #] MPIDBG_sprint_requests :
        #[ MPIDBG_sprint_statuses :
*/

static inline void MPIDBG_sprint_statuses(char *buf, int count, const MPI_Request *old_requests, const MPI_Request *new_requests, const MPI_Status *statuses)
{
    /* Assume sprintf never fail and returns >= 0... */
    buf += sprintf(buf, "(");
    int i, first = 1;
    for ( i = 0; i < count; i++ ) {
        if ( first ) {
            first = 0;
        }
        else {
            buf += sprintf(buf, ",");
        }
        if ( old_requests[i] != MPI_REQUEST_NULL && new_requests[i] == MPI_REQUEST_NULL ) {
            int ret_size = 0;
            MPI_Get_count((MPI_Status *)&statuses[i], MPI_BYTE, &ret_size);
            buf += sprintf(buf, "(source=%d,tag=%d,size=%d)", statuses[i].MPI_SOURCE, statuses[i].MPI_TAG, ret_size);
        } else {
            buf += sprintf(buf, "*");
        }
    }
    buf += sprintf(buf, ")");
}

/*
        #] MPIDBG_sprint_statuses :
    #] Utilities :
    #[ MPI APIs :
*/

/*
 * The followings are the MPI APIs with the logging.
 */

#define MPIDBG_EXTARG const char *file, int line, const char *func

/*
        #[ MPI_Send :
*/

static inline int MPIDBG_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Send: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
    int ret = MPI_Send(buf, count, datatype, dest, tag, comm);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Send: OK");
    }
    else {
        MPIDBG_Out("MPI_Send: Failed");
    }
    return ret;
}
#define MPI_Send(...) MPIDBG_Send(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Send :
        #[ MPI_Recv :
*/

static inline int MPIDBG_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Recv: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
    int ret = MPI_Recv(buf, count, datatype, source, tag, comm, status);
    if ( ret == MPI_SUCCESS ) {
        int ret_count = 0;
        MPI_Get_count(status, datatype, &ret_count);
        MPIDBG_Out("MPI_Recv: OK src=%d dest=%d tag=%d count=%d", status->MPI_SOURCE, MPIDBG_RANK,
```

```
status->MPI_TAG, ret_count);
    }
    else {
        MPIDBG_Out("MPI_Recv: Failed");
    }
    return ret;
}
#define MPI_Recv(...) MPIDBG_Recv(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Recv :
        #[ MPI_Bsend :
*/

static inline int MPIDBG_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Bsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
    int ret = MPI_Bsend(buf, count, datatype, dest, tag, comm);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Bsend: OK");
    }
    else {
        MPIDBG_Out("MPI_Bsend: Failed");
    }
    return ret;
}
#define MPI_Bsend(...) MPIDBG_Bsend(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Bsend :
        #[ MPI_Ssend :
*/

static inline int MPIDBG_Ssend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Ssend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
    int ret = MPI_Ssend(buf, count, datatype, dest, tag, comm);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Ssend: OK");
    }
    else {
        MPIDBG_Out("MPI_Ssend: Failed");
    }
    return ret;
}
#define MPI_Ssend(...) MPIDBG_Ssend(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Ssend :
        #[ MPI_Rsend :
*/

static inline int MPIDBG_Rsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Rsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
    int ret = MPI_Rsend(buf, count, datatype, dest, tag, comm);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Rsend: OK");
    }
    else {
        MPIDBG_Out("MPI_Rsend: Failed");
    }
    return ret;
}
#define MPI_Rsend(...) MPIDBG_Rsend(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Rsend :
        #[ MPI_Isend :
*/

static inline int MPIDBG_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Isend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
    int ret = MPI_Isend(buf, count, datatype, dest, tag, comm, request);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Isend: OK");
    }
    else {
        MPIDBG_Out("MPI_Isend: Failed");
    }
    return ret;
}
```

```
#define MPI_Isend(...) MPIDBG_Isend(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Isend :
        #[ MPI_Ibsend :
*/

static inline int MPIDBG_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Ibsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
    int ret = MPI_Ibsend(buf, count, datatype, dest, tag, comm, request);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Ibsend: OK");
    }
    else {
        MPIDBG_Out("MPI_Ibsend: Failed");
    }
    return ret;
}
#define MPI_Ibsend(...) MPIDBG_Ibsend(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Ibsend :
        #[ MPI_Issend :
*/

static inline int MPIDBG_Issend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Issend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
    int ret = MPI_Issend(buf, count, datatype, dest, tag, comm, request);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Issend: OK");
    }
    else {
        MPIDBG_Out("MPI_Issend: Failed");
    }
    return ret;
}
#define MPI_Issend(...) MPIDBG_Issend(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Issend :
        #[ MPI_Irsend :
*/

static inline int MPIDBG_Irsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Irsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
    int ret = MPI_Irsend(buf, count, datatype, dest, tag, comm, request);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Irsend: OK");
    }
    else {
        MPIDBG_Out("MPI_Irsend: Failed");
    }
    return ret;
}
#define MPI_Irsend(...) MPIDBG_Irsend(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Irsend :
        #[ MPI_Irecv :
*/

static inline int MPIDBG_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Irecv: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
    int ret = MPI_Irecv(buf, count, datatype, source, tag, comm, request);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Irecv: OK dest=%d", MPIDBG_RANK);
    }
    else {
        MPIDBG_Out("MPI_Irecv: Failed");
    }
    return ret;
}
#define MPI_Irecv(...) MPIDBG_Irecv(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Irecv :
        #[ MPI_Wait :
*/

```

```
static inline int MPIDBG_Wait(MPI_Request *request, MPI_Status *status, MPIDBG_EXTARG)
{
    char buf[256 * 1];  /* Enough. */
    MPI_Request old_request = *request;
    MPIDBG_sprint_requests(buf, 1, request);
    MPIDBG_Out("MPI_Wait: rank=%d request=%s", MPIDBG_RANK, buf);
    int ret = MPI_Wait(request, status);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_sprint_statuses(buf, 1, request, &old_request, status);
        MPIDBG_Out("MPI_Wait: OK rank=%d result=%s", MPIDBG_RANK, buf);
    }
    else {
        MPIDBG_Out("MPI_Wait: Failed");
    }
    return ret;
}
#define MPI_Wait(...) MPIDBG_Wait(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Wait :
        #[ MPI_Test :
*/

static inline int MPIDBG_Test(MPI_Request *request, int *flag, MPI_Status *status, MPIDBG_EXTARG)
{
    char buf[256 * 1];  /* Enough. */
    MPI_Request old_request = *request;
    MPIDBG_sprint_requests(buf, 1, request);
    MPIDBG_Out("MPI_Test: rank=%d request=%s", MPIDBG_RANK, buf);
    int ret = MPI_Test(request, flag, status);
    if ( ret == MPI_SUCCESS ) {
        if ( *flag ) {
            MPIDBG_sprint_statuses(buf, 1, request, &old_request, status);
            MPIDBG_Out("MPI_Test: OK rank=%d result=%s", MPIDBG_RANK, buf);
        }
        else {
            MPIDBG_Out("MPI_Test: OK flag=false");
        }
    }
    else {
        MPIDBG_Out("MPI_Test: Failed");
    }
    return ret;
}
#define MPI_Test(...) MPIDBG_Test(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Test :
        #[ MPI_Waitany :
*/

static inline int MPIDBG_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status
    *status, MPIDBG_EXTARG)
{
    char buf[256];  /* Enough. */
    MPI_Request old_requests[count];
    memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
    MPIDBG_sprint_requests(buf, count, array_of_requests);
    MPIDBG_Out("MPI_Waitany: rank=%d request=%s", MPIDBG_RANK, buf);
    int ret = MPI_Waitany(count, array_of_requests, index, status);
    if ( ret == MPI_SUCCESS ) {
        MPI_Status statuses[count];
        statuses[*index] = *status;
        MPIDBG_sprint_statuses(buf, count, old_requests, array_of_requests, statuses);
        MPIDBG_Out("MPI_Waitany: OK rank=%d result=%s", MPIDBG_RANK, buf);
    }
    else {
        MPIDBG_Out("MPI_Waitany: Failed");
    }
    return ret;
}
#define MPI_Waitany(...) MPIDBG_Waitany(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Waitany :
        #[ MPI_Testany :
*/

static inline int MPIDBG_Testany(int count, MPI_Request *array_of_requests, int *index, int *flag,
    MPI_Status *status, MPIDBG_EXTARG)
{
    char buf[256];  /* Enough. */
    MPI_Request old_requests[count];
    memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
    MPIDBG_sprint_requests(buf, count, array_of_requests);
    MPIDBG_Out("MPI_Testany: rank=%d request=%s", MPIDBG_RANK, buf);
    int ret = MPI_Testany(count, array_of_requests, index, flag, status);
```

```
    if ( ret == MPI_SUCCESS ) {
        if ( *flag ) {
            MPI_Status statuses[count];
            statuses[*index] = *status;
            MPIDBG_sprint_statuses(buf, count, old_requests, array_of_requests, statuses);
            MPIDBG_Out("MPI_Testany: OK rank=%d result=%s", MPIDBG_RANK, buf);
        }
        else {
            MPIDBG_Out("MPI_Testany: OK flag=false");
        }
    }
    else {
        MPIDBG_Out("MPI_Testany: Failed");
    }
    return ret;
}
#define MPI_Testany(...) MPIDBG_Testany(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Testany :
        #[ MPI_Waitall :
*/

static inline int MPIDBG_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses, MPIDBG_EXTARG)
{
    char buf[256 * count];  /* Enough. */
    MPI_Request old_requests[count];
    memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
    MPIDBG_sprint_requests(buf, count, array_of_requests);
    MPIDBG_Out("MPI_Waitall: rank=%d request=%s", MPIDBG_RANK, buf);
    int ret = MPI_Waitall(count, array_of_requests, array_of_statuses);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_sprint_statuses(buf, count, old_requests, array_of_requests, array_of_statuses);
        MPIDBG_Out("MPI_Waitall: OK rank=%d result=%s", MPIDBG_RANK, buf);
    }
    else {
        MPIDBG_Out("MPI_Waitall: Failed");
    }
    return ret;
}
#define MPI_Waitall(...) MPIDBG_Waitall(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Waitall :
        #[ MPI_Testall :
*/

static inline int MPIDBG_Testall(int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses, MPIDBG_EXTARG)
{
    char buf[256 * count];  /* Enough. */
    MPI_Request old_requests[count];
    memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
    MPIDBG_sprint_requests(buf, count, array_of_requests);
    MPIDBG_Out("MPI_Testall: rank=%d request=%s", MPIDBG_RANK, buf);
    int ret = MPI_Testall(count, array_of_requests, flag, array_of_statuses);
    if ( ret == MPI_SUCCESS ) {
        if ( *flag ) {
            MPIDBG_sprint_statuses(buf, count, old_requests, array_of_requests, array_of_statuses);
            MPIDBG_Out("MPI_Testall: OK rank=%d result=%s", MPIDBG_RANK, buf);
        }
        else {
            MPIDBG_Out("MPI_Testall: OK flag=false");
        }
    }
    else {
        MPIDBG_Out("MPI_Testall: Failed");
    }
    return ret;
}
#define MPI_Testall(...) MPIDBG_Testall(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Testall :
        #[ MPI_Waitsome :
*/

static inline int MPIDBG_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses, MPIDBG_EXTARG)
{
    char buf[256 * incount];  /* Enough. */
    MPI_Request old_requests[incount];
    memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * incount);
    MPIDBG_sprint_requests(buf, incount, array_of_requests);
    MPIDBG_Out("MPI_Waitsome: rank=%d request=%s", MPIDBG_RANK, buf);
    int ret = MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses);
```

```
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_sprint_statuses(buf, incount, old_requests, array_of_requests, array_of_statuses);
        MPIDBG_Out("MPI_Waitsome: OK rank=%d result=%s", MPIDBG_RANK, buf);
    }
    else {
        MPIDBG_Out("MPI_Waitsome: Failed");
    }
    return ret;
}
#define MPI_Waitsome(...) MPIDBG_Waitsome(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Waitsome :
        #[ MPI_Testsome :
*/

static inline int MPIDBG_Testsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses, MPIDBG_EXTARG)
{
    char buf[256 * incount];  /* Enough. */
    MPI_Request old_requests[incount];
    memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * incount);
    MPIDBG_sprint_requests(buf, incount, array_of_requests);
    MPIDBG_Out("MPI_Testsome: rank=%d request=%s", MPIDBG_RANK, buf);
    int ret = MPI_Testsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_sprint_statuses(buf, incount, old_requests, array_of_requests, array_of_statuses);
        MPIDBG_Out("MPI_Testsome: OK rank=%d result=%s", MPIDBG_RANK, buf);
    }
    else {
        MPIDBG_Out("MPI_Testsome: Failed");
    }
    return ret;
}
#define MPI_Testsome(...) MPIDBG_Testsome(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Testsome :
        #[ MPI_Iprobe :
*/

static inline int MPIDBG_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Iprobe: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
    int ret = MPI_Iprobe(source, tag, comm, flag, status);
    if ( ret == MPI_SUCCESS ) {
        if ( *flag ) {
            int ret_size = 0;
            MPI_Get_count(status, MPI_BYTE, &ret_size);
            MPIDBG_Out("MPI_Iprobe: OK src=%d dest=%d tag=%d size=%d", status->MPI_SOURCE, MPIDBG_RANK, status->MPI_TAG, ret_size);
        }
        else {
            MPIDBG_Out("MPI_Iprobe: OK flag=false");
        }
    }
    else {
        MPIDBG_Out("MPI_Iprobe: Failed");
    }
    return ret;
}
#define MPI_Iprobe(...) MPIDBG_Iprobe(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Iprobe :
        #[ MPI_Probe :
*/

static inline int MPIDBG_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Probe: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
    int ret = MPI_Probe(source, tag, comm, status);
    if ( ret == MPI_SUCCESS ) {
        int ret_size = 0;
        MPI_Get_count(status, MPI_BYTE, &ret_size);
        MPIDBG_Out("MPI_Probe: OK src=%d dest=%d tag=%d size=%d", status->MPI_SOURCE, MPIDBG_RANK, status->MPI_TAG, ret_size);
    }
    else {
        MPIDBG_Out("MPI_Probe: Failed");
    }
    return ret;
}
#define MPI_Probe(...) MPIDBG_Probe(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
```

```
        #] MPI_Probe :
        #[ MPI_Cancel :
*/

static inline int MPIDBG_Cancel(MPI_Request *request, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Cancel: rank=%d", MPIDBG_RANK);
    int ret = MPI_Cancel(request);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Cancel: OK");
    }
    else {
        MPIDBG_Out("MPI_Cancel: Failed");
    }
    return ret;
}
#define MPI_Cancel(...) MPIDBG_Cancel(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Cancel :
        #[ MPI_Test_cancelled :
*/

static inline int MPIDBG_Test_cancelled(MPI_Status *status, int *flag, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Test_cancelled: rank=%d", MPIDBG_RANK);
    int ret = MPI_Test_cancelled(status, flag);
    if ( ret == MPI_SUCCESS ) {
        if ( *flag ) {
            MPIDBG_Out("MPI_Test_cancelled: OK flag=true");
        }
        else {
            MPIDBG_Out("MPI_Test_cancelled: OK flag=false");
        }
    }
    else {
        MPIDBG_Out("MPI_Test_cancelled: Failed");
    }
    return ret;
}
#define MPI_Test_cancelled(...) MPIDBG_Test_cancelled(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Test_cancelled :
        #[ MPI_Barrier :
*/

static inline int MPIDBG_Barrier(MPI_Comm comm, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Barrier: rank=%d", MPIDBG_RANK);
    int ret = MPI_Barrier(comm);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Barrier: OK");
    }
    else {
        MPIDBG_Out("MPI_Barrier: Failed");
    }
    return ret;
}
#define MPI_Barrier(...) MPIDBG_Barrier(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Barrier :
        #[ MPI_Bcast :
*/

static inline int MPIDBG_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
    comm, MPIDBG_EXTARG)
{
    MPIDBG_Out("MPI_Bcast: root=%d count=%d", root, count);
    int ret = MPI_Bcast(buffer, count, datatype, root, comm);
    if ( ret == MPI_SUCCESS ) {
        MPIDBG_Out("MPI_Bcast: OK");
    }
    else {
        MPIDBG_Out("MPI_Bcast: Failed");
    }
    return ret;
}
#define MPI_Bcast(...) MPIDBG_Bcast(__VA_ARGS__, __FILE__, __LINE__, __func__)

/*
        #] MPI_Bcast :
    #] MPI APIs :
*/

#endif  /* MPIDBG_H */
```

4.64 mytime.h

```
#pragma once
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

double thetime();
std::string thetime_str();
```

4.65 names.c File Reference

`#include "form3.h"`

Include dependency graph for names.c:

Functions

- NAMENODE * GetNode (NAMETREE *nametree, UBYTE *name)
- int AddName (NAMETREE *nametree, UBYTE *name, WORD type, WORD number, int *nodenum)
- int GetName (NAMETREE *nametree, UBYTE *name, WORD *number, int par)
- int GetLastExprName (UBYTE *name, WORD *number)
- int GetOName (NAMETREE *nametree, UBYTE *name, WORD *number, int par)
- int GetAutoName (UBYTE *name, WORD *number)
- int GetVar (UBYTE *name, WORD *type, WORD *number, int wantedtype, int par)
- WORD EntVar (WORD type, UBYTE *name, WORD x, WORD y, WORD z, WORD d)
- int GetDollar (UBYTE *name)
- VOID DumpTree (NAMETREE *nametree)
- VOID DumpNode (NAMETREE *nametree, WORD node, WORD depth)

- int CompactifyTree (NAMETREE ∗nametree, WORD par)
- VOID CopyTree (NAMETREE ∗newtree, NAMETREE ∗oldtree, WORD node, WORD par)
- VOID LinkTree (NAMETREE ∗tree, WORD offset, WORD numnodes)
- NAMETREE ∗ MakeNameTree ()
- VOID FreeNameTree (NAMETREE ∗n)
- void ClearWildcardNames ()
- int AddWildcardName (UBYTE ∗name)
- int GetWildcardName (UBYTE ∗name)
- int AddSymbol (UBYTE ∗name, int minpow, int maxpow, int cplx, int dim)
- int CoSymbol (UBYTE ∗s)
- int AddIndex (UBYTE ∗name, int dim, int dim4)
- int CoIndex (UBYTE ∗s)
- UBYTE ∗ DoDimension (UBYTE ∗s, int ∗dim, int ∗dim4)
- int CoDimension (UBYTE ∗s)
- int AddVector (UBYTE ∗name, int cplx, int dim)
- int CoVector (UBYTE ∗s)
- int AddFunction (UBYTE ∗name, int comm, int istensor, int cplx, int symprop, int dim, int argmax, int argmin)
- int CoCommuteInSet (UBYTE ∗s)
- int CoFunction (UBYTE ∗s, int comm, int istensor)
- int CoNFunction (UBYTE ∗s)
- int CoCFunction (UBYTE ∗s)
- int CoNTensor (UBYTE ∗s)
- int CoCTensor (UBYTE ∗s)
- int DoTable (UBYTE ∗s, int par)
- int CoTable (UBYTE ∗s)
- int CoNTable (UBYTE ∗s)
- int CoCTable (UBYTE ∗s)
- void EmptyTable (TABLES T)
- int AddSet (UBYTE ∗name, WORD dim)
- int DoElements (UBYTE ∗s, SETS set, UBYTE ∗name)
- int CoSet (UBYTE ∗s)
- int DoTempSet (UBYTE ∗from, UBYTE ∗to)
- int CoAuto (UBYTE ∗inp)
- int AddDollar (UBYTE ∗name, WORD type, WORD ∗start, LONG size)
- int ReplaceDollar (WORD number, WORD newtype, WORD ∗newstart, LONG newsize)
- int AddDubious (UBYTE ∗name)
- int MakeDubious (NAMETREE ∗nametree, UBYTE ∗name, WORD ∗number)
- int NameConflict (int type, UBYTE ∗name)
- int AddExpression (UBYTE ∗name, int x, int y)
- int GetLabel (UBYTE ∗name)
- void ResetVariables (int par)
- void RemoveDollars ()
- void Globalize (int par)
- int TestName (UBYTE ∗name)

4.65.1 Detailed Description

The complete names administration. All variables with a name have to pass here to be properly registered, have structs of the proper type assigned to them etc. The file also contains the utility routines for maintaining the balanced trees that make searching for names rather fast.

Definition in file names.c.

4.65.2 Function Documentation

4.65.2.1 GetNode()

```
NAMENODE * GetNode (
            NAMETREE * nametree,
            UBYTE * name )
```

Definition at line 49 of file names.c.

4.65.2.2 AddName()

```
int AddName (
            NAMETREE * nametree,
            UBYTE * name,
            WORD type,
            WORD number,
            int * nodenum )
```

Definition at line 71 of file names.c.

4.65.2.3 GetName()

```
int GetName (
            NAMETREE * nametree,
            UBYTE * name,
            WORD * number,
            int par )
```

Definition at line 252 of file names.c.

4.65.2.4 GetLastExprName()

```
int GetLastExprName (
            UBYTE * name,
            WORD * number )
```

Definition at line 340 of file names.c.

4.65.2.5 GetOName()

```
int GetOName (
            NAMETREE * nametree,
            UBYTE * name,
            WORD * number,
            int par )
```

Definition at line 362 of file names.c.

4.65.2.6 GetAutoName()

```
int GetAutoName (
            UBYTE * name,
            WORD * number )
```

Definition at line 381 of file names.c.

4.65.2.7 GetVar()

```
int GetVar (
            UBYTE * name,
            WORD * type,
            WORD * number,
            int wantedtype,
            int par )
```

Definition at line 426 of file names.c.

4.65.2.8 EntVar()

```
WORD EntVar (
            WORD type,
            UBYTE * name,
            WORD x,
            WORD y,
            WORD z,
            WORD d )
```

Definition at line 459 of file names.c.

4.65.2.9 GetDollar()

```
int GetDollar (
            UBYTE * name )
```

Definition at line 492 of file names.c.

4.65.2.10 DumpTree()

```
VOID DumpTree (
            NAMETREE * nametree )
```

Definition at line 504 of file names.c.

4.65.2.11 DumpNode()

```
VOID DumpNode (
            NAMETREE * nametree,
            WORD node,
            WORD depth )
```

Definition at line 517 of file names.c.

4.65.2.12 CompactifyTree()

```
int CompactifyTree (
            NAMETREE * nametree,
            WORD par )
```

Definition at line 536 of file names.c.

4.65.2.13 CopyTree()

```
VOID CopyTree (
            NAMETREE * newtree,
            NAMETREE * oldtree,
            WORD node,
            WORD par )
```

Definition at line 598 of file names.c.

4.65.2.14 LinkTree()

```
VOID LinkTree (
            NAMETREE * tree,
            WORD offset,
            WORD numnodes )
```

Definition at line 686 of file names.c.

4.65.2.15 MakeNameTree()

```
NAMETREE * MakeNameTree ( )
```

Definition at line 717 of file names.c.

4.65.2.16 FreeNameTree()

```
VOID FreeNameTree (
            NAMETREE * n )
```

Definition at line 736 of file names.c.

4.65.2.17 ClearWildcardNames()

```
void ClearWildcardNames ( )
```

Definition at line 751 of file names.c.

4.65.2.18 AddWildcardName()

```
int AddWildcardName (
            UBYTE * name )
```

Definition at line 756 of file names.c.

4.65.2.19 GetWildcardName()

```
int GetWildcardName (
            UBYTE * name )
```

Definition at line 800 of file names.c.

4.65.2.20 AddSymbol()

```
int AddSymbol (
            UBYTE * name,
            int minpow,
            int maxpow,
            int cplx,
            int dim )
```

Definition at line 822 of file names.c.

4.65.2.21 CoSymbol()

```
int CoSymbol (
            UBYTE * s )
```

Definition at line 848 of file names.c.

4.65.2.22 AddIndex()

```
int AddIndex (
            UBYTE * name,
            int dim,
            int dim4 )
```

Definition at line 990 of file names.c.

4.65.2.23 CoIndex()

```
int CoIndex (
            UBYTE * s )
```

Definition at line 1014 of file names.c.

4.65.2.24 DoDimension()

```
UBYTE * DoDimension (
            UBYTE * s,
            int * dim,
            int * dim4 )
```

Definition at line 1062 of file names.c.

4.65.2.25 CoDimension()

```
int CoDimension (
            UBYTE * s )
```

Definition at line 1125 of file names.c.

4.65.2.26 AddVector()

```
int AddVector (
            UBYTE * name,
            int cplx,
            int dim )
```

Definition at line 1143 of file names.c.

4.65.2.27 CoVector()

```
int CoVector (
            UBYTE * s )
```

Definition at line 1166 of file names.c.

4.65.2.28 AddFunction()

```
int AddFunction (
            UBYTE * name,
            int comm,
            int istensor,
            int cplx,
            int symprop,
            int dim,
            int argmax,
            int argmin )
```

Definition at line 1225 of file names.c.

4.65.2.29 CoCommuteInSet()

```
int CoCommuteInSet (
            UBYTE * s )
```

Definition at line 1255 of file names.c.

4.65.2.30 CoFunction()

```
int CoFunction (
            UBYTE * s,
            int comm,
            int istensor )
```

Definition at line 1345 of file names.c.

4.65.2.31 CoNFunction()

```
int CoNFunction (
            UBYTE * s )
```

Definition at line 1523 of file names.c.

4.65.2.32 CoCFunction()

```
int CoCFunction (
            UBYTE * s )
```

Definition at line 1524 of file names.c.

4.65.2.33 CoNTensor()

```
int CoNTensor (
            UBYTE * s )
```

Definition at line 1525 of file names.c.

4.65.2.34 CoCTensor()

```
int CoCTensor (
            UBYTE * s )
```

Definition at line 1526 of file names.c.

4.65.2.35 DoTable()

```
int DoTable (
            UBYTE * s,
            int par )
```

Definition at line 1560 of file names.c.

4.65.2.36 CoTable()

```
int CoTable (
            UBYTE * s )
```

Definition at line 1923 of file names.c.

4.65.2.37 CoNTable()

```
int CoNTable (
            UBYTE * s )
```

Definition at line 1933 of file names.c.

4.65.2.38 CoCTable()

```
int CoCTable (
            UBYTE * s )
```

Definition at line 1943 of file names.c.

4.65.2.39 EmptyTable()

```
void EmptyTable (
            TABLES T )
```

Definition at line 1953 of file names.c.

4.65.2.40 AddSet()

```
int AddSet (
            UBYTE * name,
            WORD dim )
```

Definition at line 1997 of file names.c.

4.65.2.41 DoElements()

```
int DoElements (
            UBYTE * s,
            SETS set,
            UBYTE * name )
```

Definition at line 2029 of file names.c.

4.65.2.42 CoSet()

```
int CoSet (
            UBYTE * s )
```

Definition at line 2217 of file names.c.

4.65.2.43 DoTempSet()

```
int DoTempSet (
            UBYTE * from,
            UBYTE * to )
```

Definition at line 2290 of file names.c.

4.65.2.44 CoAuto()

```
int CoAuto (
            UBYTE * inp )
```

Definition at line 2382 of file names.c.

4.65.2.45 AddDollar()

```
int AddDollar (
            UBYTE * name,
            WORD type,
            WORD * start,
            LONG size )
```

Definition at line 2412 of file names.c.

4.65.2.46 ReplaceDollar()

```
int ReplaceDollar (
            WORD number,
            WORD newtype,
            WORD * newstart,
            LONG newsize )
```

Definition at line 2456 of file names.c.

4.65.2.47 AddDubious()

```
int AddDubious (
            UBYTE * name )
```

Definition at line 2489 of file names.c.

4.65.2.48 MakeDubious()

```
int MakeDubious (
            NAMETREE * nametree,
            UBYTE * name,
            WORD * number )
```

Definition at line 2503 of file names.c.

4.65.2.49 NameConflict()

```
int NameConflict (
            int type,
            UBYTE * name )
```

Definition at line 2538 of file names.c.

4.65.2.50 AddExpression()

```
int AddExpression (
            UBYTE * name,
            int x,
            int y )
```

Definition at line 2554 of file names.c.

4.65.2.51 GetLabel()

```
int GetLabel (
            UBYTE * name )
```

Definition at line 2596 of file names.c.

4.65.2.52 ResetVariables()

```
void ResetVariables (
            int par )
```

Definition at line 2641 of file names.c.

4.65.2.53 RemoveDollars()

```
void RemoveDollars ( )
```

Definition at line 2937 of file names.c.

4.65.2.54 Globalize()

```
void Globalize (
            int par )
```

Definition at line 2964 of file names.c.

4.65.2.55 TestName()

```
int TestName (
            UBYTE * name )
```

Definition at line 3063 of file names.c.

4.66 names.c

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035 /*
00036     #[ Includes :
00037 */
00038
00039 #include "form3.h"
00040
00041 /* EXTERNLOCK(dummylock) */
00042
00043 /*
00044     #] Includes :
00045
00046     #[ GetNode :
00047 */
00048
00049 NAMENODE *GetNode(NAMETREE *nametree, UBYTE *name)
00050 {
00051     NAMENODE *n;
00052     int node, newnode, i;
00053     if ( nametree->namenode == 0 ) return(0);
00054     newnode = nametree->headnode;
00055     do {
00056         node = newnode;
00057         n = nametree->namenode+node;
00058         if ( ( i = StrCmp(name,nametree->namebuffer+n->name) ) < 0 )
00059             newnode = n->left;
00060         else if ( i > 0 ) newnode = n->right;
00061         else { return(n); }
00062     } while ( newnode >= 0 );
```

```
    return(0);
}

/*
    #] GetNode :
    #[ AddName :
*/

int AddName(NAMETREE *nametree, UBYTE *name, WORD type, WORD number, int *nodenum)
{
    NAMENODE *n, *nn, *nnn;
    UBYTE *s, *ss, *sss;
    LONG *c1,*c2, j, newsize;
    int node, newnode, node3, r, rr = 0, i, retval = 0;
    if ( nametree->namenode == 0 ) {
        s = name; i = 1; while ( *s ) { i++; s++; }
        j = INITNAMESIZE;
        if ( i > j ) j = i;
        nametree->namenode = (NAMENODE *)Malloc1(INITNODESIZE*sizeof(NAMENODE),
            "new nametree in AddName");
        nametree->namebuffer = (UBYTE *)Malloc1(j,
            "new namebuffer in AddName");
        nametree->nodesize = INITNODESIZE;
        nametree->namesize = j;
        nametree->namefill = i;
        nametree->nodefill = 1;
        nametree->headnode = 0;
        n = nametree->namenode;
        n->parent = n->left = n->right = -1;
        n->balance = 0;
        n->type = type;
        n->number = number;
        n->name = 0;
        s = name;
        ss = nametree->namebuffer;
        while ( *s ) *ss++ = *s++;
        *ss = 0;
        *nodenum = 0;
        return(retval);
    }
    newnode = nametree->headnode;
    do {
        node = newnode;
        n = nametree->namenode+node;
        if ( StrCmp(name,nametree->namebuffer+n->name) < 0 ) {
            newnode = n->left; r = -1;
        }
        else {
            newnode = n->right; r = 1;
        }
    } while ( newnode >= 0 );
/*
    We are at the insertion point. Add the node.
*/
    if ( nametree->nodefill >= nametree->nodesize ) {   /* Double allocation */
        newsize = nametree->nodesize * 2;
        if ( newsize > MAXINNAMETREE ) newsize = MAXINNAMETREE;
        if ( nametree->nodefill >= MAXINNAMETREE ) {
            MesPrint("!!!More than %l names in one object",(LONG)MAXINNAMETREE);
            Terminate(-1);
        }
        nnn = (NAMENODE *)Malloc1(2*((LONG)newsize*sizeof(NAMENODE)),
            "extra names in AddName");
        c1 = (LONG *)nnn; c2 = (LONG *)nametree->namenode;
        i = (nametree->nodefill * sizeof(NAMENODE))/sizeof(LONG);
        while ( --i >= 0 ) *c1++ = *c2++;
        M_free(nametree->namenode,"nametree->namenode");
        nametree->namenode = nnn;
        nametree->nodesize = newsize;
        n = nametree->namenode+node;
    }
    *nodenum = newnode = nametree->nodefill++;
    nn = nametree->namenode+newnode;
    nn->parent = node;
    if ( r < 0 ) n->left = newnode; else n->right = newnode;
    nn->left = nn->right = -1;
    nn->type = type;
    nn->number = number;
    nn->balance = 0;
    i = 1; s = name; while ( *s ) { i++; s++; }
    while ( nametree->namefill + i >= nametree->namesize ) { /* Double alloc */
        sss = (UBYTE *)Malloc1(2*nametree->namesize,
            "extra names in AddName");
        s = sss; ss = nametree->namebuffer; j = nametree->namefill;
        while ( --j >= 0 ) *s++ = *ss++;
        M_free(nametree->namebuffer,"nametree->namebuffer");
        nametree->namebuffer = sss;
```

```
        nametree->namesize *= 2;
    }
    s = nametree->namebuffer+nametree->namefill;
    nn->name = nametree->namefill;
    retval = nametree->namefill;
    nametree->namefill += i;
    while ( *name ) *s++ = *name++;
    *s = 0;
/*
    Adjust the balance factors
*/
    while ( node >= 0 ) {
        n = nametree->namenode + node;
        if ( newnode == n->left ) rr = -1;
        else rr = 1;
        if ( n->balance == -rr ) { n->balance = 0; return(retval); }
        else if ( n->balance == rr ) break;
        n->balance = rr;
        newnode = node;
        node = n->parent;
    }
    if ( node < 0 ) return(retval);
/*
    We have to rebalance the tree. There are two basic operations.
    n/node is the unbalanced node. newnode is its child.
    rr is the old balance of n/node.
*/
    nn = nametree->namenode + newnode;
    if ( nn->balance == -rr ) {  /* The difficult case */
        if ( rr > 0 ) {
            node3 = nn->left;
            nnn = nametree->namenode + node3;
            nnn->parent = n->parent;
            n->parent = nn->parent = node3;
            if ( nnn->right >= 0 ) nametree->namenode[nnn->right].parent = newnode;
            if ( nnn->left >= 0 ) nametree->namenode[nnn->left].parent = node;
            n->right = nnn->left; nnn->left = node;
            nn->left = nnn->right; nnn->right = newnode;
            if ( nnn->balance > 0 ) { n->balance = -1; nn->balance = 0; }
            else if ( nnn->balance == 0 ) { n->balance = nn->balance = 0; }
            else                    { nn->balance = 1; n->balance = 0; }
        }
        else {
            node3 = nn->right;
            nnn = nametree->namenode + node3;
            nnn->parent = n->parent;
            n->parent = nn->parent = node3;
            if ( nnn->right >= 0 ) nametree->namenode[nnn->right].parent = node;
            if ( nnn->left >= 0 ) nametree->namenode[nnn->left].parent = newnode;
            n->left = nnn->right; nnn->right = node;
            nn->right = nnn->left; nnn->left = newnode;
            if ( nnn->balance < 0 ) { n->balance = 1; nn->balance = 0; }
            else if ( nnn->balance == 0 ) { n->balance = nn->balance = 0; }
            else                    { nn->balance = -1; n->balance = 0; }
        }
        nnn->balance = 0;
        if ( nnn->parent >= 0 ) {
            nn = nametree->namenode + nnn->parent;
            if ( node == nn->left ) nn->left = node3;
            else nn->right = node3;
        }
        if ( node == nametree->headnode ) nametree->headnode = node3;
    }
    else if ( nn->balance == rr ) {  /* The easy case */
        nn->parent = n->parent; n->parent = newnode;
        if ( rr > 0 ) {
            if ( nn->left >= 0 ) nametree->namenode[nn->left].parent = node;
            n->right = nn->left; nn->left = node;
        }
        else {
            if ( nn->right >= 0 ) nametree->namenode[nn->right].parent = node;
            n->left = nn->right; nn->right = node;
        }
        if ( nn->parent >= 0 ) {
            nnn = nametree->namenode + nn->parent;
            if ( node == nnn->left ) nnn->left = newnode;
            else nnn->right = newnode;
        }
        nn->balance = n->balance = 0;
        if ( node == nametree->headnode ) nametree->headnode = newnode;
    }
#ifdef DEBUGON
    else {  /* Cannot be. Code here for debugging only */
        MesPrint("We ran into an impossible case in AddName\n");
        DumpTree(nametree);
        Terminate(-1);
    }
```

```c
#endif
    return(retval);
}

/*
    #] AddName :
    #[ GetName :

    When AutoDeclare is an active statement.
    If par == WITHAUTO and the variable is not found we have to check:
    1: that nametree != AC.exprnames && nametree != AC.dollarnames
    2: check that the variable is not in AC.exprnames after all.
    3: call GetAutoName and return its values.
*/

int GetName(NAMETREE *nametree, UBYTE *name, WORD *number, int par)
{
    NAMENODE *n;
    int node, newnode, i;
    UBYTE *s, *t, *u;
    if ( nametree->namenode == 0 || nametree->namefill == 0 ) goto NotFound;
    newnode = nametree->headnode;
    do {
        node = newnode;
        n = nametree->namenode+node;
        if ( ( i = StrCmp(name,nametree->namebuffer+n->name) ) < 0 )
            newnode = n->left;
        else if ( i > 0 ) newnode = n->right;
        else {
            *number = n->number;
            return(n->type);
        }
    } while ( newnode >= 0 );
    s = name;
    while ( *s ) s++;
    if ( s > name && s[-1] == '_' && nametree == AC.varnames ) {
/*
            The Kronecker delta d_ is very special. It is not really a function.
*/
        if ( s == name+2 && ( *name == 'd' || *name == 'D' ) ) {
            *number = DELTA-FUNCTION;
            return(CDELTA);
        }
/*
            Test for N#_? type variables (summed indices)
*/
        if ( s > name+2 && *name == 'N' ) {
            t = name+1; i = 0;
            while ( FG.cTable[*t] == 1 ) i = 10*i + *t++ -'0';
            if ( s == t+1 ) {
                *number = i + AM.IndDum - AM.OffsetIndex;
                return(CINDEX);
            }
        }
/*
            Now test for any built in object
*/
        newnode = nametree->headnode;
        do {
            node = newnode;
            n = nametree->namenode+node;
            if ( ( i = StrHICmp(name,nametree->namebuffer+n->name) ) < 0 )
                newnode = n->left;
            else if ( i > 0 ) newnode = n->right;
            else {
                *number = n->number; return(n->type);
            }
        } while ( newnode >= 0 );
/*
            Now we test for the extra symbols of the type STR###_
            The string sits in AC.extrasym and is followed by digits.
            The name is only legal if the number is in the
            range 1,...,cbuf[AM.sbufnum].numrhs
*/
        t = name; u = AC.extrasym;
        while ( *t == *u ) { t++; u++; }
        if ( *u == 0 && *t != 0 ) { /* potential hit */
            WORD x = 0;
            while ( FG.cTable[*t] == 1 ) {
                x = 10*x + (*t++ - '0');
            }
            if ( *t == '_' && x > 0 && x <= cbuf[AM.sbufnum].numrhs ) { /* Hit */
                *number = MAXVARIABLES-x;
                return(CSYMBOL);
            }
        }
    }
```

```
NotFound:;
    if ( par != WITHAUTO || nametree == AC.autonames ) return(NAMENOTFOUND);
    return(GetAutoName(name,number));
}

/*
    #] GetName :
    #[ GetLastExprName :

    When AutoDeclare is an active statement.
    If par == WITHAUTO and the variable is not found we have to check:
    1: that nametree != AC.exprnames && nametree != AC.dollarnames
    2: check that the variable is not in AC.exprnames after all.
    3: call GetAutoName and return its values.
*/

int GetLastExprName(UBYTE *name, WORD *number)
{
    int i;
    EXPRESSIONS e;
    for ( i = NumExpressions; i > 0; i-- ) {
        e = Expressions+i-1;
        if ( StrCmp(AC.exprnames->namebuffer+e->name,name) == 0 ) {
            *number = i-1;
            return(1);
        }
    }
    return(0);
}

/*
    #] GetLastExprName :
    #[ GetOName :

    Adds the proper offsets, so we do not have to do that in the calling
    routine.
*/

int GetOName(NAMETREE *nametree, UBYTE *name, WORD *number, int par)
{
    int retval = GetName(nametree,name,number,par);
    switch ( retval ) {
        case CVECTOR: *number += AM.OffsetVector; break;
        case CINDEX:  *number += AM.OffsetIndex;  break;
        case CFUNCTION: *number += FUNCTION; break;
        default: break;
    }
    return(retval);
}

/*
    #] GetOName :
    #[ GetAutoName :

    This routine gets the automatic declarations
*/

int GetAutoName(UBYTE *name, WORD *number)
{
    UBYTE *s, c;
    int type;
    if ( GetName(AC.exprnames,name,number,NOAUTO) != NAMENOTFOUND )
                return(NAMENOTFOUND);
    s = name;
    while ( *s ) { s++; }
    if ( s[-1] == '_' ) {
        return(NAMENOTFOUND);
    }
    while ( s > name ) {
        c = *s; *s = 0;
        type = GetName(AC.autonames,name,number,NOAUTO);
        *s = c;
        switch(type) {
            case CSYMBOL: {
                SYMBOLS sym = ((SYMBOLS)(AC.AutoSymbolList.lijst)) + *number;
                *number = AddSymbol(name,sym->minpower,sym->maxpower,sym->complex,sym->dimension);
                return(type); }
            case CVECTOR: {
                VECTORS vec = ((VECTORS)(AC.AutoVectorList.lijst)) + *number;
                *number = AddVector(name,vec->complex,vec->dimension);
                return(type); }
            case CINDEX: {
                INDICES ind = ((INDICES)(AC.AutoIndexList.lijst)) + *number;
                *number = AddIndex(name,ind->dimension,ind->nmin4);
                return(type); }
            case CFUNCTION: {
                FUNCTIONS fun = ((FUNCTIONS)(AC.AutoFunctionList.lijst)) + *number;
```

```
                *number =
    AddFunction(name,fun->commute,fun->spec,fun->complex,fun->symmetric,fun->dimension,fun->maxnumargs,fun->minnumargs);
                return(type); }
            default:
                break;
        }
        s--;
    }
    return(NAMENOTFOUND);
}

/*
    #] GetAutoName :
    #[ GetVar :
*/

int GetVar(UBYTE *name, WORD *type, WORD *number, int wantedtype, int par)
{
    WORD funnum;
    int typ;
    if ( ( typ = GetName(AC.varnames,name,number,par) ) != wantedtype ) {
        if ( typ != NAMENOTFOUND ) {
            if ( wantedtype == -1 ) {
                *type = typ;
                return(1);
            }
            NameConflict(typ,name);
            MakeDubious(AC.varnames,name,&funnum);
            return(-1);
        }
        if ( ( typ = GetName(AC.exprnames,name,&funnum,par) ) != NAMENOTFOUND ) {
            if ( typ == wantedtype || wantedtype == -1 ) {
                *number = funnum; *type = typ; return(1);
            }
            NameConflict(typ,name);
            return(-1);
        }
        return(NAMENOTFOUND);
    }
    if ( typ == -1 ) { return(0); }
    *type = typ;
    return(1);
}

/*
    #] GetVar :
    #[ EntVar :
*/

WORD EntVar(WORD type, UBYTE *name, WORD x, WORD y, WORD z, WORD d)
{
    switch ( type ) {
        case CSYMBOL:
            return(AddSymbol(name,y,z,x,d));
            break;
        case CINDEX:
            return(AddIndex(name,x,z));
            break;
        case CVECTOR:
            return(AddVector(name,x,d));
            break;
        case CFUNCTION:
            return(AddFunction(name,y,z,x,0,d,-1,-1));
            break;
        case CSET:
            AC.SetList.numtemp++;
            return(AddSet(name,d));
            break;
        case CEXPRESSION:
            return(AddExpression(name,x,y));
            break;
        default:
            break;
    }
    return(-1);
}

/*
    #] EntVar :
    #[ GetDollar :
*/

int GetDollar(UBYTE *name)
{
    WORD number;
    if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) return(-1);
    return((int)number);
```

```
}

/*
    #] GetDollar :
    #[ DumpTree :
*/

VOID DumpTree(NAMETREE *nametree)
{
    if ( nametree->headnode >= 0
        && nametree->namebuffer && nametree->namenode ) {
        DumpNode(nametree,nametree->headnode,0);
    }
}

/*
    #] DumpTree :
    #[ DumpNode :
*/

VOID DumpNode(NAMETREE *nametree, WORD node, WORD depth)
{
    NAMENODE *n;
    int i;
    char *name;
    n = nametree->namenode + node;
    if ( n->left >= 0 ) DumpNode(nametree,n->left,depth+1);
    for ( i = 0; i < depth; i++ ) printf(" ");
    name = (char *)(nametree->namebuffer+n->name);
    printf("%s(%d): {%d}(%d)(%d)[%d]\n",
        name,node,n->parent,n->left,n->right,n->balance);
    if ( n->right >= 0 ) DumpNode(nametree,n->right,depth+1);
}

/*
    #] DumpNode :
    #[ CompactifyTree :
*/

int CompactifyTree(NAMETREE *nametree,WORD par)
{
    NAMETREE newtree;
    NAMENODE *n;
    LONG i, j, ns, k;
    UBYTE *s;

    for ( i = 0, j = 0, k = 0, n = nametree->namenode, ns = 0;
    i < nametree->nodefill; i++, n++ ) {
        if ( n->type != CDELETE ) {
            s = nametree->namebuffer+n->name;
            while ( *s ) { s++; ns++; }
            j++;
        }
        else k++;
    }
    if ( k == 0 ) return(0);
    if ( j == 0 ) {
        if ( nametree->namebuffer ) M_free(nametree->namebuffer,"nametree->namebuffer");
        if ( nametree->namenode ) M_free(nametree->namenode,"nametree->namenode");
        nametree->namebuffer = 0;
        nametree->namenode = 0;
        nametree->namesize = nametree->namefill =
        nametree->nodesize = nametree->nodefill =
        nametree->oldnamefill = nametree->oldnodefill = 0;
        nametree->globalnamefill = nametree->globalnodefill =
        nametree->clearnamefill = nametree->clearnodefill = 0;
        nametree->headnode = -1;
        return(0);
    }
    ns += j;
    if ( j < 10 ) j = 10;
    if ( ns < 100 ) ns = 100;
    newtree.namenode = (NAMENODE *)Malloc1(2*j*sizeof(NAMENODE),"compactify namestree");
    newtree.nodefill = 0; newtree.nodesize = 2*j;
    newtree.namebuffer = (UBYTE *)Malloc1(2*ns,"compactify namestree");
    newtree.namefill = 0; newtree.namesize = 2*ns;
    CopyTree(&newtree,nametree,nametree->headnode,par);
    newtree.namenode[newtree.nodefill»1].parent = -1;
    LinkTree(&newtree,(WORD)0,newtree.nodefill);
    newtree.headnode = newtree.nodefill » 1;
    M_free(nametree->namebuffer,"nametree->namebuffer");
    M_free(nametree->namenode,"nametree->namenode");
    nametree->namebuffer  = newtree.namebuffer;
    nametree->namenode    = newtree.namenode;
    nametree->namesize    = newtree.namesize;
    nametree->namefill    = newtree.namefill;
    nametree->nodesize    = newtree.nodesize;
```

```
    nametree->nodefill    = newtree.nodefill;
    nametree->oldnamefill = newtree.namefill;
    nametree->oldnodefill = newtree.nodefill;
    nametree->headnode    = newtree.headnode;

/*  DumpTree(nametree); */
    return(0);
}

/*
    #] CompactifyTree :
    #[ CopyTree :
*/

VOID CopyTree(NAMETREE *newtree, NAMETREE *oldtree, WORD node, WORD par)
{
    NAMENODE *n, *m;
    UBYTE *s, *t;
    n = oldtree->namenode+node;
    if ( n->left >= 0 ) CopyTree(newtree,oldtree,n->left,par);
    if ( n->type != CDELETE ) {
        m = newtree->namenode+newtree->nodefill;
        m->type = n->type;
        m->number = n->number;
        m->name = newtree->namefill;
        m->left = m->right = -1;
        m->balance = 0;
        switch ( n->type ) {
            case CSYMBOL:
                if ( par == AUTONAMES ) {
                    autosymbols[n->number].name = newtree->namefill;
                    autosymbols[n->number].node = newtree->nodefill;
                }
                else {
                    symbols[n->number].name = newtree->namefill;
                    symbols[n->number].node = newtree->nodefill;
                }
                break;
            case CINDEX :
                if ( par == AUTONAMES ) {
                    autoindices[n->number].name = newtree->namefill;
                    autoindices[n->number].node = newtree->nodefill;
                }
                else {
                    indices[n->number].name = newtree->namefill;
                    indices[n->number].node = newtree->nodefill;
                }
                break;
            case CVECTOR:
                if ( par == AUTONAMES ) {
                    autovectors[n->number].name = newtree->namefill;
                    autovectors[n->number].node = newtree->nodefill;
                }
                else {
                    vectors[n->number].name = newtree->namefill;
                    vectors[n->number].node = newtree->nodefill;
                }
                break;
            case CFUNCTION:
                if ( par == AUTONAMES ) {
                    autofunctions[n->number].name = newtree->namefill;
                    autofunctions[n->number].node = newtree->nodefill;
                }
                else {
                    functions[n->number].name = newtree->namefill;
                    functions[n->number].node = newtree->nodefill;
                }
                break;
            case CSET:
                Sets[n->number].name = newtree->namefill;
                Sets[n->number].node = newtree->nodefill;
                break;
            case CEXPRESSION:
                Expressions[n->number].name = newtree->namefill;
                Expressions[n->number].node = newtree->nodefill;
                break;
            case CDUBIOUS:
                Dubious[n->number].name = newtree->namefill;
                Dubious[n->number].node = newtree->nodefill;
                break;
            case CDOLLAR:
                Dollars[n->number].name = newtree->namefill;
                Dollars[n->number].node = newtree->nodefill;
                break;
            default:
                MesPrint("Illegal variable type in CopyTree: %d",n->type);
                break;
```

```
        }
        newtree->nodefill++;
        s = newtree->namebuffer + newtree->namefill;
        t = oldtree->namebuffer + n->name;
        while ( *t ) { *s++ = *t++; newtree->namefill++; }
        *s = 0; newtree->namefill++;
    }
    if ( n->right >= 0 ) CopyTree(newtree,oldtree,n->right,par);
}

/*
    #] CopyTree :
    #[ LinkTree :
*/

VOID LinkTree(NAMETREE *tree, WORD offset, WORD numnodes)
{
/*
    Makes the tree into a binary tree
*/
    int med,numleft,numright,medleft,medright;
    med = numnodes » 1;
    numleft = med;
    numright = numnodes - med - 1;
    medleft = numleft » 1;
    medright = ( numright » 1 ) + med + 1;
    if ( numleft > 0 ) {
        tree->namenode[offset+med].left = offset+medleft;
        tree->namenode[offset+medleft].parent = offset+med;
    }
    if ( numright > 0 ) {
        tree->namenode[offset+med].right = offset+medright;
        tree->namenode[offset+medright].parent = offset+med;
    }
    if ( numleft  > 0 ) LinkTree(tree,offset,numleft);
    if ( numright > 0 ) LinkTree(tree,offset+med+1,numright);
    while ( numleft && numright ) { numleft »= 1; numright »= 1; }
    if ( numleft ) tree->namenode[offset+med].balance = -1;
    else if ( numright ) tree->namenode[offset+med].balance = 1;
}

/*
    #] LinkTree :
    #[ MakeNameTree :
*/

NAMETREE *MakeNameTree()
{
    NAMETREE *n;
    n = (NAMETREE *)Malloc1(sizeof(NAMETREE),"new nametree");
    n->namebuffer = 0;
    n->namenode = 0;
    n->namesize = n->namefill = n->nodesize = n->nodefill =
    n->oldnamefill = n->oldnodefill = 0;
    n->globalnamefill = n->globalnodefill =
    n->clearnamefill = n->clearnodefill = 0;
    n->headnode = -1;
    return(n);
}

/*
    #] MakeNameTree :
    #[ FreeNameTree :
*/

VOID FreeNameTree(NAMETREE *n)
{
    if ( n ) {
        if ( n->namebuffer ) M_free(n->namebuffer,"nametree->namebuffer");
        if ( n->namenode ) M_free(n->namenode,"nametree->namenode");
        M_free(n,"nametree");
    }
}

/*
    #] FreeNameTree :

    #[ WildcardNames :
*/

void ClearWildcardNames()
{
    AC.NumWildcardNames = 0;
}

int AddWildcardName(UBYTE *name)
{
```

```
    GETIDENTITY
    int size = 0, tocopy, i;
    UBYTE *s = name, *t, *newbuffer;
    while ( *s ) { s++; size++; }
    for ( i = 0, t = AC.WildcardNames; i < AC.NumWildcardNames; i++ ) {
        s = name;
        while ( ( *s == *t ) && *s ) { s++; t++; }
        if ( *s == 0 && *t == 0 ) return(i+1);
        while ( *t ) t++;
        t++;
    }
    tocopy = t - AC.WildcardNames;
    if ( tocopy + size + 1 > AC.WildcardBufferSize ) {
        if ( AC.WildcardBufferSize == 0 ) {
            AC.WildcardBufferSize = size+1;
            if ( AC.WildcardBufferSize < 100 ) AC.WildcardBufferSize = 100;
        }
        else if ( size+1 >= AC.WildcardBufferSize ) {
            AC.WildcardBufferSize += size+1;
        }
        else {
            AC.WildcardBufferSize *= 2;
        }
        newbuffer = (UBYTE *)Malloc1((LONG)AC.WildcardBufferSize,"argument list names");
        t = newbuffer;
        if ( AC.WildcardNames ) {
            s = AC.WildcardNames;
            while ( tocopy > 0 ) { *t++ = *s++; tocopy--; }
            M_free(AC.WildcardNames,"AC.WildcardNames");
        }
        AC.WildcardNames = newbuffer;
        M_free(AT.WildArgTaken,"AT.WildArgTaken");
        AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
                ,"argument list names");
    }
    s = name;
    while ( *s ) *t++ = *s++;
    *t = 0;
    AC.NumWildcardNames++;
    return(AC.NumWildcardNames);
}

int GetWildcardName(UBYTE *name)
{
    UBYTE *s, *t;
    int i;
    for ( i = 0, t = AC.WildcardNames; i < AC.NumWildcardNames; i++ ) {
        s = name;
        while ( ( *s == *t ) && *s ) { s++; t++; }
        if ( *s == 0 && *t == 0 ) return(i+1);
        while ( *t ) t++;
        t++;
    }
    return(0);
}

/*
    #] WildcardNames :

    #[ AddSymbol :

    The actual addition. Special routine for additions 'on the fly'
*/

int AddSymbol(UBYTE *name, int minpow, int maxpow, int cplx, int dim)
{
    int nodenum, numsymbol = AC.Symbols->num;
    UBYTE *s = name;
    SYMBOLS sym = (SYMBOLS)FromVarList(AC.Symbols);
    bzero(sym,sizeof(struct SyMbOl));
    sym->name = AddName(*AC.activenames,name,CSYMBOL,numsymbol,&nodenum);
    sym->minpower = minpow;
    sym->maxpower = maxpow;
    sym->complex  = cplx;
    sym->flags    = 0;
    sym->node     = nodenum;
    sym->dimension= dim;
    while ( *s ) s++;
    sym->namesize = (s-name)+1;
    return(numsymbol);
}

/*
    #] AddSymbol :
    #[ CoSymbol :

    Symbol declarations.   name[#{R|I|C}][([min]:[max])]
```

```
    Note that we know already that the parentheses match properly
*/

int CoSymbol(UBYTE *s)
{
    int type, error = 0, minpow, maxpow, cplx, sgn, dim;
    WORD numsymbol;
    UBYTE *name, *oldc, c, cc;
    do {
        minpow = -MAXPOWER;
        maxpow =  MAXPOWER;
        cplx = 0;
        dim = 0;
        name = s;
        if ( ( s = SkipAName(s) ) == 0 ) {
IllForm:    MesPrint("&Illegally formed name in symbol statement");
            error = 1;
            s = SkipField(name,0);
            goto eol;
        }
        oldc = s; cc = c = *s; *s = 0;
        if ( TestName(name) ) { *s = c; goto IllForm; }
        if ( cc == '#' ) {
            s++;
                 if ( tolower(*s) == 'r' ) cplx = VARTYPENONE;
            else if ( tolower(*s) == 'c' ) cplx = VARTYPECOMPLEX;
            else if ( tolower(*s) == 'i' ) cplx = VARTYPEIMAGINARY;
            else if ( ( ( *s == '-' || *s == '+' || *s == '=' )
                 && ( s[1] >= '0' && s[1] <= '9' ) )
                 || ( *s >= '0' && *s <= '9' ) ) {
                LONG x;
                sgn = 0;
                if ( *s == '-' ) { sgn = VARTYPEMINUS; s++; }
                else if ( *s == '+' || *s == '=' ) { sgn = 0; s++; }
                x = *s -'0';
                while ( s[1] >= '0' && s[1] <= '9' ) {
                    x = 10*x + (s[1] - '0'); s++;
                }
                if ( x >= MAXPOWER || x <= 1 ) {
                    MesPrint("&Illegal value for root of unity %s",name);
                    error = 1;
                }
                else {
                    maxpow = x;
                }
                cplx = VARTYPEROOTOFUNITY | sgn;
            }
            else {
                MesPrint("&Illegal specification for complexity of symbol %s",name);
                *oldc = c;
                error = 1;
                s = SkipField(s,0);
                goto eol;
            }
            s++; cc = *s;
        }
        if ( cc == '{' ) {
            s++;
            if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
                s += 2;
                if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
                    ParseSignedNumber(dim,s)
                    if ( dim < -HALFMAX || dim > HALFMAX ) {
                        MesPrint("&Warning: dimension of %s (%d) out of range"
                        ,name,dim);
                    }
                }
                if ( *s != '}' ) goto IllDim;
                else s++;
            }
            else {
IllDim:         MesPrint("&Error: Illegal dimension field for variable %s",name);
                error = 1;
                s = SkipField(s,0);
                goto eol;
            }
            cc = *s;
        }
        if ( cc == '(' ) {
            if ( ( cplx & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
                MesPrint("&Root of unity property for %s cannot be combined with power
    restrictions",name);
                error = 1;
            }
            s++;
            if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
                ParseSignedNumber(minpow,s)
```

```
                if ( minpow < -MAXPOWER ) {
                    minpow = -MAXPOWER;
                    if ( AC.WarnFlag )
                    MesPrint("&Warning: minimum power of %s corrected to %d"
                    ,name,-MAXPOWER);
                }
            }
            if ( *s != ':' ) {
skippar:        error = 1;
                s = SkipField(s,1);
                goto eol;
            }
            else s++;
            if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
                ParseSignedNumber(maxpow,s)
                if ( maxpow > MAXPOWER ) {
                    maxpow = MAXPOWER;
                    if ( AC.WarnFlag )
                    MesPrint("&Warning: maximum power of %s corrected to %d"
                    ,name,MAXPOWER);
                }
            }
            if ( *s != ')' ) goto skippar;
            s++;
        }
        if ( ( AC.AutoDeclareFlag == 0 &&
         ( ( type = GetName(AC.exprnames,name,&numsymbol,NOAUTO) )
                != NAMENOTFOUND ) )
        || ( ( type = GetName(*(AC.activenames),name,&numsymbol,NOAUTO) ) != NAMENOTFOUND ) ) {
            if ( type != CSYMBOL ) error = NameConflict(type,name);
            else {
                SYMBOLS sym = (SYMBOLS)(AC.Symbols->lijst) + numsymbol;
                if ( ( numsymbol == AC.lPolyFunVar ) && ( AC.lPolyFunType > 0 )
                     && ( AC.lPolyFun != 0 ) && ( minpow > -MAXPOWER || maxpow < MAXPOWER ) ) {
                    MesPrint("&The symbol %s is used by power expansions in the PolyRatFun!",name);
                    error = 1;
                }
                sym->complex  = cplx;
                sym->minpower = minpow;
                sym->maxpower = maxpow;
                sym->dimension= dim;
            }
        }
        else {
            AddSymbol(name,minpow,maxpow,cplx,dim);
        }
        *oldc = c;
eol:    while ( *s == ',' ) s++;
    } while ( *s );
    return(error);
}

/*
    #] CoSymbol :
    #[ AddIndex :

    The actual addition. Special routine for additions 'on the fly'
*/

int AddIndex(UBYTE *name, int dim, int dim4)
{
    int nodenum, numindex = AC.Indices->num;
    INDICES ind = (INDICES)FromVarList(AC.Indices);
    UBYTE *s = name;
    bzero(ind,sizeof(struct InDeX));
    ind->name      = AddName(*AC.activenames,name,CINDEX,numindex,&nodenum);
    ind->type      = 0;
    ind->dimension = dim;
    ind->flags     = 0;
    ind->nmin4     = dim4;
    ind->node      = nodenum;
    while ( *s ) s++;
    ind->namesize = (s-name)+1;
    return(numindex);
}

/*
    #] AddIndex :
    #[ CoIndex :

    Index declarations. name[={number|symbol[:othersymbol]}]
*/

int CoIndex(UBYTE *s)
{
    int type, error = 0, dim, dim4;
    WORD numindex;
```

```
    UBYTE *name, *oldc, c;
    do {
        dim = AC.lDefDim;
        dim4 = AC.lDefDim4;
        name = s;
        if ( ( s = SkipAName(s) ) == 0 ) {
IllForm:    MesPrint("&Illegally formed name in index statement");
            error = 1;
            s = SkipField(name,0);
            goto eol;
        }
        oldc = s; c = *s; *s = 0;
        if ( TestName(name) ) { *s = c; goto IllForm; }
        if ( c == '=' ) {
            s++;
            if ( ( s = DoDimension(s,&dim,&dim4) ) == 0 ) {
                *oldc = c;
                error = 1;
                s = SkipField(name,0);
                goto eol;
            }
        }
        if ( ( AC.AutoDeclareFlag == 0 &&
        ( ( type = GetName(AC.exprnames,name,&numindex,NOAUTO) )
         != NAMENOTFOUND ) )
        || ( ( type = GetName(*(AC.activenames),name,&numindex,NOAUTO) ) != NAMENOTFOUND ) ) {
            if ( type != CINDEX ) error = NameConflict(type,name);
            else { /* reset the dimensions */
                indices[numindex].dimension = dim;
                indices[numindex].nmin4     = dim4;
            }
        }
        else AddIndex(name,dim,dim4);
        *oldc = c;
eol:    while ( *s == ',' ) s++;
    } while ( *s );
    return(error);
}

/*
    #] CoIndex :
    #[ DoDimension :
*/

UBYTE *DoDimension(UBYTE *s, int *dim, int *dim4)
{
    UBYTE c, *t = s;
    int type, error = 0;
    WORD numsymbol;
    NAMETREE **oldtree = AC.activenames;
    *dim4 = -NMIN4SHIFT;
    if ( FG.cTable[*s] == 1 ) {
retry:
        ParseNumber(*dim,s)
#if ( BITSINWORD/8 < 4 )
        if ( *dim >= (1 « (BITSINWORD-1)) ) goto illeg;
#endif
        *dim4 = *dim - 4;
        return(s);
    }
    else if ( ( (FG.cTable[*s] == 0 ) || ( *s == '[' ) )
        && ( s = SkipAName(s) ) != 0 ) {
        AC.activenames = &(AC.varnames);
        c = *s; *s = 0;
        if ( ( ( type = GetName(AC.exprnames,t,&numsymbol,NOAUTO) ) != NAMENOTFOUND )
        || ( ( type = GetName(AC.varnames,t,&numsymbol,WITHAUTO) ) != NAMENOTFOUND ) ) {
            if ( type != CSYMBOL ) error = NameConflict(type,t);
        }
        else {
            numsymbol = AddSymbol(t,-MAXPOWER,MAXPOWER,0,0);
            if ( *oldtree != AC.autonames && AC.WarnFlag )
            MesPrint("&Warning: Implicit declaration of %s as a symbol",t);
        }
        *dim = -numsymbol;
        if ( ( *s = c ) == ':' ) {
            s++;
            t = s;
            if ( ( s = SkipAName(s) ) == 0 ) goto illeg;
            if ( ( ( type = GetName(AC.exprnames,t,&numsymbol,NOAUTO) ) != NAMENOTFOUND )
            || ( ( type = GetName(AC.varnames,t,&numsymbol,WITHAUTO) ) != NAMENOTFOUND ) ) {
                if ( type != CSYMBOL ) error = NameConflict(type,t);
            }
            else {
                numsymbol = AddSymbol(t,-MAXPOWER,MAXPOWER,0,0);
                if ( *oldtree != AC.autonames && AC.WarnFlag )
                MesPrint("&Warning: Implicit declaration of %s as a symbol",t);
            }
```

```
            *dim4 = -numsymbol-NMIN4SHIFT;
        }
    }
    else if ( *s == '+' && FG.cTable[s[1]] == 1 ) {
        s++; goto retry;
    }
    else {
illeg:  MesPrint("&Illegal dimension specification. Should be number >= 0, symbol or symbol:symbol");
        return(0);
    }
    AC.activenames = oldtree;
    if ( error ) return(0);
    return(s);
}

/*
    #] DoDimension :
    #[ CoDimension :
*/

int CoDimension(UBYTE *s)
{
    s = DoDimension(s,&AC.lDefDim,&AC.lDefDim4);
    if ( s == 0 ) return(1);
    if ( *s != 0 ) {
        MesPrint("&Argument of dimension statement should be number >= 0, symbol or symbol:symbol");
        return(1);
    }
    return(0);
}

/*
    #] CoDimension :
    #[ AddVector :

    The actual addition. Special routine for additions 'on the fly'
*/

int AddVector(UBYTE *name, int cplx, int dim)
{
    int nodenum, numvector = AC.Vectors->num;
    VECTORS v = (VECTORS)FromVarList(AC.Vectors);
    UBYTE *s = name;
    bzero(v,sizeof(struct VeCtOr));
    v->name = AddName(*AC.activenames,name,CVECTOR,numvector,&nodenum);
    v->complex = cplx;
    v->node    = nodenum;
    v->dimension = dim;
    v->flags = 0;
    while ( *s ) s++;
    v->namesize = (s-name)+1;
    return(numvector);
}

/*
    #] AddVector :
    #[ CoVector :

    Vector declarations. The descriptor string is "(,%n)"
*/

int CoVector(UBYTE *s)
{
    int type, error = 0, dim;
    WORD numvector;
    UBYTE *name, c, *endname;
    do {
        name = s;
        dim = 0;
        if ( ( s = SkipAName(s) ) == 0 ) {
IllForm:    MesPrint("&Illegally formed name in vector statement");
            error = 1;
            s = SkipField(s,0);
        }
        else {
            c = *s; *s = 0, endname = s;
            if ( TestName(name) ) { *s = c; goto IllForm; }
            if ( c == '{' ) {
                s++;
                if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
                    s += 2;
                    if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
                        ParseSignedNumber(dim,s)
                        if ( dim < -HALFMAX || dim > HALFMAX ) {
                            MesPrint("&Warning: dimension of %s (%d) out of range"
                            ,name,dim);
                        }
```

```
                    }
                    if ( *s != '}' ) goto IllDim;
                    else s++;
                }
                else {
IllDim:             MesPrint("&Error: Illegal dimension field for variable %s",name);
                    error = 1;
                    s = SkipField(s,0);
                    while ( *s == ',' ) s++;
                    continue;
                }
            }
            if ( ( AC.AutoDeclareFlag == 0 &&
             ( ( type = GetName(AC.exprnames,name,&numvector,NOAUTO) )
             != NAMENOTFOUND ) )
            || ( ( type = GetName(*(AC.activenames),name,&numvector,NOAUTO) ) != NAMENOTFOUND ) ) {
                if ( type != CVECTOR ) error = NameConflict(type,name);
            }
            else AddVector(name,0,dim);
            *endname = c;
        }
        while ( *s == ',' ) s++;
    } while ( *s );
    return(error);
}

/*
    #] CoVector :
    #[ AddFunction :

    The actual addition. Special routine for additions 'on the fly'
*/

int AddFunction(UBYTE *name, int comm, int istensor, int cplx, int symprop, int dim, int argmax, int argmin)
{
    int nodenum, numfunction = AC.Functions->num;
    FUNCTIONS fun = (FUNCTIONS)FromVarList(AC.Functions);
    UBYTE *s = name;
    bzero(fun,sizeof(struct FuNcTiOn));
    fun->name = AddName(*AC.activenames,name,CFUNCTION,numfunction,&nodenum);
    fun->commute = comm;
    fun->spec = istensor;
    fun->complex  = cplx;
    fun->tabl = 0;
    fun->flags = 0;
    fun->node = nodenum;
    fun->symminfo = 0;
    fun->symmetric = symprop;
    fun->dimension = dim;
    fun->maxnumargs = argmax;
    fun->minnumargs = argmin;
    while ( *s ) s++;
    fun->namesize = (s-name)+1;
    return(numfunction);
}

/*
    #] AddFunction :
    #[ CoCommuteInSet :

    Commuting,f1,...,fn;
*/

int CoCommuteInSet(UBYTE *s)
{
    UBYTE *name, *ss, c, *start = s;
    WORD number, type, *g, *gg;
    int error = 0, i, len = StrLen(s), len2 = 0;
    if ( AC.CommuteInSet != 0 ) {
        g = AC.CommuteInSet;
        while ( *g ) g += *g;
        len2 = g - AC.CommuteInSet;
        if ( len2+len+3 > AC.SizeCommuteInSet ) {
            gg = (WORD *)Malloc1((len2+len+3)*sizeof(WORD),"CommuteInSet");
            for ( i = 0; i < len2; i++ ) gg[i] = AC.CommuteInSet[i];
            gg[len2] = 0;
            M_free(AC.CommuteInSet,"CommuteInSet");
            AC.CommuteInSet = gg;
            AC.SizeCommuteInSet = len+len2+3;
            g = AC.CommuteInSet+len2;
        }
    }
    else {
        AC.SizeCommuteInSet = len+2;
        g = AC.CommuteInSet = (WORD *)Malloc1((len+3)*sizeof(WORD),"CommuteInSet");
        *g = 0;
```

```c
    }
    gg = g++;
    ss = s-1;
    for(;;) {
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
        if ( *s == 0 ) {
            if ( s - start >= len ) break;
            *s = '}'; s++;
            *g = 0;
            *gg = g-gg;
            if ( *gg < 2 ) {
                MesPrint("&There should be at least two noncommuting functions or tensors in a commuting statement.");
                error = 1;
            }
            else if ( *gg == 2 ) {
                gg[2] = gg[1]; gg[3] = 0; gg[0] = 3;
            }
            gg = g++;
            continue;
        }
        if ( s > ss ) {
            if ( *s != '{' ) {
                MesPrint("&The CommuteInSet statement should have sets enclosed in {}.");
                error = 1;
                break;
            }
            ss = s;
            SKIPBRA2(ss) /* Note that parentheses were tested before */
            *ss = 0;
            s++;
        }
        name = s;
        s = SkipAName(s);
        c = *s; *s = 0;
        if ( ( type = GetName(AC.varnames,name,&number,NOAUTO) ) != CFUNCTION ) {
            MesPrint("&%s is not a function or tensor",name);
            error = 1;
        }
        else if ( functions[number].commute == 0 ){
            MesPrint("&%s is not a noncommuting function or tensor",name);
            error = 1;
        }
        else {
            *g++ = number+FUNCTION;
            functions[number].flags |= COULDCOMMUTE;
            if ( number+FUNCTION >= GAMMA && number+FUNCTION <= GAMMASEVEN ) {
                functions[GAMMA-FUNCTION].flags |= COULDCOMMUTE;
                functions[GAMMAI-FUNCTION].flags |= COULDCOMMUTE;
                functions[GAMMAFIVE-FUNCTION].flags |= COULDCOMMUTE;
                functions[GAMMASIX-FUNCTION].flags |= COULDCOMMUTE;
                functions[GAMMASEVEN-FUNCTION].flags |= COULDCOMMUTE;
            }
        }
        *s = c;
    }
    return(error);
}

/*
    #] CoCommuteInSet :
    #[ CoFunction + ...:

    Function declarations.
    The second parameter indicates commutation properties.
    The third parameter tells whether we have a tensor.
*/

int CoFunction(UBYTE *s, int comm, int istensor)
{
    int type, error = 0, cplx, symtype, dim, argmax, argmin;
    WORD numfunction, reverseorder = 0, addone;
    UBYTE *name, *oldc, *par, c, cc;
    do {
        symtype = cplx = 0, argmin = argmax = -1;
        dim = 0;
        name = s;
        if ( ( s = SkipAName(s) ) == 0 ) {
IllForm:    MesPrint("&Illegally formed function/tensor name");
            error = 1;
            s = SkipField(name,0);
            goto eol;
        }
        oldc = s; cc = c = *s; *s = 0;
        if ( TestName(name) ) { *s = c; goto IllForm; }
        if ( c == '#' ) {
            s++;
```

```
                 if ( tolower(*s) == 'r' ) cplx = VARTYPENONE;
            else if ( tolower(*s) == 'c' ) cplx = VARTYPECOMPLEX;
            else if ( tolower(*s) == 'i' ) cplx = VARTYPEIMAGINARY;
            else {
                MesPrint("&Illegal specification for complexity of %s",name);
                *oldc = c;
                error = 1;
                s = SkipField(s,0);
                goto eol;
            }
            s++; cc = *s;
        }
        if ( cc == '{' ) {
            s++;
            if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
                s += 2;
                if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
                    ParseSignedNumber(dim,s)
                    if ( dim < -HALFMAX || dim > HALFMAX ) {
                        MesPrint("&Warning: dimension of %s (%d) out of range"
                        ,name,dim);
                    }
                }
                if ( *s != '}' ) goto IllDim;
                else s++;
            }
            else {
IllDim:         MesPrint("&Error: Illegal dimension field for variable %s",name);
                error = 1;
                s = SkipField(s,0);
                goto eol;
            }
            cc = *s;
        }
        if ( cc == '(' ) {
            s++;
            if ( *s == '-' ) {
                reverseorder = REVERSEORDER;
                s++;
            }
            else {
                reverseorder = 0;
            }
            par = s;
            while ( FG.cTable[*s] == 0 ) s++;
            cc = *s; *s = 0;
            if ( s <= par ) {
illegsym:       *s = cc;
                MesPrint("&Illegal specification for symmetry of %s",name);
                *oldc = c;
                error = 1;
                s = SkipField(s,1);
                goto eol;
            }
            if ( StrICont(par,(UBYTE *)"symmetric") == 0 ) symtype = SYMMETRIC;
            else if ( StrICont(par,(UBYTE *)"antisymmetric") == 0 ) symtype = ANTISYMMETRIC;
            else if ( ( StrICont(par,(UBYTE *)"cyclesymmetric") == 0 )
            || ( StrICont(par,(UBYTE *)"cyclic") == 0 ) ) symtype = CYCLESYMMETRIC;
            else if ( ( StrICont(par,(UBYTE *)"rcyclesymmetric") == 0 )
            || ( StrICont(par,(UBYTE *)"rcyclic") == 0 )
            || ( StrICont(par,(UBYTE *)"reversecyclic") == 0 ) ) symtype = RCYCLESYMMETRIC;
            else goto illegsym;
            *s = cc;
            if ( *s != ')' || ( s[1] && s[1] != ',' && s[1] != '<' ) ) {
                Warning("&Excess information in symmetric properties currently ignored");
                s = SkipField(s,1);
            }
            else s++;
            symtype |= reverseorder;
            cc = *s;
        }
retry:;
        if ( cc == '<' ) {
            s++; addone = 0;
            if ( *s == '=' ) { addone++; s++; }
            argmax = 0;
            while ( FG.cTable[*s] == 1 ) { argmax = 10*argmax + *s++ - '0'; }
            argmax += addone;
            par = s;
            while ( FG.cTable[*s] == 0 ) s++;
            if ( s > par ) {
                cc = *s; *s = 0;
                if ( ( StrICont(par,(UBYTE *)"arguments") == 0 )
                || ( StrICont(par,(UBYTE *)"args") == 0 ) ) {}
                else {
                    Warning("&Illegal information in number of arguments properties currently
ignored");
```

```
                    error = 1;
                }
                *s = cc;
            }
            if ( argmax <= 0 ) {
                MesPrint("&Error: Cannot have fewer than 0 arguments for variable %s",name);
                error = 1;
            }
            cc = *s;
        }
        if ( cc == '>' ) {
            s++; addone = 1;
            if ( *s == '=' ) { addone = 0; s++; }
            argmin = 0;
            while ( FG.cTable[*s] == 1 ) { argmin = 10*argmin + *s++ - '0'; }
            argmin += addone;
            par = s;
            while ( FG.cTable[*s] == 0 ) s++;
            if ( s > par ) {
                cc = *s; *s = 0;
                if ( ( StrICont(par,(UBYTE *)"arguments") == 0 )
                || ( StrICont(par,(UBYTE *)"args") == 0 ) ) {}
                else {
                    Warning("&Illegal information in number of arguments properties currently ignored");
                    error = 1;
                }
                *s = cc;
            }
            cc = *s;
        }
        if ( cc == '<' ) goto retry;
        if ( ( AC.AutoDeclareFlag == 0 &&
         ( ( type = GetName(AC.exprnames,name,&numfunction,NOAUTO) )
         != NAMENOTFOUND ) )
        || ( ( type = GetName(*(AC.activenames),name,&numfunction,NOAUTO) ) != NAMENOTFOUND ) ) {
            if ( type != CFUNCTION ) error = NameConflict(type,name);
            else {
/*              FUNCTIONS fun = (FUNCTIONS)(AC.Functions->lijst) + numfunction-FUNCTION; */
                FUNCTIONS fun = (FUNCTIONS)(AC.Functions->lijst) + numfunction;

                if ( fun->tabl != 0 ) {
                    MesPrint("&Illegal attempt to change table into function");
                    error = 1;
                }

                fun->complex = cplx;
                fun->commute = comm;
                if ( istensor && fun->spec == 0 ) {
                    MesPrint("&Function %s changed to tensor",name);
                    error = 1;
                }
                else if ( istensor == 0 && fun->spec ) {
                    MesPrint("&Tensor %s changed to function",name);
                    error = 1;
                }
                fun->spec = istensor;
                if ( fun->symmetric != symtype ) {
                    fun->symmetric = symtype;
                    AC.SymChangeFlag = 1;
                }
                fun->maxnumargs = argmax;
                fun->minnumargs = argmin;
            }
        }
        else {
            AddFunction(name,comm,istensor,cplx,symtype,dim,argmax,argmin);
        }
        *oldc = c;
eol:    while ( *s == ',' ) s++;
    } while ( *s );
    return(error);
}

int CoNFunction(UBYTE *s) { return(CoFunction(s,1,0)); }
int CoCFunction(UBYTE *s) { return(CoFunction(s,0,0)); }
int CoNTensor(UBYTE *s) { return(CoFunction(s,1,2)); }
int CoCTensor(UBYTE *s) { return(CoFunction(s,0,2)); }

/*
    #] CoFunction + ...:
    #[ DoTable :

        Syntax:
        Table [check] [strict|relax] [zerofill] name(:1:2,...,regular arguments);
        name must be the name of a regular function.
        the table indices must be the first arguments.
```

```
        The parenthesis indicates 'name' as opposed to the options.

        We leave behind:
        a struct tabl in the FUNCTION struct
        Regular table:
            an array tablepointers for the pointers to elements of rhs
            in the compiler struct cbuf[T->bufnum]
            an array MINMAX T->mm with the minima and maxima
            a prototype array
            an offset in the compiler buffer for the pattern to be matched
        Sparse table:
            Just the number of dimensions
            We will keep track of the number of defined elements in totind
            and in tablepointers we will have numind+1 positions for each
            element. The first numind elements for the indices and the
            last one for the element in cbuf[T->bufnum].rhs

        Complication: to preserve speed we need a prototype and a pattern
        for each thread when we use WITHPTHREADS. This is because we write
        into those when looking for the pattern.
*/

static int nwarntab = 1;

int DoTable(UBYTE *s, int par)
{
    GETIDENTITY
    UBYTE *name, *p, *inp, c;
    int i, j, k, sparseflag = 0, rflag = 0, checkflag = 0;
    int error = 0, ret, oldcbufnum, oldEside;
    WORD funnum, type, *OldWork, *w, *ww, *t, *tt, *flags1, oldnumrhs,oldnumlhs;
    LONG oldcpointer;
    MINMAX *mm, *mm1;
    LONG x, y;
    TABLES T;
    CBUF *C;

    while ( *s == ',' ) s++;
    do {
        name = s;
        if ( ( s = SkipAName(s) ) == 0 ) {
IllForm:    MesPrint("&Illegal name or option in table declaration");
            return(1);
        }
        c = *s; *s = 0;
        if ( TestName(name) ) { *s = c; goto IllForm; }
        *s = c;
        if ( *s == '(' ) break;
        if ( *s != ',' ) {
            MesPrint("&Illegal definition of table");
            return(1);
        }
        *s = 0;
/*
        Secondary options
*/
        if      ( StrICmp(name,(UBYTE *)("check" )) == 0 ) checkflag = 1;
        else if ( StrICmp(name,(UBYTE *)("zero"  )) == 0 ) checkflag = 2;
        else if ( StrICmp(name,(UBYTE *)("one"   )) == 0 ) checkflag = 3;
        else if ( StrICmp(name,(UBYTE *)("strict")) == 0 ) rflag = 1;
        else if ( StrICmp(name,(UBYTE *)("relax" )) == 0 ) rflag = -1;
        else if ( StrICmp(name,(UBYTE *)("zerofill" )) == 0 ) { rflag = -2; checkflag = 2; }
        else if ( StrICmp(name,(UBYTE *)("onefill" )) == 0 ) { rflag = -3; checkflag = 3; }
        else if ( StrICmp(name,(UBYTE *)("sparse")) == 0 ) sparseflag |= 1;
        else if ( StrICmp(name,(UBYTE *)("base")) == 0 ) sparseflag |= 3;
        else if ( StrICmp(name,(UBYTE *)("tablebase")) == 0 ) sparseflag |= 3;
        else {
            MesPrint("&Illegal option in table definition: '%s'",name);
            error = 1;
        }
        *s++ = ',';
        while ( *s == ',' ) s++;
    } while ( *s );
    if ( name == s || *s == 0 ) {
        MesPrint("&Illegal name or option in table declaration");
        return(1);
    }
    *s = 0;     /* *s could only have been a parenthesis */
    if ( sparseflag ) {
        if ( checkflag == 1 ) rflag = 0;
        else if ( checkflag == 2 ) rflag = -2;
        else if ( checkflag == 3 ) rflag = -3;
        else rflag = -1;
    }
    if ( ( ret = GetVar(name,&type,&funnum,CFUNCTION,NOAUTO) ) ==
                    NAMENOTFOUND ) {
        if ( par == 0 ) {
```

```c
            funnum = EntVar(CFUNCTION,name,0,1,0,0);
        }
        else if ( par == 1 || par == 2 ) {
            funnum = EntVar(CFUNCTION,name,0,0,0,0);
        }
    }
    else if ( ret <= 0 ) {
        funnum = EntVar(CFUNCTION,name,0,0,0,0);
        error = 1;
    }
    else {
        if ( par == 2 ) {
            if ( nwarntab ) {
                Warning("Table now declares its (commuting) function.");
                Warning("Earlier definition in Function statement obsolete. Please remove.");
                nwarntab = 0;
            }
        }
        else {
            error = 1;
            MesPrint("&(N)(C)Tables should not be declared previously");
        }
    }
    if ( functions[funnum].spec > 0 ) {
        MesPrint("&Tensors cannot become tables");
        return(1);
    }
    if ( functions[funnum].symmetric > 0 ) {
        MesPrint("&Functions with nontrivial symmetrization properties cannot become tables");
        return(1);
    }
    if ( functions[funnum].tabl ) {
        MesPrint("&Redefinition of an existing table is not allowed.");
        return(1);
    }
    functions[funnum].tabl = T = (TABLES)Malloc1(sizeof(struct TaBlEs),"table");
/*
    Next we find the size of the table (if it is not sparse)
*/
    T->defined = T->mdefined = 0; T->sparse = sparseflag; T->mm = 0; T->flags = 0;
    T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
    T->boomlijst = 0;
    T->strict = rflag;
    T->bounds = checkflag;
    T->bufnum = inicbufs();
    T->argtail = 0;
    T->spare = 0;
    T->bufferssize = 8;
    T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"Table buffers");
    T->buffersfill = 0;
    T->buffers[T->buffersfill++] = T->bufnum;
    T->mode = 0;
    T->numdummies = 0;
    mm = T->mm;
    T->numind = 0;
    if ( rflag > 0 ) AC.MustTestTable++;
    T->totind = 0;      /* Table hasn't been checked */

    p = s; *s = '(';
    if ( sparseflag ) {
/*
        First copy the tail, just in case we will construct a tablebase
        Note that we keep the ( to indicate a tail
        The actual arguments can be found after the comma. Before we have
        the dimension which the tablebase will need for consistency checking.
*/
        inp = p+1;
        SKIPBRA3(inp)
        c = *inp; *inp = 0;
        T->argtail = strDup1(p,"argtail");
        *inp = c;
/*
        Now the regular compilation
*/
        inp = p++;
        ParseNumber(x,p)
        if ( FG.cTable[p[-1]] != 1 || ( *p != ',' && *p != ')' ) ) {
            p = inp;
            MesPrint("&First argument in a sparse table must be a number of dimensions");
            error = 1;
            x = 1;
        }
        T->numind = x;
        T->mm = (MINMAX *)Malloc1(x*sizeof(MINMAX),"table dimensions");
        T->flags = (WORD *)Malloc1(x*sizeof(WORD),"table flags");
        mm = T->mm;
        inp = p;
```

```
        if ( *inp != ')' ) inp++;
        T->totind = 0;            /* At the moment there are this many */
        T->tablepointers = 0;
        T->reserved = 0;
    }
    else {
        T->numind = 0;
        T->totind = 1;
        for(;;) {   /* Read the dimensions as far as they can be recognized */
            inp = ++p;
            if ( FG.cTable[*p] != 1 && *p != '+' && *p != '-' ) break;
            ParseSignedNumber(x,p)
            if ( FG.cTable[p[-1]] != 1 || *p != ':' ) break;
            p++;
            ParseSignedNumber(y,p)
            if ( FG.cTable[p[-1]] != 1 || ( *p != ',' && *p != ')' ) ) {
                MesPrint("&Illegal dimension field in table declaration");
                return(1);
            }
            mm1 = (MINMAX *)Malloc1((T->numind+1)*sizeof(MINMAX),"table dimensions");
            flags1 = (WORD *)Malloc1((T->numind+1)*sizeof(WORD),"table flags");
            for ( i = 0; i < T->numind; i++ ) { mm1[i] = T->mm[i]; flags1[i] = T->flags[i]; }
            if ( T->mm ) M_free(T->mm,"table dimensions");
            if ( T->flags ) M_free(T->flags,"table flags");
            T->mm = mm1;
            T->flags = flags1;
            mm = T->mm + T->numind;
            mm->mini = x; mm->maxi = y;
            T->totind *= mm->maxi-mm->mini+1;
            T->numind++;
            if ( *p == ')' ) { inp = p; break; }
        }
        w = T->tablepointers
          = (WORD *)Malloc1(TABLEEXTENSION*sizeof(WORD)*(T->totind),"table pointers");
        i = T->totind;
        for ( i = TABLEEXTENSION*T->totind; i > 0; i-- ) *w++ = -1; /* means: undefined */
        for ( i = T->numind-1, x = 1; i >= 0; i-- ) {
            T->mm[i].size = x;    /* Defines increment in this dimension */
            x *= T->mm[i].maxi - T->mm[i].mini + 1;
        }
    }
/*
    Now we redo the 'function part' and send it to the compiler.
    The prototype has to be picked up properly.
*/
    AT.WorkPointer++; /* We needs one extra word later */
    OldWork = AT.WorkPointer;
    oldcbufnum = AC.cbufnum;
    AC.cbufnum = T->bufnum;
    C = cbuf+AC.cbufnum;
    oldcpointer = C->Pointer - C->Buffer;
    oldnumlhs = C->numlhs;
    oldnumrhs = C->numrhs;
    AddLHS(AC.cbufnum);
    while ( s >= name ) *--inp = *s--;
    w = AT.WorkPointer;
    AC.ProtoType = w;
    *w++ = SUBEXPRESSION;
    *w++ = SUBEXPSIZE;
    *w++ = 0;
    *w++ = 1;
    *w++ = AC.cbufnum;
    FILLSUB(w)
    AC.WildC = w;
    AC.NwildC = 0;
    AT.WorkPointer = w + 4*AM.MaxWildcards;
    if ( ( ret = CompileAlgebra(inp,LHSIDE,AC.ProtoType) ) < 0 ) {
        error = 1; goto FinishUp;
    }
    if ( AC.NwildC && SortWild(w,AC.NwildC) ) error = 1;
    w += AC.NwildC;
    i = w-OldWork;
    OldWork[1] = i;
/*
    Basically we have to pull this pattern through Generator in case
    there are functions inside functions, or parentheses.
    We have to temporarily disable the .tabl to avoid problems with
    TestSub.
    Essential: we need to start NewSort twice to avoid the PutOut routines.
    The ground pattern is sitting in C->numrhs, but it could be that it
    has subexpressions in it. Hence it has to be worked out as the lhs in
    id statements (in comexpr.c).
*/
    OldWork[2] = C->numrhs;
    *w++ = 1; *w++ = 1; *w++ = 3;
    OldWork[-1] = w-OldWork+1;
    AT.WorkPointer = w;
```

```c
    ww = C->rhs[C->numrhs];
    for ( j = 0; j < *ww; j++ ) w[j] = ww[j];
    AT.WorkPointer = w+*w;
    if ( *ww == 0 || ww[*ww] != 0 ) {
        MesPrint("&Illegal table pattern definition");
        AC.lhdollarflag = 0;
        error = 1;
    }
    if ( error ) goto FinishUp;

    if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) { error = 1; goto FinishUp; }
    AN.RepPoint = AT.RepCount + 1;
    AC.lhdollarflag = 0; oldEside = AR.Eside; AR.Eside = LHSIDE;
    AR.Cnumlhs = C->numlhs;
    functions[funnum].tabl = 0;
    if ( Generator(BHEAD w,C->numlhs) ) {
        functions[funnum].tabl = T;
        AR.Eside = oldEside;
        LowerSortLevel(); LowerSortLevel(); goto FinishUp;
    }
    functions[funnum].tabl = T;
    AR.Eside = oldEside;
    AT.WorkPointer = w;
    if ( EndSort(BHEAD w,0) < 0 ) { LowerSortLevel(); goto FinishUp; }
    if ( *w == 0 || *(w+*w) != 0 ) {
        MesPrint("&Irregular pattern in table definition");
        error = 1;
        goto FinishUp;
    }
    LowerSortLevel();
    if ( AC.lhdollarflag ) {
        MesPrint("&Unexpanded dollar variables are not allowed in table definition");
        error = 1;
        goto FinishUp;
    }
    AT.WorkPointer = ww = w + *w;
    if ( ww[-1] != 3 || ww[-2] != 1 || ww[-3] != 1 ) {
        MesPrint("&Coefficient of pattern in table definition should be 1.");
        error = 1;
        goto FinishUp;
    }
    AC.DumNum = 0;
/*
    Now we have to allocate space for prototype+pattern
    In the case of TFORM we need extra pointers, because each worker has its own
*/
    j = *w + T->numind*2-3;
#ifdef WITHPTHREADS
    { int n;
    T->prototypeSize = ((i+j)*sizeof(WORD)+2*sizeof(WORD *)) * AM.totalnumberofthreads;
    T->prototype = (WORD **)Malloc1(T->prototypeSize,"table prototype");
    T->pattern = T->prototype + AM.totalnumberofthreads;
    t = (WORD *)(T->pattern + AM.totalnumberofthreads);
    for ( n = 0; n < AM.totalnumberofthreads; n++ ) {
        T->prototype[n] = t;
        for ( k = 0; k < i; k++ ) *t++ = OldWork[k];
    }
    T->pattern[0] = t;
    j--; w++;
    w[1] += T->numind*2;
    for ( k = 0; k < FUNHEAD; k++ ) *t++ = *w++;
    j -= FUNHEAD;
    for ( k = 0; k < T->numind; k++ ) { *t++ = -SNUMBER; *t++ = 0; j -= 2; }
    for ( k = 0; k < j; k++ ) *t++ = *w++;
    if ( sparseflag ) T->pattern[0][1] = t - T->pattern[0];
    k = t - T->pattern[0];
    for ( n = 1; n < AM.totalnumberofthreads; n++ ) {
        T->pattern[n] = t; tt = T->pattern[0];
        for ( i = 0; i < k; i++ ) *t++ = *tt++;
    }
    }
#else
    T->prototypeSize = (i+j)*sizeof(WORD);
    T->prototype = (WORD *)Malloc1(T->prototypeSize, "table prototype");
    T->pattern = T->prototype + i;
    for ( k = 0; k < i; k++ ) T->prototype[k] = OldWork[k];
    t = T->pattern;
    j--; w++;
    w[1] += T->numind*2;
    for ( k = 0; k < FUNHEAD; k++ ) *t++ = *w++;
    j -= FUNHEAD;
    for ( k = 0; k < T->numind; k++ ) { *t++ = -SNUMBER; *t++ = 0; j -= 2; }
    for ( k = 0; k < j; k++ ) *t++ = *w++;
    if ( sparseflag ) T->pattern[1] = t - T->pattern;
#endif
/*
    At this point we can pop the compilerbuffer.
```

```
*/
    C->Pointer = C->Buffer + oldcpointer;
    C->numrhs = oldnumrhs;
    C->numlhs = oldnumlhs;
/*
    Now check whether wildcards get converted to dollars (for PARALLEL)
    We give a warning!
*/
#ifdef WITHPTHREADS
    t = T->prototype[0];
#else
    t = T->prototype;
#endif
    tt = t + t[1]; t += SUBEXPSIZE;
    while ( t < tt ) {
        if ( *t == LOADDOLLAR ) {
            Warning("The use of $-variable assignments in tables disables parallel\
 execution for the whole program.");
            AM.hparallelflag |= NOPARALLEL_TBLDOLLAR;
            AC.mparallelflag |= NOPARALLEL_TBLDOLLAR;
            AddPotModdollar(t[2]);
        }
        t += t[1];
    }
FinishUp:;
    AT.WorkPointer = OldWork - 1;
    AC.cbufnum = oldcbufnum;
    if ( T->sparse ) ClearTableTree(T);
    if ( ( sparseflag & 2 ) != 0 ) {
        if ( T->spare == 0 ) { SpareTable(T); }
    }
    return(error);
}

/*
    #] DoTable :
    #[ CoTable :
*/

int CoTable(UBYTE *s)
{
    return(DoTable(s,2));
}

/*
    #] CoTable :
    #[ CoNTable :
*/

int CoNTable(UBYTE *s)
{
    return(DoTable(s,0));
}

/*
    #] CoNTable :
    #[ CoCTable :
*/

int CoCTable(UBYTE *s)
{
    return(DoTable(s,1));
}

/*
    #] CoCTable :
    #[ EmptyTable :
*/

void EmptyTable(TABLES T)
{
    int j;
    if ( T->sparse ) ClearTableTree(T);
    if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
    T->boomlijst = 0;
    for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
        finishcbuf(T->buffers[j]);
    }
    if ( T->buffers ) M_free(T->buffers,"Table buffers");
    finishcbuf(T->bufnum);
    T->bufnum = inicbufs();
    T->bufferssize = 8;
    T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"Table buffers");
    T->buffersfill = 0;
    T->buffers[T->buffersfill++] = T->bufnum;
    T->defined = T->mdefined = 0; T->flags = 0;
    T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
```

```
    T->spare = 0; T->reserved = 0;
    if ( T->spare ) {
        TABLES TT = T->spare;
        if ( TT->mm ) M_free(TT->mm,"tableminmax");
        if ( TT->flags ) M_free(TT->flags,"tableflags");
        if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
        for (j = 0; j < TT->buffersfill; j++ ) {
            finishcbuf(TT->buffers[j]);
        }
        if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
        if ( TT->buffers ) M_free(TT->buffers,"Table buffers");
        M_free(TT,"table");
        SpareTable(T);
    }
    else {
        WORD *w = T->tablepointers;
        j = T->totind;
        for ( j = TABLEEXTENSION*T->totind; j > 0; j-- ) *w++ = -1; /* means: undefined */
    }
}

/*
    #] EmptyTable :
    #[ AddSet :
*/

int AddSet(UBYTE *name, WORD dim)
{
    int nodenum, numset = AC.SetList.num;
    SETS set = (SETS)FromVarList(&AC.SetList);
    UBYTE *s;
    if ( name ) {
        set->name = AddName(AC.varnames,name,CSET,numset,&nodenum);
        s = name;
        while ( *s ) s++;
        set->namesize = (s-name)+1;
        set->node  = nodenum;
    }
    else {
        set->name = 0;
        set->namesize = 0;
        set->node  = -1;
    }
    set->first =
    set->last  = AC.SetElementList.num; /* set has no elements yet */
    set->type  = -1;                    /* undefined as of yet */
    set->dimension = dim;
    return(numset);
}

/*
    #] AddSet :
    #[ DoElements :

    Remark (25-mar-2011): If the dimension has been set (dim != MAXPOSITIVE)
    we want to test dimensions. Numbers count as dimension zero?
*/

int DoElements(UBYTE *s, SETS set, UBYTE *name)
{
    int type, error = 0, x, sgn, i;
    WORD numset, *e;
    UBYTE c, *cname;
    while ( *s ) {
        if ( *s == ',' ) { s++; continue; }
        sgn = 0;
        while ( *s == '-' || *s == '+' ) { sgn ^= 1; s++; }
        cname = s;
        if ( FG.cTable[*s] == 0 || *s == '_' || *s == '[' ) {
            if ( ( s = SkipAName(s) ) == 0 ) {
                MesPrint("&Illegal name in set definition");
                return(1);
            }
            c = *s; *s = 0;
            if ( ( ( type = GetName(AC.exprnames,cname,&numset,NOAUTO) ) == NAMENOTFOUND )
            && ( ( type = GetOName(AC.varnames,cname,&numset,WITHAUTO) ) == NAMENOTFOUND ) ) {
                DUBIOUSV dv;
                int nodenum;
                MesPrint("&%s has not been declared",cname);
/*
                We enter a 'dubious' declaration to cut down on errors
*/
                numset = AC.DubiousList.num;
                dv = (DUBIOUSV)FromVarList(&AC.DubiousList);
                dv->name = AddName(AC.varnames,cname,CDUBIOUS,numset,&nodenum);
                dv->node = nodenum;
                set->type = type = CDUBIOUS;
```

```
                set->dimension = 0;
                error = 1;
            }
            if ( set->type == -1 ) {
                if ( type == CSYMBOL ) {
                    for ( i = set->first; i < set->last; i++ ) {
                        SetElements[i] += 2*MAXPOWER;
                    }
                }
                set->type = type;
            }
            if ( set->type != type && set->type != CDUBIOUS
            && type != CDUBIOUS ) {
                if ( set->type != CNUMBER || ( type != CSYMBOL
                && type != CINDEX ) ) {
                    MesPrint(
                    "&%s has not the same type as the other members of the set"
                    ,cname);
                    error = 1;
                    set->type = CDUBIOUS;
                }
                else {
                    if ( type == CSYMBOL ) {
                        for ( i = set->first; i < set->last; i++ ) {
                            SetElements[i] += 2*MAXPOWER;
                        }
                    }
                    set->type = type;
                }
            }
            if ( set->dimension != MAXPOSITIVE ) { /* Dimension check */
                switch ( set->type ) {
                    case CSYMBOL:
                        if ( symbols[numset].dimension != set->dimension ) {
                            MesPrint("&Dimension check failed in set %s, symbol %s",
                                VARNAME(Sets,(set-Sets)),
                                VARNAME(symbols,numset));
                            error = 1;
                            set->dimension = MAXPOSITIVE;
                        }
                        break;
                    case CVECTOR:
                        if ( vectors[numset-AM.OffsetVector].dimension != set->dimension ) {
                            MesPrint("&Dimension check failed in set %s, vector %s",
                                VARNAME(Sets,(set-Sets)),
                                VARNAME(vectors,(numset-AM.OffsetVector)));
                            error = 1;
                            set->dimension = MAXPOSITIVE;
                        }
                        break;
                    case CFUNCTION:
                        if ( functions[numset-FUNCTION].dimension != set->dimension ) {
                            MesPrint("&Dimension check failed in set %s, function %s",
                                VARNAME(Sets,(set-Sets)),
                                VARNAME(functions,(numset-FUNCTION)));
                            error = 1;
                        }
                        break;
                            set->dimension = MAXPOSITIVE;
                }
            }
            if ( sgn ) {
                if ( type != CVECTOR ) {
                    MesPrint("&Illegal use of - sign in set. Can use only with vector or number");
                    error = 1;
                }
/*
                numset = AM.OffsetVector - numset;
                numset |= SPECMASK;
                numset = AM.OffsetVector - numset;
*/
                numset -= WILDMASK;
            }
            *s = c;
            if ( name == 0 && *s == '?' ) {
                s++;
                switch ( set->type ) {
                    case CSYMBOL:
                        numset = -numset; break;
                    case CVECTOR:
                        numset += WILDOFFSET; break;
                    case CINDEX:
                        numset |= WILDMASK; break;
                    case CFUNCTION:
                        numset |= WILDMASK; break;
                }
                AC.wildflag = 1;
```

```
            }
/*
            Now add the element to the set.
*/
            e = (WORD *)FromVarList(&AC.SetElementList);
            *e = numset;
            (set->last)++;
        }
        else if ( FG.cTable[*s] == 1 ) {
            ParseNumber(x,s)
            if ( sgn ) x = -x;
            if ( x >= MAXPOWER || x <= -MAXPOWER ||
            ( set->type == CINDEX && ( x < 0 || x >= AM.OffsetIndex ) ) ) {
                MesPrint("&Illegal value for set element: %d",x);
                if ( AC.firstconstindex ) {
                MesPrint("&0 <= Fixed indices < ConstIndex(which is %d)",
                    AM.OffsetIndex-1);
                MesPrint("&For setting ConstIndex, read the chapter on the setup file");
                AC.firstconstindex = 0;
                }
                error = 1;
                x = 0;
            }
/*
            Check what is allowed with the type.
*/
            if ( set->type == -1 ) {
                if ( x < 0 || x >= AM.OffsetIndex ) {
                    for ( i = set->first; i < set->last; i++ ) {
                        SetElements[i] += 2*MAXPOWER;
                    }
                    set->type = CSYMBOL;
                }
                else set->type = CNUMBER;
            }
            else if ( set->type == CDUBIOUS ) {}
            else if ( set->type == CNUMBER && x < 0 ) {
                for ( i = set->first; i < set->last; i++ ) {
                    SetElements[i] += 2*MAXPOWER;
                }
                set->type = CSYMBOL;
            }
            else if ( set->type != CSYMBOL && ( x < 0 ||
            ( set->type != CINDEX && set->type != CNUMBER ) ) ) {
                MesPrint("&Illegal mixture of element types in set");
                error = 1;
                set->type = CDUBIOUS;
            }
/*
            Allocate an element
*/
            e = (WORD *)FromVarList(&AC.SetElementList);
            (set->last)++;
            if ( set->type == CSYMBOL ) *e = x + 2*MAXPOWER;
/*          else if ( set->type == CINDEX ) *e = x; */
            else *e = x;
        }
        else {
            MesPrint("&Illegal object in list of set elements");
            return(1);
        }
    }
    return(error);
}

/*
    #] DoElements :
    #[ CoSet :

    Set declarations.
*/

int CoSet(UBYTE *s)
{
    int type, error = 0;
    UBYTE *name, c, *ss;
    SETS set;
    WORD numberofset, dim = MAXPOSITIVE;
    name = s;
    if ( ( s = SkipAName(s) ) == 0 ) {
IllForm:MesPrint("&Illegal name for set");
        return(1);
    }
    c = *s; *s = 0;
    if ( TestName(name) ) goto IllForm;
    if ( ( ( type = GetName(AC.exprnames,name,&numberofset,NOAUTO) ) != NAMENOTFOUND )
    || ( ( type = GetName(AC.varnames,name,&numberofset,NOAUTO) ) != NAMENOTFOUND ) ) {
```

```
        if ( type != CSET ) NameConflict(type,name);
        else {
            MesPrint("&There is already a set with the name %s",name);
        }
        return(1);
    }
    if ( c == 0 ) {
        numberofset = AddSet(name,0);
        set = Sets + numberofset;
        return(0);      /* empty set */
    }
    *s = c; ss = s;
    if ( *s == '{' ) {
        s++;
        if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
            s += 2;
            if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
                ParseSignedNumber(dim,s)
                if ( dim < -HALFMAX || dim > HALFMAX ) {
                    MesPrint("&Warning: dimension of %s (%d) out of range"
                    ,name,dim);
                }
            }
            if ( *s != '}' ) goto IllDim;
            else s++;
        }
        else {
IllDim:     MesPrint("&Error: Illegal dimension field for set %s",name);
            error = 1;
            s = SkipField(s,0);
        }
        while ( *s == ',' ) s++;
    }
    c = *ss; *ss = 0;
    numberofset = AddSet(name,dim);
    *ss = c;
    set = Sets + numberofset;
    if ( *s != ':' ) {
        MesPrint("&Proper syntax is `Set name:elements'");
        return(1);
    }
    s++;
    error = DoElements(s,set,name);
    AC.SetList.numtemp = AC.SetList.num;
    AC.SetElementList.numtemp = AC.SetElementList.num;
    return(error);
}

/*
    #] CoSet :
    #[ DoTempSet :

        Gets a {} set definition and returns a set number if the set is
        properly structured. This number refers either to an already
        existing set, or to a set that is defined here.
        From and to refer to the contents. They exclude the {}.
*/

int DoTempSet(UBYTE *from, UBYTE *to)
{
    int i, num, j, sgn;
    WORD *e, *ep;
    UBYTE c;
    int setnum = AddSet(0,MAXPOSITIVE);
    SETS set = Sets + setnum, setp;
    set->name = -1;
    set->type = -1;
    c = *to; *to = 0;
    AC.wildflag = 0;
    while ( *from == ',' ) from++;
    if ( *from == '<' || *from == '>' ) {
        set->type = CRANGE;
        set->first = 3*MAXPOWER;
        set->last = -3*MAXPOWER;
        while ( *from == '<' || *from == '>' ) {
            if ( *from == '<' ) {
                j = 1; from++;
                if ( *from == '=' ) { from++; j++; }
            }
            else {
                j = -1; from++;
                if ( *from == '=' ) { from++; j--; }
            }
            sgn = 1;
            while ( *from == '-' || *from == '+' ) {
                if ( *from == '-' ) sgn = -sgn;
                from++;
```

```
            }
            ParseNumber(num,from)
            if ( *from && *from != ',' ) {
                MesPrint("&Illegal number in ranged set definition");
                return(-1);
            }
            if ( sgn < 0 ) num = -num;
            if ( num >= MAXPOWER || num <= -MAXPOWER ) {
                Warning("Value in ranged set too big. Adjusted to infinity.");
                if ( num > 0 ) num =  3*MAXPOWER;
                else           num = -3*MAXPOWER;
            }
            else if ( j ==  2 ) num += 2*MAXPOWER;
            else if ( j == -2 ) num -= 2*MAXPOWER;
            if ( j > 0 ) set->first = num;
            else set->last = num;
            while ( *from == ',' ) from++;
        }
        if ( *from ) {
            MesPrint("&Definition of ranged set contains illegal objects");
            return(-1);
        }
    }
    else if ( DoElements(from,set,(UBYTE *)0) != 0 ) {
        AC.SetElementList.num = set->first;
        AC.SetList.num--; *to = c;
        return(-1);
    }
    *to = c;
/*
    Now we have to test whether this set exists already.
*/
    num = set->last - set->first;
    for ( setp = Sets, i = 0; i < AC.SetList.num-1; i++, setp++ ) {
        if ( num != setp->last - setp->first ) continue;
        if ( set->type != setp->type ) continue;
        if ( set->type == CRANGE ) {
            if ( set->first == setp->first ) return(setp-Sets);
        }
        else {
            e = SetElements + set->first;
            ep = SetElements + setp->first;
            j = num;
            while ( --j >= 0 ) if ( *e++ != *ep++ ) break;
            if ( j < 0 ) {
                AC.SetElementList.num = set->first;
                AC.SetList.num--;
                return(setp - Sets);
            }
        }
    }
    return(setnum);
}

/*
    #] DoTempSet :
    #[ CoAuto :

    To prepare first:
        Use of the proper pointers in the various declaration routines
        Proper action in .store and .clear
*/

int CoAuto(UBYTE *inp)
{
    int retval;

    AC.Symbols = &(AC.AutoSymbolList);
    AC.Vectors = &(AC.AutoVectorList);
    AC.Indices = &(AC.AutoIndexList);
    AC.Functions = &(AC.AutoFunctionList);
    AC.activenames = &(AC.autonames);
    AC.AutoDeclareFlag = WITHAUTO;

    while ( *inp == ',' ) inp++;
    retval = CompileStatement(inp);

    AC.AutoDeclareFlag = 0;
    AC.Symbols = &(AC.SymbolList);
    AC.Vectors = &(AC.VectorList);
    AC.Indices = &(AC.IndexList);
    AC.Functions = &(AC.FunctionList);
    AC.activenames = &(AC.varnames);
    return(retval);
}

/*
```

```
    #] CoAuto :
    #[ AddDollar :

    The actual addition. Special routine for additions 'on the fly'
*/

int AddDollar(UBYTE *name, WORD type, WORD *start, LONG size)
{
    int nodenum, numdollar = AP.DollarList.num;
    WORD *s, *t;
    DOLLARS dol = (DOLLARS)FromVarList(&AP.DollarList);
    dol->name = AddName(AC.dollarnames,name,CDOLLAR,numdollar,&nodenum);
    dol->type = type;
    dol->node = nodenum;
    dol->zero = 0;
    dol->numdummies = 0;
#ifdef WITHPTHREADS
    dol->pthreadslockread = dummylock;
    dol->pthreadslockwrite = dummylock;
#endif
    dol->nfactors = 0;
    dol->factors = 0;
    AddRHS(AM.dbufnum,1);
    AddLHS(AM.dbufnum);
    if ( start && size > 0 ) {
        dol->size = size;
        dol->where =
        s = (WORD *)Malloc1((size+1)*sizeof(WORD),"$-variable contents");
        t = start;
        while ( --size >= 0 ) *s++ = *t++;
        *s = 0;
    }
    else { dol->where = &(AM.dollarzero); dol->size = 0; }
    cbuf[AM.dbufnum].rhs[numdollar] = dol->where;
    cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
    cbuf[AM.dbufnum].NumTerms[numdollar] = 0;

    return(numdollar);
}

/*
    #] AddDollar :
    #[ ReplaceDollar :

    Replacements of dollar variables can happen at any time.
    For debugging purposes we should have a tracing facility.

    Not in use????
*/

int ReplaceDollar(WORD number, WORD newtype, WORD *newstart, LONG newsize)
{
    int error = 0;
    DOLLARS dol = Dollars + number;
    WORD *s, *t;
    LONG i;
    dol->type = newtype;
    if ( dol->size == newsize && newsize > 0 && newstart ) {
        s = dol->where; t = newstart; i = newsize;
        while ( --i >= 0 ) { if ( *s++ != *t++ ) break; }
        if ( i < 0 ) return(0);
    }
    if ( dol->where && dol->where != &(dol->zero) ) {
        M_free(dol->where,"dollar->where"); dol->where = &(dol->zero); dol->size = 0;
    }
    if ( newstart && newsize > 0 ) {
        dol->size = newsize;
        dol->where =
        s = (WORD *)Malloc1((newsize+1)*sizeof(WORD),"$-variable contents");
        t = newstart; i = newsize;
        while ( --i >= 0 ) *s++ = *t++;
        *s = 0;
    }
    return(error);
}

/*
    #] ReplaceDollar :
    #[ AddDubious :

    This adds a variable of which we do not know the proper type.
*/

int AddDubious(UBYTE *name)
{
    int nodenum, numdubious = AC.DubiousList.num;
    DUBIOUSV dub = (DUBIOUSV)FromVarList(&AC.DubiousList);
```

```c
    dub->name = AddName(AC.varnames,name,CDUBIOUS,numdubious,&nodenum);
    dub->node = nodenum;
    return(numdubious);
}

/*
    #] AddDubious :
    #[ MakeDubious :
*/

int MakeDubious(NAMETREE *nametree, UBYTE *name, WORD *number)
{
    NAMENODE *n;
    int node, newnode, i;
    if ( nametree->namenode == 0 ) return(-1);
    newnode = nametree->headnode;
    do {
        node = newnode;
        n = nametree->namenode+node;
        if ( ( i = StrCmp(name,nametree->namebuffer+n->name) ) < 0 )
            newnode = n->left;
        else if ( i > 0 ) newnode = n->right;
        else {
            if ( n->type != CDUBIOUS ) {
                int numdubious = AC.DubiousList.num;
                FUNCTIONS dub = (FUNCTIONS)FromVarList(&AC.DubiousList);
                dub->name = n->name;
                n->number = numdubious;
            }
            *number = n->number;
            return(CDUBIOUS);
        }
    } while ( newnode >= 0 );
    return(-1);
}

/*
    #] MakeDubious :
    #[ NameConflict :
*/

static char *nametype[] = { "symbol", "index", "vector", "function",
        "set", "expression" };
static char *plural[] = { "","n","","","","n" };

int NameConflict(int type, UBYTE *name)
{
    if ( type == NAMENOTFOUND ) {
        MesPrint("&%s has not been declared",name);
    }
    else if ( type != CDUBIOUS )
        MesPrint("&%s has been declared as a%s %s already"
            ,name,plural[type],nametype[type]);
    return(1);
}

/*
    #] NameConflict :
    #[ AddExpression :
*/

int AddExpression(UBYTE *name, int x, int y)
{
    int nodenum, numexpr = AC.ExpressionList.num;
    EXPRESSIONS expr = (EXPRESSIONS)FromVarList(&AC.ExpressionList);
    UBYTE *s;
    expr->status = x;
    expr->printflag = y;
    PUTZERO(expr->onfile);
    PUTZERO(expr->size);
    expr->renum = 0;
    expr->renumlists = 0;
    expr->hidelevel = 0;
    expr->inmem = 0;
    expr->bracketinfo = expr->newbracketinfo = 0;
    if ( name ) {
        expr->name = AddName(AC.exprnames,name,CEXPRESSION,numexpr,&nodenum);
        expr->node = nodenum;
        expr->replace = NEWLYDEFINEDEXPRESSION ;
        s = name;
        while ( *s ) s++;
        expr->namesize = (s-name)+1;
    }
    else {
        expr->replace = REDEFINEDEXPRESSION;
        expr->name = AC.TransEname;
        expr->node = -1;
```

```
        expr->namesize = 0;
    }
    expr->vflags = 0;
    expr->numdummies = 0;
    expr->numfactors = 0;
#ifdef PARALLELCODE
    expr->partodo = 0;
#endif
    return(numexpr);
}

/*
    #] AddExpression :
    #[ GetLabel :
*/

int GetLabel(UBYTE *name)
{
    int i;
    LONG newnum;
    UBYTE **NewLabelNames;
    int *NewLabel;
    for ( i = 0; i < AC.NumLabels; i++ ) {
        if ( StrCmp(name,AC.LabelNames[i]) == 0 ) return(i);
    }
    if ( AC.NumLabels >= AC.MaxLabels ) {
        newnum = 2*AC.MaxLabels;
        if ( newnum == 0 ) newnum = 10;
        if ( newnum > 32765 ) newnum = 32765;
        if ( newnum == AC.MaxLabels ) {
            MesPrint("&More than 32765 labels in one module. Please simplify.");
            Terminate(-1);
        }
        NewLabelNames = (UBYTE **)Malloc1((sizeof(UBYTE *)+sizeof(int))
                *newnum,"Labels");
        NewLabel = (int *)(NewLabelNames+newnum);
        for ( i = 0; i< AC.MaxLabels; i++ ) {
            NewLabelNames[i] = AC.LabelNames[i];
            NewLabel[i] = AC.Labels[i];
        }
        if ( AC.LabelNames ) M_free(AC.LabelNames,"Labels");
        AC.LabelNames = NewLabelNames;
        AC.Labels = NewLabel;
        AC.MaxLabels = newnum;
    }
    i = AC.NumLabels++;
    AC.LabelNames[i] = strDup1(name,"Labels");
    AC.Labels[i] = -1;
    return(i);
}

/*
    #] GetLabel :
    #[ ResetVariables :

    Resets the variables.
    par = 0  The list of temporary sets (after each .sort)
    par = 1  The list of local variables (after each .store)
    par = 2  All variables (after each .clear)
*/

void ResetVariables(int par)
{
    int i, j;
    TABLES T;
    switch ( par ) {
    case 0 : /* Only the sets without a name */
        AC.SetList.num        = AC.SetList.numtemp;
        AC.SetElementList.num = AC.SetElementList.numtemp;
        break;
    case 2 :
        for ( i = AC.SymbolList.numclear; i < AC.SymbolList.num; i++ )
            AC.varnames->namenode[symbols[i].node].type = CDELETE;
        AC.SymbolList.num = AC.SymbolList.numglobal = AC.SymbolList.numclear;
        for ( i = AC.VectorList.numclear; i < AC.VectorList.num; i++ )
            AC.varnames->namenode[vectors[i].node].type = CDELETE;
        AC.VectorList.num = AC.VectorList.numglobal = AC.VectorList.numclear;
        for ( i = AC.IndexList.numclear; i < AC.IndexList.num; i++ )
            AC.varnames->namenode[indices[i].node].type = CDELETE;
        AC.IndexList.num = AC.IndexList.numglobal = AC.IndexList.numclear;
        for ( i = AC.FunctionList.numclear; i < AC.FunctionList.num; i++ ) {
            AC.varnames->namenode[functions[i].node].type = CDELETE;
            if ( ( T = functions[i].tabl ) != 0 ) {
                if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
                if ( T->prototype ) M_free(T->prototype,"tableprototype");
                if ( T->mm ) M_free(T->mm,"tableminmax");
                if ( T->flags ) M_free(T->flags,"tableflags");
```

```
                if ( T->argtail ) M_free(T->argtail,"table arguments");
                if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
                for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
                    finishcbuf(T->buffers[j]);
                }
                /*[07apr2004 mt]:*/ /*memory leak*/
                if ( T->buffers ) M_free(T->buffers,"Table buffers");
                /*:[07apr2004 mt]*/
                finishcbuf(T->bufnum);
                if ( T->spare ) {
                    TABLES TT = T->spare;
                    if ( TT->mm ) M_free(TT->mm,"tableminmax");
                    if ( TT->flags ) M_free(TT->flags,"tableflags");
                    if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
                    for (j = 0; j < TT->buffersfill; j++ ) { /* was <= */
                        finishcbuf(TT->buffers[j]);
                    }
                    if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
                    /*[07apr2004 mt]:*/ /*memory leak*/
                    if ( TT->buffers )M_free(TT->buffers,"Table buffers");
                    /*:[07apr2004 mt]*/
                    M_free(TT,"table");
                }
                M_free(T,"table");
            }
        }
        AC.FunctionList.num = AC.FunctionList.numglobal = AC.FunctionList.numclear;
        for ( i = AC.SetList.numclear; i < AC.SetList.num; i++ ) {
            if ( Sets[i].node >= 0 )
                AC.varnames->namenode[Sets[i].node].type = CDELETE;
        }
        AC.SetList.numtemp = AC.SetList.num = AC.SetList.numglobal = AC.SetList.numclear;
        for ( i = AC.DubiousList.numclear; i < AC.DubiousList.num; i++ )
            AC.varnames->namenode[Dubious[i].node].type = CDELETE;
        AC.DubiousList.num = AC.DubiousList.numglobal = AC.DubiousList.numclear;
        AC.SetElementList.numtemp = AC.SetElementList.num =
            AC.SetElementList.numglobal = AC.SetElementList.numclear;
        CompactifyTree(AC.varnames,VARNAMES);
        AC.varnames->namefill = AC.varnames->globalnamefill = AC.varnames->clearnamefill;
        AC.varnames->nodefill = AC.varnames->globalnodefill = AC.varnames->clearnodefill;

        for ( i = AC.AutoSymbolList.numclear; i < AC.AutoSymbolList.num; i++ )
            AC.autonames->namenode[
                    ((SYMBOLS)(AC.AutoSymbolList.lijst))[i].node].type = CDELETE;
        AC.AutoSymbolList.num = AC.AutoSymbolList.numglobal
                            = AC.AutoSymbolList.numclear;
        for ( i = AC.AutoVectorList.numclear; i < AC.AutoVectorList.num; i++ )
            AC.autonames->namenode[
                    ((VECTORS)(AC.AutoVectorList.lijst))[i].node].type = CDELETE;
        AC.AutoVectorList.num = AC.AutoVectorList.numglobal
                            = AC.AutoVectorList.numclear;
        for ( i = AC.AutoIndexList.numclear; i < AC.AutoIndexList.num; i++ )
            AC.autonames->namenode[
                    ((INDICES)(AC.AutoIndexList.lijst))[i].node].type = CDELETE;
        AC.AutoIndexList.num = AC.AutoIndexList.numglobal
                            = AC.AutoIndexList.numclear;
        for ( i = AC.AutoFunctionList.numclear; i < AC.AutoFunctionList.num; i++ ) {
            AC.autonames->namenode[
                    ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].node].type = CDELETE;
            if ( ( T = ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl ) != 0 ) {
                if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
                if ( T->prototype ) M_free(T->prototype,"tableprototype");
                if ( T->mm ) M_free(T->mm,"tableminmax");
                if ( T->flags ) M_free(T->flags,"tableflags");
                if ( T->argtail ) M_free(T->argtail,"table arguments");
                if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
                for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
                    finishcbuf(T->buffers[j]);
                }
                if ( T->spare ) {
                    TABLES TT = T->spare;
                    if ( TT->mm ) M_free(TT->mm,"tableminmax");
                    if ( TT->flags ) M_free(TT->flags,"tableflags");
                    if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
                    for (j = 0; j < TT->buffersfill; j++ ) { /* was <= */
                        finishcbuf(TT->buffers[j]);
                    }
                    if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
                    M_free(TT,"table");
                }
                M_free(T,"table");
            }
        }
        AC.AutoFunctionList.num = AC.AutoFunctionList.numglobal
                            = AC.AutoFunctionList.numclear;
        CompactifyTree(AC.autonames,AUTONAMES);
        AC.autonames->namefill = AC.autonames->globalnamefill
```

```
                                = AC.autonames->clearnamefill;
        AC.autonames->nodefill = AC.autonames->globalnodefill
                                = AC.autonames->clearnodefill;
        ReleaseTB();
        break;
    case 1 :
        for ( i = AC.SymbolList.numglobal; i < AC.SymbolList.num; i++ )
            AC.varnames->namenode[symbols[i].node].type = CDELETE;
        AC.SymbolList.num = AC.SymbolList.numglobal;
        for ( i = AC.VectorList.numglobal; i < AC.VectorList.num; i++ )
            AC.varnames->namenode[vectors[i].node].type = CDELETE;
        AC.VectorList.num = AC.VectorList.numglobal;
        for ( i = AC.IndexList.numglobal; i < AC.IndexList.num; i++ )
            AC.varnames->namenode[indices[i].node].type = CDELETE;
        AC.IndexList.num = AC.IndexList.numglobal;
        for ( i = AC.FunctionList.numglobal; i < AC.FunctionList.num; i++ ) {
            AC.varnames->namenode[functions[i].node].type = CDELETE;
            if ( ( T = functions[i].tabl ) != 0 ) {
                if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
                if ( T->prototype ) M_free(T->prototype,"tableprototype");
                if ( T->mm ) M_free(T->mm,"tableminmax");
                if ( T->flags ) M_free(T->flags,"tableflags");
                if ( T->argtail ) M_free(T->argtail,"table arguments");
                if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
                for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
                    finishcbuf(T->buffers[j]);
                }
                /*[07apr2004 mt]:*/  /*memory leak*/
                if ( T->buffers ) M_free(T->buffers,"Table buffers");
                /*:[07apr2004 mt]*/
                finishcbuf(T->bufnum);
                if ( T->spare ) {
                    TABLES TT = T->spare;
                    if ( TT->mm ) M_free(TT->mm,"tableminmax");
                    if ( TT->flags ) M_free(TT->flags,"tableflags");
                    if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
                    for (j = 0; j < TT->buffersfill; j++ ) { /* was <= */
                        finishcbuf(TT->buffers[j]);
                    }
                    if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
                    /*[07apr2004 mt]:*/ /*memory leak*/
                    if ( TT->buffers ) M_free(TT->buffers,"Table buffers");
                    /*:[07apr2004 mt]*/
                    M_free(TT,"table");
                }
                M_free(T,"table");
            }
        }
#ifdef TABLECLEANUP
        {
        int j;
        WORD *tp;
        for ( i = 0; i < AC.FunctionList.numglobal; i++ ) {
/*
            Now, if the table definition is from after the .global
            while the function is from before, there is a problem.
            This could be resolved by defining CTable (=Table), Ntable
            and do away with the previous function definition.
*/
            if ( ( T = functions[i].tabl ) != 0 ) {
/*
                First restore overwritten definitions.
*/
                if ( T->sparse ) {
                    T->totind = T->mdefined;
                    for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
                        tp += T->numind;
#if TABLEEXTENSION == 2
                        tp[0] = tp[1];
#else
                        tp[0] = tp[2];
                        tp[1] = tp[3];
                        tp[4] = tp[5];
#endif
                        tp += TABLEEXTENSION;
                    }
                    RedoTableTree(T,T->totind);
                    if ( T->spare ) {
                        TABLES TT = T->spare;
                        TT->totind = TT->mdefined;
                        for ( j = 0, tp = TT->tablepointers; j < TT->totind; j++ ) {
                            tp += TT->numind;
#if TABLEEXTENSION == 2
                            tp[0] = tp[1];
#else
                            tp[0] = tp[2];
                            tp[1] = tp[3];
```

```
                            tp[4] = tp[5];
#endif
                            tp += TABLEEXTENSION;
                        }
                        RedoTableTree(TT,TT->totind);
                        cbuf[TT->bufnum].numlhs = cbuf[TT->bufnum].mnumlhs;
                        cbuf[TT->bufnum].numrhs = cbuf[TT->bufnum].mnumrhs;
                    }
                }
                else {
                    for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
#if TABLEEXTENSION == 2
                        tp[0] = tp[1];
#else
                        tp[0] = tp[2];
                        tp[1] = tp[3];
                        tp[4] = tp[5];
#endif
                    }
                    T->defined = T->mdefined;
                }
                cbuf[T->bufnum].numlhs = cbuf[T->bufnum].mnumlhs;
                cbuf[T->bufnum].numrhs = cbuf[T->bufnum].mnumrhs;
            }
        }
        }
#endif
        AC.FunctionList.num = AC.FunctionList.numglobal;
        for ( i = AC.SetList.numglobal; i < AC.SetList.num; i++ ) {
            if ( Sets[i].node >= 0 )
                AC.varnames->namenode[Sets[i].node].type = CDELETE;
        }
        AC.SetList.numtemp = AC.SetList.num = AC.SetList.numglobal;
        for ( i = AC.DubiousList.numglobal; i < AC.DubiousList.num; i++ )
            AC.varnames->namenode[Dubious[i].node].type = CDELETE;
        AC.DubiousList.num = AC.DubiousList.numglobal;
        AC.SetElementList.numtemp = AC.SetElementList.num =
            AC.SetElementList.numglobal;
        CompactifyTree(AC.varnames,VARNAMES);
        AC.varnames->namefill = AC.varnames->globalnamefill;
        AC.varnames->nodefill = AC.varnames->globalnodefill;

        for ( i = AC.AutoSymbolList.numglobal; i < AC.AutoSymbolList.num; i++ )
            AC.autonames->namenode[
                    ((SYMBOLS)(AC.AutoSymbolList.lijst))[i].node].type = CDELETE;
        AC.AutoSymbolList.num = AC.AutoSymbolList.numglobal;
        for ( i = AC.AutoVectorList.numglobal; i < AC.AutoVectorList.num; i++ )
            AC.autonames->namenode[
                    ((VECTORS)(AC.AutoVectorList.lijst))[i].node].type = CDELETE;
        AC.AutoVectorList.num = AC.AutoVectorList.numglobal;
        for ( i = AC.AutoIndexList.numglobal; i < AC.AutoIndexList.num; i++ )
            AC.autonames->namenode[
                    ((INDICES)(AC.AutoIndexList.lijst))[i].node].type = CDELETE;
        AC.AutoIndexList.num = AC.AutoIndexList.numglobal;
        for ( i = AC.AutoFunctionList.numglobal; i < AC.AutoFunctionList.num; i++ ) {
            AC.autonames->namenode[
                    ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].node].type = CDELETE;
            if ( ( T = ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl ) != 0 ) {
                if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
                if ( T->prototype ) M_free(T->prototype,"tableprototype");
                if ( T->mm ) M_free(T->mm,"tableminmax");
                if ( T->flags ) M_free(T->flags,"tableflags");
                if ( T->argtail ) M_free(T->argtail,"table arguments");
                if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
                for (j = 0; j < T->buffersfill; j++ ) { /* was <= */
                    finishcbuf(T->buffers[j]);
                }
                if ( T->spare ) {
                    TABLES TT = T->spare;
                    if ( TT->mm ) M_free(TT->mm,"tableminmax");
                    if ( TT->flags ) M_free(TT->flags,"tableflags");
                    if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
                    for (j = 0; j < TT->buffersfill; j++ ) { /* was <= */
                        finishcbuf(TT->buffers[j]);
                    }
                    if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
                    M_free(TT,"table");
                }
                M_free(T,"table");
            }
        }
        AC.AutoFunctionList.num = AC.AutoFunctionList.numglobal;

        CompactifyTree(AC.autonames,AUTONAMES);

        AC.autonames->namefill = AC.autonames->globalnamefill;
        AC.autonames->nodefill = AC.autonames->globalnodefill;
```

```
        break;
    }
}

/*
    #] ResetVariables :
    #[ RemoveDollars :
*/

void RemoveDollars()
{
    DOLLARS d;
    CBUF *C = cbuf + AM.dbufnum;
    int numdollar = AP.DollarList.num;
    if ( numdollar > 0 ) {
      while ( numdollar > AM.gcNumDollars ) {
        numdollar--;
        d = Dollars + numdollar;
        if ( d->where && d->where != &(d->zero) && d->where != &(AM.dollarzero) ) {
            M_free(d->where,"dollar->where"); d->where = &(d->zero); d->size = 0;
        }
        AC.dollarnames->namenode[d->node].type = CDELETE;
      }
      AP.DollarList.num = AM.gcNumDollars;
      CompactifyTree(AC.dollarnames,DOLLARNAMES);

      C->numrhs = C->mnumrhs;
      C->numlhs = C->mnumlhs;
    }
}

/*
    #] RemoveDollars :
    #[ Globalize :
*/

void Globalize(int par)
{
    int i, j;
    WORD *tp;
    if ( par == 1 ) {
        AC.SymbolList.numclear     = AC.SymbolList.num;
        AC.VectorList.numclear     = AC.VectorList.num;
        AC.IndexList.numclear      = AC.IndexList.num;
        AC.FunctionList.numclear   = AC.FunctionList.num;
        AC.SetList.numclear        = AC.SetList.num;
        AC.DubiousList.numclear    = AC.DubiousList.num;
        AC.SetElementList.numclear = AC.SetElementList.num;
        AC.varnames->clearnamefill = AC.varnames->namefill;
        AC.varnames->clearnodefill = AC.varnames->nodefill;

        AC.AutoSymbolList.numclear   = AC.AutoSymbolList.num;
        AC.AutoVectorList.numclear   = AC.AutoVectorList.num;
        AC.AutoIndexList.numclear    = AC.AutoIndexList.num;
        AC.AutoFunctionList.numclear = AC.AutoFunctionList.num;
        AC.autonames->clearnamefill  = AC.autonames->namefill;
        AC.autonames->clearnodefill  = AC.autonames->nodefill;
    }
/*  for ( i = AC.FunctionList.numglobal; i < AC.FunctionList.num; i++ ) { */
    for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
/*
        We need here not only the not-yet-global functions. The already
        global ones may have obtained extra elements.
*/
        if ( functions[i].tabl ) {
            TABLES T = functions[i].tabl;
            if ( T->sparse ) {
                T->mdefined = T->totind;
                for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
                    tp += T->numind;
#if TABLEEXTENSION == 2
                    tp[1] = tp[0];
#else
                    tp[2] = tp[0]; tp[3] = tp[1]; tp[5] = tp[4] & (~ELEMENTUSED);
#endif
                    tp += TABLEEXTENSION;
                }
                if ( T->spare ) {
                    TABLES TT = T->spare;
                    TT->mdefined = TT->totind;
                    for ( j = 0, tp = TT->tablepointers; j < TT->totind; j++ ) {
                        tp += TT->numind;
#if TABLEEXTENSION == 2
                        tp[1] = tp[0];
#else
                        tp[2] = tp[0]; tp[3] = tp[1]; tp[5] = tp[4] & (~ELEMENTUSED);
#endif
```

```
                        tp += TABLEEXTENSION;
                    }
                    cbuf[TT->bufnum].mnumlhs = cbuf[TT->bufnum].numlhs;
                    cbuf[TT->bufnum].mnumrhs = cbuf[TT->bufnum].numrhs;
                }
            }
            else {
                T->mdefined = T->defined;
                for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
#if TABLEEXTENSION == 2
                    tp[1] = tp[0];
#else
                    tp[2] = tp[0]; tp[3] = tp[1]; tp[5] = tp[4] & (~ELEMENTUSED);
#endif
                }
            }
            cbuf[T->bufnum].mnumlhs = cbuf[T->bufnum].numlhs;
            cbuf[T->bufnum].mnumrhs = cbuf[T->bufnum].numrhs;
        }
    }
    for ( i = AC.AutoFunctionList.numglobal; i < AC.AutoFunctionList.num; i++ ) {
        if ( ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl )
            ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl->mdefined =
                ((FUNCTIONS)(AC.AutoFunctionList.lijst))[i].tabl->defined;
    }
    AC.SymbolList.numglobal     = AC.SymbolList.num;
    AC.VectorList.numglobal     = AC.VectorList.num;
    AC.IndexList.numglobal      = AC.IndexList.num;
    AC.FunctionList.numglobal   = AC.FunctionList.num;
    AC.SetList.numglobal        = AC.SetList.num;
    AC.DubiousList.numglobal    = AC.DubiousList.num;
    AC.SetElementList.numglobal = AC.SetElementList.num;
    AC.varnames->globalnamefill = AC.varnames->namefill;
    AC.varnames->globalnodefill = AC.varnames->nodefill;

    AC.AutoSymbolList.numglobal   = AC.AutoSymbolList.num;
    AC.AutoVectorList.numglobal   = AC.AutoVectorList.num;
    AC.AutoIndexList.numglobal    = AC.AutoIndexList.num;
    AC.AutoFunctionList.numglobal = AC.AutoFunctionList.num;
    AC.autonames->globalnamefill  = AC.autonames->namefill;
    AC.autonames->globalnodefill  = AC.autonames->nodefill;
}

/*
    #] Globalize :
    #[ TestName :
*/

int TestName(UBYTE *name)
{
    if ( *name == '[' ) {
        while ( *name ) name++;
        if ( name[-1] == ']' ) return(0);
        return(-1);
    }
    while ( *name ) {
        if ( *name == '_' ) return(-1);
        name++;
    }
    return(0);
}

/*
    #] TestName :
*/
```

4.67 normal.c File Reference

`#include "form3.h"`
Include dependency graph for normal.c:

Macros

- #define MAXNUMBEROFNONCOMTERMS 2
- #define EXCHN(t1, t2, n) { WORD a,i; for(i=0;i<n;i++){x=t1[i];t1[i]=t2[i];t2[i]=a;}
- #define EXCH(x, y) { WORD a = (x); (x) = (y); (y) = a; }

Functions

- WORD CompareFunctions (WORD ∗fleft, WORD ∗fright)
- WORD Commute (WORD ∗fleft, WORD ∗fright)
- WORD Normalize (PHEAD WORD ∗term)
- WORD ExtraSymbol (WORD sym, WORD pow, WORD nsym, WORD ∗ppsym, WORD ∗ncoef)
- WORD DoTheta (PHEAD WORD ∗t)
- WORD DoDelta (WORD ∗t)
- void DoRevert (WORD ∗fun, WORD ∗tmp)
- WORD DetCommu (WORD ∗terms)
- WORD DoesCommu (WORD ∗term)
- int TreatPolyRatFun (PHEAD WORD ∗prf)
- void DropCoefficient (PHEAD WORD ∗term)
- void DropSymbols (PHEAD WORD ∗term)
- int SymbolNormalize (WORD ∗term)
- int TestFunFlag (PHEAD WORD ∗tfun)
- WORD BracketNormalize (PHEAD WORD ∗term)

4.67.1 Detailed Description

Mainly the routine Normalize. This routine brings terms to standard FORM. Currently it has one serious drawback. Its buffers are all in the stack. This means these buffers have a fixed size (NORMSIZE). In the past this has caused problems and NORMSIZE had to be increased.

It is not clear whether Normalize can be called recursively.

Definition in file normal.c.

4.67.2 Macro Definition Documentation

4.67.2.1 MAXNUMBEROFNONCOMTERMS

```
#define MAXNUMBEROFNONCOMTERMS 2
```

Definition at line 4343 of file normal.c.

4.67.2.2 EXCHN

```
#define EXCHN(
              t1,
              t2,
              n ) { WORD a,i; for(i=0;i<n;i++){x=t1[i];t1[i]=t2[i];t2[i]=a;}
```

Definition at line 5106 of file normal.c.

4.67.2.3 EXCH

```
#define EXCH(
              x,
              y ) { WORD a = (x); (x) = (y); (y) = a; }
```

Definition at line 5107 of file normal.c.

4.67.3 Function Documentation

4.67.3.1 CompareFunctions()

```
WORD CompareFunctions (
            WORD * fleft,
            WORD * fright )
```

Definition at line 48 of file normal.c.

4.67.3.2 Commute()

```
WORD Commute (
            WORD * fleft,
            WORD * fright )
```

Definition at line 112 of file normal.c.

4.67.3.3 Normalize()

```
WORD Normalize (
            PHEAD WORD * term )
```

Definition at line 187 of file normal.c.

4.67.3.4 ExtraSymbol()

```
WORD ExtraSymbol (
            WORD sym,
            WORD pow,
            WORD nsym,
            WORD * ppsym,
            WORD * ncoef )
```

Definition at line 4031 of file normal.c.

4.67.3.5 DoTheta()

```
WORD DoTheta (
            PHEAD WORD * t )
```

Definition at line 4093 of file normal.c.

4.67.3.6 DoDelta()

```
WORD DoDelta (
            WORD * t )
```

Definition at line 4190 of file normal.c.

4.67.3.7 DoRevert()

```
void DoRevert (
            WORD * fun,
            WORD * tmp )
```

Definition at line 4257 of file normal.c.

4.67.3.8 DetCommu()

```
WORD DetCommu (
            WORD * terms )
```

Definition at line 4345 of file normal.c.

4.67.3.9 DoesCommu()

```
WORD DoesCommu (
            WORD * term )
```

Definition at line 4404 of file normal.c.

4.67.3.10 TreatPolyRatFun()

```
int TreatPolyRatFun (
            PHEAD WORD * prf )
```

Definition at line 4795 of file normal.c.

4.67.3.11 DropCoefficient()

```
void DropCoefficient (
            PHEAD WORD * term )
```

Definition at line 4931 of file normal.c.

4.67.3.12 DropSymbols()

```
void DropSymbols (
            PHEAD WORD * term )
```

Definition at line 4949 of file normal.c.

4.67.3.13 SymbolNormalize()

```
int SymbolNormalize (
            WORD * term )
```

Routine normalizes terms that contain only symbols. Regular minimum and maximum properties are ignored.

We check whether there are negative powers in the output. This is not allowed.

Definition at line 4979 of file normal.c.

Referenced by InFunction().

4.67.3.14 TestFunFlag()

```
int TestFunFlag (
            PHEAD WORD * tfun )
```

Definition at line 5075 of file normal.c.

4.67.3.15 BracketNormalize()

```
WORD BracketNormalize (
            PHEAD WORD * term )
```

Definition at line 5109 of file normal.c.

4.68 normal.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes : normal.c
*/

#include "form3.h"

/*
    #] Includes :
    #[ Normalize :
        #[ CompareFunctions :
*/

WORD CompareFunctions(WORD *fleft,WORD *fright)
{
    WORD k, kk;
    if ( AC.properorderflag ) {
        if ( ( *fleft >= (FUNCTION+WILDOFFSET)
        && functions[*fleft-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION )
        || ( *fleft >= FUNCTION && *fleft < (FUNCTION + WILDOFFSET)
        && functions[*fleft-FUNCTION].spec >= TENSORFUNCTION ) ) {}
        else {
            WORD *s1, *s2, *ss1, *ss2;
            s1 = fleft+FUNHEAD; s2 = fright+FUNHEAD;
            ss1 = fleft + fleft[1]; ss2 = fright + fright[1];
            while ( s1 < ss1 && s2 < ss2 ) {
                k = CompArg(s1,s2);
                if ( k > 0 ) return(1);
                if ( k < 0 ) return(0);
                NEXTARG(s1)
                NEXTARG(s2)
            }
            if ( s1 < ss1 ) return(1);
            return(0);
        }
        k = fleft[1] - FUNHEAD;
        kk = fright[1] - FUNHEAD;
        fleft += FUNHEAD;
        fright += FUNHEAD;
        while ( k > 0 && kk > 0 ) {
            if ( *fleft < *fright ) return(0);
            else if ( *fleft++ > *fright++ ) return(1);
            k--; kk--;
        }
        if ( k > 0 ) return(1);
        return(0);
    }
    else {
        k = fleft[1] - FUNHEAD;
        kk = fright[1] - FUNHEAD;
        fleft += FUNHEAD;
        fright += FUNHEAD;
        while ( k > 0 && kk > 0 ) {
            if ( *fleft < *fright ) return(0);
            else if ( *fleft++ > *fright++ ) return(1);
            k--; kk--;
```

```
        }
        if ( k > 0 ) return(1);
        return(0);
    }
}

/*
        #] CompareFunctions :
        #[ Commute :

    This function gets two adjacent function pointers and decides
    whether these two functions should be exchanged to obtain a
    natural ordering.

    Currently there is only an ordering of gamma matrices belonging
    to different spin lines.

    Note that we skip for now the cases of (F)^(3/2) or 1/F and a few more
    of such funny functions.
*/

WORD Commute(WORD *fleft, WORD *fright)
{
    WORD fun1, fun2;
    if ( *fleft == DOLLAREXPRESSION || *fright == DOLLAREXPRESSION ) return(0);
    fun1 = ABS(*fleft); fun2 = ABS(*fright);
    if ( *fleft >= GAMMA && *fleft <= GAMMASEVEN
        && *fright >= GAMMA && *fright <= GAMMASEVEN ) {
        if ( fleft[FUNHEAD] < AM.OffsetIndex && fleft[FUNHEAD] > fright[FUNHEAD] )
            return(1);
        return(0);
    }
    if ( fun1 >= WILDOFFSET ) fun1 -= WILDOFFSET;
    if ( fun2 >= WILDOFFSET ) fun2 -= WILDOFFSET;
    if ( ( ( functions[fun1-FUNCTION].flags & COULDCOMMUTE ) == 0 )
      || ( ( functions[fun2-FUNCTION].flags & COULDCOMMUTE ) == 0 ) ) return(0);
/*
    if other conditions will come here, keep in mind that if *fleft < 0
    or *fright < 0 they are arguments in the exponent function as in f^(3/2)
*/
    if ( AC.CommuteInSet == 0 ) return(0);
/*
    The code for CompareFunctions can be stolen from the commuting case.

    We need the syntax:
        Commute Fun1,Fun2,...,Fun`n';
    For this Fun1,...,Fun`n' need to be noncommuting functions.
    These functions will commute with all members of the group.
    In the AC.paircommute buffer the representation is
        `n'+1,element1,...,element`n',`m'+1,element1,...,element`m',0
    A function can belong to more than one group.
    If a function commutes with itself, it is most efficient to make a separate
    group of two elements for it as in
        Commute T,T;   -> 3,T,T
*/
    if ( fun1 >= fun2 ) {
        WORD *group = AC.CommuteInSet, *g1, *g2, *g3;
        while ( *group > 0 ) {
            g3 = group + *group;
            g1 = group+1;
            while ( g1 < g3 ) {
                if ( *g1 == fun1 || ( fun1 <= GAMMASEVEN && fun1 >= GAMMA
                 && *g1 <= GAMMASEVEN && *g1 >= GAMMA ) ) {
                    g2 = group+1;
                    while ( g2 < g3 ) {
                        if ( g1 != g2 && ( *g2 == fun2 ||
                         ( fun2 <= GAMMASEVEN && fun2 >= GAMMA
                         && *g2 <= GAMMASEVEN && *g2 >= GAMMA ) ) ) {
                            if ( fun1 != fun2 ) return(1);
                            if ( *fleft < 0 ) return(0);
                            if ( *fright < 0 ) return(1);
                            return(CompareFunctions(fleft,fright));
                        }
                        g2++;
                    }
                    break;
                }
                g1++;
            }
            group = g3;
        }
    }
    return(0);
}

/*
        #] Commute :
```

```
        #[ Normalize :

    This is the big normalization routine. It has a great need
    to be economical.
    There is a fixed limit to the number of objects coming in.
    Something should be done about it.

*/

WORD Normalize(PHEAD WORD *term)
{
/*
    #[ Declarations :
*/
    GETBIDENTITY
    WORD *t, *m, *r, i, j, k, l, nsym, *ss, *tt, *u;
    WORD shortnum, stype;
    WORD *stop, *to = 0, *from = 0;
/*
    The next variables would be better off in the AT.WorkSpace (?)
    or as static global variables. Now they make stackallocations
    rather bothersome.
*/
    WORD psym[7*NORMSIZE],*ppsym;
    WORD pvec[NORMSIZE],*ppvec,nvec;
    WORD pdot[3*NORMSIZE],*ppdot,ndot;
    WORD pdel[2*NORMSIZE],*ppdel,ndel;
    WORD pind[NORMSIZE],nind;
    WORD *peps[NORMSIZE/3],neps;
    WORD *pden[NORMSIZE/3],nden;
    WORD *pcom[NORMSIZE],ncom;
    WORD *pnco[NORMSIZE],nnco;
    WORD *pcon[2*NORMSIZE],ncon;        /* Pointer to contractable indices */
    WORD *n_coef, ncoef;                /* Accumulator for the coefficient */
    WORD *n_llnum, *lnum, nnum;
    WORD *termout, oldtoprhs = 0, subtype;
    WORD ReplaceType, ReplaceVeto = 0, didcontr, regval = 0;
    WORD *ReplaceSub;
    WORD *fillsetexp;
    CBUF *C = cbuf+AT.ebufnum;
    WORD *ANsc = 0, *ANsm = 0, *ANsr = 0, PolyFunMode;
    LONG oldcpointer = 0;
    n_coef = TermMalloc("NormCoef");
    n_llnum = TermMalloc("n_llnum");
    lnum = n_llnum+1;

/*
    int termflag;
*/
/*
    #] Declarations :
    #[ Setup :
PrintTerm(term,"Normalize");
*/

Restart:
    didcontr = 0;
    ReplaceType = -1;
    t = term;
    if ( !*t ) { TermFree(n_coef,"NormCoef"); TermFree(n_llnum,"n_llnum"); return(regval); }
    r = t + *t;
    ncoef = r[-1];
    i = ABS(ncoef);
    r -= i;
    m = r;
    t = n_coef;
    NCOPY(t,r,i);
    termout = AT.WorkPointer;
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    fillsetexp = termout+1;
    AN.PolyNormFlag = 0; PolyFunMode = AN.PolyFunTodo;
/*
    termflag = 0;
*/
/*
    #] Setup :
    #[ First scan :
*/
    nsym = nvec = ndot = ndel = neps = nden =
    nind = ncom = nnco = ncon = 0;
    ppsym = psym;
    ppvec = pvec;
    ppdot = pdot;
    ppdel = pdel;
    t = term + 1;
conscan:;
    if ( t < m ) do {
```

```
        r = t + t[1];
        switch ( *t ) {
            case SYMBOL :
                t += 2;
                from = m;
                do {
                    if ( t[1] == 0 ) {
/*                      if ( *t == 0 || *t == MAXPOWER ) goto NormZZ; */
                        t += 2;
                        goto NextSymbol;
                    }
                    if ( *t <= DENOMINATORSYMBOL && *t >= COEFFSYMBOL ) {
/*
                        if ( AN.NoScrat2 == 0 ) {
                            AN.NoScrat2 = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"Normalize");
                        }
*/
                        if ( AN.cTerm ) m = AN.cTerm;
                        else m = term;
                        m += *m;
                        ncoef = REDLENG(ncoef);
                        if ( *t == COEFFSYMBOL ) {
                          i = t[1];
                          nnum = REDLENG(m[-1]);
                          m -= ABS(m[-1]);
                          if ( i > 0 ) {
                            while ( i > 0 ) {
                                if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)m,nnum,
                                (UWORD *)n_coef,&ncoef) ) goto FromNorm;
                                i--;
                            }
                          }
                          else if ( i < 0 ) {
                            while ( i < 0 ) {
                                if ( DivRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)m,nnum,
                                (UWORD *)n_coef,&ncoef) ) goto FromNorm;
                                i++;
                            }
                          }
                        }
                        else {
                          i = m[-1];
                          nnum = (ABS(i)-1)/2;
                          if ( *t == NUMERATORSYMBOL ) { m -= nnum + 1; }
                          else { m--; }
                          while ( *m == 0 && nnum > 1 ) { m--; nnum--; }
                          m -= nnum;
                          if ( i < 0 && *t == NUMERATORSYMBOL ) nnum = -nnum;
                          i = t[1];
                          if ( i > 0 ) {
                            while ( i > 0 ) {
                                if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)m,nnum) )
                                        goto FromNorm;
                                i--;
                            }
                          }
                          else if ( i < 0 ) {
                            while ( i < 0 ) {
                                if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)m,nnum) )
                                        goto FromNorm;
                                i++;
                            }
                          }
                        }
                        ncoef = INCLENG(ncoef);
                        t += 2;
                        goto NextSymbol;
                    }
                    else if ( *t == DIMENSIONSYMBOL ) {
                        if ( AN.cTerm ) m = AN.cTerm;
                        else m = term;
                        k = DimensionTerm(m);
                        if ( k == 0 ) goto NormZero;
                        if ( k == MAXPOSITIVE ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Dimension_ is undefined in term %t");
                            MUNLOCK(ErrorMessageLock);
                            goto NormMin;
                        }
                        if ( k == -MAXPOSITIVE ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Dimension_ out of range in term %t");
                            MUNLOCK(ErrorMessageLock);
                            goto NormMin;
                        }
                        if ( k > 0 ) { *((UWORD *)lnum) = k; nnum = 1; }
                        else { *((UWORD *)lnum) = -k; nnum = -1; }
```

```
                        ncoef = REDLENG(ncoef);
                        if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) goto FromNorm;
                        ncoef = INCLENG(ncoef);
                        t += 2;
                        goto NextSymbol;
                    }
                    if ( ( *t >= MAXPOWER && *t < 2*MAXPOWER )
                        || ( *t < -MAXPOWER && *t > -2*MAXPOWER ) ) {
/*
            #[ TO SNUMBER :
*/
                if ( *t < 0 ) {
                    *t += MAXPOWER;
                    *t = -*t;
                    if ( t[1] & 1 ) ncoef = -ncoef;
                }
                else if ( *t == MAXPOWER ) {
                    if ( t[1] > 0 ) goto NormZero;
                    goto NormInf;
                }
                else {
                    *t -= MAXPOWER;
                }
                lnum[0] = *t;
                nnum = 1;
                if ( t[1] && RaisPow(BHEAD (UWORD *)lnum,&nnum,(UWORD)(ABS(t[1]))) )
                    goto FromNorm;
                ncoef = REDLENG(ncoef);
                if ( t[1] < 0 ) {
                    if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
                        goto FromNorm;
                }
                else if ( t[1] > 0 ) {
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
                        goto FromNorm;
                }
                ncoef = INCLENG(ncoef);
/*
            #] TO SNUMBER :
*/
                        t += 2;
                        goto NextSymbol;
                    }
                    if ( ( *t <= NumSymbols && *t > -MAXPOWER )
                     && ( symbols[*t].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
                        if ( t[1] <= 2*MAXPOWER && t[1] >= -2*MAXPOWER ) {
                        t[1] %= symbols[*t].maxpower;
                        if ( t[1] < 0 ) t[1] += symbols[*t].maxpower;
                        if ( ( symbols[*t].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
                            if ( ( ( symbols[*t].maxpower & 1 ) == 0 ) &&
                            ( t[1] >= symbols[*t].maxpower/2 ) ) {
                                t[1] -= symbols[*t].maxpower/2; ncoef = -ncoef;
                            }
                        }
                        if ( t[1] == 0 ) { t += 2; goto NextSymbol; }
                        }
                    }
                    i = nsym;
                    m = ppsym;
                    if ( i > 0 ) do {
                        m -= 2;
                        if  ( *t == *m ) {
                            t++; m++;
                            if  ( *t > 2*MAXPOWER || *t < -2*MAXPOWER
                            ||  *m > 2*MAXPOWER || *m < -2*MAXPOWER ) {
                                MLOCK(ErrorMessageLock);
                                MesPrint("Illegal wildcard power combination.");
                                MUNLOCK(ErrorMessageLock);
                                goto NormMin;
                            }
                            *m += *t;
                            if ( ( t[-1] <= NumSymbols && t[-1] > -MAXPOWER )
                             && ( symbols[t[-1]].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY
    ) {
                                *m %= symbols[t[-1]].maxpower;
                                if ( *m < 0 ) *m += symbols[t[-1]].maxpower;
                                if ( ( symbols[t[-1]].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
                                    if ( ( ( symbols[t[-1]].maxpower & 1 ) == 0 ) &&
                                    ( *m >= symbols[t[-1]].maxpower/2 ) ) {
                                        *m -= symbols[t[-1]].maxpower/2; ncoef = -ncoef;
                                    }
                                }
                            }
                            if  ( *m >= 2*MAXPOWER || *m <= -2*MAXPOWER ) {
                                MLOCK(ErrorMessageLock);
                                MesPrint("Power overflow during normalization");
                                MUNLOCK(ErrorMessageLock);
```

```
                                goto NormMin;
                            }
                            if ( !*m ) {
                                m--;
                                while ( i < nsym )
                                    { *m = m[2]; m++; *m = m[2]; m++; i++; }
                                ppsym -= 2;
                                nsym--;
                            }
                            t++;
                            goto NextSymbol;
                        }
                    } while ( *t < *m && --i > 0 );
                    m = ppsym;
                    while ( i < nsym )
                        { m--; m[2] = *m; m--; m[2] = *m; i++; }
                    *m++ = *t++;
                    *m = *t++;
                    ppsym += 2;
                    nsym++;
NextSymbol:;
                } while ( t < r );
                m = from;
                break;
            case VECTOR :
                t += 2;
                do {
                    if ( t[1] == FUNNYVEC ) {
                        pind[nind++] = *t;
                        t += 2;
                    }
                    else if ( t[1] < 0 ) {
                        if ( *t == NOINDEX && t[1] == NOINDEX ) t += 2;
                        else {
                        *ppdot++ = *t++; *ppdot++ = *t++; *ppdot++ = 1; ndot++;
                        }
                    }
                    else { *ppvec++ = *t++; *ppvec++ = *t++; nvec += 2; }
                } while ( t < r );
                break;
            case DOTPRODUCT :
                t += 2;
                do {
                    if ( t[2] == 0 ) t += 3;
                    else if ( ndot > 0 && t[0] == ppdot[-3]
                        && t[1] == ppdot[-2] ) {
                        ppdot[-1] += t[2];
                        t += 3;
                        if ( ppdot[-1] == 0 ) { ppdot -= 3; ndot--; }
                    }
                    else {
                        *ppdot++ = *t++; *ppdot++ = *t++;
                        *ppdot++ = *t++; ndot++;
                    }
                } while ( t < r );
                break;
            case HAAKJE :
                break;
            case SETSET:
                if ( WildFill(BHEAD termout,term,AT.dummysubexp) < 0 ) goto FromNorm;
                i = *termout;
                t = termout; m = term;
                NCOPY(m,t,i);
                goto Restart;
            case DOLLAREXPRESSION :
/*
                We have DOLLAREXPRESSION,4,number,power
                Replace by SUBEXPRESSION and exit elegantly to let
                TestSub pick it up. Of course look for special cases first.
                Note that we have a special compiler buffer for the values.
*/
                if ( AR.Eside != LHSIDE ) {
                DOLLARS d = Dollars + t[2];
#ifdef WITHPTHREADS
                int nummodopt, ptype = -1;
                if ( AS.MultiThreaded ) {
                    for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                        if ( t[2] == ModOptdollars[nummodopt].number ) break;
                    }
                    if ( nummodopt < NumModOptdollars ) {
                        ptype = ModOptdollars[nummodopt].type;
                        if ( ptype == MODLOCAL ) {
                            d = ModOptdollars[nummodopt].dstruct+AT.identity;
                        }
                        else {
                            LOCK(d->pthreadslockread);
                        }
```

```
                    }
                }
#endif
                if ( d->type == DOLZERO ) {
#ifdef WITHPTHREADS
                    if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                    if ( t[3] == 0 ) goto NormZZ;
                    if ( t[3] < 0 ) goto NormInf;
                    goto NormZero;
                }
                else if ( d->type == DOLNUMBER ) {
                    nnum = d->where[0];
                    if ( nnum > 0 ) {
                        nnum = d->where[nnum-1];
                        if ( nnum < 0 ) { ncoef = -ncoef; nnum = -nnum; }
                        nnum = (nnum-1)/2;
                        for ( i = 1; i <= nnum; i++ ) lnum[i-1] = d->where[i];
                    }
                    if ( nnum == 0 || ( nnum == 1 && lnum[0] == 0 ) ) {
#ifdef WITHPTHREADS
                        if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                        if ( t[3] < 0 ) goto NormInf;
                        else if ( t[3] == 0 ) goto NormZZ;
                        goto NormZero;
                    }
                    if ( t[3] && RaisPow(BHEAD (UWORD *)lnum,&nnum,(UWORD)(ABS(t[3]))) ) goto FromNorm;
                    ncoef = REDLENG(ncoef);
                    if ( t[3] < 0 ) {
                        if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) {
#ifdef WITHPTHREADS
                            if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                            goto FromNorm;
                        }
                    }
                    else if ( t[3] > 0 ) {
                        if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) {
#ifdef WITHPTHREADS
                            if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                            goto FromNorm;
                        }
                    }
                    ncoef = INCLENG(ncoef);
                }
                else if ( d->type == DOLINDEX ) {
                    if ( d->index == 0 ) {
#ifdef WITHPTHREADS
                        if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                        goto NormZero;
                    }
                    if ( d->index != NOINDEX ) pind[nind++] = d->index;
                }
                else if ( d->type == DOLTERMS ) {
                    if ( t[3] >= MAXPOWER || t[3] <= -MAXPOWER ) {
                        if ( d->where[0] == 0 ) goto NormZero;
                        if ( d->where[d->where[0]] != 0 ) {
IllDollarExp:
                            MLOCK(ErrorMessageLock);
                            MesPrint("!!!Illegal $ expansion with wildcard power!!!");
                            MUNLOCK(ErrorMessageLock);
                            goto FromNorm;
                        }
/*
                        At this point we should only admit symbols and dotproducts
                        We expand the dollar directly and do not send it back.
*/
                        {   WORD *td, *tdstop, dj;
                            td = d->where+1;
                            tdstop = d->where+d->where[0];
                            if ( tdstop[-1] != 3 || tdstop[-2] != 1
                                || tdstop[-3] != 1 ) goto IllDollarExp;
                            tdstop -= 3;
                            if ( td >= tdstop ) goto IllDollarExp;
                            while ( td < tdstop ) {
                                if ( *td == SYMBOL ) {
                                    for ( dj = 2; dj < td[1]; dj += 2 ) {
                                        if ( td[dj+1] == 1 ) {
                                            *ppsym++ = td[dj];
                                            *ppsym++ = t[3];
                                            nsym++;
                                        }
                                        else if ( td[dj+1] == -1 ) {
```

```
                                        *ppsym++ = td[dj];
                                        *ppsym++ = -t[3];
                                        nsym++;
                                    }
                                    else goto IllDollarExp;
                                }
                            }
                            else if ( *td == DOTPRODUCT ) {
                                for ( dj = 2; dj < td[1]; dj += 3 ) {
                                    if ( td[dj+2] == 1 ) {
                                        *ppdot++ = td[dj];
                                        *ppdot++ = td[dj+1];
                                        *ppdot++ = t[3];
                                        ndot++;
                                    }
                                    else if ( td[dj+2] == -1 ) {
                                        *ppdot++ = td[dj];
                                        *ppdot++ = td[dj+1];
                                        *ppdot++ = -t[3];
                                        ndot++;
                                    }
                                    else goto IllDollarExp;
                                }
                            }
                            else goto IllDollarExp;
                            td += td[1];
                        }
                        regval = 2;
                        break;
                    }
                }
                t[0] = SUBEXPRESSION;
                t[4] = AM.dbufnum;
                if ( t[3] == 0 ) {
#ifdef WITHPTHREADS
                    if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                    break;
                }
                regval = 2;
                t = r;
                while ( t < m ) {
                    if ( *t == DOLLAREXPRESSION ) {
#ifdef WITHPTHREADS
                        if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                        d = Dollars + t[2];
#ifdef WITHPTHREADS
                        if ( AS.MultiThreaded ) {
                            for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                                if ( t[2] == ModOptdollars[nummodopt].number ) break;
                            }
                            if ( nummodopt < NumModOptdollars ) {
                                ptype = ModOptdollars[nummodopt].type;
                                if ( ptype == MODLOCAL ) {
                                    d = ModOptdollars[nummodopt].dstruct+AT.identity;
                                }
                                else {
                                    LOCK(d->pthreadslockread);
                                }
                            }
                        }
#endif
                        if ( d->type == DOLTERMS ) {
                            t[0] = SUBEXPRESSION;
                            t[4] = AM.dbufnum;
                        }
                    }
                    t += t[1];
                }
#ifdef WITHPTHREADS
                if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                goto RegEnd;
            }
            else {
#ifdef WITHPTHREADS
                if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                MLOCK(ErrorMessageLock);
                MesPrint("!!!This $ variation has not been implemented yet!!!");
                MUNLOCK(ErrorMessageLock);
                goto NormMin;
            }
#ifdef WITHPTHREADS
            if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
```

```
                }
                else {
                    pnco[nnco++] = t;
/*
                    The next statement should be safe as the value is used
                    only by the compiler (ie the master).
*/
                    AC.lhdollarflag = 1;
                }
                break;
            case DELTA :
                t += 2;
                do {
                    if ( *t < 0 ) {
                        if ( *t == SUMMEDIND ) {
                            if ( t[1] < -NMIN4SHIFT ) {
                                k = -t[1]-NMIN4SHIFT;
                                k = ExtraSymbol(k,1,nsym,ppsym,&ncoef);
                                nsym += k;
                                ppsym += (k « 1);
                            }
                            else if ( t[1] == 0 ) goto NormZero;
                            else {
                                if ( t[1] < 0 ) {
                                    lnum[0] = -t[1];
                                    nnum = -1;
                                }
                                else {
                                    lnum[0] = t[1];
                                    nnum = 1;
                                }
                                ncoef = REDLENG(ncoef);
                                if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
                                    goto FromNorm;
                                ncoef = INCLENG(ncoef);
                            }
                            t += 2;
                        }
                        else if ( *t == NOINDEX && t[1] == NOINDEX ) t += 2;
                        else if ( *t == EMPTYINDEX && t[1] == EMPTYINDEX ) {
                            *ppdel++ = *t++; *ppdel++ = *t++; ndel += 2;
                        }
                        else
                        if ( t[1] < 0 ) {
                            *ppdot++ = *t++; *ppdot++ = *t++; *ppdot++ = 1; ndot++;
                        }
                        else {
                            *ppvec++ = *t++; *ppvec++ = *t++; nvec += 2;
                        }
                    }
                    else {
                        if ( t[1] < 0 ) {
                            *ppvec++ = t[1]; *ppvec++ = *t; t+=2; nvec += 2;
                        }
                        else { *ppdel++ = *t++; *ppdel++ = *t++; ndel += 2; }
                    }
                } while ( t < r );
                break;
            case FACTORIAL :
/*
                (FACTORIAL,FUNHEAD+2,..,-SNUMBER,number)
*/
                if ( t[FUNHEAD] == -SNUMBER && t[1] == FUNHEAD+2
                                            && t[FUNHEAD+1] >= 0 ) {
                    if ( Factorial(BHEAD t[FUNHEAD+1],(UWORD *)lnum,&nnum) )
                        goto FromNorm;
MulIn:
                    ncoef = REDLENG(ncoef);
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) goto FromNorm;
                    ncoef = INCLENG(ncoef);
                }
                else pcom[ncom++] = t;
                break;
            case BERNOULLIFUNCTION :
/*
                (BERNOULLIFUNCTION,FUNHEAD+2,..,-SNUMBER,number)
*/
                if ( ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] >= 0 )
                && ( t[1] == FUNHEAD+2 || ( t[1] == FUNHEAD+4 &&
                t[FUNHEAD+2] == -SNUMBER && ABS(t[FUNHEAD+3]) == 1 ) ) ) {
                    WORD inum, mnum;
                    if ( Bernoulli(t[FUNHEAD+1],(UWORD *)lnum,&nnum) )
                        goto FromNorm;
                    if ( nnum == 0 ) goto NormZero;
                    inum = nnum; if ( inum < 0 ) inum = -inum;
                    inum--; inum /= 2;
                    mnum = inum;
```

```
                    while ( lnum[mnum-1] == 0 ) mnum--;
                    if ( nnum < 0 ) mnum = -mnum;
                    ncoef = REDLENG(ncoef);
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,mnum) ) goto FromNorm;
                    mnum = inum;
                    while ( lnum[inum+mnum-1] == 0 ) mnum--;
                    if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(lnum+inum),mnum) ) goto FromNorm;
                    ncoef = INCLENG(ncoef);
                    if ( t[1] == FUNHEAD+4 && t[FUNHEAD+1] == 1
                     && t[FUNHEAD+3] == -1 ) ncoef = -ncoef;
                }
                else pcom[ncom++] = t;
                break;
            case NUMARGSFUN:
/*
                Numerical function giving the number of arguments.
*/
                k = 0;
                t += FUNHEAD;
                while ( t < r ) {
                    k++;
                    NEXTARG(t);
                }
                if ( k == 0 ) goto NormZero;
                *((UWORD *)lnum) = k;
                nnum = 1;
                goto MulIn;
            case NUMFACTORS:
/*
                Numerical function giving the number of factors in an expression.
*/
                t += FUNHEAD;
                if ( *t == -EXPRESSION ) {
                    k = AS.OldNumFactors[t[1]];
                }
                else if ( *t == -DOLLAREXPRESSION ) {
                    k = Dollars[t[1]].nfactors;
                }
                else {
                    pcom[ncom++] = t;
                    break;
                }
                if ( k == 0 ) goto NormZero;
                *((UWORD *)lnum) = k;
                nnum = 1;
                goto MulIn;
            case NUMTERMSFUN:
/*
                Numerical function giving the number of terms in the single argument.
*/
                if ( t[FUNHEAD] < 0 ) {
                    if ( t[FUNHEAD] <= -FUNCTION && t[1] == FUNHEAD+1 ) break;
                    if ( t[FUNHEAD] > -FUNCTION && t[1] == FUNHEAD+2 ) {
                        if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) goto NormZero;
                        break;
                    }
                    pcom[ncom++] = t;
                    break;
                }
                if ( t[FUNHEAD] > 0 && t[FUNHEAD] == t[1]-FUNHEAD ) {
                    k = 0;
                    t += FUNHEAD+ARGHEAD;
                    while ( t < r ) {
                        k++;
                        t += *t;
                    }
                    if ( k == 0 ) goto NormZero;
                    *((UWORD *)lnum) = k;
                    nnum = 1;
                    goto MulIn;
                }
                else pcom[ncom++] = t;
                break;
            case COUNTFUNCTION:
                if ( AN.cTerm ) {
                    k = CountFun(AN.cTerm,t);
                }
                else {
                    k = CountFun(term,t);
                }
                if ( k == 0 ) goto NormZero;
                if ( k > 0 ) { *((UWORD *)lnum) = k; nnum = 1; }
                else { *((UWORD *)lnum) = -k; nnum = -1; }
                goto MulIn;
                break;
            case MAKERATIONAL:
```

```
                if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+2] == -SNUMBER
                    && t[1] == FUNHEAD+4 && t[FUNHEAD+3] > 1 ) {
                    WORD x1[2], sgn;
                    if ( t[FUNHEAD+1] == 0 ) goto NormZero;
                    if ( t[FUNHEAD+1] < 0 ) { t[FUNHEAD+1] = -t[FUNHEAD+1]; sgn = -1; }
                    else sgn = 1;
                    if ( MakeRational(t[FUNHEAD+1],t[FUNHEAD+3],x1,x1+1) ) {
                        static int warnflag = 1;
                        if ( warnflag ) {
                            MesPrint("%w Warning: fraction could not be reconstructed in MakeRational_");
                            warnflag = 0;
                        }
                        x1[0] = t[FUNHEAD+1]; x1[1] = 1;
                    }
                    if ( sgn < 0 ) { t[FUNHEAD+1] = -t[FUNHEAD+1]; x1[0] = -x1[0]; }
                    if ( x1[0] < 0 ) { sgn = -1; x1[0] = -x1[0]; }
                    else sgn = 1;
                    ncoef = REDLENG(ncoef);
                    if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)x1,sgn,
                    (UWORD *)n_coef,&ncoef) ) goto FromNorm;
                    ncoef = INCLENG(ncoef);
                }
                else {
                    WORD narg = 0, *tt, *ttstop, *arg1 = 0, *arg2 = 0;
                    UWORD *x1, *x2, *xx;
                    WORD nx1,nx2,nxx;
                    ttstop = t + t[1]; tt = t+FUNHEAD;
                    while ( tt < ttstop ) {
                        narg++;
                        if ( narg == 1 ) arg1 = tt;
                        else arg2 = tt;
                        NEXTARG(tt);
                    }
                    if ( narg != 2 ) goto defaultcase;
                    if ( *arg2 == -SNUMBER && arg2[1] <= 1 ) goto defaultcase;
                    else if ( *arg2 > 0 && ttstop[-1] < 0 ) goto defaultcase;
                    x1 = NumberMalloc("Norm-MakeRational");
                    if ( *arg1 == -SNUMBER ) {
                        if ( arg1[1] == 0 ) {
                            NumberFree(x1,"Norm-MakeRational");
                            goto NormZero;
                        }
                        if ( arg1[1] < 0 ) { x1[0] = -arg1[1]; nx1 = -1; }
                        else { x1[0] = arg1[1]; nx1 = 1; }
                    }
                    else if ( *arg1 > 0 ) {
                        WORD *tc;
                        nx1 = (ABS(arg2[-1])-1)/2;
                        tc = arg1+ARGHEAD+1+nx1;
                        if ( tc[0] != 1 ) {
                            NumberFree(x1,"Norm-MakeRational");
                            goto defaultcase;
                        }
                        for ( i = 1; i < nx1; i++ ) if ( tc[i] != 0 ) {
                            NumberFree(x1,"Norm-MakeRational");
                            goto defaultcase;
                        }
                        tc = arg1+ARGHEAD+1;
                        for ( i = 0; i < nx1; i++ ) x1[i] = tc[i];
                        if ( arg2[-1] < 0 ) nx1 = -nx1;
                    }
                    else {
                        NumberFree(x1,"Norm-MakeRational");
                        goto defaultcase;
                    }
                    x2 = NumberMalloc("Norm-MakeRational");
                    if ( *arg2 == -SNUMBER ) {
                        if ( arg2[1] <= 1 ) {
                            NumberFree(x2,"Norm-MakeRational");
                            NumberFree(x1,"Norm-MakeRational");
                            goto defaultcase;
                        }
                        else { x2[0] = arg2[1]; nx2 = 1; }
                    }
                    else if ( *arg2 > 0 ) {
                        WORD *tc;
                        nx2 = (ttstop[-1]-1)/2;
                        tc = arg2+ARGHEAD+1+nx2;
                        if ( tc[0] != 1 ) {
                            NumberFree(x2,"Norm-MakeRational");
                            NumberFree(x1,"Norm-MakeRational");
                            goto defaultcase;
                        }
                        for ( i = 1; i < nx2; i++ ) if ( tc[i] != 0 ) {
                            NumberFree(x2,"Norm-MakeRational");
                            NumberFree(x1,"Norm-MakeRational");
```

```
                            goto defaultcase;
                        }
                        tc = arg2+ARGHEAD+1;
                        for ( i = 0; i < nx2; i++ ) x2[i] = tc[i];
                    }
                    else {
                        NumberFree(x2,"Norm-MakeRational");
                        NumberFree(x1,"Norm-MakeRational");
                        goto defaultcase;
                    }
                    if ( BigLong(x1,ABS(nx1),x2,nx2) >= 0 ) {
                        UWORD *x3 = NumberMalloc("Norm-MakeRational");
                        UWORD *x4 = NumberMalloc("Norm-MakeRational");
                        WORD nx3, nx4;
                        DivLong(x1,nx1,x2,nx2,x3,&nx3,x4,&nx4);
                        for ( i = 0; i < ABS(nx4); i++ ) x1[i] = x4[i];
                        nx1 = nx4;
                        NumberFree(x4,"Norm-MakeRational");
                        NumberFree(x3,"Norm-MakeRational");
                    }
                    xx = (UWORD *)(TermMalloc("Norm-MakeRational"));
                    if ( MakeLongRational(BHEAD x1,nx1,x2,nx2,xx,&nxx) ) {
                        static int warnflag = 1;
                        if ( warnflag ) {
                            MesPrint("%w Warning: fraction could not be reconstructed in MakeRational_");
                            warnflag = 0;
                        }
                        ncoef = REDLENG(ncoef);
                        if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,x1,nx1) )
                            goto FromNorm;
                    }
                    else {
                        ncoef = REDLENG(ncoef);
                        if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,xx,nxx,
                        (UWORD *)n_coef,&ncoef) ) goto FromNorm;
                    }
                    ncoef = INCLENG(ncoef);
                    TermFree(xx,"Norm-MakeRational");
                    NumberFree(x2,"Norm-MakeRational");
                    NumberFree(x1,"Norm-MakeRational");
                }
                break;
            case TERMFUNCTION:
                if ( t[1] == FUNHEAD && AN.cTerm ) {
                    ANsr = r; ANsm = m; ANsc = AN.cTerm;
                    AN.cTerm = 0;
                    t = ANsc + 1;
                    m = ANsc + *ANsc;
                    ncoef = REDLENG(ncoef);
                    nnum = REDLENG(m[-1]);
                    m -= ABS(m[-1]);
                    if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)m,nnum,
                    (UWORD *)n_coef,&ncoef) ) goto FromNorm;
                    ncoef = INCLENG(ncoef);
                    r = t;
                }
                break;
            case FIRSTBRACKET:
                if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
                    if ( GetFirstBracket(termout,t[FUNHEAD+1]) < 0 ) goto FromNorm;
                    if ( *termout == 0 ) goto NormZero;
                    if ( *termout > 4 ) {
                        WORD *r1, *r2, *r3;
                        while ( r < m ) *t++ = *r++;
                        r1 = term + *term;
                        r2 = termout + *termout; r2 -= ABS(r2[-1]);
                        while ( r < r1 ) *r2++ = *r++;
                        r3 = termout + 1;
                        while ( r3 < r2 ) *t++ = *r3++; *term = t - term;
                        if ( AT.WorkPointer > term && AT.WorkPointer < t )
                            AT.WorkPointer = t;
                        goto Restart;
                    }
                }
                break;
            case FIRSTTERM:
            case CONTENTTERM:
                if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
                    {
                        EXPRESSIONS e = Expressions+t[FUNHEAD+1];
                        POSITION oldondisk = AS.OldOnFile[t[FUNHEAD+1]];
                        if ( e->replace == NEWLYDEFINEDEXPRESSION ) {
                            AS.OldOnFile[t[FUNHEAD+1]] = e->onfile;
                        }
                        if ( *t == FIRSTTERM ) {
                            if ( GetFirstTerm(termout,t[FUNHEAD+1]) < 0 ) goto FromNorm;
```

```
                        }
                        else if ( *t == CONTENTTERM ) {
                            if ( GetContent(termout,t[FUNHEAD+1]) < 0 ) goto FromNorm;
                        }
                        AS.OldOnFile[t[FUNHEAD+1]] = oldondisk;
                        if ( *termout == 0 ) goto NormZero;
                    }
PasteIn:;
                    {
                        WORD *r1, *r2, *r3, *r4, *r5, nr1, *rterm;
                        r2 = termout + *termout; lnum = r2 - ABS(r2[-1]);
                        nnum = REDLENG(r2[-1]);

                        r1 = term + *term; r3 = r1 - ABS(r1[-1]);
                        nr1 = REDLENG(r1[-1]);
                        if ( Mully(BHEAD (UWORD *)lnum,&nnum,(UWORD *)r3,nr1) ) goto FromNorm;
                        nnum = INCLENG(nnum); nr1 = ABS(nnum); lnum[nr1-1] = nnum;
                        rterm = TermMalloc("FirstTerm/ContentTerm");
                        r4 = rterm+1; r5 = term+1; while ( r5 < t ) *r4++ = *r5++;
                        r5 = termout+1; while ( r5 < lnum ) *r4++ = *r5++;
                        r5 = r; while ( r5 < r3 ) *r4++ = *r5++;
                        r5 = lnum; NCOPY(r4,r5,nr1);
                        *rterm = r4-rterm;
                        nr1 = *rterm; r1 = term; r2 = rterm; NCOPY(r1,r2,nr1);
                        TermFree(rterm,"FirstTerm/ContentTerm");
                        if ( AT.WorkPointer > term && AT.WorkPointer < r1 )
                            AT.WorkPointer = r1;
                        goto Restart;
                    }
                }
                else if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -DOLLAREXPRESSION ) {
                    DOLLARS d = Dollars + t[FUNHEAD+1], newd = 0;
                    int idol, ido;
#ifdef WITHPTHREADS
                    int nummodopt, dtype = -1;
                    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
                        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                            if ( t[FUNHEAD+1] == ModOptdollars[nummodopt].number ) break;
                        }
                        if ( nummodopt < NumModOptdollars ) {
                            dtype = ModOptdollars[nummodopt].type;
                            if ( dtype == MODLOCAL ) {
                                d = ModOptdollars[nummodopt].dstruct+AT.identity;
                            }
                        }
                    }
#endif
                    if ( d->where && ( d->type == DOLTERMS || d->type == DOLNUMBER ) ) {
                        newd = d;
                    }
                    else {
                        if ( ( newd = DolToTerms(BHEAD t[FUNHEAD+1]) ) == 0 )
                            goto NormZero;
                    }
                    if ( newd->where[0] == 0 ) {
                        M_free(newd,"Copy of dollar variable");
                        goto NormZero;
                    }
                    if ( *t == FIRSTTERM ) {
                        idol = newd->where[0];
                        for ( ido = 0; ido < idol; ido++ ) termout[ido] = newd->where[ido];
                    }
                    else if ( *t == CONTENTTERM ) {
                        WORD *tterm;
                        tterm = newd->where;
                        idol = tterm[0];
                        for ( ido = 0; ido < idol; ido++ ) termout[ido] = tterm[ido];
                        tterm += *tterm;
                        while ( *tterm ) {
                            if ( ContentMerge(BHEAD termout,tterm) < 0 ) goto FromNorm;
                            tterm += *tterm;
                        }
                    }
                    if ( newd != d ) {
                        if ( newd->factors ) M_free(newd->factors,"Dollar factors");
                        M_free(newd,"Copy of dollar variable");
                        newd = 0;
                    }
                    goto PasteIn;
                }
                break;
            case TERMSINEXPR:
                { LONG x;
                if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
                    x = TermsInExpression(t[FUNHEAD+1]);
multermnum:         if ( x == 0 ) goto NormZero;
                    if ( x < 0 ) {
```

```
                        x = -x;
                        if ( x > (LONG)WORDMASK ) { lnum[0] = x & WORDMASK;
                            lnum[1] = x » BITSINWORD; nnum = -2;
                        }
                        else { lnum[0] = x; nnum = -1; }
                    }
                    else if ( x > (LONG)WORDMASK ) {
                        lnum[0] = x & WORDMASK;
                        lnum[1] = x » BITSINWORD;
                        nnum = 2;
                    }
                    else { lnum[0] = x; nnum = 1; }
                    ncoef = REDLENG(ncoef);
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
                        goto FromNorm;
                    ncoef = INCLENG(ncoef);
                }
                else if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -DOLLAREXPRESSION ) {
                    x = TermsInDollar(t[FUNHEAD+1]);
                    goto multermnum;
                }
                else { pcom[ncom++] = t; }
                }
                break;
            case MATCHFUNCTION:
            case PATTERNFUNCTION:
                break;
            case BINOMIAL:
/*
                Binomial function for internal use for the moment.
                The routine in reken.c should be more efficient.
*/
                if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SNUMBER
                && t[FUNHEAD+1] >= 0 && t[FUNHEAD+2] == -SNUMBER
                && t[FUNHEAD+3] >= 0 && t[FUNHEAD+1] >= t[FUNHEAD+3] ) {
                    if ( t[FUNHEAD+1] > t[FUNHEAD+3] ) {
                        if ( GetBinom((UWORD *)lnum,&nnum,
                        t[FUNHEAD+1],t[FUNHEAD+3]) ) goto FromNorm;
                        if ( nnum == 0 ) goto NormZero;
                        goto MulIn;
                    }
                }
                else pcom[ncom++] = t;
                break;
            case SIGNFUN:
/*
                Numerical function giving (-1)^arg
*/
                if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
                    if ( ( t[FUNHEAD+1] & 1 ) != 0 ) ncoef = -ncoef;
                }
                else if ( ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+t[FUNHEAD] )
                && ( t[FUNHEAD] == ARGHEAD+1+abs(t[t[1]-1]) ) ) {
                    UWORD *numer1,*denom1;
                    WORD nsize = abs(t[t[1]-1]), nnsize, isize;
                    nnsize = (nsize-1)/2;
                    numer1 = (UWORD *)(t + FUNHEAD+ARGHEAD+1);
                    denom1 = numer1 + nnsize;
                    for ( isize = 1; isize < nnsize; isize++ ) {
                        if ( denom1[isize] ) break;
                    }
                    if ( ( denom1[0] != 1 ) || isize < nnsize ) {
                        pcom[ncom++] = t;
                    }
                    else {
                        if ( ( numer1[0] & 1 ) != 0 ) ncoef = -ncoef;
                    }
                }
                else {
                    goto doflags;
/*                  pcom[ncom++] = t; */
                }
                break;
            case SIGFUNCTION:
/*
                Numerical function giving the sign of the numerical argument
                The sign of zero is 1.
                If there are roots of unity they are part of the sign.
*/
                if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
                    if ( t[FUNHEAD+1] < 0 ) ncoef = -ncoef;
                }
                else if ( ( t[1] == FUNHEAD+2 ) && ( t[FUNHEAD] == -SYMBOL )
                    && ( ( t[FUNHEAD+1] <= NumSymbols && t[FUNHEAD+1] > -MAXPOWER )
                    && ( symbols[t[FUNHEAD+1]].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY )
    ) {
                    k = t[FUNHEAD+1];
```

```
                    from = m;
                    i = nsym;
                    m = ppsym;
                    if ( i > 0 ) do {
                        m -= 2;
                        if  ( k == *m ) {
                            m++;
                            *m = *m + 1;
                            *m %= symbols[k].maxpower;
                            if ( ( symbols[k].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
                                if ( ( ( symbols[k].maxpower & 1 ) == 0 ) &&
                                ( *m >= symbols[k].maxpower/2 ) ) {
                                    *m -= symbols[k].maxpower/2; ncoef = -ncoef;
                                }
                            }
                            if ( !*m ) {
                                m--;
                                while ( i < nsym )
                                    { *m = m[2]; m++; *m = m[2]; m++; i++; }
                                ppsym -= 2;
                                nsym--;
                            }
                            goto sigDoneSymbol;
                        }
                    } while ( k < *m && --i > 0 );
                    m = ppsym;
                    while ( i < nsym )
                        { m--; m[2] = *m; m--; m[2] = *m; i++; }
                    *m++ = k;
                    *m = 1;
                    ppsym += 2;
                    nsym++;
sigDoneSymbol:;
                    m = from;
                }
                else if ( ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+t[FUNHEAD] ) ) {
                    if ( t[FUNHEAD] == ARGHEAD+1+abs(t[t[1]-1]) ) {
                        if ( t[t[1]-1] < 0 ) ncoef = -ncoef;
                    }
/*
                    Now we should fish out the roots of unity
*/
                    else if ( ( t[FUNHEAD+ARGHEAD]+FUNHEAD+ARGHEAD == t[1] )
                    && ( t[FUNHEAD+ARGHEAD+1] == SYMBOL ) ) {
                        WORD *ts = t + FUNHEAD+ARGHEAD+3;
                        WORD its = ts[-1]-2;
                        while ( its > 0 ) {
                            if ( ( *ts != 0 ) && ( ( *ts > NumSymbols || *ts <= -MAXPOWER )
                             || ( symbols[*ts].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
) {
                                goto signogood;
                            }
                            ts += 2; its -= 2;
                        }
/*
                        Now we have only roots of unity which should be
                        registered in the list of sysmbols.
*/
                        if ( t[t[1]-1] < 0 ) ncoef = -ncoef;
                        ts = t + FUNHEAD+ARGHEAD+3;
                        its = ts[-1]-2;
                        from = m;
                        while ( its > 0 ) {
                            i = nsym;
                            m = ppsym;
                            if ( i > 0 ) do {
                                m -= 2;
                                if  ( *ts == *m ) {
                                    ts++; m++;
                                    *m += *ts;
                                    if ( ( ts[-1] <= NumSymbols && ts[-1] > -MAXPOWER ) &&
                                     ( symbols[ts[-1]].complex & VARTYPEROOTOFUNITY ) ==
VARTYPEROOTOFUNITY ) {
                                        *m %= symbols[ts[-1]].maxpower;
                                        if ( *m < 0 ) *m += symbols[ts[-1]].maxpower;
                                        if ( ( symbols[ts[-1]].complex & VARTYPEMINUS ) ==
VARTYPEMINUS ) {
                                            if ( ( ( symbols[ts[-1]].maxpower & 1 ) == 0 ) &&
                                            ( *m >= symbols[ts[-1]].maxpower/2 ) ) {
                                                *m -= symbols[ts[-1]].maxpower/2; ncoef = -ncoef;
                                            }
                                        }
                                    }
                                    if ( !*m ) {
                                        m--;
                                        while ( i < nsym )
                                            { *m = m[2]; m++; *m = m[2]; m++; i++; }
```

```
                                        ppsym -= 2;
                                        nsym--;
                                    }
                                    ts++; its -= 2;
                                    goto sigNextSymbol;
                                }
                            } while ( *ts < *m && --i > 0 );
                            m = ppsym;
                            while ( i < nsym )
                                { m--; m[2] = *m; m--; m[2] = *m; i++; }
                            *m++ = *ts++;
                            *m = *ts++;
                            ppsym += 2;
                            nsym++; its -= 2;
sigNextSymbol:;
                        }
                        m = from;
                    }
                    else {
signogood:              pcom[ncom++] = t;
                    }
                }
                else pcom[ncom++] = t;
                break;
            case ABSFUNCTION:
/*
                Numerical function giving the absolute value of the
                numerical argument. Or roots of unity.
*/
                if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
                    k = t[FUNHEAD+1];
                    if ( k < 0 ) k = -k;
                    if ( k == 0 ) goto NormZero;
                    *((UWORD *)lnum) = k; nnum = 1;
                    goto MulIn;

                }
                else if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SYMBOL ) {
                    k = t[FUNHEAD+1];
                    if ( ( k > NumSymbols || k <= -MAXPOWER )
                     || ( symbols[k].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
                        goto absnogood;
                }
                else if ( ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+t[FUNHEAD] )
                && ( t[1] == FUNHEAD+ARGHEAD+t[FUNHEAD+ARGHEAD] ) ) {
                    if ( t[FUNHEAD] == ARGHEAD+1+abs(t[t[1]-1]) ) {
                        WORD *ts;
absnosymbols:           ts = t + t[1] -1;
                        ncoef = REDLENG(ncoef);
                        nnum = REDLENG(*ts);
                        if ( nnum < 0 ) nnum = -nnum;
                        if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,
                        (UWORD *)(ts-ABS(*ts)+1),nnum,
                        (UWORD *)n_coef,&ncoef) ) goto FromNorm;
                        ncoef = INCLENG(ncoef);
                    }
/*
                    Now get rid of the roots of unity. This includes i_
*/
                    else if ( t[FUNHEAD+ARGHEAD+1] == SYMBOL ) {
                        WORD *ts = t+FUNHEAD+ARGHEAD+1;
                        WORD its = ts[1] - 2;
                        ts += 2;
                        while ( its > 0 ) {
                            if ( *ts == 0 ) { }
                            else if ( ( *ts > NumSymbols || *ts <= -MAXPOWER )
                             || ( symbols[*ts].complex & VARTYPEROOTOFUNITY )
                                != VARTYPEROOTOFUNITY ) goto absnogood;
                            its -= 2; ts += 2;
                        }
                        goto absnosymbols;
                    }
                    else {
absnogood:                  pcom[ncom++] = t;
                    }
                }
                else pcom[ncom++] = t;
                break;
            case MODFUNCTION:
            case MOD2FUNCTION:
/*
                Mod function. Does work if two arguments and the
                second argument is a positive short number
*/
                if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SNUMBER
                    && t[FUNHEAD+2] == -SNUMBER && t[FUNHEAD+3] > 1 ) {
                    WORD tmod;
```

```
                    tmod = (t[FUNHEAD+1]%t[FUNHEAD+3]);
                    if ( tmod < 0 ) tmod += t[FUNHEAD+3];
                    if ( *t == MOD2FUNCTION && tmod > t[FUNHEAD+3]/2 )
                            tmod -= t[FUNHEAD+3];
                    if ( tmod < 0 ) {
                        *((UWORD *)lnum) = -tmod;
                        nnum = -1;
                    }
                    else if ( tmod > 0 ) {
                        *((UWORD *)lnum) = tmod;
                        nnum = 1;
                    }
                    else goto NormZero;
                    goto MulIn;
                }
                else if ( t[1] > t[FUNHEAD+2] && t[FUNHEAD] > 0
                && t[FUNHEAD+t[FUNHEAD]] == -SNUMBER
                && t[FUNHEAD+t[FUNHEAD]+1] > 1
                && t[1] == FUNHEAD+2+t[FUNHEAD] ) {
                    WORD *ttt = t+FUNHEAD, iii;
                    iii = ttt[*ttt-1];
                    if ( *ttt == ttt[ARGHEAD]+ARGHEAD &&
                        ttt[ARGHEAD] == ABS(iii)+1 ) {
                        WORD ncmod = 1;
                        WORD cmod = ttt[*ttt+1];
                        iii = REDLENG(iii);
                        if ( *t == MODFUNCTION ) {
                            if ( TakeModulus((UWORD *)(ttt+ARGHEAD+1)
                            ,&iii,(UWORD *)(&cmod),ncmod,UNPACK|NOINVERSES) )
                                goto FromNorm;
                        }
                        else {
                            if ( TakeModulus((UWORD *)(ttt+ARGHEAD+1)
                            ,&iii,(UWORD *)(&cmod),ncmod,UNPACK|POSNEG|NOINVERSES) )
                                goto FromNorm;
                        }
                        if ( *t == MOD2FUNCTION && ttt[ARGHEAD+1] > cmod/2 && iii > 0 ) {
                            ttt[ARGHEAD+1] -= cmod;
                        }
                        if ( ttt[ARGHEAD+1] < 0 ) {
                            *((UWORD *)lnum) = -ttt[ARGHEAD+1];
                            nnum = -1;
                        }
                        else if ( ttt[ARGHEAD+1] > 0 ) {
                            *((UWORD *)lnum) = ttt[ARGHEAD+1];
                            nnum = 1;
                        }
                        else goto NormZero;
                        goto MulIn;
                    }
                }
                else if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
                    *((UWORD *)lnum) = t[FUNHEAD+1];
                    if ( *lnum == 0 ) goto NormZero;
                    nnum = 1;
                    goto MulIn;
                }
                else if ( ( ( t[FUNHEAD] < 0 ) && ( t[FUNHEAD] == -SNUMBER )
                && ( t[1] >= ( FUNHEAD+6+ARGHEAD ) )
                && ( t[FUNHEAD+2] >= 4+ARGHEAD )
                && ( t[t[1]-1] == t[FUNHEAD+2+ARGHEAD]-1 ) ) ||
                ( ( t[FUNHEAD] > 0 )
                && ( t[FUNHEAD]-ARGHEAD-1 == ABS(t[FUNHEAD+t[FUNHEAD]-1]) )
                && ( t[FUNHEAD+t[FUNHEAD]]-ARGHEAD-1 == t[t[1]-1] ) ) ) {
/*
                    Check that the last (long) number is integer
*/
                    WORD *ttt = t + t[1], iii, iii1;
                    UWORD coefbuf[2], *coef2, ncoef2;
                    iii = (ttt[-1]-1)/2;
                    ttt -= iii;
                    if ( ttt[-1] != 1 ) {
exitfromhere:
                        pcom[ncom++] = t;
                        break;
                    }
                    iii--;
                    for ( iii1 = 0; iii1 < iii; iii1++ ) {
                        if ( ttt[iii1] != 0 ) goto exitfromhere;
                    }
/*
                    Now we have a hit!
                    The first argument will be put in lnum.
                    It will be a rational.
                    The second argument will be a long integer in coef2.
*/
                    ttt = t + FUNHEAD;
```

```
                    if ( *ttt < 0 ) {
                        if ( ttt[1] < 0 ) {
                            nnum = -1; lnum[0] = -ttt[1]; lnum[1] = 1;
                        }
                        else {
                            nnum = 1; lnum[0] = ttt[1]; lnum[1] = 1;
                        }
                    }
                    else {
                        nnum = ABS(ttt[ttt[0]-1] - 1);
                        for ( iii = 0; iii < nnum; iii++ ) {
                            lnum[iii] = ttt[ARGHEAD+1+iii];
                        }
                        nnum = nnum/2;
                        if ( ttt[ttt[0]-1] < 0 ) nnum = -nnum;
                    }
                    NEXTARG(ttt);
                    if ( *ttt < 0 ) {
                        coef2 = coefbuf;
                        ncoef2 = 3; *coef2 = (UWORD)(ttt[1]);
                        coef2[1] = 1;
                    }
                    else {
                        coef2 = (UWORD *)(ttt+ARGHEAD+1);
                        ncoef2 = (ttt[ttt[0]-1]-1)/2;
                    }
                    if ( TakeModulus((UWORD *)lnum,&nnum,(UWORD *)coef2,ncoef2,
                                    UNPACK|NOINVERSES|FROMFUNCTION) ) {
                        goto FromNorm;
                    }
                    if ( *t == MOD2FUNCTION && nnum > 0 ) {
                        UWORD *coef3 = NumberMalloc("Mod2Function"), two = 2;
                        WORD ncoef3;
                        if ( MulLong((UWORD *)lnum,nnum,&two,1,coef3,&ncoef3) )
                            goto FromNorm;
                        if ( BigLong(coef3,ncoef3,(UWORD *)coef2,ncoef2) > 0 ) {
                            nnum = -nnum;
                            AddLong((UWORD *)lnum,nnum,(UWORD *)coef2,ncoef2
                                    ,(UWORD *)lnum,&nnum);
                            nnum = -nnum;
                        }
                        NumberFree(coef3,"Mod2Function");
                    }
/*
                    Do we have to pack? No, because the answer is not a fraction
*/
                    ncoef = REDLENG(ncoef);
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
                            goto FromNorm;
                    ncoef = INCLENG(ncoef);
                }
                else pcom[ncom++] = t;
                break;
            case EXTEUCLIDEAN:
                {
                    WORD argcount = 0, *tc, *ts, xc, xs, *tcc;
                    UWORD *Num1, *Num2, *Num3, *Num4;
                    WORD size1, size2, size3, size4, space;
                    tc = t+FUNHEAD; ts = t + t[1];
                    while ( argcount < 3 && tc < ts ) { NEXTARG(tc); argcount++; }
                    if ( argcount != 2 ) goto defaultcase;
                    if ( t[FUNHEAD] == -SNUMBER ) {
                        if ( t[FUNHEAD+1] <= 1 ) goto defaultcase;
                        if ( t[FUNHEAD+2] == -SNUMBER ) {
                            if ( t[FUNHEAD+3] <= 1 ) goto defaultcase;
                            Num2 = NumberMalloc("modinverses");
                            *Num2 = t[FUNHEAD+3]; size2 = 1;
                        }
                        else {
                            if ( ts[-1] < 0 ) goto defaultcase;
                            if ( ts[-1] != t[FUNHEAD+2]-ARGHEAD-1 ) goto defaultcase;
                            xs = (ts[-1]-1)/2;
                            tcc = ts-xs-1;
                            if ( *tcc != 1 ) goto defaultcase;
                            for ( i = 1; i < xs; i++ ) {
                                if ( tcc[i] != 0 ) goto defaultcase;
                            }
                            Num2 = NumberMalloc("modinverses");
                            size2 = xs;
                            for ( i = 0; i < xs; i++ ) Num2[i] = t[FUNHEAD+ARGHEAD+3+i];
                        }
                        Num1 = NumberMalloc("modinverses");
                        *Num1 = t[FUNHEAD+1]; size1 = 1;
                    }
                    else {
                        tc = t + FUNHEAD + t[FUNHEAD];
                        if ( tc[-1] < 0 ) goto defaultcase;
```

```
                        if ( tc[-1] != t[FUNHEAD]-ARGHEAD-1 ) goto defaultcase;
                        xc = (tc[-1]-1)/2;
                        tcc = tc-xc-1;
                        if ( *tcc != 1 ) goto defaultcase;
                        for ( i = 1; i < xc; i++ ) {
                            if ( tcc[i] != 0 ) goto defaultcase;
                        }
                        if ( *tc == -SNUMBER ) {
                            if ( tc[1] <= 1 ) goto defaultcase;
                            Num2 = NumberMalloc("modinverses");
                            *Num2 = tc[1]; size2 = 1;
                        }
                        else {
                            if ( ts[-1] < 0 ) goto defaultcase;
                            if ( ts[-1] != t[FUNHEAD+2]-ARGHEAD-1 ) goto defaultcase;
                            xs = (ts[-1]-1)/2;
                            tcc = ts-xs-1;
                            if ( *tcc != 1 ) goto defaultcase;
                            for ( i = 1; i < xs; i++ ) {
                                if ( tcc[i] != 0 ) goto defaultcase;
                            }
                            Num2 = NumberMalloc("modinverses");
                            size2 = xs;
                            for ( i = 0; i < xs; i++ ) Num2[i] = tc[ARGHEAD+1+i];
                        }
                        Num1 = NumberMalloc("modinverses");
                        size1 = xc;
                        for ( i = 0; i < xc; i++ ) Num1[i] = t[FUNHEAD+ARGHEAD+1+i];
                    }
                    Num3 = NumberMalloc("modinverses");
                    Num4 = NumberMalloc("modinverses");
                    GetLongModInverses(BHEAD Num1,size1,Num2,size2
                                ,Num3,&size3,Num4,&size4);
/*
                    Now we have to compose the answer. This needs more space
                    and hence we have to put this inside the term.
                    Compute first how much extra space we need.
                    Then move the trailing part of the term upwards.
                    Do not forget relevant pointers!!! (r, m, termout, AT.WorkPointer)
*/
                    space = 0;
                    if ( ( size3 == 1 || size3 == -1 ) && (*Num3&TOPBITONLY) == 0 ) space += 2;
                    else space += ARGHEAD + 2*ABS(size3) + 2;
                    if ( ( size4 == 1 || size4 == -1 ) && (*Num4&TOPBITONLY) == 0 ) space += 2;
                    else space += ARGHEAD + 2*ABS(size4) + 2;
                    tt = term + *term; u = tt + space;
                    while ( tt >= ts ) *--u = *--tt;
                    m += space; r += space;
                    *term += space;
                    t[1] += space;
                    if ( ( size3 == 1 || size3 == -1 ) && (*Num3&TOPBITONLY) == 0 ) {
                        *ts++ = -SNUMBER; *ts = (WORD)(*Num3);
                        if ( size3 < 0 ) *ts = -*ts;
                        ts++;
                    }
                    else {
                        *ts++ = 2*ABS(size3)+ARGHEAD+2;
                        *ts++ = 0; FILLARG(ts)
                        *ts++ = 2*ABS(size3)+1;
                        for ( i = 0; i < ABS(size3); i++ ) *ts++ = Num3[i];
                        *ts++ = 1;
                        for ( i = 1; i < ABS(size3); i++ ) *ts++ = 0;
                        if ( size3 < 0 ) *ts++ = 2*size3-1;
                        else             *ts++ = 2*size3+1;
                    }
                    if ( ( size4 == 1 || size4 == -1 ) && (*Num4&TOPBITONLY) == 0 ) {
                        *ts++ = -SNUMBER; *ts = *Num4;
                        if ( size4 < 0 ) *ts = -*ts;
                        ts++;
                    }
                    else {
                        *ts++ = 2*ABS(size4)+ARGHEAD+2;
                        *ts++ = 0; FILLARG(ts)
                        *ts++ = 2*ABS(size4)+2;
                        for ( i = 0; i < ABS(size4); i++ ) *ts++ = Num4[i];
                        *ts++ = 1;
                        for ( i = 1; i < ABS(size4); i++ ) *ts++ = 0;
                        if ( size4 < 0 ) *ts++ = 2*size4-1;
                        else             *ts++ = 2*size4+1;
                    }
                    NumberFree(Num4,"modinverses");
                    NumberFree(Num3,"modinverses");
                    NumberFree(Num1,"modinverses");
                    NumberFree(Num2,"modinverses");
                    t[2] = 0; /* mark function as clean. */
                    goto Restart;
                }
```

```
                break;
            case GCDFUNCTION:
#ifdef EVALUATEGCD
#ifdef NEWGCDFUNCTION
                {
/*
                Has two integer arguments
                Four cases: S,S, S,L, L,S, L,L
*/
                WORD *num1, *num2, size1, size2, stor1, stor2, *ttt, ti;
                if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SNUMBER
                && t[FUNHEAD+2] == -SNUMBER && t[FUNHEAD+1] != 0
                && t[FUNHEAD+3] != 0 ) {    /* Short,Short */
                    stor1 = t[FUNHEAD+1];
                    stor2 = t[FUNHEAD+3];
                    if ( stor1 < 0 ) stor1 = -stor1;
                    if ( stor2 < 0 ) stor2 = -stor2;
                    num1 = &stor1; num2 = &stor2;
                    size1 = size2 = 1;
                    goto gcdcalc;
                }
                else if ( t[1] > FUNHEAD+4 ) {
                    if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] != 0
                    && t[FUNHEAD+2] == t[1]-FUNHEAD-2 &&
                    ABS(t[t[1]-1]) == t[FUNHEAD+2]-1-ARGHEAD ) { /* Short,Long */
                        num2 = t + t[1];
                        size2 = ABS(num2[-1]);
                        ttt = num2-1;
                        num2 -= size2;
                        size2 = (size2-1)/2;
                        ti = size2;
                        while ( ti > 1 && ttt[-1] == 0 ) { ttt--; ti--; }
                        if ( ti == 1 && ttt[-1] == 1 ) {
                            stor1 = t[FUNHEAD+1];
                            if ( stor1 < 0 ) stor1 = -stor1;
                            num1 = &stor1;
                            size1 = 1;
                            goto gcdcalc;
                        }
                        else pcom[ncom++] = t;
                    }
                    else if ( t[FUNHEAD] > 0 &&
                    t[FUNHEAD]-1-ARGHEAD == ABS(t[t[FUNHEAD]+FUNHEAD-1]) ) {
                        num1 = t + FUNHEAD + t[FUNHEAD];
                        size1 = ABS(num1[-1]);
                        ttt = num1-1;
                        num1 -= size1;
                        size1 = (size1-1)/2;
                        ti = size1;
                        while ( ti > 1 && ttt[-1] == 0 ) { ttt--; ti--; }
                        if ( ti == 1 && ttt[-1] == 1 ) {
                         if ( t[1]-FUNHEAD == t[FUNHEAD]+2 && t[t[1]-2] == -SNUMBER
                         && t[t[1]-1] != 0 ) { /* Long,Short */
                            stor2 = t[t[1]-1];
                            if ( stor2 < 0 ) stor2 = -stor2;
                            num2 = &stor2;
                            size2 = 1;
                            goto gcdcalc;
                         }
                         else if ( t[1]-FUNHEAD == t[FUNHEAD]+t[FUNHEAD+t[FUNHEAD]]
                         && ABS(t[t[1]-1]) == t[FUNHEAD+t[FUNHEAD]] - ARGHEAD-1 ) {
                          num2 = t + t[1];
                          size2 = ABS(num2[-1]);
                          ttt = num2-1;
                          num2 -= size2;
                          size2 = (size2-1)/2;
                          ti = size2;
                          while ( ti > 1 && ttt[-1] == 0 ) { ttt--; ti--; }
                          if ( ti == 1 && ttt[-1] == 1 ) {
gcdcalc:                    if ( GcdLong(BHEAD (UWORD *)num1,size1,(UWORD *)num2,size2
                                ,(UWORD *)lnum,&nnum) ) goto FromNorm;
                            goto MulIn;
                          }
                          else pcom[ncom++] = t;
                         }
                         else pcom[ncom++] = t;
                        }
                        else pcom[ncom++] = t;
                    }
                    else pcom[ncom++] = t;
                }
                else pcom[ncom++] = t;
                }
#else
                {
                    WORD *gcd = AT.WorkPointer;
                    if ( ( gcd = EvaluateGcd(BHEAD t) ) == 0 ) goto FromNorm;
```

```
                    if ( *gcd == 4 && gcd[1] == 1 && gcd[2] == 1 && gcd[4] == 0 ) {
                        AT.WorkPointer = gcd;
                    }
                    else if ( gcd[*gcd] == 0 ) {
                        WORD *t1, iii, change, *num, *den, numsize, densize;
                        if ( gcd[*gcd-1] < *gcd-1 ) {
                            t1 = gcd+1;
                            for ( iii = 2; iii < t1[1]; iii += 2 ) {
                                change = ExtraSymbol(t1[iii],t1[iii+1],nsym,ppsym,&ncoef);
                                nsym += change;
                                ppsym += change « 1;
                            }
                        }
                        t1 = gcd + *gcd;
                        iii = t1[-1]; num = t1-iii; numsize = (iii-1)/2;
                        den = num + numsize; densize = numsize;
                        while ( numsize > 1 && num[numsize-1] == 0 ) numsize--;
                        while ( densize > 1 && den[densize-1] == 0 ) densize--;
                        if ( numsize > 1 || num[0] != 1 ) {
                            ncoef = REDLENG(ncoef);
                            if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)num,numsize) ) goto FromNorm;
                            ncoef = INCLENG(ncoef);
                        }
                        if ( densize > 1 || den[0] != 1 ) {
                            ncoef = REDLENG(ncoef);
                            if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)den,densize) ) goto FromNorm;
                            ncoef = INCLENG(ncoef);
                        }
                        AT.WorkPointer = gcd;
                    }
                    else {  /* a whole expression */
/*
                        Action: Put the expression in a compiler buffer.
                        Insert a SUBEXPRESSION subterm
                        Set the return value of the routine such that in
                        Generator the term gets sent again to TestSub.

                        1: put in C (ebufnum)
                        2: after that the WorkSpace is free again.
                        3: insert the SUBEXPRESSION
                        4: copy the top part of the term down
*/
                        LONG size = AT.WorkPointer - gcd;

                        ss = AddRHS(AT.ebufnum,1);
                        while ( (ss + size + 10) > C->Top ) ss = DoubleCbuffer(AT.ebufnum,ss,13);
                        tt = gcd;
                        NCOPY(ss,tt,size);
                        C->rhs[C->numrhs+1] = ss;
                        C->Pointer = ss;

                        t[0] = SUBEXPRESSION;
                        t[1] = SUBEXPSIZE;
                        t[2] = C->numrhs;
                        t[3] = 1;
                        t[4] = AT.ebufnum;
                        t += 5;
                        tt = term + *term;
                        while ( r < tt ) *t++ = *r++;
                        *term = t - term;

                        regval = 1;
                        goto RegEnd;
                    }
                }
#endif
#else
                MesPrint(" Unexpected call to EvaluateGCD");
                Terminate(-1);
#endif
                break;
            case MINFUNCTION:
            case MAXFUNCTION:
                if ( t[1] == FUNHEAD ) break;
                {
                    WORD *ttt = t + FUNHEAD;
                    WORD *tttstop = t + t[1];
                    WORD tterm[4], iii;
                    while ( ttt < tttstop ) {
                        if ( *ttt > 0 ) {
                            if ( ttt[ARGHEAD]-1 > ABS(ttt[*ttt-1]) ) goto nospec;
                            ttt += *ttt;
                        }
                        else {
                            if ( *ttt != -SNUMBER ) goto nospec;
```

```
                            ttt += 2;
                        }
                    }
/*
                    Function has only numerical arguments
                    Pick up the first argument.
*/
                    ttt = t + FUNHEAD;
                    if ( *ttt > 0 ) {
loadnew1:
                        for ( iii = 0; iii < ttt[ARGHEAD]; iii++ )
                            n_llnum[iii] = ttt[ARGHEAD+iii];
                        ttt += *ttt;
                    }
                    else {
loadnew2:
                        if ( ttt[1] == 0 ) {
                            n_llnum[0] = n_llnum[1] = n_llnum[2] = n_llnum[3] = 0;
                        }
                        else {
                            n_llnum[0] = 4;
                            if ( ttt[1] > 0 ) { n_llnum[1] = ttt[1]; n_llnum[3] = 3; }
                            else { n_llnum[1] = -ttt[1]; n_llnum[3] = -3; }
                            n_llnum[2] = 1;
                        }
                        ttt += 2;
                    }
/*
                    Now loop over the other arguments
*/
                    while ( ttt < tttstop ) {
                        if ( *ttt > 0 ) {
                            if ( n_llnum[0] == 0 ) {
                                if ( ( *t == MINFUNCTION && ttt[*ttt-1] < 0 )
                                || ( *t == MAXFUNCTION && ttt[*ttt-1] > 0 ) )
                                    goto loadnew1;
                            }
                            else {
                                ttt += ARGHEAD;
                                iii = CompCoef(n_llnum,ttt);
                                if ( ( iii > 0 && *t == MINFUNCTION )
                                || ( iii < 0 && *t == MAXFUNCTION ) ) {
                                    for ( iii = 0; iii < ttt[0]; iii++ )
                                        n_llnum[iii] = ttt[iii];
                                }
                            }
                            ttt += *ttt;
                        }
                        else {
                            if ( n_llnum[0] == 0 ) {
                                if ( ( *t == MINFUNCTION && ttt[1] < 0 )
                                || ( *t == MAXFUNCTION && ttt[1] > 0 ) )
                                    goto loadnew2;
                            }
                            else if ( ttt[1] == 0 ) {
                                if ( ( *t == MINFUNCTION && n_llnum[*n_llnum-1] > 0 )
                                || ( *t == MAXFUNCTION && n_llnum[*n_llnum-1] < 0 ) ) {
                                    n_llnum[0] = 0;
                                }
                            }
                            else {
                                tterm[0] = 4; tterm[2] = 1;
                                if ( ttt[1] < 0 ) { tterm[1] = -ttt[1]; tterm[3] = -3; }
                                else { tterm[1] = ttt[1]; tterm[3] = 3; }
                                iii = CompCoef(n_llnum,tterm);
                                if ( ( iii > 0 && *t == MINFUNCTION )
                                || ( iii < 0 && *t == MAXFUNCTION ) ) {
                                    for ( iii = 0; iii < 4; iii++ )
                                        n_llnum[iii] = tterm[iii];
                                }
                            }
                            ttt += 2;
                        }
                    }
                    if ( n_llnum[0] == 0 ) goto NormZero;
                    ncoef = REDLENG(ncoef);
                    nnum = REDLENG(n_llnum[*n_llnum-1]);
                    if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)lnum,nnum,
                    (UWORD *)n_coef,&ncoef) ) goto FromNorm;
                    ncoef = INCLENG(ncoef);
                }
                break;
            case INVERSEFACTORIAL:
                if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] >= 0 ) {
                    if ( Factorial(BHEAD t[FUNHEAD+1],(UWORD *)lnum,&nnum) )
                        goto FromNorm;
                    ncoef = REDLENG(ncoef);
```

```
                    if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) goto FromNorm;
                    ncoef = INCLENG(ncoef);
                }
                else {
nospec:             pcom[ncom++] = t;
                }
                break;
            case MAXPOWEROF:
                if ( ( t[FUNHEAD] == -SYMBOL )
                     && ( t[FUNHEAD+1] > 0 ) && ( t[1] == FUNHEAD+2 ) ) {
                    *((UWORD *)lnum) = symbols[t[FUNHEAD+1]].maxpower;
                    nnum = 1;
                    goto MulIn;
                }
                else { pcom[ncom++] = t; }
                break;
            case MINPOWEROF:
                if ( ( t[FUNHEAD] == -SYMBOL )
                     && ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+2 ) ) {
                    *((UWORD *)lnum) = symbols[t[FUNHEAD+1]].minpower;
                    nnum = 1;
                    goto MulIn;
                }
                else { pcom[ncom++] = t; }
                break;
            case PRIMENUMBER :
                if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER
                     && t[FUNHEAD+1] > 0 ) {
                    UWORD xp = (UWORD)(NextPrime(BHEAD t[FUNHEAD+1]));
                    ncoef = REDLENG(ncoef);
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,&xp,1) ) goto FromNorm;
                    ncoef = INCLENG(ncoef);
                }
                else goto defaultcase;
                break;
            case LNUMBER :
                ncoef = REDLENG(ncoef);
                if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(t+3),t[2]) ) goto FromNorm;
                ncoef = INCLENG(ncoef);
                break;
            case SNUMBER :
                if ( t[2] < 0 ) {
                    t[2] = -t[2];
                    if ( t[3] & 1 ) ncoef = -ncoef;
                }
                else if ( t[2] == 0 ) {
                    if ( t[3] < 0 ) goto NormInf;
                    goto NormZero;
                }
                lnum[0] = t[2];
                nnum = 1;
                if ( t[3] && RaisPow(BHEAD (UWORD *)lnum,&nnum,(UWORD)(ABS(t[3]))) ) goto FromNorm;
                ncoef = REDLENG(ncoef);
                if ( t[3] < 0 ) {
                    if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
                        goto FromNorm;
                }
                else if ( t[3] > 0 ) {
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
                        goto FromNorm;
                }
                ncoef = INCLENG(ncoef);
                break;
            case GAMMA :
            case GAMMAI :
            case GAMMAFIVE :
            case GAMMASIX :
            case GAMMASEVEN :
                if ( t[1] == FUNHEAD ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Gamma matrix without spin line encountered.");
                    MUNLOCK(ErrorMessageLock);
                    goto NormMin;
                }
                pnco[nnco++] = t;
                t += FUNHEAD+1;
                goto ScanCont;
            case LEVICIVITA :
                peps[neps++] = t;
                if ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) {
                    t[2] &= ~DIRTYFLAG;
                    t[2] |= DIRTYSYMFLAG;
                }
                t += FUNHEAD;
ScanCont:       while ( t < r ) {
                    if ( *t >= AM.OffsetIndex &&
                        ( *t >= AM.DumInd || ( *t < AM.WilInd &&
```

```
                        indices[*t-AM.OffsetIndex].dimension ) ) )
                        pcon[ncon++] = t;
                    t++;
                }
                break;
            case EXPONENT :
                {
                    WORD *rr;
                    k = 1;
                    rr = t + FUNHEAD;
                    if ( *rr == ARGHEAD || ( *rr == -SNUMBER && rr[1] == 0 ) )
                        k = 0;
                    if ( *rr == -SNUMBER && rr[1] == 1 ) break;
                    if ( *rr <= -FUNCTION ) k = *rr;
                    NEXTARG(rr)
                    if ( *rr == ARGHEAD || ( *rr == -SNUMBER && rr[1] == 0 ) ) {
                        if ( k == 0 ) goto NormZZ;
                        break;
                    }
                    if ( *rr == -SNUMBER && rr[1] > 0 && rr[1] < MAXPOWER && k < 0 ) {
                        k = -k;
                        if ( functions[k-FUNCTION].commute ) {
                            for ( i = 0; i < rr[1]; i++ ) pnco[nnco++] = rr-1;
                        }
                        else {
                            for ( i = 0; i < rr[1]; i++ ) pcom[ncom++] = rr-1;
                        }
                        break;
                    }
                    if ( k == 0 ) goto NormZero;
                    if ( t[FUNHEAD] == -SYMBOL && *rr == -SNUMBER && t[1] == FUNHEAD+4 ) {
                        if ( rr[1] < MAXPOWER ) {
                            t[FUNHEAD+2] = t[FUNHEAD+1]; t += FUNHEAD+2;
                            from = m;
                            goto NextSymbol;
                        }
                    }
/*
                    if ( ( t[FUNHEAD] > 0 && t[FUNHEAD+1] != 0 )
                    || ( *rr > 0 && rr[1] != 0 ) ) {}
                    else
*/
                    t[2] &= ~DIRTYSYMFLAG;

                    pnco[nnco++] = t;
                }
                break;
            case DENOMINATOR :
                t[2] &= ~DIRTYSYMFLAG;
                pden[nden++] = t;
                pnco[nnco++] = t;
                break;
            case INDEX :
                t += 2;
                do {
                    if ( *t == 0 ) goto NormZero;
                    if ( *t > 0 && *t < AM.OffsetIndex ) {
                        lnum[0] = *t++;
                        nnum = 1;
                        ncoef = REDLENG(ncoef);
                        if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
                            goto FromNorm;
                        ncoef = INCLENG(ncoef);
                    }
                    else if ( *t == NOINDEX ) t++;
                    else pind[nind++] = *t++;
                } while ( t < r );
                break;
            case SUBEXPRESSION :
                if ( t[3] == 0 ) break;
            case EXPRESSION :
                goto RegEnd;
            case ROOTFUNCTION :
/*
                Tries to take the n-th root inside the rationals
                If this is not possible, it clears all flags and
                hence tries no more.
                Notation:
                root_(power(=integer),(rational)number)
*/
                { WORD nc;
                if ( t[2] == 0 ) goto defaultcase;
                if ( t[FUNHEAD] != -SNUMBER || t[FUNHEAD+1] < 0 ) goto defaultcase;
                if ( t[FUNHEAD+2] == -SNUMBER ) {
                    if ( t[FUNHEAD+1] == 0 && t[FUNHEAD+3] == 0 ) goto NormZZ;
                    if ( t[FUNHEAD+1] == 0 ) break;
                    if ( t[FUNHEAD+3] < 0 ) {
```

```
                        AT.WorkPointer[0] = -t[FUNHEAD+3];
                        nc = -1;
                    }
                    else {
                        AT.WorkPointer[0] = t[FUNHEAD+3];
                        nc = 1;
                    }
                    AT.WorkPointer[1] = 1;
                }
                else if ( t[FUNHEAD+2] == t[1]-FUNHEAD-2
                && t[FUNHEAD+2] == t[FUNHEAD+2+ARGHEAD]+ARGHEAD
                && ABS(t[t[1]-1]) == t[FUNHEAD+2+ARGHEAD] - 1 ) {
                    WORD *r1, *r2;
                    if ( t[FUNHEAD+1] == 0 ) break;
                    i = t[t[1]-1]; r1 = t + FUNHEAD+ARGHEAD+3;
                    nc = REDLENG(i);
                    i = ABS(i) - 1;
                    r2 = AT.WorkPointer;
                    while ( --i >= 0 ) *r2++ = *r1++;
                }
                else goto defaultcase;
                if ( TakeRatRoot((UWORD *)AT.WorkPointer,&nc,t[FUNHEAD+1]) ) {
                    t[2] = 0;
                    goto defaultcase;
                }
                ncoef = REDLENG(ncoef);
                if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)AT.WorkPointer,nc) )
                        goto FromNorm;
                if ( nc < 0 ) nc = -nc;
                if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(AT.WorkPointer+nc),nc) )
                        goto FromNorm;
                ncoef = INCLENG(ncoef);
                }
                break;
            case RANDOMFUNCTION :
                {
                WORD nnc, nc, nca, nr;
                UWORD xx;
/*
                Needs one positive integer argument.
                returns (wranf()%argument)+1.
                We may call wranf several times to paste UWORDS together
                when we need long numbers.
                We make little errors when taking the % operator
                (not 100% uniform). We correct for that by redoing the
                calculation in the (unlikely) case that we are in leftover area
*/
                if ( t[1] == FUNHEAD ) goto defaultcase;
                if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER &&
                    t[FUNHEAD+1] > 0 ) {
                    if ( t[FUNHEAD+1] == 1 ) break;
redoshort:
                    ((UWORD *)AT.WorkPointer)[0] = wranf(BHEAD0);
                    ((UWORD *)AT.WorkPointer)[1] = wranf(BHEAD0);
                    nr = 2;
                    if ( ((UWORD *)AT.WorkPointer)[1] == 0 ) {
                        nr = 1;
                        if ( ((UWORD *)AT.WorkPointer)[0] == 0 ) {
                            nr = 0;
                        }
                    }
                    xx = (UWORD)(t[FUNHEAD+1]);
                    if ( nr ) {
                    DivLong((UWORD *)AT.WorkPointer,nr
                            ,&xx,1
                            ,((UWORD *)AT.WorkPointer)+4,&nnc
                            ,((UWORD *)AT.WorkPointer)+2,&nc);
                    ((UWORD *)AT.WorkPointer)[4] = 0;
                    ((UWORD *)AT.WorkPointer)[5] = 0;
                    ((UWORD *)AT.WorkPointer)[6] = 1;
                    DivLong((UWORD *)AT.WorkPointer+4,3
                            ,&xx,1
                            ,((UWORD *)AT.WorkPointer)+9,&nnc
                            ,((UWORD *)AT.WorkPointer)+7,&nca);
                    AddLong((UWORD *)AT.WorkPointer+4,3
                            ,((UWORD *)AT.WorkPointer)+7,-nca
                            ,((UWORD *)AT.WorkPointer)+9,&nnc);
                    if ( BigLong((UWORD *)AT.WorkPointer,nr
                            ,((UWORD *)AT.WorkPointer)+9,nnc) >= 0 ) goto redoshort;
                    }
                    else nc = 0;
                    if ( nc == 0 ) {
                        AT.WorkPointer[2] = (WORD)xx;
                        nc = 1;
                    }
                    ncoef = REDLENG(ncoef);
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,((UWORD *)(AT.WorkPointer))+2,nc) )
```

```
                        goto FromNorm;
                    ncoef = INCLENG(ncoef);
                }
                else if ( t[FUNHEAD] > 0 && t[1] == t[FUNHEAD]+FUNHEAD
                && ABS(t[t[1]-1]) == t[FUNHEAD]-1-ARGHEAD && t[t[1]-1] > 0 ) {
                    WORD nna, nnb, nni, nnb2, nnb2a;
                    UWORD *nnt;
                    nna = t[t[1]-1];
                    nnb2 = nna-1;
                    nnb = nnb2/2;
                    nnt = (UWORD *)(t+t[1]-1-nnb);  /* start of denominator */
                    if ( *nnt != 1 ) goto defaultcase;
                    for ( nni = 1; nni < nnb; nni++ ) {
                        if ( nnt[nni] != 0 ) goto defaultcase;
                    }
                    nnt = (UWORD *)(t + FUNHEAD + ARGHEAD + 1);

                    for ( nni = 0; nni < nnb2; nni++ ) {
                        ((UWORD *)AT.WorkPointer)[nni] = wranf(BHEAD0);
                    }
                    nnb2a = nnb2;
                    while ( nnb2a > 0 && ((UWORD *)AT.WorkPointer)[nnb2a-1] == 0 ) nnb2a--;
                    if ( nnb2a > 0 ) {
                    DivLong((UWORD *)AT.WorkPointer,nnb2a
                            ,nnt,nnb
                            ,((UWORD *)AT.WorkPointer)+2*nnb2,&nnc
                            ,((UWORD *)AT.WorkPointer)+nnb2,&nc);
                    for ( nni = 0; nni < nnb2; nni++ ) {
                        ((UWORD *)AT.WorkPointer)[nni+2*nnb2] = 0;
                    }
                    ((UWORD *)AT.WorkPointer)[3*nnb2] = 1;
                    DivLong((UWORD *)AT.WorkPointer+2*nnb2,nnb2+1
                            ,nnt,nnb
                            ,((UWORD *)AT.WorkPointer)+4*nnb2+1,&nnc
                            ,((UWORD *)AT.WorkPointer)+3*nnb2+1,&nca);
                    AddLong((UWORD *)AT.WorkPointer+2*nnb2,nnb2+1
                            ,((UWORD *)AT.WorkPointer)+3*nnb2+1,-nca
                            ,((UWORD *)AT.WorkPointer)+4*nnb2+1,&nnc);
                    if ( BigLong((UWORD *)AT.WorkPointer,nnb2a
                            ,((UWORD *)AT.WorkPointer)+4*nnb2+1,nnc) >= 0 ) goto redoshort;
                    }
                    else nc = 0;
                    if ( nc == 0 ) {
                        for ( nni = 0; nni < nnb; nni++ ) {
                            ((UWORD *)AT.WorkPointer)[nnb2+nni] = nnt[nni];
                        }
                        nc = nnb;
                    }
                    ncoef = REDLENG(ncoef);
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,((UWORD *)(AT.WorkPointer))+nnb2,nc) )
                            goto FromNorm;
                    ncoef = INCLENG(ncoef);
                }
                else goto defaultcase;
                }
                break;
            case RANPERM :
                if ( *t == RANPERM && t[1] > FUNHEAD && t[FUNHEAD] <= -FUNCTION ) {
                    WORD **pwork;
                    WORD *mm, *ww, *ow = AT.WorkPointer;
                    WORD *Array, *targ, *argstop, narg = 0, itot;
                    int ie;
                    argstop = t+t[1];
                    targ = t+FUNHEAD+1;
                    while ( targ < argstop ) {
                        narg++; NEXTARG(targ);
                    }
                    WantAddPointers(narg);
                    pwork = AT.pWorkSpace + AT.pWorkPointer;
                    targ = t+FUNHEAD+1; narg = 0;
                    while ( targ < argstop ) {
                        pwork[narg++] = targ;
                        NEXTARG(targ);
                    }
/*
                    Make a random permutation of the numbers 0,...,narg-1
                    The following code works also for narg == 0 and narg == 1
*/
                    ow = AT.WorkPointer;
                    Array = AT.WorkPointer;
                    AT.WorkPointer += narg;
                    for ( i = 0; i < narg; i++ ) Array[i] = i;
                    for ( i = 2; i <= narg; i++ ) {
                        itot = (WORD)(iranf(BHEAD i));
                        for ( j = 0; j < itot; j++ ) CYCLE1(WORD,Array,i)
                    }
                    mm = AT.WorkPointer;
```

```
                    *mm++ = -t[FUNHEAD];
                    *mm++ = t[1] - 1;
                    for ( ie = 2; ie < FUNHEAD; ie++ ) *mm++ = t[ie];
                    for ( i = 0; i < narg; i++ ) {
                        ww = pwork[Array[i]];
                        CopyArg(mm,ww);
                    }
                    mm = AT.WorkPointer; t++; ww = t;
                    i = mm[1]; NCOPY(ww,mm,i)
                    AT.WorkPointer = ow;
                    goto TryAgain;
                }
                pnco[nnco++] = t;
                break;
            case PUTFIRST : /* First argument should be a function, second a number */
                if ( ( t[2] & DIRTYFLAG ) != 0 && t[FUNHEAD] <= -FUNCTION
                && t[FUNHEAD+1] == -SNUMBER && t[FUNHEAD+2] > 0 ) {
                    WORD *rr = t+t[1], *mm = t+FUNHEAD+3, *tt, *tt1, *tt2, num = 0;
/*
                    now count the arguments. If not enough: no action.
*/
                    while ( mm < rr ) { num++; NEXTARG(mm); }
                    if ( num < t[FUNHEAD+2] ) { pnco[nnco++] = t; break; }
                    *t = -t[FUNHEAD]; mm = t+FUNHEAD+3;
                    i = t[FUNHEAD+2];
                    while ( --i > 0 ) { NEXTARG(mm); }
                    tt = TermMalloc("Select_"); /* Move selected out of the way */
                    tt1 = tt;
                    if ( *mm > 0 ) {
                        for ( i = 0; i < *mm; i++ ) *tt1++ = mm[i];
                    }
                    else if ( *mm <= -FUNCTION ) { *tt1++ = *mm; }
                    else { *tt1++ = mm[0]; *tt1++ = mm[1]; }
                    tt2 = t+FUNHEAD+3;
                    while ( tt2 < mm ) *tt1++ = *tt2++;
                    i = tt1-tt; tt1 = tt; tt2 = t+FUNHEAD;
                    NCOPY(tt2,tt1,i);
                    TermFree(tt,"Select_");
                    NEXTARG(mm);
                    while ( mm < rr ) *tt2++ = *mm++;
                    t[1] = tt2 - t;
                    rr = term + *term;
                    while ( mm < rr ) *tt2++ = *mm++;
                    *term = tt2-term;
                    goto Restart;
                }
                else pnco[nnco++] = t;
                break;
            case INTFUNCTION :
/*
                Can be resolved if the first argument is a number
                and the second argument either doesn't exist or has
                the value +1, 0, -1
                +1 : rounding up
                0  : rounding towards zero
                -1 : rounding down (same as no argument)
*/
                if ( t[1] <= FUNHEAD ) break;
                {
                    WORD *rr, den, num;
                    to = t + FUNHEAD;
                    if ( *to > 0 ) {
                        if ( *to == ARGHEAD ) goto NormZero;
                        rr = to + *to;
                        i = rr[-1];
                        j = ABS(i);
                        if ( to[ARGHEAD] != j+1 ) goto NoInteg;
                        if ( rr >= r ) k = -1;
                        else if ( *rr == ARGHEAD ) { k = 0; rr += ARGHEAD; }
                        else if ( *rr == -SNUMBER ) { k = rr[1]; rr += 2; }
                        else goto NoInteg;
                        if ( rr != r ) goto NoInteg;
                        if ( k > 1 || k < -1 ) goto NoInteg;
                        to += ARGHEAD+1;
                        j = (j-1) » 1;
                        i = ( i < 0 ) ? -j: j;
                        UnPack((UWORD *)to,i,&den,&num);
/*
                        Potentially the use of NoScrat2 is unsafe.
                        It makes the routine not reentrant, but because it is
                        used only locally and because we only call the
                        low level routines DivLong and AddLong which never
                        make calls involving Normalize, things are OK after all
*/
                        if ( AN.NoScrat2 == 0 ) {
                            AN.NoScrat2 = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"Normalize");
                        }
```

```
                        if ( DivLong((UWORD *)to,num,(UWORD *)(to+j),den
                        ,(UWORD *)AT.WorkPointer,&num,AN.NoScrat2,&den) ) goto FromNorm;
                        if ( k < 0 && den < 0 ) {
                            *AN.NoScrat2 = 1;
                            den = -1;
                            if ( AddLong((UWORD *)AT.WorkPointer,num
                            ,AN.NoScrat2,den,(UWORD *)AT.WorkPointer,&num) )
                                goto FromNorm;
                        }
                        else if ( k > 0 && den > 0 ) {
                            *AN.NoScrat2 = 1;
                            den = 1;
                            if ( AddLong((UWORD *)AT.WorkPointer,num,
                            AN.NoScrat2,den,(UWORD *)AT.WorkPointer,&num) )
                                goto FromNorm;
                        }

                    }
                    else if ( *to == -SNUMBER ) {   /* No rounding needed */
                        if ( to[1] < 0 ) { *AT.WorkPointer = -to[1]; num = -1; }
                        else if ( to[1] == 0 ) goto NormZero;
                        else { *AT.WorkPointer = to[1]; num = 1; }
                    }
                    else goto NoInteg;
                    if ( num == 0 ) goto NormZero;
                    ncoef = REDLENG(ncoef);
                    if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)AT.WorkPointer,num) )
                        goto FromNorm;
                    ncoef = INCLENG(ncoef);
                    break;
                }
NoInteg:;
/*
                Fall through if it cannot be resolved
*/
            default :
defaultcase:;
                if ( *t < FUNCTION ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Illegal code in Norm");
#ifdef DEBUGON
                    {
                        UBYTE OutBuf[140];
                        AO.OutFill = AO.OutputLine = OutBuf;
                        t = term;
                        AO.OutSkip = 3;
                        FiniLine();
                        i = *t;
                        while ( --i >= 0 ) {
                            TalToLine((UWORD)(*t++));
                            TokenToLine((UBYTE *)"  ");
                        }
                        AO.OutSkip = 0;
                        FiniLine();
                    }
#endif
                    MUNLOCK(ErrorMessageLock);
                    goto NormMin;
                }
                if ( *t == REPLACEMENT ) {
                    if ( AR.Eside != LHSIDE ) ReplaceVeto--;
                    pcom[ncom++] = t;
                    break;
                }
/*
                if ( *t == AM.termfunnum && t[1] == FUNHEAD+2
                && t[FUNHEAD] == -DOLLAREXPRESSION ) termflag++;
*/
                if ( *t == DUMMYFUN || *t == DUMMYTEN ) {}
                else {
                    if ( *t < (FUNCTION + WILDOFFSET) ) {
                        if ( ( ( functions[*t-FUNCTION].maxnumargs > 0 )
                        || ( functions[*t-FUNCTION].minnumargs > 0 ) )
                         && ( ( t[2] & DIRTYFLAG ) != 0 ) ) {
/*
                            Number of arguments is bounded. And we have not checked.
*/
                            WORD *ta = t + FUNHEAD, *tb = t + t[1];
                            int numarg = 0;
                            while ( ta < tb ) { numarg++; NEXTARG(ta) }
                            if ( ( functions[*t-FUNCTION].maxnumargs > 0 )
                            && ( numarg >= functions[*t-FUNCTION].maxnumargs ) )
                                goto NormZero;
                            if ( ( functions[*t-FUNCTION].minnumargs > 0 )
                            && ( numarg < functions[*t-FUNCTION].minnumargs ) )
                                goto NormZero;
                        }
```

```
doflags:
                        if ( ( ( t[2] & DIRTYFLAG ) != 0 ) && ( functions[*t-FUNCTION].tabl == 0 ) ) {
                            t[2] &= ~DIRTYFLAG;
                            t[2] |= DIRTYSYMFLAG;
                        }
                        if ( functions[*t-FUNCTION].commute ) { pnco[nnco++] = t; }
                        else { pcom[ncom++] = t; }
                    }
                    else {
                        if ( ( ( t[2] & DIRTYFLAG ) != 0 ) && ( functions[*t-FUNCTION-WILDOFFSET].tabl == 0 ) ) {
                            t[2] &= ~DIRTYFLAG;
                            t[2] |= DIRTYSYMFLAG;
                        }
                        if ( functions[*t-FUNCTION-WILDOFFSET].commute ) {
                            pnco[nnco++] = t;
                        }
                        else { pcom[ncom++] = t; }
                    }
                }

                /* Now hunt for contractible indices */

                if ( ( *t < (FUNCTION + WILDOFFSET)
                 && functions[*t-FUNCTION].spec >= TENSORFUNCTION ) || (
                 *t >= (FUNCTION + WILDOFFSET)
                 && functions[*t-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION ) ) {
                    if ( *t >= GAMMA && *t <= GAMMASEVEN ) t++;
                    t += FUNHEAD;
                    while ( t < r ) {
                        if ( *t >= AM.OffsetIndex && ( *t >= AM.DumInd
                        || ( *t < AM.WilInd && indices[*t-AM.OffsetIndex].dimension ) ) ) {
                            pcon[ncon++] = t;
                        }
                        else if ( *t == FUNNYWILD ) { t++; }
                        t++;
                    }
                }
                else {
                    t += FUNHEAD;
                    while ( t < r ) {
                        if ( *t > 0 ) {
/*
                            Here we should worry about a recursion
                            A problem is the possibility of a construct
                            like  f(mu+nu)
*/
                            t += *t;
                        }
                        else if ( *t <= -FUNCTION ) t++;
                        else if ( *t == -INDEX ) {
                            if ( t[1] >= AM.OffsetIndex &&
                                ( t[1] >= AM.DumInd || ( t[1] < AM.WilInd
                                && indices[t[1]-AM.OffsetIndex].dimension ) ) )
                                pcon[ncon++] = t+1;
                            t += 2;
                        }
                        else if ( *t == -SYMBOL ) {
                            if ( t[1] >= MAXPOWER && t[1] < 2*MAXPOWER ) {
                                *t = -SNUMBER;
                                t[1] -= MAXPOWER;
                            }
                            else if ( t[1] < -MAXPOWER && t[1] > -2*MAXPOWER  ) {
                                *t = -SNUMBER;
                                t[1] += MAXPOWER;
                            }
                            else t += 2;
                        }
                        else t += 2;
                    }
                }
                break;
        }
        t = r;
TryAgain:;
    } while ( t < m );
    if ( ANsc ) {
        AN.cTerm = ANsc;
        r = t = ANsr; m = ANsm;
        ANsc = ANsm = ANsr = 0;
        goto conscan;
    }
/*
    #] First scan :
    #[ Easy denominators :

    Easy denominators are denominators that can be replaced by
```

```
    negative powers of individual subterms. This may add to all
    our sublists.

*/
    if ( nden ) {
        for ( k = 0, i = 0; i < nden; i++ ) {
            t = pden[i];
            if ( ( t[2] & DIRTYFLAG ) == 0 ) continue;
            r = t + t[1]; m = t + FUNHEAD;
            if ( m >= r ) {
                for ( j = i+1; j < nden; j++ ) pden[j-1] = pden[j];
                nden--;
                for ( j = 0; j < nnco; j++ ) if ( pnco[j] == t ) break;
                for ( j++; j < nnco; j++ ) pnco[j-1] = pnco[j];
                nnco--;
                i--;
            }
            else {
                NEXTARG(m);
                if ( m >= r ) continue;
/*
                We have more than one argument. Split the function.
*/
                if ( k == 0 ) {
                    k = 1; to = termout; from = term;
                }
                while ( from < t ) *to++ = *from++;
                m = t + FUNHEAD;
                while ( m < r ) {
                    stop = to;
                    *to++ = DENOMINATOR;
                    for ( j = 1; j < FUNHEAD; j++ ) *to++ = 0;
                    if ( *m < -FUNCTION ) *to++ = *m++;
                    else if ( *m < 0 ) { *to++ = *m++; *to++ = *m++; }
                    else {
                        j = *m; while ( --j >= 0 ) *to++ = *m++;
                    }
                    stop[1] = WORDDIF(to,stop);
                }
                from = r;
                if ( i == nden - 1 ) {
                    stop = term + *term;
                    while ( from < stop ) *to++ = *from++;
                    i = *termout = WORDDIF(to,termout);
                    to = term; from = termout;
                    while ( --i >= 0 ) *to++ = *from++;
                    goto Restart;
                }
            }
        }
        for ( i = 0; i < nden; i++ ) {
            t = pden[i];
            if ( ( t[2] & DIRTYFLAG ) == 0 ) continue;
            t[2] = 0;
            if ( t[FUNHEAD] == -SYMBOL ) {
                WORD change;
                t += FUNHEAD+1;
                change = ExtraSymbol(*t,-1,nsym,ppsym,&ncoef);
                nsym += change;
                ppsym += change « 1;
                goto DropDen;
            }
            else if ( t[FUNHEAD] == -SNUMBER ) {
                t += FUNHEAD+1;
                if ( *t == 0 ) goto NormInf;
                if ( *t < 0 ) { *AT.WorkPointer = -*t; j = -1; }
                else { *AT.WorkPointer = *t; j = 1; }
                ncoef = REDLENG(ncoef);
                if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)AT.WorkPointer,j) )
                    goto FromNorm;
                ncoef = INCLENG(ncoef);
                goto DropDen;
            }
            else if ( t[FUNHEAD] == ARGHEAD ) goto NormInf;
            else if ( t[FUNHEAD] > 0 && t[FUNHEAD+ARGHEAD] ==
            t[FUNHEAD]-ARGHEAD ) {
                /* Only one term */
                r = t + t[1] - 1;
                t += FUNHEAD + ARGHEAD + 1;
                j = *r;
                m = r - ABS(*r) + 1;
                if ( j != 3 || ( ( *m != 1 ) || ( m[1] != 1 ) ) ) {
                    ncoef = REDLENG(ncoef);
                    if ( DivRat(BHEAD (UWORD *)n_coef,ncoef,(UWORD *)m,REDLENG(j),(UWORD
*)n_coef,&ncoef) ) goto FromNorm;
                    ncoef = INCLENG(ncoef);
                    j = ABS(j) - 3;
```

```
                    t[-FUNHEAD-ARGHEAD] -= j;
                    t[-ARGHEAD-1] -= j;
                    t[-1] -= j;
                    m[0] = m[1] = 1;
                    m[2] = 3;
                }
                while ( t < m ) {
                    r = t + t[1];
                    if ( *t == SYMBOL || *t == DOTPRODUCT ) {
                        k = t[1];
                        pden[i][1] -= k;
                        pden[i][FUNHEAD] -= k;
                        pden[i][FUNHEAD+ARGHEAD] -= k;
                        m -= k;
                        stop = m + 3;
                        tt = to = t;
                        from = r;
                        if ( *t == SYMBOL ) {
                            t += 2;
                            while ( t < r ) {
                                WORD change;
                                change = ExtraSymbol(*t,-t[1],nsym,ppsym,&ncoef);
                                nsym += change;
                                ppsym += change « 1;
                                t += 2;
                            }
                        }
                        else {
                            t += 2;
                            while ( t < r ) {
                                *ppdot++ = *t++;
                                *ppdot++ = *t++;
                                *ppdot++ = -*t++;
                                ndot++;
                            }
                        }
                        while ( to < stop ) *to++ = *from++;
                        r = tt;
                    }
                    t = r;
                }
                if ( pden[i][1] == 4+FUNHEAD+ARGHEAD ) {
DropDen:
                    for ( j = 0; j < nnco; j++ ) {
                        if ( pden[i] == pnco[j] ) {
                            --nnco;
                            while ( j < nnco ) {
                                pnco[j] = pnco[j+1];
                                j++;
                            }
                            break;
                        }
                    }
                    pden[i--] = pden[--nden];
                }
            }
        }
    }
/*
    #] Easy denominators :
    #[ Index Contractions :
*/
    if ( ndel ) {
        t = pdel;
        for ( i = 0; i < ndel; i += 2 ) {
            if ( t[0] == t[1] ) {
                if ( t[0] == EMPTYINDEX ) {}
                else if ( *t < AM.OffsetIndex ) {
                    k = AC.FixIndices[*t];
                    if ( k < 0 ) { j = -1; k = -k; }
                    else if ( k > 0 ) j = 1;
                    else goto NormZero;
                    goto WithFix;
                }
                else if ( *t >= AM.DumInd ) {
                    k = AC.lDefDim;
                    if ( k ) goto docontract;
                }
                else if ( *t >= AM.WilInd ) {
                    k = indices[*t-AM.OffsetIndex-WILDOFFSET].dimension;
                    if ( k ) goto docontract;
                }
                else if ( ( k = indices[*t-AM.OffsetIndex].dimension ) != 0 ) {
docontract:
                    if ( k > 0 ) {
                        j = 1;
WithFix:                shortnum = k;
```

```
                        ncoef = REDLENG(ncoef);
                        if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(&shortnum),j) )
                            goto FromNorm;
                        ncoef = INCLENG(ncoef);
                    }
                    else {
                        WORD change;
                        change = ExtraSymbol((WORD)(-k),(WORD)1,nsym,ppsym,&ncoef);
                        nsym += change;
                        ppsym += change « 1;
                    }
                    t[1] = pdel[ndel-1];
                    t[0] = pdel[ndel-2];
HaveCon:
                    ndel -= 2;
                    i -= 2;
                }
            }
            else {
                if ( *t < AM.OffsetIndex && t[1] < AM.OffsetIndex ) goto NormZero;
                j = *t - AM.OffsetIndex;
                if ( j >= 0 && ( ( *t >= AM.DumInd && AC.lDefDim )
                || ( *t < AM.WilInd && indices[j].dimension ) ) ) {
                    for ( j = i + 2, m = pdel+j; j < ndel; j += 2, m += 2 ) {
                        if ( *t == *m ) {
                            *t = m[1];
                            *m++ = pdel[ndel-2];
                            *m = pdel[ndel-1];
                            goto HaveCon;
                        }
                        else if ( *t == m[1] ) {
                            *t = *m;
                            *m++ = pdel[ndel-2];
                            *m   = pdel[ndel-1];
                            goto HaveCon;
                        }
                    }
                }
                j = t[1]-AM.OffsetIndex;
                if ( j >= 0 && ( ( t[1] >= AM.DumInd && AC.lDefDim )
                || ( t[1] < AM.WilInd && indices[j].dimension ) ) ) {
                    for ( j = i + 2, m = pdel+j; j < ndel; j += 2, m += 2 ) {
                        if ( t[1] == *m ) {
                            t[1] = m[1];
                            *m++ = pdel[ndel-2];
                            *m   = pdel[ndel-1];
                            goto HaveCon;
                        }
                        else if ( t[1] == m[1] ) {
                            t[1] = *m;
                            *m++ = pdel[ndel-2];
                            *m   = pdel[ndel-1];
                            goto HaveCon;
                        }
                    }
                }
                t += 2;
            }
        }
        if ( ndel > 0 ) {
            if ( nvec ) {
                t = pdel;
                for ( i = 0; i < ndel; i++ ) {
                    if ( *t >= AM.OffsetIndex && ( ( *t >= AM.DumInd && AC.lDefDim ) ||
                    ( *t < AM.WilInd && indices[*t-AM.OffsetIndex].dimension ) ) ) {
                        r = pvec + 1;
                        for ( j = 1; j < nvec; j += 2 ) {
                            if ( *r == *t ) {
                                if ( i & 1 ) {
                                    *r = t[-1];
                                    *t-- = pdel[--ndel];
                                    i -= 2;
                                }
                                else {
                                    *r = t[1];
                                    t[1] = pdel[--ndel];
                                    i--;
                                }
                                *t-- = pdel[--ndel];
                                break;
                            }
                            r += 2;
                        }
                    }
                    t++;
                }
            }
```

```
            if ( ndel > 0 && ncon ) {
                t = pdel;
                for ( i = 0; i < ndel; i++ ) {
                    if ( *t >= AM.OffsetIndex && ( ( *t >= AM.DumInd && AC.lDefDim ) ||
                    ( *t < AM.WilInd && indices[*t-AM.OffsetIndex].dimension ) ) ) {
                        for ( j = 0; j < ncon; j++ ) {
                            if ( *pcon[j] == *t ) {
                                if ( i & 1 ) {
                                    *pcon[j] = t[-1];
                                    *t-- = pdel[--ndel];
                                    i -= 2;
                                }
                                else {
                                    *pcon[j] = t[1];
                                    t[1] = pdel[--ndel];
                                    i--;
                                }
                                *t-- = pdel[--ndel];
                                didcontr++;
                                r = pcon[j];
                                for ( j = 0; j < nnco; j++ ) {
                                    m = pnco[j];
                                    if ( r > m && r < m+m[1] ) {
                                        m[2] |= DIRTYSYMFLAG;
                                        break;
                                    }
                                }
                                for ( j = 0; j < ncom; j++ ) {
                                    m = pcom[j];
                                    if ( r > m && r < m+m[1] ) {
                                        m[2] |= DIRTYSYMFLAG;
                                        break;
                                    }
                                }
                                for ( j = 0; j < neps; j++ ) {
                                    m = peps[j];
                                    if ( r > m && r < m+m[1] ) {
                                        m[2] |= DIRTYSYMFLAG;
                                        break;
                                    }
                                }
                                break;
                            }
                        }
                    }
                    t++;
                }
            }
        }
    }
    if ( nvec ) {
        t = pvec + 1;
        for ( i = 3; i < nvec; i += 2 ) {
            k = *t - AM.OffsetIndex;
            if ( k >= 0 && ( ( *t > AM.DumInd && AC.lDefDim )
            || ( *t < AM.WilInd && indices[k].dimension ) ) ) {
                r = t + 2;
                for ( j = i; j < nvec; j += 2 ) {
                    if ( *r == *t ) {   /* Another dotproduct */
                        *ppdot++ = t[-1];
                        *ppdot++ = r[-1];
                        *ppdot++ = 1;
                        ndot++;
                        *r-- = pvec[--nvec];
                        *r   = pvec[--nvec];
                        *t-- = pvec[--nvec];
                        *t-- = pvec[--nvec];
                        i -= 2;
                        break;
                    }
                    r += 2;
                }
            }
            t += 2;
        }
        if ( nvec > 0 && ncon ) {
            t = pvec + 1;
            for ( i = 1; i < nvec; i += 2 ) {
                k = *t - AM.OffsetIndex;
                if ( k >= 0 && ( ( *t >= AM.DumInd && AC.lDefDim )
                || ( *t < AM.WilInd && indices[k].dimension ) ) ) {
                    for ( j = 0; j < ncon; j++ ) {
                        if ( *pcon[j] == *t ) {
                            *pcon[j] = t[-1];
                            *t-- = pvec[--nvec];
                            *t-- = pvec[--nvec];
                            r = pcon[j];
```

```
                            pcon[j] = pcon[--ncon];
                            i -= 2;
                            for ( j = 0; j < nnco; j++ ) {
                                m = pnco[j];
                                if ( r > m && r < m+m[1] ) {
                                    m[2] |= DIRTYSYMFLAG;
                                    break;
                                }
                            }
                            for ( j = 0; j < ncom; j++ ) {
                                m = pcom[j];
                                if ( r > m && r < m+m[1] ) {
                                    m[2] |= DIRTYSYMFLAG;
                                    break;
                                }
                            }
                            for ( j = 0; j < neps; j++ ) {
                                m = peps[j];
                                if ( r > m && r < m+m[1] ) {
                                    m[2] |= DIRTYSYMFLAG;
                                    break;
                                }
                            }
                            break;
                        }
                    }
                }
                t += 2;
            }
        }
    }
/*
    #] Index Contractions :
    #[ NonCommuting Functions :
*/
    m = fillsetexp;
    if ( nnco ) {
        for ( i = 0; i < nnco; i++ ) {
            t = pnco[i];
            if ( ( *t >= (FUNCTION+WILDOFFSET)
            && functions[*t-FUNCTION-WILDOFFSET].spec == 0 )
            || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
            && functions[*t-FUNCTION].spec == 0 ) ) {
                DoRevert(t,m);
                if ( didcontr ) {
                    r = t + FUNHEAD;
                    t += t[1];
                    while ( r < t ) {
                        if ( *r == -INDEX && r[1] >= 0 && r[1] < AM.OffsetIndex ) {
                            *r = -SNUMBER;
                            didcontr--;
                            pnco[i][2] |= DIRTYSYMFLAG;
                        }
                        NEXTARG(r)
                    }
                }
            }
        }

        /* First should come the code for function properties. */

        /* First we test for symmetric properties and the DIRTYSYMFLAG */

        for ( i = 0; i < nnco; i++ ) {
            t = pnco[i];
            if ( *t > 0 && ( t[2] & DIRTYSYMFLAG ) && *t != DOLLAREXPRESSION ) {
                l = 0; /* to make the compiler happy */
                if ( ( *t >= (FUNCTION+WILDOFFSET)
                && ( l = functions[*t-FUNCTION-WILDOFFSET].symmetric ) > 0 )
                || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
                && ( l = functions[*t-FUNCTION].symmetric ) > 0 ) ) {
                    if ( *t >= (FUNCTION+WILDOFFSET) ) {
                        *t -= WILDOFFSET;
                        j = FullSymmetrize(BHEAD t,l);
                        *t += WILDOFFSET;
                    }
                    else j = FullSymmetrize(BHEAD t,l);
                    if ( (l & ~REVERSEORDER) == ANTISYMMETRIC ) {
                        if ( ( j & 2 ) != 0 ) goto NormZero;
                        if ( ( j & 1 ) != 0 ) ncoef = -ncoef;
                    }
                }
                else t[2] &= ~DIRTYSYMFLAG;
            }
        }

        /* Non commuting functions are then tested for commutation
```

```
            rules. If needed their order is exchanged. */

        k = nnco - 1;
        for ( i = 0; i < k; i++ ) {
            j = i;
            while ( Commute(pnco[j],pnco[j+1]) ) {
                t = pnco[j]; pnco[j] = pnco[j+1]; pnco[j+1] = t;
                l = j-1;
                while ( l >= 0 && Commute(pnco[l],pnco[l+1]) ) {
                    t = pnco[l]; pnco[l] = pnco[l+1]; pnco[l+1] = t;
                    l--;
                }
                if ( ++j >= k ) break;
            }
        }

        /* Finally they are written to output. gamma matrices
           are bundled if possible */

        for ( i = 0; i < nnco; i++ ) {
            t = pnco[i];
            if ( *t == IDFUNCTION ) AN.idfunctionflag = 1;
            if ( *t >= GAMMA && *t <= GAMMASEVEN ) {
                WORD gtype;
                to = m;
                *m++ = GAMMA;
                m++;
                FILLFUN(m)
                *m++ = stype = t[FUNHEAD];      /* type of string */
                j = 0;
                nnum = 0;
                do {
                    r = t + t[1];
                    if ( *t == GAMMAFIVE ) {
                        gtype = GAMMA5; t += FUNHEAD; goto onegammamatrix; }
                    else if ( *t == GAMMASIX ) {
                        gtype = GAMMA6; t += FUNHEAD; goto onegammamatrix; }
                    else if ( *t == GAMMASEVEN ) {
                        gtype = GAMMA7; t += FUNHEAD; goto onegammamatrix; }
                    t += FUNHEAD+1;
                    while ( t < r ) {
                        gtype = *t;
onegammamatrix:
                        if ( gtype == GAMMA5 ) {
                            if ( j == GAMMA1 ) j = GAMMA5;
                            else if ( j == GAMMA5 ) j = GAMMA1;
                            else if ( j == GAMMA7 ) ncoef = -ncoef;
                            if ( nnum & 1 ) ncoef = -ncoef;
                        }
                        else if ( gtype == GAMMA6 || gtype == GAMMA7 ) {
                            if ( nnum & 1 ) {
                                if ( gtype == GAMMA6 ) gtype = GAMMA7;
                                else                   gtype = GAMMA6;
                            }
                            if ( j == GAMMA1 ) j = gtype;
                            else if ( j == GAMMA5 ) {
                                j = gtype;
                                if ( j == GAMMA7 ) ncoef = -ncoef;
                            }
                            else if ( j != gtype ) goto NormZero;
                            else {
                                shortnum = 2;
                                ncoef = REDLENG(ncoef);
                                if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(&shortnum),1) ) goto FromNorm;
                                ncoef = INCLENG(ncoef);
                            }
                        }
                        else {
                            *m++ = gtype; nnum++;
                        }
                        t++;
                    }

                } while ( ( ++i < nnco ) && ( *(t = pnco[i]) >= GAMMA
                && *t <= GAMMASEVEN ) && ( t[FUNHEAD] == stype ) );
                i--;
                if ( j ) {
                    k = WORDDIF(m,to) - FUNHEAD-1;
                    r = m;
                    from = m++;
                    while ( --k >= 0 ) *from-- = *--r;
                    *from = j;
                }
                to[1] = WORDDIF(m,to);
            }
            else if ( *t < 0 ) {
```

```
                *m++ = -*t; *m++ = FUNHEAD; *m++ = 0;
                FILLFUN3(m)
            }
            else {
                if ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG
                        && *t != REPLACEMENT && *t != DOLLAREXPRESSION
                        && TestFunFlag(BHEAD t) ) ReplaceVeto = 1;
                k = t[1];
                NCOPY(m,t,k);
            }
        }

    }
/*
    #] NonCommuting Functions :
    #[ Commuting Functions :
*/
    if ( ncom ) {
        for ( i = 0; i < ncom; i++ ) {
            t = pcom[i];
            if ( ( *t >= (FUNCTION+WILDOFFSET)
            && functions[*t-FUNCTION-WILDOFFSET].spec == 0 )
            || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
            && functions[*t-FUNCTION].spec == 0 ) ) {
                DoRevert(t,m);
                if ( didcontr ) {
                    r = t + FUNHEAD;
                    t += t[1];
                    while ( r < t ) {
                        if ( *r == -INDEX && r[1] >= 0 && r[1] < AM.OffsetIndex ) {
                            *r = -SNUMBER;
                            didcontr--;
                            pcom[i][2] |= DIRTYSYMFLAG;
                        }
                        NEXTARG(r)
                    }
                }
            }
        }

        /* Now we test for symmetric properties and the DIRTYSYMFLAG */

        for ( i = 0; i < ncom; i++ ) {
            t = pcom[i];
            if ( *t > 0 && ( t[2] & DIRTYSYMFLAG ) ) {
                l = 0; /* to make the compiler happy */
                if ( ( *t >= (FUNCTION+WILDOFFSET)
                && ( l = functions[*t-FUNCTION-WILDOFFSET].symmetric ) > 0 )
                || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
                && ( l = functions[*t-FUNCTION].symmetric ) > 0 ) ) {
                    if ( *t >= (FUNCTION+WILDOFFSET) ) {
                        *t -= WILDOFFSET;
                        j = FullSymmetrize(BHEAD t,l);
                        *t += WILDOFFSET;
                    }
                    else j = FullSymmetrize(BHEAD t,l);
                    if ( (l & ~REVERSEORDER) == ANTISYMMETRIC ) {
                        if ( ( j & 2 ) != 0 ) goto NormZero;
                        if ( ( j & 1 ) != 0 ) ncoef = -ncoef;
                    }
                }
                else t[2] &= ~DIRTYSYMFLAG;
            }
        }
/*
        Sort the functions
        From a purists point of view this can be improved.
        There arel slow and fast arguments and no conversions are
        taken into account here.
*/
        for ( i = 1; i < ncom; i++ ) {
            for ( j = i; j > 0; j-- ) {
                WORD jj,kk;
                jj = j-1;
                t = pcom[jj];
                r = pcom[j];
                if ( *t < 0 ) {
                    if ( *r < 0 ) { if ( *t >= *r ) goto NextI; }
                    else { if ( -*t <= *r ) goto NextI; }
                    goto jexch;
                }
                else if ( *r < 0 ) {
                    if ( *t < -*r ) goto NextI;
                    goto jexch;
                }
                else if ( *t != *r ) {
                    if ( *t < *r ) goto NextI;
```

```
jexch:              t = pcom[j]; pcom[j] = pcom[jj]; pcom[jj] = t;
                    continue;
                }
                if ( AC.properorderflag ) {
                    if ( ( *t >= (FUNCTION+WILDOFFSET)
                    && functions[*t-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION )
                    || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
                    && functions[*t-FUNCTION].spec >= TENSORFUNCTION ) ) {}
                    else {
                        WORD *s1, *s2, *ss1, *ss2;
                        s1 = t+FUNHEAD; s2 = r+FUNHEAD;
                        ss1 = t + t[1]; ss2 = r + r[1];
                        while ( s1 < ss1 && s2 < ss2 ) {
                            k = CompArg(s1,s2);
                            if ( k > 0 ) goto jexch;
                            if ( k < 0 ) goto NextI;
                            NEXTARG(s1)
                            NEXTARG(s2)
                        }
                        if ( s1 < ss1 ) goto jexch;
                        goto NextI;
                    }
                    k = t[1] - FUNHEAD;
                    kk = r[1] - FUNHEAD;
                    t += FUNHEAD;
                    r += FUNHEAD;
                    while ( k > 0 && kk > 0 ) {
                        if ( *t < *r ) goto NextI;
                        else if ( *t++ > *r++ ) goto jexch;
                        k--; kk--;
                    }
                    if ( k > 0 ) goto jexch;
                    goto NextI;
                }
                else
                {
                    k = t[1] - FUNHEAD;
                    kk = r[1] - FUNHEAD;
                    t += FUNHEAD;
                    r += FUNHEAD;
                    while ( k > 0 && kk > 0 ) {
                        if ( *t < *r ) goto NextI;
                        else if ( *t++ > *r++ ) goto jexch;
                        k--; kk--;
                    }
                    if ( k > 0 ) goto jexch;
                    goto NextI;
                }
            }
NextI:;
        }
        for ( i = 0; i < ncom; i++ ) {
            t = pcom[i];
            if ( *t == THETA || *t == THETA2 ) {
                if ( ( k = DoTheta(BHEAD t) ) == 0 ) goto NormZero;
                else if ( k < 0 ) {
                    k = t[1];
                    NCOPY(m,t,k);
                }
            }
            else if ( *t == DELTA2 || *t == DELTAP ) {
                if ( ( k = DoDelta(t) ) == 0 ) goto NormZero;
                else if ( k < 0 ) {
                    k = t[1];
                    NCOPY(m,t,k);
                }
            }
            else if ( *t == AR.PolyFunInv && AR.PolyFunType == 2 ) {
/*
                If there are two arguments, exchange them, change the
                name of the function and go to dealing with PolyRatFun.
*/
                WORD *mm, *tt = t, numt = 0;
                tt += FUNHEAD;
                while ( tt < t+t[1] ) { numt++; NEXTARG(tt) }
                if ( numt == 2 ) {
                    tt = t; mm = m; k = t[1];
                    NCOPY(mm,tt,k)
                    mm = m+FUNHEAD;
                    NEXTARG(mm);
                    tt = t+FUNHEAD;
                    if ( *mm < 0 ) {
                        if ( *mm <= -FUNCTION ) { *tt++ = *mm++; }
                        else { *tt++ = *mm++;  *tt++ = *mm++; }
                    }
                    else {
                        k = *mm; NCOPY(tt,mm,k)
```

```
                    }
                    mm = m+FUNHEAD;
                    if ( *mm < 0 ) {
                        if ( *mm <= -FUNCTION ) { *tt++ = *mm++; }
                        else { *tt++ = *mm++;  *tt++ = *mm++; }
                    }
                    else {
                        k = *mm; NCOPY(tt,mm,k)
                    }
                    *t = AR.PolyFun;
                    t[2] |= MUSTCLEANPRF;
                    goto regularratfun;
                }
            }
            else if ( *t == AR.PolyFun ) {
              if ( AR.PolyFunType == 1 ) { /* Regular PolyFun with one argument */
                if ( t[FUNHEAD+1] == 0 && AR.Eside != LHSIDE &&
                t[1] == FUNHEAD + 2 && t[FUNHEAD] == -SNUMBER ) goto NormZero;
                if ( i > 0 && pcom[i-1][0] == AR.PolyFun ) {
                    if ( AN.PolyNormFlag == 0 ) {
                        AN.PolyNormFlag = 1;
                        AN.PolyFunTodo = 0;
                    }
                }
                k = t[1];
                NCOPY(m,t,k);
              }
              else if ( AR.PolyFunType == 2 ) {
/*
                PolyRatFun.
                    Regular type: Two arguments
                    Power expanded: One argument. Here to be treated as
                        AR.PolyFunType == 1, but with power cutoff.
*/
regularratfun:;
/*
                First check for zeroes.
*/
                if ( t[FUNHEAD+1] == 0 && AR.Eside != LHSIDE &&
                t[1] > FUNHEAD + 2 && t[FUNHEAD] == -SNUMBER ) {
                    u = t + FUNHEAD + 2;
                    if ( *u < 0 ) {
                        if ( *u <= -FUNCTION ) {}
                        else if ( t[1] == FUNHEAD+4 && t[FUNHEAD+2] == -SNUMBER
                            && t[FUNHEAD+3] == 0 ) goto NormPRF;
                        else if ( t[1] == FUNHEAD+4 ) goto NormZero;
                    }
                    else if ( t[1] == *u+FUNHEAD+2 ) goto NormZero;
                }
                else {
                    u = t+FUNHEAD; NEXTARG(u);
                    if ( *u == -SNUMBER && u[1] == 0 ) goto NormInf;
                }
                if ( i > 0 && pcom[i-1][0] == AR.PolyFun ) AN.PolyNormFlag = 1;
                else if ( i < ncom-1 && pcom[i+1][0] == AR.PolyFun ) AN.PolyNormFlag = 1;
                k = t[1];
                if ( AN.PolyNormFlag ) {
                    if ( AR.PolyFunExp == 0 ) {
                        AN.PolyFunTodo = 0;
                        NCOPY(m,t,k);
                    }
                    else if ( AR.PolyFunExp == 1 ) { /* get highest divergence */
                        if ( PolyFunMode == 0 ) {
                            NCOPY(m,t,k);
                            AN.PolyFunTodo = 1;
                        }
                        else {
                            WORD *mmm = m;
                            NCOPY(m,t,k);
                            if ( TreatPolyRatFun(BHEAD mmm) != 0 )
                                    goto FromNorm;
                            m = mmm+mmm[1];
                        }
                    }
                    else {
                        if ( PolyFunMode == 0 ) {
                            NCOPY(m,t,k);
                                AN.PolyFunTodo = 1;
                        }
                        else {
                            WORD *mmm = m;
                            NCOPY(m,t,k);
                            if ( ExpandRat(BHEAD mmm) != 0 )
                                    goto FromNorm;
                            m = mmm+mmm[1];
                        }
                    }
```

```
                }
                else {
                    if ( AR.PolyFunExp == 0 ) {
                        AN.PolyFunTodo = 0;
                        NCOPY(m,t,k);
                    }
                    else if ( AR.PolyFunExp == 1 ) { /* get highest divergence */
                        WORD *mmm = m;
                        NCOPY(m,t,k);
                        if ( TreatPolyRatFun(BHEAD mmm) != 0 )
                                goto FromNorm;
                        m = mmm+mmm[1];
                    }
                    else {
                        WORD *mmm = m;
                        NCOPY(m,t,k);
                        if ( ExpandRat(BHEAD mmm) != 0 )
                                goto FromNorm;
                        m = mmm+mmm[1];
                    }
                }
              }
            }
            else if ( *t > 0 ) {
                if ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG
                    && *t != REPLACEMENT && TestFunFlag(BHEAD t) ) ReplaceVeto = 1;
                k = t[1];
                NCOPY(m,t,k);
            }
            else {
                *m++ = -*t; *m++ = FUNHEAD; *m++ = 0;
                FILLFUN3(m)
            }
        }
    }
/*
    #] Commuting Functions :
    #[ Track Replace_ :
*/
    if ( ReplaceVeto < 0 ) {
/*
        We found one (or more) replace_ functions and all other
        functions are 'clean' (no dirty flag).
        Now we check whether one of these functions can be used.
        Thus far the functions go from fillsetexp to m.
        Somewhere in there there are -ReplaceVeto occurrences of REPLACEMENT.
        Hunt for the first one that fits the bill.
        Note that replace_ is a commuting function.
*/
        WORD *ma = fillsetexp, *mb, *mc;
        while ( ma < m ) {
            mb = ma + ma[1];
            if ( *ma != REPLACEMENT ) {
                ma = mb;
                continue;
            }
            if ( *ma == REPLACEMENT && ReplaceType == -1 ) {
                mc = ma;
                ReplaceType = 0;
                if ( AN.RSsize < 2*ma[1]+SUBEXPSIZE ) {
                    if ( AN.ReplaceScrat ) M_free(AN.ReplaceScrat,"AN.ReplaceScrat");
                    AN.RSsize = 2*ma[1]+SUBEXPSIZE+40;
                    AN.ReplaceScrat = (WORD *)Malloc1((AN.RSsize+1)*sizeof(WORD),"AN.ReplaceScrat");
                }
                ma += FUNHEAD;
                ReplaceSub = AN.ReplaceScrat;
                ReplaceSub += SUBEXPSIZE;
                while ( ma < mb ) {
                    if ( *ma > 0 ) goto NoRep;
                    if ( *ma <= -FUNCTION ) {
                        *ReplaceSub++ = FUNTOFUN;
                        *ReplaceSub++ = 4;
                        *ReplaceSub++ = -*ma++;
                        if ( *ma > -FUNCTION ) goto NoRep;
                        *ReplaceSub++ = -*ma++;
                    }
                    else if ( ma+4 > mb ) goto NoRep;
                    else {
                        if ( *ma == -SYMBOL ) {
                            if ( ma[2] == -SYMBOL && ma+4 <= mb )
                                *ReplaceSub++ = SYMTOSYM;
                            else if ( ma[2] == -SNUMBER && ma+4 <= mb ) {
                                *ReplaceSub++ = SYMTONUM;
                                if ( ReplaceType == 0 ) {
                                    oldtoprhs = C->numrhs;
                                    oldcpointer = C->Pointer - C->Buffer;
                                }
```

```
                                ReplaceType = 1;
                            }
                            else if ( ma[2] == ARGHEAD && ma+2+ARGHEAD <= mb ) {
                                *ReplaceSub++ = SYMTONUM;
                                *ReplaceSub++ = 4;
                                *ReplaceSub++ = ma[1];
                                *ReplaceSub++ = 0;
                                ma += 2+ARGHEAD;
                                continue;
                            }
/*
                            Next is the subexpression. We have to test that
                            it isn't vector-like or index-like
*/
                            else if ( ma[2] > 0 ) {
                                WORD *sstop, *ttstop, n;
                                ss = ma+2;
                                sstop = ss + *ss;
                                ss += ARGHEAD;
                                while ( ss < sstop ) {
                                    tt = ss + *ss;
                                    ttstop = tt - ABS(tt[-1]);
                                    ss++;
                                    while ( ss < ttstop ) {
                                        if ( *ss == INDEX ) goto NoRep;
                                        ss += ss[1];
                                    }
                                    ss = tt;
                                }
                                subtype = SYMTOSUB;
                                if ( ReplaceType == 0 ) {
                                    oldtoprhs = C->numrhs;
                                    oldcpointer = C->Pointer - C->Buffer;
                                }
                                ReplaceType = 1;
                                ss = AddRHS(AT.ebufnum,1);
                                tt = ma+2;
                                n = *tt - ARGHEAD;
                                tt += ARGHEAD;
                                while ( (ss + n + 10) > C->Top ) ss = DoubleCbuffer(AT.ebufnum,ss,14);
                                while ( --n >= 0 ) *ss++ = *tt++;
                                *ss++ = 0;
                                C->rhs[C->numrhs+1] = ss;
                                C->Pointer = ss;
                                *ReplaceSub++ = subtype;
                                *ReplaceSub++ = 4;
                                *ReplaceSub++ = ma[1];
                                *ReplaceSub++ = C->numrhs;
                                ma += 2 + ma[2];
                                continue;
                            }
                            else goto NoRep;
                        }
                        else if ( ( *ma == -VECTOR || *ma == -MINVECTOR ) && ma+4 <= mb ) {
                            if ( ma[2] == -VECTOR ) {
                                if ( *ma == -VECTOR ) *ReplaceSub++ = VECTOVEC;
                                else *ReplaceSub++ = VECTOMIN;
                            }
                            else if ( ma[2] == -MINVECTOR ) {
                                if ( *ma == -VECTOR ) *ReplaceSub++ = VECTOMIN;
                                else *ReplaceSub++ = VECTOVEC;
                            }
/*
                            Next is a vector-like subexpression
                            Search for vector nature first
*/
                            else if ( ma[2] > 0 ) {
                                WORD *sstop, *ttstop, *w, *mm, n, count;
                                WORD *v1, *v2 = 0;
                                if ( *ma == -MINVECTOR ) {
                                    ss = ma+2;
                                    sstop = ss + *ss;
                                    ss += ARGHEAD;
                                    while ( ss < sstop ) {
                                        ss += *ss;
                                        ss[-1] = -ss[-1];
                                    }
                                    *ma = -VECTOR;
                                }
                                ss = ma+2;
                                sstop = ss + *ss;
                                ss += ARGHEAD;
                                while ( ss < sstop ) {
                                    tt = ss + *ss;
                                    ttstop = tt - ABS(tt[-1]);
                                    ss++;
                                    count = 0;
```

```
                                    while ( ss < ttstop ) {
                                        if ( *ss == INDEX ) {
                                            n = ss[1] - 2; ss += 2;
                                            while ( --n >= 0 ) {
                                                if ( *ss < MINSPEC ) count++;
                                                ss++;
                                            }
                                        }
                                        else ss += ss[1];
                                    }
                                    if ( count != 1 ) goto NoRep;
                                    ss = tt;
                                }
                                subtype = VECTOSUB;
                                if ( ReplaceType == 0 ) {
                                    oldtoprhs = C->numrhs;
                                    oldcpointer = C->Pointer - C->Buffer;
                                }
                                ReplaceType = 1;
                                mm = AddRHS(AT.ebufnum,1);
                                *ReplaceSub++ = subtype;
                                *ReplaceSub++ = 4;
                                *ReplaceSub++ = ma[1];
                                *ReplaceSub++ = C->numrhs;
                                w = ma+2;
                                n = *w - ARGHEAD;
                                w += ARGHEAD;
                                while ( (mm + n + 10) > C->Top )
                                    mm = DoubleCbuffer(AT.ebufnum,mm,15);
                                while ( --n >= 0 ) *mm++ = *w++;
                                *mm++ = 0;
                                C->rhs[C->numrhs+1] = mm;
                                C->Pointer = mm;
                                mm = AddRHS(AT.ebufnum,1);
                                w = ma+2;
                                n = *w - ARGHEAD;
                                w += ARGHEAD;
                                while ( (mm + n + 13) > C->Top )
                                    mm = DoubleCbuffer(AT.ebufnum,mm,16);
                                sstop = w + n;
                                while ( w < sstop ) {
                                    tt = w + *w; ttstop = tt - ABS(tt[-1]);
                                    ss = mm; mm++; w++;
                                    while ( w < ttstop ) {      /* Subterms */
                                        if ( *w != INDEX ) {
                                            n = w[1];
                                            NCOPY(mm,w,n);
                                        }
                                        else {
                                            v1 = mm;
                                            *mm++ = *w++;
                                            *mm++ = n = *w++;
                                            n -= 2;
                                            while ( --n >= 0 ) {
                                                if ( *w >= MINSPEC ) *mm++ = *w++;
                                                else v2 = w++;
                                            }
                                            n = WORDDIF(mm,v1);
                                            if ( n != v1[1] ) {
                                                if ( n <= 2 ) mm -= 2;
                                                else v1[1] = n;
                                                *mm++ = VECTOR;
                                                *mm++ = 4;
                                                *mm++ = *v2;
                                                *mm++ = FUNNYVEC;
                                            }
                                        }
                                    }
                                    while ( w < tt ) *mm++ = *w++;
                                    *ss = WORDDIF(mm,ss);
                                }
                                *mm++ = 0;
                                C->rhs[C->numrhs+1] = mm;
                                C->Pointer = mm;
                                if ( mm > C->Top ) {
                                    MLOCK(ErrorMessageLock);
                                    MesPrint("Internal error in Normalize with extra compiler buffer");
                                    MUNLOCK(ErrorMessageLock);
                                    Terminate(-1);
                                }
                                ma += 2 + ma[2];
                                continue;
                            }
                            else goto NoRep;
                        }
                        else if ( *ma == -INDEX ) {
```

```
                            if ( ( ma[2] == -INDEX || ma[2] == -VECTOR )
                            && ma+4 <= mb )
                                *ReplaceSub++ = INDTOIND;
                            else if ( ma[1] >= AM.OffsetIndex ) {
                                if ( ma[2] == -SNUMBER && ma+4 <= mb
                                && ma[3] >= 0 && ma[3] < AM.OffsetIndex )
                                    *ReplaceSub++ = INDTOIND;
                                else if ( ma[2] == ARGHEAD && ma+2+ARGHEAD <= mb ) {
                                    *ReplaceSub++ = INDTOIND;
                                    *ReplaceSub++ = 4;
                                    *ReplaceSub++ = ma[1];
                                    *ReplaceSub++ = 0;
                                    ma += 2+ARGHEAD;
                                    continue;
                                }
                                else goto NoRep;
                            }
                            else goto NoRep;
                        }
                        else goto NoRep;
                        *ReplaceSub++ = 4;
                        *ReplaceSub++ = ma[1];
                        *ReplaceSub++ = ma[3];
                        ma += 4;
                    }

                }
                AN.ReplaceScrat[1] = ReplaceSub-AN.ReplaceScrat;
/*
                Success. This means that we have to remove the replace_
                from the functions. It starts at mc and end at mb.
*/
                while ( mb < m ) *mc++ = *mb++;
                m = mc;
                break;
NoRep:
                if ( ReplaceType > 0 ) {
                    C->numrhs = oldtoprhs;
                    C->Pointer = C->Buffer + oldcpointer;
                }
                ReplaceType = -1;
                if ( ++ReplaceVeto >= 0 ) break;
            }
            ma = mb;
        }
    }
/*
    #] Track Replace_ :
    #[ LeviCivita tensors :
*/
    if ( neps ) {
        to = m;
        for ( i = 0; i < neps; i++ ) {  /* Put the indices in order */
            t = peps[i];
            if ( ( t[2] & DIRTYSYMFLAG ) != DIRTYSYMFLAG ) continue;
            t[2] &= ~DIRTYSYMFLAG;
            if ( AR.Eside == LHSIDE || AR.Eside == LHSIDEX ) {
                        /* Potential problems with FUNNYWILD */
/*
                First make sure all FUNNIES are at the end.
                Then sort separately
*/
                r = t + FUNHEAD;
                m = tt = t + t[1];
                while ( r < m ) {
                    if ( *r != FUNNYWILD ) { r++; continue; }
                    k = r[1]; u = r + 2;
                    while ( u < tt ) {
                        u[-2] = *u;
                        if ( *u != FUNNYWILD ) ncoef = -ncoef;
                        u++;
                    }
                    tt[-2] = FUNNYWILD; tt[-1] = k; m -= 2;
                }
                t += FUNHEAD;
                do {
                    for ( r = t + 1; r < m; r++ ) {
                        if ( *r < *t ) { k = *r; *r = *t; *t = k; ncoef = -ncoef; }
                        else if ( *r == *t ) goto NormZero;
                    }
                    t++;
                } while ( t < m );
                do {
                    for ( r = t + 2; r < tt; r += 2 ) {
                        if ( r[1] < t[1] ) {
                            k = r[1]; r[1] = t[1]; t[1] = k; ncoef = -ncoef; }
                        else if ( r[1] == t[1] ) goto NormZero;
```

```
                    }
                    t += 2;
                } while ( t < tt );
            }
            else {
                m = t + t[1];
                t += FUNHEAD;
                do {
                    for ( r = t + 1; r < m; r++ ) {
                        if ( *r < *t ) { k = *r; *r = *t; *t = k; ncoef = -ncoef; }
                        else if ( *r == *t ) goto NormZero;
                    }
                    t++;
                } while ( t < m );
            }
        }

        /* Sort the tensors */

        for ( i = 0; i < (neps-1); i++ ) {
            t = peps[i];
            for ( j = i+1; j < neps; j++ ) {
                r = peps[j];
                if ( t[1] > r[1] ) {
                    peps[i] = m = r; peps[j] = r = t; t = m;
                }
                else if ( t[1] == r[1] ) {
                    k = t[1] - FUNHEAD;
                    m = t + FUNHEAD;
                    r += FUNHEAD;
                    do {
                        if ( *r < *m ) {
                            m = peps[j]; peps[j] = t; peps[i] = t = m;
                            break;
                        }
                        else if ( *r++ > *m++ ) break;
                    } while ( --k > 0 );
                }
            }
        }
        m = to;
        for ( i = 0; i < neps; i++ ) {
            t = peps[i];
            k = t[1];
            NCOPY(m,t,k);
        }
    }
/*
    #] LeviCivita tensors :
    #[ Delta :
*/
    if ( ndel ) {
        r = t = pdel;
        for ( i = 0; i < ndel; i += 2, r += 2 ) {
            if ( r[1] < r[0] ) { k = *r; *r = r[1]; r[1] = k; }
        }
        for ( i = 2; i < ndel; i += 2, t += 2 ) {
            r = t + 2;
            for ( j = i; j < ndel; j += 2 ) {
                if ( *r > *t ) { r += 2; }
                else if ( *r < *t ) {
                    k = *r; *r++ = *t; *t++ = k;
                    k = *r; *r++ = *t; *t-- = k;
                }
                else {
                    if ( *++r < t[1] ) {
                        k = *r; *r = t[1]; t[1] = k;
                    }
                    r++;
                }
            }
        }
        t = pdel;
        *m++ = DELTA;
        *m++ = ndel + 2;
        i = ndel;
        NCOPY(m,t,i);
    }
/*
    #] Delta :
    #[ Loose Vectors/Indices :
*/
    if ( nind ) {
        t = pind;
        for ( i = 0; i < nind; i++ ) {
            r = t + 1;
            for ( j = i+1; j < nind; j++ ) {
```

```
                if ( *r < *t ) {
                    k = *r; *r = *t; *t = k;
                }
                r++;
            }
            t++;
        }
        t = pind;
        *m++ = INDEX;
        *m++ = nind + 2;
        i = nind;
        NCOPY(m,t,i);
    }
/*
    #] Loose Vectors/Indices :
    #[ Vectors :
*/
    if ( nvec ) {
        t = pvec;
        for ( i = 2; i < nvec; i += 2 ) {
            r = t + 2;
            for ( j = i; j < nvec; j += 2 ) {
                if ( *r == *t ) {
                    if ( *++r < t[1] ) {
                        k = *r; *r = t[1]; t[1] = k;
                    }
                    r++;
                }
                else if ( *r < *t ) {
                    k = *r; *r++ = *t; *t++ = k;
                    k = *r; *r++ = *t; *t-- = k;
                }
                else { r += 2; }
            }
            t += 2;
        }
        t = pvec;
        *m++ = VECTOR;
        *m++ = nvec + 2;
        i = nvec;
        NCOPY(m,t,i);
    }
/*
    #] Vectors :
    #[ Dotproducts :
*/
    if ( ndot ) {
        to = m;
        m = t = pdot;
        i = ndot;
        while ( --i >= 0 ) {
            if ( *t > t[1] ) { j = *t; *t = t[1]; t[1] = j; }
            t += 3;
        }
        t = m;
        ndot *= 3;
        m += ndot;
        while ( t < (m-3) ) {
            r = t + 3;
            do {
                if ( *r == *t ) {
                    if ( *++r == *++t ) {
                        r++;
                        if ( ( *r < MAXPOWER && t[1] < MAXPOWER )
                        || ( *r > -MAXPOWER && t[1] > -MAXPOWER ) ) {
                            t++;
                            *t += *r;
                            if ( *t > MAXPOWER || *t < -MAXPOWER ) {
                                MLOCK(ErrorMessageLock);
                                MesPrint("Exponent of dotproduct out of range: %d",*t);
                                MUNLOCK(ErrorMessageLock);
                                goto NormMin;
                            }
                            ndot -= 3;
                            *r-- = *--m;
                            *r-- = *--m;
                            *r   = *--m;
                            if ( !*t ) {
                                ndot -= 3;
                                *t-- = *--m;
                                *t-- = *--m;
                                *t   = *--m;
                                t -= 3;
                                break;
                            }
                        }
                        else if ( *r < *++t ) {
```

```
                            k = *r; *r++ = *t; *t = k;
                        }
                        else r++;
                        t -= 2;
                    }
                    else if ( *r < *t ) {
                        k = *r; *r++ = *t; *t++ = k;
                        k = *r; *r++ = *t; *t = k;
                        t -= 2;
                    }
                    else { r += 2; t--; }
                }
                else if ( *r < *t ) {
                    k = *r; *r++ = *t; *t++ = k;
                    k = *r; *r++ = *t; *t++ = k;
                    k = *r; *r++ = *t; *t   = k;
                    t -= 2;
                }
                else { r += 3; }
            } while ( r < m );
            t += 3;
        }
        m = to;
        t = pdot;
        if ( ( i = ndot ) > 0 ) {
            *m++ = DOTPRODUCT;
            *m++ = i + 2;
            NCOPY(m,t,i);
        }
    }
/*
    #] Dotproducts :
    #[ Symbols :
*/
    if ( nsym ) {
        nsym «= 1;
        t = psym;
        *m++ = SYMBOL;
        r = m;
        *m++ = ( i = nsym ) + 2;
        if ( i ) { do {
            if ( !*t ) {
                if ( t[1] < (2*MAXPOWER) ) {        /* powers of i */
                    if ( t[1] & 1 ) { *m++ = 0; *m++ = 1; }
                    else *r -= 2;
                    if ( *++t & 2 ) ncoef = -ncoef;
                    t++;
                }
            }
            else if ( *t <= NumSymbols && *t > -2*MAXPOWER ) {  /* Put powers in range */
                if ( ( ( ( t[1] > symbols[*t].maxpower ) && ( symbols[*t].maxpower < MAXPOWER ) ) ||
                        ( ( t[1] < symbols[*t].minpower ) && ( symbols[*t].minpower > -MAXPOWER ) ) )
    &&
                        ( t[1] < 2*MAXPOWER ) && ( t[1] > -2*MAXPOWER ) ) {
                    if ( i <= 2 || t[2] != *t ) goto NormZero;
                }
                if ( AN.ncmod == 1 && ( AC.modmode & ALSOPOWERS ) != 0 ) {
                    if ( AC.cmod[0] == 1 ) t[1] = 0;
                    else if ( t[1] >= 0 ) t[1] = 1 + (t[1]-1)%(AC.cmod[0]-1);
                    else {
                        t[1] = -1 - (-t[1]-1)%(AC.cmod[0]-1);
                        if ( t[1] < 0 ) t[1] += (AC.cmod[0]-1);
                    }
                }
                if ( ( t[1] < (2*MAXPOWER) && t[1] >= MAXPOWER )
                || ( t[1] > -(2*MAXPOWER) && t[1] <= -MAXPOWER ) ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Exponent out of range: %d",t[1]);
                    MUNLOCK(ErrorMessageLock);
                    goto NormMin;
                }
                if ( AT.TrimPower && AR.PolyFunVar == *t && t[1] > AR.PolyFunPow ) {
                    goto NormZero;
                }
                else if ( t[1] ) {
                    *m++ = *t++;
                    *m++ = *t++;
                }
                else { *r -= 2; t += 2; }
            }
            else {
                *m++ = *t++; *m++ = *t++;
            }
        } while ( (i-=2) > 0 ); }
        if ( *r <= 2 ) m = r-1;
    }
/*
```

```
    #] Symbols :
    #[ Do Replace_ :
*/
    stop = (WORD *)(((UBYTE *)(termout)) + AM.MaxTer);
    i = ABS(ncoef);
    if ( ( m + i ) > stop ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Term too complex during normalization");
        MUNLOCK(ErrorMessageLock);
        goto NormMin;
    }
    if ( ReplaceType >= 0 ) {
        t = n_coef;
        i--;
        NCOPY(m,t,i);
        *m++ = ncoef;
        t = termout;
        *t = WORDDIF(m,t);
        if ( ReplaceType == 0 ) {
            AT.WorkPointer = termout+*termout;
            WildFill(BHEAD term,termout,AN.ReplaceScrat);
            termout = term + *term;
        }
        else {
            AT.WorkPointer = r = termout + *termout;
            WildFill(BHEAD r,termout,AN.ReplaceScrat);
            i = *r; m = term;
            NCOPY(m,r,i);
            termout = m;


            r = m = term;
            r += *term; r -= ABS(r[-1]);
            m++;
            while ( m < r ) {
                if ( *m >= FUNCTION && m[1] > FUNHEAD &&
                functions[*m-FUNCTION].spec != TENSORFUNCTION )
                    m[2] |= DIRTYFLAG;
                m += m[1];
            }
        }
/*
        The next 'reset' cannot be done. We still need the expression
        in the buffer. Note though that this may cause a runaway pointer
        if we are not very careful.

        C->numrhs = oldtoprhs;
        C->Pointer = C->Buffer + oldcpointer;
*/
        TermFree(n_llnum,"n_llnum");
        TermFree(n_coef,"NormCoef");
        return(1);
    }
    else {
        t = termout;
        k = WORDDIF(m,t);
        *t = k + i;
        m = term;
        NCOPY(m,t,k);
        i--;
        t = n_coef;
        NCOPY(m,t,i);
        *m++ = ncoef;
    }
/*
    #] Do Replace_ :
    #[ Errors and Finish :
*/
RegEnd:
    if ( termout < term + *term && termout >= term ) AT.WorkPointer = term + *term;
    else AT.WorkPointer = termout;
/*
    if ( termflag ) {   We have to assign the term to $variable(s)
        TermAssign(term);
    }
*/
    TermFree(n_llnum,"n_llnum");
    TermFree(n_coef,"NormCoef");
    return(regval);

NormInf:
    MLOCK(ErrorMessageLock);
    MesPrint("Division by zero during normalization");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);

NormZZ:
```

```
    MLOCK(ErrorMessageLock);
    MesPrint("0^0 during normalization of term");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);

NormPRF:
    MLOCK(ErrorMessageLock);
    MesPrint("0/0 in polyratfun during normalization of term");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);

NormZero:
    *term = 0;
    AT.WorkPointer = termout;
    TermFree(n_llnum,"n_llnum");
    TermFree(n_coef,"NormCoef");
    return(regval);

NormMin:
    TermFree(n_llnum,"n_llnum");
    TermFree(n_coef,"NormCoef");
    return(-1);

FromNorm:
    MLOCK(ErrorMessageLock);
    MesCall("Norm");
    MUNLOCK(ErrorMessageLock);
    TermFree(n_llnum,"n_llnum");
    TermFree(n_coef,"NormCoef");
    return(-1);

/*
    #] Errors and Finish :
*/
}

/*
        #] Normalize :
        #[ ExtraSymbol :
*/

WORD ExtraSymbol(WORD sym, WORD pow, WORD nsym, WORD *ppsym, WORD *ncoef)
{
    WORD *m, i;
    i = nsym;
    m = ppsym - 2;
    while   ( i > 0 ) {
        if ( sym == *m ) {
            m++;
            if  ( pow > 2*MAXPOWER || pow < -2*MAXPOWER
            ||  *m > 2*MAXPOWER || *m < -2*MAXPOWER ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal wildcard power combination.");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            *m += pow;

            if ( ( sym <= NumSymbols && sym > -MAXPOWER )
             && ( symbols[sym].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
                *m %= symbols[sym].maxpower;
                if ( *m < 0 ) *m += symbols[sym].maxpower;
                if ( ( symbols[sym].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
                    if ( ( ( symbols[sym].maxpower & 1 ) == 0 ) &&
                        ( *m >= symbols[sym].maxpower/2 ) ) {
                        *m -= symbols[sym].maxpower/2; *ncoef = -*ncoef;
                    }
                }
            }

            if  ( *m >= 2*MAXPOWER || *m <= -2*MAXPOWER ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Power overflow during normalization");
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
            if ( !*m ) {
                m--;
                while ( i < nsym )
                    { *m = m[2]; m++; *m = m[2]; m++; i++; }
                return(-1);
            }
            return(0);
        }
        else if ( sym < *m ) {
            m -= 2;
            i--;
```

```
        }
        else break;
    }
    m = ppsym;
    while ( i < nsym )
        { m--; m[2] = *m; m--; m[2] = *m; i++; }
    *m++ = sym;
    *m = pow;
    return(1);
}

/*
        #] ExtraSymbol :
        #[ DoTheta :
*/

WORD DoTheta(PHEAD WORD *t)
{
    GETBIDENTITY
    WORD k, *r1, *r2, *tstop, type;
    WORD ia, *ta, *tb, *stopa, *stopb;
    if ( AC.BracketNormalize ) return(-1);
    type = *t;
    k = t[1];
    tstop = t + k;
    t += FUNHEAD;
    if ( k <= FUNHEAD ) return(1);
    r1 = t;
    NEXTARG(r1)
    if ( r1 == tstop ) {
/*
        One argument
*/
        if ( *t == ARGHEAD ) {
            if ( type == THETA ) return(1);
            else return(0);  /* THETA2 */
        }
        if ( *t < 0 ) {
            if ( *t == -SNUMBER ) {
                if ( t[1] < 0 ) return(0);
                else {
                    if ( type == THETA2 && t[1] == 0 ) return(0);
                    else return(1);
                }
            }
            return(-1);
        }
        k = t[*t-1];
        if ( *t == ABS(k)+1+ARGHEAD ) {
            if ( k > 0 ) return(1);
            else return(0);
        }
        return(-1);
    }
/*
    At least two arguments
*/
    r2 = r1;
    NEXTARG(r2)
    if ( r2 < tstop ) return(-1);   /* More than 2 arguments */
/*
    Note now that zero has to be treated specially
    We take the criteria from the symmetrize routine
*/
    if ( *t == -SNUMBER && *r1 == -SNUMBER ) {
        if ( t[1] > r1[1] ) return(0);
        else if ( t[1] < r1[1] ) {
            return(1);
        }
        else if ( type == THETA ) return(1);
        else return(0);   /* THETA2 */
    }
    else if ( t[1] == 0 && *t == -SNUMBER ) {
        if ( *r1 > 0 ) { }
        else if ( *t < *r1 ) return(1);
        else if ( *t > *r1 ) return(0);
    }
    else if ( r1[1] == 0 && *r1 == -SNUMBER ) {
        if ( *t > 0 ) { }
        else if ( *t < *r1 ) return(1);
        else if ( *t > *r1 ) return(0);
    }
    r2 = AT.WorkPointer;
    if ( *t < 0 ) {
        ta = r2;
        ToGeneral(t,ta,0);
        r2 += *r2;
```

```
    }
    else ta = t;
    if ( *r1 < 0 ) {
        tb = r2;
        ToGeneral(r1,tb,0);
    }
    else tb = r1;
    stopa = ta + *ta;
    stopb = tb + *tb;
    ta += ARGHEAD; tb += ARGHEAD;
    while ( ta < stopa ) {
        if ( tb >= stopb ) return(0);
        if ( ( ia = CompareTerms(BHEAD ta,tb,(WORD)1) ) < 0 ) return(0);
        if ( ia > 0 ) return(1);
        ta += *ta;
        tb += *tb;
    }
    if ( type == THETA ) return(1);
    else return(0); /* THETA2 */
}

/*
        #] DoTheta :
        #[ DoDelta :
*/

WORD DoDelta(WORD *t)
{
    WORD k, *r1, *r2, *tstop, isnum, isnum2, type = *t;
    if ( AC.BracketNormalize ) return(-1);
    k = t[1];
    if ( k <= FUNHEAD ) goto argzero;
    if ( k == FUNHEAD+ARGHEAD && t[FUNHEAD] == ARGHEAD ) goto argzero;
    tstop = t + k;
    t += FUNHEAD;
    r1 = t;
    NEXTARG(r1)
    if ( *t < 0 ) {
        k = 1;
        if ( *t == -SNUMBER ) { isnum = 1; k = t[1]; }
        else isnum = 0;
    }
    else {
        k = t[*t-1];
        k = ABS(k);
        if ( k == *t-ARGHEAD-1 ) isnum = 1;
        else isnum = 0;
        k = 1;
    }
    if ( r1 >= tstop ) {        /* Single argument */
        if ( !isnum ) return(-1);
        if ( k == 0 ) goto argzero;
        goto argnonzero;
    }
    r2 = r1;
    NEXTARG(r2)
    if ( r2 < tstop ) return(-1);
    if ( *r1 < 0 ) {
        if ( *r1 == -SNUMBER ) { isnum2 = 1; }
        else isnum2 = 0;
    }
    else {
        k = r1[*r1-1];
        k = ABS(k);
        if ( k == *r1-ARGHEAD-1 ) isnum2 = 1;
        else isnum2 = 0;
    }
    if ( isnum != isnum2 ) return(-1);
    tstop = r1;
    while ( t < tstop && r1 < r2 ) {
        if ( *t != *r1 ) {
            if ( !isnum ) return(-1);
            goto argnonzero;
        }
        t++; r1++;
    }
    if ( t != tstop || r1 != r2 ) {
        if ( !isnum ) return(-1);
        goto argnonzero;
    }
argzero:
    if ( type == DELTA2 ) return(1);
    else return(0);
argnonzero:
    if ( type == DELTA2 ) return(0);
    else return(1);
}
```

```c

/*
        #] DoDelta :
        #[ DoRevert :
*/

void DoRevert(WORD *fun, WORD *tmp)
{
    WORD *t, *r, *m, *to, *tt, *mm, i, j;
    to = fun + fun[1];
    r = fun + FUNHEAD;
    while ( r < to ) {
        if ( *r <= 0 ) {
            if ( *r == -REVERSEFUNCTION ) {
                m = r; mm = m+1;
                while ( mm < to ) *m++ = *mm++;
                to--;
                (fun[1])--;
                fun[2] |= DIRTYSYMFLAG;
            }
            else if ( *r <= -FUNCTION ) r++;
            else {
                if ( *r == -INDEX && r[1] < MINSPEC ) *r = -VECTOR;
                r += 2;
            }
        }
        else {
            if ( ( *r > ARGHEAD )
            && ( r[ARGHEAD+1] == REVERSEFUNCTION )
            && ( *r == (r[ARGHEAD]+ARGHEAD) )
            && ( r[ARGHEAD] == (r[ARGHEAD+2]+4) )
            && ( *(r+*r-3) == 1 )
            && ( *(r+*r-2) == 1 )
            && ( *(r+*r-1) == 3 ) ) {
                mm = r;
                r += ARGHEAD + 1;
                tt = r + r[1];
                r += FUNHEAD;
                m = tmp;
                t = r;
                j = 0;
                while ( t < tt ) {
                    NEXTARG(t)
                    j++;
                }
                while ( --j >= 0 ) {
                    i = j;
                    t = r;
                    while ( --i >= 0 ) {
                        NEXTARG(t)
                    }
                    if ( *t > 0 ) {
                        i = *t;
                        NCOPY(m,t,i);
                    }
                    else if ( *t <= -FUNCTION ) *m++ = *t++;
                    else { *m++ = *t++; *m++ = *t++; }
                }
                i = WORDDIF(m,tmp);
                m = tmp;
                t = mm;
                r = t + *t;
                NCOPY(t,m,i);
                m = r;
                r = t;
                i = WORDDIF(to,m);
                NCOPY(t,m,i);
                fun[1] = WORDDIF(t,fun);
                to = t;
                fun[2] |= DIRTYSYMFLAG;
            }
            else r += *r;
        }
    }
}

/*
        #] DoRevert :
    #] Normalize :
    #[ DetCommu :

    Determines the number of terms in an expression that contain
    noncommuting objects. This can be used to see whether products of
    this expression can be evaluated with binomial coefficients.

    We don't try to be fancy. If a term contains noncommuting objects
    we are not looking whether they can commute with complete other
```

```
    terms.

    If the number gets too large we cut it off.
*/

#define MAXNUMBEROFNONCOMTERMS 2

WORD DetCommu(WORD *terms)
{
    WORD *t, *tnext, *tstop;
    WORD num = 0;
    if ( *terms == 0 ) return(0);
    if ( terms[*terms] == 0 ) return(0);
    t = terms;
    while ( *t ) {
        tnext = t + *t;
        tstop = tnext - ABS(tnext[-1]);
        t++;
        while ( t < tstop ) {
            if ( *t >= FUNCTION ) {
                if ( functions[*t-FUNCTION].commute ) {
                    num++;
                    if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
                    break;
                }
            }
            else if ( *t == SUBEXPRESSION ) {
                if ( cbuf[t[4]].CanCommu[t[2]] ) {
                    num++;
                    if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
                    break;
                }
            }
            else if ( *t == EXPRESSION ) {
                num++;
                if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
                break;
            }
            else if ( *t == DOLLAREXPRESSION ) {
/*
                Technically this is not correct. We have to test first
                whether this is MODLOCAL (in TFORM) and if so, use the
                local version. Anyway, this should be rare to never
                occurring because dollars should be replaced.
*/
                if ( cbuf[AM.dbufnum].CanCommu[t[2]] ) {
                    num++;
                    if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
                    break;
                }
            }
            t += t[1];
        }
        t = tnext;
    }
    return(num);
}

/*
    #] DetCommu :
    #[ DoesCommu :

    Determines the number of noncommuting objects in a term.
    If the number gets too large we cut it off.
*/

WORD DoesCommu(WORD *term)
{
    WORD *tstop;
    WORD num = 0;
    if ( *term == 0 ) return(0);
    tstop = term + *term;
    tstop = tstop - ABS(tstop[-1]);
    term++;
    while ( term < tstop ) {
        if ( ( *term >= FUNCTION ) && ( functions[*term-FUNCTION].commute ) ) {
            num++;
            if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
        }
        term += term[1];
    }
    return(num);
}

/*
    #] DoesCommu :
    #[ PolyNormPoly :
```

```c

        Normalizes a polynomial
*/

#ifdef EVALUATEGCD
WORD *PolyNormPoly (PHEAD WORD *Poly) {

    GETBIDENTITY;
    WORD *buffer = AT.WorkPointer;
    WORD *p;
    if ( NewSort(BHEAD0) ) { Terminate(-1); }
    AR.CompareRoutine = (void *)&CompareSymbols;
    while ( *Poly ) {
        p = Poly + *Poly;
        if ( SymbolNormalize(Poly) < 0 ) return(0);
        if ( StoreTerm(BHEAD Poly) ) {
            AR.CompareRoutine = (void *)&Compare1;
            LowerSortLevel();
            Terminate(-1);
        }
        Poly = p;
    }
    if ( EndSort(BHEAD buffer,1) < 0 ) {
        AR.CompareRoutine = (void *)&Compare1;
        Terminate(-1);
    }
    p = buffer;
    while ( *p ) p += *p;
    AR.CompareRoutine = (void *)&Compare1;
    AT.WorkPointer = p + 1;
    return(buffer);
}
#endif

/*
    #] PolyNormPoly :
    #[ EvaluateGcd :

    Try to evaluate the GCDFUNCTION gcd_.
    This function can have a number of arguments which can be numbers
    and/or polynomials. If there are objects that aren't SYMBOLS or numbers
    it cannot work currently.

    To make this work properly we have to intervene in proces.c
     proces.c:                      if ( Normalize(BHEAD m) ) {
1060 proces.c:                          if ( Normalize(BHEAD r) ) {
1126?proces.c:                          if ( Normalize(BHEAD term) ) {
     proces.c:              if ( Normalize(BHEAD AT.WorkPointer) ) goto PasErr;
2308!proces.c:      if ( ( retnorm = Normalize(BHEAD term) ) != 0 ) {
     proces.c:                  ReNumber(BHEAD term); Normalize(BHEAD term);
     proces.c:              if ( Normalize(BHEAD v) ) Terminate(-1);
     proces.c:      if ( Normalize(BHEAD w) ) { LowerSortLevel(); goto PolyCall; }
     proces.c:      if ( Normalize(BHEAD term) ) goto PolyCall;
*/
#ifdef EVALUATEGCD

WORD *EvaluateGcd(PHEAD WORD *subterm)
{
    GETBIDENTITY
    WORD *oldworkpointer = AT.WorkPointer, *work1, *work2, *work3;
    WORD *t, *tt, *ttt, *t1, *t2, *t3, *t4, *tstop;
    WORD ct, nnum;
    UWORD gcdnum, stor;
    WORD *lnum=n_llnum+1;
    WORD *num1, *num2, *num3, *den1, *den2, *den3;
    WORD sizenum1, sizenum2, sizenum3, sizeden1, sizeden2, sizeden3;
    int i, isnumeric = 0, numarg = 0 /*, sizearg */;
    LONG size;
/*
    Step 1: Look for -SNUMBER or -SYMBOL arguments.
            If encountered, treat everybody with it.
*/
    tt = subterm + subterm[1]; t = subterm + FUNHEAD;

    while ( t < tt ) {
        numarg++;
        if ( *t == -SNUMBER ) {
            if ( t[1] == 0 ) {
gcdzero:;
                MLOCK(ErrorMessageLock);
                MesPrint("Trying to take the GCD involving a zero term.");
                MUNLOCK(ErrorMessageLock);
                return(0);
            }
            gcdnum = ABS(t[1]);
            t1 = subterm + FUNHEAD;
            while ( gcdnum > 1 && t1 < tt ) {
```

```
                if ( *t1 == -SNUMBER ) {
                    stor = ABS(t1[1]);
                    if ( stor == 0 ) goto gcdzero;
                    if ( GcdLong(BHEAD (UWORD *)&stor,1,(UWORD *)&gcdnum,1,
                                    (UWORD *)lnum,&nnum) ) goto FromGCD;
                    gcdnum = lnum[0];
                    t1 += 2;
                    continue;
                }
                else if ( *t1 == -SYMBOL ) goto gcdisone;
                else if ( *t1 < 0 ) goto gcdillegal;
/*
                Now we have to go through all the terms in the argument.
                This includes long numbers.
*/
                ttt = t1 + *t1;
                ct = *ttt; *ttt = 0;
                if ( t1[1] != 0 ) { /* First normalize the argument */
                    t1 = PolyNormPoly(BHEAD t1+ARGHEAD);
                }
                else t1 += ARGHEAD;
                while ( *t1 ) {
                    t1 += *t1;
                    i = ABS(t1[-1]);
                    t2 = t1 - i;
                    i = (i-1)/2;
                    t3 = t2+i-1;
                    while ( t3 > t2 && *t3 == 0 ) { t3--; i--; }
                    if ( GcdLong(BHEAD (UWORD *)t2,(WORD)i,(UWORD *)&gcdnum,1,
                                    (UWORD *)lnum,&nnum) ) {
                        *ttt = ct;
                        goto FromGCD;
                    }
                    gcdnum = lnum[0];
                    if ( gcdnum == 1 ) {
                        *ttt = ct;
                        goto gcdisone;
                    }
                }
                *ttt = ct;
                t1 = ttt;
                AT.WorkPointer = oldworkpointer;
            }
            if ( gcdnum == 1 ) goto gcdisone;
            oldworkpointer[0] = 4;
            oldworkpointer[1] = gcdnum;
            oldworkpointer[2] = 1;
            oldworkpointer[3] = 3;
            oldworkpointer[4] = 0;
            AT.WorkPointer = oldworkpointer + 5;
            return(oldworkpointer);
        }
        else if ( *t == -SYMBOL ) {
            t1 = subterm + FUNHEAD;
            i = t[1];
            while ( t1 < tt ) {
                if ( *t1 == -SNUMBER ) goto gcdisone;
                if ( *t1 == -SYMBOL ) {
                    if ( t1[1] != i ) goto gcdisone;
                    t1 += 2; continue;
                }
                if ( *t1 < 0 ) goto gcdillegal;
                ttt = t1 + *t1;
                ct = *ttt; *ttt = 0;
                if ( t1[1] != 0 ) { /* First normalize the argument */
                    t2 = PolyNormPoly(BHEAD t1+ARGHEAD);
                }
                else t2 = t1 + ARGHEAD;
                while ( *t2 ) {
                    t3 = t2+1;
                    t2 = t2 + *t2;
                    tstop = t2 - ABS(t2[-1]);
                    while ( t3 < tstop ) {
                        if ( *t3 != SYMBOL ) {
                            *ttt = ct;
                            goto gcdillegal;
                        }
                        t4 = t3 + 2;
                        t3 += t3[1];
                        while ( t4 < t3 ) {
                            if ( *t4 == i && t4[1] > 0 ) goto nextterminarg;
                            t4 += 2;
                        }
                    }
                    *ttt = ct;
                    goto gcdisone;
nextterminarg:;
```

```
                }
                *ttt = ct;
                t1 = ttt;
                AT.WorkPointer = oldworkpointer;
            }
            oldworkpointer[0] = 8;
            oldworkpointer[1] = SYMBOL;
            oldworkpointer[2] = 4;
            oldworkpointer[3] = t[1];
            oldworkpointer[4] = 1;
            oldworkpointer[5] = 1;
            oldworkpointer[6] = 1;
            oldworkpointer[7] = 3;
            oldworkpointer[8] = 0;
            AT.WorkPointer = oldworkpointer+9;
            return(oldworkpointer);
        }
        else if ( *t < 0 ) {
gcdillegal:;
            MLOCK(ErrorMessageLock);
            MesPrint("Illegal object in gcd_ function. Object not a number or a symbol.");
            MUNLOCK(ErrorMessageLock);
            goto FromGCD;
        }
        else if ( ABS(t[*t-1]) == *t-ARGHEAD-1 ) isnumeric = numarg;
        else if ( t[1] != 0 ) {
            ttt = t + *t; ct = *ttt; *ttt = 0;
            t = PolyNormPoly(BHEAD t+ARGHEAD);
            *ttt = ct;
            if ( t[*t] == 0 && ABS(t[*t-1]) == *t-ARGHEAD-1 ) isnumeric = numarg;
            AT.WorkPointer = oldworkpointer;
            t = ttt;
        }
        t += *t;
    }
/*
    At this point there are only generic arguments.
    There are however still two cases:
    1: There is an argument that is purely numerical
       In that case we have to take the gcd of all coefficients
    2: All arguments are nontrivial polynomials.
       Here we don't worry so much about the factor. (???)
    We know whether case 1 occurs when isnumeric > 0.
    We can look up numarg to get a good starting value.
*/
    AT.WorkPointer = oldworkpointer;
    if ( isnumeric ) {
        t = subterm + FUNHEAD;
        for ( i = 1; i < isnumeric; i++ ) {
            NEXTARG(t);
        }
        if ( t[1] != 0 ) {  /* First normalize the argument */
            ttt = t + *t; ct = *ttt; *ttt = 0;
            t = PolyNormPoly(BHEAD t+ARGHEAD);
            *ttt = ct;
        }
        t += *t;
        i = (ABS(t[-1])-1)/2;
        den1 = t - 1 - i;
        num1 = den1 - i;
        sizenum1 = sizeden1 = i;
        while ( sizenum1 > 1 && num1[sizenum1-1] == 0 ) sizenum1--;
        while ( sizeden1 > 1 && den1[sizeden1-1] == 0 ) sizeden1--;
        work1 = AT.WorkPointer+1; work2 = work1+sizenum1;
        for ( i = 0; i < sizenum1; i++ ) work1[i] = num1[i];
        for ( i = 0; i < sizeden1; i++ ) work2[i] = den1[i];
        num1 = work1; den1 = work2;
        AT.WorkPointer = work2 = work2 + sizeden1;
        t = subterm + FUNHEAD;
        while ( t < tt ) {
            ttt = t + *t; ct = *ttt; *ttt = 0;
            if ( t[1] != 0 ) {
                t = PolyNormPoly(BHEAD t+ARGHEAD);
            }
            else t += ARGHEAD;
            while ( *t ) {
                t += *t;
                i = (ABS(t[-1])-1)/2;
                den2 = t - 1 - i;
                num2 = den2 - i;
                sizenum2 = sizeden2 = i;
                while ( sizenum2 > 1 && num2[sizenum2-1] == 0 ) sizenum2--;
                while ( sizeden2 > 1 && den2[sizeden2-1] == 0 ) sizeden2--;
                num3 = AT.WorkPointer;
                if ( GcdLong(BHEAD (UWORD *)num2,sizenum2,(UWORD *)num1,sizenum1,
                                   (UWORD *)num3,&sizenum3) ) goto FromGCD;
                sizenum1 = sizenum3;
```

```
                for ( i = 0; i < sizenum1; i++ ) num1[i] = num3[i];
                den3 = AT.WorkPointer;
                if ( GcdLong(BHEAD (UWORD *)den2,sizeden2,(UWORD *)den1,sizeden1,
                                    (UWORD *)den3,&sizeden3) ) goto FromGCD;
                sizeden1 = sizeden3;
                for ( i = 0; i < sizeden1; i++ ) den1[i] = den3[i];
                if ( sizenum1 == 1 && num1[0] == 1 && sizeden1 == 1 && den1[1] == 1 )
                    goto gcdisone;
            }
            *ttt = ct;
            t = ttt;
            AT.WorkPointer = work2;
        }
        AT.WorkPointer = oldworkpointer;
/*
        Now copy the GCD to the 'output'
*/
        if ( sizenum1 > sizeden1 ) {
            while ( sizenum1 > sizeden1 ) den1[sizeden1++] = 0;
        }
        else if ( sizenum1 < sizeden1 ) {
            while ( sizenum1 < sizeden1 ) num1[sizenum1++] = 0;
        }
        t = oldworkpointer;
        i = 2*sizenum1+1;
        *t++ = i+1;
        if ( num1 != t ) { NCOPY(t,num1,sizenum1); }
        else t += sizenum1;
        if ( den1 != t ) { NCOPY(t,den1,sizeden1); }
        else t += sizeden1;
        *t++ = i;
        *t++ = 0;
        AT.WorkPointer = t;
        return(oldworkpointer);
    }
/*
    Now the real stuff with only polynomials.
    Pick up the shortest term to start.
    We are a bit brutish about this.
*/
    t = subterm + FUNHEAD;
    AT.WorkPointer += AM.MaxTer/sizeof(WORD);
    work2 = AT.WorkPointer;
/*
    sizearg = subterm[1];
*/
    i = 0; work3 = 0;
    while ( t < tt ) {
        i++;
        work1 = AT.WorkPointer;
        ttt = t + *t; ct = *ttt; *ttt = 0;
        t = PolyNormPoly(BHEAD t+ARGHEAD);
        if ( *work1 < AT.WorkPointer-work1 ) {
/*
            sizearg = AT.WorkPointer-work1;
*/
            numarg = i;
            work3 = work1;
        }
        *ttt = ct; t = ttt;
    }
    *AT.WorkPointer++ = 0;
/*
    We have properly normalized arguments and the shortest is indicated in work3
*/
    work1 = work3;
    while ( *work2 ) {
        if ( work2 != work3 ) {
            work1 = PolyGCD2(BHEAD work1,work2);
        }
        while ( *work2 ) work2 += *work2;
        work2++;
    }
    work2 = work1;
    while ( *work2 ) work2 += *work2;
    size = work2 - work1 + 1;
    t = oldworkpointer;
    NCOPY(t,work1,size);
    AT.WorkPointer = t;
    return(oldworkpointer);

gcdisone:;
    oldworkpointer[0] = 4;
    oldworkpointer[1] = 1;
    oldworkpointer[2] = 1;
    oldworkpointer[3] = 3;
    oldworkpointer[4] = 0;
```

```
    AT.WorkPointer = oldworkpointer+5;
    return(oldworkpointer);
FromGCD:
    MLOCK(ErrorMessageLock);
    MesCall("EvaluateGcd");
    MUNLOCK(ErrorMessageLock);
    return(0);
}

#endif

/*
    #] EvaluateGcd :
    #[ TreatPolyRatFun :

    if ( AR.PolyFunExp == 1 ) we have to trim the contents of the polyratfun
    down to its most divergent term and give it coefficient +1. This is done
    by taking the terms with the least power in the variable in the numerator
    and in the denominator and then combine them.
    Answer is either PolyRatFun(ep^n,1) or PolyRatFun(1,1) or PolyRatFun(1,ep^n)
*/

int TreatPolyRatFun(PHEAD WORD *prf)
{
    WORD *t, *tstop, *r, *rstop, *m, *mstop;
    WORD exp1 = MAXPOWER, exp2 = MAXPOWER;
    t = prf+FUNHEAD;
    if ( *t < 0 ) {
        if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) {
            if ( exp1 > 1 ) exp1 = 1;
            t += 2;
        }
        else {
            if ( exp1 > 0 ) exp1 = 0;
            NEXTARG(t)
        }
    }
    else {
        tstop = t + *t;
        t += ARGHEAD;
        while ( t < tstop ) {
/*
            Now look for the minimum power of AR.PolyFunVar
*/
            r = t+1;
            t += *t;
            rstop = t - ABS(t[-1]);
            while ( r < rstop ) {
                if ( *r != SYMBOL ) { r += r[1]; continue; }
                m = r;
                mstop = m + m[1];
                m += 2;
                while ( m < mstop ) {
                    if ( *m == AR.PolyFunVar ) {
                        if ( m[1] < exp1 ) exp1 = m[1];
                        break;
                    }
                    m += 2;
                }
                if ( m == mstop ) {
                    if ( exp1 > 0 ) exp1 = 0;
                }
                break;
            }
            if ( r == rstop ) {
                if ( exp1 > 0 ) exp1 = 0;
            }
        }
        t = tstop;
    }
    if ( *t < 0 ) {
        if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) {
            if ( exp2 > 1 ) exp2 = 1;
            t += 2;
        }
        else {
            if ( exp2 > 0 ) exp2 = 0;
        }
    }
    else {
        tstop = t + *t;
        t += ARGHEAD;
        while ( t < tstop ) {
/*
            Now look for the minimum power of AR.PolyFunVar
*/
            r = t+1;
            t += *t;
```

```
            rstop = t - ABS(t[-1]);
            while ( r < rstop ) {
                if ( *r != SYMBOL ) { r += r[1]; continue; }
                m = r;
                mstop = m + m[1];
                m += 2;
                while ( m < mstop ) {
                    if ( *m == AR.PolyFunVar ) {
                        if ( m[1] < exp2 ) exp2 = m[1];
                        break;
                    }
                    m += 2;
                }
                if ( m == mstop ) {
                    if ( exp2 > 0 ) exp2 = 0;
                }
                break;
            }
            if ( r == rstop ) {
                if ( exp2 > 0 ) exp2 = 0;
            }
        }
    }
/*
    Now we can compose the output.
    Notice that the output can never be longer than the input provided
    we never can have arguments that consist of just a function.
*/
    exp1 = exp1-exp2;
/*  if ( exp1 > 0 ) exp1 = 0; */
    t = prf+FUNHEAD;
    if ( exp1 == 0 ) {
        *t++ = -SNUMBER; *t++ = 1;
        *t++ = -SNUMBER; *t++ = 1;
    }
    else if ( exp1 > 0 ) {
        if ( exp1 == 1 ) {
            *t++ = -SYMBOL; *t++ = AR.PolyFunVar;
        }
        else {
            *t++ = 8+ARGHEAD;
            *t++ = 0;
            FILLARG(t);
            *t++ = 8; *t++ = SYMBOL; *t++ = 4; *t++ = AR.PolyFunVar;
            *t++ = exp1; *t++ = 1; *t++ = 1; *t++ = 3;
        }
        *t++ = -SNUMBER; *t++ = 1;
    }
    else {
        *t++ = -SNUMBER; *t++ = 1;
        if ( exp1 == -1 ) {
            *t++ = -SYMBOL; *t++ = AR.PolyFunVar;
        }
        else {
            *t++ = 8+ARGHEAD;
            *t++ = 0;
            FILLARG(t);
            *t++ = 8; *t++ = SYMBOL; *t++ = 4; *t++ = AR.PolyFunVar;
            *t++ = -exp1; *t++ = 1; *t++ = 1; *t++ = 3;
        }
    }
    prf[2] = 0;  /* Clean */
    prf[1] = t - prf;
    return(0);
}

/*
    #] TreatPolyRatFun :
    #[ DropCoefficient :
*/

void DropCoefficient(PHEAD WORD *term)
{
    GETBIDENTITY
    WORD *t = term + *term;
    WORD n, na;
    n = t[-1]; na = ABS(n);
    t -= na;
    if ( n == 3 && t[0] == 1 && t[1] == 1 ) return;
    *AN.RepPoint = 1;
    t[0] = 1; t[1] = 1; t[2] = 3;
    *term -= (na-3);
}

/*
    #] DropCoefficient :
    #[ DropSymbols :
```

```
*/

void DropSymbols(PHEAD WORD *term)
{
    GETBIDENTITY
    WORD *tend = term + *term, *t1, *t2, *tstop;
    tstop = tend - ABS(tend[-1]);
    t1 = term+1;
    while ( t1 < tstop ) {
        if ( *t1 == SYMBOL ) {
            *AN.RepPoint = 1;
            t2 = t1+t1[1];
            while ( t2 < tend ) *t1++ = *t2++;
            *term = t1 - term;
            break;
        }
        t1 += t1[1];
    }
}

/*
    #] DropSymbols :
    #[ SymbolNormalize :
*/
int SymbolNormalize(WORD *term)
{
    GETIDENTITY
    WORD buffer[7*NORMSIZE], *t, *b, *bb, *tt, *m, *tstop;
    int i;
    b = buffer;
    *b++ = SYMBOL; *b++ = 2;
    t = term + *term;
    tstop = t - ABS(t[-1]);
    t = term + 1;
    while ( t < tstop ) {   /* Step 1: collect symbols */
      if ( *t == SYMBOL && t < tstop ) {
        for ( i = 2; i < t[1]; i += 2 ) {
            bb = buffer+2;
            while ( bb < b ) {
                if ( bb[0] == t[i] ) {  /* add powers */
                    bb[1] += t[i+1];
                    if ( bb[1] > MAXPOWER || bb[1] < -MAXPOWER ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Power in SymbolNormalize out of range");
                        MUNLOCK(ErrorMessageLock);
                        return(-1);
                    }
                    if ( bb[1] == 0 ) {
                        b -= 2;
                        while ( bb < b ) {
                            bb[0] = bb[2]; bb[1] = bb[3]; bb += 2;
                        }
                    }
                    goto Nexti;
                }
                else if ( bb[0] > t[i] ) { /* insert it */
                    m = b;
                    while ( m > bb ) { m[1] = m[-1]; m[0] = m[-2]; m -= 2; }
                    b += 2;
                    bb[0] = t[i];
                    bb[1] = t[i+1];
                    goto Nexti;
                }
                bb += 2;
            }
            if ( bb >= b ) { /* add it to the end */
                *b++ = t[i]; *b++ = t[i+1];
            }
Nexti:;
        }
      }
      else {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal term in SymbolNormalize");
        MUNLOCK(ErrorMessageLock);
        return(-1);
      }
      t += t[1];
    }
    buffer[1] = b - buffer;
/*
    Veto negative powers
*/
    if ( AT.LeaveNegative == 0 ) {
        b = buffer; bb = b + b[1]; b += 3;
        while ( b < bb ) {
            if ( *b < 0 ) {
```

```
                MLOCK(ErrorMessageLock);
                MesPrint("Negative power in SymbolNormalize");
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
            b += 2;
        }
    }
/*
    Now we use the fact that the new term will not be longer than the old one
    Actually it should be shorter when there is more than one subterm!
    Copy back.
*/
    i = buffer[1];
    b = buffer; tt = term + 1;
    if ( i > 2 ) { NCOPY(tt,b,i) }
    if ( tt < tstop ) {
        i = term[*term-1];
        if ( i < 0 ) i = -i;
        *term -= (tstop-tt);
        NCOPY(tt,tstop,i)
    }
    return(0);
}

/*
    #] SymbolNormalize :
    #[ TestFunFlag :

    Tests whether a function still has unsubstituted subexpressions
    This function has its dirtyflag on!
*/

int TestFunFlag(PHEAD WORD *tfun)
{
    WORD *t, *tstop, *r, *rstop, *m, *mstop;
    if ( functions[*tfun-FUNCTION].spec ) return(0);
    tstop = tfun + tfun[1];
    t = tfun + FUNHEAD;
    while ( t < tstop ) {
        if ( *t < 0 ) { NEXTARG(t); continue; }
        rstop = t + *t;
        if ( t[1] == 0 ) { t = rstop; continue; }
        r = t + ARGHEAD;
        while ( r < rstop ) { /* Here we loop over terms */
            m = r+1; mstop = r+*r; mstop -= ABS(mstop[-1]);
            while ( m < mstop ) {   /* Loop over the subterms */
                if ( *m == SUBEXPRESSION || *m == EXPRESSION || *m == DOLLAREXPRESSION ) return(1);
                if ( ( *m >= FUNCTION ) && ( ( m[2] & DIRTYFLAG ) == DIRTYFLAG )
                    && ( *m != REPLACEMENT ) && TestFunFlag(BHEAD m) ) return(1);
                m += m[1];
            }
            r += *r;
        }
        t += *t;
    }
    return(0);
}

/*
    #] TestFunFlag :
    #[ BracketNormalize :
*/

#define EXCHN(t1,t2,n) { WORD a,i; for(i=0;i<n;i++){x=t1[i];t1[i]=t2[i];t2[i]=a;}
#define EXCH(x,y) { WORD a = (x); (x) = (y); (y) = a; }

WORD BracketNormalize(PHEAD WORD *term)
{
    WORD *stop = term+*term-3, *t, *tt, *tstart, *r;
    WORD *oldwork = AT.WorkPointer;
    WORD *termout;
    WORD i, ii, j;
    termout = AT.WorkPointer = term+*term;
/*
    First collect all functions and sort them
*/
    tt = termout+1; t = term+1;
    while ( t < stop ) {
        if ( *t >= FUNCTION ) { i = t[1]; NCOPY(tt,t,i); }
        else t += t[1];
    }
    if ( tt > termout+1 && tt-termout-1 > termout[2] ) { /* sorting */
        r = termout+1; ii = tt-r;
        for ( i = 0; i < ii-FUNHEAD; i += FUNHEAD ) {   /* Bubble sort */
            for ( j = i+FUNHEAD; j > 0; j -= FUNHEAD ) {
                if ( functions[r[j-FUNHEAD]-FUNCTION].commute
```

```
                  && functions[r[j]-FUNCTION].commute == 0 ) break;
                if ( r[j-FUNHEAD] > r[j] ) EXCH(r[j-FUNHEAD],r[j])
                else break;
            }
        }
    }

    tstart = tt; t = term + 1; *tt++ = DELTA; *tt++ = 2;
    while ( t < stop ) {
        if ( *t == DELTA ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
        else t += t[1];
    }
    if ( tstart[1] > 2 ) {
        for ( r = tstart+2; r < tstart+tstart[1]; r += 2 ) {
            if ( r[0] > r[1] ) EXCH(r[0],r[1])
        }
    }
    if ( tstart[1] > 4 ) { /* sorting */
        r = tstart+2; ii = tstart[1]-2;
        for ( i = 0; i < ii-2; i += 2 ) {   /* Bubble sort */
            for ( j = i+2; j > 0; j -= 2 ) {
                if ( r[j-2] > r[j] ) {
                    EXCH(r[j-2],r[j])
                    EXCH(r[j-1],r[j+1])
                }
                else if ( r[j-2] < r[j] ) break;
                else {
                    if ( r[j-1] > r[j+1] ) EXCH(r[j-1],r[j+1])
                    else break;
                }
            }
        }
        tt = tstart+tstart[1];
    }
    else if ( tstart[1] == 2 ) { tt = tstart; }
    else tt = tstart+4;

    tstart = tt; t = term + 1; *tt++ = INDEX; *tt++ = 2;
    while ( t < stop ) {
        if ( *t == INDEX ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
        else t += t[1];
    }
    if ( tstart[1] >= 4 ) { /* sorting */
        r = tstart+2; ii = tstart[1]-2;
        for ( i = 0; i < ii-1; i += 1 ) {   /* Bubble sort */
            for ( j = i+1; j > 0; j -= 1 ) {
                if ( r[j-1] > r[j] ) EXCH(r[j-1],r[j])
                else break;
            }
        }
        tt = tstart+tstart[1];
    }
    else if ( tstart[1] == 2 ) { tt = tstart; }
    else tt = tstart+3;

    tstart = tt; t = term + 1; *tt++ = DOTPRODUCT; *tt++ = 2;
    while ( t < stop ) {
        if ( *t == DOTPRODUCT ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
        else t += t[1];
    }
    if ( tstart[1] > 5 ) { /* sorting */
        r = tstart+2; ii = tstart[1]-2;
        for ( i = 0; i < ii; i += 3 ) {
            if ( r[i] > r[i+1] ) EXCH(r[i],r[i+1])
        }
        for ( i = 0; i < ii-3; i += 3 ) {   /* Bubble sort */
            for ( j = i+3; j > 0; j -= 3 ) {
                if ( r[j-3] < r[j] ) break;
                if ( r[j-3] > r[j] ) {
                    EXCH(r[j-3],r[j])
                    EXCH(r[j-2],r[j+1])
                }
                else {
                    if ( r[j-2] > r[j+1] ) EXCH(r[j-2],r[j+1])
                    else break;
                }
            }
        }
        tt = tstart+tstart[1];
    }
    else if ( tstart[1] == 2 ) { tt = tstart; }
    else {
        if ( tstart[2] > tstart[3] ) EXCH(tstart[2],tstart[3])
        tt = tstart+5;
    }

    tstart = tt; t = term + 1; *tt++ = SYMBOL; *tt++ = 2;
```

```
    while ( t < stop ) {
        if ( *t == SYMBOL ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
        else t += t[1];
    }
    if ( tstart[1] > 4 ) { /* sorting */
        r = tstart+2; ii = tstart[1]-2;
        for ( i = 0; i < ii-2; i += 2 ) {   /* Bubble sort */
            for ( j = i+2; j > 0; j -= 2 ) {
                if ( r[j-2] > r[j] ) EXCH(r[j-2],r[j])
                else break;
            }
        }
        tt = tstart+tstart[1];
    }
    else if ( tstart[1] == 2 ) { tt = tstart; }
    else tt = tstart+4;

    tstart = tt; t = term + 1; *tt++ = SETSET; *tt++ = 2;
    while ( t < stop ) {
        if ( *t == SETSET ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
        else t += t[1];
    }
    if ( tstart[1] > 4 ) { /* sorting */
        r = tstart+2; ii = tstart[1]-2;
        for ( i = 0; i < ii-2; i += 2 ) {   /* Bubble sort */
            for ( j = i+2; j > 0; j -= 2 ) {
                if ( r[j-2] > r[j] ) {
                    EXCH(r[j-2],r[j])
                    EXCH(r[j-1],r[j+1])
                }
                else break;
            }
        }
        tt = tstart+tstart[1];
    }
    else if ( tstart[1] == 2 ) { tt = tstart; }
    else tt = tstart+4;
    *tt++ = 1; *tt++ = 1; *tt++ = 3;
    t = term; i = *termout = tt - termout; tt = termout;
    NCOPY(t,tt,i);
    AT.WorkPointer = oldwork;
    return(0);
}

/*
    #] BracketNormalize :
*/
```

4.69 notation.c File Reference

`#include "form3.h"`

Include dependency graph for notation.c:

Functions

- int NormPolyTerm (PHEAD WORD *term)
- int ConvertToPoly (PHEAD WORD *term, WORD *outterm, WORD *comlist, WORD par)

- int LocalConvertToPoly (PHEAD WORD ∗term, WORD ∗outterm, WORD startebuf, WORD par)
- int ConvertFromPoly (PHEAD WORD ∗term, WORD ∗outterm, WORD from, WORD to, WORD offset, WORD par)
- WORD FindSubterm (WORD ∗subterm)
- WORD FindLocalSubterm (PHEAD WORD ∗subterm, WORD startebuf)
- void PrintSubtermList (int from, int to)
- void PrintExtraSymbol (int num, WORD ∗terms, int par)
- WORD FindSubexpression (WORD ∗subexpr)
- int ExtraSymFun (PHEAD WORD ∗term, WORD level)
- int PruneExtraSymbols (WORD downto)

4.69.1 Detailed Description

Contains the functions that deal with the rewriting and manipulation of expressions/terms in polynomial representation.

Definition in file notation.c.

4.69.2 Function Documentation

4.69.2.1 NormPolyTerm()

```
int NormPolyTerm (
            PHEAD WORD * term )
```

Definition at line 53 of file notation.c.

4.69.2.2 ConvertToPoly()

```
int ConvertToPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD * comlist,
            WORD par )
```

Definition at line 307 of file notation.c.

4.69.2.3 LocalConvertToPoly()

```
int LocalConvertToPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD startebuf,
            WORD par )
```

Converts a generic term to polynomial notation in which there are only symbols and brackets. During conversion there will be only symbols. Brackets are stripped. Objects that need 'translation' are put inside a special compiler buffer and represented by a symbol. The numbering of the extra symbols is down from the maximum. In principle there can be a problem when running into the already assigned ones. This uses the FindTree for searching in the global tree and then looks further in the AT.ebufnum. This allows fully parallel processing. Hence we need no locks. Cannot be used in the same module as ConvertToPoly.

Definition at line 510 of file notation.c.

4.69.2.4 ConvertFromPoly()

```
int ConvertFromPoly (
            PHEAD WORD * term,
            WORD * outterm,
            WORD from,
            WORD to,
            WORD offset,
            WORD par )
```

Definition at line 660 of file notation.c.

4.69.2.5 FindSubterm()

```
WORD FindSubterm (
            WORD * subterm )
```

Definition at line 773 of file notation.c.

4.69.2.6 FindLocalSubterm()

```
WORD FindLocalSubterm (
            PHEAD WORD * subterm,
            WORD startebuf )
```

Definition at line 842 of file notation.c.

4.69.2.7 PrintSubtermList()

```
void PrintSubtermList (
            int from,
            int to )
```

Definition at line 895 of file notation.c.

4.69.2.8 PrintExtraSymbol()

```
void PrintExtraSymbol (
            int num,
            WORD * terms,
            int par )
```

Definition at line 1007 of file notation.c.

4.69.2.9 FindSubexpression()

```
WORD FindSubexpression (
            WORD * subexpr )
```

Definition at line 1093 of file notation.c.

4.69.2.10 ExtraSymFun()

```
int ExtraSymFun (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1145 of file notation.c.

4.69.2.11 PruneExtraSymbols()

```
int PruneExtraSymbols (
            WORD downto )
```

Definition at line 1213 of file notation.c.

4.70 notation.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ Includes :
00034 */
00035
00036 #include "form3.h"
00037
00038 /*
00039     #] Includes :
00040         #[ NormPolyTerm :
00041
00042         Brings a term to normal form.
00043
00044         This routine knows objects of the following types:
00045             SYMBOL
00046             HAAKJE
00047             SNUMBER
00048             LNUMBER
00049         The SNUMBER and LNUMBER are worked into the coefficient.
00050         One of the essences here is that everything can be done in place.
00051 */
00052
00053 int NormPolyTerm(PHEAD WORD *term)
00054 {
00055     WORD *tcoef, ncoef, *tstop, *tfill, *t, *tt;
00056     int equal, i;
00057     WORD *r1, *r2, *r3, *r4, *r5, *rfirst, rv;
```

```
    WORD *lnum, nnum;       /* Scratch, originally for factorials */
/*
    One: find the coefficient
*/
    tcoef = term+*term;
    ncoef = tcoef[-1];
    tstop = tcoef - ABS(tcoef[-1]);
    tfill = t = term + 1;
    rfirst = 0;
    if ( t >= tstop ) { return(*term); }
    while ( t < tstop ) {
        switch ( *t ) {
            case SYMBOL:
                if ( rfirst == 0 ) {
/*
                    Here we only need to sort
                    1: assume no equals. Bubble.
*/
                    rfirst = t;
                    r2 = rfirst+4; tt = r3 = t + t[1]; equal = 0;
                    while ( r2 < r3 ) {
                        r1 = r2 - 2;
                        if ( *r2 > *r1 ) { r2 += 2; continue; }
                        if ( *r2 == *r1 ) { r2 += 2; equal = 1; continue; }
                        rv = *r1; *r1 = *r2; *r2 = rv;
                        r1 -= 2; r2 -= 2; r4 = r2 + 2;
                        while ( r1 > t ) {
                            if ( *r2 >= *r1 ) { r2 = r4; break; }
                            rv = *r1; *r1 = *r2; *r2 = rv;
                            r1 -= 2; r2 -= 2;
                        }
                    }
/*
                    2: hunt down the equal objects
                       postpone eliminating zero powers.
*/
                    if ( equal ) {
                        r1 = t+2; r2 = r1+2;
                        while ( r2 < r3 ) {
                            if ( *r1 == *r2 ) {
                                r1[1] += r2[1];
                                r4 = r2+2;
                                while ( r4 < r3 ) *r2++ = *r4++;
                                t[1] -= 2;
                                r2 = r1 + 2; r3 -= 2;
                            }
                        }
                    }
                }
                else {
/*
                    Here we only need to insert
*/
                    r1 = t + 2; tt = r3 = t + t[1];
                    while ( r1 < r3 ) {
                        r2 = rfirst+2; r4 = rfirst + rfirst[1];
                        while ( r2 < r4 ) {
                            if ( *r1 == *r2 ) {
                                r2[1] += r1[1];
                                break;
                            }
                            else if ( *r2 > *r1 ) {
                                r5 = r4;
                                while ( r5 > r2 ) { r5[1] = r5[-1]; r5[0] = r5[-2]; r5 -= 2; }
                                rfirst[1] += 2;
                                *r2 = *r1; r2[1] = r1[1];
                                break;
                            }
                            r2 += 2;
                        }
                        if ( r2 == r4 ) {
                            rfirst[1] += 2;
                            *r2++ = *r1++; *r2++ = *r1++;
                        }
                        else r1 += 2;
                    }
                }
                t = tt;
                break;
            case HAAKJE:    /* Here we skip brackets */
                t += t[1];
                break;
            case SNUMBER:
                if ( t[2] < 0 ) {
                    t[2] = -t[2];
                    if ( t[3] & 1 ) ncoef = -ncoef;
                }
```

```
                else if ( t[2] == 0 ) {
                    if ( t[3] < 0 ) goto NormInf;
                    goto NormZero;
                }
                lnum = TermMalloc("lnum");
                lnum[0] = t[2];
                nnum = 1;
                if ( t[3] && RaisPow(BHEAD (UWORD *)lnum,&nnum,(UWORD)(ABS(t[3]))) ) goto FromNorm;
                ncoef = REDLENG(ncoef);
                if ( t[3] < 0 ) {
                    if ( Divvy(BHEAD (UWORD *)tstop,&ncoef,(UWORD *)lnum,nnum) )
                        goto FromNorm;
                }
                else if ( t[3] > 0 ) {
                    if ( Mully(BHEAD (UWORD *)tstop,&ncoef,(UWORD *)lnum,nnum) )
                        goto FromNorm;
                }
                ncoef = INCLENG(ncoef);
                t += t[1];
                TermFree(lnum,"lnum");
                break;
            case LNUMBER:
                ncoef = REDLENG(ncoef);
                if ( Mully(BHEAD (UWORD *)tstop,&ncoef,(UWORD *)(t+3),t[2]) ) goto FromNorm;
                ncoef = INCLENG(ncoef);
                t += t[1];
                break;
            default:
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal code in NormPolyTerm");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
                break;
        }
    }
/*
    Now we try to eliminate objects to the power zero.
*/
    if ( rfirst ) {
        r2 = rfirst+2;
        r3 = rfirst + rfirst[1];
        while ( r2 < r3 ) {
            if ( r2[1] == 0 ) {
                r1 = r2 + 2;
                while ( r1 < r3 ) { r1[-2] = r1[0]; r1[-1] = r1[1]; r1 += 2; }
                r3 -= 2;
                rfirst[1] -= 2;
            }
            else { r2 += 2; }
        }
        if ( rfirst[1] < 4 ) rfirst = 0;
    }
/*
    Finally we put the term together
*/
    if ( rfirst ) {
        i = rfirst[1];
        NCOPY(tfill,rfirst,i)
    }
    i = ABS(ncoef)-1;
    NCOPY(tfill,tstop,i)
    *tfill++ = ncoef;
    *term = tfill - term;
    return(*term);
NormZero:
    *term = 0;
    return(0);
NormInf:
    MLOCK(ErrorMessageLock);
    MesPrint("0^0 in NormPolyTerm");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(-1);
FromNorm:
    MLOCK(ErrorMessageLock);
    MesCall("NormPolyTerm");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(-1);
}

/*
        #] NormPolyTerm :
        #[ ComparePoly :
*/
#ifdef WITHCOMPAREPOLY

```

```
WORD ComparePoly(WORD *term1, WORD *term2, WORD level)
{
    WORD *t1, *t2, *t3, *t4, *tstop1, *tstop2;
    tstop1 = term1 + *term1;
    tstop1 -= ABS(tstop1[-1]);
    tstop2 = term2 + *term2;
    tstop2 -= ABS(tstop2[-1]);
    t1 = term1+1;
    t2 = term2+1;
    while ( t1 < tstop1 && t2 < tstop2 ) {
        if ( *t1 == *t2 ) {
            if ( *t1 == HAAKJE ) {
                if ( t1[2] != t2[2] ) return(t2[2]-t1[2]);
                t1 += t1[1]; t2 += t2[1];
            }
            else {  /* must be type SYMBOL */
                t3 = t1 + t1[1]; t4 = t2 + t2[1];
                t1 += 2; t2 += 2;
                while ( t1 < t3 && t2 < t4 ) {
                    if ( *t1 != *t2 ) return(*t2-*t1);
                    if ( t1[1] != t2[1] ) return(t2[1]-t1[1]);
                    t1 += 2; t2 += 2;
                }
                if ( t1 < t3 ) return(-1);
                if ( t2 < t4 ) return(1);
            }
        }
        else return(*t2-*t1);
    }
    if ( t1 < tstop1 ) return(-1);
    if ( t2 < tstop2 ) return(1);
    return(0);
}

#endif

/*
        #] ComparePoly :
        #[ ConvertToPoly :
*/
static int FirstWarnConvertToPoly = 1;

int ConvertToPoly(PHEAD WORD *term, WORD *outterm, WORD *comlist, WORD par)
{
    WORD *tout, *tstop, ncoef, *t, *r, *tt, *ttwo = 0;
    int i, action = 0;
    tt = term + *term;
    ncoef = ABS(tt[-1]);
    tstop = tt - ncoef;
    tout = outterm+1;
    t = term + 1;
    if ( comlist[2] == DOALL ) {
      while ( t < tstop ) {
        if ( *t == SYMBOL ) {
            r = t+2;
            t += t[1];
            while ( r < t ) {
                if ( r[1] > 0 ) {
                    *tout++ = SYMBOL;
                    *tout++ = 4;
                    *tout++ = r[0];
                    *tout++ = r[1];
                }
                else {
                    tout[1] = SYMBOL;
                    tout[2] = 4;
                    tout[3] = r[0];
                    tout[4] = -1;
                    i = FindSubterm(tout+1);
                    *tout++ = SYMBOL;
                    *tout++ = 4;
                    *tout++ = MAXVARIABLES-i;
                    *tout++ = -r[1];
                    action = 1;
                }
                r += 2;
            }
        }
        else if ( *t == DOTPRODUCT ) {
            r = t + 2;
            t += t[1];
            while ( r < t ) {
                tout[1] = DOTPRODUCT;
                tout[2] = 5;
                tout[3] = r[0];
                tout[4] = r[1];
                if ( r[2] < 0 ) {
```

```
                    tout[5] = -1;
                }
                else {
                    tout[5] = 1;
                }
                i = FindSubterm(tout+1);
                *tout++ = SYMBOL;
                *tout++ = 4;
                *tout++ = MAXVARIABLES-i;
                *tout++ = ABS(r[2]);
                r += 3;
                action = 1;
            }
        }
        else if ( *t == VECTOR ) {
            r = t + 2;
            t += t[1];
            while ( r < t ) {
                tout[1] = VECTOR;
                tout[2] = 4;
                tout[3] = r[0];
                tout[4] = r[1];
                i = FindSubterm(tout+1);
                *tout++ = SYMBOL;
                *tout++ = 4;
                *tout++ = MAXVARIABLES-i;
                *tout++ = 1;
                r += 2;
                action = 1;
            }
        }
        else if ( *t == INDEX ) {
            r = t + 2;
            t += t[1];
            while ( r < t ) {
                tout[1] = INDEX;
                tout[2] = 3;
                tout[3] = r[0];
                i = FindSubterm(tout+1);
                *tout++ = SYMBOL;
                *tout++ = 4;
                *tout++ = MAXVARIABLES-i;
                *tout++ = 1;
                r++;
                action = 1;
            }
        }
        else if ( *t == HAAKJE) {
            if ( par ) {
                tout[0] = 1; tout[1] = 1; tout[2] = 3;
                *outterm = (tout+3)-outterm;
                if ( NormPolyTerm(BHEAD outterm) < 0 ) return(-1);
                tout = outterm + *outterm;
                tout -= 3;
                i = t[1]; NCOPY(tout,t,i);
                ttwo = tout-1;
            }
            else { t += t[1]; }
        }
        else if ( *t >= FUNCTION ) {
            i = FindSubterm(t);
            t += t[1];
            *tout++ = SYMBOL;
            *tout++ = 4;
            *tout++ = MAXVARIABLES-i;
            *tout++ = 1;
            action = 1;
        }
        else {
            if ( FirstWarnConvertToPoly ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal object in conversion to polynomial notation");
                MUNLOCK(ErrorMessageLock);
                FirstWarnConvertToPoly = 0;
            }
            return(-1);
        }
    }
    NCOPY(tout,tstop,ncoef)
    if ( ttwo ) {
      WORD hh = *ttwo;
      *ttwo = tout-ttwo;
      if ( ( i = NormPolyTerm(BHEAD ttwo) ) >= 0 ) i = action;
      tout = ttwo + *ttwo;
      *ttwo = hh;
      *outterm = tout - outterm;
    }
```

```
      else {
        *outterm = tout-outterm;
        if ( ( i = NormPolyTerm(BHEAD outterm) ) >= 0 ) i = action;
      }
    }
    else if ( comlist[2] == ONLYFUNCTIONS ) {
      while ( t < tstop ) {
        if ( *t >= FUNCTION ) {
            if ( comlist[1] == 3 ) {
                i = FindSubterm(t);
                t += t[1];
                *tout++ = SYMBOL;
                *tout++ = 4;
                *tout++ = MAXVARIABLES-i;
                *tout++ = 1;
                action = 1;
            }
            else {
                for ( i = 3; i < comlist[1]; i++ ) {
                    if ( *t == comlist[i] ) break;
                }
                if ( i < comlist[1] ) {
                    i = FindSubterm(t);
                    t += t[1];
                    *tout++ = SYMBOL;
                    *tout++ = 4;
                    *tout++ = MAXVARIABLES-i;
                    *tout++ = 1;
                    action = 1;
                }
                else {
                    i = t[1]; NCOPY(tout,t,i);
                }
            }
        }
        else {
            i = t[1]; NCOPY(tout,t,i);
        }
      }
      NCOPY(tout,tstop,ncoef)
      *outterm = tout-outterm;
      Normalize(BHEAD outterm);
      i = action;
    }
    else {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal internal code in conversion to polynomial notation");
        MUNLOCK(ErrorMessageLock);
        i = -1;
    }
    return(i);
}

/*
        #] ConvertToPoly :
        #[ LocalConvertToPoly :
*/
int LocalConvertToPoly(PHEAD WORD *term, WORD *outterm, WORD startebuf, WORD par)
{
    WORD *tout, *tstop, ncoef, *t, *r, *tt, *ttwo = 0;
    int i, action = 0;
    tt = term + *term;
    ncoef = ABS(tt[-1]);
    tstop = tt - ncoef;
    tout = outterm+1;
    t = term + 1;
    while ( t < tstop ) {
        if ( *t == SYMBOL ) {
            r = t+2;
            t += t[1];
            while ( r < t ) {
                if ( r[1] > 0 ) {
                    *tout++ = SYMBOL;
                    *tout++ = 4;
                    *tout++ = r[0];
                    *tout++ = r[1];
                }
                else {
                    tout[1] = SYMBOL;
                    tout[2] = 4;
                    tout[3] = r[0];
                    tout[4] = -1;
                    i = FindLocalSubterm(BHEAD tout+1,startebuf);
                    *tout++ = SYMBOL;
                    *tout++ = 4;
                    *tout++ = MAXVARIABLES-i;
                    *tout++ = -r[1];
```

```
                    action = 1;
                }
                r += 2;
            }
        }
        else if ( *t == DOTPRODUCT ) {
            r = t + 2;
            t += t[1];
            while ( r < t ) {
                tout[1] = DOTPRODUCT;
                tout[2] = 5;
                tout[3] = r[0];
                tout[4] = r[1];
                if ( r[2] < 0 ) {
                    tout[5] = -1;
                }
                else {
                    tout[5] = 1;
                }
                i = FindLocalSubterm(BHEAD tout+1,startebuf);
                *tout++ = SYMBOL;
                *tout++ = 4;
                *tout++ = MAXVARIABLES-i;
                *tout++ = ABS(r[2]);
                r += 3;
                action = 1;
            }
        }
        else if ( *t == VECTOR ) {
            r = t + 2;
            t += t[1];
            while ( r < t ) {
                tout[1] = VECTOR;
                tout[2] = 4;
                tout[3] = r[0];
                tout[4] = r[1];
                i = FindLocalSubterm(BHEAD tout+1,startebuf);
                *tout++ = SYMBOL;
                *tout++ = 4;
                *tout++ = MAXVARIABLES-i;
                *tout++ = 1;
                r += 2;
                action = 1;
            }
        }
        else if ( *t == INDEX ) {
            r = t + 2;
            t += t[1];
            while ( r < t ) {
                tout[1] = INDEX;
                tout[2] = 3;
                tout[3] = r[0];
                i = FindLocalSubterm(BHEAD tout+1,startebuf);
                *tout++ = SYMBOL;
                *tout++ = 4;
                *tout++ = MAXVARIABLES-i;
                *tout++ = 1;
                r++;
                action = 1;
            }
        }
        else if ( *t == HAAKJE) {
            if ( par ) {
                tout[0] = 1; tout[1] = 1; tout[2] = 3;
                *outterm = (tout+3)-outterm;
                if ( NormPolyTerm(BHEAD outterm) < 0 ) return(-1);
                tout = outterm + *outterm;
                tout -= 3;
                i = t[1]; NCOPY(tout,t,i);
                ttwo = tout-1;
            }
            else { t += t[1]; }
        }
        else if ( *t >= FUNCTION ) {
            i = FindLocalSubterm(BHEAD t,startebuf);
            t += t[1];
            *tout++ = SYMBOL;
            *tout++ = 4;
            *tout++ = MAXVARIABLES-i;
            *tout++ = 1;
            action = 1;
        }
        else {
            if ( FirstWarnConvertToPoly ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal object in conversion to polynomial notation");
                MUNLOCK(ErrorMessageLock);
```

```
                FirstWarnConvertToPoly = 0;
            }
            return(-1);
        }
    }
    NCOPY(tout,tstop,ncoef)
    if ( ttwo ) {
        WORD hh = *ttwo;
        *ttwo = tout-ttwo;
        if ( ( i = NormPolyTerm(BHEAD ttwo) ) >= 0 ) i = action;
        tout = ttwo + *ttwo;
        *ttwo = hh;
        *outterm = tout - outterm;
    }
    else {
        *outterm = tout-outterm;
        if ( ( i = NormPolyTerm(BHEAD outterm) ) >= 0 ) i = action;
    }
    return(i);
}

/*
        #] LocalConvertToPoly :
        #[ ConvertFromPoly :

        Converts a generic term from polynomial notation to the original
        in which the extra symbols have been replaced by their values.
        The output is in outterm.
        We only deal with the extra symbols in the range from < i <= to
        The output has to be sent to TestSub because it may contain
        subexpressions when extra symbols have been replaced.
*/

int ConvertFromPoly(PHEAD WORD *term, WORD *outterm, WORD from, WORD to, WORD offset, WORD par)
{
    WORD *tout, *tstop, *tstop1, ncoef, *t, *r, *tt;
    int i;
/*  first = 1; */
    tt = term + *term;
    tout = outterm+1;
    ncoef = ABS(tt[-1]);
    tstop = tt - ncoef;
/*
    r = t = term + 1;
    while ( t < tstop ) {
        if ( *t == SYMBOL ) {
            tstop1 = t + t[1];
            tt = t + 2;
            while ( tt < tstop1 ) {
                if ( ( *tt < MAXVARIABLES - to )
                  || ( *tt >= MAXVARIABLES - from ) ) {
                    tt += 2;
                }
                else break;
            }
            if ( tt >= tstop1 ) { t = tstop1; continue; }
            while ( r < t ) *tout++ = *r++;
            t += 2;
            first = 0;
            while ( t < tstop1 ) {
                if ( ( *t < MAXVARIABLES - to )
                  || ( *t >= MAXVARIABLES - from ) ) {
                    *tout++ = SYMBOL;
                    *tout++ = 4;
                    *tout++ = *t++;
                    *tout++ = *t++;
                }
                else {
                    *tout++ = SUBEXPRESSION;
                    *tout++ = SUBEXPSIZE;
                    *tout++ = MAXVARIABLES - *t++ + offset;
                    *tout++ = *t++;
                    if ( par ) *tout++ = AT.ebufnum;
                    else       *tout++ = AM.sbufnum;
                    FILLSUB(tout)
                }
            }
            r = t;
        }
        else {
            t += t[1];
        }
    }
    if ( first ) {
        i = *term; t = term;
        NCOPY(outterm,t,i);
        return(*term);
```

```
    }
    while ( r < t ) *tout++ = *r++;
    NCOPY(tout,tstop,ncoef)
    *outterm = tout-outterm;
*/
    t = term + 1;
    while ( t < tstop ) {
        if ( *t == SYMBOL ) {
            tstop1 = t + t[1];
            tt = t + 2;
            while ( tt < tstop1 ) {
                if ( ( *tt < MAXVARIABLES - to )
                  || ( *tt >= MAXVARIABLES - from ) ) {
                    tt += 2;
                }
                else {
                    *tout++ = SUBEXPRESSION;
                    *tout++ = SUBEXPSIZE;
                    *tout++ = MAXVARIABLES - *tt++ + offset;
                    *tout++ = *tt++;
                    if ( par ) *tout++ = AT.ebufnum;
                    else       *tout++ = AM.sbufnum;
                    FILLSUB(tout)
                }
            }
            r = tout; t += 2;
            *tout++ = SYMBOL; *tout++ = 0;
            while ( t < tstop1 ) {
                if ( ( *t < MAXVARIABLES - to )
                  || ( *t >= MAXVARIABLES - from ) ) {
                    *tout++ = *t++;
                    *tout++ = *t++;
                }
                else { t += 2; }
            }
            r[1] = tout - r;
            if ( r[1] <= 2 ) tout = r;
        }
        else {
            i = t[1]; NCOPY(tout,t,i)
        }
    }
    NCOPY(tout,tstop,ncoef)
    *outterm = tout-outterm;
    return(*outterm);
}

/*
        #] ConvertFromPoly :
        #[ FindSubterm :

        In this routine we look up a variable.
        If we don't find it we will enter it in the subterm compiler buffer
        Searching is by tree structure.
        Adding changes the tree.

        Notice that in TFORM we should be in sequential mode.
*/

WORD FindSubterm(WORD *subterm)
{
    WORD old[5], *ss, *term, number;
    CBUF *C = cbuf + AM.sbufnum;
    LONG oldCpointer;
    term = subterm-1;
    ss = subterm+subterm[1];
/*
        Convert to proper term
*/
    old[0] = *term; old[1] = ss[0]; old[2] = ss[1]; old[3] = ss[2]; old[4] = ss[3];
    ss[0] = 1; ss[1] = 1; ss[2] = 3; ss[3] = 0; *term = subterm[1]+4;
/*
        We may have to add the term to the compiler
        buffer and then to the tree. This cannot be done in parallel and
        hence we have to set a lock.
*/
    LOCK(AM.sbuflock);

    oldCpointer = C->Pointer-C->Buffer; /* Offset of course !!!!!*/
    AddRHS(AM.sbufnum,1);
    AddNtoC(AM.sbufnum,*term,term,8);
    AddToCB(C,0)
/*
        See whether we have this one already. If not, insert it in the tree.
*/
    number = InsTree(AM.sbufnum,C->numrhs);
/*
```

```
        Restore old values and return what is needed.
*/
    if ( number < (C->numrhs) ) {   /* It existed already */
        C->Pointer = oldCpointer + C->Buffer;
        C->numrhs--;
    }
    else {
        GETIDENTITY
        WORD dim = DimensionSubterm(subterm);

        if ( dim == -MAXPOSITIVE ) {    /* Give error message but continue */
            WORD *old = AN.currentTerm;
            AN.currentTerm = term;
            MLOCK(ErrorMessageLock);
            MesPrint("Dimension out of range in %t");
            MUNLOCK(ErrorMessageLock);
            AN.currentTerm = old;
        }
/*
        Store the dimension
*/
        C->dimension[number] = dim;
    }
    UNLOCK(AM.sbuflock);

    *term = old[0]; ss[0] = old[1]; ss[1] = old[2]; ss[2] = old[3]; ss[3] = old[4];
    return(number);
}

/*
        #] FindSubterm :
        #[ FindLocalSubterm :

        In this routine we look up a variable.
        If we don't find it we will enter it in the subterm compiler buffer
        Searching is by tree structure.
        Adding changes the tree.

        Notice that in TFORM we should be in sequential mode.
*/

WORD FindLocalSubterm(PHEAD WORD *subterm, WORD startebuf)
{
    WORD old[5], *ss, *term, number, i, j, *t1, *t2;
    CBUF *C = cbuf + AT.ebufnum;
    term = subterm-1;
    ss = subterm+subterm[1];
/*
        Convert to proper term
*/
    old[0] = *term; old[1] = ss[0]; old[2] = ss[1]; old[3] = ss[2]; old[4] = ss[3];
    ss[0] = 1; ss[1] = 1; ss[2] = 3; ss[3] = 0; *term = subterm[1]+4;
/*
        First see whether we have this one already in the global buffer.
*/
    number = FindTree(AM.sbufnum,term);
    if ( number > 0 ) goto wearehappy;
/*
    Now look whether it is in the ebufnum between startebuf and numrhs
    Note however that we need an offset of (numxsymbol-startebuf)
*/
    for ( i = startebuf+1; i <= C->numrhs; i++ ) {
        t1 = C->rhs[i]; t2 = term;
        if ( *t1 == *t2 ) {
            j = *t1;
            while ( *t1 == *t2 && j > 0 ) { t1++; t2++; j--; }
            if ( j <= 0 ) {
                number = i-startebuf+numxsymbol;
                goto wearehappy;
            }
        }
    }
/*
    Now we have to add it to cbuf[AT.ebufnum]
*/
    AddRHS(AT.ebufnum,1);
    AddNtoC(AT.ebufnum,*term,term,9);
    AddToCB(C,0)
    number = C->numrhs-startebuf+numxsymbol;
wearehappy:
    *term = old[0]; ss[0] = old[1]; ss[1] = old[2]; ss[2] = old[3]; ss[3] = old[4];
    return(number);
}

/*
        #] FindLocalSubterm :
        #[ PrintSubtermList :
```

```c

        Prints all the expressions in the subterm compiler buffer.
        The format is such that they give definitions of the temporary
        variables of which the contents are stored in this buffer.
        These variables have the names Z_123 etc.
*/

void PrintSubtermList(int from,int to)
{
    UBYTE buffer[80], *out, outbuffer[300];
    int first, i, ii, inc = 1;
    WORD *term;
    CBUF *C = cbuf + AM.sbufnum;
/*
    if ( to < from ) inc = -1;
    if ( to == from ) inc = 0;
*/
    if ( from <= to ) {
        inc = 1; to += inc;
    }
    else {
        inc = -1; to += inc;
    }
    AO.OutFill = AO.OutputLine = outbuffer;
    AO.OutStop = AO.OutputLine+AC.LineLength;
    AO.IsBracket = 0;
    AO.OutSkip = 3;

    if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
        TokenToLine((UBYTE *)"      ");
        AO.OutSkip = 7;
    }
    else if ( ( AO.Optimize.debugflags & 1 ) == 1 ) {}
    else if ( AO.OutSkip > 0 ) {
        for ( i = 0; i < AO.OutSkip; i++ ) TokenToLine((UBYTE *)" ");
    }
    i = from;
    do {
        if ( ( AO.Optimize.debugflags & 1 ) == 1 ) {
            TokenToLine((UBYTE *)"id ");
            for ( ii = 3; ii < AO.OutSkip; ii++ ) TokenToLine((UBYTE *)" ");
        }
/*
        if ( AC.OutputMode == NORMALFORMAT ) {
            TokenToLine((UBYTE *)"id ");
        }
*/
        else if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {}
        else { TokenToLine((UBYTE *)" "); }

        out = StrCopy((UBYTE *)AC.extrasym,buffer);
        if ( AC.extrasymbols == 0 ) {
            out = NumCopy(i,out);
            out = StrCopy((UBYTE *)"_",out);
        }
        else if ( AC.extrasymbols == 1 ) {
            out = AddArrayIndex(i,out);
        }
        out = StrCopy((UBYTE *)"=",out);
        TokenToLine(buffer);
        term = C->rhs[i];
        first = 1;
        if ( *term == 0 ) {
            out = StrCopy((UBYTE *)"0",buffer);
            if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE ) {
                out = StrCopy((UBYTE *)";",out);
            }
            TokenToLine(buffer);
        }
        else {
            while ( *term ) {
                if ( WriteInnerTerm(term,first) ) Terminate(-1);
                term += *term;
                first = 0;
            }
            if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE ) {
                out = StrCopy((UBYTE *)";",buffer);
                TokenToLine(buffer);
            }
        }
/*
        There is a problem with FiniLine because it prepares for a
        continuation line in fortran mode.
        But the next statement should start on a blank line.
*/
/*
        FiniLine();
```

```
        if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
            AO.OutFill = AO.OutputLine;
            TokenToLine((UBYTE *)"      ");
            AO.OutSkip = 7;
        }
*/
        if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
            AO.OutSkip = 6;
            FiniLine();
            AO.OutSkip = 7;
        }
        else {
            FiniLine();
        }
        i += inc;
    } while ( i != to );
}

/*
        #] PrintSubtermList :
        #[ PrintExtraSymbol :

        Prints the definition of extra symbol num as the contents
        of the expression in terms.
        The parameter par has three options:
            EXTRASYMBOL      num is interpreted as the number of an extra symbol
            REGULARSYMBOL    num is interpreted as the number of a symbol.
                             It could still be an extra symbol.
            EXPRESSIONNUMBER num is the number of an expression.
        terms contains the rhs expression.
*/

void PrintExtraSymbol(int num, WORD *terms,int par)
{
    UBYTE buffer[80], *out, outbuffer[300];
    int first, i;
    WORD *term;

    AO.OutFill = AO.OutputLine = outbuffer;
    AO.OutStop = AO.OutputLine+AC.LineLength;
    AO.IsBracket = 0;

    if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
        TokenToLine((UBYTE *)"      ");
        AO.OutSkip = 7;
    }
    else if ( ( AO.Optimize.debugflags & 1 ) == 1 ) {
        TokenToLine((UBYTE *)"id ");
        for ( i = 3; i < AO.OutSkip; i++ ) TokenToLine((UBYTE *)" ");
    }
    else if ( AO.OutSkip > 0 ) {
        for ( i = 0; i < AO.OutSkip; i++ ) TokenToLine((UBYTE *)" ");
    }
    out = buffer;
    switch ( par ) {
        case REGULARSYMBOL:
            if ( num >= MAXVARIABLES-cbuf[AM.sbufnum].numrhs ) {
                num = MAXVARIABLES-num;
            }
            else {
                out = StrCopy(FindSymbol(num),out);
/*              out = StrCopy(VARNAME(symbols,num),out); */
                break;
            }
        case EXTRASYMBOL:
            out = StrCopy(FindExtraSymbol(num),out);
/*
            out = StrCopy((UBYTE *)AC.extrasym,out);
            if ( AC.extrasymbols == 0 ) {
                out = NumCopy(num,out);
                out = StrCopy((UBYTE *)"_",out);
            }
            else if ( AC.extrasymbols == 1 ) {
                out = AddArrayIndex(num,out);
            }
*/
            break;
        case EXPRESSIONNUMBER:
            out = StrCopy(EXPRNAME(num),out);
            break;
        default:
            MesPrint("Illegal option in PrintExtraSymbol");
            Terminate(-1);
    }
    out = StrCopy((UBYTE *)"=",out);
    TokenToLine(buffer);
    term = terms;
```

```
    first = 1;
    if ( *term == 0 ) {
        out = StrCopy((UBYTE *)"0",buffer);
        TokenToLine(buffer);
    }
    else {
        while ( *term ) {
            if ( WriteInnerTerm(term,first) ) Terminate(-1);
            term += *term;
            first = 0;
        }
    }
    if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE ) {
        out = StrCopy((UBYTE *)";",buffer);
        TokenToLine(buffer);
    }
    FiniLine();
}

/*
        #] PrintExtraSymbol :
        #[ FindSubexpression :

        In this routine we look up a subexpression.
        If we don't find it we will enter it in the subterm compiler buffer
        Searching is by tree structure.
        Adding changes the tree.

        Notice that in TFORM we should be in sequential mode.
*/

WORD FindSubexpression(WORD *subexpr)
{
    WORD *term, number;
    CBUF *C = cbuf + AM.sbufnum;
    LONG oldCpointer;

    term = subexpr;
    while ( *term ) term += *term;
    number = term - subexpr;
/*
        We may have to add the subexpression to the tree.
        This requires a lock.
*/
    LOCK(AM.sbuflock);

    oldCpointer = C->Pointer-C->Buffer; /* Offset of course !!!!!*/
    AddRHS(AM.sbufnum,1);
/*
        Add the terms to the compiler buffer. Paste on a zero.
*/
    AddNtoC(AM.sbufnum,number,subexpr,10);
    AddToCB(C,0)
/*
        See whether we have this one already. If not, insert it in the tree.
*/
    number = InsTree(AM.sbufnum,C->numrhs);
/*
        Restore old values and return what is needed.
*/
    if ( number < (C->numrhs) ) {   /* It existed already */
        C->Pointer = oldCpointer + C->Buffer;
        C->numrhs--;
    }
    else {
        GETIDENTITY
        WORD dim = DimensionExpression(BHEAD subexpr);
/*
        Store the dimension
*/
        C->dimension[number] = dim;
    }

    UNLOCK(AM.sbuflock);

    return(number);
}

/*
        #] FindSubexpression :
        #[ ExtraSymFun :
*/

int ExtraSymFun(PHEAD WORD *term,WORD level)
{
    WORD *oldworkpointer = AT.WorkPointer;
    WORD *termout, *t1, *t2, *t3, *tstop, *tend, i;
```

```
    int retval = 0;
    tend = termout = term + *term;
    tstop = tend - ABS(tend[-1]);
    t3 = t1 = term+1; t2 = termout+1;
/*
    First refind the function(s). There is at least one.
*/
    while ( t1 < tstop ) {
        if ( *t1 == EXTRASYMFUN && t1[1] == FUNHEAD+2 ) {
            if ( t1[FUNHEAD] == -SNUMBER && t1[FUNHEAD+1] <= numxsymbol
                            && t1[FUNHEAD+1] > 0 ) {
                i = t1[FUNHEAD+1];
            }
            else if ( t1[FUNHEAD] == -SYMBOL && t1[FUNHEAD+1] < MAXVARIABLES
                            && t1[FUNHEAD+1] >= MAXVARIABLES-numxsymbol ) {
                i = MAXVARIABLES - t1[FUNHEAD+1];
            }
            else goto nocase;
            while ( t3 < t1 ) *t2++ = *t3++;
/*
                Now inset the rhs pointer
*/
            *t2++ = SUBEXPRESSION;
            *t2++ = SUBEXPSIZE;
            *t2++ = i;
            *t2++ = 1;
            *t2++ = AM.sbufnum;
            FILLSUB(t2)
            t3 = t1 = t1 + t1[1];
        }
        else if ( *t1 == EXTRASYMFUN && t1[1] == FUNHEAD ) {
            while ( t3 < t1 ) *t2++ = *t3++;
            t3 = t1 = t1 + t1[1];
        }
        else {
nocase:;
            t1 = t1 + t1[1];
        }
    }
    while ( t3 < tend ) *t2++ = *t3++;
    *termout = t2 - termout;
    AT.WorkPointer = t2;
    if ( AT.WorkPointer >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        AT.WorkPointer = oldworkpointer;
        return(-1);
    }
    retval = Generator(BHEAD termout,level);
    AT.WorkPointer = oldworkpointer;
    if ( retval < 0 ) {
        MLOCK(ErrorMessageLock);
        MesCall("ExtraSymFun");
        MUNLOCK(ErrorMessageLock);
    }
    return(retval);
}

/*
        #] ExtraSymFun :
        #[ PruneExtraSymbols :
*/

int PruneExtraSymbols(WORD downto)
{
    CBUF *C = cbuf + AM.sbufnum;
    if ( downto < C->numrhs && downto >= 0 ) {  /* !!!!! */
        ClearTree(AM.sbufnum);
        C->numrhs = downto;
        if ( downto == 0 ) {
            C->Pointer = C->Buffer;
        }
        else {
            WORD *w = C->rhs[downto], i;
            while ( *w ) w += *w;
            C->Pointer = w+1;
            for ( i = 1; i <= downto; i++ ) {
                InsTree(AM.sbufnum,i);
            }
        }
    }
    return(0);
}

/*
        #] PruneExtraSymbols :
```

```
01236 */
```

4.71 opera.c File Reference

```
#include "form3.h"
```

Include dependency graph for opera.c:

Functions

- WORD EpfFind (PHEAD WORD ∗term, WORD ∗params)
- WORD EpfCon (PHEAD WORD ∗term, WORD ∗params, WORD num, WORD level)
- WORD EpfGen (WORD number, WORD ∗inlist, WORD ∗kron, WORD ∗perm, WORD sgn)
- WORD Trick (WORD ∗in, TRACES ∗t)
- WORD Trace4no (WORD number, WORD ∗kron, TRACES ∗t)
- WORD Trace4 (PHEAD WORD ∗term, WORD ∗params, WORD num, WORD level)
- WORD Trace4Gen (PHEAD TRACES ∗t, WORD number)
- WORD TraceNno (WORD number, WORD ∗kron, TRACES ∗t)
- WORD TraceN (PHEAD WORD ∗term, WORD ∗params, WORD num, WORD level)
- WORD TraceNgen (PHEAD TRACES ∗t, WORD number)
- WORD Traces (PHEAD WORD ∗term, WORD ∗params, WORD num, WORD level)
- WORD TraceFind (PHEAD WORD ∗term, WORD ∗params)
- WORD Chisholm (PHEAD WORD ∗term, WORD level)
- WORD TenVecFind (PHEAD WORD ∗term, WORD ∗params)
- WORD TenVec (PHEAD WORD ∗term, WORD ∗params, WORD num, WORD level)

4.71.1 Detailed Description

Contains the 'operations' These are the trace routines, the contractions of the Levi-Civita tensors and the tensor to vector/vector to tensor routines. The trace and contraction routines are done in a special way (see commentary with the FIXEDGLOBALS struct)

Definition in file opera.c.

4.71.2 Function Documentation

4.71.2.1 EpfFind()

```
WORD EpfFind (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 58 of file opera.c.

4.71.2.2 EpfCon()

```
WORD EpfCon (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )
```

Definition at line 212 of file opera.c.

4.71.2.3 EpfGen()

```
WORD EpfGen (
            WORD number,
            WORD * inlist,
            WORD * kron,
            WORD * perm,
            WORD sgn )
```

Definition at line 282 of file opera.c.

4.71.2.4 Trick()

```
WORD Trick (
            WORD * in,
            TRACES * t )
```

Definition at line 331 of file opera.c.

4.71.2.5 Trace4no()

```
WORD Trace4no (
            WORD number,
            WORD * kron,
            TRACES * t )
```

Definition at line 418 of file opera.c.

4.71.2.6 Trace4()

```
WORD Trace4 (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )
```

Definition at line 694 of file opera.c.

4.71.2.7 Trace4Gen()

```
WORD Trace4Gen (
            PHEAD TRACES * t,
            WORD number )
```

Definition at line 856 of file opera.c.

4.71.2.8 TraceNno()

```
WORD TraceNno (
            WORD number,
            WORD * kron,
            TRACES * t )
```

Definition at line 1341 of file opera.c.

4.71.2.9 TraceN()

```
WORD TraceN (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )
```

Definition at line 1382 of file opera.c.

4.71.2.10 TraceNgen()

```
WORD TraceNgen (
            PHEAD TRACES * t,
            WORD number )
```

Definition at line 1446 of file opera.c.

4.71.2.11 Traces()

```
WORD Traces (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )
```

Definition at line 1831 of file opera.c.

4.71.2.12 TraceFind()

```
WORD TraceFind (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 1853 of file opera.c.

4.71.2.13 Chisholm()

```
WORD Chisholm (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1963 of file opera.c.

4.71.2.14 TenVecFind()

```
WORD TenVecFind (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 2178 of file opera.c.

4.71.2.15 TenVec()

```
WORD TenVec (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )
```

Definition at line 2312 of file opera.c.

4.72 opera.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes : opera.c
*/

#include "form3.h"
/*
    int hulp;
```

```
*/

/*
    #] Includes :
    #[ Operations :
        #[ EpfFind :            WORD EpfFind(term,params)

        Searches for a pair of Levi-Civita tensors that should be
        contracted.
        If a match is found its settings are recorded in AT.TMout.
        type indicates the number of indices that is searched for,
        unless all are searched for (type = 0).
        number is the number of tensors that should survive contraction.

*/

WORD EpfFind(PHEAD WORD *term, WORD *params)
{
    GETBIDENTITY
    WORD *t, *m, *r, n1 = 0, n2, min = -1, count, fac;
    WORD *c1 = 0, *c2 = 0, sgn = 1;
    WORD *tstop, *mstop;
    UWORD *facto = (UWORD *)AT.WorkPointer;
    WORD ncoef,nfac;
    WORD number, type;
    if ( ( AT.WorkPointer = (WORD *)(facto + AM.MaxTal) ) > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    number = params[3];
    type = params[4];
    t = term;
    GETSTOP(t,tstop);
    t++;
    if ( !type ) {
        while ( *t != LEVICIVITA && t < tstop ) t += t[1];
        if ( t >= tstop ) return(0);
        m = t;
        while ( *m == LEVICIVITA && m < tstop ) { n1++; m += m[1]; }
AllLev:
        if ( n1 <= (number+1) || n1 <= 1 ) return(0);
        mstop = m;
        m = t + t[1];
        do {
            while ( m[1] == t[1] ) {
                m += FUNHEAD;
                r = t+FUNHEAD;
                count = fac = n1 = n2 = t[1] - FUNHEAD;
                while ( n1 && n2 ) {
                    if ( *m > *r ) {
                        r++; n2--;
                    }
                    else if ( *m < *r ) {
                        m++; n1--;
                    }
                    else {
                        if ( *m >= AM.OffsetIndex &&
                        ( ( *m >= AM.IndDum && AC.lDefDim == fac ) ||
                        ( *m < AM.IndDum &&
                        indices[*m-AM.OffsetIndex].dimension == fac ) ) ) {
                            count--;
                        }
                        r++; m++; n1--; n2--;
                    }
                }
                m += n1;
                if ( min < 0 || count < min ) {
                    c1 = t;
                    c2 = m - fac - FUNHEAD;
                    min = count;
                }
                if ( m >= mstop ) break;
            }
            t += t[1];
        } while ( ( m = t + t[1] ) < mstop );
    }
    else {
        fac = type + FUNHEAD;
        while ( *t != LEVICIVITA && t < tstop ) t += t[1];
        while ( *t == LEVICIVITA && t < tstop && t[1] != fac ) t += t[1];
        if ( t >= tstop ) return(0);
        m = t;
        while ( *m == LEVICIVITA && m < tstop && m[1] == fac ) { n1++; m += m[1]; }
        goto AllLev;
    }
```

```
/*
    We have now the two tensors that give the minimum contraction
    in c1 and c2.
    Prepare the AT.TMout array;
*/
    if ( min < 0 ) return(0);   /* No matching pair! */
    t = c1;
    mstop = c2;
    fac = t[1] - FUNHEAD;
    m = AT.TMout;
    *m++ = 3 + (min«1);        /* The full length */
    *m++ = CONTRACT;
    if ( number < 0 ) *m++ = 1;
    else *m++ = 0;
    n1 = n2 = t[1] - FUNHEAD;
    r = c1 + FUNHEAD;
    c1 = m;
    m = c2 + FUNHEAD;
    c2 = c1 + min;
    while ( n1 && n2 ) {
        if ( *m > *r ) { *c1++ = *r++; n2--; }
        else if ( *m < *r ) { *c2++ = *m++; n1--; }
        else {
            if ( *m < AM.OffsetIndex || ( *m < AM.IndDum &&
            ( indices[*m-AM.OffsetIndex].dimension != fac ) ) ||
            ( *m >= AM.IndDum && AC.lDefDim != fac ) ) {
                *c1++ = *r++; *c2++ = *m++;
            }
            else { if ( ( n1 ^ n2 ) & 1 ) sgn = -sgn; r++; m++; }
            n1--; n2--;
        }
    }
    if ( n1 ) { NCOPY(c2,m,n1); }
    else if ( n2 ) { NCOPY(c1,r,n2); }
    fac -= min;
    m = t + t[1];
    while ( m < mstop ) *t++ = *m++;
    m += m[1];
    while ( m < tstop ) *t++ = *m++;
    *t++ = SUBEXPRESSION;
    *t++ = SUBEXPSIZE;
    *t++ = -1;
    *t++ = 1;
    *t++ = DUMMYBUFFER;
    FILLSUB(t)
    r = term;
    r += *r - 1;
    mstop = r;
    ncoef = REDLENG(*r);
    tstop = t;
    while ( m < mstop ) *t++ = *m++;
    if ( Factorial(BHEAD fac,facto,&nfac) || Mully(BHEAD (UWORD *)tstop,&ncoef,facto,nfac) ) {
        MLOCK(ErrorMessageLock);
        MesCall("EpfFind");
        MUNLOCK(ErrorMessageLock);
        SETERROR(-1)
    }
    tstop += (ABS(ncoef))«1;
    if ( sgn < 0 ) ncoef = -ncoef;
    ncoef «= 1;
    *tstop++ = (ncoef<0)?(ncoef-1):(ncoef+1);
    *term = WORDDIF(tstop,term);
    return(1);
}

/*
        #] EpfFind :
        #[ EpfCon :             WORD EpfCon(term,params,num,level)

        Contraction of two strings of indices/vectors. They come
        from LeviCivita tensors that are being contracted.
        term is the term with the subterm to be replaced.
        params is the full indicator:
            Length, number, raise, parameters.
        Length is the length of the parameter field.
        number is the number of the operation.
        raise tells whether level should be raised afterwards.
        the parameters are the two strings.
        level is the id level.
        The factorial has been multiplied in already.

*/

WORD EpfCon(PHEAD WORD *term, WORD *params, WORD num, WORD level)
{
    GETBIDENTITY
    WORD *kron, *perm, *termout, *tstop, size2;
```

```
    WORD *m, *t, sizes, sgn = 0, i;
    sizes = *params - 3;
    kron = AT.WorkPointer;
    perm = (AT.WorkPointer += sizes);
    termout = (AT.WorkPointer += sizes);
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    if ( AT.WorkPointer > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    params += 2;
    if ( !(*params++) ) level--;
    size2 = sizes»1;
    if ( !size2 ) goto DoOnce;
    while ( ( sgn = EpfGen(size2,params,kron,perm,sgn) ) != 0 ) {
DoOnce:
        t = term;
        GETSTOP(t,tstop);
        m = termout;
        tstop -= SUBEXPSIZE;
        while ( t < tstop ) *m++ = *t++;
        if ( t[2] != num || *t != SUBEXPRESSION ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Serious error in EpfCon");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        tstop += SUBEXPSIZE;
        if ( sizes ) {
            *m++ = DELTA;
            *m++ = sizes + 2;
            t = kron;
            i = sizes;
            if ( i ) { NCOPY(m,t,i); }
        }
        t = tstop;
        tstop = term + *term;
        while ( t < tstop ) *m++ = *t++;
        *termout = WORDDIF(m,termout);
        m--;
        if ( sgn < 0 ) *m = - *m;
        if ( *termout ) {
            *AN.RepPoint = 1;
            AR.expchanged = 1;
            AT.WorkPointer = termout + *termout;
            if ( Generator(BHEAD termout,level) < 0 ) goto EpfCall;
        }
    }
    AT.WorkPointer = kron;
    return(0);
EpfCall:
    if ( AM.tracebackflag ) {
        MLOCK(ErrorMessageLock);
        MesCall("EpfCon");
        MUNLOCK(ErrorMessageLock);
    }
    return(-1);
}

/*
        #] EpfCon :
        #[ EpfGen :             WORD EpfGen(number,inlist,kron,perm,sgn)
*/

WORD EpfGen(WORD number, WORD *inlist, WORD *kron, WORD *perm, WORD sgn)
{
    WORD i, *in2, k, a;
    if ( !sgn ) {
        in2 = inlist + number;
        number «= 1;
        for ( i = 1; i < number; i += 2 ) {
            *perm++ = i;
            *perm++ = i;
            *kron++ = *inlist++;
            *kron++ = *in2++;
        }
        if ( number <= 0 ) return(0);
        else return(1);
    }
    number «= 1;
    i = number - 1;
    while ( ( i -= 2 ) >= 0 ) {
        if ( ( k = perm[i] ) != i ) {
            sgn = -sgn;
            a = kron[i];
```

```
            kron[i] = kron[k];
            kron[k] = a;
        }
        if ( ( k = ( perm[i] += 2 ) ) < number ) {
            a = kron[i];
            kron[i] = kron[k];
            kron[k] = a;
            sgn = - sgn;
            for ( k = i + 2; k < number; k += 2 ) perm[k] = k;
            return(sgn);
        }
    }
    return(0);
}

/*
        #] EpfGen :
        #[ Trick :              WORD Trick(in,t)

        This routine implements the identity:
        g_(j,mu)*g_(j,nu)*g_(j,ro)=e_(mu,nu,ro,si)*g5_(j)*g_(j,si)
        +d_(mu,nu)*g_(j,ro)-d_(mu,ro)*g_(j,nu)+d_(nu,ro)*g_(j,mu)
        which is for 4 dimensions only!

        Note that z->gamm = 1 if there is no g5 present.

*/

WORD Trick(WORD *in, TRACES *t)
{
    WORD n, n1;
    n = t->stap;
    n1 = t->step1;
    switch ( t->eers[n] ) {
        case 0: {
            WORD *p;
            p = t->pepf + t->mepf[n];
            *p++ = *in++;
            *p++ = *in++;
            *p++ = *in;
            *p   = ++(t->mdum);
            (t->mepf[n1]) += 4;
            *in  = t->mdum;
            t->gamm = - t->gamm;
            t->eers[n] = 5;
            break;
        }
        case 4: {
            WORD *p;
            p = t->pdel + t->mdel[n];
            (t->mepf[n1]) -= 4;
            (t->mdum)--;
            *p++ = *in++;
            *p   = *in++;
            *in  = *(t->pepf + t->mepf[n] + 2);
            (t->mdel[n1]) += 2;
            t->gamm = - t->gamm;
            break;
        }
        case 3: {
            t->sign1 = - t->sign1;
            *(t->pdel + t->mdel[n] + 1) = in[2];
            in[2] = in[1];
            break;
        }
        case 2: {
            t->sign1 = - t->sign1;
            in[2] = in[0];
            *(t->pdel + t->mdel[n]) = in[1];
            break;
        }
        case 1: {
            in[2] = *(t->pdel + t->mdel[n] + 1);
            (t->mdel[n1]) -= 2;
            break;
        }
        default: {
            return(0);
        }
    }
    return(--(t->eers[n]));
}

/*
        #] Trick :
        #[ Trace4no :           WORD Trace4no(number,kron,t)

```

```
        Takes the trace of a string of gamma matrices in 4 dimensions.
        There is no test for indices or vectors that are the same.
        The four dimensions refer to the contraction in the algebra:
        g_(i,a,b,c) = e_(a,b,c,d)*g_(i,5_,d) + g_(i,a)*d_(b,c)
                    - g_(i,b)*d_(a,c) + g_(i,c)*d_(a,b)
        This simplifies life very much and leads to shorter expressions
        than in the n dimensional case.

        Parameters:
        number: the number of vectors/indices in inlist.
        inlist: the indices/vectors in the string.
        kron:   the output delta's.
        gamma5: the potential gamma5 in front.
        t:      the struct for scratch manipulations.
        stack:  the space to put all scratch arrays in.

        The return value is zero if there are no more terms, 1 if a
        term was generated with a positive sign and -1 if a term was
        generated with a negative sign.
        The value of one is increased to two if the first 4 values
        in kron should be interpreted as a Levi-Civita tensor.

        Note that kron should have more places reserved than the number
        of indices in inlist, because it will contain dummy indices
        temporarily. In principle there can be 'number*1/4' extra dummies.

*/

WORD Trace4no(WORD number, WORD *kron, TRACES *t)
{
    WORD i;
    WORD *p, *m;
    WORD retval, *stop, oldsign;
    if ( !t->sgn ) {        /* Startup */
        if ( ( number < 0 ) || ( number & 1 ) ) return(0);
        if ( number <= 2 ) {
            if ( t->gamma5 == GAMMA5 ) return(0);
            if ( number == 2 ) {
                *kron++ = *t->inlist;
                *kron++ = t->inlist[1];
            }
            return(1);
        }
        t->sgn = 1;
        {
            WORD nhalf = number » 1;
            WORD ndouble = number « 1;
            p = t->eers;
            t->eers = p;    p += nhalf;
            t->mepf = p;    p += nhalf;
            t->mdel = p;    p += nhalf;
            t->pdel = p;    p += number + nhalf;
            t->pepf = p;    p += ndouble;
            t->e4 = p;      p += number;
            t->e3 = p;      p += ndouble;
            t->nt3 = p;     p += nhalf;
            t->nt4 = p;     p += nhalf;
            t->j3 = p;      p += ndouble;
            t->j4 = p;
        }
        t->mepf[0] = 0;
        t->mdel[0] = 0;
        t->mdum = AM.mTraceDum;
        t->kstep = -2;
        t->step1 = 0;
        t->sign1 = 1;
        t->lc3 = -1;
        t->lc4 = -1;
        t->gamm = 1;

        do {
            t->stap = (t->step1)++;
            t->kstep += 2;
            t->eers[t->stap] = 0;
            t->mepf[t->step1] = t->mepf[t->stap];
            t->mdel[t->step1] = t->mdel[t->stap];
CallTrick:  while ( !Trick(t->inlist+t->kstep,t) ) {
                t->kstep -= 2;
                t->step1 = (t->stap)--;
                if ( t->stap < 0 ) {
                    return(0);
                }
            }
        } while ( t->kstep < (number-4) );
/*
        Take now the trace of the leftover matrices.
        If gamma5 causes the term to vanish there will be a
```

```
        renewed call to Trick for its next term.
*/
        t->sign2 = t->sign1;
        if ( ( t->gamma5 == GAMMA7 ) && ( t->gamm == -1 ) ) {
            t->sign2 = - t->sign2;
        }
        else if ( ( t->gamma5 == GAMMA5 ) && ( t->gamm == 1 ) ) {
            goto CallTrick;
        }
        else if ( ( t->gamma5 == GAMMA1 ) && ( t->gamm == -1 ) ) {
            goto CallTrick;
        }
        p = t->pdel + t->mdel[t->step1];
        *p++ = t->inlist[t->kstep+2];
        *p++ = t->inlist[t->kstep+3];
/*
        Now the trace has been expressed in terms of Levi-Civita tensors
        and Kronecker delta's.
        The Levi-Civita tensors are in t->pepf
        and there are t->mepf[step1] elements in this array.
        The Kronecker delta's are in t->pdel
        and there are t->mdel[step1] elements in this array.

        Next we rake the Levi-Civita tensors together such that there
        is an optimal use of the contractions.
*/
        {
            WORD ae;
            ae = t->mepf[t->step1];
            t->ad = t->mdel[t->step1]+2;
            t->a4 = 0;
            t->a3 = 0;
            while ( ( ae -= 4 ) >= 0 ) {
                if ( t->pepf[ae] > AM.mTraceDum && t->pepf[ae] <= t->mdum ) {
                    p = t->e3 + t->a3;
                    m = t->pepf + ae;
                    for ( i = 0; i < 3; i++ ) {
                        p[3] = m[3-i];
                        *p++ = m[i-4];
                    }
                    t->a3 += 6;
                    ae -= 4;
                }
                else {
                    p = t->e4 + t->a4;
                    m = t->pepf + ae;
                    for ( i = 0; i < 4; i++ ) *p++ = *m++;
                    t->a4 += 4;
                }
            }
        }
/*
        Now e3 contains pairs of LeviCivita tensors that have
        three indices each and a3 is the total number of indices.
        e4 contains individual tensors with 4 indices.
        Some indices may be contracted with Kronecker delta's.

        Contract the e3 tensors first.
*/

        while ( t->a3 > 0 ) {
            t->nt3[++(t->lc3)] = 0;
            while ( ( t->nt3[t->lc3] = EpfGen(3,t->e3+t->a3-6,
            t->pdel+t->ad,t->j3+6*t->lc3,oldsign = t->nt3[t->lc3]) ) == 0 ) {
                if ( oldsign < 0 ) t->sign2 = - t->sign2;
                (t->lc3)--;
NextE3:         if ( t->lc3 < 0 ) goto CallTrick;
                t->ad -= 6;
                t->a3 += 6;
            }
            if ( oldsign ) {
                if ( oldsign != t->nt3[t->lc3] ) t->sign2 = - t->sign2;
            }
            else if ( t->nt3[t->lc3] < 0 ) t->sign2 = - t->sign2;
            t->a3 -= 6;
            t->ad += 6;
        }
/*
        Contract the e4 tensors.
*/
        while ( t->a4 > 4 ) {
            t->nt4[++(t->lc4)] = 0;
            while ( ( t->nt4[t->lc4] = EpfGen(4,t->e4+t->a4-8,
            t->pdel+t->ad,t->j4+8*t->lc4,oldsign = t->nt4[t->lc4]) ) == 0 ) {
                if ( oldsign < 0 ) t->sign2 = - t->sign2;
                (t->lc4)--;
NextE4:         if ( t->lc4 < 0 ) goto NextE3;
```

```
                t->ad -= 8;
                t->a4 += 8;
            }
            if ( oldsign ) {
                if ( oldsign != t->nt4[t->lc4] ) t->sign2 = - t->sign2;
            }
            else if ( t->nt4[t->lc4] < 0 ) t->sign2 = - t->sign2;
            t->a4 -= 8;
            t->ad += 8;
        }
/*
        Finally the extra dummy indices can be eliminated.
        Note that there are currently t->ad words in t->pdel forming
        t->ad / 2 Kronecker delta's. We are however not allowed to
        alter anything in these arrays, so the results should be
        copied to kron.
*/
        m = kron;
        if ( t->a4 == 4 ) {
            p = t->e4;
            *m++ = *p++; *m++ = *p++; *m++ = *p++; *m++ = *p++;
            retval = 2;
        }
        else retval = 1;
        if ( t->sign2 < 0 ) retval = - retval;
        p = t->pdel;
        for ( i = 0; i < t->ad; i++ ) *m++ = *p++;
        p = kron;
        if ( t->a4 == 4 ) {
/*
            Test for dummies in the last position of the e_.
*/
            stop = p + t->ad + 4;
            p += 3;
            while ( *p >= AM.mTraceDum && *p <= t->mdum ) {
                m = p + 1;
                do {
                    if ( *m == *p ) {
                        *p = m[1];
                        *m = *--stop;
                        m[1] = *--stop;
                        break;
                    }
                    else if ( m[1] == *p ) {
                        *p = *m;
                        *m = *--stop;
                        m[1] = *--stop;
                        break;
                    }
                    else m += 2;
                } while ( m < stop );
            }
            p++;
        }
        else stop = p + t->ad;
        while ( p < (stop-2) ) {
            while ( *p >= AM.mTraceDum && *p <= t->mdum ) {
                m = p + 2;
                do {
                    if ( *m == *p ) {
                        *p = m[1];
                        *m = *--stop;
                        m[1] = *--stop;
                        break;
                    }
                    else if ( m[1] == *p ) {
                        *p = *m;
                        *m = *--stop;
                        m[1] = *--stop;
                        break;
                    }
                    else m += 2;
                } while ( m < stop );
            }
            p++;
            while ( *p >= AM.mTraceDum && *p <= t->mdum ) {
                m = p + 1;
                do {
                    if ( *m == *p ) {
                        *p = m[1];
                        *m = *--stop;
                        m[1] = *--stop;
                        break;
                    }
                    else if ( m[1] == *p ) {
                        *p = *m;
                        *m = *--stop;
```

```
                        m[1] = *--stop;
                        break;
                    }
                    else m += 2;
                } while ( m < stop );
            }
            p++;
        }
        return(retval);
    }
    if ( number <= 2 ) return(0);
    else { goto NextE4; }
}

/*
        #] Trace4no :
        #[ Trace4 :             WORD Trace4(term,params,num,level)

        Generates traces of the string of gamma matrices in 'instring'.

        The difference with the routine tracen ( for n dimensions )
        lies in the treatment of gamma 5 and the specific form
        of the Chisholm identities. The identities used here are
        g(mu)*g(a1)*...*g(an)*g(mu)=
        n=odd:  -2*g(an)*...*g(a1)    ( reversed order )
        n=even: 2*g(an)*g(a1)*...*g(a(n-1))
            +2*g(a(n-1))*...*g(a1)*g(an)
        There is a special case for n=2 : 4*d(a1,a2)*gi

        The main difference with the old fortran version lies in
        the recursion that is used here. That cleans up the variables
        very much.

        The contents of the AT.TMout array are:
        length,type,gamma5,gamma's

        The space for the vectors in t is at most 14 * number.

        The condition params[5] == 0 corresponds to finding gamma6*gamma7
        during the pick up of the matrices.

*/

WORD Trace4(PHEAD WORD *term, WORD *params, WORD num, WORD level)
{
    GETBIDENTITY
    TRACES *t;
    WORD *p, *m, number, i;
    WORD *OldW;
    WORD j, minimum, minimum2, *min, *stopper;
    OldW = AT.WorkPointer;
    if ( AN.numtracesctack >= AN.intracestack ) {
        number = AN.intracestack + 2;
        t = (TRACES *)Malloc1(number*sizeof(TRACES),"TRACES-struct");
        if ( AN.tracestack ) {
            for ( i = 0; i < AN.intracestack; i++ ) { t[i] = AN.tracestack[i]; }
            M_free(AN.tracestack,"TRACES-struct");
        }
        AN.tracestack = t;
        AN.intracestack = number;
    }

    number = *params - 6;
    if ( number < 0 || ( number & 1 ) || !params[5] ) return(0);

    t = AN.tracestack + AN.numtracesctack;
    AN.numtracesctack++;

    t->finalstep = ( params[2] & 16 ) ? 1 : 0;
    t->gamma5 = params[3];
    if ( t->finalstep && t->gamma5 != GAMMA1 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Gamma5 not allowed in this option of the trace command");
        MUNLOCK(ErrorMessageLock);
        AN.numtracesctack--;
        SETERROR(-1)
    }
    t->inlist = AT.WorkPointer;
    t->accup = t->accu = t->inlist + number;
    t->perm = t->accu + (number«1);
    t->eers = t->perm + number;
    if ( ( AT.WorkPointer += 19 * number ) >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
```

```
    t->num = num;
    t->level = level;
    p = t->inlist;
    m = params+6;
    for ( i = 0; i < number; i++ ) *p++ = *m++;
    t->termp = term;
    t->factor = params[4];
    t->allsign = params[5];
    if ( number >= 10 || ( t->gamma5 != GAMMA1 && number > 4 ) ) {
/*
        The next code should `normal order' the string
        We need the lexicographic smallest string, taking also the
        reverse strings into account.
*/
        minimum = 0; min = t->inlist;
        stopper = min + number;
        for ( i = 1; i < number; i++ ) {
            p = min;
            m = t->inlist + i;
            for ( j = 0; j < number; j++ ) {
                if ( *p < *m ) break;
                if ( *p > *m ) {
                    min = t->inlist+i;
                    minimum = i;
                    break;
                }
                p++; m++;
                if ( m >= stopper ) m = t->inlist;
                if ( p >= stopper ) p = t->inlist;
            }
        }
        p = min;
        min = m = AT.WorkPointer;
        i = number;
        while ( --i >= 0 ) {
            *m++ = *p++;
            if ( p >= stopper ) p = t->inlist;
        }
        p = t->inlist;
        m = min;
        i = number;
        while ( --i >= 0 ) *p++ = *m++;
        p = t->inlist;
        m = stopper;
        while ( p < m ) {   /* reverse string */
            i = *p; *p++ = *--m; *m = i;
        }
        minimum2 = 0;
        for ( i = 0; i < number; i++ ) {
            p = min;
            m = t->inlist + i;
            for ( j = 0; j < number; j++ ) {
                if ( *p < *m ) break;
                if ( *p > *m ) {
                    m = t->inlist + i;
                    p = min;
                    j = number;
                    while ( --j >= 0 ) {
                        *p++ = *m++;
                        if ( m >= stopper ) m = t->inlist;
                    }
                    minimum2 = i;
                    break;
                }
                p++; m++;
                if ( m >= stopper ) m = t->inlist;
            }
        }
        minimum ^= minimum2;
        if ( ( minimum & 1 ) != 0 ) {
            if ( t->gamma5 == GAMMA5 ) t->allsign = - t->allsign;
            else if ( t->gamma5 != GAMMA1 )
                t->gamma5 = GAMMA6 + GAMMA7 - t->gamma5;
        }
        p = min; m = t->inlist; i = number;
        while ( --i >= 0 ) *m++ = *p++;
/*
        Now the trace is in normal order
*/
    }
    number = Trace4Gen(BHEAD t,number);
    AT.WorkPointer = OldW;
    AN.numtracesctack--;
    return(number);
}

/*
```

```
        #] Trace4 :
        #[ Trace4Gen :          WORD Trace4Gen(t,number)

        The recursive breakdown of a trace in 4 dimensions.
        We test first whether the trace has zero or two gamma's left.
        This case can be done quickly.
        Next we test whether we can eliminate adjacent objects that are
        the same.
        Then we test for Chisholm identities (I). First for identities
        with an odd number of gamma matrices (II), then for those with an
        even number of matrices (III). The special thing here is the demand
        that the contraction be between indices with 4 dimensions only.
        Then there is a scan for objects that are the same, not regarding
        their type (IV). This is exactly the same as in n dimensions.
        Finally we have a string left in which all objects are different (V).
        This case is treated by the routine Trace4no (no stands for no
        objects are the same).

        In case I we have one d_ of which the result of the contraction
        has not yet been fixed.
        Case II gives just a reordering of the matrices and a factor -2.
        Case III gives two terms: one for the anti commutation, such that
        the number of intermediate matrices becomes odd and the other
        from the Chisholm rule for an odd number of matrices. Both have
        a factor 2.
        Case IV gives m+1 terms when m is the number of matrices inbetween.
        We take the shortest path. The sign alternates and all terms have
        a factor two, except for the last one.

*/

WORD Trace4Gen(PHEAD TRACES *t, WORD number)
{
    GETBIDENTITY
    WORD *termout, *stop;
    WORD *p, *m, oldval;
    WORD *pold, *mold, diff, *oldstring, cp;
/*
            #[ Special cases :
*/
    if ( number <= 2 ) {    /* Special cases */
        if ( t->gamma5 == GAMMA5 ) return(0);
        termout = AT.WorkPointer;
        p = t->termp;
        stop = p + *p;
        m = termout;
        p++;
        if ( p < stop ) do {
            if ( *p == SUBEXPRESSION && p[2] == t->num ) {
                oldstring = p;
                p = t->termp;
                do { *m++ = *p++; } while ( p < oldstring );
                p += p[1];
                *m++ = AC.lUniTrace[0];
                *m++ = AC.lUniTrace[1];
                *m++ = AC.lUniTrace[2];
                *m++ = AC.lUniTrace[3];
                if ( number == 2 || t->accup > t->accu ) {
                    oldstring = m;
                    *m++ = DELTA;
                    *m++ = 4;
                    if ( number == 2 ) {
                        *m++ = t->inlist[0];
                        *m++ = t->inlist[1];
                    }
                    if ( t->accup > t->accu ) {
                        pold = p;
                        p = t->accu;
                        while ( p < t->accup ) *m++ = *p++;
                        oldstring[1] = WORDDIF(m,oldstring);
                        p = pold;
                    }
                }
                if ( t->factor ) {
                    *m++ = SNUMBER;
                    *m++ = 4;
                    *m++ = 2;
                    *m++ = t->factor;
                }
                do { *m++ = *p++; } while ( p < stop );
                *termout = WORDDIF(m,termout);
                if ( t->allsign < 0 ) m[-1] = -m[-1];
                if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
                    MLOCK(ErrorMessageLock);
                    MesWork();
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
```

```
                }
                *AN.RepPoint = 1;
                AR.expchanged = 1;
                if ( *termout ) {
                    *AN.RepPoint = 1;
                    AR.expchanged = 1;
                    if ( Generator(BHEAD termout,t->level) ) goto TracCall;
                }
                AT.WorkPointer= termout;
                return(0);
            }
            p += p[1];
        } while ( p < stop );
        return(0);
    }
/*
            #] Special cases :
            #[ Adjacent objects :
*/
    p = t->inlist;
    stop = p + number - 1;
    if ( *p == *stop ) {        /* First and last of string */
        oldval = *p;
        *(t->accup)++ = *p;
        *(t->accup)++ = *p;
        m = p+1;
        while ( m < stop ) *p++ = *m++;
        if ( t->gamma5 != GAMMA1 ) {
            if ( t->gamma5 == GAMMA5 ) t->allsign = - t->allsign;
            else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
            else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
        }
        if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
        t = AN.tracestack + AN.numtracesctack - 1;
        if ( t->gamma5 != GAMMA1 ) {
            if ( t->gamma5 == GAMMA5 ) t->allsign = - t->allsign;
            else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
            else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
        }
        while ( p > t->inlist ) *--m = *--p;
        *p = *stop = oldval;
        t->accup -= 2;
        return(0);
    }
    do {
        if ( *p == p[1] ) {     /* Adjacent in string */
            oldval = *p;
            pold = p;
            m = p+2;
            *(t->accup)++ = *p;
            *(t->accup)++ = *p;
            while ( m <= stop ) *p++ = *m++;
            if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
            t = AN.tracestack + AN.numtracesctack - 1;
            while ( p > pold ) *--m = *--p;
            *p++ = oldval;
            *p++ = oldval;
            t->accup -= 2;
            return(0);
        }
        p++;
    } while ( p < stop );
/*
            #] Adjacent objects :
            #[ Odd Contraction :
*/
    p = t->inlist;
    stop = p + number;
    do {
        if ( *p >= AM.OffsetIndex && (
        ( *p < WILDOFFSET + AM.OffsetIndex &&
         indices[*p-AM.OffsetIndex].dimension == 4 )
        || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim == 4 ) ) ) {
            m = p+2;
            while ( m < stop ) {
                if ( *p == *m ) {
                    pold = p;
                    mold = m;
                    oldval = *p;
                    (t->factor)++;
                    t->allsign = - t->allsign;
                    *p++ = *--m;
                    m--;
                    while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
                    p = mold - 1;
                    m = mold + 1;
                    while ( m < stop ) *p++ = *m++;
```

```
                    if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
                    t = AN.tracestack + AN.numtracesctack - 1;
                    m--;
                    while ( m > mold ) *m-- = *--p;
                    p = pold;
                    *m-- = oldval;
                    *m-- = *p;
                    *p++ = oldval;
                    while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
                    t->allsign = - t->allsign;
                    (t->factor)--;
                    return(0);
                }
                m += 2;
            }
        }
        p++;
    } while ( p < stop );
/*
            #] Odd Contraction :
            #[ Even Contraction :
        First the case with two matrices inbetween.
*/
    p = t->inlist;
    stop = p + number;
    do {
        if ( *p >= AM.OffsetIndex && (
        ( *p < WILDOFFSET + AM.OffsetIndex &&
         indices[*p-AM.OffsetIndex].dimension == 4 )
        || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim == 4 ) ) ) {
            m = p+3;
            if ( m >= stop ) m -= number;
            if ( *p == *m ) {
                WORD oldfactor, old5;
                oldstring = AT.WorkPointer;
                AT.WorkPointer += number;
                oldfactor = t->allsign;
                old5 = t->gamma5;
                if ( m < p ) cp = (WORDDIF(m,t->inlist) + 1 ) & 1;
                else cp = 0;
                if ( cp && ( t->gamma5 != GAMMA1 ) ) {
                    if ( t->gamma5 == GAMMA5 ) t->allsign = -t->allsign;
                    else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
                    else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
                }
                mold = m;
                p = oldstring;
                m = t->inlist;
                while ( m < stop ) *p++ = *m++;
/*
        Rotate the string
*/
                m = mold + 1;
                p = t->inlist;
                while ( m < stop ) *p++ = *m++;
                m = oldstring;
                if ( !cp && ((WORDDIF(stop,p))&1) != 0 && ( t->gamma5 != GAMMA1 ) ) {
                    if ( t->gamma5 == GAMMA5 ) t->allsign = -t->allsign;
                    else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
                    else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
                }
                while ( p < stop ) *p++ = *m++;
                t->factor += 2;
                m = p - 3;
                p = t->inlist;
                oldval = number - 4;
                while ( oldval > 0 ) {
                    if ( *p >= AM.OffsetIndex && (
                    ( *p < WILDOFFSET + AM.OffsetIndex &&
                     indices[*p-AM.OffsetIndex].dimension )
                    || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim ) ) ) {
                        if ( *p == *m ) {
                            *p = m[1];
                            break;
                        }
                        else if ( *p == m[1] ) {
                            *p = *m;
                            break;
                        }
                    }
                    p++; oldval--;
                }
                if ( oldval <= 0 ) {
                    *(t->accup)++ = *m++;
                    *(t->accup)++ = *m++;
                }
                if ( Trace4Gen(BHEAD t,number-4) ) goto TracCall;
```

```
                t = AN.tracestack + AN.numtracesctack - 1;
                t->factor -= 2;
                if ( oldval <= 0 ) t->accup -= 2;
                t->gamma5 = old5;
                t->allsign = oldfactor;
                AT.WorkPointer = p = oldstring;
                m = t->inlist;
                while ( m < stop ) *m++ = *p++;
                return(0);
            }
        }
        p++;
    } while ( p < stop );
/*
        The case with at least 4 matrices inbetween.
*/
    p = t->inlist;
    stop = p + number;
    do {
        if ( *p >= AM.OffsetIndex && (
        ( *p < WILDOFFSET + AM.OffsetIndex &&
         indices[*p-AM.OffsetIndex].dimension == 4 )
        || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim == 4 ) ) ) {
            m = p+5;
            while ( m < stop ) {
                if ( *p == *m ) {
                    WORD *pex, *mex;
                    pold = p;
                    mold = m;
                    oldval = *p;
/*
            g_(1,mu)*g_(1,a1)*...*g_(1,aj)*g_(1,an)*g_(1,mu) ->
            first:
            2*g_(1,an)*g_(1,a1)*...*g_(1,aj)
*/
                    (t->factor)++;
/*
                    The variable hulp seems unnecessary
                    *p = hulp = m[-1];
*/
                    *p = m[-1];
                    p = m - 1;
                    m++;
                    while ( m < stop ) *p++ = *m++;
                    if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
                    t = AN.tracestack + AN.numtracesctack - 1;
                    pex = p; mex = m;
                    p = pold;
                    m = mold - 2;
                    while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
/*
            and then:
            2*g_(1,aj)*...*g_(1,a1)*g_(1,an)
*/
                    if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
                    t = AN.tracestack + AN.numtracesctack - 1;
                    p = pold;
                    m = mold - 2;
                    while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
                    m = mex;
                    p = pex;
                    m--;
                    while ( m > mold ) *m-- = *--p;
                    m = mold;
                    *m-- = oldval;
                    p = pold;
                    *m = *p;
                    *p = oldval;
                    (t->factor)--;
                    return(0);
                }
                m += 2;
            }
        }
        p++;
    } while ( p < stop );
/*
            #] Even Contraction :
            #[ Same Objects :
*/
    p = t->inlist;
    stop = p + number - 1;
    diff = 2;
    do {
        p = t->inlist;
        while ( p <= stop ) {
            m = p + diff;
```

```
            if ( m > stop ) m -= number;
            if ( *p == *m ) {
                WORD oldfactor, c, old5;
                oldfactor = t->allsign;
                old5 = t->gamma5;
                cp = (WORDDIF(m,t->inlist)) & 1;
                if ( !cp && ( t->gamma5 != GAMMA1 ) ) {
                    if ( t->gamma5 == GAMMA5 ) t->allsign = -t->allsign;
                    else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
                    else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
                }
                oldstring = AT.WorkPointer;
                AT.WorkPointer += number;
                mold = m;
                oldval = *p;
                p = oldstring;
                m = t->inlist;
                while ( m <= stop ) *p++ = *m++;
/*
        Rotate the string
*/
                m = mold + 1;
                p = t->inlist;
                while ( m <= stop ) *p++ = *m++;
                m = oldstring;
                while ( p <= stop ) *p++ = *m++;
                (t->factor)++;
                p -= diff + 1;
                m = stop;
                *(t->accup) = oldval;
                t->accup += 2;
                m--;
                while ( m > p ) {
                    c = t->accup[-1];
                    t->accup[-1] = *m;
                    *m = c;
                    if ( Trace4Gen(BHEAD t,number-2) ) goto Trac4Call;
                    t = AN.tracestack + AN.numtracesctack - 1;
                    m--;
                    t->allsign = - t->allsign;
                }
                c = t->accup[-1];
                t->accup[-1] = *m;
                *m = c;
                (t->factor)--;
                if ( Trace4Gen(BHEAD t,number-2) ) goto Trac4Call;
                t = AN.tracestack + AN.numtracesctack - 1;
                t->accup -= 2;
                t->allsign = oldfactor;
                AT.WorkPointer = p = oldstring;
                m = t->inlist;
                while ( m <= stop ) *m++ = *p++;
                t->gamma5 = old5;
                return(0);
            }
            p++;
        }
    } while ( ++diff <= (number»1) );
/*
            #] Same Objects :
            #[ All Different :

        Here we have a string with all different objects.

*/
    t->sgn = 0;
    termout = AT.WorkPointer;
    for(;;) {
        if ( t->finalstep == 0 ) diff = Trace4no(number,t->accup,t);
        else                     diff = TraceNno(number,t->accup,t);
/*  while ( ( diff = Trace4no(number,t->accup,t) ) != 0 ) */
        if ( diff == 0 ) break;
        p = t->termp;
        stop = p + *p;
        m = termout;
        p++;
        if ( p < stop ) do {
            if ( *p == SUBEXPRESSION && p[2] == t->num ) {
                oldstring = p;
                p = t->termp;
                do { *m++ = *p++; } while ( p < oldstring );
                p += p[1];
                pold = p;
                *m++ = AC.lUniTrace[0];
                *m++ = AC.lUniTrace[1];
                *m++ = AC.lUniTrace[2];
                *m++ = AC.lUniTrace[3];
```

```
                *m++ = SNUMBER;
                *m++ = 4;
                *m++ = 2;
                *m++ = t->factor;
                p = t->accup;
                oldval = number;
                if ( diff == 2 || diff == -2 ) {
                    *m++ = LEVICIVITA;
                    *m++ = 4+FUNHEAD;
                    *m++ = DIRTYFLAG;
                    FILLFUN3(m)
                    *m++ = *p++; *m++ = *p++; *m++ = *p++; *m++ = *p++;
                    oldval -= 4;
                }
                if ( oldval > 0 || t->accup > t->accu ) {
                    oldstring = m;
                    *m++ = DELTA;
                    *m++ = oldval + 2;
                    if ( oldval > 0 ) NCOPY(m,p,oldval);
                    if ( t->accup > t->accu ) {
                        p = t->accu;
                        while ( p < t->accup ) *m++ = *p++;
                        oldstring[1] = WORDDIF(m,oldstring);
                    }
                }
                p = pold;
                do { *m++ = *p++; } while ( p < stop );
                *termout = WORDDIF(m,termout);
                if ( ( diff ^ t->allsign ) < 0 ) m[-1] = - m[-1];
                if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
                    MLOCK(ErrorMessageLock);
                    MesWork();
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                if ( *termout ) {
                    *AN.RepPoint = 1;
                    AR.expchanged = 1;
                    if ( Generator(BHEAD termout,t->level) ) {
                        AT.WorkPointer = termout;
                        goto TracCall;
                    }
                    t = AN.tracestack + AN.numtracesctack - 1;
                }
                break;
            }
            p += p[1];
        } while ( p < stop );
    }
    AT.WorkPointer = termout;
    return(0);

/*
            #] All Different :
*/
Trac4Call:
    AT.WorkPointer = oldstring;
TracCall:
    if ( AM.tracebackflag ) {
        MLOCK(ErrorMessageLock);
        MesCall("Trace4Gen");
        MUNLOCK(ErrorMessageLock);
    }
    return(-1);
}

/*
        #] Trace4Gen :
        #[ TraceNno :           WORD TraceNno(number,kron,t)

        Routine takes the trace in N dimensions of a string
        of gamma matrices. It is assumed that there are no
        contractions and no vectors that are the same. For
        the treatment of those cases there are special routines,
        that call this routine as a final stage.
        The calling routine must reserve 'number' WORDs for perm
        and kron each.
        kron and perm may not be altered during the generation!

*/

WORD TraceNno(WORD number, WORD *kron, TRACES *t)
{
    WORD i, j, a, *p;
    if ( !t->sgn ) {
        if ( !number || ( number & 1 ) ) return(0);
        p = t->inlist;
```

```
        for ( i = 0; i < number; i++ ) {
            t->perm[i] = i;
            *kron++ = *p++;
        }
        t->sgn = 1;
        return(1);
    }
    else {
        i = number - 3;
        while ( i > 0 ) {
            a = kron[i];
            p = t->perm + i;
            for ( j = i + 1; j <= *p; j++ ) kron[j-1] = kron[j];
            kron[(*p)++] = a;
            if ( *p < number ) {
                a = kron[*p];
                j = *p;
                while ( j >= (i+1) ) { kron[j] = kron[j-1]; j--; }
                kron[i] = a;
                number -= 2;
                for ( j = i+2; j < number; j += 2 ) t->perm[j] = j;
                t->sgn = - t->sgn;
                return(t->sgn);
            }
            i -= 2;
        }
    }
    return(0);
}

/*
        #] TraceNno :
        #[ TraceN :             WORD TraceN(term,params,num,level)
*/

WORD TraceN(PHEAD WORD *term, WORD *params, WORD num, WORD level)
{
    GETBIDENTITY
    TRACES *t;
    WORD *p, *m, number, i;
    WORD *OldW;
    if ( params[3] != GAMMA1 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Gamma5 not allowed in n-trace");
        MUNLOCK(ErrorMessageLock);
        SETERROR(-1)
    }
    OldW = AT.WorkPointer;
    if ( AN.numtracesctack >= AN.intracestack ) {
        number = AN.intracestack + 2;
        t = (TRACES *)Malloc1(number*sizeof(TRACES),"TRACES-struct");
        if ( AN.tracestack ) {
            for ( i = 0; i < AN.intracestack; i++ ) { t[i] = AN.tracestack[i]; }
            M_free(AN.tracestack,"TRACES-struct");
        }
        AN.tracestack = t;
        AN.intracestack = number;
    }
    number = *params - 6;
    if ( number < 0 || ( number & 1 ) || !params[5] ) return(0);

    t = AN.tracestack + AN.numtracesctack;
    AN.numtracesctack++;

    t->inlist = AT.WorkPointer;
    t->accup = t->accu = t->inlist + number;
    t->perm = t->accu + number;
    if ( ( AT.WorkPointer += 3 * number ) >= AT.WorkTop ) {
        AN.numtracesctack--;
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    t->num = num;
    t->level = level;
    p = t->inlist;
    m = params+6;
    for ( i = 0; i < number; i++ ) *p++ = *m++;
    t->termp = term;
    t->factor = params[4];
    t->allsign = params[5];
    number = TraceNgen(BHEAD t,number);
    AT.WorkPointer = OldW;
    AN.numtracesctack--;
    return(number);
}
```

```

/*
        #] TraceN :
        #[ TraceNgen :          WORD TraceNgen(t,number)

        This routine is a simplified version of Trace4Gen. We know here
        only three cases: Adjacent objects, same objects and all different.
        The othere difference lies of course in the struct which is now
        not of type TRACES, but of type TRACES.

*/

WORD TraceNgen(PHEAD TRACES *t, WORD number)
{
    GETBIDENTITY
    WORD *termout, *stop;
    WORD *p, *m, oldval;
    WORD *pold, *mold, diff, *oldstring;
/*
            #[ Special cases :
*/
    if ( number <= 2 ) {    /* Special cases */
        termout = AT.WorkPointer;
        p = t->termp;
        stop = p + *p;
        m = termout;
        p++;
        if ( p < stop ) do {
            if ( *p == SUBEXPRESSION && p[2] == t->num ) {
                oldstring = p;
                p = t->termp;
                do { *m++ = *p++; } while ( p < oldstring );
                p += p[1];
                *m++ = AC.lUniTrace[0];
                *m++ = AC.lUniTrace[1];
                *m++ = AC.lUniTrace[2];
                *m++ = AC.lUniTrace[3];
                if ( number == 2 || t->accup > t->accu ) {
                    oldstring = m;
                    *m++ = DELTA;
                    *m++ = 4;
                    if ( number == 2 ) {
                        *m++ = t->inlist[0];
                        *m++ = t->inlist[1];
                    }
                    if ( t->accup > t->accu ) {
                        pold = p;
                        p = t->accu;
                        while ( p < t->accup ) *m++ = *p++;
                        oldstring[1] = WORDDIF(m,oldstring);
                        p = pold;
                    }
                }
                if ( t->factor ) {
                    *m++ = SNUMBER;
                    *m++ = 4;
                    *m++ = 2;
                    *m++ = t->factor;
                }
                do { *m++ = *p++; } while ( p < stop );
                *termout = WORDDIF(m,termout);
                if ( t->allsign < 0 ) m[-1] = -m[-1];
                if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
                    MLOCK(ErrorMessageLock);
                    MesWork();
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                if ( *termout ) {
                    *AN.RepPoint = 1;
                    AR.expchanged = 1;
                    if ( Generator(BHEAD termout,t->level) ) goto TracCall;
                }
                AT.WorkPointer= termout;
                return(0);
            }
            p += p[1];
        } while ( p < stop );
        return(0);
    }
/*
            #] Special cases :
            #[ Adjacent objects :
*/
    p = t->inlist;
    stop = p + number - 1;
    if ( *p == *stop ) {        /* First and last of string */
```

```
        oldval = *p;
        *(t->accup)++ = *p;
        *(t->accup)++ = *p;
        m = p+1;
        while ( m < stop ) *p++ = *m++;
        if ( TraceNgen(BHEAD t,number-2) ) goto TracCall;
        t = AN.tracestack + AN.numtracesctack - 1;
        while ( p > t->inlist ) *--m = *--p;
        *p = *stop = oldval;
        t->accup -= 2;
        return(0);
    }
    do {
        if ( *p == p[1] ) {     /* Adjacent in string */
            oldval = *p;
            pold = p;
            m = p+2;
            *(t->accup)++ = *p;
            *(t->accup)++ = *p;
            while ( m <= stop ) *p++ = *m++;
            if ( TraceNgen(BHEAD t,number-2) ) goto TracCall;
            t = AN.tracestack + AN.numtracesctack - 1;
            while ( p > pold ) *--m = *--p;
            *p++ = oldval;
            *p++ = oldval;
            t->accup -= 2;
            return(0);
        }
        p++;
    } while ( p < stop );
/*
            #] Adjacent objects :
            #[ Same Objects :
*/
    p = t->inlist;
    stop = p + number - 1;
    diff = 2;
    do {
        p = t->inlist;
        while ( p <= stop ) {
            m = p + diff;
            if ( m > stop ) m -= number;
            if ( *p == *m ) {
                WORD oldfactor, c;
                oldstring = AT.WorkPointer;
                AT.WorkPointer += number;
                mold = m;
                oldval = *p;
                p = oldstring;
                m = t->inlist;
                while ( m <= stop ) *p++ = *m++;
/*
        Rotate the string
*/
                {
                    m = mold + 1;
                    p = t->inlist;
                    while ( m <= stop ) *p++ = *m++;
                    m = oldstring;
                    while ( p <= stop ) *p++ = *m++;
                }
                oldfactor = t->allsign;
                (t->factor)++;
                p -= diff + 1;
                m = stop;
                if ( oldval >= ( AM.OffsetIndex + WILDOFFSET ) ||
                    ( oldval >= AM.OffsetIndex
                    && indices[oldval-AM.OffsetIndex].dimension ) ) {
                    m--;
/*
        We distinguish 4 cases:
        m-p=1   Use g_(1,mu,a,mu) = (2-d_(mu,mu))*g_(1,a)
        m-p=2   Use g_(1,mu,a,b,mu) = 4*d_(a,b)+(d_(mu,mu)-4)*g_(1,a,b)
        m-p=3   Use g_(1,mu,a,b,c,mu) = -2*g_(1,c,b,a)
                        -(d_(mu,mu)-4)*g_(1,a,b,c)
        m-p>3   Reduce down to m-p=3 with the old technique
*/
                    while ( m > (p+3) ) {
                        c = *p;
                        *p = *m;
                        *m = c;
                        if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                        t = AN.tracestack + AN.numtracesctack - 1;
                        m--;
                        t->allsign = - t->allsign;
                    }
                    switch ( WORDDIF(m,p) ) {
```

```
                        case 1:
                            c = *p;
                            *p = *m;
                            *m = c;
                            if ( oldval < ( AM.OffsetIndex + WILDOFFSET )
                            && indices[oldval-AM.OffsetIndex].nmin4
                            != -NMIN4SHIFT ) {
                                t->allsign = - t->allsign;
                                if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                                t = AN.tracestack + AN.numtracesctack - 1;
                                (t->factor)--;
                                *(t->accup)++ = SUMMEDIND;
                                *(t->accup)++ =
                                    indices[oldval-AM.OffsetIndex].nmin4;
                            }
                            else
                            {
                                if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                                t = AN.tracestack + AN.numtracesctack - 1;
                                t->allsign = - t->allsign;
                                (t->factor)--;
                                *(t->accup)++ = oldval;
                                *(t->accup)++ = oldval;
                            }
                            if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                            t = AN.tracestack + AN.numtracesctack - 1;
                            t->accup -= 2;
                            break;
                        case 2:
                            { WORD one, two;
                            one = *p = p[1];
                            two = p[1] = *m;
                            (t->factor)++;          /* 4 */
                            *(t->accup)++ = *p;     /* d_(a,b) */
                            *(t->accup)++ = *m;
                            if ( TraceNgen(BHEAD t,number-4) ) goto TracnCall;
                            t = AN.tracestack + AN.numtracesctack - 1;
                            *p = one; p[1] = two;
                            t->accup -= 2;
                            if ( oldval < ( AM.OffsetIndex + WILDOFFSET )
                            && indices[oldval-AM.OffsetIndex].nmin4
                            != -NMIN4SHIFT ) {
                                t->factor -= 2;
                                *(t->accup)++ = SUMMEDIND;
                                *(t->accup)++ =
                                    indices[oldval-AM.OffsetIndex].nmin4;
                            }
                            else {
                                t->allsign = - t->allsign;
                                if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                                t = AN.tracestack + AN.numtracesctack - 1;
                                t->allsign = - t->allsign;
                                t->factor -= 2;
                                *(t->accup)++ = oldval;
                                *(t->accup)++ = oldval;
                            }
                            if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                            t = AN.tracestack + AN.numtracesctack - 1;
                            t->accup -= 2;
                            }
                            break;
                        default:
                            c = *p;
                            *p = *m;
                            *m = c;
                            c = m[-1]; m[-1] = m[-2]; m[-2] = c;
                            t->allsign = - t->allsign;
                            if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                            t = AN.tracestack + AN.numtracesctack - 1;
                            m--;
                            c = *p;
                            *p = *m;
                            *m = c;
                            (t->factor)--;
                            if ( oldval < ( AM.OffsetIndex + WILDOFFSET )
                            && indices[oldval-AM.OffsetIndex].nmin4
                            != -NMIN4SHIFT ) {
                                *(t->accup)++ = SUMMEDIND;
                                *(t->accup)++ =
                                    indices[oldval-AM.OffsetIndex].nmin4;
                                if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                                t = AN.tracestack + AN.numtracesctack - 1;
                                t->accup -= 2;
                                t->allsign = - t->allsign;
                            }
                            else
                            {
```

```
                                *(t->accup)++ = oldval;
                                *(t->accup)++ = oldval;
                                if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                                t = AN.tracestack + AN.numtracesctack - 1;
                                t->accup -= 2;
                                t->allsign = - t->allsign;
                                t->factor += 2;
                                if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                                t = AN.tracestack + AN.numtracesctack - 1;
                                t->factor -= 2;
                            }
                            break;
                        }
                    }
                    else {
                        *(t->accup) = oldval;
                        t->accup += 2;
                        m--;
                        while ( m > p ) {
                            c = t->accup[-1];
                            t->accup[-1] = *m;
                            *m = c;
                            if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                            t = AN.tracestack + AN.numtracesctack - 1;
                            m--;
                            t->allsign = - t->allsign;
                        }
                        c = t->accup[-1];
                        t->accup[-1] = *m;
                        *m = c;
                        (t->factor)--;
                        if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
                        t = AN.tracestack + AN.numtracesctack - 1;
                        t->accup -= 2;
                    }
                    t->allsign = oldfactor;
                    p = oldstring;
                    m = t->inlist;
                    while ( m <= stop ) *m++ = *p++;
                    AT.WorkPointer = oldstring;
                    return(0);
                }
                p++;
            }
            diff++;
        } while ( diff <= (number»1) );
/*
                #] Same Objects :
                #[ All Different :

            Here we have a string with all different objects.

*/
        t->sgn = 0;
        termout = AT.WorkPointer;
        while ( ( diff = TraceNno(number,t->accup,t) ) != 0 ) {
            p = t->termp;
            stop = p + *p;
            m = termout;
            p++;
            if ( p < stop ) do {
                if ( *p == SUBEXPRESSION && p[2] == t->num ) {
                    oldstring = p;
                    p = t->termp;
                    do { *m++ = *p++; } while ( p < oldstring );
                    p += p[1];
                    pold = p;
                    *m++ = AC.lUniTrace[0];
                    *m++ = AC.lUniTrace[1];
                    *m++ = AC.lUniTrace[2];
                    *m++ = AC.lUniTrace[3];
                    *m++ = SNUMBER;
                    *m++ = 4;
                    *m++ = 2;
                    *m++ = t->factor;
                    p = t->accup;
                    oldval = number;
                    oldstring = m;
                    *m++ = DELTA;
                    *m++ = oldval + 2;
                    NCOPY(m,p,oldval);
                    if ( t->accup > t->accu ) {
                        p = t->accu;
                        while ( p < t->accup ) *m++ = *p++;
                        oldstring[1] = WORDDIF(m,oldstring);
                    }
                    p = pold;
```

```
                do { *m++ = *p++; } while ( p < stop );
                *termout = WORDDIF(m,termout);
                if ( ( diff ^ t->allsign ) < 0 ) m[-1] = - m[-1];
                if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
                    MLOCK(ErrorMessageLock);
                    MesWork();
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                if ( *termout ) {
                    *AN.RepPoint = 1;
                    AR.expchanged = 1;
                    if ( Generator(BHEAD termout,t->level) ) {
                        AT.WorkPointer = termout;
                        goto TracCall;
                    }
                    t = AN.tracestack + AN.numtracesctack - 1;
                }
                break;
            }
            p += p[1];
        } while ( p < stop );
    }
    AT.WorkPointer = termout;
    return(0);

/*
            #] All Different :
*/
TracnCall:
    AT.WorkPointer = oldstring;
TracCall:
    if ( AM.tracebackflag ) {
        MLOCK(ErrorMessageLock);
        MesCall("TraceNGen");
        MUNLOCK(ErrorMessageLock);
    }
    return(-1);
}

/*
        #] TraceNgen :
        #[ Traces :             WORD Traces(term,params,num,level)

        The contents of the AT.TMout array are:
        length,type,subtype,gamma5,factor,sign,gamma's

*/

WORD Traces(PHEAD WORD *term, WORD *params, WORD num, WORD level)
{
    GETBIDENTITY
    switch ( AT.TMout[2] ) {    /* Subtype gives dimension */
        case 0:
            return(TraceN(BHEAD term,params,num,level));
        case 4:
            return(Trace4(BHEAD term,params,num,level));
        case 12:
            return(Trace4(BHEAD term,params,num,level));
        case 20:
            return(Trace4(BHEAD term,params,num,level));
        default:
            return(0);
    }
}

/*
        #] Traces :
        #[ TraceFind :          WORD TraceFind(term,params)
*/

WORD TraceFind(PHEAD WORD *term, WORD *params)
{
    GETBIDENTITY
    WORD *p, *m, *to;
    WORD *termout, *stop, *stop2, number = 0;
    WORD first = 1;
    WORD type, spinline, sp;
    type = params[3];
    spinline = params[4];
    if ( spinline < 0 ) {   /* $ variable. Evaluate */
        sp = DolToIndex(BHEAD -spinline);
        if ( AN.ErrorInDollar || sp < 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("$%s does not have an index value in trace statement in module %l",
                DOLLARNAME(Dollars,-spinline),AC.CModule);
            MUNLOCK(ErrorMessageLock);
```

```
            return(0);
        }
        spinline = sp;
    }
    to = AT.TMout;
    to++;
    *to++ = TAKETRACE;
    *to++ = type;
    *to++ = GAMMA1;
    *to++ = 0;              /* Powers of two */
    *to++ = 1;              /* sign */
    p = term;
    m = p + *p - 1;
    stop = m - ABS(*m);
    termout = m = AT.WorkPointer;
    m++;
    p++;
    while ( p < stop ) {
        stop2 = p + p[1];
        if ( *p == GAMMA && p[FUNHEAD] == spinline ) {
            if ( first ) {
                *m++ = SUBEXPRESSION;
                *m++ = SUBEXPSIZE;
                *m++ = -1;
                *m++ = 1;
                *m++ = DUMMYBUFFER;
                FILLSUB(m)
                first = 0;
            }
            p += FUNHEAD+1;
            while ( p < stop2 ) {
                if ( *p == GAMMA5 ) {
                    if ( AT.TMout[3] == GAMMA5 ) AT.TMout[3] = GAMMA1;
                    else if ( AT.TMout[3] == GAMMA1 ) AT.TMout[3] = GAMMA5;
                    else if ( AT.TMout[3] == GAMMA7 ) AT.TMout[5] = -AT.TMout[5];
                    if ( number & 1 ) AT.TMout[5] = - AT.TMout[5];
                    p++;
                }
                else if ( *p == GAMMA6 ) {
                    if ( number & 1 ) goto F7;
F6:                 if ( AT.TMout[3] == GAMMA6 ) (AT.TMout[4])++;
                    else if ( AT.TMout[3] == GAMMA1 ) AT.TMout[3] = GAMMA6;
                    else if ( AT.TMout[3] == GAMMA5 ) AT.TMout[3] = GAMMA6;
                    else if ( AT.TMout[3] == GAMMA7 ) AT.TMout[5] = 0;
                    p++;
                }
                else if ( *p == GAMMA7 ) {
                    if ( number & 1 ) goto F6;
F7:                 if ( AT.TMout[3] == GAMMA7 ) (AT.TMout[4])++;
                    else if ( AT.TMout[3] == GAMMA1 ) AT.TMout[3] = GAMMA7;
                    else if ( AT.TMout[3] == GAMMA5 ) {
                        AT.TMout[3] = GAMMA7;
                        AT.TMout[5] = -AT.TMout[5];
                    }
                    else if ( AT.TMout[3] == GAMMA6 ) AT.TMout[5] = 0;
                    p++;
                }
                else {
                    *to++ = *p++;
                    number++;
                }
            }
        }
        else {
            while ( p < stop2 ) *m++ = *p++;
        }
    }
    if ( first ) return(0);
    AT.TMout[0] = WORDDIF(to,AT.TMout);
    to = term;
    to += *to;
    while ( p < to ) *m++ = *p++;
    *termout = WORDDIF(m,termout);
    to = term;
    p = termout;
    do { *to++ = *p++; } while ( p < m );
    AT.WorkPointer = term + *term;
    return(1);
}

/*
        #] TraceFind :
        #[ Chisholm :           WORD Chisholm(term,level,num)

        Routines for reorganizing traces.
        The command
            Chisholm,1;
```

```
        will collect the gamma matrices in spinline 1 and see whether
        they have an index in common with another gamma matrix. If this
        is the case the identity
            g_(2,mu)*Tr[g_(1,mu)*S(2)] = S(2)+SR(2)
        is applied (SR is the reversed string).
*/

WORD Chisholm(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    WORD *t, *r, *m, *s, *tt, *rr;
    WORD *mat, *matpoint, *termout, *rdo;
    CBUF *C = cbuf+AM.rbufnum;
    WORD i, j, num = C->lhs[level][2], gam5;
    WORD norm = 0, k, *matp;
/*
    #[ Find : Find possible matrices
*/
    mat = matpoint = AT.WorkPointer;
    t = term;
    r = t + *t - 1; r -= ABS(*r);
    t++;
    i = 0;
    gam5 = GAMMA1;
    while ( t < r ) {
        if ( *t == GAMMA && t[FUNHEAD] == num ) {
            m = t + t[1];
            t += FUNHEAD+1;
            while ( t < m ) {
                if ( *t >= 0 || *t < MINSPEC ) i++;
                else {
                    if ( gam5 == GAMMA1 ) gam5 = *t;
                    else if ( gam5 == GAMMA5 ) {
                        if ( *t == GAMMA5 ) gam5 = GAMMA1;
                        else if ( *t != GAMMA1 ) gam5 = *t;
                    }
                }
                *matpoint++ = *t++;
            }
        }
        else t += t[1];
    }
    if ( ( i & 1 ) != 0 ) return(0);    /* odd trace */
/*
    #] Find :
    #[ Test : Test for contracted index

    This code should be modified.

    We have to check for all possible matches if C->lhs[level][3] == 1
    and the trace contains a gamma5, gamma6 or gamma7.
    Then we normalize by the number of possible contractions (norm) and
    do all of them. This way the Levi-Civita tensors have a maximum
    chance of cancelling each other. This option is activated with
    `contract' and `symmetrize'. Defaults are that they are on, but
    they can be switched off with nocontract and nosymmetrize.
*/
    s = mat;
    while ( s < matpoint ) {
/*
        if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
        indices[*s-AM.OffsetIndex].dimension == 0 ) ) {
*/
        if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
        indices[*s-AM.OffsetIndex].dimension != 4 )
        || ( ( AC.lDefDim != 4 ) && ( *s >= ( AM.OffsetIndex + WILDOFFSET ) ) ) ) {
            s++; continue;
        }
        t = term+1;
        while ( t < r ) {
            if ( *t == GAMMA && t[FUNHEAD] != num ) {
                m = t + t[1];
                t += FUNHEAD+1;
                while ( t < m ) {
                    if ( *t == *s ) {
                        norm++;
                    }
                    t++;
                }
            }
            else t += t[1];
        }
        s++;
    }
    if ( norm == 0 ) return(Generator(BHEAD term,level));  /* No Action */
/*
    #] Test :
```

```
    #[ Do : Process the string

    tt: The subterm
    t:  The matrix
    s:  The matrix in the relevant string

    Cycle the string in mat so that s is at the end.
    Copy the part till the critical GAMMA.
    Copy inside the critical string, copy S, copy tail inside string.
    Important to remember where S is so that we can reverse it later.
    Add term UnitTrace/2/norm.
    Copy rest of term.
    Continue execution with S.
    Reverse S.
    Continue execution with SR.
*/

    if ( C->lhs[level][3] == 0 /* || gam5 == GAMMA1 */ ) norm = 1;

    matp = matpoint;
    for ( k = 0; k < norm; k++ ) {
        matpoint = matp;
        s = mat;
        while ( s < matpoint ) {
/*
            if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
            indices[*s-AM.OffsetIndex].dimension == 0 ) ) {
*/
            if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
            indices[*s-AM.OffsetIndex].dimension != 4 ) ) {
                s++; continue;
            }
            t = term+1;
            while ( t < r ) {
                if ( *t == GAMMA && t[FUNHEAD] != num ) {
                    tt = t;
                    m = t + t[1];
                    t += FUNHEAD+1;
                    while ( t < m ) {
                        if ( *t == *s ) {
                            i = WORDDIF(t,tt);
                            m = mat;
                            while ( m <= s ) *matpoint++ = *m++;
                            t = mat;
                            while ( m < matpoint ) *t++ = *m++;
                            termout = t;
                            m = termout + 1;
                            t = term + 1;
                            while ( t < tt ) {
                                if ( *t != GAMMA || t[FUNHEAD] != num ) {
                                    j = t[1];
                                    NCOPY(m,t,j);
                                }
                                else t += t[1];
                            }

                            tt += tt[1];
                            rdo = m;
                            j = i;
                            while ( --j >= 0 ) *m++ = *t++;
                            matpoint = m;
                            s = mat;
                            while ( s < termout ) *m++ = *s++;
                            m--;
                            t++;
                            while ( t < tt ) *m++ = *t++;
                            rdo[1] = WORDDIF(m,rdo);

                            *m++ = AC.lUniTrace[0];
                            *m++ = AC.lUniTrace[1];
                            *m++ = AC.lUniTrace[2];
                            *m++ = AC.lUniTrace[3];
                            *m++ = SNUMBER;
                            *m++ = 4;
                            *m++ = 2*norm;
                            *m++ = -1;

                            while ( t < r ) {
                                if ( *t != GAMMA || t[FUNHEAD] != num ) {
                                    j = t[1];
                                    NCOPY(m,t,j);
                                }
                                else t += t[1];
                            }
                            rr = term + *term;
                            while ( t < rr ) *m++ = *t++;

```

```
                            *termout = WORDDIF(m,termout);
                            rr = m;
                            t = termout;
                            j = *termout;
                            NCOPY(m,t,j);
                            AT.WorkPointer = m;
                            if ( Generator(BHEAD t,level) ) goto ChisCall;

                            j = WORDDIF(termout,mat)-1;
                            t = matpoint;
                            m = t + j;
                            AT.WorkPointer = rr;
                            while ( m > t ) {
                                i = *--m; *m = *t; *t++ = i;
                            }

                            if ( Generator(BHEAD termout,level) ) goto ChisCall;
                            AT.WorkPointer = mat;

                            goto NextK;
                        }
                        t++;
                    }
                }
                else t += t[1];
            }
            s++;
        }
NextK:;
    }
    return(0);
/*
    #] Do :
*/
ChisCall:
    if ( AM.tracebackflag ) {
        MLOCK(ErrorMessageLock);
        MesCall("Chisholm");
        MUNLOCK(ErrorMessageLock);
    }
    return(-1);
}

/*
        #] Chisholm :
        #[ TenVecFind :         WORD TenVecFind(term,params)
*/

WORD TenVecFind(PHEAD WORD *term, WORD *params)
{
    GETBIDENTITY
    WORD *t, *w, *m, *tstop;
    WORD i, mode, thevector, thetensor, spectator;
    thetensor = params[3];
    thevector = params[4];
    mode = params[5];
    if ( thetensor < 0 ) {  /* $-expression */
        thetensor = DolToTensor(BHEAD -thetensor);
        if ( thetensor < FUNCTION ) {
            if ( thevector > 0 ) {
                thetensor = DolToTensor(BHEAD thevector);
                if ( thetensor < FUNCTION ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("$%s should have been a tensor in module %l"
                        ,DOLLARNAME(Dollars,params[4]),AC.CModule);
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                thevector = DolToVector(BHEAD -params[3]);
                if ( thevector >= 0 ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("$%s should have been a vector in module %l"
                        ,DOLLARNAME(Dollars,-params[3]),AC.CModule);
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
            }
            else {
                MLOCK(ErrorMessageLock);
                MesPrint("$%s should have been a tensor in module %l"
                    ,DOLLARNAME(Dollars,-params[3]),AC.CModule);
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
        }
    }
    if ( thevector > 0 ) {  /* $-expression */
```

```
        thevector = DolToVector(BHEAD thevector);
        if ( thevector >= 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("$%s should have been a vector in module %l"
                ,DOLLARNAME(Dollars,params[4]),AC.CModule);
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
    }
    if ( ( mode & 1 ) != 0 ) {  /* Vector to tensor */
        GETSTOP(term,tstop);
        t = term + 1;
        while ( t < tstop ) {
            if ( *t == DOTPRODUCT ) {
                i = t[1] - 2; t += 2;
                while ( i > 0 ) {
                    spectator = 0;
                    if ( t[2] < 0 ) {}
                    else if ( *t == thevector && t[1] == thevector ) {
                        if ( ( mode & 2 ) == 0 ) spectator = thevector;
                    }
                    else if ( *t == thevector ) spectator = t[1];
                    else if ( t[1] == thevector ) spectator = *t;
                    if ( spectator ) {
                        if ( ( mode & 8 ) == 0 ) goto match;
                        w = SetElements + Sets[params[6]].first;
                        m = SetElements + Sets[params[6]].last;
                        while ( w < m ) {
                            if ( *w == spectator ) break;
                            w++;
                        }
                        if ( w >= m ) goto match;
                    }
                    t += 3;
                    i -= 3;
                }
            }
            else if ( *t == VECTOR ) {
                i = t[1] - 2; t += 2;
                while ( i > 0 ) {
                    if ( *t == thevector ) goto match;
                    t += 2;
                    i -= 2;
                }
            }
            else if ( *t == thetensor ) t += t[1];
            else if ( *t >= FUNCTION ) {
                if ( functions[*t-FUNCTION].spec > 0 ) {
                    w = t + t[1];
                    t += FUNHEAD;
                    while ( t < w ) {
                        if ( *t == thevector ) goto match;
                        t++;
                    }
                }
                else if ( ( mode & 4 ) != 0 ) {
                    w = t + t[1];
                    t += FUNHEAD;
                    while ( t < w ) {
                        if ( *t == -VECTOR && t[1] == thevector ) goto match;
                        else if ( *t > 0 ) t += *t;
                        else if ( *t <= -FUNCTION ) t++;
                        else t += 2;
                    }
                }
                else t += t[1];
            }
            else t += t[1];
        }
    }
    else {                      /* Tensor to Vector */
        GETSTOP(term,tstop);
        t = term+1;
        while ( t < tstop ) {
            if ( *t == thetensor ) goto match;
            t += t[1];
        }
    }
    return(0);
match:
    AT.TMout[0] = 5;
    AT.TMout[1] = TENVEC;
    AT.TMout[2] = thetensor;
    AT.TMout[3] = thevector;
    AT.TMout[4] = mode;
    if ( ( mode & 8 ) != 0 ) { AT.TMout[0] = 6; AT.TMout[5] = params[6]; }
    return(1);
```

```
}

/*
        #] TenVecFind :
        #[ TenVec :             WORD TenVec(term,params,num,level)
*/

WORD TenVec(PHEAD WORD *term, WORD *params, WORD num, WORD level)
{
    GETBIDENTITY
    WORD *t, *m, *w, *termout, *tstop, *outlist, *ou, *ww, *mm;
    WORD i, j, k, x, mode, thevector, thetensor, DumNow, spectator;
    DUMMYUSE(num);
    thetensor = params[2];
    thevector = params[3];
    mode = params[4];
    termout = AT.WorkPointer;
    DumNow = AR.CurDum = DetCurDum(BHEAD term);
    if ( ( mode & 1 ) != 0 ) {  /* Vector to tensor */
        AT.WorkPointer += *term;
        ou = outlist = AT.WorkPointer;
        GETSTOP(term,tstop);
        t = term + 1;
        m = termout + 1;
        while ( t < tstop ) {
            if ( *t == DOTPRODUCT ) {
                i = t[1] - 2;
                w = m;
                *m++ = *t++; *m++ = *t++;
                while ( i > 0 ) {
                    spectator = 0;
                    if ( t[2] < 0 ) {
                        *m++ = *t++; *m++ = *t++; *m++ = *t++;
                    }
                    else if ( *t == thevector && t[1] == thevector ) {
                        if ( ( mode & 2 ) == 0 ) spectator = thevector;
                        else {
                            *m++ = *t++; *m++ = *t++; *m++ = *t++;
                        }
                    }
                    else if ( *t == thevector ) spectator = t[1];
                    else if ( t[1] == thevector ) spectator = *t;
                    else {
                        *m++ = *t++; *m++ = *t++; *m++ = *t++;
                    }
                    if ( spectator ) {
                        if ( ( mode & 8 ) == 0 ) goto noveto;
                        ww = SetElements + Sets[params[5]].first;
                        mm = SetElements + Sets[params[5]].last;
                        while ( ww < mm ) {
                            if ( *ww == spectator ) break;
                            ww++;
                        }
                        if ( ww < mm ) {
                            *m++ = *t++; *m++ = *t++; *m++ = *t++;
                        }
                        else {
noveto:                 if ( spectator == thevector ) {
                            for ( j = 0; j < t[2]; j++ ) {
                                *ou++ = ++AR.CurDum;
                                *ou++ = AR.CurDum;
                            }
                            t += 3;
                        }
                        else {
                            for ( j = 0; j < t[2]; j++ ) *ou++ = spectator;
                            t += 3;
                        }}
                    }
                    i -= 3;
                }
                w[1] = WORDDIF(m,w);
                if ( w[1] == 2 ) m = w;
            }
            else if ( *t == VECTOR ) {
                i = t[1] - 2; w = m;
                *m++ = *t++; *m++ = *t++;
                while ( i > 0 ) {
                    if ( *t == thevector ) {
                        *ou++ = t[1];
                        t += 2;
                    }
                    else { *m++ = *t++; *m++ = *t++; }
                    i -= 2;
                }
                w[1] = WORDDIF(m,w);
```

```
                if ( w[1] == 2 ) m = w;
            }
            else if ( *t == thetensor ) {
                i = t[1] - FUNHEAD;
                t += FUNHEAD;
                NCOPY(ou,t,i);
            }
            else if ( *t >= FUNCTION ) {
                if ( functions[*t-FUNCTION].spec > 0 ) {
                    w = t + t[1];
                    i = FUNHEAD;
                    NCOPY(m,t,i);
                    while ( t < w ) {
                        if ( *t == thevector ) {
                            *m++ = ++AR.CurDum;
                            *ou++ = AR.CurDum;
                            t++;
                        }
                        else *m++ = *t++;
                    }
                }
                else if ( ( mode & 4 ) != 0 ) {
                    w = t + t[1];
                    i = FUNHEAD;
                    NCOPY(m,t,i);
                    while ( t < w ) {
                        if ( *t == -VECTOR && t[1] == thevector ) {
                            *m++ = -INDEX;
                            *m++ = ++AR.CurDum;
                            *ou++ = AR.CurDum;
                            t += 2;
                        }
                        else if ( *t > 0 ) {
                            i = *t;
                            NCOPY(m,t,i);
                        }
                        else if ( *t <= -FUNCTION ) *m++ = *t++;
                        else { *m++ = *t++; *m++ = *t++; }
                    }
                }
                else goto docopy;
            }
            else {
docopy:
                i = t[1];
                NCOPY(m,t,i);
            }
        }
        i = WORDDIF(ou,outlist);
        if ( i > 0 ) {
            for ( j = 1; j < i; j++ ) {
                if ( outlist[j-1] > outlist[j] ) {
                    x = outlist[j-1]; outlist[j-1] = outlist[j]; outlist[j] = x;
                    for ( k = j-1; k > 0; k-- ) {
                        if ( outlist[k-1] <= outlist[k] ) break;
                        x = outlist[k-1]; outlist[k-1] = outlist[k]; outlist[k] = x;
                    }
                }
            }

            *m++ = thetensor;
            *m++ = FUNHEAD + i;
            *m++ = DIRTYSYMFLAG;
            FILLFUN3(m)
            ou = outlist;
            NCOPY(m,ou,i);
        }
        w = term + *term;
        while ( t < w ) *m++ = *t++;
    }
    else {                      /* Tensor to Vector */
        GETSTOP(term,tstop);
        t = term+1;
        m = termout+1;
        while ( t < tstop ) {
            if ( *t != thetensor ) {
                i = t[1];
                NCOPY(m,t,i);
            }
            else {
                i = t[1] - FUNHEAD;
                t += FUNHEAD;
                if ( i > 0 ) {
                    w = m; m += 2;
                    while ( --i >= 0 ) {
                        *m++ = thevector;
                        *m++ = *t++;
```

```
                        }
                        *w = DELTA;
                        w[1] = WORDDIF(m,w);
                    }
                }
            }
            w = term + *term;
            while ( t < w ) *m++ = *t++;
        }
        *termout = WORDDIF(m,termout);
        AT.WorkPointer = m;
        *AT.TMout = 0;
        if ( Generator(BHEAD termout,level) ) goto fromTenVec;
        AR.CurDum = DumNow;
        AT.WorkPointer = termout;
        return(0);
fromTenVec:
        if ( AM.tracebackflag ) {
            MLOCK(ErrorMessageLock);
            MesCall("TenVec");
            MUNLOCK(ErrorMessageLock);
        }
        return(-1);
}

/*
            #] TenVec :
        #] Operations :
*/
```

4.73 parallel.c File Reference

```
#include "form3.h"
#include "vector.h"
```

Include dependency graph for parallel.c:

Data Structures

- struct NoDe
- struct dollar_buf
- struct bufIPstruct

Macros

- #define PRINTFBUF(TEXT, TERM, SIZE) {}
- #define SWAP(x, y)
- #define PACK_LONG(p, n)
- #define UNPACK_LONG(p, n)
- #define CHECK(condition) _CHECK(condition, __FILE__, __LINE__)
- #define _CHECK(condition, file, line) __CHECK(condition, file, line)

- #define __CHECK(condition, file, line)
- #define DBGOUT(lv1, lv2, a) do { if (lv1 >= lv2) { printf a; fflush(stdout); } } while (0)
- #define DBGOUT_NINTERMS(lv, a)
- #define PF_STATS_SIZE 5
- #define PF_SNDFILEBUFSIZE 4096
- #define recvBuffer logBuffer /* (master) The buffer for receiving messages. */

Typedefs

- typedef struct NoDe NODE
- typedef struct bufIPstruct **bufIPstruct_t**

Functions

- LONG PF_RealTime (int)
- int PF_LibInit (int *, char ***)
- int PF_LibTerminate (int)
- int PF_Probe (int *)
- int PF_RecvWbuf (WORD *, LONG *, int *)
- int PF_IRecvRbuf (PF_BUFFER *, int, int)
- int PF_WaitRbuf (PF_BUFFER *, int, LONG *)
- int PF_RawSend (int dest, void *buf, LONG l, int tag)
- LONG PF_RawRecv (int *src, void *buf, LONG thesize, int *tag)
- int PF_RawProbe (int *src, int *tag, int *bytesize)
- int PF_EndSort (void)
- WORD PF_Deferred (WORD *term, WORD level)
- int PF_Processor (EXPRESSIONS e, WORD i, WORD LastExpression)
- int PF_Init (int *argc, char ***argv)
- int PF_Terminate (int errorcode)
- LONG PF_GetSlaveTimes (void)
- LONG PF_BroadcastNumber (LONG x)
- void PF_BroadcastBuffer (WORD **buffer, LONG *length)
- int PF_BroadcastString (UBYTE *str)
- int PF_BroadcastPreDollar (WORD **dbuffer, LONG *newsize, int *numterms)
- int PF_CollectModifiedDollars (void)
- int PF_BroadcastModifiedDollars (void)
- int PF_BroadcastRedefinedPreVars (void)
- int PF_StoreInsideInfo (void)
- int PF_RestoreInsideInfo (void)
- int PF_BroadcastCBuf (int bufnum)
- int PF_BroadcastExpFlags (void)
- int PF_BroadcastExpr (EXPRESSIONS e, FILEHANDLE *file)
- int PF_BroadcastRHS (void)
- int PF_InParallelProcessor (void)
- int PF_SendFile (int to, FILE *fd)
- int PF_RecvFile (int from, FILE *fd)
- void PF_MLock (void)
- void PF_MUnlock (void)
- LONG PF_WriteFileToFile (int handle, UBYTE *buffer, LONG size)
- void PF_FlushStdOutBuffer (void)
- void PF_FreeErrorMessageBuffers (void)

Variables

- PARALLELVARS PF

4.73.1 Detailed Description

Message passing library independent functions of parform

This file contains functions needed for the parallel version of form3 these functions need no real link to the message passing libraries, they only need some interface dependent preprocessor definitions (check parallel.h). So there still need two different objectfiles to be compiled for mpi and pvm!

Definition in file parallel.c.

4.73.2 Macro Definition Documentation

4.73.2.1 PRINTFBUF

```
#define PRINTFBUF(
            TEXT,
            TERM,
            SIZE ) {}
```

Definition at line 117 of file parallel.c.

4.73.2.2 SWAP

```
#define SWAP(
            x,
            y )
```

Value:

```
do { \
    char swap_tmp__[sizeof(x) == sizeof(y) ? (int)sizeof(x) : -1]; \
    memcpy(swap_tmp__, &y, sizeof(x)); \
    memcpy(&y, &x, sizeof(x)); \
    memcpy(&x, swap_tmp__, sizeof(x)); \
} while (0)
```

Swaps the variables *x* and *y*. If sizeof(x) != sizeof(y) then a compilation error will occur. A set of memcpy calls with constant sizes is expected to be inlined by the optimisation.

Definition at line 124 of file parallel.c.

4.73.2.3 PACK_LONG

```
#define PACK_LONG(
            p,
            n )
```

Value:

```
do { \
    *(p)++ = (UWORD)((ULONG)(n) & (ULONG)WORDMASK); \
    *(p)++ = (UWORD)(((ULONG)(n) » BITSINWORD) & (ULONG)WORDMASK); \
} while (0)
```

Packs a LONG value *n* to a WORD buffer *p*.

Definition at line 135 of file parallel.c.

4.73.2.4 UNPACK_LONG

```
#define UNPACK_LONG(
            p,
            n )
```

Value:

```
do { \
    (n) = (LONG)((((ULONG)(p)[1] & (ULONG)WORDMASK) « BITSINWORD) | ((ULONG)(p)[0] & (ULONG)WORDMASK));
  \
    (p) += 2; \
} while (0)
```

Unpacks a LONG value *n* from a WORD buffer *p*.

Definition at line 144 of file parallel.c.

4.73.2.5 CHECK

```
#define CHECK(
            condition ) _CHECK(condition, __FILE__, __LINE__)
```

A simple check for unrecoverable errors.

Definition at line 153 of file parallel.c.

4.73.2.6 _CHECK

```
#define _CHECK(
            condition,
            file,
            line ) __CHECK(condition, file, line)
```

Definition at line 154 of file parallel.c.

4.73.2.7 __CHECK

```
#define __CHECK(
            condition,
            file,
            line )
```

Value:

```
do { \
    if ( !(condition) ) { \
        Error0("Fatal error at " file ":" #line); \
        Terminate(-1); \
    } \
} while (0)
```

Definition at line 155 of file parallel.c.

4.73.2.8 DBGOUT

```
#define DBGOUT(
            lv1,
            lv2,
            a ) do { if ( lv1 >= lv2 ) { printf a; fflush(stdout); } } while (0)
```

Definition at line 166 of file parallel.c.

4.73.2.9 DBGOUT_NINTERMS

```
#define DBGOUT_NINTERMS(
            lv,
            a )
```

Definition at line 169 of file parallel.c.

4.73.2.10 PF_STATS_SIZE

```
#define PF_STATS_SIZE 5
```

Definition at line 178 of file parallel.c.

4.73.2.11 PF_SNDFILEBUFSIZE

```
#define PF_SNDFILEBUFSIZE 4096
```

Definition at line 4198 of file parallel.c.

4.73.2.12 recvBuffer

```
#define recvBuffer logBuffer /* (master) The buffer for receiving messages.  */
```

Definition at line 4306 of file parallel.c.

4.73.3 Typedef Documentation

4.73.3.1 NODE

```
typedef struct NoDe NODE
```

A node for the tree of losers in the final sorting on the master.

4.73.4 Function Documentation

4.73.4.1 PF_RealTime()

```
LONG PF_RealTime (
            int i )
```

Returns the realtime in 1/100 sec. as a LONG.

Parameters

i	the timer will be reset if i == 0.

Returns

the real elapsed time in 1/100 second.

Definition at line 101 of file mpi.c.

Referenced by PF_Init().

4.73.4.2 PF_LibInit()

```
int PF_LibInit (
            int * argcp,
            char *** argvp )
```

Performs all library dependent initializations.

Parameters

argcp	pointer to the number of arguments.
argvp	pointer to the arguments.

Returns

0 if OK, nonzero on error.

Definition at line 123 of file mpi.c.

Referenced by PF_Init().

4.73.4.3 PF_LibTerminate()

```
int PF_LibTerminate (
            int error )
```

Exits mpi, when there is an error either indicated or happening, returnvalue is 1, else returnvalue is 0.

Parameters

error	an error code.

Returns

0 if OK, nonzero on error.

Definition at line 209 of file mpi.c.

Referenced by PF_Terminate().

4.73.4.4 PF_Probe()

```
int PF_Probe (
            int * src )
```

Probes the next incoming message. If src == PF_ANY_SOURCE this operation is blocking, otherwise nonbloking.

Parameters

in,out	*src*	the source process number. In output, the process number of actual found source.

Returns

the tag value of the next incoming message if found, 0 if a nonbloking probe (input src != PF_ANY_SOURCE) did not find any messages. A negative returned value indicates an error.

Definition at line 230 of file mpi.c.

4.73.4.5 PF_RecvWbuf()

```
int PF_RecvWbuf (
            WORD * b,
            LONG * s,
            int * src )
```

Blocking receive of a `WORD` buffer.

Parameters

out	*b*	the buffer to store the received data.
in,out	*s*	the size of the buffer. The output value is the actual size of the received data.
in,out	*src*	the source process number. The output value is the process number of actual source.

Returns

the received message tag. A negative value indicates an error.

Definition at line 337 of file mpi.c.

4.73.4.6 PF_IRecvRbuf()

```
int PF_IRecvRbuf (
            PF_BUFFER * r,
            int bn,
            int from )
```

Posts nonblocking receive for the active receive buffer. The buffer is filled from `full` to `stop`.

Parameters

r	the PF_BUFFER struct for the nonblocking receive.
bn	the index of the cyclic buffer.
from	the source process number.

Returns

0 if OK, nonzero on error.

Definition at line 366 of file mpi.c.

4.73.4.7 PF_WaitRbuf()

```
int PF_WaitRbuf (
            PF_BUFFER * r,
            int bn,
            LONG * size )
```

Waits for the buffer *bn* to finish a pending nonblocking receive. It returns the received tag and in `*size` the number of field received. If the receive is already finished, just return the flag and size, else wait for it to finish, but also check for other pending receives.

Parameters

	r	the PF_BUFFER struct for the pending nonblocking receive.
	bn	the index of the cyclic buffer.
out	*size*	the actual size of received data.

Returns

the received message tag. A negative value indicates an error.

Definition at line 400 of file mpi.c.

4.73.4.8 PF_RawSend()

```
int PF_RawSend (
            int dest,
            void * buf,
            LONG l,
            int tag )
```

Sends *l* bytes from *buf* to *dest*. Returns 0 on success, or -1.

Parameters

dest	the destination process number.
buf	the send buffer.
l	the size of data in the send buffer in bytes.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 463 of file mpi.c.

Referenced by PF_MUnlock(), and PF_SendFile().

4.73.4.9 PF_RawRecv()

```
LONG PF_RawRecv (
            int * src,
            void * buf,
            LONG thesize,
            int * tag )
```

Receives not more than *thesize* bytes from *src*, returns the actual number of received bytes, or -1 on failure.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
out	*buf*	the receive buffer.
	thesize	the size of the receive buffer in bytes.
out	*tag*	the message tag of the actual received message.

Returns

the actual sizeof received data in bytes, or -1 on failure.

Definition at line 484 of file mpi.c.

Referenced by PF_RecvFile().

4.73.4.10 PF_RawProbe()

```
int PF_RawProbe (
            int * src,
            int * tag,
            int * bytesize )
```

Probes an incoming message.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
in,out	*tag*	the message tag. In output, that of the actual received message.
out	*bytesize*	the size of incoming data in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 508 of file mpi.c.

4.73.4.11 PF_EndSort()

```
int PF_EndSort (
            void  )
```

Finishes a master sorting with collecting terms from slaves. Called by EndSort().

If this is not the masterprocess, just initialize the sendbuffers and return 0, else PF_EndSort() sends the rest of the terms in the sendbuffer to the next slave and a dummy message to all slaves with tag PF_ENDSORT_MSGTAG. Then it receives the sorted terms, sorts them using a recursive 'tree of losers' (PF_GetLoser()) and writes them to the outputfile.

Returns

1 if the sorting on the master was done. 0 if EndSort() still must perform a regular sorting becuase it is not at the ground level or not on the master or in the sequential mode or in the InParallel mode. -1 if an error occured.

Remarks

The slaves will send the sorted terms back to the master in the regular sorting (after the initialization of the send buffer in PF_EndSort()). See PutOut() and FlushOut().

This function has been changed such that when it returns 1, AM.S0->TermsLeft is set correctly. But AM.S0->GenTerms is not set: it will be set after collecting the statistics from the slaves at the end of PF_Processor(). (TU 30 Jun 2011)

Definition at line 864 of file parallel.c.

References FlushOut(), and PutOut().

Referenced by EndSort().

Here is the call graph for this function:

4.73.4.12 PF_Deferred()

```
WORD PF_Deferred (
            WORD * term,
            WORD level )
```

Replaces Deferred() on the slaves.

Parameters

term	the term that must be multiplied by the contents of the current bracket.
level	the compiler level.

Returns

0 if OK, nonzero on error.

Definition at line 1208 of file parallel.c.

References Generator(), and InsertTerm().

Referenced by Generator().

Here is the call graph for this function:

4.73.4.13 PF_Processor()

```
int PF_Processor (
            EXPRESSIONS e,
            WORD i,
            WORD LastExpression )
```

Replaces parts of Processor() on the masters and slaves. On the master PF_Processor() is responsible for proper distribution of terms from the input file to the slaves. On the slaves it calls Generator() for all the terms that this process gets, but PF_GetTerm() gets terms from the master (not directly from infile).

Parameters

e	The pointer to the current expression.
i	The index for the current expression.
LastExpression	The flag indicating whether it is the last expression.

Returns

0 if OK, nonzero on error.

Definition at line 1540 of file parallel.c.

References EndSort(), Generator(), LowerSortLevel(), NewSort(), PACK_LONG, PF_ISendSbuf(), PF_LongSinglePack(), PF_LongSingleReceive(), PF_LongSingleSend(), PF_LongSingleUnpack(), PF_PrepareLongSinglePack(), PutOut(), StoreTerm(), SWAP, TimeCPU(), and WriteStats().

Referenced by Processor().

Here is the call graph for this function:

4.73.4.14 PF_Init()

```
int PF_Init (
            int * argc,
            char *** argv )
```

All the library independent stuff. PF_LibInit() should do all library dependent initializations.

Parameters

argc	pointer to the number of arguments.
argv	pointer to the arguments.

Returns

0 if OK, nonzero on error.

Definition at line 1953 of file parallel.c.

References PF_Broadcast(), PF_LibInit(), PF_Pack(), PF_PreparePack(), PF_RealTime(), and PF_Unpack().

Here is the call graph for this function:

4.73.4.15 PF_Terminate()

```
int PF_Terminate (
            int errorcode )
```

Performs the finalization of ParFORM. To be called by Terminate().

Parameters

error	an error code.

Returns

0 if OK, nonzero on error.

Definition at line 2047 of file parallel.c.

References PF_LibTerminate().

Here is the call graph for this function:

4.73.4.16 PF_GetSlaveTimes()

```
LONG PF_GetSlaveTimes (
            void  )
```

Returns the total CPU time of all slaves together. This function must be called on the master and all slaves.

Returns

on the master, the sum of CPU times on all slaves.

Definition at line 2063 of file parallel.c.

References TimeCPU().

Here is the call graph for this function:

4.73.4.17 PF_BroadcastNumber()

```
LONG PF_BroadcastNumber (
            LONG x )
```

Broadcasts a LONG value from the master to the all slaves.

Parameters

x	the number to be broadcast (set on the master).

Returns

the synchronised result.

Definition at line 2083 of file parallel.c.

References PF_Bcast().

Referenced by DoCheckpoint(), PF_BroadcastBuffer(), and StartVariables().

Here is the call graph for this function:

4.73.4.18 PF_BroadcastBuffer()

```
void PF_BroadcastBuffer (
            WORD ** buffer,
            LONG * length )
```

Broadcasts a buffer from the master to all the slaves.

Parameters

in,out	*buffer*	on the master, the buffer to be broadcast. On the slaves, the buffer will be allocated if the length is greater than 0. The caller must free it.
in,out	*length*	on the master, the length of the buffer to be broadcast. On the slaves, it receives the length of transfered buffer. The actual transfer occurs only if the length is greater than 0.

Definition at line 2110 of file parallel.c.

References PF_Bcast(), and PF_BroadcastNumber().

Here is the call graph for this function:

4.73.4.19 PF_BroadcastString()

```
int PF_BroadcastString (
            UBYTE * str )
```

Broadcasts a string from the master to all slaves.

Parameters

in,out	*str*	The pointer to a null-terminated string.

Returns

0 if OK, nonzero on error.

Definition at line 2152 of file parallel.c.

References PF_Broadcast(), PF_PackString(), PF_PreparePack(), and PF_UnpackString().

Here is the call graph for this function:

4.73.4.20 PF_BroadcastPreDollar()

```
int PF_BroadcastPreDollar (
            WORD ** dbuffer,
            LONG * newsize,
            int * numterms )
```

Broadcasts dollar variables set as a preprocessor variables. Only the master is able to make an assignment like #$a=g; where g is an expression: only the master has an access to the expression. So, the master broadcasts the result to slaves.

The result is in ∗dbuffer of the size is ∗newsize (in number of WORDs), +1 for trailing zero. For slave newsize and numterms are output parameters.

Parameters

in,out	*dbuffer*	the buffer for a dollar variable.
in,out	*newsize*	the size of the dollar variable in WORDs.
in,out	*numterms*	the number of terms in the dollar variable.

Returns

0 if OK, nonzero on error.

Definition at line 2207 of file parallel.c.

References PF_Broadcast(), PF_Pack(), PF_PreparePack(), and PF_Unpack().

Here is the call graph for this function:

4.73.4.21 PF_CollectModifiedDollars()

```
int PF_CollectModifiedDollars (
            void  )
```

Combines modified dollar variables on the all slaves, and store them into those on the master.

The potentially modified dollar variables are given in PotModdollars, and the number of them is given by NumPot↩ Moddollars.

The current module could be executed in parallel only if all potentially modified variables are listed in ModOptdollars, otherwise the module was switched to the sequential mode.

Returns

0 if OK, nonzero on error.

Definition at line 2495 of file parallel.c.

References EndSort(), Generator(), LowerSortLevel(), NewSort(), PF_LongSinglePack(), PF_LongSingleReceive(), PF_LongSingleSend(), PF_LongSingleUnpack(), PF_PrepareLongSinglePack(), VectorInit, VectorPtr, VectorReserve, and VectorSize.

Here is the call graph for this function:

4.73.4.22 PF_BroadcastModifiedDollars()

```
int PF_BroadcastModifiedDollars (
            void  )
```

Broadcasts modified dollar variables on the master to the all slaves.

The potentially modified dollar variables are given in PotModdollars, and the number of them is given by NumPot↩Moddollars.

The current module could be executed in parallel only if all potentially modified variables are listed in ModOptdollars, otherwise the module was switched to the sequential mode. In either cases, we need to broadcast them.

Returns

0 if OK, nonzero on error.

Definition at line 2774 of file parallel.c.

References PF_LongMultiBroadcast(), and PF_PrepareLongMultiPack().

Here is the call graph for this function:

4.73.4.23 PF_BroadcastRedefinedPreVars()

```
int PF_BroadcastRedefinedPreVars (
            void  )
```

Broadcasts preprocessor variables, which were changed by the Redefine statements in the current module, from the master to the all slaves.

The potentially redefined preprocessor variables are given in AC.pfirstnum, and the number of them is given by AC.numpfirstnum. For an actually redefined variable, the corresponding value in AC.inputnumbers is non-negative.

Returns

0 if OK, nonzero on error.

Definition at line 2991 of file parallel.c.

References PF_LongMultiBroadcast(), PF_PrepareLongMultiPack(), PutPreVar(), VectorPtr, and VectorReserve.

Here is the call graph for this function:

4.73.4.24 PF_StoreInsideInfo()

```
int PF_StoreInsideInfo (
            void  )
```

Definition at line 3081 of file parallel.c.

4.73.4.25 PF_RestoreInsideInfo()

```
int PF_RestoreInsideInfo (
            void  )
```

Definition at line 3107 of file parallel.c.

4.73.4.26 PF_BroadcastCBuf()

```
int PF_BroadcastCBuf (
            int bufnum )
```

Broadcasts a compiler buffer specified by *bufnum* from the master to the all slaves.

Parameters

bufnum	The index of the compiler buffer to be broadcast.

Returns

0 if OK, nonzero on error.

Definition at line 3133 of file parallel.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::dimension, finishcbuf(), CbUf::lhs, CbUf::numdum, CbUf::NumTerms, PF_LongMultiBroadcast(), PF_PrepareLongMultiPack(), CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Here is the call graph for this function:

4.73.4.27 PF_BroadcastExpFlags()

```
int PF_BroadcastExpFlags (
            void  )
```

Broadcasts AR.expflags and several properties of each expression, e.g., e->vflags, from the master to all slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3244 of file parallel.c.

References PF_LongMultiBroadcast(), and PF_PrepareLongMultiPack().

Here is the call graph for this function:

4.73.4.28 PF_BroadcastExpr()

```
int PF_BroadcastExpr (
            EXPRESSIONS e,
            FILEHANDLE * file )
```

Broadcasts an expression from the master to the all slaves.

Parameters

e	The expression to be broadcast.
file	The file in which the expression is sitting.

Returns

0 if OK, nonzero on error.

Definition at line 3536 of file parallel.c.

Referenced by PF_BroadcastRHS().

4.73.4.29 PF_BroadcastRHS()

```
int PF_BroadcastRHS (
            void  )
```

Broadcasts expressions appearing in the right-hand side from the master to the all slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3564 of file parallel.c.

References PF_BroadcastExpr().

Referenced by Processor().

Here is the call graph for this function:

4.73.4.30 PF_InParallelProcessor()

```
int PF_InParallelProcessor (
            void  )
```

Processes expressions in the InParallel mode, i.e., dividing expressions marked by partodo over the slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3611 of file parallel.c.

Referenced by Processor().

4.73.4.31 PF_SendFile()

```
int PF_SendFile (
            int to,
            FILE * fd )
```

Sends a file to the process specified by *to*.

Parameters

to	the destination process number.
fd	the file to be sent.

Returns

the size of sent data in bytes, or -1 on error.

Definition at line 4207 of file parallel.c.

References PF_RawSend().

Referenced by DoCheckpoint().

Here is the call graph for this function:

4.73.4.32 PF_RecvFile()

```
int PF_RecvFile (
            int from,
            FILE * fd )
```

Receives a file from the process specified by *from*.

Parameters

from	the source process number.
fd	the file to save the received data.

Returns

the size of received data in bytes, or -1 on error.

Definition at line 4245 of file parallel.c.

References PF_RawRecv().

Referenced by DoCheckpoint().

Here is the call graph for this function:

4.73.4.33 PF_MLock()

```
void PF_MLock (
            void )
```

A function called by MLOCK(ErrorMessageLock) for slaves.

Definition at line 4326 of file parallel.c.

References VectorClear.

4.73.4.34 PF_MUnlock()

```
void PF_MUnlock (
            void  )
```

A function called by MUNLOCK(ErrorMessageLock) for slaves.

Definition at line 4342 of file parallel.c.

References PF_RawSend(), VectorEmpty, VectorPtr, and VectorSize.

Here is the call graph for this function:

4.73.4.35 PF_WriteFileToFile()

```
LONG PF_WriteFileToFile (
            int handle,
            UBYTE * buffer,
            LONG size )
```

Replaces WriteFileToFile() on the master and slaves.

It copies the given buffer into internal buffers if called between MLOCK(ErrorMessageLock) and MUNLOCK(Error↩ MessageLock) for slaves and handle is StdOut or LogHandle, otherwise calls WriteFileToFile().

Parameters

handle	a file handle that specifies the output.
buffer	a pointer to the source buffer containing the data to be written.
size	the size of data to be written in bytes.

Returns

the actual size of data written to the output in bytes.

Definition at line 4371 of file parallel.c.

References VectorClear, VectorPtr, VectorPushBacks, and VectorSize.

4.73.4.36 PF_FlushStdOutBuffer()

```
void PF_FlushStdOutBuffer (
            void  )
```

Explicitly Flushes the buffer for the standard output on the master, which is used if PF_ENABLE_STDOUT_↩ BUFFERING is defined.

Definition at line 4465 of file parallel.c.

References VectorClear, VectorPtr, and VectorSize.

4.73.4.37 PF_FreeErrorMessageBuffers()

```
void PF_FreeErrorMessageBuffers (
            void  )
```

Frees the buffers allocated for the synchronized output.

Currently, not used anywhere, but could be used in PF_Terminate().

Definition at line 4601 of file parallel.c.

References VectorFree.

4.73.5 Variable Documentation

4.73.5.1 PF

```
PARALLELVARS PF
```

Definition at line 90 of file parallel.c.

4.74 parallel.c

Go to the documentation of this file.

```
00001
00011 /* #[ License : */
00012 /*
00013  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00014  *   When using this file you are requested to refer to the publication
00015  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016  *   This is considered a matter of courtesy as the development was paid
00017  *   for by FOM the Dutch physics granting agency and we would like to
00018  *   be able to track its scientific use to convince FOM of its value
00019  *   for the community.
00020  *
00021  *   This file is part of FORM.
00022  *
00023  *   FORM is free software: you can redistribute it and/or modify it under the
00024  *   terms of the GNU General Public License as published by the Free Software
00025  *   Foundation, either version 3 of the License, or (at your option) any later
00026  *   version.
00027  *
00028  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00031  *   details.
00032  *
```

```
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ includes :
*/
#include "form3.h"
#include "vector.h"

/*
#define PF_DEBUG_BCAST_LONG
#define PF_DEBUG_BCAST_BUF
#define PF_DEBUG_BCAST_PREDOLLAR
#define PF_DEBUG_BCAST_RHSEXPR
#define PF_DEBUG_BCAST_DOLLAR
#define PF_DEBUG_BCAST_PREVAR
#define PF_DEBUG_BCAST_CBUF
#define PF_DEBUG_BCAST_EXPRFLAGS
#define PF_DEBUG_REDUCE_DOLLAR
*/

/* mpi.c */
LONG PF_RealTime(int);
int PF_LibInit(int*, char***);
int PF_LibTerminate(int);
int PF_Probe(int*);
int PF_RecvWbuf(WORD*,LONG*,int*);
int PF_IRecvRbuf(PF_BUFFER*,int,int);
int PF_WaitRbuf(PF_BUFFER *,int,LONG *);
int PF_RawSend(int dest, void *buf, LONG l, int tag);
LONG PF_RawRecv(int *src,void *buf,LONG thesize,int *tag);
int PF_RawProbe(int *src, int *tag, int *bytesize);

/* Private functions */

static int PF_WaitAllSlaves(void);

static void PF_PackRedefinedPreVars(void);
static void PF_UnpackRedefinedPreVars(void);

static int PF_Wait4MasterIP(int tag);
static int PF_DoOneExpr(void);
static int PF_ReadMaster(void);/*reads directly to its scratch!*/
static int PF_Slave2MasterIP(int src);/*both master and slave*/
static int PF_Master2SlaveIP(int dest, EXPRESSIONS e);
static int PF_WalkThrough(WORD *t, LONG l, LONG chunk, LONG *count);
static int PF_SendChunkIP(FILEHANDLE *curfile,  POSITION *position, int to, LONG thesize);
static int PF_RecvChunkIP(FILEHANDLE *curfile, int from, LONG thesize);

static void PF_ReceiveErrorMessage(int src, int tag);
static void PF_CatchErrorMessages(int *src, int *tag);
static void PF_CatchErrorMessagesForAll(void);
static int PF_ProbeWithCatchingErrorMessages(int *src);

/* Variables */

PARALLELVARS PF;
#ifdef MPI2
 WORD *PF_shared_buff;
#endif

static LONG     PF_goutterms;  /* (master) Total out terms at PF_EndSort(), used in PF_Statistics(). */
static POSITION PF_exprsize;   /* (master) The size of the expression at PF_EndSort(), used in PF_Processor(). */

/*
    This will work well only under Linux, see
        #ifdef PF_WITH_SCHED_YIELD
    below in PF_WaitAllSlaves().
*/
#ifdef PF_WITH_SCHED_YIELD
 #include <sched.h>
#endif

#ifdef PF_WITHLOG
 #define PRINTFBUF(TEXT,TERM,SIZE)  { UBYTE lbuf[24]; if(PF.log){ WORD iii;\
  NumToStr(lbuf,AC.CModule); \
  fprintf(stderr,"[%d|%s] %s : ",PF.me,lbuf,(char*)TEXT);\
  if(TERM){ fprintf(stderr,"[%d] ",(int)(*TERM));\
    if((SIZE)<500 && (SIZE)>0) for(iii=1;iii<(SIZE);iii++)\
      fprintf(stderr,"%d ",TERM[iii]); }\
  fprintf(stderr,"\n");\
  fflush(stderr); } }
#else
 #define PRINTFBUF(TEXT,TERM,SIZE) {}
```

```
#endif

#define SWAP(x, y) \
    do { \
        char swap_tmp__[sizeof(x) == sizeof(y) ? (int)sizeof(x) : -1]; \
        memcpy(swap_tmp__, &y, sizeof(x)); \
        memcpy(&y, &x, sizeof(x)); \
        memcpy(&x, swap_tmp__, sizeof(x)); \
    } while (0)

#define PACK_LONG(p, n) \
    do { \
        *(p)++ = (UWORD)((ULONG)(n) & (ULONG)WORDMASK); \
        *(p)++ = (UWORD)(((ULONG)(n) » BITSINWORD) & (ULONG)WORDMASK); \
    } while (0)

#define UNPACK_LONG(p, n) \
    do { \
        (n) = (LONG)((((ULONG)(p)[1] & (ULONG)WORDMASK) « BITSINWORD) | ((ULONG)(p)[0] & (ULONG)WORDMASK)); \
        (p) += 2; \
    } while (0)

#define CHECK(condition) _CHECK(condition, __FILE__, __LINE__)
#define _CHECK(condition, file, line) __CHECK(condition, file, line)
#define __CHECK(condition, file, line) \
    do { \
        if ( !(condition) ) { \
            Error0("Fatal error at " file ":" #line); \
            Terminate(-1); \
        } \
    } while (0)

/*
 * For debugging.
 */
#define DBGOUT(lv1, lv2, a) do { if ( lv1 >= lv2 ) { printf a; fflush(stdout); } } while (0)

/* (AN.ninterms of master) == max(AN.ninterms of slaves) == sum(PF_linterms of slaves) at EndSort().
*/
#define DBGOUT_NINTERMS(lv, a)
/* #define DBGOUT_NINTERMS(lv, a) DBGOUT(1, lv, a) */

/*
    #] includes :
    #[ statistics :
        #[ variables : (should be part of a struct?)
*/
static LONG PF_linterms;     /* local interms on this proces: PF_Proces */
#define PF_STATS_SIZE 5
static LONG **PF_stats = NULL;/* space for collecting statistics of all procs */
static LONG PF_laststat;     /* last realtime when statistics were printed */
static LONG PF_statsinterval;/* timeinterval for printing statistics */
/*
        #] variables :
        #[ PF_Statistics :
*/

static int PF_Statistics(LONG **stats, int proc)
{
    GETIDENTITY
    LONG real, cpu;
    WORD rpart, cpart;
    int i, j;

    if ( AT.SS == AM.S0 && PF.me == MASTER ) {
        real = PF_RealTime(PF_TIME); rpart = (WORD)(real%100); real /= 100;

        if ( PF_stats == NULL ) {
            PF_stats = (LONG**)Malloc1(PF.numtasks*sizeof(LONG*),"PF_stats 1");
            for ( i = 0; i < PF.numtasks; i++ ) {
                PF_stats[i] = (LONG*)Malloc1(PF_STATS_SIZE*sizeof(LONG),"PF_stats 2");
                for ( j = 0; j < PF_STATS_SIZE; j++ ) PF_stats[i][j] = 0;
            }
        }
        if ( proc > 0 ) for ( i = 0; i < PF_STATS_SIZE; i++ ) PF_stats[proc][i] = stats[0][i];

        if ( real >= PF_laststat + PF_statsinterval || proc == 0 ) {
            LONG sum[PF_STATS_SIZE];

            for ( i = 0; i < PF_STATS_SIZE; i++ ) sum[i] = 0;
            sum[0] = cpu = TimeCPU(1);
            cpart = (WORD)(cpu%1000);
            cpu /= 1000;
            cpart /= 10;
            if ( AC.OldParallelStats ) MesPrint("");
            if ( proc > 0 && AC.StatsFlag && AC.OldParallelStats ) {
```

```
                MesPrint("proc          CPU         in        gen       left       byte");
                MesPrint("%3d  : %7l.%2i %10l",0,cpu,cpart,AN.ninterms);
            }
            else if ( AC.StatsFlag && AC.OldParallelStats ) {
                MesPrint("proc          CPU         in        gen       out        byte");
                MesPrint("%3d  : %7l.%2i %10l %10l %10l",0,cpu,cpart,AN.ninterms,0,PF_goutterms);
            }

            for ( i = 1; i < PF.numtasks; i++ ) {
                cpart = (WORD)(PF_stats[i][0]%1000);
                cpu = PF_stats[i][0] / 1000;
                cpart /= 10;
                if ( AC.StatsFlag && AC.OldParallelStats )
                    MesPrint("%3d  : %7l.%2i %10l %10l %10l",i,cpu,cpart,
                            PF_stats[i][2],PF_stats[i][3],PF_stats[i][4]);
                for ( j = 0; j < PF_STATS_SIZE; j++ ) sum[j] += PF_stats[i][j];
            }
            cpart = (WORD)(sum[0]%1000);
            cpu = sum[0] / 1000;
            cpart /= 10;
            if ( AC.StatsFlag && AC.OldParallelStats ) {
                MesPrint("Sum  = %7l.%2i %10l %10l %10l",cpu,cpart,sum[2],sum[3],sum[4]);
                MesPrint("Real = %7l.%2i %20s (%l) %16s",
                        real,rpart,AC.Commercial,AC.CModule,EXPRNAME(AR.CurExpr));
                MesPrint("");
            }
            PF_laststat = real;
        }
    }
    return(0);
}
/*
        #] PF_Statistics :
    #] statistics :
    #[ sort.c :
        #[ sort variables :
*/

typedef struct NoDe {
    struct NoDe *left;
    struct NoDe *rght;
    int lloser;
    int rloser;
    int lsrc;
    int rsrc;
} NODE;

/*
    should/could be put in one struct
*/
static  NODE *PF_root;          /* root of tree of losers */
static  WORD PF_loser;          /* this is the last loser */
static  WORD **PF_term;         /* these point to the active terms */
static  WORD **PF_newcpos;      /* new coeffs of merged terms */
static  WORD *PF_newclen;       /* length of new coefficients */

/*
    preliminary: could also write somewhere else?
*/

static  WORD *PF_WorkSpace;     /* used in PF_EndSort() */
static  UWORD *PF_ScratchSpace; /* used in PF_GetLoser() */

/*
        #] sort variables :
        #[ PF_AllocBuf :
*/

static PF_BUFFER *PF_AllocBuf(int nbufs, LONG bsize, WORD free)
{
    PF_BUFFER *buf;
    UBYTE *p, *stop;
    LONG allocsize;
    int i;

    allocsize =
        (LONG)(sizeof(PF_BUFFER) + 4*nbufs*sizeof(WORD*) + (nbufs-free)*bsize);

    allocsize +=
        (LONG)( nbufs * (  2 * sizeof(MPI_Status)
                         +     sizeof(MPI_Request)
                         +     sizeof(MPI_Datatype)
                        )  );
    allocsize += (LONG)( nbufs * 3 * sizeof(int) );

    if ( ( buf = (PF_BUFFER*)Malloc1(allocsize,"PF_AllocBuf") ) == NULL ) return(NULL);

```

```
    p = ((UBYTE *)buf) + sizeof(PF_BUFFER);
    stop = ((UBYTE *)buf) + allocsize;

    buf->numbufs = nbufs;
    buf->active = 0;

    buf->buff    = (WORD**)p;         p += buf->numbufs*sizeof(WORD*);
    buf->fill    = (WORD**)p;         p += buf->numbufs*sizeof(WORD*);
    buf->full    = (WORD**)p;         p += buf->numbufs*sizeof(WORD*);
    buf->stop    = (WORD**)p;         p += buf->numbufs*sizeof(WORD*);
    buf->status  = (MPI_Status *)p;   p += buf->numbufs*sizeof(MPI_Status);
    buf->retstat = (MPI_Status *)p;   p += buf->numbufs*sizeof(MPI_Status);
    buf->request = (MPI_Request *)p;  p += buf->numbufs*sizeof(MPI_Request);
    buf->type    = (MPI_Datatype *)p; p += buf->numbufs*sizeof(MPI_Datatype);
    buf->index   = (int *)p;          p += buf->numbufs*sizeof(int);

    for ( i = 0; i < buf->numbufs; i++ ) buf->request[i] = MPI_REQUEST_NULL;
    buf->tag     = (int *)p;          p += buf->numbufs*sizeof(int);
    buf->from    = (int *)p;          p += buf->numbufs*sizeof(int);
/*
        and finally the real bufferspace
*/
    for ( i = free; i < buf->numbufs; i++ ) {
        buf->buff[i] = (WORD*)p; p += bsize;
        buf->stop[i] = (WORD*)p;
        buf->fill[i] = buf->full[i] = buf->buff[i];
    }
    if ( p != stop ) {
        MesPrint("Error in PF_AllocBuf p = %x stop = %x\n",p,stop);
        return(NULL);
    }
    return(buf);
}

/*
        #] PF_AllocBuf :
        #[ PF_InitTree :
*/

static int PF_InitTree(void)
{
    GETIDENTITY
    PF_BUFFER **rbuf = PF.rbufs;
    UBYTE *p, *stop;
    int numrbufs,numtasks = PF.numtasks;
    int i, j, src, numnodes;
    int numslaves = numtasks - 1;
    LONG size;
/*
        #[ the buffers : for the new coefficients and the terms
           we need one for each slave
*/
    if ( PF_term == NULL ) {
        size =  2*numtasks*sizeof(WORD*) + sizeof(WORD)*
            ( numtasks*(1 + AM.MaxTal) + (AM.MaxTer/sizeof(WORD)+1) + 2*(AM.MaxTal+2));

        PF_term = (WORD **)Malloc1(size,"PF_term");
        stop = ((UBYTE*)PF_term) + size;
        p = ((UBYTE*)PF_term) + numtasks*sizeof(WORD*);

        PF_newcpos = (WORD **)p;  p += sizeof(WORD*) * numtasks;
        PF_newclen =  (WORD *)p;  p += sizeof(WORD)  * numtasks;
        for ( i = 0; i < numtasks; i++ ) {
            PF_newcpos[i] = (WORD *)p; p += sizeof(WORD)*AM.MaxTal;
            PF_newclen[i] = 0;
        }
        PF_WorkSpace = (WORD *)p;    p += AM.MaxTer+sizeof(WORD);
        PF_ScratchSpace = (UWORD*)p; p += 2*(AM.MaxTal+2)*sizeof(UWORD);

        if ( p != stop ) { MesPrint("error in PF_InitTree"); return(-1); }
    }
/*
        #] the buffers :
        #[ the receive buffers :
*/
    numrbufs = PF.numrbufs;
/*
        this is the size we have in the combined sortbufs for one slave
*/
    size = (AT.SS->sTop2 - AT.SS->lBuffer - 1)/(PF.numtasks - 1);

    if ( rbuf == NULL ) {
        if ( ( rbuf = (PF_BUFFER**)Malloc1(numtasks*sizeof(PF_BUFFER*), "Master: rbufs") ) == NULL ) return(-1);
        if ( (rbuf[0] = PF_AllocBuf(1,0,1) ) == NULL ) return(-1);
        for ( i = 1; i < numtasks; i++ ) {
            if (!(rbuf[i] = PF_AllocBuf(numrbufs,sizeof(WORD)*size,1))) return(-1);
```

```
        }
    }
    rbuf[0]->buff[0] = AT.SS->lBuffer;
    rbuf[0]->full[0] = rbuf[0]->fill[0] = rbuf[0]->buff[0];
    rbuf[0]->stop[0] = rbuf[1]->buff[0] = rbuf[0]->buff[0] + 1;
    rbuf[1]->full[0] = rbuf[1]->fill[0] = rbuf[1]->buff[0];
    for ( i = 2; i < numtasks; i++ ) {
        rbuf[i-1]->stop[0] = rbuf[i]->buff[0] = rbuf[i-1]->buff[0] + size;
        rbuf[i]->full[0] = rbuf[i]->fill[0] = rbuf[i]->buff[0];
    }
    rbuf[numtasks-1]->stop[0] = rbuf[numtasks-1]->buff[0] + size;

    for ( i = 1; i < numtasks; i++ ) {
        for ( j = 0; j < rbuf[i]->numbufs; j++ ) {
            rbuf[i]->full[j] = rbuf[i]->fill[j] = rbuf[i]->buff[j] + AM.MaxTer/sizeof(WORD) + 2;
        }
        PF_term[i] = rbuf[i]->fill[rbuf[i]->active];
        *PF_term[i] = 0;
        PF_IRecvRbuf(rbuf[i],rbuf[i]->active,i);
    }
    rbuf[0]->active = 0;
    PF_term[0] = rbuf[0]->buff[0];
    PF_term[0][0] = 0;  /* PF_term[0] is used for a zero term. */
    PF.rbufs = rbuf;
/*
        #] the receive buffers :
        #[ the actual tree :

     calculate number of nodes in mergetree and allocate space for them
*/
    if ( numslaves < 3 ) numnodes = 1;
    else {
        numnodes = 2;
        while ( numnodes < numslaves ) numnodes *= 2;
        numnodes -= 1;
    }

    if ( PF_root == NULL )
    if ( ( PF_root = (NODE*)Malloc1(sizeof(NODE)*numnodes,"nodes in mergtree") ) == NULL )
        return(-1);
/*
        then initialize all the nodes
*/
    src = 1;
    for ( i = 0; i < numnodes; i++ ) {
        if ( 2*(i+1) <= numnodes ) {
            PF_root[i].left = &(PF_root[2*(i+1)-1]);
            PF_root[i].lsrc = 0;
        }
        else {
            PF_root[i].left = 0;
            if ( src < numtasks ) PF_root[i].lsrc = src++;
            else                  PF_root[i].lsrc = 0;
        }
        PF_root[i].lloser = 0;
    }
    for ( i = 0; i < numnodes; i++ ) {
        if ( 2*(i+1)+1 <= numnodes ) {
            PF_root[i].rght = &(PF_root[2*(i+1)]);
            PF_root[i].rsrc = 0;
        }
        else {
            PF_root[i].rght = 0;
            if (src<numtasks) PF_root[i].rsrc = src++;
            else              PF_root[i].rsrc = 0;
        }
        PF_root[i].rloser = 0;
    }
/*
        #] the actual tree :
*/
    return(numnodes);
}

/*
        #] PF_InitTree :
        #[ PF_PutIn :
*/

static WORD *PF_PutIn(int src)
{
    int tag;
    WORD im, r;
    WORD *m1, *m2;
    LONG size;
    PF_BUFFER *rbuf = PF.rbufs[src];
    int a = rbuf->active;
```

```
    int next = a+1 >= rbuf->numbufs ? 0 : a+1 ;
    WORD *lastterm = PF_term[src];
    WORD *term = rbuf->fill[a];

    if ( src <= 0 ) return(PF_term[0]);

    if ( rbuf->full[a] == rbuf->buff[a] + AM.MaxTer/sizeof(WORD) + 2 ) {
/*
            very first term from this src
*/
        tag = PF_WaitRbuf(rbuf,a,&size);
        rbuf->full[a] += size;
        if ( tag == PF_ENDBUFFER_MSGTAG ) *rbuf->full[a]++ = 0;
        else if ( rbuf->numbufs > 1 ) {
/*
                post a nonblock. recv. for the next buffer
*/
            rbuf->full[next] = rbuf->buff[next] + AM.MaxTer/sizeof(WORD) + 2;
            size = (LONG)(rbuf->stop[next] - rbuf->full[next]);
            PF_IRecvRbuf(rbuf,next,src);
        }
    }
    if ( *term == 0 && term != rbuf->full[a] ) return(PF_term[0]);
/*
        exception is for rare cases when the terms fitted exactly into buffer
*/
    if ( term + *term > rbuf->full[a] || term + 1 >= rbuf->full[a] ) {
newterms:
        m1 = rbuf->buff[next] + AM.MaxTer/sizeof(WORD) + 1;
        if ( *term < 0 || term == rbuf->full[a] ) {
/*
            copy term and lastterm to the new buffer, so that they end at m1
*/
            m2 = rbuf->full[a] - 1;
            while ( m2 >= term ) *m1-- = *m2--;
            rbuf->fill[next] = term = m1 + 1;
            m2 = lastterm + *lastterm - 1;
            while ( m2 >= lastterm ) *m1-- = *m2--;
            lastterm = m1 + 1;
        }
        else {
/*
            copy beginning of term to the next buffer so that it ends at m1
*/
            m2 = rbuf->full[a] - 1;
            while ( m2 >= term ) *m1-- = *m2--;
            rbuf->fill[next] = term = m1 + 1;
        }
        if ( rbuf->numbufs == 1 ) {
            rbuf->full[a] = rbuf->buff[a] + AM.MaxTer/sizeof(WORD) + 2;
            size = (LONG)(rbuf->stop[a] - rbuf->full[a]);
            PF_IRecvRbuf(rbuf,a,src);
        }
/*
            wait for new terms in the next buffer
*/
        rbuf->full[next] = rbuf->buff[next] + AM.MaxTer/sizeof(WORD) + 2;
        tag = PF_WaitRbuf(rbuf,next,&size);
        rbuf->full[next] += size;
        if ( tag == PF_ENDBUFFER_MSGTAG ) {
            *rbuf->full[next]++ = 0;
        }
        else if ( rbuf->numbufs > 1 ) {
/*
            post a nonblock. recv. for active buffer, it is not needed anymore
*/
            rbuf->full[a] = rbuf->buff[a] + AM.MaxTer/sizeof(WORD) + 2;
            size = (LONG)(rbuf->stop[a] - rbuf->full[a]);
            PF_IRecvRbuf(rbuf,a,src);
        }
/*
            now savely make next buffer active
*/
        a = rbuf->active = next;
    }

    if ( *term < 0 ) {
/*
            We need to decompress the term
*/
        im = *term;
        r = term[1] - im + 1;
        m1 = term + 2;
        m2 = lastterm - im + 1;
        while ( ++im <= 0 ) *--m1 = *--m2;
        *--m1 = r;
        rbuf->fill[a] = term = m1;
```

```
        if ( term + *term > rbuf->full[a] ) goto newterms;
    }
    rbuf->fill[a] += *term;
    return(term);
}

/*
        #] PF_PutIn :
        #[ PF_GetLoser :
*/

static int PF_GetLoser(NODE *n)
{
    GETIDENTITY
    WORD comp;

    if ( PF_loser == 0 ) {
/*
            this is for the right initialization of the tree only
*/
        if ( n->left ) n->lloser = PF_GetLoser(n->left);
        else {
            n->lloser = n->lsrc;
            if ( *(PF_term[n->lsrc] = PF_PutIn(n->lsrc)) == 0) n->lloser = 0;
        }
        PF_loser = 0;
        if ( n->rght ) n->rloser = PF_GetLoser(n->rght);
        else{
            n->rloser = n->rsrc;
            if ( *(PF_term[n->rsrc] = PF_PutIn(n->rsrc)) == 0 ) n->rloser = 0;
        }
        PF_loser = 0;
    }
    else if ( PF_loser == n->lloser ) {
        if ( n->left ) n->lloser = PF_GetLoser(n->left);
        else {
            n->lloser = n->lsrc;
            if ( *(PF_term[n->lsrc] = PF_PutIn(n->lsrc)) == 0 ) n->lloser = 0;
        }
    }
    else if ( PF_loser == n->rloser ) {
newright:
        if ( n->rght ) n->rloser = PF_GetLoser(n->rght);
        else {
            n->rloser = n->rsrc;
            if ( *(PF_term[n->rsrc] = PF_PutIn(n->rsrc)) == 0 ) n->rloser = 0;
        }
    }
    if ( n->lloser > 0 && n->rloser > 0 ) {
        comp = CompareTerms(BHEAD PF_term[n->lloser],PF_term[n->rloser],(WORD)0);
        if ( comp > 0 )     return(n->lloser);
        else if (comp < 0 ) return(n->rloser);
        else {
/*
        #[ terms are equal :
*/
            WORD *lcpos, *rcpos;
            UWORD *newcpos;
            WORD lclen, rclen, newclen, newnlen;
            SORTING *S = AT.SS;

            if ( S->PolyWise ) {
/*
            #[ Here we work with PolyFun :
*/
                WORD *tt1, *w;
                WORD r1,r2;
                WORD *ml = PF_term[n->lloser];
                WORD *mr = PF_term[n->rloser];

                if ( ( r1 = (int)*PF_term[n->lloser] ) <= 0 ) r1 = 20;
                if ( ( r2 = (int)*PF_term[n->rloser] ) <= 0 ) r2 = 20;
                tt1 = ml;
                ml += S->PolyWise;
                mr += S->PolyWise;
                if ( S->PolyFlag == 2 ) {
                    w = poly_ratfun_add(BHEAD ml,mr);
                    if ( *tt1 + w[1] - ml[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
                        MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l is too small",AM.MaxTer);
                        Terminate(-1);
                    }
                    AT.WorkPointer = w;
                }
                else {
                    w = AT.WorkPointer;
                    if ( w + ml[1] + mr[1] > AT.WorkTop ) {
```

```
                        MesPrint("A WorkSpace of %10l is too small",AM.WorkSize);
                        Terminate(-1);
                    }
                    AddArgs(BHEAD ml,mr,w);
                }
                r1 = w[1];
                if ( r1 <= FUNHEAD || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) ) {
                    goto cancelled;
                }
                if ( r1 == ml[1] ) {
                    NCOPY(ml,w,r1);
                }
                else if ( r1 < ml[1] ) {
                    r2 = ml[1] - r1;
                    mr = w + r1;
                    ml += ml[1];
                    while ( --r1 >= 0 ) *--ml = *--mr;
                    mr = ml - r2;
                    r1 = S->PolyWise;
                    while ( --r1 >= 0 ) *--ml = *--mr;
                    *ml -= r2;
                    PF_term[n->lloser] = ml;
                }
                else {
                    r2 = r1 - ml[1];
                    if ( r2 > 2*AM.MaxTal )
                        MesPrint("warning: new term in polyfun is large");
                    mr = tt1 - r2;
                    r1 = S->PolyWise;
                    ml = tt1;
                    *ml += r2;
                    PF_term[n->lloser] = mr;
                    NCOPY(mr,ml,r1);
                    r1 = w[1];
                    NCOPY(mr,w,r1);
                }
                PF_newclen[n->rloser] = 0;
                PF_loser = n->rloser;
                goto newright;
/*
            #] Here we work with PolyFun :
*/
            }
            if ( ( lclen = PF_newclen[n->lloser] ) != 0 ) lcpos = PF_newcpos[n->lloser];
            else {
                lcpos = PF_term[n->lloser];
                lclen = *(lcpos += *lcpos - 1);
                lcpos -= ABS(lclen) - 1;
            }
            if ( ( rclen = PF_newclen[n->rloser] ) != 0 ) rcpos = PF_newcpos[n->rloser];
            else {
                rcpos = PF_term[n->rloser];
                rclen = *(rcpos += *rcpos - 1);
                rcpos -= ABS(rclen) -1;
            }
            lclen = ( (lclen > 0) ? (lclen-1) : (lclen+1) ) » 1;
            rclen = ( (rclen > 0) ? (rclen-1) : (rclen+1) ) » 1;
            newcpos = PF_ScratchSpace;
            if ( AddRat(BHEAD (UWORD *)lcpos,lclen,(UWORD *)rcpos,rclen,newcpos,&newnlen) )
    return(-1);
            if ( AN.ncmod != 0 ) {
                if ( ( AC.modmode & POSNEG ) != 0 ) {
                    NormalModulus(newcpos,&newnlen);
                }
                if ( BigLong(newcpos,newnlen,(UWORD *)AC.cmod,ABS(AN.ncmod)) >=0 ) {
                    WORD ii;
                    SubPLon(newcpos,newnlen,(UWORD *)AC.cmod,ABS(AN.ncmod),newcpos,&newnlen);
                    newcpos[newnlen] = 1;
                    for ( ii = 1; ii < newnlen; ii++ ) newcpos[newnlen+ii] = 0;
                }
            }
            if ( newnlen == 0 ) {
/*
                    terms cancel, get loser of left subtree and then of right subtree
*/
cancelled:
                PF_loser = n->lloser;
                PF_newclen[n->lloser] = 0;
                if ( n->left ) n->lloser = PF_GetLoser(n->left);
                else {
                    n->lloser = n->lsrc;
                    if ( *(PF_term[n->lsrc] = PF_PutIn(n->lsrc)) == 0 ) n->lloser = 0;
                }
                PF_loser = n->rloser;
                PF_newclen[n->rloser] = 0;
                goto newright;
            }
```

```
            else {
/*
                    keep the left term and get the loser of right subtree
*/
                newnlen «= 1;
                newclen = ( newnlen > 0 ) ? ( newnlen + 1 ) : ( newnlen - 1 );
                if ( newnlen < 0 ) newnlen = -newnlen;
                PF_newclen[n->lloser] = newclen;
                lcpos = PF_newcpos[n->lloser];
                if ( newclen < 0 ) newclen = -newclen;
                while ( newclen-- ) *lcpos++ = *newcpos++;
                PF_loser = n->rloser;
                PF_newclen[n->rloser] = 0;
                goto newright;
            }
/*
        #] terms are equal :
*/
        }
    }
    if (n->lloser > 0) return(n->lloser);
    if (n->rloser > 0) return(n->rloser);
    return(0);
}
/*
        #] PF_GetLoser :
        #[ PF_EndSort :
*/

int PF_EndSort(void)
{
    GETIDENTITY
    FILEHANDLE *fout = AR.outfile;
    PF_BUFFER *sbuf=PF.sbuf;
    SORTING *S = AT.SS;
    WORD *outterm,*pp;
    LONG size, noutterms;
    POSITION position, oldposition;
    WORD i,cc;
    int oldgzipCompress;

    if ( AT.SS != AT.S0 || !PF.parallel ) return 0;

    if ( PF.me != MASTER ) {
/*
        #[ the slaves have to initialize their sendbuffer :

        this is a slave and it's PObuffer should be the minimum of the
        sortiosize on the master and the POsize of our file.
        First save the original PObuffer and POstop of the outfile
*/
        size = (S->sTop2 - S->lBuffer - 1)/(PF.numtasks - 1);
        size -= (AM.MaxTer/sizeof(WORD) + 2);
        if ( fout->POsize < (LONG)(size*sizeof(WORD)) ) size = fout->POsize/sizeof(WORD);
        if ( sbuf == NULL ) {
            if ( (sbuf = PF_AllocBuf(PF.numsbufs, size*sizeof(WORD), 1)) == NULL ) return -1;
            sbuf->active = 0;
            PF.sbuf = sbuf;
        }
        sbuf->buff[0] = fout->PObuffer;
        sbuf->stop[0] = fout->PObuffer+size;
        if ( sbuf->stop[0] > fout->POstop ) return -1;
        for ( i = 0; i < PF.numsbufs; i++ )
            sbuf->fill[i] = sbuf->full[i] = sbuf->buff[i];

        fout->PObuffer = sbuf->buff[sbuf->active];
        fout->POstop = sbuf->stop[sbuf->active];
        fout->POsize = size*sizeof(WORD);
        fout->POfill = fout->POfull = fout->PObuffer;
/*
        #] the slaves have to initialize their sendbuffer :
*/
        return(0);
    }
/*
        this waits for all slaves to be ready to send terms back
*/
    PF_WaitAllSlaves(); /* Note, the returned value should be 0 on success. */
/*
        Now collect the terms of all slaves and merge them.
        PF_GetLoser gives the position of the smallest term, which is the real
        work. The smallest term needs to be copied to the outbuf: use PutOut.
*/
    PF_InitTree();
    if ( AR.PolyFun == 0 ) { S->PolyFlag = 0; }
    else if ( AR.PolyFunType == 1 ) { S->PolyFlag = 1; }
    else if ( AR.PolyFunType == 2 ) {
```

```
        if ( AR.PolyFunExp == 2
          || AR.PolyFunExp == 3 ) S->PolyFlag = 1;
        else                      S->PolyFlag = 2;
    }
    *AR.CompressPointer = 0;
    SeekScratch(fout, &position);
    oldposition = position;
    oldgzipCompress = AR.gzipCompress;
    AR.gzipCompress = 0;

    noutterms = 0;

    while ( PF_loser >= 0 ) {
        if ( (PF_loser = PF_GetLoser(PF_root)) == 0 ) break;
        outterm = PF_term[PF_loser];
        noutterms++;

        if ( PF_newclen[PF_loser] != 0 ) {
/*
            #[ this is only when new coeff was too long :
*/
            outterm = PF_WorkSpace;
            pp = PF_term[PF_loser];
            cc = *pp;
            while ( cc-- ) *outterm++ = *pp++;
            outterm = (outterm[-1] > 0) ? outterm-outterm[-1] : outterm+outterm[-1];
            if ( PF_newclen[PF_loser] > 0 ) cc =  (WORD)PF_newclen[PF_loser] - 1;
            else                            cc = -(WORD)PF_newclen[PF_loser] - 1;
            pp =  PF_newcpos[PF_loser];
            while ( cc-- ) *outterm++ = *pp++;
            *outterm++ = PF_newclen[PF_loser];
            *PF_WorkSpace = outterm - PF_WorkSpace;
            outterm = PF_WorkSpace;
            *PF_newcpos[PF_loser] = 0;
            PF_newclen[PF_loser] = 0;
/*
            #] this is only when new coeff was too long :
*/
        }
        PRINTFBUF("PF_EndSort to PutOut: ",outterm,*outterm);
        PutOut(BHEAD outterm,&position,fout,1);
    }
    if ( FlushOut(&position,fout,0) ) {
        AR.gzipCompress = oldgzipCompress;
        return(-1);
    }
    S->TermsLeft = PF_goutterms = noutterms;
    DIFPOS(PF_exprsize, position, oldposition);
    AR.gzipCompress = oldgzipCompress;
    return(1);
}

/*
        #] PF_EndSort :
    #] sort.c :
    #[ proces.c :
        #[ variables :
*/

static  WORD *PF_CurrentBracket;

/*
        #] variables :
        #[ PF_GetTerm :
*/

static WORD PF_GetTerm(WORD *term)
{
    GETIDENTITY
    FILEHANDLE *fi = AC.RhsExprInModuleFlag && PF.rhsInParallel ? &PF.slavebuf : AR.infile;
    WORD i;
    WORD *next, *np, *last, *lp = 0, *nextstop, *tp=term;

    /* Only on the slaves. */

    AN.deferskipped = 0;
    if ( fi->POfill >= fi->POfull || fi->POfull == fi->PObuffer ) {
ReceiveNew:
      {
/*
        #[ receive new terms from master :
*/
        int src = MASTER, tag;
        int follow = 0;
        LONG size,cpu,space = 0;

        if ( PF.log ) {
```

```
            fprintf(stderr,"[%d] Starting to send to Master\n",PF.me);
            fflush(stderr);
        }

        cpu = TimeCPU(1);
        PF_PreparePack();
        PF_Pack(&cpu               ,1,PF_LONG);
        PF_Pack(&space             ,1,PF_LONG);
        PF_Pack(&PF_linterms       ,1,PF_LONG);
        PF_Pack(&(AM.S0->GenTerms) ,1,PF_LONG);
        PF_Pack(&(AM.S0->TermsLeft),1,PF_LONG);
        PF_Pack(&follow            ,1,PF_INT );

        if ( PF.log ) {
            fprintf(stderr,"[%d] Now sending with tag = %d\n",PF.me,PF_READY_MSGTAG);
            fflush(stderr);
        }

        PF_Send(MASTER, PF_READY_MSGTAG);

        if ( PF.log ) {
            fprintf(stderr,"[%d] returning from send\n",PF.me);
            fflush(stderr);
        }

        size = fi->POstop - fi->PObuffer - 1;
#ifdef AbsolutelyExtra
        PF_Receive(MASTER,PF_ANY_MSGTAG,&src,&tag);
#ifdef MPI2
        if ( tag == PF_TERM_MSGTAG ) {
            PF_Unpack(&size, 1, PF_LONG);
            if ( PF_Put_target(src) == 0 ) {
                printf("PF_Put_target error ...\n");
            }
        }
        else {
            PF_RecvWbuf(fi->PObuffer,&size,&src);
        }
#else
        PF_RecvWbuf(fi->PObuffer,&size,&src);
#endif
#endif
        tag=PF_RecvWbuf(fi->PObuffer,&size,&src);

        fi->POfill = fi->PObuffer;
        /* Get AN.ninterms which sits in the first 2 WORDs. */
        {
            LONG ninterms;
            UNPACK_LONG(fi->POfill, ninterms);
            if ( fi->POfill < fi->POfull ) {
                DBGOUT_NINTERMS(2, ("PF.me=%d AN.ninterms=%d PF_linterms=%d ninterms=%d GET\n",
(int)PF.me, (int)AN.ninterms, (int)PF_linterms, (int)ninterms));
                AN.ninterms = ninterms - 1;
            } else {
                DBGOUT_NINTERMS(2, ("PF.me=%d AN.ninterms=%d PF_linterms=%d ninterms=%d GETEND\n",
(int)PF.me, (int)AN.ninterms, (int)PF_linterms, (int)ninterms));
            }
        }
        fi->POfull = fi->PObuffer + size;
        if ( tag == PF_ENDSORT_MSGTAG ) *fi->POfull++ = 0;
/*
        #] receive new terms from master :
*/
      }
      if ( PF_CurrentBracket ) *PF_CurrentBracket = 0;
    }
    if ( *fi->POfill == 0 ) {
        fi->POfill = fi->POfull = fi->PObuffer;
        *term = 0;
        goto RegRet;
    }
    if ( AR.DeferFlag ) {
        if ( !PF_CurrentBracket ) {
/*
        #[ alloc space :
*/
            PF_CurrentBracket =
                    (WORD*)Malloc1(AM.MaxTer,"PF_CurrentBracket");
            *PF_CurrentBracket = 0;
/*
        #] alloc space :
*/
        }
        while ( *PF_CurrentBracket ) {  /* "for each term in the buffer" */
/*
        #[ test : bracket & skip if it's equal to the last in PF_CurrentBracket
*/
```

```
            next = fi->POfill;
            nextstop = next + *next; nextstop -= ABS(nextstop[-1]);
            next++;
            last = PF_CurrentBracket+1;
            while ( next < nextstop ) {
/*
                    scan the next term and PF_CurrentBracket
*/
                if ( *last == HAAKJE && *next == HAAKJE ) {
/*
                    the part outside brackets is equal => skip this term
*/
                    PRINTFBUF("PF_GetTerm skips",fi->POfill,*fi->POfill);
                    break;
                }
/*
                    check if the current subterms are equal
*/
                np = next; next += next[1];
                lp = last; last += last[1];
                while ( np < next ) if ( *lp++ != *np++ ) goto strip;
            }
/*
                go on to next term
*/
            fi->POfill += *fi->POfill;
            AN.deferskipped++;
/*
                the usual checks
*/
            if ( fi->POfill >= fi->POfull || fi->POfull == fi->PObuffer )
                goto ReceiveNew;
            if ( *fi->POfill == 0 ) {
                fi->POfill = fi->POfull = fi->PObuffer;
                *term = 0;
                goto RegRet;
            }
/*
        #] test :
*/
        }
/*
        #[ copy :

        this term to CurrentBracket and the part outside of bracket
        to WorkSpace at term
*/
strip:
        next = fi->POfill;
        nextstop = next + *next; nextstop -= ABS(nextstop[-1]);
        next++;
        tp++;
        lp = PF_CurrentBracket + 1;
        while ( next < nextstop ) {
            if ( *next == HAAKJE ) {
                fi->POfill += *fi->POfill;
                while ( next < fi->POfill ) *lp++ = *next++;
                *PF_CurrentBracket = lp - PF_CurrentBracket;
                *lp = 0;
                *tp++ = 1;
                *tp++ = 1;
                *tp++ = 3;
                *term = WORDDIF(tp,term);
                PRINTFBUF("PF_GetTerm new brack",PF_CurrentBracket,*PF_CurrentBracket);
                PRINTFBUF("PF_GetTerm POfill",fi->POfill,*fi->POfill);
                goto RegRet;
            }
            np = next; next += next[1];
            while ( np < next ) *tp++ = *lp++ = *np++;
        }
        tp = term;
/*
        #] copy :
*/
    }

    i = *fi->POfill;
    while ( i-- ) *tp++ = *fi->POfill++;
RegRet:
    PRINTFBUF("PF_GetTerm returns",term,*term);
    return(*term);
}

/*
        #] PF_GetTerm :
        #[ PF_Deferred :
*/
```

```
WORD PF_Deferred(WORD *term, WORD level)
{
    GETIDENTITY
    WORD *bra, *bstop;
    WORD *tstart;
    FILEHANDLE *fi = AC.RhsExprInModuleFlag && PF.rhsInParallel ? &PF.slavebuf : AR.infile;
    WORD *next = fi->POfill;
    WORD *termout = AT.WorkPointer;
    WORD *oldwork = AT.WorkPointer;

    AT.WorkPointer = (WORD *)((UBYTE *)(AT.WorkPointer) + AM.MaxTer);
    AR.DeferFlag = 0;

    PRINTFBUF("PF_Deferred (Term)   ",term,*term);
    PRINTFBUF("PF_Deferred (Bracket)",PF_CurrentBracket,*PF_CurrentBracket);

    bra = bstop = PF_CurrentBracket;
    if ( *bstop > 0 ) {
        bstop += *bstop;
        bstop -= ABS(bstop[-1]);
    }
    bra++;
    while ( *bra != HAAKJE && bra < bstop ) bra += bra[1];
    if ( bra >= bstop ) {   /* No deferred action! */
        AT.WorkPointer = term + *term;
        if ( Generator(BHEAD term,level) ) goto DefCall;
        AR.DeferFlag = 1;
        AT.WorkPointer = oldwork;
        return(0);
    }
    bstop = bra;
    tstart = bra + bra[1];
    bra = PF_CurrentBracket;
    tstart--;
    *tstart = bra + *bra - tstart;
    bra++;
/*
        Status of affairs:
        First bracket content starts at tstart.
        Next term starts at next.
        The outside of the bracket runs from bra = PF_CurrentBracket to bstop.
*/
    for(;;) {
        if ( InsertTerm(BHEAD term,0,AM.rbufnum,tstart,termout,0) < 0 ) {
            goto DefCall;
        }
/*
            call Generator with new composed term
*/
        AT.WorkPointer = termout + *termout;
        if ( Generator(BHEAD termout,level) ) goto DefCall;
        AT.WorkPointer = termout;
        tstart = next + 1;
        if ( tstart >= fi->POfull ) goto ThatsIt;
        next += *next;
/*
            compare with current bracket
*/
        while ( bra <= bstop ) {
            if ( *bra != *tstart ) goto ThatsIt;
            bra++; tstart++;
        }
/*
            now bra and tstart should both be a HAAKJE
*/
        bra--; tstart--;
        if ( *bra != HAAKJE || *tstart != HAAKJE ) goto ThatsIt;
        tstart += tstart[1];
        tstart--;
        *tstart = next - tstart;
        bra = PF_CurrentBracket + 1;
    }

ThatsIt:
/*
    AT.WorkPointer = oldwork;
*/
    AR.DeferFlag = 1;
    return(0);
DefCall:
    MesCall("PF_Deferred");
    SETERROR(-1);
}

/*
        #] PF_Deferred :
```

```
        #[ PF_Wait4Slave :
*/

static LONG **PF_W4Sstats = 0;

static int PF_Wait4Slave(int src)
{
    int j, tag, next;

    tag = PF_ANY_MSGTAG;
    PF_CatchErrorMessages(&src, &tag);
    PF_Receive(src, tag, &next, &tag);

    if ( tag != PF_READY_MSGTAG ) {
        MesPrint("[%d] PF_Wait4Slave: received MSGTAG %d",(WORD)PF.me,(WORD)tag);
        return(-1);
    }
    if ( PF_W4Sstats == 0 ) {
        PF_W4Sstats = (LONG**)Malloc1(sizeof(LONG*),"");
        PF_W4Sstats[0] = (LONG*)Malloc1(PF_STATS_SIZE*sizeof(LONG),"");
    }
    PF_Unpack(PF_W4Sstats[0],PF_STATS_SIZE,PF_LONG);
    PF_Statistics(PF_W4Sstats,next);

    PF_Unpack(&j,1,PF_INT);

    if ( j ) {
/*
        actions depending on rest of information in last message
*/
    }
    return(next);
}

/*
        #] PF_Wait4Slave :
        #[ PF_Wait4SlaveIP :
*/
/*
    array of expression numbers for PF_InParallel processor.
    Each time the master sends expression "i" to the slave
    "next" it sets partodoexr[next]=i:
*/
static WORD *partodoexr=NULL;

static int PF_Wait4SlaveIP(int *src)
{
    int j,tag,next;

    tag = PF_ANY_MSGTAG;
    PF_CatchErrorMessages(src, &tag);
    PF_Receive(*src, tag, &next, &tag);
    *src=tag;
    if ( PF_W4Sstats == 0 ) {
        PF_W4Sstats = (LONG**)Malloc1(sizeof(LONG*),"");
        PF_W4Sstats[0] = (LONG*)Malloc1(PF_STATS_SIZE*sizeof(LONG),"");
    }

    PF_Unpack(PF_W4Sstats[0],PF_STATS_SIZE,PF_LONG);
    if ( tag == PF_DATA_MSGTAG )
        AR.CurExpr = partodoexr[next];
    PF_Statistics(PF_W4Sstats,next);

    PF_Unpack(&j,1,PF_INT);

    if ( j ) {
    /* actions depending on rest of information in last message */
    }

    return(next);
}
/*
        #] PF_Wait4SlaveIP :
        #[ PF_WaitAllSlaves :
*/

static int PF_WaitAllSlaves(void)
{
    int i, readySlaves, tag, next = PF_ANY_SOURCE;
    UBYTE *has_sent = 0;

    has_sent = (UBYTE*)Malloc1(sizeof(UBYTE)*(PF.numtasks + 1),"PF_WaitAllSlaves");
    for ( i = 0; i < PF.numtasks; i++ ) has_sent[i] = 0;

    for ( readySlaves = 1; readySlaves < PF.numtasks; ) {
        if ( next != PF_ANY_SOURCE) { /*Go to the next slave:*/
            do{ /*Note, here readySlaves<PF.numtasks, so this loop can't be infinite*/
```

```
                if ( ++next >= PF.numtasks ) next = 1;
            } while ( has_sent[next] == 1 );
        }
/*
            Here PF_ProbeWithCatchingErrorMessages() is BLOCKING function if next = PF_ANY_SOURCE:
*/
        tag = PF_ProbeWithCatchingErrorMessages(&next);
/*
            Here next != PF_ANY_SOURCE
*/
        switch ( tag ) {
            case PF_BUFFER_MSGTAG:
            case PF_ENDBUFFER_MSGTAG:
/*
                    Slaves are ready to send their results back
*/
                if ( has_sent[next] == 0 ) {
                    has_sent[next] = 1;
                    readySlaves++;
                }
                else {  /*error?*/
                    fprintf(stderr,"ERROR next=%d tag=%d\n",next,tag);
                }
/*
                    Note, we do NOT read results here! Messages from these slaves will be read
                    only after all slaves are ready, further in caller function
*/
                break;
            case 0:
/*
                    The slave is not ready. Just go to the next slave.
                    It may appear that there are no more ready slaves, and the master
                    will wait them in infinite loop. Stupid situation - the master can
                    receive buffers from ready slaves!
*/
#ifdef PF_WITH_SCHED_YIELD
/*
                        Relinquish the processor:
*/
                    sched_yield();
#endif
                break;
            case PF_DATA_MSGTAG:
                tag=next;
                next=PF_Wait4SlaveIP(&tag);
/*
    tag must be == PF_DATA_MSGTAG!
*/
                PF_Statistics(PF_stats,0);
                PF_Slave2MasterIP(next);
                PF_Master2SlaveIP(next,NULL);
                if ( has_sent[next] == 0 ) {
                    has_sent[next]=1;
                    readySlaves++;
                }else{
                    /*error?*/
                    fprintf(stderr,"ERROR next=%d tag=%d\n",next,tag);
                }/*if ( has_sent[next] == 0 )*/
                break;
            case PF_EMPTY_MSGTAG:
                tag=next;
                next=PF_Wait4SlaveIP(&tag);
/*
    tag must be == PF_EMPTY_MSGTAG!
*/
                PF_Master2SlaveIP(next,NULL);
                if ( has_sent[next] == 0 ) {
                    has_sent[next]=1;
                    readySlaves++;
                }else{
                    /*error?*/
                    fprintf(stderr,"ERROR next=%d tag=%d\n",next,tag);
                }/*if ( has_sent[next] == 0 )*/
                break;
            case PF_READY_MSGTAG:
/*
                    idle slave
                    May be only PF_READY_MSGTAG:
*/
                next = PF_Wait4Slave(next);
                if ( next == -1 ) return(next); /*Cannot be!*/
                if ( has_sent[0] == 0 ) {  /*Send the last chunk to the slave*/
                    PF.sbuf->active = 0;
                    has_sent[0] = 1;
                }
                else {
/*
```

```
                        Last chunk was sent, so just send to slave ENDSORT
                        AN.ninterms must be sent because the slave expects it:
*/
                    PACK_LONG(PF.sbuf->fill[next], AN.ninterms);
/*
                        This will tell to the slave that there are no more terms:
*/
                    *(PF.sbuf->fill[next])++ = 0;
                    PF.sbuf->active = next;
                }
/*
                    Send ENDSORT
*/
                PF_ISendSbuf(next,PF_ENDSORT_MSGTAG);
                break;
            default:
/*
                    Error?
                    Indicates the error. This will force exit from the main loop:
*/
                MesPrint("!!!Unexpected MPI message src=%d tag=%d.", next, tag);
                readySlaves = PF.numtasks+1;
                break;
        }
    }

    if ( has_sent ) M_free(has_sent,"PF_WaitAllSlaves");
/*
        0 on sucess (exit from the main loop by loop condition), or -1 if fails
        (exit from the main loop since readySlaves=PF.numtasks+1):
*/
    return(PF.numtasks-readySlaves);
}

/*
        #] PF_WaitAllSlaves :
        #[ PF_Processor :
*/

int PF_Processor(EXPRESSIONS e, WORD i, WORD LastExpression)
{
    GETIDENTITY
    WORD *term = AT.WorkPointer;
    LONG dd = 0;
    PF_BUFFER *sb = PF.sbuf;
    WORD j, *s, next;
    LONG size, cpu;
    POSITION position;
    int k, src, tag;
    FILEHANDLE *oldoutfile = AR.outfile;

#ifdef MPI2
    if ( PF_shared_buff == NULL ) {
        if ( PF_SMWin_Init() == 0 ) {
            MesPrint("PF_SMWin_Init error");
            exit(-1);
        }
    }
#endif

    if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer ) ) > AT.WorkTop ) return(MesWork());

    /* For redefine statements. */
    if ( AC.numpfirstnum > 0 ) {
        for ( j = 0; j < AC.numpfirstnum; j++ ) {
            AC.inputnumbers[j] = -1;
        }
    }

    if ( AC.mparallelflag != PARALLELFLAG ) return(0);

    if ( PF.me == MASTER ) {
/*
        #[ Master:
            #[ write prototype to outfile:
*/
        WORD oldBracketOn = AR.BracketOn;
        WORD *oldBrackBuf = AT.BrackBuf;
        WORD oldbracketindexflag = AT.bracketindexflag;

        LONG maxinterms;     /* the maximum number of terms in the bucket */
        int cmaxinterms;     /* a variable controling the transition of maxinterms */
        LONG termsinbucket;  /* the number of filled terms in the bucket */
        LONG ProcessBucketSize = AC.mProcessBucketSize;

        if ( PF.log && AC.CModule >= PF.log )
            MesPrint("[%d] working on expression %s in module %l",PF.me,EXPRNAME(i),AC.CModule);
```

```
        if ( GetTerm(BHEAD term) <= 0 ) {
            MesPrint("[%d] Expression %d has problems in scratchfile",PF.me,i);
            return(-1);
        }
        term[3] = i;
        if ( AR.outtohide ) {
            SeekScratch(AR.hidefile,&position);
            e->onfile = position;
            if ( PutOut(BHEAD term,&position,AR.hidefile,0) < 0 ) return(-1);
        }
        else {
            SeekScratch(AR.outfile,&position);
            e->onfile = position;
            if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) return(-1);
        }
        AR.DeferFlag = 0;  /* The master leave the brackets!!! */
        AR.Eside = RHSIDE;
        if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
            AR.BracketOn = 1;
            AT.BrackBuf = AM.BracketFactors;
            AT.bracketindexflag = 1;
        }
        if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
/*
            #] write prototype to outfile:
            #[ initialize sendbuffer if necessary:

            the size of the sendbufs is:
            MIN(1/PF.numtasks*(AT.SS->sBufsize+AT.SS->lBufsize),AR.infile->POsize)
            No allocation for extra buffers necessary, just make sb->buf... point
            to the right places in the sortbuffers.
*/
        NewSort(BHEAD0);   /* we need AT.SS to be set for this!!! */
        if ( sb == 0 || sb->buff[0] != AT.SS->lBuffer ) {
            size = (LONG)((AT.SS->sTop2 - AT.SS->lBuffer)/(PF.numtasks));
            if ( size > (LONG)(AR.infile->POsize/sizeof(WORD) - 1) )
                size = AR.infile->POsize/sizeof(WORD) - 1;
            if ( sb == 0 ) {
                if ( ( sb = PF_AllocBuf(PF.numtasks,size*sizeof(WORD),PF.numtasks) ) == NULL )
                    return(-1);
            }
            sb->buff[0] = AT.SS->lBuffer;
            sb->full[0] = sb->fill[0] = sb->buff[0];
            for ( j = 1; j < PF.numtasks; j++ ) {
                sb->stop[j-1] = sb->buff[j] = sb->buff[j-1] + size;
            }
            sb->stop[PF.numtasks-1] = sb->buff[PF.numtasks-1] + size;
            PF.sbuf = sb;
        }
        for ( j = 0; j < PF.numtasks; j++ ) {
            sb->full[j] = sb->fill[j] = sb->buff[j];
        }
/*
            #] initialize sendbuffer if necessary:
            #[ loop for all terms in infile:
*/
        /*
         * The initial value of maxinterms is determined by the user given
         * ProcessBucketSize and the number of terms in the current expression.
         * We make the initial maxinterms smaller, so that we get the all
         * workers busy as soon as possible.
         */
        maxinterms = ProcessBucketSize / 100;
        if ( maxinterms > e->counter / (PF.numtasks - 1) / 4 )
            maxinterms = e->counter / (PF.numtasks - 1) / 4;
        if ( maxinterms < 1 ) maxinterms = 1;
        cmaxinterms = 0;
        /*
         * Copy them always to sb->buff[0]. When that is full, wait for
         * the next slave to accept terms, exchange sb->buff[0] and
         * sb->buff[next], send sb->buff[next] to next slave and go on
         * filling the now empty sb->buff[0].
         */
        AN.ninterms = 0;
        termsinbucket = 0;
        PACK_LONG(sb->fill[0], 1);
        while ( GetTerm(BHEAD term) ) {
            AN.ninterms++; dd = AN.deferskipped;
            if ( AC.CollectFun && *term <= (LONG)(AM.MaxTer/(2*sizeof(WORD))) ) {
                if ( GetMoreTerms(term) < 0 ) {
                    LowerSortLevel(); return(-1);
                }
            }
            PRINTFBUF("PF_Processor gets",term,*term);
            if ( termsinbucket >= maxinterms || sb->fill[0] + *term >= sb->stop[0] ) {
                next = PF_Wait4Slave(PF_ANY_SOURCE);

```

```
                sb->fill[next] = sb->fill[0];
                sb->full[next] = sb->full[0];
                SWAP(sb->stop[next], sb->stop[0]);
                SWAP(sb->buff[next], sb->buff[0]);
                sb->fill[0] = sb->full[0] = sb->buff[0];
                sb->active = next;

#ifdef MPI2
                if ( PF_Put_origin(next) == 0 ) {
                    printf("PF_Put_origin error...\n");
                }
#else
                PF_ISendSbuf(next,PF_TERM_MSGTAG);
#endif
                /* Initialize the next bucket. */
                termsinbucket = 0;
                PACK_LONG(sb->fill[0], AN.ninterms);
                /*
                 * For the "slow startup". We double maxinterms up to ProcessBucketSize
                 * after (houpefully) the all workers got some terms.
                 */
                if ( cmaxinterms >= PF.numtasks - 2 ) {
                    maxinterms *= 2;
                    if ( maxinterms >= ProcessBucketSize ) {
                        cmaxinterms = -1;
                        maxinterms = ProcessBucketSize;
                    }
                }
                else if ( cmaxinterms >= 0 ) {
                    cmaxinterms++;
                }
            }
            j = *(s = term);
            NCOPY(sb->fill[0], s, j);
            termsinbucket++;
        }
        /* NOTE: The last chunk will be sent to a slave at EndSort() => PF_EndSort()
         *       => PF_WaitAllSlaves(). */
        AN.ninterms += dd;
/*
            #] loop for all terms in infile:
            #[ Clean up & EndSort:
*/
        if ( LastExpression ) {
            UpdateMaxSize();
            if ( AR.infile->handle >= 0 ) {
                CloseFile(AR.infile->handle);
                AR.infile->handle = -1;
                remove(AR.infile->name);
                PUTZERO(AR.infile->POposition);
            }
            AR.infile->POfill = AR.infile->POfull = AR.infile->PObuffer;
        }
        if ( AR.outtohide ) AR.outfile = AR.hidefile;
        PF.parallel = 1;
        if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) return(-1);
        PF.parallel = 0;
        if ( AR.outtohide ) {
            AR.outfile = oldoutfile;
            AR.hidefile->POfull = AR.hidefile->POfill;
        }
        UpdateMaxSize();
        AR.BracketOn = oldBracketOn;
        AT.BrackBuf = oldBrackBuf;
        if ( ( e->vflags & TOBEFACTORED ) != 0 )
            poly_factorize_expression(e);
        else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
         && ( ( e->vflags & ISFACTORIZED ) != 0 ) )
            poly_unfactorize_expression(e);
        AT.bracketindexflag = oldbracketindexflag;
        AR.GetFile = 0;
        AR.outtohide = 0;
        /*
         * NOTE: e->numdummies, e->vflags and AR.exprflags will be updated
         *       after gathering the information from all slaves.
         */
/*
            #] Clean up & EndSort:
            #[ Collect (stats,prepro,...):
*/
        DBGOUT_NINTERMS(1, ("PF.me=%d AN.ninterms=%d ENDSORT\n", (int)PF.me, (int)AN.ninterms));
        PF_CatchErrorMessagesForAll();
        e->numdummies = 0;
        for ( k = 1; k < PF.numtasks; k++ ) {
            PF_LongSingleReceive(PF_ANY_SOURCE, PF_ENDSORT_MSGTAG, &src, &tag);
            PF_LongSingleUnpack(PF_stats[src], PF_STATS_SIZE, PF_LONG);
            {
```

```
                WORD numdummies, expchanged;
                PF_LongSingleUnpack(&numdummies, 1, PF_WORD);
                PF_LongSingleUnpack(&expchanged, 1, PF_WORD);
                if ( e->numdummies < numdummies ) e->numdummies = numdummies;
                AR.expchanged |= expchanged;
            }
            /* Now handle redefined preprocessor variables. */
            if ( AC.numpfirstnum > 0 ) PF_UnpackRedefinedPreVars();
        }
        if ( ! AC.OldParallelStats ) {
            /* Now we can calculate AT.SS->GenTerms from the statistics of the slaves. */
            LONG genterms = 0;
            for ( k = 1; k < PF.numtasks; k++ ) {
                genterms += PF_stats[k][3];
            }
            AT.SS->GenTerms = genterms;
            WriteStats(&PF_exprsize, 2);
            Expressions[AR.CurExpr].size = PF_exprsize;
        }
        PF_Statistics(PF_stats,0);
/*
            #] Collect (stats,prepro,...):
            #[ Update flags :
*/
        if ( AM.S0->TermsLeft ) e->vflags &= ~ISZERO;
        else e->vflags |= ISZERO;
        if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
        if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
        if ( AR.expchanged ) AR.expflags |= ISUNMODIFIED;
/*
            #] Update flags :
        #] Master:
*/
    }
    else {
/*
        #[ Slave :
*/
/*
            #[ Generator Loop & EndSort :

            loop for all terms to get from master, call Generator for each of them
            then call EndSort and do cleanup (to be implemented)
*/
        WORD oldBracketOn = AR.BracketOn;
        WORD *oldBrackBuf = AT.BrackBuf;
        WORD oldbracketindexflag = AT.bracketindexflag;

        /* For redefine statements. */
        if ( AC.numpfirstnum > 0 ) {
            for ( j = 0; j < AC.numpfirstnum; j++ ) {
                AC.inputnumbers[j] = -1;
            }
        }

        SeekScratch(AR.outfile,&position);
        e->onfile = position;
        AR.DeferFlag = AC.ComDefer;
        AR.Eside = RHSIDE;
        if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
            AR.BracketOn = 1;
            AT.BrackBuf = AM.BracketFactors;
            AT.bracketindexflag = 1;
        }
        NewSort(BHEAD0);
        AR.MaxDum = AM.IndDum;
        AN.ninterms = 0;
        PF_linterms = 0;
        PF.parallel = 1;
#ifdef MPI2
        AR.infile->POfull = AR.infile->POfill = AR.infile->PObuffer = PF_shared_buff;
#endif
        {
            FILEHANDLE *fi = AC.RhsExprInModuleFlag && PF.rhsInParallel ? &PF.slavebuf : AR.infile;
            fi->POfull = fi->POfill = fi->PObuffer;
        }
        /* FIXME: AN.ninterms is still broken when AN.deferskipped is non-zero.
         *        It still needs some work, also in PF_GetTerm(). (TU 30 Aug 2011) */
        while ( PF_GetTerm(term) ) {
            PF_linterms++; AN.ninterms++; dd = AN.deferskipped;
            AT.WorkPointer = term + *term;
            AN.RepPoint = AT.RepCount + 1;
            if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
                StoreTerm(BHEAD term);
                continue;
            }
            if ( AR.DeferFlag ) {
```

```
                AR.CurDum = AN.IndDum = Expressions[AR.CurExpr].numdummies + AM.IndDum;
            }
            else {
                AN.IndDum = AM.IndDum;
                AR.CurDum = ReNumber(BHEAD term);
            }
            if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
            if ( AN.ncmod ) {
                if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
                else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
            }
            else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
            if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
                && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
                PolyFunClean(BHEAD term);
            }
            if ( Generator(BHEAD term,0) ) {
                MesPrint("[%d] PF_Processor: Error in Generator",PF.me);
                LowerSortLevel(); return(-1);
            }
            PF_linterms += dd; AN.ninterms += dd;
        }
        PF_linterms += dd; AN.ninterms += dd;
        {
            /*
             * EndSort() overrides AR.outfile->PObuffer etc. (See also PF_EndSort()),
             * but it causes a problem because
             *  (1) PF_EndSort() sets AR.outfile->PObuffer to a send-buffer.
             *  (2) RevertScratch() clears AR.infile, but then swaps buffers of AR.infile
             *      and AR.outfile.
             *  (3) RHS expressions are stored to AR.infile->PObuffer.
             *  (4) Again, PF_EndSort() sets AR.outfile->PObuffer, but now AR.outfile->PObuffer
             *      == AR.infile->PObuffer because of (1) and (2).
             *  (5) The result goes to AR.outfile. This breaks the RHS expressions,
             *      which may be needed for the next expression.
             * Solution: backup & restore AR.outfile->PObuffer etc. (TU 14 Sep 2011)
             */
            FILEHANDLE *fout = AR.outfile;
            WORD *oldbuff = fout->PObuffer;
            WORD *oldstop = fout->POstop;
            LONG  oldsize = fout->POsize;
            if ( EndSort(BHEAD AM.S0->sBuffer, 0) < 0 ) return -1;
            fout->PObuffer = oldbuff;
            fout->POstop   = oldstop;
            fout->POsize   = oldsize;
            fout->POfill = fout->POfull = fout->PObuffer;
        }
        AR.BracketOn = oldBracketOn;
        AT.BrackBuf = oldBrackBuf;
        AT.bracketindexflag = oldbracketindexflag;
/*
            #] Generator Loop & EndSort :
            #[ Collect (stats,prepro...) :
*/
        DBGOUT_NINTERMS(1, ("PF.me=%d AN.ninterms=%d PF_linterms=%d ENDSORT\n", (int)PF.me,
    (int)AN.ninterms, (int)PF_linterms));
        PF_PrepareLongSinglePack();
        cpu = TimeCPU(1);
        size = 0;
        PF_LongSinglePack(&cpu,              1, PF_LONG);
        PF_LongSinglePack(&size,             1, PF_LONG);
        PF_LongSinglePack(&PF_linterms,      1, PF_LONG);
        PF_LongSinglePack(&AM.S0->GenTerms,  1, PF_LONG);
        PF_LongSinglePack(&AM.S0->TermsLeft, 1, PF_LONG);
        {
            WORD numdummies = AR.MaxDum - AM.IndDum;
            PF_LongSinglePack(&numdummies,    1, PF_WORD);
            PF_LongSinglePack(&AR.expchanged, 1, PF_WORD);
        }
        /* Now handle redefined preprocessor variables. */
        if ( AC.numpfirstnum > 0 ) PF_PackRedefinedPreVars();
        PF_LongSingleSend(MASTER, PF_ENDSORT_MSGTAG);
/*
            #] Collect (stats,prepro...) :

        This operation is moved to the beginning of each block, see PreProcessor
        in pre.c.

        #] Slave :
*/
        if ( PF.log ) {
            UBYTE lbuf[24];
            NumToStr(lbuf,AC.CModule);
            fprintf(stderr,"[%d|%s] Endsort,Collect,Broadcast done\n",PF.me,lbuf);
            fflush(stderr);
        }
    }
```

```
    return(0);
}

/*
        #] PF_Processor :
    #] proces.c :
    #[ startup :, prepro & compile
        #[ PF_Init :
*/

int PF_Init(int *argc, char ***argv)
{
/*
        this should definitly be somewhere else ...
*/
    PF_CurrentBracket = 0;

    PF.numtasks = 0; /* number of tasks, is determined in PF_LibInit ! */
    PF.numsbufs = 2; /* might be changed by the environment variable on the master ! */
    PF.numrbufs = 2; /* might be changed by the environment variable on the master ! */

    PF_LibInit(argc,argv);
    PF_RealTime(PF_RESET);

    PF.log = 0;
    PF.parallel = 0;
    PF_statsinterval = 10;
    PF.rhsInParallel=1;
    PF.exprbufsize=4096;/*in WORDs*/

#ifdef PF_WITHGETENV
    if ( PF.me == MASTER ) {
        char *c;
/*
            get these from the environment at the moment sould be in setfile/tail
*/
        if ( ( c = getenv("PF_LOG") ) != 0 ) {
            if ( *c ) PF.log = (int)atoi(c);
            else PF.log = 1;
            fprintf(stderr,"[%d] changing PF.log to %d\n",PF.me,PF.log);
            fflush(stderr);
        }
        if ( ( c = (char*)getenv("PF_RBUFS") ) != 0 ) {
            PF.numrbufs = (int)atoi(c);
            fprintf(stderr,"[%d] changing numrbufs to: %d\n",PF.me,PF.numrbufs);
            fflush(stderr);
        }
        if ( ( c = (char*)getenv("PF_SBUFS") ) != 0 ) {
            PF.numsbufs = (int)atoi(c);
            fprintf(stderr,"[%d] changing numsbufs to: %d\n",PF.me,PF.numsbufs);
            fflush(stderr);
        }
        if ( PF.numsbufs > 10 ) PF.numsbufs = 10;
        if ( PF.numsbufs <  1 ) PF.numsbufs = 1;
        if ( PF.numrbufs >  2 ) PF.numrbufs = 2;
        if ( PF.numrbufs <  1 ) PF.numrbufs = 1;

        if ( ( c = getenv("PF_STATS") ) ) {
            UBYTE lbuf[24];
            PF_statsinterval = (int)atoi(c);
            NumToStr(lbuf,PF_statsinterval);
            fprintf(stderr,"[%d] changing PF_statsinterval to %s\n",PF.me,lbuf);
            fflush(stderr);
            if ( PF_statsinterval < 1 ) PF_statsinterval = 10;
        }
    }
#endif
/*
    #[ Broadcast settings from getenv: could also be done in PF_DoSetup
*/
    if ( PF.me == MASTER ) {
        PF_PreparePack();
        PF_Pack(&PF.log,1,PF_INT);
        PF_Pack(&PF.numrbufs,1,PF_WORD);
        PF_Pack(&PF.numsbufs,1,PF_WORD);
    }
    PF_Broadcast();
    if ( PF.me != MASTER ) {
        PF_Unpack(&PF.log,1,PF_INT);
        PF_Unpack(&PF.numrbufs,1,PF_WORD);
        PF_Unpack(&PF.numsbufs,1,PF_WORD);
        if ( PF.log ) {
            fprintf(stderr, "[%d] log=%d rbufs=%d sbufs=%d\n",
                    PF.me, PF.log, PF.numrbufs, PF.numsbufs);
            fflush(stderr);
        }
    }
```

```
/*
    #] Broadcast settings from getenv:
*/
    return(0);
}
/*
        #] PF_Init :
        #[ PF_Terminate :
*/

int PF_Terminate(int errorcode)
{
    return PF_LibTerminate(errorcode);
}

/*
        #] PF_Terminate :
        #[ PF_GetSlaveTimes :
*/

LONG PF_GetSlaveTimes(void)
{
    LONG slavetimes = 0;
    LONG t = PF.me == MASTER ? 0 : AM.SumTime + TimeCPU(1);
    MPI_Reduce(&t, &slavetimes, 1, PF_LONG, MPI_SUM, MASTER, PF_COMM);
    return slavetimes;
}

/*
        #] PF_GetSlaveTimes :
    #] startup :
    #[ PF_BroadcastNumber :
*/

LONG PF_BroadcastNumber(LONG x)
{
#ifdef PF_DEBUG_BCAST_LONG
    if ( PF.me == MASTER ) {
        MesPrint("» Broadcast LONG: %l", x);
    }
#endif
    PF_Bcast(&x, sizeof(LONG));
    return x;
}

/*
    #] PF_BroadcastNumber :
    #[ PF_BroadcastBuffer :
*/

void PF_BroadcastBuffer(WORD **buffer, LONG *length)
{
    WORD *p;
    LONG rest;
#ifdef PF_DEBUG_BCAST_BUF
    if ( PF.me == MASTER ) {
        MesPrint("» Broadcast Buffer: length=%l", *length);
    }
#endif
    /* Initialize the buffer on the slaves. */
    if ( PF.me != MASTER ) {
        *buffer = NULL;
    }
    /* Broadcast the length of the buffer. */
    *length = PF_BroadcastNumber(*length);
    if ( *length <= 0 ) return;
    /* Allocate the buffer on the slaves. */
    if ( PF.me != MASTER ) {
        *buffer = (WORD *)Malloc1(*length * sizeof(WORD), "PF_BroadcastBuffer");
    }
    /* Broadcast the data in the buffer. */
    p = *buffer;
    rest = *length;
    while ( rest > 0 ) {
        int l = rest < (LONG)PF.exprbufsize ? (int)rest : PF.exprbufsize;
        PF_Bcast(p, l * sizeof(WORD));
        p += l;
        rest -= l;
    }
}

/*
    #] PF_BroadcastBuffer :
    #[ PF_BroadcastString :
*/

int PF_BroadcastString(UBYTE *str)
```

```
{
    int clength = 0;
/*
        If string does not fit to the PF_buffer, it
        will be split into chanks. Next chank is started at  str+clength
*/
        UBYTE *cstr=str;
/*
        Note, compilation is performed INDEPENDENTLY on AC.mparallelflag!
        No if ( AC.mparallelflag == PARALLELFLAG ) !!
*/
    do {
        cstr += clength; /*at each step for all slaves and master */

        if ( MASTER == PF.me ) { /*Pack str*/
/*
                initialize buffers
*/
            if ( PF_PreparePack() != 0 ) Terminate(-1);
            if ( ( clength = PF_PackString(cstr) ) <0  ) Terminate(-1);
        }
        PF_Broadcast();

        if ( MASTER != PF.me ) {
/*
                Slave - unpack received string
                For slaves buffers are initialised automatically.
*/
            if ( ( clength = PF_UnpackString(cstr) ) < 0 ) Terminate(-1);
        }
    } while ( cstr[clength-1] != '\0' );
    return (0);
}

/*
    #] PF_BroadcastString :
    #[ PF_BroadcastPreDollar :
*/

int PF_BroadcastPreDollar(WORD **dbuffer, LONG *newsize, int *numterms)
{
    int err = 0;
    LONG i;
/*
        Note, compilation is performed INDEPENDENTLY on AC.mparallelflag!
        No if(AC.mparallelflag==PARALLELFLAG) !!
*/
    if ( MASTER == PF.me ) {
/*
            The problem is that sometimes dollar variables are longer
            than PF_packbuf! So we split long expression into chunks.
            There are n filled chunks and one portially filled chunk:
*/
        LONG n = ((*newsize)+1)/PF_maxDollarChunkSize;
/*
            ...and one more chunk for the rest; if the expression fits to
            the buffer without splitting, the latter will be the only one.

            PF_maxDollarChunkSize is the maximal number of items fitted to
            the buffer. It is calculated in PF_LibInit() in mpi.c.
            PF_maxDollarChunkSize is calculated for the first step, when
            two fields (numterms and newsize, see below) are already packed.
            For simplicity, this value is used also for all steps, in
            despite  of it is   a bit less than maximally available space.
*/
        WORD *thechunk = *dbuffer;

        err = PF_PreparePack();             /* initialize buffers */
        err |= PF_Pack(numterms,1,PF_INT);
        err |= PF_Pack(newsize,1,PF_LONG); /* pack the size */
/*
            Pack and broadcast completely filled chunks.
            It may happen, this loop is not entered at all:
*/
        for ( i = 0; i < n; i++ ) {
            err |= PF_Pack(thechunk,PF_maxDollarChunkSize,PF_WORD);
            err |= PF_Broadcast();
            thechunk +=PF_maxDollarChunkSize;
            PF_PreparePack();
        }
/*
            Pack and broadcast the rest:
*/
        if ( ( n = ( (*newsize)+1)%PF_maxDollarChunkSize ) != 0 ) {
            err |= PF_Pack(thechunk,n,PF_WORD);
            err |= PF_Broadcast();
        }
```

```
#ifdef PF_DEBUG_BCAST_PREDOLLAR
        MesPrint("» Broadcast PreDollar: newsize=%d numterms=%d", (int)*newsize, *numterms);
#endif
    }
    if ( MASTER != PF.me ) {  /* Slave - unpack received buffer */
        WORD *thechunk;
        LONG n, therest, thesize;
        err |= PF_Broadcast();
        err |=PF_Unpack(numterms,1,PF_INT);
        err |=PF_Unpack(newsize,1,PF_LONG);
/*
            Now we know the buffer size.
*/
        thesize = (*newsize)+1;
/*
            Evaluate the number of completely filled chunks. The last step must be
            treated separately, so -1:
*/
        n = (thesize/PF_maxDollarChunkSize) - 1;
/*
            Note, here n can be <0, this is ok.
*/
        therest = thesize % PF_maxDollarChunkSize;
        thechunk = *dbuffer =
            (WORD*)Malloc1( thesize * sizeof(WORD),"$-buffer slave");
        if ( thechunk == NULL ) return(err|4);
/*
            Unpack completely filled chunks and receive the next portion.
            It may happen, this loop is not entered at all:
*/
        for ( i = 0; i < n; i++ ) {
            err |= PF_Unpack(thechunk,PF_maxDollarChunkSize,PF_WORD);
            thechunk += PF_maxDollarChunkSize;
            err |= PF_Broadcast();
        }
/*
            Now the last completely filled chunk:
*/
        if ( n >= 0 ) {
            err |= PF_Unpack(thechunk,PF_maxDollarChunkSize,PF_WORD);
            thechunk += PF_maxDollarChunkSize;
            if ( therest != 0 ) err |= PF_Broadcast();
        }
/*
            Unpack the rest (it is already received!):
*/
        if ( therest != 0 ) err |= PF_Unpack(thechunk,therest,PF_WORD);
    }
    return (err);
}

/*
    #] PF_BroadcastPreDollar :
    #[ Synchronization of modified dollar variables :
        #[ Helper functions :
            #[ dollarlen :
*/

static inline LONG dollarlen(const WORD *terms)
{
    const WORD *p = terms;
    while ( *p ) p += *p;
    return p - terms;  /* Not including the null terminator. */
}

/*
            #] dollarlen :
            #[ dollar_mod_type :
*/

static inline WORD dollar_mod_type(WORD index)
{
    int i;
    for ( i = 0; i < NumModOptdollars; i++ )
        if ( ModOptdollars[i].number == index ) break;
    if ( i >= NumModOptdollars ) return -1;
    return ModOptdollars[i].type;
}


/*
            #] dollar_mod_type :
        #] Helper functions :
        #[ PF_CollectModifiedDollars :
*/

/*
```

```
            #[ dollar_to_be_collected :
*/

static inline int dollar_to_be_collected(WORD index)
{
    switch ( dollar_mod_type(index) ) {
        case MODSUM:
        case MODMAX:
        case MODMIN:
            return 1;
        default:
            return 0;
    }
}

/*
            #] dollar_to_be_collected :
            #[ copy_dollar :
*/

static inline void copy_dollar(WORD index, WORD type, const WORD *where, LONG size)
{
    DOLLARS d = Dollars + index;

    CleanDollarFactors(d);

    if ( type != DOLZERO && where != NULL && where != &AM.dollarzero && where[0] != 0 && size > 0 ) {
        if ( size > d->size || size < d->size / 4 ) {  /* Reallocate if not enough or too much. */
            if ( d->where && d->where != &AM.dollarzero )
                M_free(d->where, "old content of dollar");
            d->where = Malloc1(sizeof(WORD) * size, "copy buffer to dollar");
            d->size  = size;
        }
        d->type  = type;
        WCOPY(d->where, where, size);
    }
    else {
        if ( d->where && d->where != &AM.dollarzero )
            M_free(d->where, "old content of dollar");
        d->type  = DOLZERO;
        d->where = &AM.dollarzero;
        d->size  = 0;
    }
}

/*
            #] copy_dollar :
            #[ compare_two_expressions :
*/

static inline int compare_two_expressions(const WORD *e1, const WORD *e2)
{
    GETIDENTITY
    /*
     * We consider the cases that
     *   (1) the expression has no term,
     *   (2) the expression has only one term and it is a number,
     *   (3) otherwise.
     * Assume that the expressions are sorted and all terms are normalized.
     * The numerators of the coefficients must never be zero.
     *
     * Note that TwoExprCompare() is not adequate for our purpose
     * (as of 6 Aug. 2013), e.g., TwoExprCompare({0}, {4, 1, 1, -1}, LESS)
     * returns TRUE.
     */
    if ( e1[0] == 0 ) {
        if ( e2[0] == 0 ) {
            return(0);
        }
        else if ( e2[e2[0]] == 0 && e2[0] == ABS(e2[e2[0] - 1]) + 1 ) {
            if ( e2[e2[0] - 1] > 0 )
                return(-1);
            else
                return(+1);
        }
    }
    else if ( e1[e1[0]] == 0 && e1[0] == ABS(e1[e1[0] - 1]) + 1 ) {
        if ( e2[0] == 0 ) {
            if ( e1[e1[0] - 1] > 0 )
                return(+1);
            else
                return(-1);
        }
        else if ( e2[e2[0]] == 0 && e2[0] == ABS(e2[e2[0] - 1]) + 1 ) {
            return(CompCoef((WORD *)e1, (WORD *)e2));
        }
    }
```

```
    /* The expressions are not so simple. Define the order by each term. */
    while ( e1[0] && e2[0] ) {
        int c = CompareTerms(BHEAD (WORD *)e1, (WORD *)e2, 1);
        if ( c < 0 )
            return(-1);
        else if ( c > 0 )
            return(+1);
        e1 += e1[0];
        e2 += e2[0];
    }
    if ( e1[0] ) return(+1);
    if ( e2[0] ) return(-1);
    return(0);
}

/*
            #] compare_two_expressions :
            #[ Variables :
*/

typedef struct {
    VectorStruct(WORD) buf;
    LONG size;
    WORD type;
    PADPOINTER(1,0,1,0);
} dollar_buf;

/* Buffers used to store data for each variable from each slave. */
static Vector(dollar_buf, dollar_slave_bufs);

/*
            #] Variables :
*/

int PF_CollectModifiedDollars(void)
{
    int i, j, ndollars;
    /*
     * If the current module was executed in the sequential mode,
     * there are no modified module on the slaves.
     */
    if ( AC.mparallelflag != PARALLELFLAG && !AC.partodoflag ) return 0;
    /*
     * Count the number of (potentially) modified dollar variables, which we need to collect.
     * Here we need to collect all max/min/sum variables.
     */
    ndollars = 0;
    for ( i = 0; i < NumPotModdollars; i++ ) {
        WORD index = PotModdollars[i];
        if ( dollar_to_be_collected(index) ) ndollars++;
    }
    if ( ndollars == 0 ) return 0;  /* No dollars to be collected. */

    if ( PF.me == MASTER ) {
/*
            #[ Master :
*/
        int nslaves, nvars;
        /* Prepare receive buffers. We need ndollars*(PF.numtasks-1) buffers. */
        int nbufs = ndollars * (PF.numtasks - 1);
        VectorReserve(dollar_slave_bufs, nbufs);
        for ( i = VectorSize(dollar_slave_bufs); i < nbufs; i++ ) {
            VectorInit(VectorPtr(dollar_slave_bufs)[i].buf);
        }
        VectorSize(dollar_slave_bufs) = nbufs;
        /* Receive data from each slave. */
        for ( nslaves = 1; nslaves < PF.numtasks; nslaves++ ) {
            int src;
            PF_LongSingleReceive(PF_ANY_SOURCE, PF_DOLLAR_MSGTAG, &src, NULL);
            nvars = 0;
            for ( i = 0; i < NumPotModdollars; i++ ) {
                WORD index = PotModdollars[i];
                dollar_buf *b;
                if ( !dollar_to_be_collected(index) ) continue;
                b = &VectorPtr(dollar_slave_bufs)[(PF.numtasks - 1) * nvars + (src - 1)];
                PF_LongSingleUnpack(&b->type, 1, PF_WORD);
                if ( b->type != DOLZERO ) {
                    LONG size;
                    WORD *where;
                    PF_LongSingleUnpack(&size, 1, PF_LONG);
                    VectorReserve(b->buf, size + 1);
                    where = VectorPtr(b->buf);
                    PF_LongSingleUnpack(where, size, PF_WORD);
                    where[size] = 0;  /* The null terminator is needed. */
                    b->size = size + 1;  /* Including the null terminator. */
                    /* Note that we don't collect factored stuff for max/min/sum variables. */
                }
```

```
                else {
                    VectorReserve(b->buf, 1);
                    VectorPtr(b->buf)[0] = 0;
                    b->size = 0;
                }
                nvars++;
            }
        }
        /*
         * Combine received dollars. The FORM reference manual says maximum/minimum/sum
         * $-variables must have a numerical value, however, this routine should work also
         * for non-numerical cases, although the maximum/minimum value for non-numerical
         * terms has ambiguity.
         */
        nvars = 0;
        for ( i = 0; i < NumPotModdollars; i++ ) {
            WORD index = PotModdollars[i];
            WORD dtype;
            DOLLARS d;
            dollar_buf *b;
            if ( !dollar_to_be_collected(index) ) continue;
            d = Dollars + index;
            b = &VectorPtr(dollar_slave_bufs)[(PF.numtasks - 1) * nvars];
            dtype = dollar_mod_type(index);
            switch ( dtype ) {
                case MODMAX:
                case MODMIN: {
/*
            #[ MODMAX & MODMIN :
*/
                    int selected = 0;
                    for ( j = 1; j < PF.numtasks - 1; j++ ) {
                        int c = compare_two_expressions(VectorPtr(b[j].buf), VectorPtr(b[selected].buf));
                        if ( (dtype == MODMAX && c > 0) || (dtype == MODMIN && c < 0) )
                            selected = j;
                    }
                    b = b + selected;
                    copy_dollar(index, b->type, VectorPtr(b->buf), b->size);
/*
            #] MODMAX & MODMIN :
*/
                    break;
                }
                case MODSUM: {
/*
            #[ MODSUM :
*/
                    GETIDENTITY
                    int err = 0;

                    CBUF *C = cbuf + AM.rbufnum;
                    WORD *oldwork = AT.WorkPointer, *oldcterm = AN.cTerm;
                    WORD olddefer = AR.DeferFlag, oldnumlhs = AR.Cnumlhs, oldnumrhs = C->numrhs;

                    LONG size;
                    WORD type, *dbuf;

                    AN.cTerm = 0;
                    AR.DeferFlag = 0;

                    if ( ((WORD *)((UBYTE *)AT.WorkPointer + AM.MaxTer)) > AT.WorkTop ) {
                        err = -1;
                        goto cleanup;
                        MesWork();
                    }

                    if ( NewSort(BHEAD0) ) {
                        err = -1;
                        goto cleanup;
                    }
                    if ( NewSort(BHEAD0) ) {
                        LowerSortLevel();
                        err = -1;
                        goto cleanup;
                    }

                    /*
                     * Sum up the original $-variable in the master and $-variables on all slaves.
                     * Note that $-variables on the slaves are set to zero at the beginning of
                     * the module (See also DoExecute()).
                     */
                    for ( j = 0; j < PF.numtasks; j++ ) {
                        const WORD *r;
                        for ( r = j == 0 ? Dollars[index].where : VectorPtr(b[j - 1].buf); *r; r += *r ) {
                            WCOPY(AT.WorkPointer, r, *r);
```

```
                            AT.WorkPointer += *r;
                            AR.Cnumlhs = 0;
                            if ( Generator(BHEAD oldwork, 0) ) {
                                LowerSortLevel(); LowerSortLevel();
                                err = -1;
                                goto cleanup;
                            }
                            AT.WorkPointer = oldwork;
                        }
                    }

                    size = EndSort(BHEAD (WORD *)&dbuf, 2);
                    if ( size < 0 ) {
                        LowerSortLevel();
                        err = -1;
                        goto cleanup;
                    }
                    LowerSortLevel();

                    /* Find special cases. */
                    type = DOLTERMS;
                    if ( dbuf[0] == 0 ) {
                        type = DOLZERO;
                    }
                    else if ( dbuf[dbuf[0]] == 0 ) {
                        const WORD *t = dbuf, *w;
                        WORD n, nsize;
                        n = *t;
                        nsize = t[n - 1];
                        if ( nsize < 0 ) nsize = -nsize;
                        if ( nsize == n - 1 ) {
                            nsize = (nsize - 1) / 2;
                            w = t + 1 + nsize;
                            if ( *w == 1 ) {
                                w++; while ( w < t + n - 1 ) { if ( *w ) break; w++; }
                                if ( w >= t + n - 1 ) type =  DOLNUMBER;
                            }
                            else if ( n == 7 && t[6] == 3 && t[5] == 1 && t[4] == 1 && t[1] == INDEX && t[2] == 3 ) {
                                type = DOLINDEX;
                                d->index = t[3];
                            }
                        }
                    }
                    copy_dollar(index, type, dbuf, dollarlen(dbuf) + 1);
                    M_free(dbuf, "temporary dollar buffer");
cleanup:
                    AR.Cnumlhs = oldnumlhs;
                    C->numrhs = oldnumrhs;
                    AR.DeferFlag = olddefer;
                    AN.cTerm = oldcterm;
                    AT.WorkPointer = oldwork;

                    if ( err ) return err;
/*
            #] MODSUM :
*/
                    break;
                }
            }
            if ( d->type == DOLTERMS )
                cbuf[AM.dbufnum].CanCommu[index] = numcommute(d->where, &cbuf[AM.dbufnum].NumTerms[index]);
            cbuf[AM.dbufnum].rhs[index] = d->where;
            nvars++;
#ifdef PF_DEBUG_REDUCE_DOLLAR
            MesPrint("« Reduce $-var: %s", AC.dollarnames->namebuffer + d->name);
#endif
        }
/*
            #] Master :
*/
    }
    else {
/*
            #[ Slave :
*/
        PF_PrepareLongSinglePack();
        /* Pack each variable. */
        for ( i = 0; i < NumPotModdollars; i++ ) {
            WORD index = PotModdollars[i];
            DOLLARS d;
            if ( !dollar_to_be_collected(index) ) continue;
            d = Dollars + index;
            PF_LongSinglePack(&d->type, 1, PF_WORD);
            if ( d->type != DOLZERO ) {
                /*
```

```
                 * NOTE: d->size is the allocated buffer size for d->where in WORDs.
                 *       So dollarlen(d->where) can be < d->size-1. (TU 15 Dec 2011)
                 */
                LONG size = dollarlen(d->where);
                PF_LongSinglePack(&size, 1, PF_LONG);
                PF_LongSinglePack(d->where, size, PF_WORD);
                /* Note that we don't collect factored stuff for max/min/sum variables. */
            }
        }
        PF_LongSingleSend(MASTER, PF_DOLLAR_MSGTAG);
/*
            #] Slave :
*/
    }
    return 0;
}

/*
        #] PF_CollectModifiedDollars :
        #[ PF_BroadcastModifiedDollars :
*/

/*
            #[ dollar_to_be_broadcast :
*/

static inline int dollar_to_be_broadcast(WORD index)
{
    switch ( dollar_mod_type(index) ) {
        case MODLOCAL:
            return 0;
        default:
            return 1;
    }
}

/*
            #] dollar_to_be_broadcast :
*/

int PF_BroadcastModifiedDollars(void)
{
    int i, j, ndollars;
    /*
     * Count the number of (potentially) modified dollar variables, which we need to broadcast.
     * Here we need to broadcast all non-local variables.
     */
    ndollars = 0;
    for ( i = 0; i < NumPotModdollars; i++ ) {
        WORD index = PotModdollars[i];
        if ( dollar_to_be_broadcast(index) ) ndollars++;
    }
    if ( ndollars == 0 ) return 0;  /* No dollars to be broadcast. */

    if ( PF.me == MASTER ) {
/*
            #[ Master :
*/
        PF_PrepareLongMultiPack();
        /* Pack each variable. */
        for ( i = 0; i < NumPotModdollars; i++ ) {
            WORD index = PotModdollars[i];
            DOLLARS d;
            if ( !dollar_to_be_broadcast(index) ) continue;
            d = Dollars + index;
            PF_LongMultiPack(&d->type, 1, PF_WORD);
            if ( d->type != DOLZERO ) {
                /*
                 * NOTE: d->size is the allocated buffer size for d->where in WORDs.
                 *       So dollarlen(d->where) can be < d->size-1. (TU 15 Dec 2011)
                 */
                LONG size = dollarlen(d->where);
                PF_LongMultiPack(&size, 1, PF_LONG);
                PF_LongMultiPack(d->where, size, PF_WORD);
                /* ...and the factored stuff. */
                PF_LongMultiPack(&d->nfactors, 1, PF_WORD);
                if ( d->nfactors > 1 ) {
                    for ( j = 0; j < d->nfactors; j++ ) {
                        FACDOLLAR *f = &d->factors[j];
                        PF_LongMultiPack(&f->type, 1, PF_WORD);
                        PF_LongMultiPack(&f->size, 1, PF_LONG);
                        if ( f->size > 0 )
                            PF_LongMultiPack(f->where, f->size, PF_WORD);
                        else
                            PF_LongMultiPack(&f->value, 1, PF_WORD);
                    }
                }
```

```
            }
#ifdef PF_DEBUG_BCAST_DOLLAR
            MesPrint("» Broadcast $-var: %s", AC.dollarnames->namebuffer + d->name);
#endif
        }
/*
            #] Master :
*/
    }
    if ( PF_LongMultiBroadcast() ) return -1;
    if ( PF.me != MASTER ) {
/*
            #[ Slave :
*/
        for ( i = 0; i < NumPotModdollars; i++ ) {
            WORD index = PotModdollars[i];
            DOLLARS d;
            if ( !dollar_to_be_broadcast(index) ) continue;
            d = Dollars + index;
            /* Clear the contents of the dollar variable. */
            if ( d->where && d->where != &AM.dollarzero )
                M_free(d->where, "old content of dollar");
            d->where = &AM.dollarzero;
            d->size = 0;
            CleanDollarFactors(d);
            /* Unpack and store the contents. */
            PF_LongMultiUnpack(&d->type, 1, PF_WORD);
            if ( d->type != DOLZERO ) {
                LONG size;
                PF_LongMultiUnpack(&size, 1, PF_LONG);
                d->size = size + 1;
                d->where = (WORD *)Malloc1(sizeof(WORD) * d->size, "dollar content");
                PF_LongMultiUnpack(d->where, size, PF_WORD);
                d->where[size] = 0;  /* The null terminator is needed. */
                /* ...and the factored stuff. */
                PF_LongMultiUnpack(&d->nfactors, 1, PF_WORD);
                if ( d->nfactors > 1 ) {
                    d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR) * d->nfactors, "dollar factored stuff");
                    for ( j = 0; j < d->nfactors; j++ ) {
                        FACDOLLAR *f = &d->factors[j];
                        PF_LongMultiUnpack(&f->type, 1, PF_WORD);
                        PF_LongMultiUnpack(&f->size, 1, PF_LONG);
                        if ( f->size > 0 ) {
                            f->where = (WORD *)Malloc1(sizeof(WORD) * (f->size + 1), "dollar factor content");
                            PF_LongMultiUnpack(f->where, f->size, PF_WORD);
                            f->where[f->size] = 0;  /* The null terminator is needed. */
                            f->value = 0;
                        }
                        else {
                            f->where = NULL;
                            PF_LongMultiUnpack(&f->value, 1, PF_WORD);
                        }
                    }
                }
            }
            if ( d->type == DOLTERMS )
                cbuf[AM.dbufnum].CanCommu[index] = numcommute(d->where, &cbuf[AM.dbufnum].NumTerms[index]);
            cbuf[AM.dbufnum].rhs[index] = d->where;
        }
/*
            #] Slave :
*/
    }
    return 0;
}

/*
        #] PF_BroadcastModifiedDollars :
    #] Synchronization of modified dollar variables :
    #[ Synchronization of redefined preprocessor variables :
        #[ Variables :
*/

/* A buffer used in receivers. */
static Vector(UBYTE, prevarbuf);

/*
        #] Variables :
        #[ PF_PackRedefinedPreVars :
*/

static void PF_PackRedefinedPreVars(void)
{
    int i;
```

```
    /* First, pack the number of redefined preprocessor variables. */
    int nredefs = 0;
    for ( i = 0; i < AC.numpfirstnum; i++ )
        if ( AC.inputnumbers[i] >= 0 ) nredefs++;
    PF_LongSinglePack(&nredefs, 1, PF_INT);
    /* Then, pack each variable. */
    for ( i = 0; i < AC.numpfirstnum; i++ )
        if ( AC.inputnumbers[i] >= 0) {
            WORD index = AC.pfirstnum[i];
            UBYTE *value = PreVar[index].value;
            int bytes = strlen((char *)value);
            PF_LongSinglePack(&index, 1, PF_WORD);
            PF_LongSinglePack(&bytes, 1, PF_INT);
            PF_LongSinglePack(value, bytes, PF_BYTE);
            PF_LongSinglePack(&AC.inputnumbers[i], 1, PF_LONG);
    }
}

/*
        #] PF_PackRedefinedPreVars :
        #[ PF_UnpackRedefinedPreVars :
*/

static void PF_UnpackRedefinedPreVars(void)
{
    int i, j;
    /* Unpack the number of redefined preprocessor variables. */
    int nredefs;
    PF_LongSingleUnpack(&nredefs, 1, PF_INT);
    if ( nredefs > 0 ) {
        /* Then unpack each variable. */
        for ( i = 0; i < nredefs; i++ ) {
            WORD index;
            int bytes;
            UBYTE *value;
            LONG inputnumber;
            PF_LongSingleUnpack(&index, 1, PF_WORD);
            PF_LongSingleUnpack(&bytes, 1, PF_INT);
            VectorReserve(prevarbuf, bytes + 1);
            value = VectorPtr(prevarbuf);
            PF_LongSingleUnpack(value, bytes, PF_BYTE);
            value[bytes] = '\0';  /* The null terminator is needed. */
            PF_LongSingleUnpack(&inputnumber, 1, PF_LONG);
            /* Put this variable if it must be updated. */
            for ( j = 0; j < AC.numpfirstnum; j++ )
                if ( AC.pfirstnum[j] == index ) break;
            if ( AC.inputnumbers[j] < inputnumber ) {
                AC.inputnumbers[j] = inputnumber;
                PutPreVar(PreVar[index].name, value, NULL, 1);
            }
        }
    }
}

/*
        #] PF_UnpackRedefinedPreVars :
        #[ PF_BroadcastRedefinedPreVars :
*/

int PF_BroadcastRedefinedPreVars(void)
{
    /*
     * NOTE: Because the compilation is performed on the all processes
     * independently on AC.mparallelflag, we always have to broadcast redefined
     * preprocessor variables from the master to the all slaves.
     */
    if ( PF.me == MASTER ) {
/*
            #[ Master :
*/
        int i, nredefs;
        PF_PrepareLongMultiPack();
        /* First, pack the number of redefined preprocessor variables. */
        nredefs = 0;
        for ( i = 0; i < AC.numpfirstnum; i++ )
            if ( AC.inputnumbers[i] >= 0 ) nredefs++;
        PF_LongMultiPack(&nredefs, 1, PF_INT);
        /* Then, pack each variable. */
        for ( i = 0; i < AC.numpfirstnum; i++ )
            if ( AC.inputnumbers[i] >= 0) {
                WORD index = AC.pfirstnum[i];
                UBYTE *value = PreVar[index].value;
                int bytes = strlen((char *)value);
                PF_LongMultiPack(&index, 1, PF_WORD);
                PF_LongMultiPack(&bytes, 1, PF_INT);
                PF_LongMultiPack(value, bytes, PF_BYTE);
#ifdef PF_DEBUG_BCAST_PREVAR
```

```
                MesPrint("» Broadcast PreVar: %s = \"%s\"", PreVar[index].name, value);
#endif
            }
/*
            #] Master :
*/
    }
    if ( PF_LongMultiBroadcast() ) return -1;
    if ( PF.me != MASTER ) {
/*
            #[ Slave :
*/
        int i, nredefs;
        /* Unpack the number of redefined preprocessor variables. */
        PF_LongMultiUnpack(&nredefs, 1, PF_INT);
        if ( nredefs > 0 ) {
            /* Then unpack each variable and put it. */
            for ( i = 0; i < nredefs; i++ ) {
                WORD index;
                int bytes;
                UBYTE *value;
                PF_LongMultiUnpack(&index, 1, PF_WORD);
                PF_LongMultiUnpack(&bytes, 1, PF_INT);
                VectorReserve(prevarbuf, bytes + 1);
                value = VectorPtr(prevarbuf);
                PF_LongMultiUnpack(value, bytes, PF_BYTE);
                value[bytes] = '\0';  /* The null terminator is needed. */
                PutPreVar(PreVar[index].name, value, NULL, 1);
            }
        }
/*
            #] Slave :
*/
    }
    return 0;
}

/*
        #] PF_BroadcastRedefinedPreVars :
    #] Synchronization of redefined preprocessor variables :
    #[ Preprocessor Inside instruction :
        #[ Variables :
*/

/* Saved values of AC.RhsExprInModuleFlag, PotModdollars and AC.pfirstnum. */
static WORD oldRhsExprInModuleFlag;
static Vector(WORD, oldPotModdollars);
static Vector(WORD, oldpfirstnum);

/*
        #] Variables :
        #[ PF_StoreInsideInfo :
*/

/*
 * Saves the current values of AC.RhsExprInModuleFlag, PotModdollars
 * and AC.pfirstnum.
 *
 * Called by DoInside().
 *
 * @return  0  if OK, nonzero on error.
 */
int PF_StoreInsideInfo(void)
{
    int i;
    oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
    VectorClear(oldPotModdollars);
    for ( i = 0; i < NumPotModdollars; i++ )
        VectorPushBack(oldPotModdollars, PotModdollars[i]);
    VectorClear(oldpfirstnum);
    for ( i = 0; i < AC.numpfirstnum; i++ )
        VectorPushBack(oldpfirstnum, AC.pfirstnum[i]);
    return 0;
}

/*
        #] PF_StoreInsideInfo :
        #[ PF_RestoreInsideInfo :
*/

/*
 * Restores the saved values of AC.RhsExprInModuleFlag, PotModdollars
 * and AC.pfirstnum.
 *
 * Called by DoEndInside().
 *
 * @return  0  if OK, nonzero on error.
```

```
 */
int PF_RestoreInsideInfo(void)
{
    int i;
    AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
    NumPotModdollars = VectorSize(oldPotModdollars);
    for ( i = 0; i < NumPotModdollars; i++ )
        PotModdollars[i] = VectorPtr(oldPotModdollars)[i];
    AC.numpfirstnum = VectorSize(oldpfirstnum);
    for ( i = 0; i < AC.numpfirstnum; i++ )
        AC.pfirstnum[i] = VectorPtr(oldpfirstnum)[i];
    return 0;
}

/*
        #] PF_RestoreInsideInfo :
    #] Preprocessor Inside instruction :
    #[ PF_BroadcastCBuf :
*/

int PF_BroadcastCBuf(int bufnum)
{
    CBUF *C = cbuf + bufnum;
    int i;
    LONG l;
    if ( PF.me == MASTER ) {
/*
        #[ Master :
*/
        PF_PrepareLongMultiPack();
        /* Pack CBUF struct except pointers. */
        PF_LongMultiPack(&C->BufferSize, 1, PF_LONG);
        PF_LongMultiPack(&C->numlhs, 1, PF_INT);
        PF_LongMultiPack(&C->numrhs, 1, PF_INT);
        PF_LongMultiPack(&C->maxlhs, 1, PF_INT);
        PF_LongMultiPack(&C->maxrhs, 1, PF_INT);
        PF_LongMultiPack(&C->mnumlhs, 1, PF_INT);
        PF_LongMultiPack(&C->mnumrhs, 1, PF_INT);
        PF_LongMultiPack(&C->numtree, 1, PF_INT);
        PF_LongMultiPack(&C->rootnum, 1, PF_INT);
        PF_LongMultiPack(&C->MaxTreeSize, 1, PF_INT);
        /* Now pointers. Pointer, lhs and rhs are packed as offsets. We don't pack Top. */
        l = C->Pointer - C->Buffer;
        PF_LongMultiPack(&l, 1, PF_LONG);
        PF_LongMultiPack(C->Buffer, l, PF_WORD);
        for ( i = 0; i < C->numlhs + 1; i++ ) {
            l = C->lhs[i] - C->Buffer;
            PF_LongMultiPack(&l, 1, PF_LONG);
        }
        for ( i = 0; i < C->numrhs + 1; i++ ) {
            l = C->rhs[i] - C->Buffer;
            PF_LongMultiPack(&l, 1, PF_LONG);
        }
        PF_LongMultiPack(C->CanCommu, C->numrhs + 1, PF_LONG);
        PF_LongMultiPack(C->NumTerms, C->numrhs + 1, PF_LONG);
        PF_LongMultiPack(C->numdum, C->numrhs + 1, PF_WORD);
        PF_LongMultiPack(C->dimension, C->numrhs + 1, PF_WORD);
        if ( C->MaxTreeSize > 0 )
            PF_LongMultiPack(C->boomlijst, (C->numtree + 1) * (sizeof(COMPTREE) / sizeof(int)),
    PF_INT);
#ifdef PF_DEBUG_BCAST_CBUF
        MesPrint("» Broadcast CBuf %d", bufnum);
#endif
/*
        #] Master :
*/
    }
    if ( PF_LongMultiBroadcast() ) return -1;
    if ( PF.me != MASTER ) {
/*
        #[ Slave :
*/
        /* First, free already allocated buffers. */
        finishcbuf(bufnum);
        /* Unpack CBUF struct except pointers. */
        PF_LongMultiUnpack(&C->BufferSize, 1, PF_LONG);
        PF_LongMultiUnpack(&C->numlhs, 1, PF_INT);
        PF_LongMultiUnpack(&C->numrhs, 1, PF_INT);
        PF_LongMultiUnpack(&C->maxlhs, 1, PF_INT);
        PF_LongMultiUnpack(&C->maxrhs, 1, PF_INT);
        PF_LongMultiUnpack(&C->mnumlhs, 1, PF_INT);
        PF_LongMultiUnpack(&C->mnumrhs, 1, PF_INT);
        PF_LongMultiUnpack(&C->numtree, 1, PF_INT);
        PF_LongMultiUnpack(&C->rootnum, 1, PF_INT);
        PF_LongMultiUnpack(&C->MaxTreeSize, 1, PF_INT);
        /* Allocate new buffers. */
        C->Buffer = (WORD *)Malloc1(C->BufferSize * sizeof(WORD), "compiler buffer");
```

```
        C->Top = C->Buffer + C->BufferSize;
        C->lhs = (WORD **)Malloc1(C->maxlhs * sizeof(WORD *), "compiler buffer");
        C->rhs = (WORD **)Malloc1(C->maxrhs * (sizeof(WORD *) + 2 * sizeof(LONG) + 2 * sizeof(WORD)), "compiler buffer");
        C->CanCommu = (LONG *)(C->rhs + C->maxrhs);
        C->NumTerms = C->CanCommu + C->maxrhs;
        C->numdum = (WORD *)(C->NumTerms + C->maxrhs);
        C->dimension = C->numdum + C->maxrhs;
        if ( C->MaxTreeSize > 0 )
            C->boomlijst = (COMPTREE *)Malloc1(C->MaxTreeSize * sizeof(COMPTREE), "compiler buffer");
        /* Unpack buffers. */
        PF_LongMultiUnpack(&l, 1, PF_LONG);
        PF_LongMultiUnpack(C->Buffer, l, PF_WORD);
        C->Pointer = C->Buffer + l;
        for ( i = 0; i < C->numlhs + 1; i++ ) {
            PF_LongMultiUnpack(&l, 1, PF_LONG);
            C->lhs[i] = C->Buffer + l;
        }
        for ( i = 0; i < C->numrhs + 1; i++ ) {
            PF_LongMultiUnpack(&l, 1, PF_LONG);
            C->rhs[i] = C->Buffer + l;
        }
        PF_LongMultiUnpack(C->CanCommu, C->numrhs + 1, PF_LONG);
        PF_LongMultiUnpack(C->NumTerms, C->numrhs + 1, PF_LONG);
        PF_LongMultiUnpack(C->numdum, C->numrhs + 1, PF_WORD);
        PF_LongMultiUnpack(C->dimension, C->numrhs + 1, PF_WORD);
        if ( C->MaxTreeSize > 0 )
            PF_LongMultiUnpack(C->boomlijst, (C->numtree + 1) * (sizeof(COMPTREE) / sizeof(int)), PF_INT);
/*
        #] Slave :
*/
    }
    return 0;
}

/*
    #] PF_BroadcastCBuf :
    #[ PF_BroadcastExpFlags :
*/

int PF_BroadcastExpFlags(void)
{
    WORD i;
    EXPRESSIONS e;
    if ( PF.me == MASTER ) {
/*
        #[ Master :
*/
        PF_PrepareLongMultiPack();
        PF_LongMultiPack(&AR.expflags, 1, PF_WORD);
        for ( i = 0; i < NumExpressions; i++ ) {
            e = &Expressions[i];
            PF_LongMultiPack(&e->counter,    1, PF_WORD);
            PF_LongMultiPack(&e->vflags,     1, PF_WORD);
            PF_LongMultiPack(&e->numdummies, 1, PF_WORD);
            PF_LongMultiPack(&e->numfactors, 1, PF_WORD);
#ifdef PF_DEBUG_BCAST_EXPRFLAGS
            MesPrint("» Broadcast ExprFlags: %s", AC.exprnames->namebuffer + e->name);
#endif
        }
/*
        #] Master :
*/
    }
    if ( PF_LongMultiBroadcast() ) return -1;
    if ( PF.me != MASTER ) {
/*
        #[ Slave :
*/
        PF_LongMultiUnpack(&AR.expflags, 1, PF_WORD);
        for ( i = 0; i < NumExpressions; i++ ) {
            e = &Expressions[i];
            PF_LongMultiUnpack(&e->counter,    1, PF_WORD);
            PF_LongMultiUnpack(&e->vflags,     1, PF_WORD);
            PF_LongMultiUnpack(&e->numdummies, 1, PF_WORD);
            PF_LongMultiUnpack(&e->numfactors, 1, PF_WORD);
        }
/*
        #] Slave :
*/
    }
    return 0;
}

/*
    #] PF_BroadcastExpFlags :
```

```
    #[ PF_SetScratch :
*/

static void PF_SetScratch(FILEHANDLE *f,POSITION *position)
{
    if(
            ( f->handle >= 0) && ISGEPOS(*position,f->POposition) &&
            ( ISGEPOSINC(*position,f->POposition,(f->POfull-f->PObuffer)*sizeof(WORD)) ==0 )
        )/*position is inside the buffer! SetScratch() will do nothing.*/
            f->POfull=f->PObuffer;/*force SetScratch() to re-read the position from the beginning:*/
    SetScratch(f,position);
}

/*
    #] PF_SetScratch :
    #[ PF_pushScratch :
*/

static int PF_pushScratch(FILEHANDLE *f)
{
    LONG size,RetCode;
    if ( f->handle < 0){
        /*Create the file*/
        if ( ( RetCode = CreateFile(f->name) ) >= 0 ) {
            f->handle = (WORD)RetCode;
            PUTZERO(f->filesize);
            PUTZERO(f->POposition);
        }
        else{
            MesPrint("Cannot create scratch file %s",f->name);
            return(-1);
        }
    }/*if ( f->handle < 0)*/
    size = (f->POfill-f->PObuffer)*sizeof(WORD);
    if( size > 0 ){
        SeekFile(f->handle,&(f->POposition),SEEK_SET);
        if ( WriteFile(f->handle,(UBYTE *)(f->PObuffer),size) != size ){
            MesPrint("Error while writing to disk. Disk full?");
            return(-1);
        }
        ADDPOS(f->filesize,size);
        ADDPOS(f->POposition,size);
        f->POfill = f->POfull=f->PObuffer;
    }/*if( size > 0 )*/
    return(0);
}

/*
    #] PF_pushScratch :
    #[ Broadcasting RHS expressions :
        #[ PF_WalkThroughExprMaster :
    Returns <=0 if the expression is ready, or dl+1;
*/

static int PF_WalkThroughExprMaster(FILEHANDLE *curfile, int dl)
{
    LONG l=0;
    for(;;){
        if(curfile->POfull-curfile->POfill < dl){
            POSITION pos;
            SeekScratch(curfile,&pos);
            PF_SetScratch(curfile,&pos);
        }/*if(curfile->POfull-curfile->POfill < dl)*/
        curfile->POfill+=dl;
        l+=dl;
        if( l >= PF.exprbufsize){
            if( l == PF.exprbufsize){
                if( *(curfile->POfill) == 0)/*expression is ready*/
                    return(0);
                }
            l-=PF.exprbufsize;
            curfile->POfill-=l;
            return l+1;
        }

        dl=*(curfile->POfill);
        if(dl == 0)
            return l-PF.exprbufsize;

        if(dl<0){/*compressed term*/
            if(curfile->POfull-curfile->POfill < 1){
                POSITION pos;
                SeekScratch(curfile,&pos);
                PF_SetScratch(curfile,&pos);
            }/*if(curfile->POfull-curfile->POfill < 1)*/
            dl=*(curfile->POfill+1)+2;
        }/*if(*(curfile->POfill)<0)*/
```

```
    }/*for(;;)*/
}

/*
        #] PF_WalkThroughExprMaster :
        #[ PF_WalkThroughExprSlave :
    Returns <=0 if the expression is ready, or dl+1;
*/

static int PF_WalkThroughExprSlave(FILEHANDLE *curfile, LONG *counter, int dl)
{
    LONG l=0;
    for(;;){
        if(curfile->POstop-curfile->POfill < dl){
            if(PF_pushScratch(curfile))
                return(-PF.exprbufsize-1);
        }
        curfile->POfill+=dl;
        curfile->POfull=curfile->POfill;
        l+=dl;
        if( l >= PF.exprbufsize){
            if( l == PF.exprbufsize){
                /*
                 * This access is valid because PF.exprbufsize+1 WORDs are
                 * broadcasted, this shortcut is not mandatory though. (TU 15 Sep 2011)
                 */
                if( *(curfile->POfill) == 0)/*expression is ready*/
                    return(0);
                }
            l-=PF.exprbufsize;
            curfile->POfill-=l;
            curfile->POfull=curfile->POfill;
            return l+1;
        }

        dl=*(curfile->POfill);
        if(dl == 0)
            return l-PF.exprbufsize;
        (*counter)++;
        if(dl<0){/*compressed term*/
            if(curfile->POstop-curfile->POfill < 1){
                if(PF_pushScratch(curfile))
                    return(-PF.exprbufsize-1);
            }
            /*
             * This access is always valid because PF.exprbufsize+1 WORDs are
             * broadcasted. (TU 15 Sep 2011)
             */
            dl=*(curfile->POfill+1)+2;
        }/*if(*(curfile->POfill)<0)*/
    }/*for(;;)*/
}

/*
        #] PF_WalkThroughExprSlave :
        #[ PF_rhsBCastMaster :
*/

static int PF_rhsBCastMaster(FILEHANDLE *curfile, EXPRESSIONS e)
{
    LONG l=1;/*PF_WalkThroughExpr returns length + 1*/
    SetScratch(curfile,&(e->onfile));
    do{
        /*
         * We need to broadcast PF.exprbufsize+1 WORDs because PF_WalkThroughExprSlave
         * may access to an additional 1 WORD. It is better to rewrite the routines
         * in such a way as to broadcast only PF.exprbufsize WORDs. (TU 15 Sep 2011)
         */
        if ( curfile->POfull - curfile->POfill < PF.exprbufsize + 1 ) {
            POSITION pos;
            SeekScratch(curfile,&pos);
            PF_SetScratch(curfile,&pos);
        }
        if ( PF_Bcast(curfile->POfill, (PF.exprbufsize + 1) * sizeof(WORD)) )
            return -1;
        l=PF_WalkThroughExprMaster(curfile,l-1);
    }while(l>0);
    if(l<0)/*The tail is extra, decrease POfill*/
        curfile->POfill-=l;
    return(0);
}

/*
        #] PF_rhsBCastMaster :
        #[ PF_rhsBCastSlave :
*/

```

```
static int PF_rhsBCastSlave(FILEHANDLE *curfile, EXPRESSIONS e)
{
    LONG l=1;/*PF_WalkThroughExpr returns length + 1*/
    LONG counter = 0;
    do{
        /*
         * We need to broadcast PF.exprbufsize+1 WORDs because PF_WalkThroughExprSlave
         * may access to an additional 1 WORD. It is better to rewrite the routines
         * in such a way as to broadcast only PF.exprbufsize WORDs. (TU 15 Sep 2011)
         */
        if ( curfile->POstop - curfile->POfill < PF.exprbufsize + 1 ) {
            if(PF_pushScratch(curfile))
                return(-1);
        }
        if ( PF_Bcast(curfile->POfill, (PF.exprbufsize + 1) * sizeof(WORD)) )
            return(-1);
        l = PF_WalkThroughExprSlave(curfile, &counter, l - 1);
    }while(l>0);
    if(l<0){/*The tail is extra, decrease POfill*/
        if(l<-PF.exprbufsize)/*error due to a PF_pushScratch() failure */
            return(-1);
        curfile->POfill-=l;
    }
    if ( curfile->handle >= 0 ) {
        if ( PF_pushScratch(curfile) ) return -1;
    }
    curfile->POfull=curfile->POfill;
    if ( curfile != AR.hidefile ) AR.InInBuf = curfile->POfull-curfile->PObuffer;
    else                          AR.InHiBuf = curfile->POfull-curfile->PObuffer;
    CHECK(counter == e->counter + 1);  /* The first term is the prototype. */
    return(0);
}

/*
        #] PF_rhsBCastSlave :
        #[ PF_BroadcastExpr :
*/

int PF_BroadcastExpr(EXPRESSIONS e, FILEHANDLE *file)
{
    if ( PF.me == MASTER ) {
        if ( PF_rhsBCastMaster(file, e) ) return -1;
#ifdef PF_DEBUG_BCAST_RHSEXPR
        MesPrint("» Broadcast RhsExpr: %s", AC.exprnames->namebuffer + e->name);
#endif
    }
    else {
        POSITION pos;
        SetEndHScratch(file, &pos);
        e->onfile = pos;
        if ( PF_rhsBCastSlave(file, e) ) return -1;
    }
    return 0;
}

/*
        #] PF_BroadcastExpr :
        #[ PF_BroadcastRHS :
*/

int PF_BroadcastRHS(void)
{
    int i;
    for ( i = 0; i < NumExpressions; i++ ) {
        EXPRESSIONS e = &Expressions[i];
        if ( !(e->vflags & ISINRHS) ) continue;
        switch ( e->status ) {
            case LOCALEXPRESSION:
            case SKIPLEXPRESSION:
            case DROPLEXPRESSION:
            case GLOBALEXPRESSION:
            case SKIPGEXPRESSION:
            case DROPGEXPRESSION:
            case HIDELEXPRESSION:
            case HIDEGEXPRESSION:
            case INTOHIDELEXPRESSION:
            case INTOHIDEGEXPRESSION:
                if ( PF_BroadcastExpr(e, AR.infile) ) return -1;
                break;
            case HIDDENLEXPRESSION:
            case HIDDENGEXPRESSION:
            case DROPHLEXPRESSION:
            case DROPHGEXPRESSION:
            case UNHIDELEXPRESSION:
            case UNHIDEGEXPRESSION:
                if ( PF_BroadcastExpr(e, AR.hidefile) ) return -1;
                break;
```

```
        }
    }
    if ( PF.me != MASTER )
        UpdatePositions();
    return 0;
}

/*
        #] PF_BroadcastRHS :
    #] Broadcasting RHS expressions :
    #[ InParallel mode :
        #[ PF_InParallelProcessor :
*/

int PF_InParallelProcessor(void)
{
    GETIDENTITY
    int i, next,tag;
    EXPRESSIONS e;
    /*
     * Skip expressions with zero terms. All the master and slaves need to
     * change the "partodo" flag.
     */
    if ( PF.numtasks >= 3 ) {
        for ( i = 0; i < NumExpressions; i++ ) {
            e = Expressions + i;
            if ( e->partodo > 0 && e->counter == 0 ) {
                e->partodo = 0;
            }
        }
    }
    if(PF.me == MASTER){
        if ( PF.numtasks >= 3 ) {
            partodoexr = (WORD*)Malloc1(sizeof(WORD)*(PF.numtasks+1),"PF_InParallelProcessor");
            for ( i = 0; i < NumExpressions; i++ ) {
                e = Expressions+i;
                if ( e->partodo <= 0 ) continue;
                switch(e->status){
                    case LOCALEXPRESSION:
                    case GLOBALEXPRESSION:
                    case UNHIDELEXPRESSION:
                    case UNHIDEGEXPRESSION:
                    case INTOHIDELEXPRESSION:
                    case INTOHIDEGEXPRESSION:
                        tag=PF_ANY_SOURCE;
                        next=PF_Wait4SlaveIP(&tag);
                        if(next<0)
                            return(-1);
                        if(tag == PF_DATA_MSGTAG){
                            PF_Statistics(PF_stats,0);
                            if(PF_Slave2MasterIP(next))
                                return(-1);
                        }
                        if(PF_Master2SlaveIP(next,e))
                            return(-1);
                        partodoexr[next]=i;
                        break;
                    default:
                        e->partodo = 0;
                        continue;
                }/*switch(e->status)*/
            }/*for ( i = 0; i < NumExpressions; i++ )*/
            /*Here some slaves are working, other are waiting on PF_Send.
                Wait all of them.*/
            /*At this point no new slaves may be launched so PF_WaitAllSlaves()
                does not modify partodoexr[].*/
            if(PF_WaitAllSlaves())
                return(-1);

            if ( AC.CollectFun ) AR.DeferFlag = 0;
            if(partodoexr){
                M_free(partodoexr,"PF_InParallelProcessor");
                partodoexr=NULL;
            }/*if(partodoexr)*/
        }/*if ( PF.numtasks >= 3 ) */
        else {
            for ( i = 0; i < NumExpressions; i++ ) {
                Expressions[i].partodo = 0;
            }
        }
        return(0);
    }/*if(PF.me == MASTER)*/
    /*Slave:*/
    if(PF_Wait4MasterIP(PF_EMPTY_MSGTAG))
        return(-1);
    /*master is ready to listen to me*/
    do{
```

```
        WORD *oldwork= AT.WorkPointer;
        tag=PF_ReadMaster();/*reads directly to its scratch!*/
        if(tag<0)
            return(-1);
        if(tag == PF_DATA_MSGTAG){
            oldwork = AT.WorkPointer;

            /* For redefine statements. */
            if ( AC.numpfirstnum > 0 ) {
                int j;
                for ( j = 0; j < AC.numpfirstnum; j++ ) {
                    AC.inputnumbers[j] = -1;
                }
            }

            if(PF_DoOneExpr())/*the processor*/
                return(-1);
            if(PF_Wait4MasterIP(PF_DATA_MSGTAG))
                return(-1);
            if(PF_Slave2MasterIP(PF.me))/*both master and slave*/
                return(-1);
            AT.WorkPointer=oldwork;
        }/*if(tag == PF_DATA_MSGTAG)*/
    }while(tag!=PF_EMPTY_MSGTAG);
    PF.exprtodo=-1;
    return(0);
}/*PF_InParallelProcessor*/

/*
        #] PF_InParallelProcessor :
        #[ PF_Wait4MasterIP :
*/

static int PF_Wait4MasterIP(int tag)
{
    int follow = 0;
    LONG cpu,space = 0;

    if(PF.log){
        fprintf(stderr,"[%d] Starting to send to Master\n",PF.me);
        fflush(stderr);
    }

    PF_PreparePack();
    cpu = TimeCPU(1);
    PF_Pack(&cpu               ,1,PF_LONG);
    PF_Pack(&space             ,1,PF_LONG);
    PF_Pack(&PF_linterms       ,1,PF_LONG);
    PF_Pack(&(AM.S0->GenTerms) ,1,PF_LONG);
    PF_Pack(&(AM.S0->TermsLeft),1,PF_LONG);
    PF_Pack(&follow            ,1,PF_INT );

    if(PF.log){
        fprintf(stderr,"[%d] Now sending with tag = %d\n",PF.me,tag);
        fflush(stderr);
    }

    PF_Send(MASTER, tag);

    if(PF.log){
        fprintf(stderr,"[%d] returning from send\n",PF.me);
        fflush(stderr);
    }
    return(0);
}
/*
        #] PF_Wait4MasterIP :
        #[ PF_DoOneExpr :
*/

static int PF_DoOneExpr(void)/*the processor*/
{
                GETIDENTITY
                EXPRESSIONS e;
                int i;
                WORD *term;
                POSITION position, outposition;
                FILEHANDLE *fi, *fout;
                LONG dd = 0;
                WORD oldBracketOn = AR.BracketOn;
                WORD *oldBrackBuf = AT.BrackBuf;
                WORD oldbracketindexflag = AT.bracketindexflag;
                e = Expressions + PF.exprtodo;
                i = PF.exprtodo;
                AR.CurExpr = i;
                AR.SortType = AC.SortType;
                AR.expchanged = 0;
```

```
                if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
                    AR.BracketOn = 1;
                    AT.BrackBuf = AM.BracketFactors;
                    AT.bracketindexflag = 1;
                }

                position = AS.OldOnFile[i];
                if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENGEXPRESSION ) {
                    AR.GetFile = 2; fi = AR.hidefile;
                }
                else {
                    AR.GetFile = 0; fi = AR.infile;
                }
/*
                PUTZERO(fi->POposition);
                if ( fi->handle >= 0 ) {
                    fi->POfill = fi->POfull = fi->PObuffer;
                }
*/
                SetScratch(fi,&position);
                term = AT.WorkPointer;
                AR.CompressPointer = AR.CompressBuffer;
                AR.CompressPointer[0] = 0;
                AR.KeptInHold = 0;
                if ( GetTerm(BHEAD term) <= 0 ) {
                    MesPrint("Expression %d has problems in scratchfile",i);
                    Terminate(-1);
                }
                if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
                term[3] = i;
                PUTZERO(outposition);
                fout = AR.outfile;
                fout->POfill = fout->POfull = fout->PObuffer;
                fout->POposition = outposition;
                if ( fout->handle >= 0 ) {
                    fout->POposition = outposition;
                }
/*
                The next statement is needed because we need the system
                to believe that the expression is at position zero for
                the moment. In this worker, with no memory of other expressions,
                it is. This is needed for when a bracket index is made
                because there e->onfile is an offset. Afterwards, when the
                expression is written to its final location in the masters
                output e->onfile will get its real value.
*/
                PUTZERO(e->onfile);
                if ( PutOut(BHEAD term,&outposition,fout,0) < 0 ) return -1;

                AR.DeferFlag = AC.ComDefer;

/*              AR.sLevel = AB[0]->R.sLevel;*/
                term = AT.WorkPointer;
                NewSort(BHEAD0);
                AR.MaxDum = AM.IndDum;
                AN.ninterms = 0;
                while ( GetTerm(BHEAD term) ) {
                  SeekScratch(fi,&position);
                  AN.ninterms++; dd = AN.deferskipped;
                  if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
                      StoreTerm(BHEAD term);
                  }
                  else {
                  if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)sizeof(WORD))) ) {
                    if ( GetMoreTerms(term) < 0 ) {
                      LowerSortLevel(); return(-1);
                    }
                    SeekScratch(fi,&position);
                  }
                  AT.WorkPointer = term + *term;
                  AN.RepPoint = AT.RepCount + 1;
                  if ( AR.DeferFlag ) {
                    AR.CurDum = AN.IndDum = Expressions[PF.exprtodo].numdummies;
                  }
                  else {
                    AN.IndDum = AM.IndDum;
                    AR.CurDum = ReNumber(BHEAD term);
                  }
                  if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
                  if ( AN.ncmod ) {
                    if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
                    else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
                  }
                  else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
                  if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
                        && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
                        PolyFunClean(BHEAD term);
```

```
                    }
                    if ( Generator(BHEAD term,0) ) {
                      LowerSortLevel(); return(-1);
                    }
                    AN.ninterms += dd;
                    }
                    SetScratch(fi,&position);
                    if ( fi == AR.hidefile ) {
                      AR.InHiBuf = (fi->POfull-fi->PObuffer)
                          -DIFBASE(position,fi->POposition)/sizeof(WORD);
                    }
                    else {
                      AR.InInBuf = (fi->POfull-fi->PObuffer)
                          -DIFBASE(position,fi->POposition)/sizeof(WORD);
                    }
                  }
                  AN.ninterms += dd;
                  if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) return(-1);
                  e->numdummies = AR.MaxDum - AM.IndDum;
                  AR.BracketOn = oldBracketOn;
                  AT.BrackBuf = oldBrackBuf;
                  if ( ( e->vflags & TOBEFACTORED ) != 0 )
                          poly_factorize_expression(e);
                  else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
                   && ( ( e->vflags & ISFACTORIZED ) != 0 ) )
                          poly_unfactorize_expression(e);
                  if ( AM.S0->TermsLeft )   e->vflags &= ~ISZERO;
                  else                      e->vflags |= ISZERO;
                  if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
/*                if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
                  if ( AR.expchanged )    AR.expflags |= ISUNMODIFIED;*/
                  AR.GetFile = 0;
                  AT.bracketindexflag = oldbracketindexflag;

                  fout->POfull = fout->POfill;
    return(0);
}

/*
        #] PF_DoOneExpr :
        #[ PF_Slave2MasterIP :
*/

typedef struct bufIPstruct {
    LONG i;
    struct ExPrEsSiOn e;
} bufIPstruct_t;

static int PF_Slave2MasterIP(int src)/*both master and slave*/
{
    EXPRESSIONS e;
    bufIPstruct_t exprData;
    int i,l;
    FILEHANDLE *fout=AR.outfile;
    POSITION pos;
    /*Here we know the length of data to send in advance:
        slave has the only one expression in its scratch file, and it sends
        this information to the master.*/
    if(PF.me != MASTER){/*slave*/
        e = Expressions + PF.exprtodo;
        /*Fill in the expression data:*/
        memcpy(&(exprData.e), e, sizeof(struct ExPrEsSiOn));
        SeekScratch(fout,&pos);
        exprData.i=BASEPOSITION(pos);
        /*Send the metadata:*/
        if(PF_RawSend(MASTER,&exprData,sizeof(bufIPstruct_t),0))
            return(-1);
        i=exprData.i;
        SETBASEPOSITION(pos,0);
        do{
            int blen=PF.exprbufsize*sizeof(WORD);
            if(i<blen)
                blen=i;
            l=PF_SendChunkIP(fout,&pos, MASTER, blen);
            /*Here always l == blen!*/
            if(l<0)
                return(-1);
            ADDPOS(pos,l);
            i-=l;
        }while(i>0);
        if ( fout->handle >= 0 ) { /* Now get rid of the file */
            CloseFile(fout->handle);
            fout->handle = -1;
            remove(fout->name);
            PUTZERO(fout->POposition);
            PUTZERO(fout->filesize);
            fout->POfill = fout->POfull = fout->PObuffer;
```

```
        }
        /* Now handle redefined preprocessor variables. */
        if ( AC.numpfirstnum > 0 ) {
            PF_PrepareLongSinglePack();
            PF_PackRedefinedPreVars();
            PF_LongSingleSend(MASTER, PF_MISC_MSGTAG);
        }
        return(0);
    }/*if(PF.me != MASTER)*/
    /*Master*/
    /*partodoexr[src] is the number of expression.*/
    e = Expressions +partodoexr[src];
    /*Get metadata:*/
    if (PF_RawRecv(&src, &exprData,sizeof(bufIPstruct_t),&i)!= sizeof(bufIPstruct_t))
        return(-1);
    /*Fill in the expression data:*/
/*  memcpy(e, &(exprData.e), sizeof(struct ExPrEsSiOn)); */
    e->counter    = exprData.e.counter;
    e->vflags     = exprData.e.vflags;
    e->numdummies = exprData.e.numdummies;
    e->numfactors = exprData.e.numfactors;
    if ( !(e->vflags & ISZERO) )       AR.expflags |= ISZERO;
    if ( !(e->vflags & ISUNMODIFIED) ) AR.expflags |= ISUNMODIFIED;
    SeekScratch(fout,&pos);
    e->onfile = pos;
    i=exprData.i;
    while(i>0){
        int blen=PF.exprbufsize*sizeof(WORD);
        if(i<blen)
            blen=i;
        l=PF_RecvChunkIP(fout,src,blen);
        /*Here always l == blen!*/
        if(l<0)
            return(-1);
        i-=l;
    }
    /* Now handle redefined preprocessor variables. */
    if ( AC.numpfirstnum > 0 ) {
        PF_LongSingleReceive(src, PF_MISC_MSGTAG, NULL, NULL);
        PF_UnpackRedefinedPreVars();
    }
    return(0);
}

/*
        #] PF_Slave2MasterIP :
        #[ PF_Master2SlaveIP :
*/

static int PF_Master2SlaveIP(int dest, EXPRESSIONS e)
{
    bufIPstruct_t exprData;
    FILEHANDLE *fi;
    POSITION pos;
    int l;
    LONG ll=0,count=0;
    WORD *t;
    if(e==NULL){/*Say to the slave that no more job:*/
        if(PF_RawSend(dest,&exprData,sizeof(bufIPstruct_t),PF_EMPTY_MSGTAG))
            return(-1);
        return(0);
    }
    memcpy(&(exprData.e), e, sizeof(struct ExPrEsSiOn));
    exprData.i=e-Expressions;
    if ( AC.StatsFlag && AC.OldParallelStats ) {
        MesPrint("");
        MesPrint(" Sending expression %s to slave %d",EXPRNAME(exprData.i),dest);
    }
    if(PF_RawSend(dest,&exprData,sizeof(bufIPstruct_t),PF_DATA_MSGTAG))
        return(-1);
    if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENGEXPRESSION )
        fi = AR.hidefile;
    else
        fi = AR.infile;
    pos=e->onfile;
    SetScratch(fi,&pos);
    do{
        l=PF_SendChunkIP(fi, &pos, dest, PF.exprbufsize*sizeof(WORD));
        if(l<0)
            return(-1);
        t=fi->PObuffer+ (DIFBASE(pos,fi->POposition))/sizeof(WORD);
        ll=PF_WalkThrough(t,ll,l/sizeof(WORD),&count);
        ADDPOS(pos,l);
    }while(ll>-2);
    return(0);
}
```

```
/*
        #] PF_Master2SlaveIP :
        #[ PF_ReadMaster :
*/

static int PF_ReadMaster(void)/*reads directly to its scratch!*/
{
    bufIPstruct_t exprData;
    int tag,m=MASTER;
    EXPRESSIONS e;
    FILEHANDLE *fi;
    POSITION pos;
    LONG count=0;
    WORD *t;
    LONG ll=0;
    int l;
    /*Get metadata:*/
    if (PF_RawRecv(&m, &exprData,sizeof(bufIPstruct_t),&tag)!= sizeof(bufIPstruct_t))
        return(-1);

    if(tag == PF_EMPTY_MSGTAG)/*No data, no job*/
        return(tag);

    /*data expected, tag must be == PF_DATA_MSTAG!*/
    PF.exprtodo=exprData.i;
    e=Expressions + PF.exprtodo;
    /*Fill in the expression data:*/
/*  memcpy(e, &(exprData.e), sizeof(struct ExPrEsSiOn)); */
    if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENGEXPRESSION )
        fi = AR.hidefile;
    else
        fi = AR.infile;
    SetEndHScratch(fi,&pos);
    e->onfile=AS.OldOnFile[PF.exprtodo]=pos;

    do{
        l=PF_RecvChunkIP(fi,MASTER,PF.exprbufsize*sizeof(WORD));
        if(l<0)
            return(-1);
        t=fi->POfull-l/sizeof(WORD);
        ll=PF_WalkThrough(t,ll,l/sizeof(WORD),&count);
    }while(ll>-2);
    /*Now -ll-2 is the number of "extra" elements transferred from the master.*/
    fi->POfull-=-ll-2;
    fi->POfill=fi->POfull;
    return(PF_DATA_MSGTAG);
}

/*
        #] PF_ReadMaster :
        #[ PF_SendChunkIP :
    thesize is in bytes. Returns the number of sent bytes or <0 on error:
*/

static int PF_SendChunkIP(FILEHANDLE *curfile, POSITION *position, int to, LONG thesize)
{
    LONG l=thesize;
    if(
        ISLESSPOS(*position,curfile->POposition) ||
        ISGEPOSINC(*position,curfile->POposition,
        ((curfile->POfull-curfile->PObuffer)*sizeof(WORD)-thesize) )
    ){
        if(curfile->handle< 0)
            l=(curfile->POfull-curfile->PObuffer)*sizeof(WORD) - (LONG)(position->p1);
        else{
            PF_SetScratch(curfile,position);
            if(
                ISGEPOSINC(*position,curfile->POposition,
                ((curfile->POfull-curfile->PObuffer)*sizeof(WORD)-thesize) )
                )
            l=(curfile->POfull-curfile->PObuffer)*sizeof(WORD) - (LONG)position->p1;
        }
    }
    /*Now we are able to sent l bytes from the
        curfile->PObuffer[position-curfile->POposition]*/
    if(PF_RawSend(to,curfile->PObuffer+ (DIFBASE(*position,curfile->POposition))/sizeof(WORD),l,0))
        return(-1);
    return(l);
}

/*
        #] PF_SendChunkIP :
        #[ PF_RecvChunkIP :
    thesize is in bytes. Returns the number of sent bytes or <0 on error:
*/

static int PF_RecvChunkIP(FILEHANDLE *curfile, int from, LONG thesize)
```

```
{
    LONG receivedBytes;

    if( (LONG)((curfile->POstop - curfile->POfull)*sizeof(WORD)) < thesize )
        if(PF_pushScratch(curfile))
            return(-1);
    /*Now there is enough space from curfile->POfill to curfile->POstop*/
    {/*Block:*/
        int tag=0;
        receivedBytes=PF_RawRecv(&from,curfile->POfull,thesize,&tag);
    }/*:Block*/
    if(receivedBytes >= 0 ){
        curfile->POfull+=receivedBytes/sizeof(WORD);
        curfile->POfill=curfile->POfull;
    }/*if(receivedBytes >= 0 )*/
    return(receivedBytes);
}

/*
        #] PF_RecvChunkIP :
        #[ PF_WalkThrough :
    Returns:
    >=  0 -- initial offset,
        -1 -- the first element of t contains the length of the tail of compressed term,
    <= -2 -- -(d+2), where d is the number of extra transferred elements.
    Expects:
    l -- initial offset or -1,
    chunk -- number of transferred elements (not bytes!)
    *count -- incremented each time a new term is found
*/

static int PF_WalkThrough(WORD *t, LONG l, LONG chunk, LONG *count)
{
    if(l<0) /*==-1!*/
        l=(*t)+1;/*the first element of t contains the length of
                        the tail of compressed term*/
    else{
        if(l>=chunk)/*next term is out of the chunk*/
            return(l-chunk);
        t+=l;
        chunk-=l;/*note, l was less than chunk so chunk >0!*/
        l=*t;
    }
    /*Main loop:*/
    while(l!=0){
        if(l>0){/*an offset to the next term*/
            if(l<chunk){
                t+=l;
                chunk-=l;/*note, l was less than chunk so chunk >0!*/
                l=*t;
                (*count)++;
            }/*if(l<chunk)*/
            else
                return(l-chunk);
        }/*if(l>0)*/
        else{ /* l<0 */
            if(chunk < 2)/*i.e., chunk == 1*/
                return(-1);/*the first WORD in the next chunk is length of the tail of the compressed term*/
            l=*(t+1)+2;/*+2 since
                    1. t points to the length field -1,
                    2. the size of a tail of compressed term is equal to the number of WORDs in this tail*/
        }
    }/*while(l!=0)*/
    return(-1-chunk);/* -(2+(chunk-1)), chunk>0 ! */
}

/*
        #] PF_WalkThrough :
    #] InParallel mode :
    #[ PF_SendFile :
*/

#define PF_SNDFILEBUFSIZE 4096

int PF_SendFile(int to, FILE *fd)
{
    size_t len=0;
    if(fd == NULL){
        if(PF_RawSend(to,&to,sizeof(int),PF_EMPTY_MSGTAG))
            return(-1);
        return(0);
    }
    for(;;){
        char buf[PF_SNDFILEBUFSIZE];
        size_t l;
```

```
        l=fread(buf, 1, PF_SNDFILEBUFSIZE, fd);
        len+=l;
        if(l==PF_SNDFILEBUFSIZE){
            if(PF_RawSend(to,buf,PF_SNDFILEBUFSIZE,PF_BUFFER_MSGTAG))
                return(-1);
        }
        else{
            if(PF_RawSend(to,buf,l,PF_ENDBUFFER_MSGTAG))
                return(-1);
            break;
        }
    }/*for(;;)*/
    return(len);
}

/*
    #] PF_SendFile :
    #[ PF_RecvFile :
*/

int PF_RecvFile(int from, FILE *fd)
{
    size_t len=0;
    int tag;
    do{
        char buf[PF_SNDFILEBUFSIZE];
        int l;
            l=PF_RawRecv(&from,buf,PF_SNDFILEBUFSIZE,&tag);
            if(l<0)
                return(-1);
            if(tag == PF_EMPTY_MSGTAG)
                return(-1);

            if( fwrite(buf,l,1,fd)!=1 )
                return(-1);
            len+=l;
    }while(tag!=PF_ENDBUFFER_MSGTAG);
    return(len);
}

/*
    #] PF_RecvFile :
    #[ Synchronised output :
        #[ Explanations :
*/

/*
 * If the master and slaves output statistics or error messages to the same stream
 * or file (e.g., the standard output or the log file) simultaneously, then
 * a mixing of their outputs can occur. To avoid this, TFORM uses a lock of
 * ErrorMessageLock, but there is no locking functionality in the original MPI
 * specification. We need to synchronise the output from the master and slaves.
 *
 * The idea of the synchronised output (by, e.g., MesPrint()) implemented here is
 *   Slaves:
 *     1. Save the output by WriteFile() (set to PF_WriteFileToFile())
 *        into some buffers between MLOCK(ErrorMessageLock) and
 *        MUNLOCK(ErrorMessageLock), which call PF_MLock() and PF_MUnlock(),
 *        respectively. The output for AM.StdOut and AC.LogHandle are saved to
 *        the buffers.
 *     2. At MUNLOCK(ErrorMessageLock), send the output in the buffer to the master,
 *        with PF_STDOUT_MSGTAG or PF_LOG_MSGTAG.
 *   Master:
 *     1. Receive the buffered output from slaves, and write them by
 *        WriteFileToFile().
 *   The main problem is how and where the master receives messages from
 *   the slaves (PF_ReceiveErrorMessage()). For this purpose there are three
 *   helper functions: PF_CatchErrorMessages() and PF_CatchErrorMessagesForAll()
 *   which remove messages with PF_STDOUT_MSGTAG or PF_LOG_MSGTAG from the top
 *   of the message queue, and PF_ProbeWithCatchingErrorMessages() which is same as
 *   PF_Probe() except removing these messages.
 */

/*
        #] Explanations :
        #[ Variables :
*/

static int errorMessageLock = 0;     /* (slaves) The lock count. See PF_MLock() and PF_MUnlock(). */
static Vector(UBYTE, stdoutBuffer);  /* (slaves) The buffer for AM.StdOut. */
static Vector(UBYTE, logBuffer);     /* (slaves) The buffer for AC.LogHandle. */
#define recvBuffer logBuffer         /* (master) The buffer for receiving messages. */

/*
 * If PF_ENABLE_STDOUT_BUFFERING is defined, the master performs the line buffering
 * (using stdoutBuffer) at PF_WriteFileToFile().
 */
```

```
#ifndef PF_ENABLE_STDOUT_BUFFERING
#ifdef UNIX
#define PF_ENABLE_STDOUT_BUFFERING
#endif
#endif

/*
        #] Variables :
        #[ PF_MLock :
*/

void PF_MLock(void)
{
    /* Only on slaves. */
    if ( errorMessageLock++ > 0 ) return;
    VectorClear(stdoutBuffer);
    VectorClear(logBuffer);
}

/*
        #] PF_MLock :
        #[ PF_MUnlock :
*/

void PF_MUnlock(void)
{
    /* Only on slaves. */
    if ( --errorMessageLock > 0 ) return;
    if ( !VectorEmpty(stdoutBuffer) ) {
        PF_RawSend(MASTER, VectorPtr(stdoutBuffer), VectorSize(stdoutBuffer), PF_STDOUT_MSGTAG);
    }
    if ( !VectorEmpty(logBuffer) ) {
        PF_RawSend(MASTER, VectorPtr(logBuffer), VectorSize(logBuffer), PF_LOG_MSGTAG);
    }
}

/*
        #] PF_MUnlock :
        #[ PF_WriteFileToFile :
*/

LONG PF_WriteFileToFile(int handle, UBYTE *buffer, LONG size)
{
    if ( PF.me != MASTER && errorMessageLock > 0 ) {
        if ( handle == AM.StdOut ) {
            VectorPushBacks(stdoutBuffer, buffer, size);
            return size;
        }
        else if ( handle == AC.LogHandle ) {
            VectorPushBacks(logBuffer, buffer, size);
            return size;
        }
    }
#ifdef PF_ENABLE_STDOUT_BUFFERING
    /*
     * On my computer, sometimes a single linefeed "\n" sent to the standard
     * output is ignored on the execution of mpiexec. A typical example is:
     *   $ cat foo.c
     *     #include <unistd.h>
     *     int main() {
     *       write(1, "    ", 4);
     *       write(1, "\n", 1);
     *       write(1, "    ", 4);
     *       write(1, "123\n", 4);
     *       return 0;
     *     }
     * or even as a shell script:
     *   $ cat foo.sh
     *     #! bin/sh
     *     printf "    "
     *     printf "\n"
     *     printf "    "
     *     printf "123\n"
     * When I ran it on mpiexec
     *   $ while :; do mpiexec -np 1 ./foo.sh; done
     * I observed the single linefeed (printf "\n") was sometimes ignored. Even
     * though this phenomenon might be specific to my environment, I added this
     * code because someone may encounter a similar phenomenon and feel it
     * frustrating. (TU 16 Jun 2011)
     *
     * Phenomenon:
     *   A single linefeed sent to the standard output occasionally ignored
     *   on mpiexec.
     *
     * Environment:
     *   openSUSE 11.4 (x86_64)
     *   kernel: 2.6.37.6-0.5-desktop
```

```
     *   gcc: 4.5.1 20101208
     *   mpich2-1.3.2p1 configured with '--enable-shared --with-pm=smpd'
     *
     * Solution:
     *   In Unix (in which Uwrite() calls write() system call without any buffering),
     *   we perform the line buffering here. A single linefeed is also buffered.
     *
     * XXX:
     *   At the end of the program the buffered output (text without LF) will not be flushed,
     *   i.e., will not be written to the standard output. This is not problematic at a normal run.
     *   The buffer can be explicitly flushed by PF_FlushStdOutBuffer().
     */
    if ( PF.me == MASTER && handle == AM.StdOut ) {
        size_t oldsize;
        /* Assume the newline character is LF (when UNIX is defined). */
        if ( (size > 0 && buffer[size - 1] != LINEFEED) || (size == 1 && buffer[0] == LINEFEED) ) {
            VectorPushBacks(stdoutBuffer, buffer, size);
            return size;
        }
        if ( (oldsize = VectorSize(stdoutBuffer)) > 0 ) {
            LONG ret;
            VectorPushBacks(stdoutBuffer, buffer, size);
            ret = WriteFileToFile(handle, VectorPtr(stdoutBuffer), VectorSize(stdoutBuffer));
            VectorClear(stdoutBuffer);
            if ( ret < 0 ) {
                return ret;
            }
            else if ( ret < (LONG)oldsize ) {
                return 0;  /* This means the buffered output in previous calls is lost. */
            }
            else {
                return ret - (LONG)oldsize;
            }
        }
    }
#endif
    return WriteFileToFile(handle, buffer, size);
}

/*
        #] PF_WriteFileToFile :
        #[ PF_FlushStdOutBuffer :
*/

void PF_FlushStdOutBuffer(void)
{
#ifdef PF_ENABLE_STDOUT_BUFFERING
    if ( PF.me == MASTER && VectorSize(stdoutBuffer) > 0 ) {
        WriteFileToFile(AM.StdOut, VectorPtr(stdoutBuffer), VectorSize(stdoutBuffer));
        VectorClear(stdoutBuffer);
    }
#endif
}

/*
        #] PF_FlushStdOutBuffer :
        #[ PF_ReceiveErrorMessage :
*/

static void PF_ReceiveErrorMessage(int src, int tag)
{
    /* Only on the master. */
    int size;
    int ret = PF_RawProbe(&src, &tag, &size);
    CHECK(ret == 0);
    switch ( tag ) {
        case PF_STDOUT_MSGTAG:
        case PF_LOG_MSGTAG:
            VectorReserve(recvBuffer, size);
            ret = PF_RawRecv(&src, VectorPtr(recvBuffer), size, &tag);
            CHECK(ret == size);
            if ( size > 0 ) {
                int handle = (tag == PF_STDOUT_MSGTAG) ? AM.StdOut : AC.LogHandle;
#ifdef PF_ENABLE_STDOUT_BUFFERING
                if ( handle == AM.StdOut ) PF_WriteFileToFile(handle, VectorPtr(recvBuffer), size);
                else
#endif
                WriteFileToFile(handle, VectorPtr(recvBuffer), size);
            }
            break;
    }
}

/*
        #] PF_ReceiveErrorMessage :
        #[ PF_CatchErrorMessages :
*/
```

```
04516
04525 static void PF_CatchErrorMessages(int *src, int *tag)
04526 {
04527     /* Only on the master. */
04528     for (;;) {
04529         int asrc = *src;
04530         int atag = *tag;
04531         int ret = PF_RawProbe(&asrc, &atag, NULL);
04532         CHECK(ret == 0);
04533         if ( atag == PF_STDOUT_MSGTAG || atag == PF_LOG_MSGTAG ) {
04534             PF_ReceiveErrorMessage(asrc, atag);
04535             continue;
04536         }
04537         *src = asrc;
04538         *tag = atag;
04539         break;
04540     }
04541 }
04542
04543 /*
04544         #] PF_CatchErrorMessages :
04545         #[ PF_CatchErrorMessagesForAll :
04546 */
04547
04552 static void PF_CatchErrorMessagesForAll(void)
04553 {
04554     /* Only on the master. */
04555     int i;
04556     for ( i = 1; i < PF.numtasks; i++ ) {
04557         int src = i;
04558         int tag = PF_ANY_MSGTAG;
04559         PF_CatchErrorMessages(&src, &tag);
04560     }
04561 }
04562
04563 /*
04564         #] PF_CatchErrorMessagesForAll :
04565         #[ PF_ProbeWithCatchingErrorMessages :
04566 */
04567
04577 static int PF_ProbeWithCatchingErrorMessages(int *src)
04578 {
04579     for (;;) {
04580         int newsrc = *src;
04581         int tag = PF_Probe(&newsrc);
04582         if ( tag == PF_STDOUT_MSGTAG || tag == PF_LOG_MSGTAG ) {
04583             PF_ReceiveErrorMessage(newsrc, tag);
04584             continue;
04585         }
04586         if ( tag > 0 ) *src = newsrc;
04587         return tag;
04588     }
04589 }
04590
04591 /*
04592         #] PF_ProbeWithCatchingErrorMessages :
04593         #[ PF_FreeErrorMessageBuffers :
04594 */
04595
04601 void PF_FreeErrorMessageBuffers(void)
04602 {
04603     VectorFree(stdoutBuffer);
04604     VectorFree(logBuffer);
04605 }
04606
04607 /*
04608         #] PF_FreeErrorMessageBuffers :
04609     #] Synchronised output :
04610 */
```

4.75 parallel.h File Reference

`#include <mpi.h>`
Include dependency graph for parallel.h:

Data Structures

- struct PF_BUFFER
- struct ParallelVars

Macros

- #define MASTER 0
- #define PF_RESET 0
- #define PF_TIME 1
- #define PF_TERM_MSGTAG 10 /∗ master -> slave: sending terms ∗/
- #define PF_ENDSORT_MSGTAG 11 /∗ master -> slave: no more terms to be distributed, slave -> master: after EndSort() ∗/
- #define PF_DOLLAR_MSGTAG 12 /∗ slave -> master: sending $-variables ∗/
- #define PF_BUFFER_MSGTAG 20 /∗ slave -> master: sending sorted terms, or in PF_SendFile()/PF_RecvFile() ∗/
- #define PF_ENDBUFFER_MSGTAG 21 /∗ same as PF_BUFFER_MSGTAG, but indicates the end of operation ∗/
- #define PF_READY_MSGTAG 30 /∗ slave -> master: slave is idle and can accept terms ∗/
- #define PF_DATA_MSGTAG 50 /∗ InParallel, DoCheckpoint() ∗/
- #define PF_EMPTY_MSGTAG 52 /∗ InParallel, DoCheckpoint(), PF_SendFile(), PF_RecvFile() ∗/
- #define PF_STDOUT_MSGTAG 60 /∗ slave -> master: sending text to the stdout ∗/
- #define PF_LOG_MSGTAG 61 /∗ slave -> master: sending text to the log file ∗/
- #define PF_OPT_MCTS_MSGTAG 70 /∗ master <-> slave: optimization ∗/
- #define PF_OPT_HORNER_MSGTAG 71 /∗ master <-> slave: optimization ∗/
- #define PF_OPT_COLLECT_MSGTAG 72 /∗ slave -> master: optimization ∗/
- #define PF_MISC_MSGTAG 100
- #define GNUC_PREREQ(major, minor, patchlevel) 0
- #define indices ((INDICES)(AC.IndexList.lijst))
- #define PF_ANY_SOURCE MPI_ANY_SOURCE
- #define PF_ANY_MSGTAG MPI_ANY_TAG
- #define PF_COMM MPI_COMM_WORLD
- #define PF_BYTE MPI_BYTE
- #define PF_INT MPI_INT
- #define PF_LongMultiPack(buffer, count, type) PF_LongMultiPackImpl(buffer, count, sizeof_datatype(type), type)
- #define PF_LongMultiUnpack(buffer, count, type) PF_LongMultiUnpackImpl(buffer, count, sizeof_datatype(type), type)

Typedefs

- typedef struct ParallelVars **PARALLELVARS**

Functions

- int PF_ISendSbuf (int, int)
- int PF_Bcast (void ∗buffer, int count)
- int PF_RawSend (int, void ∗, LONG, int)
- LONG PF_RawRecv (int ∗, void ∗, LONG, int ∗)
- int PF_PreparePack (void)
- int PF_Pack (const void ∗buffer, size_t count, MPI_Datatype type)
- int PF_Unpack (void ∗buffer, size_t count, MPI_Datatype type)
- int PF_PackString (const UBYTE ∗str)
- int PF_UnpackString (UBYTE ∗str)
- int PF_Send (int to, int tag)
- int PF_Receive (int src, int tag, int ∗psrc, int ∗ptag)
- int PF_Broadcast (void)
- int PF_PrepareLongSinglePack (void)
- int PF_LongSinglePack (const void ∗buffer, size_t count, MPI_Datatype type)
- int PF_LongSingleUnpack (void ∗buffer, size_t count, MPI_Datatype type)
- int PF_LongSingleSend (int to, int tag)
- int PF_LongSingleReceive (int src, int tag, int ∗psrc, int ∗ptag)
- int PF_PrepareLongMultiPack (void)
- int PF_LongMultiPackImpl (const void ∗buffer, size_t count, size_t eSize, MPI_Datatype type)
- int PF_LongMultiUnpackImpl (void ∗buffer, size_t count, size_t eSize, MPI_Datatype type)
- int PF_LongMultiBroadcast (void)
- int PF_EndSort (void)
- WORD PF_Deferred (WORD ∗, WORD)
- int PF_Processor (EXPRESSIONS, WORD, WORD)
- int PF_Init (int ∗, char ∗∗∗)
- int PF_Terminate (int)
- LONG PF_GetSlaveTimes (void)
- LONG PF_BroadcastNumber (LONG)
- void PF_BroadcastBuffer (WORD ∗∗buffer, LONG ∗length)
- int PF_BroadcastString (UBYTE ∗)
- int PF_BroadcastPreDollar (WORD ∗∗, LONG ∗, int ∗)
- int PF_CollectModifiedDollars (void)
- int PF_BroadcastModifiedDollars (void)
- int PF_BroadcastRedefinedPreVars (void)
- int PF_BroadcastCBuf (int bufnum)
- int PF_BroadcastExpFlags (void)
- int PF_StoreInsideInfo (void)
- int PF_RestoreInsideInfo (void)
- int PF_BroadcastExpr (EXPRESSIONS e, FILEHANDLE ∗file)
- int PF_BroadcastRHS (void)
- int PF_InParallelProcessor (void)
- int PF_SendFile (int to, FILE ∗fd)
- int PF_RecvFile (int from, FILE ∗fd)
- void PF_MLock (void)
- void PF_MUnlock (void)
- LONG PF_WriteFileToFile (int, UBYTE ∗, LONG)
- void PF_FlushStdOutBuffer (void)

Variables

- PARALLELVARS PF
- LONG PF_maxDollarChunkSize

4.75.1 Detailed Description

Header file with things relevant to ParForm.

Definition in file parallel.h.

4.75.2 Macro Definition Documentation

4.75.2.1 MASTER

```
#define MASTER 0
```

Definition at line 39 of file parallel.h.

4.75.2.2 PF_RESET

```
#define PF_RESET 0
```

Definition at line 41 of file parallel.h.

4.75.2.3 PF_TIME

```
#define PF_TIME 1
```

Definition at line 42 of file parallel.h.

4.75.2.4 PF_TERM_MSGTAG

```
#define PF_TERM_MSGTAG 10 /* master -> slave:  sending terms */
```

Definition at line 44 of file parallel.h.

4.75.2.5 PF_ENDSORT_MSGTAG

```
#define PF_ENDSORT_MSGTAG 11 /* master -> slave:  no more terms to be distributed, slave ->
master:  after EndSort() */
```

Definition at line 45 of file parallel.h.

4.75.2.6 PF_DOLLAR_MSGTAG

```
#define PF_DOLLAR_MSGTAG 12 /* slave -> master:  sending $-variables */
```

Definition at line 46 of file parallel.h.

4.75.2.7 PF_BUFFER_MSGTAG

```
#define PF_BUFFER_MSGTAG 20 /* slave -> master:  sending sorted terms, or in PF_SendFile()/PF_RecvFile()
*/
```

Definition at line 47 of file parallel.h.

4.75.2.8 PF_ENDBUFFER_MSGTAG

```
#define PF_ENDBUFFER_MSGTAG 21 /* same as PF_BUFFER_MSGTAG, but indicates the end of operation
*/
```

Definition at line 48 of file parallel.h.

4.75.2.9 PF_READY_MSGTAG

```
#define PF_READY_MSGTAG 30 /* slave -> master:  slave is idle and can accept terms */
```

Definition at line 49 of file parallel.h.

4.75.2.10 PF_DATA_MSGTAG

```
#define PF_DATA_MSGTAG 50 /* InParallel, DoCheckpoint() */
```

Definition at line 50 of file parallel.h.

4.75.2.11 PF_EMPTY_MSGTAG

```
#define PF_EMPTY_MSGTAG 52 /* InParallel, DoCheckpoint(), PF_SendFile(), PF_RecvFile() */
```

Definition at line 51 of file parallel.h.

4.75.2.12 PF_STDOUT_MSGTAG

```
#define PF_STDOUT_MSGTAG 60 /* slave -> master:  sending text to the stdout */
```

Definition at line 52 of file parallel.h.

4.75.2.13 PF_LOG_MSGTAG

```
#define PF_LOG_MSGTAG 61 /* slave -> master:  sending text to the log file */
```

Definition at line 53 of file parallel.h.

4.75.2.14 PF_OPT_MCTS_MSGTAG

```
#define PF_OPT_MCTS_MSGTAG 70 /* master <-> slave:  optimization */
```

Definition at line 54 of file parallel.h.

4.75.2.15 PF_OPT_HORNER_MSGTAG

```
#define PF_OPT_HORNER_MSGTAG 71 /* master <-> slave:  optimization */
```

Definition at line 55 of file parallel.h.

4.75.2.16 PF_OPT_COLLECT_MSGTAG

```
#define PF_OPT_COLLECT_MSGTAG 72 /* slave -> master:  optimization */
```

Definition at line 56 of file parallel.h.

4.75.2.17 PF_MISC_MSGTAG

```
#define PF_MISC_MSGTAG 100
```

Definition at line 57 of file parallel.h.

4.75.2.18 GNUC_PREREQ

```
#define GNUC_PREREQ(
              major,
              minor,
              patchlevel ) 0
```

Definition at line 67 of file parallel.h.

4.75.2.19 indices

```
#define indices ((INDICES)(AC.IndexList.lijst))
```

Definition at line 111 of file parallel.h.

4.75.2.20 PF_ANY_SOURCE

```
#define PF_ANY_SOURCE MPI_ANY_SOURCE
```

Definition at line 121 of file parallel.h.

4.75.2.21 PF_ANY_MSGTAG

```
#define PF_ANY_MSGTAG MPI_ANY_TAG
```

Definition at line 122 of file parallel.h.

4.75.2.22 PF_COMM

```
#define PF_COMM MPI_COMM_WORLD
```

Definition at line 123 of file parallel.h.

4.75.2.23 PF_BYTE

```
#define PF_BYTE MPI_BYTE
```

Definition at line 124 of file parallel.h.

4.75.2.24 PF_INT

```
#define PF_INT MPI_INT
```

Definition at line 125 of file parallel.h.

4.75.2.25 PF_LongMultiPack

```
#define PF_LongMultiPack(
            buffer,
            count,
            type ) PF_LongMultiPackImpl(buffer, count, sizeof_datatype(type), type)
```

Definition at line 233 of file parallel.h.

4.75.2.26 PF_LongMultiUnpack

```
#define PF_LongMultiUnpack(
            buffer,
            count,
            type ) PF_LongMultiUnpackImpl(buffer, count, sizeof_datatype(type), type)
```

Definition at line 234 of file parallel.h.

4.75.3 Function Documentation

4.75.3.1 PF_ISendSbuf()

```
int PF_ISendSbuf (
            int to,
            int tag )  [extern]
```

Nonblocking send operation. It sends everything from `buff` to `fill` of the active buffer. Depending on *tag* it also can do waiting for other sends to finish or set the active buffer to the next one.

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 261 of file mpi.c.

Referenced by FlushOut(), PF_Processor(), and PutOut().

4.75.3.2 PF_Bcast()

```
int PF_Bcast (
            void * buffer,
            int count )  [extern]
```

Broadcasts a message from the master to slaves.

Parameters

in,out	*buffer*	the starting address of buffer. The contents in this buffer on the master will be transferred to those on the slaves.
	count	the length of the buffer in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 440 of file mpi.c.

Referenced by PF_BroadcastBuffer(), and PF_BroadcastNumber().

4.75.3.3 PF_RawSend()

```
int PF_RawSend (
            int dest,
            void * buf,
            LONG l,
            int tag )  [extern]
```

Sends *l* bytes from *buf* to *dest*. Returns 0 on success, or -1.

Parameters

dest	the destination process number.
buf	the send buffer.
l	the size of data in the send buffer in bytes.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 463 of file mpi.c.

4.75.3.4 PF_RawRecv()

```
LONG PF_RawRecv (
            int * src,
            void * buf,
            LONG thesize,
            int * tag )  [extern]
```

Receives not more than *thesize* bytes from *src*, returns the actual number of received bytes, or -1 on failure.

Parameters

in,out	*src*	the source process number. In output, that of the actual received message.
out	*buf*	the receive buffer.
	thesize	the size of the receive buffer in bytes.
out	*tag*	the message tag of the actual received message.

Returns

the actual sizeof received data in bytes, or -1 on failure.

Definition at line 484 of file mpi.c.

4.75.3.5 PF_PreparePack()

```
int PF_PreparePack (
            void  )  [extern]
```

Prepares for the next pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 624 of file mpi.c.

Referenced by PF_BroadcastPreDollar(), PF_BroadcastString(), and PF_Init().

4.75.3.6 PF_Pack()

```
int PF_Pack (
            const void * buffer,
            size_t count,
            MPI_Datatype type )  [extern]
```

Adds data into the pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 642 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by PF_BroadcastPreDollar(), and PF_Init().

4.75.3.7 PF_Unpack()

```
int PF_Unpack (
            void * buffer,
            size_t count,
            MPI_Datatype type )  [extern]
```

Retrieves the next data in the pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 671 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by PF_BroadcastPreDollar(), and PF_Init().

4.75.3.8 PF_PackString()

```
int PF_PackString (
            const UBYTE * str )  [extern]
```

Packs a string *str* into the packed buffer PF_packbuf, including the trailing zero.

The first element (PF_INT) is the length of the packed portion of the string. If the string does not fit to the buffer PF↩_packbuf, the function packs only the initial portion. It returns the number of packed bytes, so if (str[length-1]=='\0') then the whole string fits to the buffer, if not, then the rest (str+length) bust be packed and send again. On error, the function returns the negative error code.

One exception: the string "\0!\0" is used as an image of the NULL, so all 3 characters will be packed.

Parameters

str	a string to be packed.

Returns

the number of packed bytes, or a negative value on failure.

Definition at line 706 of file mpi.c.

Referenced by PF_BroadcastString().

4.75.3.9 PF_UnpackString()

```
int PF_UnpackString (
            UBYTE * str )  [extern]
```

Unpacks a string to *str* from the packed buffer PF_packbuf, including the trailing zero.

It returns the number of unpacked bytes, so if (str[length-1]=='\0') then the whole string was unpacked, if not, then the rest must be appended to (str+length). On error, the function returns the negative error code.

Parameters

out	*str*	the buffer to store the unpacked string

Returns

the number of unpacked bytes, or a negative value on failure.

Definition at line 774 of file mpi.c.

Referenced by PF_BroadcastString().

4.75.3.10 PF_Send()

```
int PF_Send (
            int to,
            int tag )  [extern]
```

Sends the contents in the pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
  PF_PreparePack();
  // Packing operations here...
  PF_Send(DEST, TAG);
}
else if ( PF.me == DEST ) {
  PF_Receive(SRC, TAG, &actual_src, &actual_tag);
  // Unpacking operations here...
}
```

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 822 of file mpi.c.

References MPI_ERRCODE_CHECK.

4.75.3.11 PF_Receive()

```
int PF_Receive (
            int src,
            int tag,
            int * psrc,
            int * ptag )  [extern]
```

Receives data into the pack buffer from the process specified by *src*. This function allows &src == psrc or &tag == ptag. Either *psrc* or *ptag* can be NULL.

See the example of PF_Send().

Parameters

	src	the source process number (can be PF_ANY_SOURCE).
	tag	the source message tag (can be PF_ANY_TAG).
out	*psrc*	the actual source process number of received message.
out	*ptag*	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 848 of file mpi.c.

References MPI_ERRCODE_CHECK.

4.75.3.12 PF_Broadcast()

```
int PF_Broadcast (
            void  )  [extern]
```

Broadcasts the contents in the pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
  PF_PreparePack();
  // Packing operations here...
}
PF_Broadcast();
if ( PF.me != MASTER ) {
  // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 883 of file mpi.c.

References MPI_ERRCODE_CHECK.

Referenced by PF_BroadcastPreDollar(), PF_BroadcastString(), and PF_Init().

4.75.3.13 PF_PrepareLongSinglePack()

```
int PF_PrepareLongSinglePack (
            void  )  [extern]
```

Prepares for the next long-single-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1451 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.75.3.14 PF_LongSinglePack()

```
int PF_LongSinglePack (
            const void * buffer,
            size_t count,
            MPI_Datatype type )  [extern]
```

Adds data into the "long single" pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1469 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.75.3.15 PF_LongSingleUnpack()

```
int PF_LongSingleUnpack (
            void * buffer,
            size_t count,
            MPI_Datatype type )  [extern]
```

Retrieves the next data in the "long single" pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1503 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.75.3.16 PF_LongSingleSend()

```
int PF_LongSingleSend (
            int to,
            int tag )  [extern]
```

Sends the contents in the "long single" pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
  PF_PrepareLongSinglePack();
  // Packing operations here...
  PF_LongSingleSend(DEST, TAG);
}
else if ( PF.me == DEST ) {
  PF_LongSingleReceive(SRC, TAG, &actual_src, &actual_tag);
  // Unpacking operations here...
}
```

Parameters

to	the destination process number.
tag	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1540 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.75.3.17 PF_LongSingleReceive()

```
int PF_LongSingleReceive (
            int src,
            int tag,
            int * psrc,
            int * ptag )  [extern]
```

Receives data into the "long single" pack buffer from the process specified by *src*. This function allows &src == psrc or &tag == ptag. Either *psrc* or *ptag* can be NULL.

See the example of PF_LongSingleSend().

Parameters

	src	the source process number (can be PF_ANY_SOURCE).
	tag	the source message tag (can be PF_ANY_TAG).
out	*psrc*	the actual source process number of received message.
out	*ptag*	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1583 of file mpi.c.

Referenced by PF_CollectModifiedDollars(), and PF_Processor().

4.75.3.18 PF_PrepareLongMultiPack()

```
int PF_PrepareLongMultiPack (
            void  )  [extern]
```

Prepares for the next long-multi-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1643 of file mpi.c.

Referenced by PF_BroadcastCBuf(), PF_BroadcastExpFlags(), PF_BroadcastModifiedDollars(), and PF_BroadcastRedefinedPreVars

4.75.3.19 PF_LongMultiPackImpl()

```
int PF_LongMultiPackImpl (
            const void * buffer,
            size_t count,
            size_t eSize,
            MPI_Datatype type )  [extern]
```

Adds data into the "long multi" pack buffer.

Parameters

buffer	the pointer to the buffer storing the data to be packed.
count	the number of elements in the buffer.
eSize	the byte size of each element of data.
type	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1662 of file mpi.c.

References PF_LongMultiPackImpl().

Referenced by PF_LongMultiPackImpl().

Here is the call graph for this function:

4.75.3.20 PF_LongMultiUnpackImpl()

```
int PF_LongMultiUnpackImpl (
            void * buffer,
            size_t count,
            size_t eSize,
            MPI_Datatype type )  [extern]
```

Retrieves the next data in the "long multi" pack buffer.

Parameters

out	*buffer*	the pointer to the buffer to store the unpacked data.
	count	the number of elements of data to be received.
	eSize	the byte size of each element of data.
	type	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1721 of file mpi.c.

References PF_LongMultiUnpackImpl().

Referenced by PF_LongMultiUnpackImpl().

Here is the call graph for this function:

4.75.3.21 PF_LongMultiBroadcast()

```
int PF_LongMultiBroadcast (
            void  )  [extern]
```

Broadcasts the contents in the "long multi" pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
  PF_PrepareLongMultiPack();
  // Packing operations here...
}
PF_LongMultiBroadcast();
if ( PF.me != MASTER ) {
  // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 1807 of file mpi.c.

Referenced by PF_BroadcastCBuf(), PF_BroadcastExpFlags(), PF_BroadcastModifiedDollars(), and PF_BroadcastRedefinedPreVars

4.75.3.22 PF_EndSort()

```
int PF_EndSort (
            void  )  [extern]
```

Finishes a master sorting with collecting terms from slaves. Called by EndSort().

If this is not the masterprocess, just initialize the sendbuffers and return 0, else PF_EndSort() sends the rest of the terms in the sendbuffer to the next slave and a dummy message to all slaves with tag PF_ENDSORT_MSGTAG. Then it receives the sorted terms, sorts them using a recursive 'tree of losers' (PF_GetLoser()) and writes them to the outputfile.

Returns

1 if the sorting on the master was done. 0 if EndSort() still must perform a regular sorting becuase it is not at the ground level or not on the master or in the sequential mode or in the InParallel mode. -1 if an error occured.

Remarks

The slaves will send the sorted terms back to the master in the regular sorting (after the initialization of the send buffer in PF_EndSort()). See PutOut() and FlushOut().

This function has been changed such that when it returns 1, AM.S0->TermsLeft is set correctly. But AM.S0->GenTerms is not set: it will be set after collecting the statistics from the slaves at the end of PF_Processor(). (TU 30 Jun 2011)

Definition at line 864 of file parallel.c.

References FlushOut(), and PutOut().

Referenced by EndSort().

Here is the call graph for this function:

4.75.3.23 PF_Deferred()

```
WORD PF_Deferred (
            WORD * term,
            WORD level )  [extern]
```

Replaces Deferred() on the slaves.

Parameters

term	the term that must be multiplied by the contents of the current bracket.
level	the compiler level.

Returns

0 if OK, nonzero on error.

Definition at line 1208 of file parallel.c.

References Generator(), and InsertTerm().

Referenced by Generator().

Here is the call graph for this function:

4.75.3.24 PF_Processor()

```
int PF_Processor (
            EXPRESSIONS e,
            WORD i,
            WORD LastExpression )  [extern]
```

Replaces parts of Processor() on the masters and slaves. On the master PF_Processor() is responsible for proper distribution of terms from the input file to the slaves. On the slaves it calls Generator() for all the terms that this process gets, but PF_GetTerm() gets terms from the master (not directly from infile).

Parameters

e	The pointer to the current expression.
i	The index for the current expression.
LastExpression	The flag indicating whether it is the last expression.

Returns

0 if OK, nonzero on error.

Definition at line 1540 of file parallel.c.

References EndSort(), Generator(), LowerSortLevel(), NewSort(), PACK_LONG, PF_ISendSbuf(), PF_LongSinglePack(), PF_LongSingleReceive(), PF_LongSingleSend(), PF_LongSingleUnpack(), PF_PrepareLongSinglePack(), PutOut(), StoreTerm(), SWAP, TimeCPU(), and WriteStats().

Referenced by Processor().

Here is the call graph for this function:

4.75.3.25 PF_Init()

```
int PF_Init (
            int * argc,
            char *** argv )  [extern]
```

All the library independent stuff. PF_LibInit() should do all library dependent initializations.

Parameters

argc	pointer to the number of arguments.
argv	pointer to the arguments.

Returns

0 if OK, nonzero on error.

Definition at line 1953 of file parallel.c.

References PF_Broadcast(), PF_LibInit(), PF_Pack(), PF_PreparePack(), PF_RealTime(), and PF_Unpack().

Here is the call graph for this function:

4.75.3.26 PF_Terminate()

```
int PF_Terminate (
            int errorcode )  [extern]
```

Performs the finalization of ParFORM. To be called by Terminate().

Parameters

error	an error code.

Returns

0 if OK, nonzero on error.

Definition at line 2047 of file parallel.c.

References PF_LibTerminate().

Here is the call graph for this function:

4.75.3.27 PF_GetSlaveTimes()

```
LONG PF_GetSlaveTimes (
            void  )  [extern]
```

Returns the total CPU time of all slaves together. This function must be called on the master and all slaves.

Returns

on the master, the sum of CPU times on all slaves.

Definition at line 2063 of file parallel.c.

References TimeCPU().

Here is the call graph for this function:

4.75.3.28 PF_BroadcastNumber()

```
LONG PF_BroadcastNumber (
            LONG x )  [extern]
```

Broadcasts a LONG value from the master to the all slaves.

Parameters

x	the number to be broadcast (set on the master).

Returns

the synchronised result.

Definition at line 2083 of file parallel.c.

References PF_Bcast().

Referenced by DoCheckpoint(), PF_BroadcastBuffer(), and StartVariables().

Here is the call graph for this function:

4.75.3.29 PF_BroadcastBuffer()

```
void PF_BroadcastBuffer (
            WORD ** buffer,
            LONG * length )  [extern]
```

Broadcasts a buffer from the master to all the slaves.

Parameters

in,out	*buffer*	on the master, the buffer to be broadcast. On the slaves, the buffer will be allocated if the length is greater than 0. The caller must free it.
in,out	*length*	on the master, the length of the buffer to be broadcast. On the slaves, it receives the length of transfered buffer. The actual transfer occurs only if the length is greater than 0.

Definition at line 2110 of file parallel.c.

References PF_Bcast(), and PF_BroadcastNumber().

Here is the call graph for this function:

4.75.3.30 PF_BroadcastString()

```
int PF_BroadcastString (
            UBYTE * str )  [extern]
```

Broadcasts a string from the master to all slaves.

Parameters

in,out	*str*	The pointer to a null-terminated string.

Returns

0 if OK, nonzero on error.

Definition at line 2152 of file parallel.c.

References PF_Broadcast(), PF_PackString(), PF_PreparePack(), and PF_UnpackString().

Here is the call graph for this function:

4.75.3.31 PF_BroadcastPreDollar()

```
int PF_BroadcastPreDollar (
            WORD ** dbuffer,
            LONG * newsize,
            int * numterms )  [extern]
```

Broadcasts dollar variables set as a preprocessor variables. Only the master is able to make an assignment like #$a=g; where g is an expression: only the master has an access to the expression. So, the master broadcasts the result to slaves.

The result is in ∗dbuffer of the size is ∗newsize (in number of WORDs), +1 for trailing zero. For slave newsize and numterms are output parameters.

Parameters

in,out	*dbuffer*	the buffer for a dollar variable.
in,out	*newsize*	the size of the dollar variable in WORDs.
in,out	*numterms*	the number of terms in the dollar variable.

Returns

0 if OK, nonzero on error.

Definition at line 2207 of file parallel.c.

References PF_Broadcast(), PF_Pack(), PF_PreparePack(), and PF_Unpack().

Here is the call graph for this function:

4.75.3.32 PF_CollectModifiedDollars()

```
int PF_CollectModifiedDollars (
            void  )  [extern]
```

Combines modified dollar variables on the all slaves, and store them into those on the master.

The potentially modified dollar variables are given in PotModdollars, and the number of them is given by NumPot↩Moddollars.

The current module could be executed in parallel only if all potentially modified variables are listed in ModOptdollars, otherwise the module was switched to the sequential mode.

Returns

0 if OK, nonzero on error.

Definition at line 2495 of file parallel.c.

References EndSort(), Generator(), LowerSortLevel(), NewSort(), PF_LongSinglePack(), PF_LongSingleReceive(), PF_LongSingleSend(), PF_LongSingleUnpack(), PF_PrepareLongSinglePack(), VectorInit, VectorPtr, VectorReserve, and VectorSize.

Here is the call graph for this function:

4.75.3.33 PF_BroadcastModifiedDollars()

```
int PF_BroadcastModifiedDollars (
            void  )  [extern]
```

Broadcasts modified dollar variables on the master to the all slaves.

The potentially modified dollar variables are given in PotModdollars, and the number of them is given by NumPot↩Moddollars.

The current module could be executed in parallel only if all potentially modified variables are listed in ModOptdollars, otherwise the module was switched to the sequential mode. In either cases, we need to broadcast them.

Returns

0 if OK, nonzero on error.

Definition at line 2774 of file parallel.c.

References PF_LongMultiBroadcast(), and PF_PrepareLongMultiPack().

Here is the call graph for this function:

4.75.3.34 PF_BroadcastRedefinedPreVars()

```
int PF_BroadcastRedefinedPreVars (
            void  )  [extern]
```

Broadcasts preprocessor variables, which were changed by the Redefine statements in the current module, from the master to the all slaves.

The potentially redefined preprocessor variables are given in AC.pfirstnum, and the number of them is given by AC.numpfirstnum. For an actually redefined variable, the corresponding value in AC.inputnumbers is non-negative.

Returns

0 if OK, nonzero on error.

Definition at line 2991 of file parallel.c.

References PF_LongMultiBroadcast(), PF_PrepareLongMultiPack(), PutPreVar(), VectorPtr, and VectorReserve.

Here is the call graph for this function:

4.75.3.35 PF_BroadcastCBuf()

```
int PF_BroadcastCBuf (
            int bufnum )  [extern]
```

Broadcasts a compiler buffer specified by *bufnum* from the master to the all slaves.

Parameters

bufnum	The index of the compiler buffer to be broadcast.

Returns

0 if OK, nonzero on error.

Definition at line 3133 of file parallel.c.

References CbUf::boomlijst, CbUf::Buffer, CbUf::BufferSize, CbUf::CanCommu, CbUf::dimension, finishcbuf(), CbUf::lhs, CbUf::numdum, CbUf::NumTerms, PF_LongMultiBroadcast(), PF_PrepareLongMultiPack(), CbUf::Pointer, CbUf::rhs, and CbUf::Top.

Here is the call graph for this function:

4.75.3.36 PF_BroadcastExpFlags()

```
int PF_BroadcastExpFlags (
            void ) [extern]
```

Broadcasts AR.expflags and several properties of each expression, e.g., e->vflags, from the master to all slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3244 of file parallel.c.

References PF_LongMultiBroadcast(), and PF_PrepareLongMultiPack().

Here is the call graph for this function:

4.75.3.37 PF_StoreInsideInfo()

```
int PF_StoreInsideInfo (
            void  )  [extern]
```

Definition at line 3081 of file parallel.c.

4.75.3.38 PF_RestoreInsideInfo()

```
int PF_RestoreInsideInfo (
            void  )  [extern]
```

Definition at line 3107 of file parallel.c.

4.75.3.39 PF_BroadcastExpr()

```
int PF_BroadcastExpr (
            EXPRESSIONS e,
            FILEHANDLE * file )  [extern]
```

Broadcasts an expression from the master to the all slaves.

Parameters

e	The expression to be broadcast.
file	The file in which the expression is sitting.

Returns

0 if OK, nonzero on error.

Definition at line 3536 of file parallel.c.

Referenced by PF_BroadcastRHS().

4.75.3.40 PF_BroadcastRHS()

```
int PF_BroadcastRHS (
            void  )  [extern]
```

Broadcasts expressions appearing in the right-hand side from the master to the all slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3564 of file parallel.c.

References PF_BroadcastExpr().

Referenced by Processor().

Here is the call graph for this function:

4.75.3.41 PF_InParallelProcessor()

```
int PF_InParallelProcessor (
            void  )  [extern]
```

Processes expressions in the InParallel mode, i.e., dividing expressions marked by partodo over the slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3611 of file parallel.c.

Referenced by Processor().

4.75.3.42 PF_SendFile()

```
int PF_SendFile (
            int to,
            FILE * fd )  [extern]
```

Sends a file to the process specified by *to*.

Parameters

to	the destination process number.
fd	the file to be sent.

Returns

the size of sent data in bytes, or -1 on error.

Definition at line 4207 of file parallel.c.

References PF_RawSend().

Referenced by DoCheckpoint().

Here is the call graph for this function:

4.75.3.43 PF_RecvFile()

```
int PF_RecvFile (
            int from,
            FILE * fd )  [extern]
```

Receives a file from the process specified by *from*.

Parameters

from	the source process number.
fd	the file to save the received data.

Returns

the size of received data in bytes, or -1 on error.

Definition at line 4245 of file parallel.c.

References PF_RawRecv().

Referenced by DoCheckpoint().

Here is the call graph for this function:

4.75.3.44 PF_MLock()

```
void PF_MLock (
            void  )  [extern]
```

A function called by MLOCK(ErrorMessageLock) for slaves.

Definition at line 4326 of file parallel.c.

References VectorClear.

4.75.3.45 PF_MUnlock()

```
void PF_MUnlock (
            void  )  [extern]
```

A function called by MUNLOCK(ErrorMessageLock) for slaves.

Definition at line 4342 of file parallel.c.

References PF_RawSend(), VectorEmpty, VectorPtr, and VectorSize.

Here is the call graph for this function:

4.75.3.46 PF_WriteFileToFile()

```
LONG PF_WriteFileToFile (
            int handle,
            UBYTE * buffer,
            LONG size )  [extern]
```

Replaces WriteFileToFile() on the master and slaves.

It copies the given buffer into internal buffers if called between MLOCK(ErrorMessageLock) and MUNLOCK(Error↩MessageLock) for slaves and handle is StdOut or LogHandle, otherwise calls WriteFileToFile().

Parameters

handle	a file handle that specifies the output.
buffer	a pointer to the source buffer containing the data to be written.
size	the size of data to be written in bytes.

Returns

the actual size of data written to the output in bytes.

Definition at line 4371 of file parallel.c.

References VectorClear, VectorPtr, VectorPushBacks, and VectorSize.

4.75.3.47 PF_FlushStdOutBuffer()

```
void PF_FlushStdOutBuffer (
            void  )  [extern]
```

Explicitly Flushes the buffer for the standard output on the master, which is used if PF_ENABLE_STDOUT_↩ BUFFERING is defined.

Definition at line 4465 of file parallel.c.

References VectorClear, VectorPtr, and VectorSize.

4.75.4 Variable Documentation

4.75.4.1 PF

```
PARALLELVARS PF  [extern]
```

Definition at line 90 of file parallel.c.

4.75.4.2 PF_maxDollarChunkSize

```
LONG PF_maxDollarChunkSize  [extern]
```

Definition at line 69 of file mpi.c.

4.76 parallel.h

Go to the documentation of this file.

```
#ifndef __PARALLEL__
#define __PARALLEL__

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
```

```
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

/*
    #[ macros & definitions :
*/
#define MASTER 0

#define PF_RESET 0
#define PF_TIME  1

#define PF_TERM_MSGTAG          10  /* master -> slave: sending terms */
#define PF_ENDSORT_MSGTAG       11  /* master -> slave: no more terms to be distributed, slave -> master: after EndSort() */
#define PF_DOLLAR_MSGTAG        12  /* slave -> master: sending $-variables */
#define PF_BUFFER_MSGTAG        20  /* slave -> master: sending sorted terms, or in PF_SendFile()/PF_RecvFile() */
#define PF_ENDBUFFER_MSGTAG     21  /* same as PF_BUFFER_MSGTAG, but indicates the end of operation */
#define PF_READY_MSGTAG         30  /* slave -> master: slave is idle and can accept terms */
#define PF_DATA_MSGTAG          50  /* InParallel, DoCheckpoint() */
#define PF_EMPTY_MSGTAG         52  /* InParallel, DoCheckpoint(), PF_SendFile(), PF_RecvFile() */
#define PF_STDOUT_MSGTAG        60  /* slave -> master: sending text to the stdout */
#define PF_LOG_MSGTAG           61  /* slave -> master: sending text to the log file */
#define PF_OPT_MCTS_MSGTAG      70  /* master <-> slave: optimization */
#define PF_OPT_HORNER_MSGTAG    71  /* master <-> slave: optimization */
#define PF_OPT_COLLECT_MSGTAG   72  /* slave -> master: optimization */
#define PF_MISC_MSGTAG         100

/*
 * A macro for checking the version of gcc.
 */
#if defined(__GNUC__) && defined(__GNUC_MINOR__) && defined(__GNUC_PATCHLEVEL__)
#  define GNUC_PREREQ(major, minor, patchlevel) \
     ((__GNUC__ « 16) + (__GNUC_MINOR__ « 8) + __GNUC_PATCHLEVEL__ >= \
     ((major) « 16) + ((minor) « 8) + (patchlevel))
#else
#  define GNUC_PREREQ(major, minor, patchlevel) 0
#endif

/*
 * The macro "indices" defined in variable.h collides with some function
 * argument names in the MPI-3.0 standard.
 */
#undef indices

/* Avoid messy padding warnings which may appear in mpi.h. */
#if GNUC_PREREQ(4, 6, 0)
#  pragma GCC diagnostic push
#  pragma GCC diagnostic ignored "-Wpadded"
#  pragma GCC diagnostic ignored "-Wunused-parameter"
#endif
#if defined(__clang__) && defined(__has_warning)
#  pragma clang diagnostic push
#  if __has_warning("-Wpadded")
#    pragma clang diagnostic ignored "-Wpadded"
#  endif
#  if __has_warning("-Wunused-parameter")
#    pragma clang diagnostic ignored "-Wunused-parameter"
#  endif
#endif

#  ifdef __cplusplus
     /*
      * form3.h (which includes parallel.h) is included from newpoly.h as
      *   extern "C" {
      *   #include "form3.h"
      *   }
      * On the other hand, C++ interfaces to MPI are defined in mpi.h if it is
      * included from C++ sources. We first leave from the C-linkage, include
      * mpi.h, and then go back to the C-linkage.
      * (TU 7 Jun 2011)
      */
}
#    include <mpi.h>
extern "C" {
#  else
#    include <mpi.h>
#  endif

/* Now redefine "indices" in the same way as in variable.h. */
```

```
#define indices ((INDICES)(AC.IndexList.lijst))

/* Restore the warning settings. */
#if GNUC_PREREQ(4, 6, 0)
#  pragma GCC diagnostic pop
#endif
#if defined(__clang__) && defined(__has_warning)
#  pragma clang diagnostic pop
#endif

#  define PF_ANY_SOURCE MPI_ANY_SOURCE
#  define PF_ANY_MSGTAG MPI_ANY_TAG
#  define PF_COMM MPI_COMM_WORLD
#  define PF_BYTE MPI_BYTE
#  define PF_INT  MPI_INT
#  if defined(ILP32)
#    define PF_WORD MPI_SHORT
#    define PF_LONG MPI_LONG
#  elif defined(LLP64)
#    define PF_WORD MPI_INT
#    define PF_LONG MPI_LONG_LONG_INT
#  elif defined(LP64)
#    define PF_WORD MPI_INT
#    define PF_LONG MPI_LONG
#  endif

/*
    #] macros & definitions :
    #[ s/r-bufs :
*/

typedef struct {
    WORD **buff;
    WORD **fill;
    WORD **full;
    WORD **stop;
    MPI_Status *status;
    MPI_Status *retstat;
    MPI_Request *request;
    MPI_Datatype *type;   /* this is needed in PF_Wait for Get_count */
    int *index;           /* dummies for returnvalues */
    int *tag;             /* for the version with blocking send/receives */
    int *from;
    int numbufs;          /* number of cyclic buffers */
    int active;           /* flag telling which buffer is active */
    PADPOINTER(0,2,0,0);
} PF_BUFFER;

/*
    #] s/r-bufs :
    #[ global variables used by the PF_functions : need to be known everywhere
*/

typedef struct ParallelVars {
    FILEHANDLE  slavebuf;       /* (slave) allocated if there are RHS expressions */
    /* special buffers for nonblocking, unbuffered send/receives */
    PF_BUFFER  *sbuf;           /* set of cyclic send buffers for master _and_ slave */
    PF_BUFFER **rbufs;          /* array of sets of cyclic receive buffers for master */
    int         me;             /* Internal number of task: master is 0 */
    int         numtasks;       /* total number of tasks */
    int         parallel;       /* flags telling the master and slaves to do the sorting parallel */
                                /* [05nov2003 mt] This flag must be set to 0 in iniModule! */
    int         rhsInParallel;  /* flag for parallel executing even if there are RHS expressions */
    int         mkSlaveInfile;  /* flag tells that slavebuf is used on the slaves */
    int         exprbufsize;    /* buffer size in WORDs to be used for transferring expressions */
    int         exprtodo;       /* >= 0: the expression to do in InParallel, -1: otherwise */
    int         log;            /* flag for logging mode */
    WORD        numsbufs;       /* number of cyclic send buffers (PF.sbuf->numbufs) */
    WORD        numrbufs;       /* number of cyclic receive buffers (PF.rbufs[i]->numbufs,
    i=1,...numtasks-1) */
    PADPOSITION(2,0,8,2,0);
} PARALLELVARS;

extern PARALLELVARS PF;
/*[04oct2005 mt]:*/
/*for broadcasting dollarvars, see parallel.c:PF_BroadcastPreDollar():*/
extern LONG PF_maxDollarChunkSize;
/*:[04oct2005 mt]*/

/*
    #] global variables used by the PF_functions :
    #[ Function prototypes :
*/

/* mpi.c */
extern int    PF_ISendSbuf(int,int);
extern int    PF_Bcast(void *buffer, int count);
```

```
extern int    PF_RawSend(int,void *,LONG,int);
extern LONG   PF_RawRecv(int *,void *,LONG,int *);

extern int    PF_PreparePack(void);
extern int    PF_Pack(const void *buffer, size_t count, MPI_Datatype type);
extern int    PF_Unpack(void *buffer, size_t count, MPI_Datatype type);
extern int    PF_PackString(const UBYTE *str);
extern int    PF_UnpackString(UBYTE *str);
extern int    PF_Send(int to, int tag);
extern int    PF_Receive(int src, int tag, int *psrc, int *ptag);
extern int    PF_Broadcast(void);

extern int    PF_PrepareLongSinglePack(void);
extern int    PF_LongSinglePack(const void *buffer, size_t count, MPI_Datatype type);
extern int    PF_LongSingleUnpack(void *buffer, size_t count, MPI_Datatype type);
extern int    PF_LongSingleSend(int to, int tag);
extern int    PF_LongSingleReceive(int src, int tag, int *psrc, int *ptag);

extern int    PF_PrepareLongMultiPack(void);
extern int    PF_LongMultiPackImpl(const void *buffer, size_t count, size_t eSize, MPI_Datatype type);
extern int    PF_LongMultiUnpackImpl(void *buffer, size_t count, size_t eSize, MPI_Datatype type);
extern int    PF_LongMultiBroadcast(void);

static inline size_t sizeof_datatype(MPI_Datatype type)
{
    if ( type == PF_BYTE ) return sizeof(char);
    if ( type == PF_INT  ) return sizeof(int);
    if ( type == PF_WORD ) return sizeof(WORD);
    if ( type == PF_LONG ) return sizeof(LONG);
    return(0);
}

#define PF_LongMultiPack(buffer, count, type) PF_LongMultiPackImpl(buffer, count, sizeof_datatype(type), type)
#define PF_LongMultiUnpack(buffer, count, type) PF_LongMultiUnpackImpl(buffer, count, sizeof_datatype(type), type)

/* parallel.c */
extern int    PF_EndSort(void);
extern WORD   PF_Deferred(WORD *,WORD);
extern int    PF_Processor(EXPRESSIONS,WORD,WORD);
extern int    PF_Init(int*,char ***);
extern int    PF_Terminate(int);
extern LONG   PF_GetSlaveTimes(void);
extern LONG   PF_BroadcastNumber(LONG);
extern void   PF_BroadcastBuffer(WORD **buffer, LONG *length);
extern int    PF_BroadcastString(UBYTE *);
extern int    PF_BroadcastPreDollar(WORD **, LONG *,int *);
extern int    PF_CollectModifiedDollars(void);
extern int    PF_BroadcastModifiedDollars(void);
extern int    PF_BroadcastRedefinedPreVars(void);
extern int    PF_BroadcastCBuf(int bufnum);
extern int    PF_BroadcastExpFlags(void);
extern int    PF_StoreInsideInfo(void);
extern int    PF_RestoreInsideInfo(void);
extern int    PF_BroadcastExpr(EXPRESSIONS e, FILEHANDLE *file);
extern int    PF_BroadcastRHS(void);
extern int    PF_InParallelProcessor(void);
extern int    PF_SendFile(int to, FILE *fd);
extern int    PF_RecvFile(int from, FILE *fd);
extern void   PF_MLock(void);
extern void   PF_MUnlock(void);
extern LONG   PF_WriteFileToFile(int,UBYTE *,LONG);
extern void   PF_FlushStdOutBuffer(void);

/*
    #] Function prototypes :
*/

#endif
```

4.77 pattern.c File Reference

`#include "form3.h"`
Include dependency graph for pattern.c:

Macros

- #define PutInBuffers(pow)

Functions

- WORD TestMatch (PHEAD WORD ∗term, WORD ∗level)
- VOID Substitute (PHEAD WORD ∗term, WORD ∗pattern, WORD power)
- WORD FindAll (PHEAD WORD ∗term, WORD ∗pattern, WORD level, WORD ∗par)
- int TestSelect (WORD ∗term, WORD ∗setp)
- VOID SubsInAll (PHEAD0)
- VOID TransferBuffer (int from, int to, int spectator)
- int TakeIDfunction (PHEAD WORD ∗term)

4.77.1 Detailed Description

Top level pattern matching routines. More pattern matching is found in findpat.c, function.c, symmetr.c and smart.c. The last three files contain the matching inside functions. The file pattern.c contains also the very important routine Substitute. All regular pattern matching is just the finding of the pattern and indicating what are the wildcards etc. The routine Substitute does the actual removal of the pattern and replaces it by a subterm of the type SUBEXPRESSION.

Definition in file pattern.c.

4.77.2 Macro Definition Documentation

4.77.2.1 PutInBuffers

```
#define PutInBuffers(
              pow )
```

Value:

```
AddRHS(AT.ebufnum,1); \
*out++ = SUBEXPRESSION; \
*out++ = SUBEXPSIZE; \
*out++ = C->numrhs; \
*out++ = pow; \
*out++ = AT.ebufnum; \
FILLSUB(out) \
r = AT.pWorkSpace[rhs+i]; \
if ( *r > 0 ) { \
    oldinr = r[*r]; r[*r] = 0; \
    AddNtoC(AT.ebufnum,(*r+1-ARGHEAD),(r+ARGHEAD),14); \
    r[*r] = oldinr; \
} \
else { \
    ToGeneral(r,buffer,1); \
    buffer[buffer[0]] = 0; \
    AddNtoC(AT.ebufnum,buffer[0]+1,buffer,15); \
}
```

Definition at line 2193 of file pattern.c.

4.77.3 Function Documentation

4.77.3.1 TestMatch()

```
WORD TestMatch (
            PHEAD WORD * term,
            WORD * level )
```

This routine governs the pattern matching. If it decides that a substitution should be made, this can be either the insertion of a right hand side (C->rhs) or the automatic generation of terms as a result of an operation (like trace). The object to be replaced is removed from term and a subexpression pointer is inserted. If the substitution is made more than once there can be more subexpression pointers. Its number is positive as it corresponds to the level at which the C->rhs can be found in the compiler output. The subexpression pointer contains the wildcard substitution information. The power is found in ∗AT.TMout. For operations the subexpression pointer is negative and corresponds to an address in the array AT.TMout. In this array are then the instructions for the routine to be called and its number in the array 'Operations' The format is here: length,functionnumber,length-2 parameters

There is a certain complexity wrt repeat levels. Another complication is the poking of the wildcard values in the subexpression prototype in the compiler buffer. This was how things were done in the past with sequential FORM, but with the advent of TFORM this cannot be maintained. Now, for TFORM we make a copy of it. 7-may-2008 (JV): We cannot yet guarantee that this has been done 100% correctly. There are errors that occur in TFORM only and that may indicate problems.

Definition at line 97 of file pattern.c.

References AddRHS(), EndSort(), Generator(), CbUf::lhs, LowerSortLevel(), and NewSort().

Referenced by Generator().

Here is the call graph for this function:

4.77.3.2 Substitute()

```
VOID Substitute (
            PHEAD WORD * term,
            WORD * pattern,
            WORD power )
```

Definition at line 640 of file pattern.c.

4.77.3.3 FindAll()

```
WORD FindAll (
            PHEAD WORD * term,
            WORD * pattern,
            WORD level,
            WORD * par )
```

Definition at line 1197 of file pattern.c.

4.77.3.4 TestSelect()

```
int TestSelect (
            WORD * term,
            WORD * setp )
```

Definition at line 1748 of file pattern.c.

4.77.3.5 SubsInAll()

```
VOID SubsInAll (
            PHEAD0  )
```

Definition at line 1982 of file pattern.c.

4.77.3.6 TransferBuffer()

```
VOID TransferBuffer (
            int from,
            int to,
            int spectator )
```

Definition at line 2143 of file pattern.c.

4.77.3.7 TakeIDfunction()

```
int TakeIDfunction (
            PHEAD WORD * term )
```

Definition at line 2213 of file pattern.c.

4.78 pattern.c

Go to the documentation of this file.

```
00001
00012 /* #[ License : */
00013 /*
00014  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00015  *   When using this file you are requested to refer to the publication
00016  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00017  *   This is considered a matter of courtesy as the development was paid
00018  *   for by FOM the Dutch physics granting agency and we would like to
00019  *   be able to track its scientific use to convince FOM of its value
00020  *   for the community.
00021  *
00022  *   This file is part of FORM.
00023  *
00024  *   FORM is free software: you can redistribute it and/or modify it under the
00025  *   terms of the GNU General Public License as published by the Free Software
00026  *   Foundation, either version 3 of the License, or (at your option) any later
00027  *   version.
00028  *
00029  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00030  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00031  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00032  *   details.
00033  *
00034  *   You should have received a copy of the GNU General Public License along
00035  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00036  */
00037 /* #] License : */
00038 /*
00039 !!! Notice the change in OnePV in FindAll (7-may-2008 JV).
00040
00041     #[ Includes : pattern.c
00042 */
00043
00044 #include "form3.h"
00045
00046 /*
00047     #] Includes :
00048     #[ Patterns :
00049         #[ Rules :
```

```c

        There are several rules governing the allowable replacements.
        1:  Multi with anything but symbols or dotproducts reverts
            to many.
        2:  Each symbol can have only one (wildcard) power, so
            x^2*x^n? is illegal.
        3:  when a single vector is used it replaces all occurences
            of the vector. Therefore q*q(mu) or q*q(mu) cannot occur.
            Also q*q cannot be done.
        4:  Loose vector elements are replaced with p(mu), dotproducts
            with p?.q.
        5:  p?.q? is allowed.
        6:  x^n? can revert to n = 0 if there is no power of x.
        7:  x?^n? must match some x. There could be an ambiguity otherwise.

        #] Rules :
        #[ TestMatch :          WORD TestMatch(term,level)
*/

WORD TestMatch(PHEAD WORD *term, WORD *level)
{
    GETBIDENTITY
    WORD *ll, *m, *w, *llf, *OldWork, *StartWork, *ww, *mm, *t, *OldTermBuffer = 0;
    WORD power = 0, match = 0, i, msign = 0, ll2;
    int numdollars = 0, protosize, oldallnumrhs;
    CBUF *C = cbuf+AM.rbufnum, *CC;
    AT.idallflag = 0;
    do {
/*
        #[ Preliminaries :
*/
    ll = C->lhs[*level];
    if ( *ll == TYPEEXPRESSION ) {
/*
        Expressions are not subject to anything.
*/
        return(0);
    }
    else if ( *ll == TYPEREPEAT ) {
        *++AN.RepPoint = 0;
        return(0);          /* Will force the next level */
    }
    else if ( *ll == TYPEENDREPEAT ) {
        if ( *AN.RepPoint ) {
            AN.RepPoint[-1] = 1;        /* Mark the higher level as dirty */
            *AN.RepPoint = 0;
            *level = ll[2];         /* Level to jump back to */
        }
        else {
            AN.RepPoint--;
            if ( AN.RepPoint < AT.RepCount ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Internal problems with REPEAT count");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
        }
        return(0);          /* Force the next level */
    }
    else if ( *ll == TYPEOPERATION ) {
/*
        Operations have always their own level.
*/
        if ( (*(FG.OperaFind[ll[2]]))(BHEAD term,ll) ) return(-1);
        else return(0);
    }
/*
        #] Preliminaries :
*/
    OldWork = AT.WorkPointer;
    if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
    ww = AT.WorkPointer;
/*
        Here we need to make a copy of the subexpression object because we
        will be writing the values of the wildcards in it.
        Originally we copied it into the private version of the compiler buffer
        that is used for scratch space (ebufnum). This caused errors in the
        routines like ScanFunctions when the ebufnum Buffer was expanded
        and inpat was still pointing at the old Buffer. This expansion
        could be done in AddWild and hence cannot be fixed at > 100 places.
        The solution is to use AN.patternbuffer (JV 16-mar-2009).
*/
    {
        WORD *ta = ll, *ma;
        int ja = ta[1];
/*
        New code (16-mar-2009) JV
```

```
*/
        if ( ( ja + 2 ) > AN.patternbuffersize ) {
            if ( AN.patternbuffer ) M_free(AN.patternbuffer,"AN.patternbuffer");
            AN.patternbuffersize = 2 * ja + 2;
            AN.patternbuffer = (WORD *)Malloc1(AN.patternbuffersize * sizeof(WORD),
                    "AN.patternbuffer");
        }
        ma = AN.patternbuffer;
        m = ma + IDHEAD;
        NCOPY(ma,ta,ja);
        *ma = 0;
    }
    AN.FullProto = m;
    AN.WildValue = w = m + SUBEXPSIZE;
    protosize = IDHEAD + m[1];
    m += m[1];
    AN.WildStop = m;
    StartWork = ww;
    ll2 = ll[2];
/*
        #[ Expand dollars :
*/
    if ( ( ll[4] & DOLLARFLAG ) != 0 ) {    /* We have at least one dollar in the pattern */
        WORD oldRepPoint = *AN.RepPoint, olddefer = AR.DeferFlag;
        AR.Eside = LHSIDEX;
/*
        Copy into WorkSpace. This means that AN.patternbuffer will be free.
*/
        ww = AT.WorkPointer; i = m[0]; mm = m;
        NCOPY(ww,mm,i);
        *StartWork += 3;
        *ww++ = 1; *ww++ = 1; *ww++ = 3;
        AT.WorkPointer = ww;
        AR.DeferFlag = 0;
        NewSort(BHEAD0);
        if ( Generator(BHEAD StartWork,AR.Cnumlhs) ) {
            LowerSortLevel();
            AT.WorkPointer = OldWork;
            AR.DeferFlag = olddefer;
            return(-1);
        }
        AT.WorkPointer = ww;
        if ( EndSort(BHEAD ww,0) < 0 ) {}
        AR.DeferFlag = olddefer;
        if ( *ww == 0 || *(ww+*ww) != 0 ) {
            if ( AP.lhdollarerror == 0 ) {
/*
                If race condition we just get more error messages
*/
                MLOCK(ErrorMessageLock);
                MesPrint("&LHS must be one term");
                MUNLOCK(ErrorMessageLock);
                AP.lhdollarerror = 1;
            }
            AT.WorkPointer = OldWork;
            return(-1);
        }
        m = ww; ww = m + *m;
        if ( m[*m-1] < 0 ) { msign = 1; m[*m-1] = -m[*m-1]; }
        if ( *ww || m[*m-1] != 3 || m[*m-2] != 1 || m[*m-3] != 1 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Dollar variable develops into an illegal pattern in id-statement");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        *m -= m[*m-1];
        if ( ( *m + 1 + protosize ) > AN.patternbuffersize ) {
            if ( AN.patternbuffer ) M_free(AN.patternbuffer,"AN.patternbuffer");
            AN.patternbuffersize = 2 * (*m) + 2 + protosize;
            AN.patternbuffer = (WORD *)Malloc1(AN.patternbuffersize * sizeof(WORD),
                    "AN.patternbuffer");
            mm = ll; ww = AN.patternbuffer; i = protosize;
            NCOPY(ww,mm,i);
            AN.FullProto = AN.patternbuffer + IDHEAD;
            AN.WildValue = w = AN.FullProto + SUBEXPSIZE;
            AN.WildStop = AN.patternbuffer + protosize;
        }
        mm = AN.patternbuffer + protosize;
        i = *m;
        NCOPY(mm,m,i);
        m = AN.patternbuffer + protosize;
        AR.Eside = RHSIDE;
        *mm = 0;
/*
        Test the pattern. If only wildcard powers -> SUBONCE
*/
        {
```

```
            WORD *mmm = m + *m, *m1 = m+1, jm, noveto = 0;
            while ( m1 < mmm ) {
                if ( *m1 == SYMBOL ) {
                    for ( jm = 2; jm < m1[1]; jm+=2 ) {
                        if ( m1[jm+1] < MAXPOWER && m1[jm+1] > -MAXPOWER ) break;
                    }
                    if ( jm < m1[1] ) { noveto = 1; break; }
                }
                else if ( *m1 == DOTPRODUCT ) {
                    for ( jm = 2; jm < m1[1]; jm+=3 ) {
                        if ( m1[jm+2] < MAXPOWER && m1[jm+2] > -MAXPOWER ) break;
                    }
                    if ( jm < m1[1] ) { noveto = 1; break; }
                }
                else { noveto = 1; break; }
                m1 += m1[1];
            }
            if ( noveto == 0 ) {
                ll2 = ll2 & ~SUBMASK;
                ll2 |= SUBONCE;
            }
        }
        AT.WorkPointer = ww = StartWork;
        *AN.RepPoint = oldRepPoint;
    }
/*
        #] Expand dollars :

    In case of id,all we have to check at this point that there are only
    functions in the pattern.
*/
    if ( ( ll2 & SUBMASK ) == SUBALL ) {
        WORD *t = AN.patternbuffer+IDHEAD, *tt;
        WORD *tstop, *ttstop, ii;
        t += t[1]; tstop = t + *t; t++;
        while ( t < tstop ) {
            if ( *t < FUNCTION ) break;
            t += t[1];
        }
        if ( t < tstop ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Error: id,all can only be used with (products of) functions and/or tensors.");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        OldTermBuffer = AN.termbuffer;
        AN.termbuffer = TermMalloc("id,all");
/*
        Now make sure that only regular functions and tensors can take part.
*/
        tt = term; ttstop = tt+*tt; ttstop -= ABS(ttstop[-1]); tt++;
        t = AN.termbuffer+1;
        while ( tt < ttstop ) {
            if ( *tt >= FUNCTION && *tt != AR.PolyFun && *tt != AR.PolyFunInv ) {
                ii = tt[1]; NCOPY(t,tt,ii);
            }
            else tt += tt[1];
        }
        *t++ = 1; *t++ = 1; *t++ = 3; AN.termbuffer[0] = t-AN.termbuffer;
    }
/*
    To be puristic, we need to check that all wildcards in the prototype
    are actually present. If the LHS contained a replace_ this may not be
    the case.
*/
    ClearWild(BHEAD0);
    while ( w < AN.WildStop ) {
        if ( *w == LOADDOLLAR ) numdollars++;
        w += w[1];
    }
    AN.RepFunNum = 0;
    /* rep = */ AN.RepFunList = AT.WorkPointer;
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer/2);
    if ( AT.WorkPointer >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    AN.DisOrderFlag = ll2 & SUBDISORDER;
    AN.nogroundlevel = 0;
    switch ( ll2 & SUBMASK ) {
        case SUBONLY :
            /* Must be an exact match */
            AN.UseFindOnly = 1; AN.ForFindOnly = 0;
            if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind ||
                FindOnly(BHEAD term,m) ) ) {
```

```
                power = 1;
                if ( msign ) term[term[0]-1] = -term[term[0]-1];
            }
            else power = 0;
            break;
        case SUBMANY :
            AN.UseFindOnly = -1;
            if ( ( power = FindRest(BHEAD term,m) ) > 0 ) {
                if ( ( power = FindOnce(BHEAD term,m) ) > 0 ) {
                    AN.UseFindOnly = 0;
                    do {
                        if ( msign ) term[term[0]-1] = -term[term[0]-1];
                        Substitute(BHEAD term,m,1);
                        if ( numdollars ) {
                            WildDollars(BHEAD (WORD *)0);
                            numdollars = 0;
                        }
                        if ( ww < term+term[0] ) ww = term+term[0];
                        ClearWild(BHEAD0);
                        AT.WorkPointer = ww;
/*                      if ( rep < ww ) {*/
                            AN.RepFunNum = 0;
                            /* rep = */ AN.RepFunList = ww;
                            AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer/2);
                            if ( AT.WorkPointer >= AT.WorkTop ) {
                                MLOCK(ErrorMessageLock);
                                MesWork();
                                MUNLOCK(ErrorMessageLock);
                                return(-1);
                            }
/*
                        }
                        else {
                            AN.RepFunList = rep;
                            AN.RepFunNum = 0;
                        }
*/
                        AN.nogroundlevel = 0;
                    } while ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind ||
                            FindOnce(BHEAD term,m) ) );
                    match = 1;
                }
                else if ( power < 0 ) {
                    do {
                        if ( msign ) term[term[0]-1] = -term[term[0]-1];
                        Substitute(BHEAD term,m,1);
                        if ( numdollars ) {
                            WildDollars(BHEAD (WORD *)0);
                            numdollars = 0;
                        }
                        if ( ww < term+term[0] ) ww = term+term[0];
                        ClearWild(BHEAD0);
                        AT.WorkPointer = ww;
/*                      if ( rep < ww ) { */
                            AN.RepFunNum = 0;
                            /* rep = */ AN.RepFunList = ww;
                            AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer/2);
                            if ( AT.WorkPointer >= AT.WorkTop ) {
                                MLOCK(ErrorMessageLock);
                                MesWork();
                                MUNLOCK(ErrorMessageLock);
                                return(-1);
                            }
/*
                        }
                        else {
                            AN.RepFunList = rep;
                            AN.RepFunNum = 0;
                        }
*/
                    } while ( FindRest(BHEAD term,m) );
                    match = 1;
                }
            }
            else if ( power < 0 ) {
                if ( FindOnce(BHEAD term,m) ) {
                    do {
                        if ( msign ) term[term[0]-1] = -term[term[0]-1];
                        Substitute(BHEAD term,m,1);
                        if ( numdollars ) {
                            WildDollars(BHEAD (WORD *)0);
                            numdollars = 0;
                        }
                        if ( ww < term+term[0] ) ww = term+term[0];
                        ClearWild(BHEAD0);
                        AT.WorkPointer = ww;
/*                      if ( rep < ww ) { */
```

```
                            AN.RepFunNum = 0;
                            /* rep = */ AN.RepFunList = ww;
                            AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer/2);
                            if ( AT.WorkPointer >= AT.WorkTop ) {
                                MLOCK(ErrorMessageLock);
                                MesWork();
                                MUNLOCK(ErrorMessageLock);
                                return(-1);
                            }
/*
                        }
                        else {
                            AN.RepFunList = rep;
                            AN.RepFunNum = 0;
                        }
*/
                    } while ( FindOnce(BHEAD term,m) );
                    match = 1;
                }
            }
            if ( match ) {
                if ( ( ll2 & SUBAFTER ) != 0 ) *level = AC.Labels[ll[3]];
            }
            else {
                if ( ( ll2 & SUBAFTERNOT ) != 0 ) *level = AC.Labels[ll[3]];
            }
            goto nextlevel;
        case SUBONCE :
            AN.UseFindOnly = 0;
            if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind || FindOnce(BHEAD term,m) ) ) {
                power = 1;
                if ( msign ) term[term[0]-1] = -term[term[0]-1];
            }
            else power = 0;
            break;
        case SUBMULTI :
            power = FindMulti(BHEAD term,m);
            if ( ( power & 1 ) != 0 && msign ) term[term[0]-1] = -term[term[0]-1];
            break;
        case SUBVECTOR :
            while ( ( power = FindAll(BHEAD term,m,*level,(WORD *)0) ) != 0 ) {
                if ( ( power & 1 ) != 0 && msign ) term[term[0]-1] = -term[term[0]-1];
                match = 1;
            }
            break;
        case SUBSELECT :
            llf = ll + IDHEAD;  llf += llf[1];  llf += *llf;
            AN.UseFindOnly = 1; AN.ForFindOnly = llf;
            if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind || FindOnly(BHEAD term,m) ) ) {
                if ( msign ) term[term[0]-1] = -term[term[0]-1];
/*
                The following code needs to be hacked a bit to allow for
                all types of sets and for occurrence anywhere in the term
                The code at the end of FindOnly is a bit mysterious.
*/
                if ( llf[1] > 2 ) {
                    WORD *t1, *t2;
                    if ( *term > AN.sizeselecttermundo ) {
                        if ( AN.selecttermundo ) M_free(AN.selecttermundo,"AN.selecttermundo");
                        AN.sizeselecttermundo = *term +10;
                        AN.selecttermundo = (WORD *)Malloc1(
                            AN.sizeselecttermundo*sizeof(WORD),"AN.selecttermundo");
                    }
                    t1 = term; t2 = AN.selecttermundo; i = *term;
                    NCOPY(t2,t1,i);
                }
                power = 1;
                Substitute(BHEAD term,m,power);
                if ( llf[1] > 2 ) {
                    if ( TestSelect(term,llf) ) {
                        WORD *t1, *t2;
                        power = 0;
                        t1 = term; t2 = AN.selecttermundo; i = *t2;
                        NCOPY(t1,t2,i);
#if IDHEAD > 3
                        if ( ( ll2 & SUBAFTERNOT ) != 0 ) {
                            *level = AC.Labels[ll[3]];
                        }
#endif
                        goto nextlevel;
                    }
                }
                if ( numdollars ) {
                    WildDollars(BHEAD (WORD *)0);
                    numdollars = 0;
                }
                match = 1;
```

```
                if ( ( ll2 & SUBAFTER ) != 0 ) {
                    *level = AC.Labels[ll[3]];
                }
            }
            else {
                if ( ( ll2 & SUBAFTERNOT ) != 0 ) {
                    *level = AC.Labels[ll[3]];
                }
                power = 0;
            }
            goto nextlevel;
        case SUBALL:
            AN.UseFindOnly = 0;
            CC = cbuf+AT.allbufnum;
            oldallnumrhs = CC->numrhs;
            t = AddRHS(AT.allbufnum,1);
            *t = 0;
            AT.idallflag = 1;
            AT.idallmaxnum = ll[5];
            AT.idallnum = 0;
            if ( FindRest(BHEAD AN.termbuffer,m) || AT.idallflag > 1 ) {
                WORD *t, *tstop, *tt, first = 1, ii;
                power = 1;
                *CC->Pointer++ = 0;
                if ( msign ) term[term[0]-1] = -term[term[0]-1];
/*
                If we come here the matches are all already in the
                compiler buffer. All we need to do is take out all
                functions and replace them by a SUBEXPRESSION that
                points to this buffer.
                Note: the PolyFun/PolyRatFun should be excluded from this.
                This works because each match writes incrementally to
                the buffer using the routine SubsInAll.

                The call to WildDollars should be made in Generator.....
*/
                t = term; tstop = t + *t; ii = ABS(tstop[-1]); tstop -= ii;
                tt = AT.WorkPointer+1;
                t++;
                while ( t < tstop ) {
                    if ( *t >= FUNCTION && *t != AR.PolyFun && *t != AR.PolyFunInv ) {
                        if ( first ) { /* SUBEXPRESSION */
                            *tt++ = SUBEXPRESSION;
                            *tt++ = SUBEXPSIZE;
                            *tt++ = CC->numrhs;
                            *tt++ = 1;
                            *tt++ = AT.allbufnum;
                            FILLSUB(tt)
                            first = 0;
                        }
                        t += t[1];
                    }
                    else {
                        i = t[1]; NCOPY(tt,t,i);
                    }
                }
                if ( ( ll[4] & NORMALIZEFLAG ) != 0 ) {
/*
                    In case of the normalization option, we have to divide
                    by AT.idallnum;
*/
                    WORD na = t[ii-1];
                    na = REDLENG(na);
                    for ( i = 0; i < ii; i++ ) tt[i] = t[i];
                    Divvy(BHEAD (UWORD *)tt,&na,(UWORD *)(&(AT.idallnum)),1);
                    na = INCLENG(na);
                    ii = ABS(na);
                    tt[ii-1] = na;
                    tt += ii;
                }
                else {
                    NCOPY(tt,t,ii);
                }
                ii = tt-AT.WorkPointer;
                *(AT.WorkPointer) = ii;
                tt = AT.WorkPointer; t = term;
                NCOPY(t,tt,ii);

                if ( ( ll2 & SUBAFTER ) != 0 ) { /* ifmatch -> */
                    *level = AC.Labels[ll[3]];
                }
                TermFree(AN.termbuffer,"id,all");
                AN.termbuffer = OldTermBuffer;
                AT.WorkPointer = AN.RepFunList;
                AT.idallflag = 0;
                CC->Pointer[0] = 0;
                TransferBuffer(AT.aebufnum,AT.ebufnum,AT.allbufnum);
```

```
                return(1);
            }
            AT.idallflag = 0;
            power = 0;
            CC->numrhs = oldallnumrhs;
            TermFree(AN.termbuffer,"id,all");
            AN.termbuffer = OldTermBuffer;
            break;
        default :
            break;
    }
    if ( power ) {
        Substitute(BHEAD term,m,power);
        if ( numdollars ) {
            WildDollars(BHEAD (WORD *)0);
            numdollars = 0;
        }
        match = 1;
        if ( ( ll2 & SUBAFTER ) != 0 ) { /* ifmatch -> */
            *level = AC.Labels[ll[3]];
        }
    }
    else {
        AT.WorkPointer = AN.RepFunList;
        if ( ( ll2 & SUBAFTERNOT ) != 0 ) { /* ifnomatch -> */
            *level = AC.Labels[ll[3]];
        }
    }
nextlevel:;
    } while ( (*level)++ < AR.Cnumlhs && C->lhs[*level][0] == TYPEIDOLD );
    (*level)--;
    AT.WorkPointer = AN.RepFunList;
    return(match);
}

/*
        #] TestMatch :
        #[ Substitute :         VOID Substitute(term,pattern,power)
*/

VOID Substitute(PHEAD WORD *term, WORD *pattern, WORD power)
{
    GETBIDENTITY
    WORD *TemTerm;
    WORD *t, *m;
    WORD *tstop, *mstop;
    WORD *xstop, *ystop;
    WORD nt, *fill, nq, mt;
    WORD *q, *subterm, *tcoef, oldval1 = 0, newval3, i = 0;
    WORD PutExpr = 0, sign = 0;
    TemTerm = AT.WorkPointer;
    if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer*2) ) > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    m = pattern;
    mstop = m + *m;
    m++;
    t = term;
    t += *term - 1;
    tcoef = t;
    tstop = t - ABS(*t) + 1;
    t = term;
    t++;
    fill = TemTerm;
    fill++;
    if ( m < mstop ) { do {
/*
            #[ SYMBOLS :
*/
        if ( *m == SYMBOL ) {
            ystop = m + m[1];
            m += 2;
            while ( *t != SYMBOL && t < tstop ) {
                nq = t[1];
                NCOPY(fill,t,nq);
            }
            if ( t >= tstop ) goto SubCoef;
            *fill++ = SYMBOL;
            fill++;
            subterm = fill;
            xstop = t + t[1];
            t += 2;
            do {
                if ( *m == *t && t < xstop ) {
```

```
                    nt = t[1];
                    mt = m[1];
                    if ( mt >= 2*MAXPOWER ) {
                        if ( CheckWild(BHEAD mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
                            nt -= AN.oldvalue;
                            goto SubsL1;
                        }
                    }
                    else if ( mt <= -2*MAXPOWER ) {
                        if ( CheckWild(BHEAD -mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
                            nt += AN.oldvalue;
                            goto SubsL1;
                        }
                    }
                    else {
                        nt -= mt * power;
SubsL1:                 if ( nt ) {
                            *fill++ = *t;
                            *fill++ = nt;
                        }
                    }
                    m += 2; t+= 2;
                }
                else if ( *m >= 2*MAXPOWER ) {
                    while ( t < xstop ) { *fill++ = *t++; *fill++ = *t++; }
                    nq = WORDDIF(fill,subterm);
                    fill = subterm;
                    while ( nq > 0 ) {
                        if ( !CheckWild(BHEAD *m-2*MAXPOWER,SYMTOSYM,*fill,&newval3) ) {
                            mt = m[1];
                            if ( mt >= 2*MAXPOWER ) {
                                if ( CheckWild(BHEAD mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
                                    if ( fill[1] -= AN.oldvalue ) goto SubsL2;
                                }
                            }
                            else if ( mt <= -2*MAXPOWER ) {
                                if ( CheckWild(BHEAD -mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
                                    if ( fill[1] += AN.oldvalue ) goto SubsL2;
                                }
                            }
                            else {
                                if ( fill[1] -= mt * power ) {
SubsL2:                             fill += nq;
                                    nq = 0;
                                }
                            }
                            break;
                        }
                        nq -= 2;
                        fill += 2;
                    }
                    if ( nq ) {
                        nq -= 2;
                        q = fill + 2;
                        while ( --nq >= 0 ) *fill++ = *q++;
                    }
                    m += 2;
                }
                else if ( *m < *t || t >= xstop ) { m += 2; }
                else { *fill++ = *t++; *fill++ = *t++; }
            } while ( m < ystop );
            while ( t < xstop ) *fill++ = *t++;
            nq = WORDDIF(fill,subterm);
            if ( nq > 0 ) {
                nq += 2;
                subterm[-1] = nq;
            }
            else { fill = subterm; fill -= 2; }
        }
/*
            #] SYMBOLS :
            #[ DOTPRODUCTS :
*/
        else if ( *m == DOTPRODUCT ) {
            ystop = m + m[1];
            m += 2;
            while ( *t > DOTPRODUCT && t < tstop ) {
                nq = t[1];
                NCOPY(fill,t,nq);
            }
            if ( t >= tstop ) goto SubCoef;
            if ( *t != DOTPRODUCT ) {
                m = ystop;
                goto EndLoop;
            }
            *fill++ = DOTPRODUCT;
            fill++;
```

```
            subterm = fill;
            xstop = t + t[1];
            t += 2;
            do {
                if ( *m == *t && m[1] == t[1] && t < xstop ) {
                    nt = t[2];
                    mt = m[2];
                    if ( mt >= 2*MAXPOWER ) {
                        if ( CheckWild(BHEAD mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
                            nt -= AN.oldvalue;
                            goto SubsL3;
                        }
                    }
                    else if ( mt <= -2*MAXPOWER ) {
                        if ( CheckWild(BHEAD -mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
                            nt += AN.oldvalue;
                            goto SubsL3;
                        }
                    }
                    else {
                        nt -= mt * power;
SubsL3:                 if ( nt ) {
                            *fill++ = *t++;
                            *fill++ = *t;
                            *fill++ = nt;
                            t += 2;
                        }
                        else t += 3;
                    }
                    m += 3;
                }
                else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
                    while ( t < xstop ) {
                        *fill++ = *t++; *fill++ = *t++; *fill++ = *t++;
                    }
                    oldval1 = 1;
                    goto SubsL4;
                }
                else if ( m[1] >= (AM.OffsetVector+WILDOFFSET) ) {
                    while ( *m >= *t && t < xstop ) {
                        *fill++ = *t++; *fill++ = *t++; *fill++ = *t++;
                    }
                    oldval1 = 0;
SubsL4:             nq = WORDDIF(fill,subterm);
                    fill = subterm;
                    while ( nq > 0 ) {
                        if ( ( oldval1 && ( (
                           !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*fill,&newval3)
                        && !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,fill[1],&newval3)
                        ) || (
                           !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*fill,&newval3)
                        && !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,fill[1],&newval3)
                        ) ) ) || ( !oldval1 && ( (
                           *m == *fill
                        && !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,fill[1],&newval3)
                        ) || (
                           !CheckWild(BHEAD m[1]-WILDOFFSET,VECTOVEC,*fill,&newval3)
                        && *m == fill[1] ) ) ) ) {
                            mt = m[2];
                            if ( mt >= 2*MAXPOWER ) {
                                if ( CheckWild(BHEAD mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
                                    if ( fill[2] -= AN.oldvalue )
                                            goto SubsL5;
                                }
                            }
                            else if ( mt <= -2*MAXPOWER ) {
                                if ( CheckWild(BHEAD -mt-2*MAXPOWER,SYMTONUM,-MAXPOWER,&newval3) ) {
                                    if ( fill[2] += AN.oldvalue )
                                            goto SubsL5;
                                }
                            }
                            else {
                                if ( fill[2] -= mt * power ) {
SubsL5:                             fill += nq;
                                    nq = 0;
                                }
                            }
                            m += 3;
                            break;
                        }
                        fill += 3; nq -= 3;
                    }
                    if ( nq ) {
                        nq -= 3;
                        q = fill + 3;
                        while ( --nq >= 0 ) *fill++ = *q++;
                    }
```

```
                }
                else if ( t >= xstop || *m < *t || ( *m == *t && m[1] < t[1] ) )
                    { m += 3; }
                else {
                    *fill++ = *t++; *fill++ = *t++; *fill++ = *t++;
                }
            } while ( m < ystop );
            while ( t < xstop ) *fill++ = *t++;
            nq = WORDDIF(fill,subterm);
            if ( nq > 0 ) {
                nq += 2;
                subterm[-1] = nq;
            }
            else { fill = subterm; fill -= 2; }
        }
/*
            #] DOTPRODUCTS :
            #[ FUNCTIONS :
*/
        else if ( *m >= FUNCTION ) {
            while ( *t >= FUNCTION || *t == SUBEXPRESSION ) {
                nt = WORDDIF(t,term);
                for ( mt = 0; mt < AN.RepFunNum; mt += 2 ) {
                    if ( nt == AN.RepFunList[mt] ) break;
                }
                if ( mt >= AN.RepFunNum ) {
                    nq = t[1];
                    NCOPY(fill,t,nq);
                }
                else {
                    WORD *oldt = 0;
                    if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
                        oldt = t;
                        if ( ( i = AN.RepFunList[mt+1] ) > 0 ) {
                            *fill++ = GAMMA;
                            *fill++ = i + FUNHEAD+1;
                            FILLFUN(fill)
                            nq = i + 1;
                            t += FUNHEAD;
                            NCOPY(fill,t,nq);
                        }
                        t = oldt;
                    }
                    else if ( ( *t == LEVICIVITA ) || ( *t >= FUNCTION
                    && (functions[*t-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
                                    ) sign += AN.RepFunList[mt+1];
                    else if ( *m >= FUNCTION+WILDOFFSET
                    && (functions[*m-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER) == ANTISYMMETRIC
                                    ) sign += AN.RepFunList[mt+1];
                    if ( !PutExpr ) {
                        xstop = t + t[1];
                        t = AN.FullProto;
                        nq = t[1];
                        t[3] = power;
                        NCOPY(fill,t,nq);
                        t = xstop;
                        PutExpr = 1;
                    }
                    else t += t[1];
                    if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
                        i = oldt[1] - m[1] - i;
                        if ( i > 0 ) {
                            *fill++ = GAMMA;
                            *fill++ = i + FUNHEAD+1;
                            FILLFUN(fill)
                            *fill++ = oldt[FUNHEAD];
                            t = t - i;
                            NCOPY(fill,t,i);
                        }
                    }
                    break;
                }
            }
            m += m[1];
        }
/*
            #] FUNCTIONS :
            #[ VECTORS :
*/
        else if ( *m == VECTOR ) {
            while ( *t > VECTOR ) {
                nq = t[1];
                NCOPY(fill,t,nq);
            }
            xstop = t + t[1];
            ystop = m + m[1];
            t += 2;
```

```
            m += 2;
            *fill++ = VECTOR;
            fill++;
            subterm = fill;
            do {
                if ( *m == *t && m[1] == t[1] ) {
                    m += 2; t += 2;
                }
                else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
                    while ( t < xstop ) *fill++ = *t++;
                    nq = WORDDIF(fill,subterm);
                    fill = subterm;
                    if ( m[1] < (AM.OffsetIndex+WILDOFFSET) ) {
                        do {
                            if ( m[1] == fill[1] &&
                            !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*fill,&newval3) )
                                break;
                            fill += 2;
                            nq -= 2;
                        } while ( nq > 0 );
                    }
                    else {      /* Double wildcard */
                        do {
                            if ( !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3)
                            && !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*fill,&newval3) )
                                break;
                            if ( *fill == oldval1 && fill[1] == AN.oldvalue ) break;
                            fill += 2;
                            nq -= 2;
                        } while ( nq > 0 );
                    }
                    nq -= 2;
                    q = fill + 2;
                    if ( nq > 0 ) { NCOPY(fill,q,nq); }
                    m += 2;
                }
                else if ( *m <= *t &&
                m[1] >= (AM.OffsetIndex + WILDOFFSET) ) {
                    while ( *m == *t && t < xstop )
                        { *fill++ = *t++; *fill++ = *t++; }
                    nq = WORDDIF(fill,subterm);
                    fill = subterm;
                    do {
                        if ( *m == *fill &&
                        !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3) )
                            break;
                        nq -= 2;
                        fill += 2;
                    } while ( nq > 0 );
                    nq -= 2;
                    q = fill + 2;
                    if ( nq > 0 ) { NCOPY(fill,q,nq); }
                    m += 2;
                }
                else { *fill++ = *t++; *fill++ = *t++; }
            } while ( m < ystop );
            while ( t < xstop ) *fill++ = *t++;
            nq = WORDDIF(fill,subterm);
            if ( nq > 0 ) {
                nq += 2;
                subterm[-1] = nq;
            }
            else { fill = subterm; fill -= 2; }
        }
/*
            #] VECTORS :
            #[ INDICES :

            Currently without wildcards
*/
        else if ( *m == INDEX ) {
            while ( *t > INDEX ) {
                nq = t[1];
                NCOPY(fill,t,nq);
            }
            xstop = t + t[1];
            ystop = m + m[1];
            t += 2;
            m += 2;
            *fill++ = INDEX;
            fill++;
            subterm = fill;
            do {
                if ( *m == *t ) {
                    m += 1; t += 1;
                }
                else if ( *m >= (AM.OffsetIndex+WILDOFFSET) ) {
```

```
                    while ( t < xstop ) *fill++ = *t++;
                    nq = WORDDIF(fill, subterm);
                    fill = subterm;
                    do {
                        if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*fill,&newval3) ) {
                            break;
                        }
                        fill += 1;
                        nq -= 1;
                    } while ( nq > 0 );
                    nq -= 1;
                    if ( nq > 0 ) {
                        q = fill + 1;
                        NCOPY(fill,q,nq);
                    }
                    m += 1;
                }
                else {
                    *fill++ = *t++;
                }
            } while ( m < ystop );
            while ( t < xstop ) *fill++ = *t++;
            nq = WORDDIF(fill,subterm);
            if ( nq > 0 ) {
                nq += 2;
                subterm[-1] = nq;
            }
            else { fill = subterm; fill -= 2; }
        }
/*
            #] INDICES :
            #[ DELTAS :
*/
        else if ( *m == DELTA ) {
            while ( *t > DELTA ) {
                nq = t[1];
                NCOPY(fill,t,nq);
            }
            xstop = t + t[1];
            ystop = m + m[1];
            t += 2;
            m += 2;
            *fill++ = DELTA;
            fill++;
            subterm = fill;
            do {
                if ( *t == *m && t[1] == m[1] ) { m += 2; t += 2; }
                else if ( *m >= (AM.OffsetIndex+WILDOFFSET) ) { /* Two dummies */
                    while ( t < xstop ) *fill++ = *t++;
/*                  fill = subterm; */
                    oldval1 = 1;
                    goto SubsL6;
                }
                else if ( m[1] >= (AM.OffsetIndex+WILDOFFSET) ) {
                    while ( (*m == *t || *m == t[1] ) && ( t < xstop ) ) {
                        *fill++ = *t++; *fill++ = *t++;
                    }
                    oldval1 = 0;
SubsL6:             nq = WORDDIF(fill,subterm);
                    fill = subterm;
                    do {
                        if ( ( oldval1 && ( (
                           !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*fill,&newval3)
                        && !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3)
                        ) || (
                           !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,*fill,&newval3)
                        && !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,fill[1],&newval3)
                        ) ) ) || ( !oldval1 && ( (
                           *m == *fill
                        && !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3)
                        ) || (
                           *m == fill[1]
                        && !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,*fill,&newval3)
                        ) ) ) ) break;
                        fill += 2;
                        nq -= 2;
                    } while ( nq > 0 );
                    nq -= 2;
                    if ( nq > 0 ) {
                        q = fill + 2;
                        NCOPY(fill,q,nq);
                    }
                    m += 2;
                }
                else {
                    *fill++ = *t++; *fill++ = *t++;
                }
```

```
            } while ( m < ystop );
            while ( t < xstop ) *fill++ = *t++;
            nq = WORDDIF(fill,subterm);
            if ( nq > 0 ) {
                nq += 2;
                subterm[-1] = nq;
            }
            else { fill = subterm; fill -= 2; }
        }
/*
            #] DELTAS :
*/
EndLoop:;
    } while ( m < mstop ); }
    while ( t < tstop ) *fill++ = *t++;
SubCoef:
    if ( !PutExpr ) {
        t = AN.FullProto;
        nq = t[1];
        t[3] = power;
        NCOPY(fill,t,nq);
    }
    t = tcoef;
    nq = ABS(*t);
    t = tstop;
    NCOPY(fill,t,nq);
    nq = WORDDIF(fill,TemTerm);
    fill = term;
    t = TemTerm;
    *fill++ = nq--;
    t++;
    NCOPY(fill,t,nq);
    if ( sign ) {
        if ( ( sign & 1 ) != 0 ) fill[-1] = -fill[-1];
    }
    if ( AT.WorkPointer < fill ) AT.WorkPointer = fill;
    AN.RepFunNum = 0;
}

/*
        #] Substitute :
        #[ FindSpecial :        WORD FindSpecial(term)

    Routine to detect symplifications regarding the special functions
    exponent, denominator.


WORD FindSpecial(WORD *term)
{
    WORD *t;
    WORD *tstop;
    t = term; t += *t - 1; tstop = t - ABS(*t) + 1; t = term;
    t++;
    if ( t < tstop ) { do {
        if ( *t == EXPONENT ) {
            Exponents can become simpler when:
            a: the exponent of an expression becomes an integer.
            b: The expression becomes zero.
        }
        else if ( *t == DENOMINATOR ) {
            Denominators can become simpler when:
            a: The denominator is a single term without functions.
            b: An overall coefficient can be removed.
            c: An overall object can be removed.
            The task is here to bring the denominator in an unique form.
        }
        t += *t;
    } while ( t < tstop ); }
    return(0);
}

        #] FindSpecial :
        #[ FindAll :            WORD FindAll(term,pattern,level,par)
*/

WORD FindAll(PHEAD WORD *term, WORD *pattern, WORD level, WORD *par)
{
    GETBIDENTITY
    WORD *t, *m, *r, *mm, rnum;
    WORD *tstop, *mstop, *TwoProto, *vwhere = 0, oldv, oldvv, vv, level2;
    WORD v, nq, OffNum = AM.OffsetVector + WILDOFFSET, i, ii = 0, jj;
    WORD fromindex, *intens, notflag1 = 0, notflag2 = 0;
    CBUF *C;
    C = cbuf+AM.rbufnum;
    v = pattern[3];     /* The vector to be found */
    m = t = term;
    m += *m;
```

```
    m -= ABS(m[-1]);
    t++;
    if ( t < m ) do {
        tstop = t + t[1];
        fromindex = 2;
/*
            #[ VECTOR :
*/
        if ( *t == VECTOR ) {
            r = t;
            r += 2;
InVect:
            while ( r < tstop ) {
                oldv = *r;
                if ( v >= OffNum ) {
                    vwhere = AN.FullProto + 3 + SUBEXPSIZE;
                    if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
                        WORD *afirst, *alast, j;
                        j = vwhere[3];
                        if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
                        else { notflag1 = 0; }
                        afirst = SetElements + Sets[j].first;
                        alast  = SetElements + Sets[j].last;
                        ii = 1;
                        if ( notflag1 == 0 ) {
                          do {
                            if ( *afirst == *r ) {
                                if ( vwhere[1] == SETTONUM ) {
                                    AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
                                    AN.FullProto[11+SUBEXPSIZE] = ii;
                                }
                                else if ( vwhere[4] >= 0 ) {
                                    oldv = *(afirst - Sets[j].first
                                    + Sets[vwhere[4]].first);
                                }
                                goto DoVect;
                            }
                            ii++;
                          } while ( ++afirst < alast );
                        }
                        else {
                          do {
                            if ( *afirst == *r ) break;
                          } while ( ++afirst < alast );
                          if ( afirst >= alast ) goto DoVect;
                        }
                    }
                    else goto DoVect;
                }
                else if ( v == *r ) {
DoVect:             m = AT.WorkPointer;
                    tstop = t;
                    t = term;
                    mstop = t + *t;
                    do { *m++ = *t++; } while ( t < tstop );
                    vwhere = m;
                    t = AN.FullProto;
                    nq = t[1];
                    t[3] = 1;
                    NCOPY(m,t,nq);
                    t = tstop;
                    if ( fromindex == 1 ) m[-1] = FUNNYVEC;
                    else m[-1] = r[1];      /* The index is always here! */
                    if ( v >= OffNum ) vwhere[3+SUBEXPSIZE] = oldv;
                    if ( vwhere[1] > 12+SUBEXPSIZE ) {
                        vwhere[11+SUBEXPSIZE] = ii;
                        vwhere[8+SUBEXPSIZE] = SYMTONUM;
                    }
                    if ( t[1] > fromindex+2 ) {
                        *m++ = *t++;
                        *m++ = *t++ - fromindex;
                        while ( t < r ) *m++ = *t++;
                        t += fromindex;
                    }
                    else t += t[1];
                    do { *m++ = *t++; } while ( t < mstop );
                    *AT.WorkPointer = nq = WORDDIF(m,AT.WorkPointer);
                    m = AT.WorkPointer;
                    t = term;
                    NCOPY(t,m,nq);
                    AT.WorkPointer = t;
                    return(1);
                }
                r += fromindex;
            }
        }
/*
```

```
            #] VECTOR :
            #[ DOTPRODUCT :
*/
        else if ( *t == DOTPRODUCT ) {
            r = t;
            r += 2;
            do {
                if ( ( i = r[2] ) < 0 ) goto NextDot;
                if ( *r == r[1] ) {     /* p.p */
                    oldv = *r;
                    if ( v == *r ) {    /* v.v */
TwoVec:                 m = AT.WorkPointer;
                        tstop = t;
                        t = term;
                        mstop = t + *t;
                        do { *m++ = *t++; } while ( t < tstop );
                        do {
                            vwhere = m;
                            t = AN.FullProto;
                            nq = t[1];
                            t[3] = 2;
                            NCOPY(m,t,nq);
                            m[-1] = ++AR.CurDum;
                            if ( v >= OffNum ) vwhere[3+SUBEXPSIZE] = oldv;
                        } while ( --i > 0 );
CopRest:                t = tstop;
                        if ( t[1] > 5 ) {
                            *m++ = *t++;
                            *m++ = *t++ - 3;
                            while ( t < r ) *m++ = *t++;
                            t += 3;
                        }
                        else t += t[1];
                        do { *m++ = *t++; } while ( t < mstop );
                        *AT.WorkPointer = nq = WORDDIF(m,AT.WorkPointer);
                        m = AT.WorkPointer;
                        t = term;
                        NCOPY(t,m,nq);
                        AT.WorkPointer = t;
                        return(1);
                    }
                    else if ( v >= OffNum ) {   /* v?.v? */
                        vwhere = AN.FullProto + 3+SUBEXPSIZE;
                        if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
                            WORD *afirst, *alast, j;
                            j = vwhere[3];
                            if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
                            else { notflag1 = 0; }
                            afirst = SetElements + Sets[j].first;
                            alast  = SetElements + Sets[j].last;
                            ii = 1;
                            if ( notflag1 == 0 ) {
                              do {
                                if ( *afirst == *r ) {
                                    if ( vwhere[1] == SETTONUM ) {
                                        AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
                                        AN.FullProto[11+SUBEXPSIZE] = ii;
                                    }
                                    else if ( vwhere[4] >= 0 ) {
                                        oldv = *(afirst - Sets[j].first
                                        + Sets[vwhere[4]].first);
                                    }
                                    goto TwoVec;
                                }
                                ii++;
                              } while ( ++afirst < alast );
                            }
                            else {
                              do {
                                if ( *afirst == *r ) break;
                              } while ( ++afirst < alast );
                              if ( afirst >= alast ) goto TwoVec;
                            }
                        }
                        else goto TwoVec;
                    }
                }
                else {
                    if ( v == r[1] ) { r[1] = *r; *r = v; }
                    oldv = *r;
                    oldvv = r[1];
                    if ( v == *r ) {
                        if ( !par ) { while ( ++level <= AR.Cnumlhs
                        && C->lhs[level][0] == TYPEIDOLD ) {
                            m = C->lhs[level];
                            m += IDHEAD;
                            if ( m[-IDHEAD+2] == SUBVECTOR ) {
```

```
                            if ( ( vv = m[m[1]+3] ) == r[1] ) {
OnePV:                          TwoProto = AN.FullProto;
TwoPV:                          m = AT.WorkPointer;
                                tstop = t;
                                t = term;
                                mstop = t + *t;
                                do { *m++ = *t++; } while ( t < tstop );
                                do {
                                    t = AN.FullProto;
                                    vwhere = m + 3 +SUBEXPSIZE;
                                    nq = t[1];
                                    t[3] = 1;
                                    NCOPY(m,t,nq);
                                    m[-1] = ++AR.CurDum;
                                    if ( v >= OffNum ) *vwhere = oldv;
                                    if ( vwhere[-2-SUBEXPSIZE] > 12+SUBEXPSIZE ) {
                                        vwhere[8] = ii;
                                        vwhere[5] = SYMTONUM;
                                    }
                                    t = TwoProto;
                                    vwhere = m + 3+SUBEXPSIZE;
                                    mm = m;
                                    nq = t[1];
                                    t[3] = 1;
                                    NCOPY(m,t,nq);
/*
        The next two lines repair a bug. without them it takes twice
        the rhs of the first vector.
*/
                                    mm[2] = C->lhs[level][IDHEAD+2];
                                    mm[4] = C->lhs[level][IDHEAD+4];
                                    m[-1] = AR.CurDum;
                                    if ( vv >= OffNum ) *vwhere = oldvv;
                                } while ( --i > 0 );
                                goto CopRest;
                            }
                            else if ( vv > OffNum ) {
                                vwhere = AN.FullProto + 3+SUBEXPSIZE;
                                if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
                                    WORD *afirst, *alast, j;
                                    j = vwhere[3];
                                    if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
                                    else { notflag1 = 0; }
                                    afirst = SetElements + Sets[j].first;
                                    alast  = SetElements + Sets[j].last;
                                    if ( notflag1 == 0 ) {
                                      ii = 1;
                                      do {
                                        if ( *afirst == r[1] ) {
                                            if ( vwhere[1] == SETTONUM ) {
                                                AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
                                                AN.FullProto[11+SUBEXPSIZE] = ii;
                                            }
                                            else if ( vwhere[4] >= 0 ) {
                                                oldvv = *(afirst - Sets[j].first
                                                + Sets[vwhere[4]].first);
                                            }
                                            goto OnePV;
                                        }
                                        ii++;
                                      } while ( ++afirst < alast );
                                    }
                                    else {
                                      do {
                                        if ( *afirst == *r ) break;
                                      } while ( ++afirst < alast );
                                      if ( afirst >= alast ) goto OnePV;
                                    }
                                }
                                else goto OnePV;
                            }
                            }
                        }}
/*
            v.q with v matching and no match for the q, also
            not in following idold statements.
            Notice that a following q.p? cannot match.
*/
                        rnum = r[1];
OneOnly:                m = AT.WorkPointer;
                        tstop = t;
                        t = term;
                        mstop = t + *t;
                        do { *m++ = *t++; } while ( t < tstop );
                        vwhere = m;
                        t = AN.FullProto;
                        nq = t[1];
```

```
                        t[3] = i;
                        NCOPY(m,t,nq);
                        m[-4] = INDTOIND;
                        m[-1] = rnum;
                        if ( v >= OffNum ) vwhere[3+SUBEXPSIZE] = oldv;
                        goto CopRest;
                    }
                    else if ( v >= OffNum ) {
                        vwhere = AN.FullProto + 3+SUBEXPSIZE;
                        if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
                            WORD *afirst, *alast, *bfirst, *blast, j;
                            j = vwhere[3];
                            if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
                            else { notflag1 = 0; }
                            afirst = SetElements + Sets[j].first;
                            alast  = SetElements + Sets[j].last;
                            ii = 1;
                            if ( notflag1 == 0 ) {
                              do {
                                if ( *afirst == *r ) {
                                    if ( vwhere[1] == SETTONUM ) {
                                        AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
                                        AN.FullProto[11+SUBEXPSIZE] = ii;
                                    }
                                    else if ( vwhere[4] >= 0 ) {
                                        oldv = *(afirst - Sets[j].first
                                        + Sets[vwhere[4]].first);
                                    }
Hitlevel1:                          level2 = level;
                                    do {
                                        if ( !par ) m = C->lhs[level2];
                                        else m = par;
                                        m += IDHEAD;
                                        if ( m[-IDHEAD+2] == SUBVECTOR ) {
                                        if ( ( vv = m[m[1]+3] ) == r[1] )
                                            goto OnePV;
                                        else if ( vv >= OffNum ) {
                                            if ( m[SUBEXPSIZE+4] != FROMSET &&
                                            m[SUBEXPSIZE+4] != SETTONUM ) goto OnePV;
                                            j = m[SUBEXPSIZE+6];
                                            if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag2 = 1; }
                                            else { notflag2 = 0; }
                                            bfirst = SetElements + Sets[j].first;
                                            blast  = SetElements + Sets[j].last;
                                            jj = 1;
                                            if ( notflag2 == 0 ) {
                                              do {
                                                if ( *bfirst == r[1] ) {
                                                    if ( m[SUBEXPSIZE+4] == SETTONUM ) {
                                                        m[SUBEXPSIZE+8] = SYMTONUM;
                                                        m[SUBEXPSIZE+11] = jj;
                                                    }
                                                    else if ( m[SUBEXPSIZE+7] >= 0 ) {
                                                        oldvv = *(bfirst - Sets[j].first
                                                        + Sets[m[SUBEXPSIZE+7]].first);
                                                    }
                                                    goto OnePV;
                                                }
                                                jj++;
                                              } while ( ++bfirst < blast );
                                            }
                                            else {
                                              do {
                                                if ( *bfirst == r[1] ) break;
                                              } while ( ++bfirst < blast );
                                              if ( bfirst >= blast ) goto OnePV;
                                            }
                                        }
                                            }
                                    } while ( ++level2 < AR.Cnumlhs &&
                                    C->lhs[level2][0] == TYPEIDOLD );
                                    rnum = r[1];
                                    goto OneOnly;
                                }
                                else if ( *afirst == r[1] ) {
                                    if ( vwhere[1] == SETTONUM ) {
                                        AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
                                        AN.FullProto[11+SUBEXPSIZE] = ii;
                                    }
                                    else if ( vwhere[4] >= 0 ) {
                                        oldv = *(afirst - Sets[j].first
                                        + Sets[vwhere[4]].first);
                                    }
Hitlevel2:                          level2 = level;
                                    while ( ++level2 < AR.Cnumlhs &&
                                    C->lhs[level2][0] == TYPEIDOLD ) {
                                        if ( !par ) m = C->lhs[level2];
```

```
                                        else m = par;
                                        m += IDHEAD;
                                        if ( m[-IDHEAD+2] == SUBVECTOR ) {
                                        if ( ( vv = m[6] ) == *r )
                                            goto OnePV;
                                        else if ( vv >= OffNum ) {
                                            if ( m[SUBEXPSIZE+4] != FROMSET && m[SUBEXPSIZE+4]
                                            != SETTONUM ) {
                                                j = *r;
                                                *r = r[1];
                                                r[1] = j;
                                                goto OnePV;
                                            }
                                            j = m[SUBEXPSIZE+6];
                                            bfirst = SetElements + Sets[j].first;
                                            blast  = SetElements + Sets[j].last;
                                            jj = 1;
                                            do {
                                                if ( *bfirst == *r ) {
                                                    if ( m[SUBEXPSIZE+4] == SETTONUM ) {
                                                        m[SUBEXPSIZE+8] = SYMTONUM;
                                                        m[SUBEXPSIZE+11] = jj;
                                                    }
                                                    else if ( m[SUBEXPSIZE+7] >= 0 ) {
                                                        oldvv = *(bfirst - Sets[j].first
                                                        + Sets[m[SUBEXPSIZE+7]].first);
                                                    }
                                                    j = *r;
                                                    *r = r[1];
                                                    r[1] = j;
                                                    j = oldv; oldv = oldvv; oldvv = j;
                                                    goto OnePV;
                                                }
                                                jj++;
                                            } while ( ++bfirst < blast );
                                        }
                                            }
                                    }
                                    jj = *r; *r = r[1]; r[1] = jj;
                                    jj = oldv; oldv = oldvv; oldvv = j;
                                    rnum = r[1];
                                    goto OneOnly;
                                }
                                ii++;
                              } while ( ++afirst < alast );
                            }
                            else {
                              do {
                                if ( *afirst == *r ) break;
                              } while ( ++afirst < alast );
                              if ( afirst >= alast ) goto Hitlevel1;
                              do {
                                if ( *afirst == r[1] ) break;
                              } while ( ++afirst < alast );
                              if ( afirst >= alast ) goto Hitlevel2;
                            }
                        }
                        else { /* Matches twice */
                            vv = v;
                            TwoProto = AN.FullProto;
                            goto TwoPV;
                        }
                    }
                }
NextDot:            r += 3;
            } while ( r < tstop );
        }
/*
            #] DOTPRODUCT :
            #[ LEVICIVITA :
*/
        else if ( *t == LEVICIVITA ) {
            intens = 0;
            r = t;
            r += FUNHEAD;
OneVect:;
            while ( r < tstop ) {
                oldv = *r;
                if ( v >= OffNum && *r < -10 ) {
                    vwhere = AN.FullProto + 3+SUBEXPSIZE;
                    if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
                        WORD *afirst, *alast, j;
                        j = vwhere[3];
                        if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
                        else { notflag1 = 0; }
                        afirst = SetElements + Sets[j].first;
                        alast  = SetElements + Sets[j].last;
```

```
                        ii = 1;
                        if ( notflag1 == 0 ) {
                          do {
                            if ( *afirst == *r ) {
                                if ( vwhere[1] == SETTONUM ) {
                                    AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
                                    AN.FullProto[11+SUBEXPSIZE] = ii;
                                }
                                else if ( vwhere[4] >= 0 ) {
                                    oldv = *(afirst - Sets[j].first
                                    + Sets[vwhere[4]].first);
                                }
                                goto DoVect;
                            }
                            ii++;
                          } while ( ++afirst < alast );
                        }
                        else {
                          do {
                            if ( *afirst == *r ) break;
                          } while ( ++afirst < alast );
                          if ( afirst >= alast ) goto DoVect;
                        }
                    }
                    else goto LeVect;
                }
                else if ( v == *r ) {
LeVect:             m = AT.WorkPointer;
                    mstop = term + *term;
                    t = term;
                    *r = ++AR.CurDum;
                    if ( intens ) *intens = DIRTYSYMFLAG;
                    do { *m++ = *t++; } while ( t < tstop );
                    t = AN.FullProto;
                    nq = t[1];
                    t[3] = 1;
                    if ( v >= OffNum ) *vwhere = oldv;
                    NCOPY(m,t,nq);
                    m[-1] = AR.CurDum;
                    t = tstop;
                    do { *m++ = *t++; } while ( t < mstop );
                    *AT.WorkPointer = nq = WORDDIF(m,AT.WorkPointer);
                    m = AT.WorkPointer;
                    t = term;
                    NCOPY(t,m,nq);
                    AT.WorkPointer = t;
                    return(1);
                }
                r++;
            }
        }
/*
            #] LEVICIVITA :
            #[ GAMMA :
*/
        else if ( *t == GAMMA ) {
            intens = 0;
            r = t;
            r += FUNHEAD+1;
            if ( r < tstop ) goto OneVect;
        }
/*
            #] GAMMA :
            #[ INDEX :
*/
        else if ( *t == INDEX ) {   /* The 'forgotten' part */
            r = t;
            r += 2;
            fromindex = 1;
            goto InVect;
        }
/*
            #] INDEX :
            #[ FUNCTION :
*/
        else if ( *t >= FUNCTION ) {
            if ( *t >= FUNCTION
             && functions[*t-FUNCTION].spec >= TENSORFUNCTION
             && t[1] > FUNHEAD ) {
/*
                Tensors are linear in their vectors!
*/
                r = t;
                r += FUNHEAD;
                intens = t+2;
                goto OneVect;
            }
```

```
        }
/*
            #] FUNCTION :
*/
        t += t[1];
    } while ( t < m );
    return(0);
}

/*
        #] FindAll :
        #[ TestSelect :

        Returns 1 if any of the objects in any of the sets in setp
        occur anywhere in the term
*/

int TestSelect(WORD *term, WORD *setp)
{
    WORD *tstop, *t, *s, *el, *elstop, *termstop, *tt, n, ns;
    GETSTOP(term,tstop);
    term += 1;
    while ( term < tstop ) {
    switch ( *term ) {
        case SYMBOL:
            n = term[1] - 2;
            t = term + 2;
            while ( n > 0 ) {
                ns = setp[1] - 2;
                s = setp + 2;
                while ( --ns >= 0 ) {
                    if ( Sets[*s].type != CSYMBOL ) { s++; continue; }
                    el = SetElements + Sets[*s].first;
                    elstop = SetElements + Sets[*s].last;
                    while ( el < elstop ) {
                        if ( *el++ == *t ) return(1);
                    }
                    s++;
                }
                n -= 2;
                t += 2;
            }
            break;
        case VECTOR:
            n = term[1] - 2;
            t = term + 2;
            while ( n > 0 ) {
                ns = setp[1] - 2;
                s = setp + 2;
                while ( --ns >= 0 ) {
                    if ( Sets[*s].type != CVECTOR ) { s++; continue; }
                    el = SetElements + Sets[*s].first;
                    elstop = SetElements + Sets[*s].last;
                    while ( el < elstop ) {
                        if ( *el++ == *t ) return(1);
                    }
                    s++;
                }
                t++;
                ns = setp[1] - 2;
                s = setp + 2;
                while ( --ns >= 0 ) {
                    if ( Sets[*s].type != CINDEX
                    && Sets[*s].type != CNUMBER ) { s++; continue; }
                    el = SetElements + Sets[*s].first;
                    elstop = SetElements + Sets[*s].last;
                    while ( el < elstop ) {
                        if ( *el++ == *t ) return(1);
                    }
                    s++;
                }
                n -= 2;
                t++;
            }
            break;
        case INDEX:
            n = term[1] - 2;
            t = term + 2;
            goto dotensor;
        case DOTPRODUCT:
            n = term[1] - 2;
            t = term + 2;
            while ( n > 0 ) {
                ns = setp[1] - 2;
                s = setp + 2;
                while ( --ns >= 0 ) {
                    if ( Sets[*s].type != CVECTOR ) { s++; continue; }
```

```
                    el = SetElements + Sets[*s].first;
                    elstop = SetElements + Sets[*s].last;
                    while ( el < elstop ) {
                        if ( *el++ == *t ) return(1);
                    }
                    s++;
                }
                t++;
                ns = setp[1] - 2;
                s = setp + 2;
                while ( --ns >= 0 ) {
                    if ( Sets[*s].type != CVECTOR ) { s++; continue; }
                    el = SetElements + Sets[*s].first;
                    elstop = SetElements + Sets[*s].last;
                    while ( el < elstop ) {
                        if ( *el++ == *t ) return(1);
                    }
                    s++;
                }
                n -= 3;
                t += 2;
            }
            break;
        case DELTA:
            n = term[1] - 2;
            t = term + 2;
            goto dotensor;
        default:
            if ( *term < FUNCTION ) break;
            ns = setp[1] - 2;
            s = setp + 2;
            while ( --ns >= 0 ) {
                if ( Sets[*s].type != CFUNCTION ) { s++; continue; }
                el = SetElements + Sets[*s].first;
                elstop = SetElements + Sets[*s].last;
                while ( el < elstop ) {
                    if ( *el++ == *term ) return(1);
                }
                s++;
            }
            if ( functions[*term-FUNCTION].spec ) {
                n = term[1] - FUNHEAD;
                t = term + FUNHEAD;
dotensor:
                while ( n > 0 ) {
                    ns = setp[1] - 2;
                    s = setp + 2;
                    while ( --ns >= 0 ) {
                        if ( *t < MINSPEC ) {
                            if ( Sets[*s].type != CVECTOR ) { s++; continue; }
                        }
                        else if ( *t >= 0 ) {
                            if ( Sets[*s].type != CINDEX
                            && Sets[*s].type != CNUMBER ) { s++; continue; }
                        }
                        else { s++; continue; }
                        el = SetElements + Sets[*s].first;
                        elstop = SetElements + Sets[*s].last;
                        while ( el < elstop ) {
                            if ( *el++ == *t ) return(1);
                        }
                        s++;
                    }
                    t++;
                    n--;
                }
            }
            else {
                termstop = term + term[1];
                tt = term + FUNHEAD;
                while ( tt < termstop ) {
                    if ( *tt < 0 ) {
                        if ( *tt == -SYMBOL ) {
                            ns = setp[1] - 2;
                            s = setp + 2;
                            while ( --ns >= 0 ) {
                                if ( Sets[*s].type != CSYMBOL ) { s++; continue; }
                                el = SetElements + Sets[*s].first;
                                elstop = SetElements + Sets[*s].last;
                                while ( el < elstop ) {
                                    if ( *el++ == tt[1] ) return(1);
                                }
                                s++;
                            }
                            tt += 2;
                        }
                        else if ( *tt == -VECTOR || *tt == -MINVECTOR ) {
```

```
                            ns = setp[1] - 2;
                            s = setp + 2;
                            while ( --ns >= 0 ) {
                                if ( Sets[*s].type != CVECTOR ) { s++; continue; }
                                el = SetElements + Sets[*s].first;
                                elstop = SetElements + Sets[*s].last;
                                while ( el < elstop ) {
                                    if ( *el++ == tt[1] ) return(1);
                                }
                                s++;
                            }
                            tt += 2;
                        }
                        else if ( *tt == -INDEX ) {
                            ns = setp[1] - 2;
                            s = setp + 2;
                            while ( --ns >= 0 ) {
                                if ( Sets[*s].type != CINDEX
                                && Sets[*s].type != CNUMBER ) { s++; continue; }
                                el = SetElements + Sets[*s].first;
                                elstop = SetElements + Sets[*s].last;
                                while ( el < elstop ) {
                                    if ( *el++ == tt[1] ) return(1);
                                }
                                s++;
                            }
                            tt += 2;
                        }
                        else if ( *tt <= -FUNCTION ) {
                            ns = setp[1] - 2;
                            s = setp + 2;
                            while ( --ns >= 0 ) {
                                if ( Sets[*s].type != CFUNCTION ) { s++; continue; }
                                el = SetElements + Sets[*s].first;
                                elstop = SetElements + Sets[*s].last;
                                while ( el < elstop ) {
                                    if ( *el++ == -(*tt) ) return(1);
                                }
                                s++;
                            }
                            tt++;
                        }
                        else tt += 2;
                    }
                    else {
                        t = tt + ARGHEAD;
                        tt += *tt;
                        while ( t < tt ) {
                            if ( TestSelect(t,setp) ) return(1);
                            t += *t;
                        }
                    }
                }
            }
            break;
    }
    term += term[1];
    }
    return(0);
}

/*
        #] TestSelect :
        #[ SubsInAll :          VOID SubsInAll()

        This routine takes a match in id,all and stores it away in
        the AT.allbufnum 'compiler' buffer, after taking out the pattern.
        The main problem here is that id,all usually has (lots of) wildcards
        and their assignments are on stack and the difficult ones are in
        AT.ebufnum. Popping the stack while looking for more matches would
        loose those. Hence we have to copy them into yet another compiler
        buffer: AT.aebufnum. Because this may involve many matches and
        because the original term has only a limited number of arguments,
        it will pay to look for already existing ones in this buffer.
        (to be done later).
*/

VOID SubsInAll(PHEAD0)
{
    GETBIDENTITY
    WORD *TemTerm;
    WORD *t, *m, *term;
    WORD *tstop, *mstop, *xstop;
    WORD nt, *fill, nq, mt;
    WORD *tcoef, i = 0;
    WORD PutExpr = 0, sign = 0;
/*
```

```
    We start with building the term in the WorkSpace.
    Afterwards we will transfer it to AT.allbufnum.
    We have to make sure there is room in the WorkSpace.
*/
    AT.idallflag = 2;
    TemTerm = AT.WorkPointer;
    if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer*2) ) > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    m = AN.patternbuffer + IDHEAD; m += m[1];
    mstop = m + *m;
    m++;
    term = AN.termbuffer;
    tstop = term + *term; tcoef = tstop-1; tstop -= ABS(tstop[-1]);
    t = term;
    t++;
    fill = TemTerm;
    fill++;
    while ( m < mstop ) {
        while ( t < tstop ) {
            nt = WORDDIF(t,term);
            for ( mt = 0; mt < AN.RepFunNum; mt += 2 ) {
                if ( nt == AN.RepFunList[mt] ) break;
            }
            if ( mt >= AN.RepFunNum ) {
                nq = t[1];
                NCOPY(fill,t,nq);
            }
            else {
                WORD *oldt = 0;
                if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
                    oldt = t;
                    if ( ( i = AN.RepFunList[mt+1] ) > 0 ) {
                        *fill++ = GAMMA;
                        *fill++ = i + FUNHEAD+1;
                        FILLFUN(fill)
                        nq = i + 1;
                        t += FUNHEAD;
                        NCOPY(fill,t,nq);
                    }
                    t = oldt;
                }
                else if ( ( *t == LEVICIVITA ) || ( *t >= FUNCTION
                && (functions[*t-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
                                ) sign += AN.RepFunList[mt+1];
                else if ( *m >= FUNCTION+WILDOFFSET
                && (functions[*m-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER) == ANTISYMMETRIC
                                ) sign += AN.RepFunList[mt+1];
                if ( !PutExpr ) {
                    WORD *pstart = fill, *p, *w, *ww;
                    xstop = t + t[1];
                    t = AN.FullProto;
                    nq = t[1];
                    t[3] = 1;
                    NCOPY(fill,t,nq);
                    t = xstop;
                    PutExpr = 1;
/*
                    Here we need provisions for keeping wildcard matches
                    that reside in AT.ebufnum. We will move them to
                    AT.aebufnum.
                    Problem: the SUBEXPRESSION assumes automatically
                    that the compiler buffer is AT.ebufnum. We have to
                    correct that in TranferBuffer.
*/
                    p = pstart + SUBEXPSIZE;
                    while ( p < fill ) {
                        switch ( *p ) {
                            case SYMTOSUB:
                            case VECTOSUB:
                            case INDTOSUB:
                            case ARGTOARG:
                            case ARLTOARL:
                                w = cbuf[AT.ebufnum].rhs[p[3]];
                                ww = cbuf[AT.ebufnum].rhs[p[3]+1];
/*
                                Here we could search for whether this
                                object sits in the buffer already.
                                To be done later.
                                By the way: ww-w fits inside a WORD.
*/
                                AddRHS(AT.aebufnum,1);
                                AddNtoC(AT.aebufnum,ww-w,w,11);
                                p[3] = cbuf[AT.aebufnum].numrhs;
```

```
                                cbuf[AT.aebufnum].rhs[p[3]+1] = cbuf[AT.aebufnum].Pointer;
                                p += p[1];
                                break;
                            case FROMSET:
                            case SETTONUM:
                            case LOADDOLLAR:
                                p += p[1];
                                break;
                            default:
                                p += p[1];
                                break;
                        }

                    }
                }
                else t += t[1];
                if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
                    i = oldt[1] - m[1] - i;
                    if ( i > 0 ) {
                        *fill++ = GAMMA;
                        *fill++ = i + FUNHEAD+1;
                        FILLFUN(fill)
                        *fill++ = oldt[FUNHEAD];
                        t = t - i;
                        NCOPY(fill,t,i);
                    }
                }
                break;
            }
        }
        m += m[1];
    }
    while ( t < tstop ) *fill++ = *t++;
    if ( !PutExpr ) {
        t = AN.FullProto;
        nq = t[1];
        t[3] = 1;
        NCOPY(fill,t,nq);
    }
    t = tcoef;
    nq = ABS(*t);
    t = tstop;
    NCOPY(fill,t,nq);
    if ( sign ) {
        if ( ( sign & 1 ) != 0 ) fill[-1] = -fill[-1];
    }
    *TemTerm = fill-TemTerm;
/*
    And now we copy this to AT.allbufnum
*/
    AddNtoC(AT.allbufnum,TemTerm[0],TemTerm,12);
    cbuf[AT.allbufnum].Pointer[0] = 0;
    AN.RepFunNum = 0;
}

/*
        #] SubsInAll :
        #[ TransferBuffer :

        Adds the whole content of a (compiler)buffer to another buffer.
        In spectator we have an expression in the RHS that needs the
        wildcard resolutions adapted by an offset.
*/

VOID TransferBuffer(int from,int to,int spectator)
{
    CBUF *C  = cbuf + spectator;
    CBUF *Cf = cbuf + from;
    CBUF *Ct = cbuf + to;
    int offset = Ct->numrhs;
    LONG i;
    WORD *t, *tt, *ttt, *tstop, size;
    for ( i = 1; i <= Cf->numrhs; i++ ) {
        size = Cf->rhs[i+1]-Cf->rhs[i];
        AddRHS(to,1);
        AddNtoC(to,size,Cf->rhs[i],13);
    }
    Ct->rhs[Ct->numrhs+1] = Ct->Pointer;
    Cf->numrhs = 0;
/*
    Now we have to update the 'pointers' in the spectator.
*/
    t = C->rhs[C->numrhs];
    while ( *t ) {
        tt = t+1; t += *t;
        tstop = t-ABS(t[-1]);
        while ( tt < tstop ) {
```

```
            if ( *tt == SUBEXPRESSION ) {
                ttt = tt+SUBEXPSIZE; tt += tt[1];
                while ( ttt < tt ) {
                    switch ( *ttt ) {
                        case SYMTOSUB:
                        case VECTOSUB:
                        case INDTOSUB:
                        case ARGTOARG:
                        case ARLTOARL:
                            ttt[3] += offset;
                            break;
                        default:
                            break;
                    }
                    ttt += 4;
                }
            }
            else tt += tt[1];
        }
    }
}

/*
        #] TransferBuffer :
        #[ TakeIDfunction :
*/

#define PutInBuffers(pow) \
    AddRHS(AT.ebufnum,1); \
    *out++ = SUBEXPRESSION; \
    *out++ = SUBEXPSIZE; \
    *out++ = C->numrhs; \
    *out++ = pow; \
    *out++ = AT.ebufnum; \
    FILLSUB(out) \
    r = AT.pWorkSpace[rhs+i]; \
    if ( *r > 0 ) { \
        oldinr = r[*r]; r[*r] = 0; \
        AddNtoC(AT.ebufnum,(*r+1-ARGHEAD),(r+ARGHEAD),14); \
        r[*r] = oldinr; \
    } \
    else { \
        ToGeneral(r,buffer,1); \
        buffer[buffer[0]] = 0; \
        AddNtoC(AT.ebufnum,buffer[0]+1,buffer,15); \
    }

int TakeIDfunction(PHEAD WORD *term)
{
    WORD *tstop, *t, *r, *m, *f, *nextf, *funstop, *left, *l, *newterm;
    WORD *out, oldinr, pow;
    WORD buffer[20];
    int i, ii, j, numsub, numfound = 0, first;
    LONG lhs,rhs;
    CBUF *C;
    GETSTOP(term,tstop);
    for ( t = term+1; t < tstop; t += t[1] ) { if ( *t == IDFUNCTION ) break; }
    if ( t >= tstop ) return(0);
/*
    Step 1: test validity
*/
    funstop = t + t[1]; f = t + FUNHEAD;
    left = term + *term;
    l = left+1; numsub = 0;
    while ( f < funstop ) {
        nextf = f; NEXTARG(nextf)
        if ( nextf >= funstop ) { return(0); } /* odd number of arguments */
        if ( *f == -SYMBOL ) { *l++ = SYMBOL; *l++ = 4; *l++ = f[1]; *l++ = 1; }
        else if ( *f < -FUNCTION ) { *l++ = *f; *l++ = FUNHEAD; FILLFUN(l) }
        else if ( *f > 0 ) {
            if ( *f != f[ARGHEAD]+ARGHEAD ) goto noaction;
            if ( nextf[-1] != 3 || nextf[-2] != 1 || nextf[-3] != 1 ) goto noaction;
            if ( f[ARGHEAD] <= 4 ) goto noaction;
            if ( f[ARGHEAD] != f[ARGHEAD+2]+4 ) goto noaction;
            if ( f[ARGHEAD] == 8 && f[ARGHEAD+1] == SYMBOL ) {
                for ( i = 0; i < 4; i++ ) *l++ = f[ARGHEAD+1+i];
            }
            else if ( f[ARGHEAD] == 9 && f[ARGHEAD+1] == DOTPRODUCT ) {
                for ( i = 0; i < 5; i++ ) *l++ = f[ARGHEAD+1+i];
            }
            else if ( f[ARGHEAD+1] >= FUNCTION ) {
                for ( i = 0; i < f[ARGHEAD+1]-4; i++ ) *l++ = f[ARGHEAD+1+i];
            }
            else goto noaction;
        }
        else goto noaction;
        numsub++;
```

```
        f = nextf;
        NEXTARG(f)
    }
    C = cbuf+AT.ebufnum;
    AT.WorkPointer = l;
    *left = l-left;
/*
    Put the pointers to the lhs and the rhs in the pointer workspace
*/
    WantAddPointers(2*numsub);
    lhs = AT.pWorkPointer;
    rhs = lhs+numsub;
    AT.pWorkPointer = rhs+numsub;
    f = t + FUNHEAD; l = left+1;
    for ( i = 0; i < numsub; i++ ) {
        AT.pWorkSpace[lhs+i] = l; l += l[1];
        NEXTARG(f);
        AT.pWorkSpace[rhs+i] = f;
        NEXTARG(f);
    }
/*
    Take out the patterns and replace them by SUBEXPRESSIONs pointing at
    the e buffer. We put the resulting term above the left sides.
    Note that we take out only the first id_ if there is more than one!
*/
    first = 1;
    t = term+1; newterm = AT.WorkPointer; out = newterm+1;
    while ( t < tstop ) {
        if ( *t == IDFUNCTION && first ) { first = 0; t += t[1]; continue; }
        if ( *t >= FUNCTION ) {
            for ( i = 0; i < numsub; i++ ) {
                m = AT.pWorkSpace[lhs+i];
                if ( *m != *t ) continue;
                for ( j = 1; j < t[1]; j++ ) {
                    if ( m[j] != t[j] ) break;
                }
                if ( j != t[1] ) continue;
                numfound++;
/*
                We have a match! Set up a SUBEXPRESSION subterm and put the
                corresponding rhs in the eBuffer.
*/
                PutInBuffers(1)
                t += t[1];
            }
            if ( i == numsub ) {    /* no match. Just copy to output. */
                j = t[1]; NCOPY(out,t,j)
            }
        }
        else if ( *t == SYMBOL ) {
            for ( i = 0; i < numsub; i++ ) {
                m = AT.pWorkSpace[lhs+i];
                if ( *m != SYMBOL ) continue;
                for ( ii = 2; ii < t[1]; ii += 2 ) {
                    if ( m[2] != t[ii] ) continue;
                    pow = t[ii+1]/m[3];
                    if ( pow <= 0 ) continue;
                    t[ii+1] = t[ii+1]%m[3];
                    numfound++;
/*
                    Create the proper rhs in the eBuffer and set up a
                    SUBEXPRESSION subterm.
*/
                    PutInBuffers(pow)
                }
            }
/*
            Now we copy whatever remains of the SYMBOL subterm to the output
*/
            m = out; *out++ = t[0]; *out++ = t[1];
            for ( ii = 2; ii < t[1]; ii += 2 ) {
                if ( t[ii+1] ) { *out++ = t[ii]; *out++ = t[ii+1]; }
            }
            m[1] = out-m;
            if ( m[1] == 2 ) out = m;
            t += t[1];
        }
        else if ( *t == DOTPRODUCT ) {
            for ( i = 0; i < numsub; i++ ) {
                m = AT.pWorkSpace[lhs+i];
                if ( *m != DOTPRODUCT ) continue;
                for ( ii = 2; ii < t[1]; ii += 3 ) {
                    if ( m[2] != t[ii] || m[3] != t[ii+1] ) continue;
                    pow = t[ii+2]/m[4];
                    if ( pow <= 0 ) continue;
                    t[ii+2] = t[ii+2]%m[4];
                    numfound++;
```

```
/*
                    Create the proper rhs in the eBuffer and set up a
                    SUBEXPRESSION subterm.
*/
                    PutInBuffers(pow)
                }
            }
/*
            Now we copy whatever remains of the DOTPRODUCT subterm to the output
*/
            m = out; *out++ = t[0]; *out++ = t[1];
            for ( ii = 2; ii < t[1]; ii += 3 ) {
                if ( t[ii+2] ) { *out++ = t[ii]; *out++ = t[ii+1]; *out++ = t[ii+2]; }
            }
            m[1] = out-m;
            if ( m[1] == 2 ) out = m;
            t += t[1];
        }
        else {
            j = t[1]; NCOPY(out,t,j)
        }
    }
/*
    Copy the coefficient and set the size.
*/
    t = tstop; r = term+*term; while ( t < r ) *out++ = *t++;
    *newterm = out-newterm;
/*
    Finally we move the new term over the original term.
*/
    i = *newterm;
    t = term; r = newterm; NCOPY(t,r,i)
/*
    At this point we can return and if the calling Generator jumps back to
    its start, TestSub can take care of the expansions of SUBEXPRESSIONs.
*/
    AT.pWorkPointer = lhs;
    AT.WorkPointer = t;
    return(numfound);
noaction:
    return(0);
}

/*
        #] TakeIDfunction :
    #] Patterns :
*/

```

4.79 poly.h

```
#pragma once
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

extern "C" {
#include "form3.h"
}

```

```
#include <string>
#include <vector>

// macros for tform
#ifndef WITHPTHREADS
#define POLY_GETIDENTITY(X)
#define POLY_STOREIDENTITY
#else
#define POLY_GETIDENTITY(X) ALLPRIVATES *B = (X).Bpointer
#define POLY_STOREIDENTITY Bpointer = B
#endif

// maximum size of the hash table used for multiplication and division

const int POLY_MAX_HASH_SIZE = MiN(1«20, MAXPOSITIVE);

class poly {

public:

    // variables
#ifdef WITHPTHREADS
    ALLPRIVATES *Bpointer;
#endif

    WORD *terms;
    LONG size_of_terms;
    WORD modp,modn;

    // constructors/destructor
    poly (PHEAD int, WORD=-1, WORD=1);
    poly (PHEAD const UWORD *, WORD, WORD=-1, WORD=1);
    poly (const poly &, WORD=-1, WORD=1);
    ~poly ();

    // operators
    poly& operator+= (const poly&);
    poly& operator-= (const poly&);
    poly& operator*= (const poly&);
    poly& operator/= (const poly&);
    poly& operator%= (const poly&);

    const poly operator+ (const poly&) const;
    const poly operator- (const poly&) const;
    const poly operator* (const poly&) const;
    const poly operator/ (const poly&) const;
    const poly operator% (const poly&) const;

    bool operator== (const poly&) const;
    bool operator!= (const poly&) const;
    poly& operator= (const poly &);
    WORD& operator[] (int);
    const WORD& operator[] (int) const;

    // memory management
    void termscopy (const WORD *, int, int);
    void check_memory(int);
    void expand_memory(int);

    // check type of polynomial
    bool is_zero () const;
    bool is_one () const;
    bool is_integer () const;
    bool is_monomial () const;
    int is_dense_univariate () const;

    // properties
    int sign () const;
    int degree (int) const;
    int total_degree () const;
    int first_variable () const;
    int number_of_terms () const;
    const std::vector<int> all_variables () const;
    const poly integer_lcoeff () const;
    const poly lcoeff_univar (int) const;
    const poly lcoeff_multivar (int) const;
    const poly coefficient (int, int) const;
    const poly derivative (int) const;

    // modulo calculus
    void setmod(WORD, WORD=1);
    void coefficients_modulo (UWORD *, WORD, bool);

    // simple polynomials
    static const poly simple_poly (PHEAD int, int=0, int=1, int=0, int=1);
    static const poly simple_poly (PHEAD int, const poly&, int=1, int=0, int=1);

```

```
    // conversion from/to form notation
    static void get_variables (PHEAD std::vector<WORD *>, bool, bool);
    static const poly argument_to_poly (PHEAD WORD *, bool, bool, poly *den=NULL);
    static void poly_to_argument (const poly &, WORD *, bool);
    static void poly_to_argument_with_den (const poly &, WORD, const UWORD *, WORD *, bool);
    int size_of_form_notation () const;
    int size_of_form_notation_with_den (WORD) const;
    const poly & normalize ();

    // operations for coefficient lists
    static const poly from_coefficient_list (PHEAD const std::vector<WORD> &, int, WORD);
    static const std::vector<WORD> to_coefficient_list (const poly &);
    static const std::vector<WORD> coefficient_list_divmod (const std::vector<WORD> &, const std::vector<WORD> &, WORD, int);

    // string output for debugging
    const std::string to_string() const;

    // monomials
    static int monomial_compare (PHEAD const WORD *, const WORD *);
    WORD last_monomial_index () const;
    WORD* last_monomial () const;
    int compare_degree_vector(const poly &) const;
    std::vector<int> degree_vector() const;
    int compare_degree_vector(const std::vector<int> &) const;

    // (internal) mathematical operations
    static void add (const poly &, const poly &, poly &);
    static void sub (const poly &, const poly &, poly &);
    static void mul (const poly &, const poly &, poly &);
    static void div (const poly &, const poly &, poly &);
    static void mod (const poly &, const poly &, poly &);
    static void divmod (const poly &, const poly &, poly &, poly &, bool only_divides);
    static bool divides (const poly &, const poly &);

    static void mul_one_term (const poly &, const poly &, poly &);
    static void mul_univar (const poly &, const poly &, poly &, int);
    static void mul_heap (const poly &, const poly &, poly &);

    static void divmod_one_term (const poly &, const poly &, poly &, poly &, bool);
    static void divmod_univar (const poly &, const poly &, poly &, poly &, int, bool);
    static void divmod_heap (const poly &, const poly &, poly &, poly &, bool);

    static void push_heap (PHEAD WORD **, int);
    static void pop_heap (PHEAD WORD **, int);

    PADPOINTER(1,0,2,0);
};

// comparison class for monomials (for std::sort)
class monomial_larger {

public:

#ifndef WITHPTHREADS
    monomial_larger() {}
#else
    ALLPRIVATES *B;
    monomial_larger(ALLPRIVATES *b): B(b) {}
#endif

    bool operator()(const WORD *a, const WORD *b) {
        return poly::monomial_compare(BHEAD a, b) > 0;
    }
};

// stream operator
std::ostream& operator« (std::ostream &, const poly &);

// inline function definitions

/*   Checks whether the terms array is large enough to add another
 *   term (of size AM.MaxTal) to the polynomials. In case not, it is
 *   expanded.
 */
inline void poly::check_memory (int i) {
    POLY_GETIDENTITY(*this);
    if (i + 3 + AN.poly_num_vars + AM.MaxTal >= size_of_terms) expand_memory(i + AM.MaxTal);
//  Used to be i+2 but there should also be space for a trailing zero
}

// indexing operators
inline WORD& poly::operator[] (int i) {
    return terms[i];
}

inline const WORD& poly::operator[] (int i) const {
```

```
    return terms[i];
}

/*   Copies "num" WORD-sized terms from the pointer "source" to the
 *   current polynomial at index "dest"
 */
inline void poly::termscopy (const WORD *source, int dest, int num) {
    memcpy (terms+dest, source, num*sizeof(WORD));
}
```

4.80 polyfact.h

```
#pragma once
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

#include <vector>
#include <string>

// First prime modulo which factorization is tried. Too small results
// in more unsuccesful attempts; too large is slower.
const int POLYFACT_FIRST_PRIME = 17;

// Fraction of [1,p) that is used for substitutions of variables. Too
// small results in more unsuccesful attempts; too large is slower.
const int POLYFACT_IDEAL_FRACTION = 5;

// Number of ideals that are tried before failure due to unlucky
// choices is accepted.
const int POLYFACT_MAX_IDEAL_TRIES = 3;

// Number of confirmations for the minimal number of factors before
// Hensel lifting is started.
const int POLYFACT_NUM_CONFIRMATIONS = 3;

// Maximum number of equations for predetermination of coefficients
// for multivariate Hensel lifting
const int POLYFACT_MAX_PREDETERMINATION = 10000;

class poly;

class factorized_poly {
    /*   Class for representing a factorized polynomial
     *   The polynomial is given by: PRODUCT(factor[i] ^ power[i])
     */
public:
    std::vector<poly> factor;
    std::vector<int> power;

    void add_factor(const poly &f, int p=1);
    const std::string tostring () const;
    friend std::ostream& operator« (std::ostream &out, const poly &p);
};

std::ostream& operator« (std::ostream &out, const factorized_poly &a);

namespace polyfact {

    // factorization routine
```

```
    const factorized_poly factorize (const poly &a);

    // methods for squarefree factorization
    const factorized_poly squarefree_factors (const poly &a);
    const factorized_poly squarefree_factors_Yun (const poly &a);
    const factorized_poly squarefree_factors_modp (const poly &a);

    // methods for choosing suitable reductions
    const std::vector<poly> factorize_squarefree (const poly &a, const std::vector<int> &x);
    WORD choose_prime (const poly &a, const std::vector<int> &x, WORD p=0);
    WORD choose_prime_power (const poly &a, WORD p);
    const std::vector<int> choose_ideal (const poly &a, int p, const factorized_poly &lc, const std::vector<int> &x);

    // methods for univariate factorization
    const std::vector<std::vector<WORD> > Berlekamp_Qmatrix (const poly &a);
    const std::vector<poly> Berlekamp_find_factors (const poly &a, const std::vector<std::vector<WORD> > &Q);
    const std::vector<poly> combine_factors (const poly &a, const std::vector<poly> &f);

    // methods for Hensel lifting
    const std::vector<poly> extended_gcd_Euclidean_lifted (const poly &a, const poly &b);
    const std::vector<poly> solve_Diophantine_univariate (const std::vector<poly> &a, const poly &b);
    const std::vector<poly> solve_Diophantine_multivariate (const std::vector<poly> &a, const poly &b, const std::vector<int> &x, const std::vector<int> &c, int d);
    const std::vector<poly> lift_coefficients (const poly &a, const std::vector<poly> &f);
    const std::vector<poly> lift_variables (const poly &a, const std::vector<poly> &f, const std::vector<int> &x, const std::vector<int> &c, const std::vector<poly> &lc);
    void predetermine (int dep, const std::vector<std::vector<int> > &state, std::vector<std::vector<std::vector<int> > > &terms, std::vector<int> &term, int sumdeg=0);

}
```

4.81 polygcd.h

```
#pragma once
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

#include <vector>
#include <map>

class poly; // polynomial class
class gcd_heuristic_failed {}; // class for throwing exceptions

// whether or not to use the heuristic before Zippel's algorithm
#define POLYGCD_USE_HEURISTIC

// maximum number of words in a coefficient for gcd_heuristic to continue
const int POLYGCD_HEURISTIC_MAX_DIGITS = 1000;

// maximum number of retries after the heuristic has failed
const int POLYGCD_HEURISTIC_MAX_TRIES = 10;

// a fudge factor, which improves efficiency
const int POLYGCD_HEURISTIC_MAX_ADD_RANDOM = 10;

// maximum cached power in substitute_last and sparse_interpolation_get_mul_list
const int POLYGCD_RAISPOWMOD_CACHE_SIZE = 1000;
```

```
namespace polygcd {

    // functions to call the gcd routines
    const poly integer_gcd (const poly &a, const poly &b);
    const poly integer_content (const poly &a);

    const poly gcd (const poly &a, const poly &b);
    const poly content_univar (const poly &a, int x);
    const poly content_multivar (const poly &a, int x);

    const std::vector<WORD> coefficient_list_gcd (const std::vector<WORD> &a, const std::vector<WORD> &b, WORD p);

    // internal functions
    const poly gcd_heuristic (const poly &a, const poly &b, const std::vector<int> &x, int max_tries=POLYGCD_HEURISTIC_MAX_TRIES);
    const poly gcd_Euclidean (const poly &a, const poly &b);
    const poly gcd_modular (const poly &a, const poly &b, const std::vector<int> &x);
    const poly gcd_modular_dense_interpolation (const poly &a, const poly &b, const std::vector<int> &x, const poly &s);
    const poly gcd_modular_sparse_interpolation (const poly &a, const poly &b, const std::vector<int> &x, const poly &s);

    const std::vector<int> sparse_interpolation_get_mul_list (const poly &a, const std::vector<int> &x, const std::vector<int> &c);
    void sparse_interpolation_mul_poly (poly &a, const std::vector<int> &m);
    const poly sparse_interpolation_reduce_poly (const poly &a, const std::vector<int> &x);
    const poly sparse_interpolation_fix_poly (const poly &a, int x);

    const poly chinese_remainder (const poly &a1, const poly &m1, const poly &a2, const poly &m2);
    const poly substitute(const poly &a, int x, int c);
    const std::map<std::vector<int>,poly> full_bracket(const poly &a, const std::vector<int>& filter);
    const poly bracket(const poly &a, const std::vector<int>& pattern, const std::vector<int>& filter);
    const std::map<std::vector<int>,int> bracket_count(const poly &a, const std::vector<int>& filter);
    const poly gcd_linear (const poly &a, const poly &b);
}
```

4.82 portsignals.h File Reference

`#include <signal.h>`

Include dependency graph for portsignals.h:

Macros

- #define FATAL_SIG_ERROR 4
- #define NSIG (1024)
- #define SIGSEGV (NSIG+1)
- #define SIGFPE (NSIG+2)
- #define SIGILL (NSIG+3)

- #define SIGEMT (NSIG+4)
- #define SIGSYS (NSIG+5)
- #define SIGPIPE (NSIG+6)
- #define SIGLOST (NSIG+7)
- #define SIGXCPU (NSIG+8)
- #define SIGXFSZ (NSIG+9)
- #define SIGTERM (NSIG+10)
- #define SIGINT (NSIG+11)
- #define SIGQUIT (NSIG+12)
- #define SIGHUP (NSIG+13)
- #define SIGALRM (NSIG+14)
- #define SIGVTALRM (NSIG+15)
- #define SIGPROF (NSIG+16)

4.82.1 Detailed Description

Contains definitions for signals used/intercepted in FORM.

Some systems (especially LINUX) have not enough signals available so some of the (!documented!) signals are not defined. This file contains the definition of all signals used in the program. If the signal is not defined we define it as unused (>NSIG).

The include of signal.h must be first, before we try to define undefined signals.

Definition in file portsignals.h.

4.82.2 Macro Definition Documentation

4.82.2.1 FATAL_SIG_ERROR

```
#define FATAL_SIG_ERROR 4
```

Definition at line 47 of file portsignals.h.

4.82.2.2 NSIG

```
#define NSIG (1024)
```

Definition at line 53 of file portsignals.h.

4.82.2.3 SIGSEGV

```
#define SIGSEGV (NSIG+1)
```

Definition at line 57 of file portsignals.h.

4.82.2.4 SIGFPE

```
#define SIGFPE (NSIG+2)
```

Definition at line 60 of file portsignals.h.

4.82.2.5 SIGILL

```
#define SIGILL (NSIG+3)
```

Definition at line 63 of file portsignals.h.

4.82.2.6 SIGEMT

```
#define SIGEMT (NSIG+4)
```

Definition at line 66 of file portsignals.h.

4.82.2.7 SIGSYS

```
#define SIGSYS (NSIG+5)
```

Definition at line 69 of file portsignals.h.

4.82.2.8 SIGPIPE

```
#define SIGPIPE (NSIG+6)
```

Definition at line 72 of file portsignals.h.

4.82.2.9 SIGLOST

```
#define SIGLOST (NSIG+7)
```

Definition at line 75 of file portsignals.h.

4.82.2.10 SIGXCPU

```
#define SIGXCPU (NSIG+8)
```

Definition at line 78 of file portsignals.h.

4.82.2.11 SIGXFSZ

```
#define SIGXFSZ (NSIG+9)
```

Definition at line 81 of file portsignals.h.

4.82.2.12 SIGTERM

```
#define SIGTERM (NSIG+10)
```

Definition at line 84 of file portsignals.h.

4.82.2.13 SIGINT

```
#define SIGINT (NSIG+11)
```

Definition at line 87 of file portsignals.h.

4.82.2.14 SIGQUIT

```
#define SIGQUIT (NSIG+12)
```

Definition at line 90 of file portsignals.h.

4.82.2.15 SIGHUP

```
#define SIGHUP (NSIG+13)
```

Definition at line 93 of file portsignals.h.

4.82.2.16 SIGALRM

```
#define SIGALRM (NSIG+14)
```

Definition at line 96 of file portsignals.h.

4.82.2.17 SIGVTALRM

```
#define SIGVTALRM (NSIG+15)
```

Definition at line 99 of file portsignals.h.

4.82.2.18 SIGPROF

```
#define SIGPROF (NSIG+16)
```

Definition at line 102 of file portsignals.h.

4.83 portsignals.h

Go to the documentation of this file.

```
#ifndef PORTSIGNAL_H
#define PORTSIGNAL_H

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

#include <signal.h>

#define FATAL_SIG_ERROR 4

#ifndef NSIG
/*
    The value of NSIG must be enough to fall outside the range of defined signals
*/
#define NSIG (1024)
#endif

#ifndef SIGSEGV
#define SIGSEGV (NSIG+1)
#endif
#ifndef SIGFPE
#define SIGFPE (NSIG+2)
#endif
#ifndef SIGILL
#define SIGILL (NSIG+3)
#endif
#ifndef SIGEMT
#define SIGEMT (NSIG+4)
#endif
#ifndef SIGSYS
#define SIGSYS (NSIG+5)
#endif
#ifndef SIGPIPE
#define SIGPIPE  (NSIG+6)
#endif
#ifndef SIGLOST
#define SIGLOST (NSIG+7)
#endif
#ifndef SIGXCPU
#define SIGXCPU  (NSIG+8)
#endif
#ifndef SIGXFSZ
#define SIGXFSZ (NSIG+9)
#endif
#ifndef SIGTERM
#define SIGTERM (NSIG+10)
#endif
#ifndef SIGINT
#define SIGINT (NSIG+11)
#endif
#ifndef SIGQUIT
#define SIGQUIT (NSIG+12)
#endif
#ifndef SIGHUP
#define SIGHUP (NSIG+13)
#endif
#ifndef SIGALRM
#define SIGALRM (NSIG+14)
```

```
#endif
#ifndef SIGVTALRM
#define SIGVTALRM (NSIG+15)
#endif
#ifndef SIGPROF
#define SIGPROF (NSIG+16)
#endif

#endif
```

4.84 pre.c File Reference

```
#include "form3.h"
#include "vector.h"
```

Include dependency graph for pre.c:

Macros

- #define SKIPBUFSIZE 20
- #define KILL "kill"
- #define KILLALL "killall"
- #define DAEMON "daemon"
- #define SHELL "shell"
- #define STDERR "stderr"
- #define TRUE_EXPR "true"
- #define FALSE_EXPR "false"
- #define NOSHELL "noshell"
- #define TERMINAL "terminal"

Functions

- UBYTE GetInput ()
- VOID ClearPushback ()
- UBYTE GetChar (int level)
- VOID CharOut (UBYTE c)
- VOID UnsetAllowDelay ()
- UBYTE * GetPreVar (UBYTE *name, int flag)
- int PutPreVar (UBYTE *name, UBYTE *value, UBYTE *args, int mode)
- VOID PopPreVars (int tonumber)
- VOID IniModule (int type)
- VOID IniSpecialModule (int type)
- VOID PreProcessor ()

- int PreProInstruction ()
- int LoadInstruction (int mode)
- int LoadStatement (int type)
- int ExpandTripleDots (int par)
- KEYWORD * FindKeyWord (UBYTE *theword, KEYWORD *table, int size)
- KEYWORD * FindInKeyWord (UBYTE *theword, KEYWORD *table, int size)
- int TheDefine (UBYTE *s, int mode)
- int DoCommentChar (UBYTE *s)
- int DoPreAssign (UBYTE *s)
- int DoDefine (UBYTE *s)
- int DoRedefine (UBYTE *s)
- int ClearMacro (UBYTE *name)
- int TheUndefine (UBYTE *name)
- int DoUndefine (UBYTE *s)
- int DoInclude (UBYTE *s)
- int DoReverseInclude (UBYTE *s)
- int Include (UBYTE *s, int type)
- int DoPreExchange (UBYTE *s)
- int DoCall (UBYTE *s)
- int DoDebug (UBYTE *s)
- int DoTerminate (UBYTE *s)
- int DoDo (UBYTE *s)
- int DoBreakDo (UBYTE *s)
- int DoElse (UBYTE *s)
- int DoElseif (UBYTE *s)
- int DoEnddo (UBYTE *s)
- int DoEndif (UBYTE *s)
- int DoEndprocedure (UBYTE *s)
- int DoIf (UBYTE *s)
- int DoIfdef (UBYTE *s, int par)
- int DoInside (UBYTE *s)
- int DoEndInside (UBYTE *s)
- int DoMessage (UBYTE *s)
- int DoPipe (UBYTE *s)
- int DoPrcExtension (UBYTE *s)
- int DoPreOut (UBYTE *s)
- int DoPrePrintTimes (UBYTE *s)
- int DoPreAppend (UBYTE *s)
- int DoPreCreate (UBYTE *s)
- int DoPreRemove (UBYTE *s)
- int DoPreClose (UBYTE *s)
- int DoPreWrite (UBYTE *s)
- int DoProcedure (UBYTE *s)
- int DoPreBreak (UBYTE *s)
- int DoPreCase (UBYTE *s)
- int DoPreDefault (UBYTE *s)
- int DoPreEndSwitch (UBYTE *s)
- int DoPreSwitch (UBYTE *s)
- int DoPreShow (UBYTE *s)
- int DoSystem (UBYTE *s)
- int PreLoad (PRELOAD *p, UBYTE *start, UBYTE *stop, int mode, char *message)
- int PreSkip (UBYTE *start, UBYTE *stop, int mode)
- VOID StartPrepro ()
- int EvalPreIf (UBYTE *s)
- UBYTE * PreIfEval (UBYTE *s, int *value)

- int PreCmp (int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int cmpop)
- int PreEq (int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int eqop)
- UBYTE * pParseObject (UBYTE *s, int *type, LONG *val2)
- UBYTE * PreCalc ()
- UBYTE * PreEval (UBYTE *s, LONG *x)
- void AddToPreTypes (int type)
- void MessPreNesting (int par)
- int DoPreAddSeparator (UBYTE *s)
- int DoPreRmSeparator (UBYTE *s)
- int DoExternal (UBYTE *s)
- int DoPrompt (UBYTE *s)
- int DoSetExternal (UBYTE *s)
- int DoSetExternalAttr (UBYTE *s)
- int DoRmExternal (UBYTE *s)
- int DoFromExternal (UBYTE *s)
- int DoToExternal (UBYTE *s)
- UBYTE * defineChannel (UBYTE *s, HANDLERS *h)
- int writeToChannel (int wtype, UBYTE *s, HANDLERS *h)
- int DoFactDollar (UBYTE *s)
- WORD GetDollarNumber (UBYTE **inp, DOLLARS d)
- int DoSetRandom (UBYTE *s)
- int DoOptimize (UBYTE *s)
- int DoClearOptimize (UBYTE *s)
- int DoSkipExtraSymbols (UBYTE *s)
- int DoPreReset (UBYTE *s)
- int DoPreAppendPath (UBYTE *s)
- int DoPrePrependPath (UBYTE *s)

4.84.1 Detailed Description

This is the preprocessor and all its routines.

Definition in file pre.c.

4.84.2 Macro Definition Documentation

4.84.2.1 SKIPBUFSIZE

```
#define SKIPBUFSIZE 20
```

Definition at line 4207 of file pre.c.

4.84.2.2 KILL

```
#define KILL "kill"
```

Definition at line 5331 of file pre.c.

4.84.2.3 KILLALL

```
#define KILLALL "killall"
```

Definition at line 5332 of file pre.c.

4.84.2.4 DAEMON

```
#define DAEMON "daemon"
```

Definition at line 5333 of file pre.c.

4.84.2.5 SHELL

```
#define SHELL "shell"
```

Definition at line 5334 of file pre.c.

4.84.2.6 STDERR

```
#define STDERR "stderr"
```

Definition at line 5335 of file pre.c.

4.84.2.7 TRUE_EXPR

```
#define TRUE_EXPR "true"
```

Definition at line 5337 of file pre.c.

4.84.2.8 FALSE_EXPR

```
#define FALSE_EXPR "false"
```

Definition at line 5338 of file pre.c.

4.84.2.9 NOSHELL

```
#define NOSHELL "noshell"
```

Definition at line 5339 of file pre.c.

4.84.2.10 TERMINAL

```
#define TERMINAL "terminal"
```

Definition at line 5340 of file pre.c.

4.84.3 Function Documentation

4.84.3.1 GetInput()

```
UBYTE GetInput ( )
```

Definition at line 118 of file pre.c.

4.84.3.2 ClearPushback()

```
VOID ClearPushback ( )
```

Definition at line 147 of file pre.c.

4.84.3.3 GetChar()

```
UBYTE GetChar (
            int level )
```

Definition at line 165 of file pre.c.

4.84.3.4 CharOut()

```
VOID CharOut (
            UBYTE c )
```

Definition at line 409 of file pre.c.

4.84.3.5 UnsetAllowDelay()

```
VOID UnsetAllowDelay ( )
```

Definition at line 430 of file pre.c.

4.84.3.6 GetPreVar()

```
UBYTE * GetPreVar (
            UBYTE * name,
            int flag )
```

Definition at line 456 of file pre.c.

4.84.3.7 PutPreVar()

```
int PutPreVar (
            UBYTE * name,
            UBYTE * value,
            UBYTE * args,
            int mode )
```

Inserts/Updates a preprocessor variable in the name administration.

Parameters

name	Character string with the variable name.
value	Character string with a possible value. Special case: if this argument is zero, then we have no value. Note: This is different from having an empty argument! This should only occur when the name starts with a ?
args	Character string with possible arguments.
mode	=0: always create a new name entry, =1: try to do a redefinition if possible.

Returns

Index of used entry in name list.

Definition at line 638 of file pre.c.

References pReVaR::argnames, pReVaR::name, pReVaR::nargs, pReVaR::value, and pReVaR::wildarg.

Referenced by Generator(), PF_BroadcastRedefinedPreVars(), StartVariables(), and TheDefine().

4.84.3.8 PopPreVars()

```
VOID PopPreVars (
            int tonumber )
```

Definition at line 739 of file pre.c.

4.84.3.9 IniModule()

```
VOID IniModule (
            int type )
```

Definition at line 754 of file pre.c.

4.84.3.10 IniSpecialModule()

```
VOID IniSpecialModule (
            int type )
```

Definition at line 856 of file pre.c.

4.84.3.11 PreProcessor()

```
VOID PreProcessor ( )
```

Definition at line 866 of file pre.c.

4.84.3.12 PreProInstruction()

```
int PreProInstruction ( )
```

Definition at line 1087 of file pre.c.

4.84.3.13 LoadInstruction()

```
int LoadInstruction (
            int mode )
```

Definition at line 1176 of file pre.c.

4.84.3.14 LoadStatement()

```
int LoadStatement (
            int type )
```

Definition at line 1374 of file pre.c.

4.84.3.15 ExpandTripleDots()

```
int ExpandTripleDots (
            int par )
```

Definition at line 1509 of file pre.c.

4.84.3.16 FindKeyWord()

```
KEYWORD * FindKeyWord (
            UBYTE * theword,
            KEYWORD * table,
            int size )
```

Definition at line 1877 of file pre.c.

4.84.3.17 FindInKeyWord()

```
KEYWORD * FindInKeyWord (
            UBYTE * theword,
            KEYWORD * table,
            int size )
```

Definition at line 1908 of file pre.c.

4.84.3.18 TheDefine()

```
int TheDefine (
            UBYTE * s,
            int mode )
```

Preprocessor assignment. Possible arguments and values are treated and the new preprocessor variable is put into the name administration.

Parameters

s	Pointer to the character string following the preprocessor command.
mode	Bitmask. 0-bit clear: always create a new name entry, 0-bit set: try to redefine an existing name, 1-bit set: ignore preprocessor if/switch status.

Returns

zero: no errors, negative number: errors.

Definition at line 1938 of file pre.c.

References PutPreVar().

Here is the call graph for this function:

4.84.3.19 DoCommentChar()

```
int DoCommentChar (
            UBYTE * s )
```

Definition at line 2011 of file pre.c.

4.84.3.20 DoPreAssign()

```
int DoPreAssign (
            UBYTE * s )
```

Definition at line 2042 of file pre.c.

4.84.3.21 DoDefine()

```
int DoDefine (
            UBYTE * s )
```

Definition at line 2073 of file pre.c.

4.84.3.22 DoRedefine()

```
int DoRedefine (
            UBYTE * s )
```

Definition at line 2083 of file pre.c.

4.84.3.23 ClearMacro()

```
int ClearMacro (
            UBYTE * name )
```

Definition at line 2095 of file pre.c.

4.84.3.24 TheUndefine()

```
int TheUndefine (
            UBYTE * name )
```

Definition at line 2123 of file pre.c.

4.84.3.25 DoUndefine()

```
int DoUndefine (
            UBYTE * s )
```

Definition at line 2177 of file pre.c.

4.84.3.26 DoInclude()

```
int DoInclude (
            UBYTE * s )
```

Definition at line 2229 of file pre.c.

4.84.3.27 DoReverseInclude()

```
int DoReverseInclude (
            UBYTE * s )
```

Definition at line 2236 of file pre.c.

4.84.3.28 Include()

```
int Include (
            UBYTE * s,
            int type )
```

Definition at line 2243 of file pre.c.

4.84.3.29 DoPreExchange()

```
int DoPreExchange (
            UBYTE * s )
```

Definition at line 2404 of file pre.c.

4.84.3.30 DoCall()

```
int DoCall (
            UBYTE * s )
```

Definition at line 2465 of file pre.c.

4.84.3.31 DoDebug()

```
int DoDebug (
            UBYTE * s )
```

Definition at line 2634 of file pre.c.

4.84.3.32 DoTerminate()

```
int DoTerminate (
            UBYTE * s )
```

Definition at line 2671 of file pre.c.

4.84.3.33 DoDo()

```
int DoDo (
            UBYTE * s )
```

Definition at line 2700 of file pre.c.

4.84.3.34 DoBreakDo()

```
int DoBreakDo (
            UBYTE * s )
```

Definition at line 2914 of file pre.c.

4.84.3.35 DoElse()

```
int DoElse (
            UBYTE * s )
```

Definition at line 2983 of file pre.c.

4.84.3.36 DoElseif()

```
int DoElseif (
            UBYTE * s )
```

Definition at line 3020 of file pre.c.

4.84.3.37 DoEnddo()

```
int DoEnddo (
            UBYTE * s )
```

Definition at line 3055 of file pre.c.

4.84.3.38 DoEndif()

```
int DoEndif (
            UBYTE * s )
```

Definition at line 3201 of file pre.c.

4.84.3.39 DoEndprocedure()

```
int DoEndprocedure (
            UBYTE * s )
```

Definition at line 3229 of file pre.c.

4.84.3.40 DoIf()

```
int DoIf (
            UBYTE * s )
```

Definition at line 3255 of file pre.c.

4.84.3.41 DoIfdef()

```
int DoIfdef (
            UBYTE * s,
            int par )
```

Definition at line 3279 of file pre.c.

4.84.3.42 DoInside()

```
int DoInside (
            UBYTE * s )
```

Definition at line 3319 of file pre.c.

4.84.3.43 DoEndInside()

```
int DoEndInside (
            UBYTE * s )
```

Definition at line 3412 of file pre.c.

4.84.3.44 DoMessage()

```
int DoMessage (
            UBYTE * s )
```

Definition at line 3544 of file pre.c.

4.84.3.45 DoPipe()

```
int DoPipe (
            UBYTE * s )
```

Definition at line 3558 of file pre.c.

4.84.3.46 DoPrcExtension()

```
int DoPrcExtension (
            UBYTE * s )
```

Definition at line 3581 of file pre.c.

4.84.3.47 DoPreOut()

```
int DoPreOut (
            UBYTE * s )
```

Definition at line 3615 of file pre.c.

4.84.3.48 DoPrePrintTimes()

```
int DoPrePrintTimes (
            UBYTE * s )
```

Definition at line 3638 of file pre.c.

4.84.3.49 DoPreAppend()

```
int DoPreAppend (
            UBYTE * s )
```

Definition at line 3655 of file pre.c.

4.84.3.50 DoPreCreate()

```
int DoPreCreate (
            UBYTE * s )
```

Definition at line 3698 of file pre.c.

4.84.3.51 DoPreRemove()

```
int DoPreRemove (
            UBYTE * s )
```

Definition at line 3738 of file pre.c.

4.84.3.52 DoPreClose()

```
int DoPreClose (
            UBYTE * s )
```

Definition at line 3771 of file pre.c.

4.84.3.53 DoPreWrite()

```
int DoPreWrite (
            UBYTE * s )
```

Definition at line 3815 of file pre.c.

4.84.3.54 DoProcedure()

```
int DoProcedure (
            UBYTE * s )
```

Definition at line 3856 of file pre.c.

4.84.3.55 DoPreBreak()

```
int DoPreBreak (
            UBYTE * s )
```

Definition at line 3898 of file pre.c.

4.84.3.56 DoPreCase()

```
int DoPreCase (
            UBYTE * s )
```

Definition at line 3922 of file pre.c.

4.84.3.57 DoPreDefault()

```
int DoPreDefault (
            UBYTE * s )
```

Definition at line 3961 of file pre.c.

4.84.3.58 DoPreEndSwitch()

```
int DoPreEndSwitch (
            UBYTE * s )
```

Definition at line 3985 of file pre.c.

4.84.3.59 DoPreSwitch()

```
int DoPreSwitch (
            UBYTE * s )
```

Definition at line 4012 of file pre.c.

4.84.3.60 DoPreShow()

```
int DoPreShow (
            UBYTE * s )
```

Definition at line 4066 of file pre.c.

4.84.3.61 DoSystem()

```
int DoSystem (
            UBYTE * s )
```

Definition at line 4099 of file pre.c.

4.84.3.62 PreLoad()

```
int PreLoad (
            PRELOAD * p,
            UBYTE * start,
            UBYTE * stop,
            int mode,
            char * message )
```

Definition at line 4126 of file pre.c.

4.84.3.63 PreSkip()

```
int PreSkip (
            UBYTE * start,
            UBYTE * stop,
            int mode )
```

Definition at line 4209 of file pre.c.

4.84.3.64 StartPrepro()

```
VOID StartPrepro ( )
```

Definition at line 4264 of file pre.c.

4.84.3.65 EvalPreIf()

```
int EvalPreIf (
            UBYTE * s )
```

Definition at line 4292 of file pre.c.

4.84.3.66 PreIfEval()

```
UBYTE * PreIfEval (
            UBYTE * s,
            int * value )
```

Definition at line 4327 of file pre.c.

4.84.3.67 PreCmp()

```
int PreCmp (
            int type,
            int val,
            UBYTE * t,
            int type2,
            int val2,
            UBYTE * t2,
            int cmpop )
```

Definition at line 4455 of file pre.c.

4.84.3.68 PreEq()

```
int PreEq (
            int type,
            int val,
            UBYTE * t,
            int type2,
            int val2,
            UBYTE * t2,
            int eqop )
```

Definition at line 4479 of file pre.c.

4.84.3.69 pParseObject()

```
UBYTE * pParseObject (
            UBYTE * s,
            int * type,
            LONG * val2 )
```

Definition at line 4508 of file pre.c.

4.84.3.70 PreCalc()

```
UBYTE * PreCalc ( )
```

Definition at line 4857 of file pre.c.

4.84.3.71 PreEval()

```
UBYTE * PreEval (
            UBYTE * s,
            LONG * x )
```

Definition at line 4933 of file pre.c.

4.84.3.72 AddToPreTypes()

```
void AddToPreTypes (
            int type )
```

Definition at line 5077 of file pre.c.

4.84.3.73 MessPreNesting()

```
void MessPreNesting (
            int par )
```

Definition at line 5095 of file pre.c.

4.84.3.74 DoPreAddSeparator()

```
int DoPreAddSeparator (
            UBYTE * s )
```

Definition at line 5118 of file pre.c.

4.84.3.75 DoPreRmSeparator()

```
int DoPreRmSeparator (
            UBYTE * s )
```

Definition at line 5143 of file pre.c.

4.84.3.76 DoExternal()

```
int DoExternal (
            UBYTE * s )
```

Definition at line 5160 of file pre.c.

4.84.3.77 DoPrompt()

```
int DoPrompt (
            UBYTE * s )
```

Definition at line 5242 of file pre.c.

4.84.3.78 DoSetExternal()

```
int DoSetExternal (
            UBYTE * s )
```

Definition at line 5275 of file pre.c.

4.84.3.79 DoSetExternalAttr()

```
int DoSetExternalAttr (
            UBYTE * s )
```

Definition at line 5345 of file pre.c.

4.84.3.80 DoRmExternal()

```
int DoRmExternal (
            UBYTE * s )
```

Definition at line 5461 of file pre.c.

4.84.3.81 DoFromExternal()

```
int DoFromExternal (
            UBYTE * s )
```

Definition at line 5526 of file pre.c.

4.84.3.82 DoToExternal()

```
int DoToExternal (
            UBYTE * s )
```

Definition at line 5721 of file pre.c.

4.84.3.83 defineChannel()

```
UBYTE * defineChannel (
            UBYTE * s,
            HANDLERS * h )
```

Definition at line 5768 of file pre.c.

4.84.3.84 writeToChannel()

```
int writeToChannel (
            int wtype,
            UBYTE * s,
            HANDLERS * h )
```

Definition at line 5803 of file pre.c.

4.84.3.85 DoFactDollar()

```
int DoFactDollar (
            UBYTE * s )
```

Definition at line 6355 of file pre.c.

4.84.3.86 GetDollarNumber()

```
WORD GetDollarNumber (
            UBYTE ** inp,
            DOLLARS d )
```

Definition at line 6392 of file pre.c.

4.84.3.87 DoSetRandom()

```
int DoSetRandom (
            UBYTE * s )
```

Definition at line 6482 of file pre.c.

4.84.3.88 DoOptimize()

```
int DoOptimize (
            UBYTE * s )
```

Definition at line 6525 of file pre.c.

4.84.3.89 DoClearOptimize()

```
int DoClearOptimize (
            UBYTE * s )
```

Definition at line 6658 of file pre.c.

4.84.3.90 DoSkipExtraSymbols()

```
int DoSkipExtraSymbols (
            UBYTE * s )
```

Definition at line 6678 of file pre.c.

4.84.3.91 DoPreReset()

```
int DoPreReset (
            UBYTE * s )
```

Definition at line 6717 of file pre.c.

4.84.3.92 DoPreAppendPath()

```
int DoPreAppendPath (
            UBYTE * s )
```

Appends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #appendpath <path>

Definition at line 6875 of file pre.c.

4.84.3.93 DoPrePrependPath()

```
int DoPrePrependPath (
            UBYTE * s )
```

Prepends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #prependpath <path>

Definition at line 6892 of file pre.c.

4.85 pre.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes :
*/
#include "form3.h"

static UBYTE pushbackchar = 0;
static int oldmode = 0;
static int stopdelay = 0;
static STREAM *oldstream = 0;
static UBYTE underscore[2] = {'_',0};
static PREVAR *ThePreVar = 0;

static KEYWORD precommands[] = {
     {"add"           , DoPreAdd       , 0, 0}
    ,{"addseparator" , DoPreAddSeparator,0,0}
    ,{"append"        , DoPreAppend    , 0, 0}
    ,{"appendpath"    , DoPreAppendPath, 0, 0}
    ,{"assign"        , DoPreAssign    , 0, 0}
    ,{"break"         , DoPreBreak     , 0, 0}
    ,{"breakdo"       , DoBreakDo      , 0, 0}
    ,{"call"          , DoCall         , 0, 0}
    ,{"case"          , DoPreCase      , 0, 0}
    ,{"clearoptimize", DoClearOptimize, 0, 0}
    ,{"close"         , DoPreClose     , 0, 0}
    ,{"closedictionary", DoPreCloseDictionary,0,0}
    ,{"commentchar"   , DoCommentChar  , 0, 0}
    ,{"create"        , DoPreCreate    , 0, 0}
    ,{"debug"         , DoDebug        , 0, 0}
    ,{"default"       , DoPreDefault   , 0, 0}
    ,{"define"        , DoDefine       , 0, 0}
    ,{"do"            , DoDo           , 0, 0}
    ,{"else"          , DoElse         , 0, 0}
    ,{"elseif"        , DoElseif       , 0, 0}
    ,{"enddo"         , DoEnddo        , 0, 0}
    ,{"endif"         , DoEndif        , 0, 0}
    ,{"endinside"     , DoEndInside    , 0, 0}
    ,{"endprocedure" , DoEndprocedure , 0, 0}
    ,{"endswitch"     , DoPreEndSwitch , 0, 0}
    ,{"exchange"      , DoPreExchange  , 0, 0}
    ,{"external"      , DoExternal     , 0, 0}
    ,{"factdollar"    , DoFactDollar   , 0, 0}
    ,{"fromexternal" , DoFromExternal , 0, 0}
    ,{"if"            , DoIf           , 0, 0}
    ,{"ifdef"         , (TFUN)DoIfdef  , 1, 0}
    ,{"ifndef"        , (TFUN)DoIfdef  , 2, 0}
    ,{"include"       , DoInclude      , 0, 0}
    ,{"inside"        , DoInside       , 0, 0}
    ,{"message"       , DoMessage      , 0, 0}
    ,{"opendictionary", DoPreOpenDictionary,0,0}
    ,{"optimize"      , DoOptimize     , 0, 0}
    ,{"pipe"          , DoPipe         , 0, 0}
    ,{"preout"        , DoPreOut       , 0, 0}
    ,{"prependpath"   , DoPrePrependPath,0, 0}
    ,{"printtimes"    , DoPrePrintTimes, 0, 0}
    ,{"procedure"     , DoProcedure    , 0, 0}
```

```
    ,{"procedureextension" , DoPrcExtension   , 0, 0}
    ,{"prompt"       , DoPrompt       , 0, 0}
    ,{"redefine"     , DoRedefine     , 0, 0}
    ,{"remove"       , DoPreRemove    , 0, 0}
    ,{"reset"        , DoPreReset     , 0, 0}
    ,{"reverseinclude"   , DoReverseInclude   , 0, 0}
    ,{"rmexternal"   , DoRmExternal   , 0, 0}
    ,{"rmseparator"  , DoPreRmSeparator,0, 0}
    ,{"setexternal"  , DoSetExternal  , 0, 0}
    ,{"setexternalattr"  , DoSetExternalAttr  , 0, 0}
    ,{"setrandom"    , DoSetRandom    , 0, 0}
    ,{"show"         , DoPreShow      , 0, 0}
    ,{"skipextrasymbols" , DoSkipExtraSymbols , 0, 0}
    ,{"switch"       , DoPreSwitch    , 0, 0}
    ,{"system"       , DoSystem       , 0, 0}
    ,{"terminate"    , DoTerminate    , 0, 0}
    ,{"toexternal"   , DoToExternal   , 0, 0}
    ,{"undefine"     , DoUndefine     , 0, 0}
    ,{"usedictionary", DoPreUseDictionary,0,0}
    ,{"write"        , DoPreWrite     , 0, 0}
};

/*
    #] Includes :
    # [ PreProcessor :
        #[ GetInput :

        Gets one input character. If we reach the end of a stream
        we pop to the previous stream and try again.
        If there are no more streams we let this be known.
*/

UBYTE GetInput()
{
    UBYTE c;
    while ( AC.CurrentStream ) {
        c = GetFromStream(AC.CurrentStream);
        if ( c != ENDOFSTREAM ) {
#ifdef WITHMPI
            if ( PF.me == MASTER
                 && AC.NoShowInput <= 0
                 && AC.CurrentStream->type != PREVARSTREAM )
#else
            if ( AC.NoShowInput <= 0 && AC.CurrentStream->type != PREVARSTREAM )
#endif
                CharOut(c);
            return(c);
        }
        AC.CurrentStream = CloseStream(AC.CurrentStream);
        if ( stopdelay && AC.CurrentStream == oldstream ) {
            stopdelay = 0; AP.AllowDelay = 1;
        }
    }
    return(ENDOFINPUT);
}

/*
        #] GetInput :
        #[ ClearPushback :
*/

VOID ClearPushback()
{
    pushbackchar = 0;
}

/*
        #] ClearPushback :
        #[ GetChar :

        Reads one character. If it encounters a quote it immediately
        takes the whole preprocessor variable and opens a stream
        for it and starts reading the stream.
        Note that we have to take special precautions for escaped quotes.
        That is why we remember the previous character. We allow the
        (dubious?) construction of ending a stream with a backslash and
        then using it to escape an object in the parent stream.
*/

UBYTE GetChar(int level)
{
    UBYTE namebuf[MAXPRENAMESIZE+2], c, *s, *t;
    static UBYTE lastchar, charinbuf = 0;
    int i, j, raiselow, olddelay;
    STREAM *stream;
    if ( level > 0 ) {
        lastchar = '`';
```

```
        goto higherlevel;
    }
    if ( pushbackchar ) { c = pushbackchar; pushbackchar = 0; return(c); }
    if ( charinbuf ) { c = charinbuf; charinbuf = 0; return(c); }
    c = GetInput();
    for(;;) {
        if ( c == '\\' ) {
            charinbuf = GetInput();
            if ( charinbuf != LINEFEED ) {
                pushbackchar = charinbuf;
                charinbuf = 0;
                break;
            }
            charinbuf = 0;  /* Escaped linefeed -> skip leading blanks */
            while ( ( c = GetInput() ) == ' ' || c == '\t' ) {}
        }
        else if ( c == '\" || c == '`' ) {
            if ( AP.DelayPrevar == 1 && c == '\" ) {
                AP.DelayPrevar = 0;
                break;
            }
            lastchar = c;
higherlevel:
            c = GetInput();
            if ( c == '!' && lastchar == '`' ) {
                if ( stopdelay == 0 ) oldstream = AC.CurrentStream;
                AP.AllowDelay = 0;
                stopdelay = 1;
                c = GetInput();
            }
            if ( c == '~' && lastchar == '`' ) {
                if ( AP.AllowDelay ) {
                    pushbackchar = c;
                    c = lastchar;
                    AP.DelayPrevar = 1;
                    break;
                }
            }
            else {
                pushbackchar = c;
            }
            olddelay = AP.DelayPrevar;
            AP.DelayPrevar = 0;
            i = 0; lastchar = 0;
            for (;;) {
                if ( pushbackchar ) { c = pushbackchar; pushbackchar = 0; }
                else { c = GetInput(); }
                if ( c == ENDOFINPUT || ( ( c == '\" || c == LINEFEED )
                && lastchar != '\\' ) ) {
                    break;
                }
                if ( c == '{' ) { /* Try the preprocessor calculator */
                    if ( PreCalc() == 0 ) Terminate(-1);
                    c = GetInput();    /* This is either a { or a number */
                    if ( c == '{' ) {
                        MesPrint("@Illegal set inside preprocessor variable name");
                        Terminate(-1);
                    }
                }
                if ( c == '`' && lastchar != '\\' ) {
                    c = GetChar(1);
                    if ( c == ENDOFINPUT || ( ( c == '\" || c == LINEFEED )
                    && lastchar != '\\' ) ) {
                        break;
                    }
                }
                if ( lastchar == '\\' ) { i--; lastchar = 0; }
                else lastchar = c;
                namebuf[i++] = c;
                if ( i > MAXPRENAMESIZE ) {
                    namebuf[i] = 0;
                    Error1("Preprocessor variable name too long: ",namebuf);
                }
            }
            namebuf[i++] = 0;
            if ( c != '\" ) {
                Error1("Unmatched quotes for preprocessor variable",namebuf);
            }
            AP.DelayPrevar = olddelay;
            if ( namebuf[0] == '$' ) {
                raiselow = PRENOACTION;
                if ( AP.PreproFlag && *AP.preStart) {
                    s = EndOfToken(AP.preStart);
                    c = *s; *s = 0;
                    if ( ( StrICmp(AP.preStart,(UBYTE *)"ifdef") == 0
                    || StrICmp(AP.preStart,(UBYTE *)"ifndef") == 0 )
                    && GetDollar(namebuf+1) < 0 ) {
```

```
                        *s = c; c = ' ';
                        break;
                    }
                    *s = c;
                }
                else {
                    s = EndOfToken(namebuf+1);
                    if ( *s == '[' ) { while ( *s ) s++; }
                }
                if ( *s == '-' && s[1] == '-' && s[2] == 0 )
                    raiselow = PRELOWERAFTER;
                else if ( *s == '+' && s[1] == '+' && s[2] == 0 )
                    raiselow = PRERAISEAFTER;
                c = *s; *s = 0;
                if ( OpenStream(namebuf+1,DOLLARSTREAM,0,raiselow) == 0 ) {
                    *s = c;
                    MesPrint("@Undefined variable %s used as preprocessor variable",
                        namebuf);
                    Terminate(-1);
                }
                *s = c;
            }
            else {
                raiselow = PRENOACTION;
                if ( AP.PreproFlag && *AP.preStart) {
                    s = EndOfToken(AP.preStart);
                    c = *s; *s = 0;
                    if ( ( StrICmp(AP.preStart,(UBYTE *)"ifdef") == 0
                    || StrICmp(AP.preStart,(UBYTE *)"ifndef") == 0 )
                    && GetPreVar(namebuf,WITHOUTERROR) == 0 ) {
                        *s = c; c = ' ';
                        break;
                    }
                    *s = c;
                }
                s = EndOfToken(namebuf);
                if ( *s == '_' ) s++;
                if ( *s == '-' && s[1] == '-' && s[2] == 0 )
                    raiselow = PRELOWERAFTER;
                else if ( *s == '+' && s[1] == '+' && s[2] == 0 )
                    raiselow = PRERAISEAFTER;
                else if ( *s == '(' && namebuf[i-2] == ')' ) {
/*
                    Now count the arguments and separate them by zeroes
                    Check on the ?var construction and if present, reset
                    some comma's.
                    Make the assignments of the variables
                    Run the macro.
                    Undefine the variables
*/
                    int nargs = 1;
                    PREVAR *p;
                    *s++ = 0; namebuf[i-2] = 0;
                    if ( StrICmp(namebuf,(UBYTE *)"random_") == 0 ) {
                        UBYTE *ranvalue;
                        ranvalue = PreRandom(s);
                        PutPreVar(namebuf,ranvalue,(UBYTE *)"?a",1);
                        M_free(ranvalue,"PreRandom");
                        goto dostream;
                    }
                    else if ( StrICmp(namebuf,(UBYTE *)"tolower_") == 0 ) {
                        UBYTE *ss = s;
                        while ( *ss ) { *ss = (UBYTE)(tolower(*ss)); ss++; }
                        PutPreVar(namebuf,s,(UBYTE *)"?a",1);
                        goto dostream;
                    }
                    else if ( StrICmp(namebuf,(UBYTE *)"toupper_") == 0 ) {
                        UBYTE *ss = s;
                        while ( *ss ) { *ss = (UBYTE)(toupper(*ss)); ss++; }
                        PutPreVar(namebuf,s,(UBYTE *)"?a",1);
                        goto dostream;
                    }
                    while ( *s ) {
                        if ( *s == '\\' ) s++;
                        if ( *s == ',' ) { *s = 0; nargs++; }
                        s++;
                    }
                    GetPreVar(namebuf,WITHERROR);
                    p = ThePreVar;
                    if ( p == 0 ) {
                        MesPrint("@Illegal use of arguments in preprocessor variable %s",namebuf);
                        Terminate(-1);
                    }
                    if ( p->nargs <= 0 || ( p->wildarg == 0 && nargs != p->nargs )
                        || ( p->wildarg > 0 && nargs < p->nargs-1 ) ) {
                        MesPrint("@Arguments of macro %s do not match",namebuf);
                        Terminate(-1);
```

```
                    }
                    if ( p->wildarg > 0 ) {
/*
                        Change some zeroes into commas
*/
                        s = namebuf;
                        for ( j = 0; j < p->wildarg; j++ ) {
                            while ( *s ) s++;
                            s++;
                        }
                        for ( j = 0; j < nargs-p->nargs; j++ ) {
                            while ( *s ) s++;
                            *s++ = ',';
                        }
                    }
/*
                        Now we can make the assignments
*/
                    s = namebuf;
                    while ( *s ) s++; s++;
                    t = p->argnames;
                    for ( j = 0; j < p->nargs; j++ ) {
                        if ( ( nargs == p->nargs-1 ) && ( *t == '?' ) ) {
                            PutPreVar(t,0,0,0);
                        }
                        else {
                            PutPreVar(t,s,0,0);
                            while ( *s ) s++; s++;
                        }
                        while ( *t ) t++; t++;
                    }
                }
dostream:;
                if ( ( stream = OpenStream(namebuf,PREVARSTREAM,0,raiselow) ) == 0 ) {
/*
                    Eat comma before or after. This is `no value'
*/
                }
                else if ( stream->inbuffer == 0 ) {
                    c = GetInput();
                    if ( level > 0 && c == '\" ) return(c);
                    goto endofloop;
                }
            }
            c = GetInput();
        }
        else if ( c == '{' ) { /* Try the preprocessor calculator */
            if ( PreCalc() == 0 ) Terminate(-1);
            c = GetInput();    /* This is either a { or a number */
            break;
        }
        else break;
endofloop:;
    }
    return(c);
}

/*
        #] GetChar :
        #[ CharOut :
*/

VOID CharOut(UBYTE c)
{
    if ( c == LINEFEED ) {
        AM.OutBuffer[AP.InOutBuf++] = c;
        WriteString(INPUTOUT,AM.OutBuffer,AP.InOutBuf);
        AP.InOutBuf = 0;
    }
    else {
        if ( AP.InOutBuf >= AM.OutBufSize || c == LINEFEED ) {
            WriteString(INPUTOUT,AM.OutBuffer,AP.InOutBuf);
            AP.InOutBuf = 0;
        }
        AM.OutBuffer[AP.InOutBuf++] = c;
    }
}

/*
        #] CharOut :
        #[ UnsetAllowDelay :
*/

VOID UnsetAllowDelay()
{
    if ( ThePreVar != 0 ) {
        if ( ThePreVar->nargs > 0 ) AP.AllowDelay = 0;
```

```
    }
}

/*
        #] UnsetAllowDelay :
        #[ GetPreVar :

        We use the model of a heap. If the same name has been used more
        than once the last definition is used. This gives the impression
        of local variables.

        There are two types: The regular ones and the expression variables.
        The last ones are like UNCHANGED_exprname and ZERO_exprname or
        UNCHANGED_ and ZERO_.
*/

static UBYTE *yes = (UBYTE *)"1";
static UBYTE *no  = (UBYTE *)"0";
static UBYTE numintopolynomial[12];
#include "vector.h"
static Vector(UBYTE, exprstr);  /* Used for numactiveexprs_ and activeexprnames_. */

UBYTE *GetPreVar(UBYTE *name, int flag)
{
    GETIDENTITY
    int i, mode;
    WORD number;
    UBYTE *t, c = 0, *tt = 0;
    t = name; while ( *t ) t++;
    if ( t[-1] == '-' && t[-2] == '-' && t-2 > name && t[-3] != '_' ) {
        t -= 2; c = *t; *t = 0; tt = t;
    }
    else if ( t[-1] == '+' && t[-2] == '+' && t-2 > name && t[-3] != '_' ) {
        t -= 2; c = *t; *t = 0; tt = t;
    }
    else if ( StrICmp(name,(UBYTE *)"time_") == 0 ) {
        UBYTE millibuf[24];
        LONG millitime, timepart;
        int timepart1, timepart2;
        static char timestring[40];
/*      millitime = TimeCPU(1); */
        millitime = GetRunningTime();
        timepart = millitime%1000;
        millitime /= 1000;
        timepart /= 10;
        timepart1 = timepart / 10;
        timepart2 = timepart % 10;
        NumToStr(millibuf,millitime);
        sprintf(timestring,"%s.%1d%1d",millibuf,timepart1,timepart2);
        return((UBYTE *)timestring);
    }
    else if ( ( StrICmp(name,(UBYTE *)"timer_") == 0 )
            || ( StrICmp(name,(UBYTE *)"stopwatch_") == 0 ) ) {
        static char timestring[40];
        sprintf(timestring,"%ld",(GetRunningTime() - AP.StopWatchZero));
        return((UBYTE *)timestring);
    }
    else if ( StrICmp(name, (UBYTE *)"numactiveexprs_") == 0 ) {
        /* the number of active expressions */
        int n = 0;
        for ( i = 0; i < NumExpressions; i++ ) {
            EXPRESSIONS e = Expressions + i;
            switch ( e->status ) {
                case LOCALEXPRESSION:
                case GLOBALEXPRESSION:
                case UNHIDELEXPRESSION:
                case UNHIDEGEXPRESSION:
                case INTOHIDELEXPRESSION:
                case INTOHIDEGEXPRESSION:
                    n++;
                    break;
            }
        }
        VectorReserve(exprstr, 41);  /* up to 128-bit */
        LongCopy(n, (char *)VectorPtr(exprstr));
        return VectorPtr(exprstr);
    }
    else if ( StrICmp(name, (UBYTE *)"activeexprnames_") == 0 ) {
        /* the list of active expressions separated by commas */
        int j = 0;
        VectorReserve(exprstr, 16);  /* at least 1 character for '\0' */
        for ( i = 0; i < NumExpressions; i++ ) {
            UBYTE *p, *s;
            int len, k;
            EXPRESSIONS e = Expressions + i;
            switch ( e->status ) {
                case LOCALEXPRESSION:
```

```
                case GLOBALEXPRESSION:
                case UNHIDELEXPRESSION:
                case UNHIDEGEXPRESSION:
                case INTOHIDELEXPRESSION:
                case INTOHIDEGEXPRESSION:
                    s = AC.exprnames->namebuffer + e->name;
                    len = StrLen(s);
                    VectorSize(exprstr) = j;  /* j bytes must be copied in extending the buffer. */
                    VectorReserve(exprstr, j + len * 2 + 1);
                    p = VectorPtr(exprstr);
                    if ( j > 0 ) p[j++] = ',';
                    for ( k = 0; k < len; k++ ) {
                        if ( s[k] == ',' || s[k] == '|' ) p[j++] = '\\';
                        p[j++] = s[k];
                    }
                    break;
            }
        }
        VectorPtr(exprstr)[j] = '\0';
        return VectorPtr(exprstr);
    }
    else if ( StrICmp(name, (UBYTE *)"path_") == 0 ) {
        /* the current FORM path (for debugging both in .c and .frm) */
        if ( AM.Path ) {
            return(AM.Path);
        }
        else {
            return((UBYTE *)"");
        }
    }
    t = name;
    while ( *t && *t != '_' ) t++;
    for ( i = NumPre-1; i >= 0; i-- ) {
        if ( *t == '_' && ( StrICmp(name,PreVar[i].name) == 0 ) ) {
            if ( c ) *tt = c;
            ThePreVar = PreVar+i;
            return(PreVar[i].value);
        }
        else if ( StrCmp(name,PreVar[i].name) == 0 ) {
            if ( c ) *tt = c;
            ThePreVar = PreVar+i;
            return(PreVar[i].value);
        }
    }
    if ( *t == '_' ) {
        if ( StrICmp(name,(UBYTE *)"EXTRASYMBOLS_") == 0 ) goto extrashort;
        *t = 0;
        if ( StrICmp(name,(UBYTE *)"UNCHANGED") == 0 ) mode = 1;
        else if ( StrICmp(name,(UBYTE *)"ZERO") == 0 ) mode = 0;
        else if ( StrICmp(name,(UBYTE *)"SHOWINPUT") == 0 ) {
            *t++ = '_';
            if ( c ) *tt = c;
            if ( AC.NoShowInput > 0 ) return(no);
            else return(yes);
        }
        else if ( StrICmp(name,(UBYTE *)"EXTRASYMBOLS") == 0 ) {
            *t++ = '_';
extrashort:;
            number = cbuf[AM.sbufnum].numrhs;
            t = numintopolynomial;
            NumCopy(number,t);
            return(numintopolynomial);
        }
        else mode = -1;
        *t++ = '_';
        if ( mode >= 0 ) {
            ThePreVar = 0;
            if ( *t ) {
                if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
                    if ( c ) *tt = c;
                    if ( ( Expressions[number].vflags & ( 1 « mode ) ) != 0 )
                        return(yes);
                    else return(no);
                }
            }
            else {
/*
                Here we have to test all active results.
                These are in `negative' so the flags have to be zero.
*/
                if ( c ) *tt = c;
                if ( ( AR.expflags & ( 1 « mode ) ) == 0 ) return(yes);
                else return(no);
            }
        }
    }
    if ( ( t = (UBYTE *)(getenv((char *)(name))) ) != 0 ) {
```

```
        if ( c ) *tt = c;
        ThePreVar = 0;
        return(t);
    }
    if ( c ) *tt = c;
    if ( flag == WITHERROR ) {
        Error1("Undefined preprocessor variable",name);
    }
    return(0);
}

/*
        #] GetPreVar :
        #[ PutPreVar :
*/

int PutPreVar(UBYTE *name, UBYTE *value, UBYTE *args, int mode)
{
    int i, ii, num = 2, nnum = 2, numargs = 0;
    UBYTE *s, *t, *u = 0;
    PREVAR *p;
    if ( value == 0 && name[0] != '?' ) {
        MesPrint("@Illegal empty value for preprocessor variable %s",name);
        Terminate(-1);
    }
    if ( args ) {
        s = args; num++;
        while ( *s ) {
            if ( *s != ' ' && *s != '\t' ) num++;
            s++;
        }
    }
    if ( mode == 1 ) {
        i = NumPre;
        while ( --i >= 0 ) {
            if ( StrCmp(name,PreVar[i].name) == 0 ) {
                u = PreVar[i].name;
                break;
            }
        }
    }
    else i = -1;
    if ( i < 0 ) { p = (PREVAR *)FromList(&AP.PreVarList); ii = p - PreVar; }
    else         { p = &(PreVar[i]);            ii = i; }
    if ( value ) {
        s = value; while ( *s ) { s++; num++; }
    }
    else num = 1;
    if ( i >= 0 ) {
        if ( p->value ) {
            s = p->value;
            while ( *s ) { s++; nnum++; }
        }
        else nnum = 1;
        if ( nnum >= num ) {
/*
            We can keep this in place
*/
            if ( value && p->value ) {
                s = value;
                t = p->value;
                while ( *s ) *t++ = *s++; *t = 0;
            }
            else p->value = 0;
            return(i);
        }
    }
    s = name;  while ( *s ) { s++; num++; }
    t = (UBYTE *)Malloc1(num,"PreVariable");
    p->name = t;
    s = name;  while ( *s ) *t++ = *s++; *t++ = 0;
    if ( value ) {
        p->value = t;
        s = value; while ( *s ) *t++ = *s++; *t = 0;
        if ( AM.atstartup && t[-1] == '\n' ) t[-1] = 0;
    }
    else p->value = 0;
    p->wildarg = 0;
    if ( args ) {
        int first = 1;
        t++; p->argnames = t;
        s = args;
        while ( *s ) {
            if ( *s == ' ' || *s == '\t' ) { s++; continue; }
            if ( *s == ',' ) {
                s++; *t++ = 0; numargs++;
                while ( *s == ' ' || *s == '\t' ) s++;
```

```
                if ( *s == '?' ) {
                    if ( p->wildarg > 0 ) {
                        Error0("More than one ?var in #define");
                    }
                    p->wildarg = numargs;
                }
            }
            else if ( *s == '?' && first ) {
                p->wildarg = 1; *t++ = *s++;
            }
            else { *t++ = *s++; }
            first = 0;
        }
        *t = 0;
        numargs++;
        p->nargs = numargs;
    }
    else {
        p->nargs = 0;
        p->argnames = 0;
    }
    if ( u ) M_free(u,"replace PreVar value");
    return(ii);
}

/*
        #] PutPreVar :
        #[ PopPreVars :
*/

VOID PopPreVars(int tonumber)
{
    PREVAR *p = &(PreVar[NumPre]);
    while ( NumPre > tonumber ) {
        NumPre--; p--;
        M_free(p->name,"popping PreVar");
        p->name = p->value = 0;
    }
}

/*
        #] PopPreVars :
        #[ IniModule :
*/

VOID IniModule(int type)
{
    GETIDENTITY
    WORD **w, i;
    CBUF *C = cbuf+AC.cbufnum;
    /*[05nov2003 mt]:*/
#ifdef WITHMPI
    /* To prevent
     *   (1) FlushOut() and PutOut() on the slaves to send a mess to the master
     *       compiling a module,
     *   (2) EndSort() called from poly_factorize_expression() on the master
     *       waits for the slaves.
     */
    PF.parallel=0;
    /*BTW, this was the bug preventing usage of more than 1 expression!*/
#endif

    AR.BracketOn = 0;
    AR.StoreData.dirtyflag = 0;
    AC.bracketindexflag = 0;
    AT.bracketindexflag = 0;

/*[06nov2003 mt]:*/
#ifdef WITHMPI
    /* This flag may be set in the procedure tokenize(). */
    AC.RhsExprInModuleFlag = 0;
/*[20oct2009 mt]:*/
    PF.mkSlaveInfile=0;
    PF.slavebuf.PObuffer=NULL;
    for(i=0; i<NumExpressions; i++)
        Expressions[i].vflags &= ~ISINRHS;
/*:[20oct2009 mt]*/
#endif
/*:[06nov2003 mt]*/

    /*[19nov2003 mt]:*/
    /*The module counter:*/
    (AC.CModule)++;
    /*:[19nov2003 mt]*/

    if ( !type ) {
        if ( C->rhs ) {
```

```
            w = C->rhs; i = C->maxrhs;
            do { *w++ = 0; } while ( --i > 0 );
        }
        if ( C->lhs ) {
            w = C->lhs; i = C->maxlhs;
            do { *w++ = 0; } while ( --i > 0 );
        }
    }
    C->numlhs = C->numrhs = 0;
    ClearTree(AC.cbufnum);
    while ( AC.NumLabels > 0 ) {
        AC.NumLabels--;
        if ( AC.LabelNames[AC.NumLabels] ) M_free(AC.LabelNames[AC.NumLabels],"LabelName");
    }

    C->Pointer = C->Buffer;

    AC.Commercial[0] = 0;

    AC.IfStack = AC.IfHeap;
    AC.arglevel = 0;
    AC.termlevel = 0;
    AC.IfLevel = 0;
    AC.WhileLevel = 0;
    AC.RepLevel = 0;
    AC.insidelevel = 0;
    AC.doloopelevel = 0;
    AC.MustTestTable = 0;
    AO.PrintType = 0;               /* Otherwise statistics can get spoiled */
    AC.ComDefer = 0;
    AC.CollectFun = 0;
    AM.S0->PolyWise = 0;
    AC.SymChangeFlag = 0;
    AP.lhdollarerror = 0;
    AR.PolyFun = AC.lPolyFun;
    AR.PolyFunInv = AC.lPolyFunInv;
    AR.PolyFunType = AC.lPolyFunType;
    AR.PolyFunExp = AC.lPolyFunExp;
    AR.PolyFunVar = AC.lPolyFunVar;
    AR.PolyFunPow = AC.lPolyFunPow;
    AC.mparallelflag = AC.parallelflag | AM.hparallelflag;
    AC.inparallelflag = 0;
    AC.mProcessBucketSize = AC.ProcessBucketSize;
    NumPotModdollars = 0;
    AC.topolynomialflag = 0;
#ifdef WITHPTHREADS
    if ( AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
    else AS.MultiThreaded = 0;
    for ( i = 1; i < AM.totalnumberofthreads; i++ ) {
        AB[i]->T.S0->PolyWise = 0;
    }
#endif
    OpenTemp();
}

/*
        #] IniModule :
        #[ IniSpecialModule :
*/

VOID IniSpecialModule(int type)
{
    DUMMYUSE(type);
}

/*
        #] IniSpecialModule :
        #[ PreProcessor :
*/

VOID PreProcessor()
{
    int moduletype = FIRSTMODULE;
    int specialtype = 0;
    int error1 = 0, error2 = 0, retcode, numstatement, retval;
    UBYTE c, *t, *s;
    AP.StopWatchZero = GetRunningTime();
    AC.compiletype = 0;
    AP.PreContinuation = 0;
    AP.PreAssignLevel = 0;
    AP.gNumPre = NumPre;
    AC.iPointer = AC.iBuffer;
    AC.iPointer[0] = 0;

    if ( AC.CheckpointFlag == -1 ) DoRecovery(&moduletype);
    AC.CheckpointStamp = Timer(0);

```

```
    for(;;) {
/*      if ( A.StatisticsFlag ) CharOut(LINEFEED); */

        IniModule(moduletype);

        /*Re-define preprocessor variable CMODULE_ as a current module number, starting from 1*/
        /*The module counter is AC.CModule, it is incremented in IniModule*/
        {
            UBYTE buf[24];/*64/Log_2[10] = 19.3, this is enough for any integer*/
            NumToStr(buf,AC.CModule);
            PutPreVar((UBYTE *)"CMODULE_",buf,0,1);
        }

        if ( specialtype ) IniSpecialModule(specialtype);

        numstatement = 0;
        for(;;) {   /* Read a single line/statement */
            c = GetChar(0);
            if ( c == AP.ComChar ) {  /* This line is commentary */
                LoadInstruction(5);
                if ( AC.CurrentStream->FoldName ) {
                    t = AP.preStart;
                    if ( *t && t[1] && t[2] == '#' && t[3] == ']' ) {
                        t += 4;
                        while ( *t == ' ' || *t == '\t' ) t++;
                        s = AC.CurrentStream->FoldName;
                        while ( *s == *t ) { s++; t++; }
                        if ( *s == 0 && ( *t == ' ' || *t == '\t'
                        || *t == ':' ) ) {
                            while ( *t == ' ' || *t == '\t' ) t++;
                            if ( *t == ':' ) {
                                AC.CurrentStream = CloseStream(AC.CurrentStream);
                            }
                        }
                    }
                }
                *AP.preStart = 0;
                continue;
            }
            while ( c == ' ' || c == '\t' ) c = GetChar(0);
            if ( c == LINEFEED ) continue;
            if ( c == ENDOFINPUT ) {
/*              CharOut(LINEFEED); */
                Warning(".end instruction generated");
                moduletype = ENDMODULE; specialtype = 0;
                goto endmodule; /* Fake one */
            }
            if ( c == '#' ) {
                if ( PreProInstruction() ) { error1++; error2++; AP.preError++; }
                *AP.preStart = 0;
            }
            else if ( c == '.' ) {
                if ( ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ||
                ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) ) {
                    LoadInstruction(1);
                    continue;
                }
                if ( ModuleInstruction(&moduletype,&specialtype) ) { error2++; AP.preError++; }
                if ( specialtype ) SetSpecialMode(moduletype,specialtype);
                if ( AP.PreInsideLevel != 0 ) {
                    MesPrint("@end of module instructions may not be used inside");
                    MesPrint("@the scope of a %#inside %#endinside construction.");
                    Terminate(-1);
                }
                if ( AC.RepLevel > 0 ) {
                    MesPrint("&EndRepeat statement(s) missing");
                    error2++; AP.preError++;
                }
                if ( AC.tablecheck == 0 ) {
                    AC.tablecheck = 1;
                    if ( TestTables() ) { error2++; AP.preError++; }
                }
                if ( AP.PreContinuation ) {
                    error1++; error2++;
                    MesPrint("&Unfinished statement. Missing ;?");
                }
                if ( moduletype == GLOBALMODULE ) MakeGlobal();
                else {
endmodule:          if ( error2 == 0 && AM.qError == 0 ) {
                        retcode = ExecModule(moduletype);
#ifdef WITHMPI
                        if(PF.slavebuf.PObuffer!=NULL){
                            M_free(PF.slavebuf.PObuffer,"PF inbuf");
                            PF.slavebuf.PObuffer=NULL;
                        }
#endif
                        UpdatePositions();
```

```
                        if ( retcode < 0 ) error1++;
                        if ( retcode ) { error2++; AP.preError++; }
                    }
                    else {
                        EXPRESSIONS e;
                        WORD j;
                        for ( j = 0, e = Expressions; j < NumExpressions; j++, e++ ) {
                            if ( e->replace == NEWLYDEFINEDEXPRESSION ) e->replace =
    REGULAREXPRESSION;
                        }
                    }
                    switch ( moduletype ) {
                        case STOREMODULE:
                            if ( ExecStore() ) error1++;
                            break;
                        case CLEARMODULE:
                            FullCleanUp();
                            error1 = error2 = AP.preError = 0;
                            AM.atstartup = 1;
                            PutPreVar((UBYTE *)"DATE_",(UBYTE *)MakeDate(),0,1);
                            AM.atstartup = 0;
                            if ( AM.resetTimeOnClear ) {
#ifdef WITHPTHREADS
                                ClearAllThreads();
#endif
                                AM.SumTime += TimeCPU(1);
                                TimeCPU(0);
                            }
                            AP.StopWatchZero = GetRunningTime();
                            break;
                        case ENDMODULE:
                            Terminate( -( error1 | error2 ) );
                    }
                }
                AC.tablecheck = 0;
                AC.compiletype = 0;
                if ( AC.exprfillwarning > 0 ) {
                    AC.exprfillwarning = 0;
                }
                if ( AC.CheckpointFlag && error1 == 0 && error2 == 0 ) DoCheckpoint(moduletype);
                break;  /* start a new module */
            }
            else {
                if ( ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ||
                ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) ) {
                    pushbackchar = c;
                    LoadInstruction(5);
                    continue;
                }
                UngetChar(c);
                if ( AP.PreContinuation ) {
                    retval = LoadStatement(OLDSTATEMENT);
                }
                else {
                    numstatement++;
                    AC.CurrentStream->prevline = AC.CurrentStream->linenumber;
                    retval = LoadStatement(NEWSTATEMENT);
                }
                if ( retval < 0 ) {
                    error1++;
                    if ( retval == -1 ) AP.PreContinuation = 0;
                    else AP.PreContinuation = 1;
                    TryRecover(0);
                }
                else if ( retval > 0 ) AP.PreContinuation = 0;
                else AP.PreContinuation = 1;
                if ( error1 == 0 && !AP.PreContinuation ) {
                    if ( ( AP.PreDebug & PREPROONLY ) == 0 ) {
                        int onpmd = NumPotModdollars;
#ifdef WITHMPI
                        WORD oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
                        if ( AP.PreAssignFlag ) AC.RhsExprInModuleFlag = 0;
#endif
                        if ( AP.PreOut || ( AP.PreDebug & DUMPTOCOMPILER )
                                == DUMPTOCOMPILER )
                                    MesPrint(" %s",AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]);
                        retcode = CompileStatement(AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]);
                        if ( retcode < 0 ) error1++;
                        if ( retcode ) { error2++; AP.preError++; }
                        if ( AP.PreAssignFlag ) {
                            if ( retcode == 0 ) {
                                if ( ( retcode = CatchDollar(0) ) < 0 ) error1++;
                                else if ( retcode > 0 ) { error2++; AP.preError++; }
                            }
                            else CatchDollar(-1);
                            POPPREASSIGNLEVEL;
                            if ( AP.PreAssignLevel <=0 )
```

```
                                AP.PreAssignFlag = 0;
                            NumPotModdollars = onpmd;
#ifdef WITHMPI
                            AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
#endif
                        }
                    }
                    else {
                        MesPrint(" %s",AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]);
                    }
                }
                else if ( !AP.PreContinuation ) {
                    if ( AP.PreAssignLevel > 0 ) {
                        POPPREASSIGNLEVEL;
                        if ( AP.PreAssignLevel <=0 )
                            AP.PreAssignFlag = 0;
                    }
                }
/*
                if ( !AP.PreContinuation ) AP.PreAssignFlag = 0;
*/
            }
        }
    }
}

/*
        #] PreProcessor :
        #[ PreProInstruction :
*/

int PreProInstruction()
{
    UBYTE *s, *t;
    KEYWORD *key;
    AP.PreproFlag = 1;
    AP.preFill = 0;
    AP.AllowDelay = 0;
    AP.DelayPrevar = 0;

    oldmode = 0;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) {
        LoadInstruction(3);
        if ( ( StrICmp(AP.preStart,(UBYTE *)"case") == 0
        || StrICmp(AP.preStart,(UBYTE *)"default") == 0 )
        && AP.PreSwitchModes[AP.PreSwitchLevel] == SEARCHINGPRECASE ) {
            LoadInstruction(0);
        }
        else if ( StrICmp(AP.preStart,(UBYTE *)"assign ") == 0 ) {}
        else { LoadInstruction(1); }
    }
    else if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) {
        LoadInstruction(3);
        if ( ( StrICmp(AP.preStart,(UBYTE *)"else") == 0
        || StrICmp(AP.preStart,(UBYTE *)"elseif") == 0 )
        && AP.PreIfStack[AP.PreIfLevel] == LOOKINGFORELSE ) {
            LoadInstruction(0);
        }
        else if ( StrICmp(AP.preStart,(UBYTE *)"assign ") == 0 ) {}
        else {
            LoadInstruction(1);
        }
    }
    else {
        LoadInstruction(0);
    }
    AP.PreproFlag = 0;
    t = AP.preStart;
    if ( *t == '-' ) {
        if ( AP.PreSwitchModes[AP.PreSwitchLevel] == EXECUTINGPRESWITCH
            && AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF )
            AC.NoShowInput = 1;
    }
    else if ( *t == '+' ) {
        if ( AP.PreSwitchModes[AP.PreSwitchLevel] == EXECUTINGPRESWITCH
            && AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF )
            AC.NoShowInput = 0;
    }
    else if ( *t == ':' ) {}
    else {
retry:;
        key = FindKeyWord(t,precommands,sizeof(precommands)/sizeof(KEYWORD));
        s = EndOfToken(t);
        if ( key == 0 ) {
            if ( *s == ';' ) {
                *s = 0; goto retry;
            }
```

```
            else {
                *s = 0;
                MesPrint("@Unrecognized preprocessor instruction: %s",t);
                return(-1);
            }
        }
        while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
        t = s;
        while ( *t ) t++;
        while ( ( t[-1] == ';' ) && ( t[-2] != '\\' ) ) {
            t--; *t = 0;
        }
        if ( key->type ) return(((TFUN1)key->func)(s,key->type));
        else return((key->func)(s));
    }
    return(0);
}

/*
        #] PreProInstruction :
        #[ LoadInstruction :

        0:  preprocessor instruction that may involve matching of brackets
        1:  runs straight to end-of-line
        2:  runs to ;
        3:  only gets one word without `' interpretation.
        5:  with pushbackchar, but inside commentary. -> 1

To be added:
    In define, redefine, call and listed do we may have delayed substitution
    of preprocessor variables.
*/

int LoadInstruction(int mode)
{
    UBYTE *s, *sstart, *t, c, cp;
    LONG position, fillpos = 0;
    int bralevel = 0, parlevel = 0, first = 1;
    int quotelevel = 0;
    if ( AP.preFill ) {
        s = AP.preFill;
        AP.preFill = 0;
        if ( s[1] != LINEFEED && s[1] != ENDOFINPUT ) {
            s[0] = s[1]; s++;
        }
        else { oldmode = mode; return(0); }
    }
    else { s = AP.preStart; }
    sstart = s; *s = 0;
    for(;;) {
        if ( ( mode & 1 ) == 1 ) {
            if ( pushbackchar && ( mode == 3 || mode == 5 ) ) {
                c = pushbackchar; pushbackchar = 0;
            }
            else c = GetInput();
        }
        else {
            c = GetChar(0);
        }

        if ( mode == 2 && c == ';' ) break;
        if ( ( mode == 1 || mode == 5 ) && c == LINEFEED ) break;
        if ( mode == 3 && FG.cTable[c] != 0 ) {
            if ( c == '$' ) {
                pushbackchar = '$';
                *s++ = 'a'; *s++ = 's'; *s++ = 's'; *s++ = 'i';
                *s++ = 'g'; *s++ = 'n'; *s++ = ' '; *s = 0;
            }
            AP.preFill = s; *s++ = 0; *s = c;
            oldmode = mode;
            return(0);
        }
        if ( mode == 0 && first ) {
            if ( c == '$' ) {
dodollar:       s = sstart;
                *s++ = 'a'; *s++ = 's'; *s++ = 's'; *s++ = 'i';
                *s++ = 'g'; *s++ = 'n'; *s = 0;
                pushbackchar = c;
                oldmode = mode;
                return(0);
            }
            if ( c == ' ' || c == '\t' || c == ',' ) {}
            else first = 0;
        }
        else if ( mode == 1 && first && c == '$' && oldmode == 3 ) goto dodollar;
        if ( c == ENDOFINPUT || ( c == LINEFEED
/*      && bralevel == 0  */
```

```
        && quotelevel == 0 ) ) {
            if ( mode == 2 && c == ENDOFINPUT ) {
                MesPrint("@Unexpected end of instruction");
                oldmode = mode;
                return(-1);
            }
/*
            if ( mode == 0 && bralevel ) {
                MesPrint("@Unmatched brackets");
                oldmode = mode;
                return(-1);
            }
*/
            if ( mode != 2 ) break;
        }
        if ( quotelevel ) {
            if ( c == '\\' ) {
                if ( ( mode == 1 ) || ( mode == 5 ) ) c = GetInput();
                else {
                    c = GetChar(0);
                }
                if ( c == ENDOFINPUT ) {
                    MesPrint("@Unmatched \"");
                    if ( mode == 2 && c == ENDOFINPUT ) {
                        MesPrint("@Unexpected end of instruction");
                    }
/*
                    if ( mode == 0 && bralevel ) {
                        MesPrint("@Unmatched brackets");
                    }
*/
                    oldmode = mode;
                    return(-1);
                }
                else if ( c == LINEFEED ) {}
                else if ( c == '"' ) { *s++ = '\\'; }
                else {
                    *s++ = '\\';
                }
            }
            else if ( c == '"' ) {
                quotelevel = 0;
                AP.AllowDelay = 0;
            }
        }
        else if ( c == '\\' ) {
            if ( ( mode == 1 ) || ( mode == 5 ) ) cp = GetInput();
            else {
                cp = GetChar(0);
            }
            if ( cp == LINEFEED ) continue;
            if ( mode != 2 || cp != ';' ) *s++ = c;
            c = cp;
        }
        else if ( c == '"' ) {
/*
            Now look back in the buffer and determine what the keyword is.
            If it is define or redefine, put AllowDelay to 1.
*/
            t = AP.preStart;
            while ( FG.cTable[*t] <= 1 ) t++;
            cp = *t; *t = 0;
            if ( ( StrICmp(AP.preStart,(UBYTE *)"define") == 0 )
                || ( StrICmp(AP.preStart,(UBYTE *)"redefine") == 0 ) ) {
                AP.AllowDelay = 1;
                oldstream = AC.CurrentStream;
            }
            *t = cp;
            quotelevel = 1;
        }
        else if ( quotelevel == 0 && bralevel == 0 && c == '(' ) {
            t = AP.preStart;
            while ( FG.cTable[*t] <= 1 ) t++;
            cp = *t; *t = 0;
            if ( ( parlevel == 0 )
                 && ( StrICmp(AP.preStart,(UBYTE *)"call") == 0 ) ) {
                AP.AllowDelay = 1;
                oldstream = AC.CurrentStream;
            }
            *t = cp;
            parlevel++;
        }
        else if ( quotelevel == 0 && bralevel == 0 && c == ')' ) {
            parlevel--;
        }
        else if ( quotelevel == 0 && parlevel == 0 && c == '{' ) {
            t = AP.preStart;
```

```
            while ( FG.cTable[*t] <= 1 ) t++;
            cp = *t; *t = 0;
            if ( ( bralevel == 0 )
                && ( ( StrICmp(AP.preStart,(UBYTE *)"call") == 0 )
                || ( StrICmp(AP.preStart,(UBYTE *)"do") == 0 ) ) ) {
                AP.AllowDelay = 1;
                oldstream = AC.CurrentStream;
            }
            *t = cp;
            bralevel++;
        }
        else if ( quotelevel == 0 && parlevel == 0 && c == '}' ) {
            bralevel--;
            if ( bralevel < 0 ) {
                if ( mode != 5 ) {
                    MesPrint("@Unmatched brackets");
                    oldmode = mode;
                    return(-1);
                }
                bralevel = 0;
            }
        }
        if ( s >= (AP.preStop-1) ) {
            UBYTE **ppp;
            position = s - AP.preStart;
            if ( AP.preFill ) fillpos = AP.preFill - AP.preStart;
            ppp = &(AP.preStart); /* to avoid a compiler warning */
            if ( DoubleLList((VOID ***)ppp,&AP.pSize,sizeof(UBYTE),
            "instruction buffer") ) { *s = 0; oldmode = mode; return(-1); }
            AP.preStop = AP.preStart + AP.pSize-3;
            s = AP.preStart + position;
            if ( AP.preFill ) AP.preFill = fillpos + AP.preStart;
        }
        *s++ = c;
    }
    *s = 0;
    oldmode = mode;
    if ( mode == 0 ) {
        if ( ExpandTripleDots(1) < 0 ) return(-1);
    }
    return(0);
}

/*
        #] LoadInstruction :
        #[ LoadStatement :

        Puts the current string together in the input buffer.
        Does things like placing comma's where needed and expand ...
        We force a comma after the keyword. Before 8-sep-2009 the program might
        not put a comma if a + or - followed. And then the compiler ate
        the + or - and we needed repair code in the routines that used the
        + or - (Print, modulus, multiply and (a)bracket). This worked but
        the problem was with statements like  Dimension -4; which then would
        be processed as Dimension 4; (JV)
*/

int LoadStatement(int type)
{
    UBYTE *s, c, cp;
    int retval = 0, stringlevel = 0, newstatement = 0;
    if ( type == NEWSTATEMENT ) { AP.eat = 1; newstatement = 1;
        s = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; }
    else { s = AC.iPointer; *s = 0; c = ' '; goto blank; }
    *s = 0;
    for(;;) {
        c = GetChar(0);
        if ( c == ENDOFINPUT ) { retval = -1; break; }
        if ( stringlevel == 0 ) {
            if ( c == LINEFEED ) { retval = 0; break; }
            if ( c == ';' ) {
                if ( AP.eat < 0 ) s--;
                while ( ( c = GetChar(0) ) == ' ' || c == '\t' ) {}
                if ( c != LINEFEED ) UngetChar(c);
                retval = 1;
                break;
            }
        }
        if ( c == '\\' ) {
            cp = GetChar(0);
            if ( cp == LINEFEED ) continue;
            *s++ = c;
            c = cp;
        }
        if ( c == '"' ) {
            if ( stringlevel == 0 ) stringlevel = 1;
            else stringlevel = 0;
```

```
            AP.eat = 0;
        }
        else if ( stringlevel == 0 ) {
            if ( c == '\t' ) c = ' ';
            if ( c == ' ' ) {
blank:          if ( newstatement < 0 ) newstatement = 0;
                if ( AP.eat && ( newstatement == 0 ) ) continue;
                c = ',';
                AP.eat = -2;
                if ( newstatement > 0 ) newstatement = -1;
            }
            else if ( chartype[c] <= 3 ) {
                AP.eat = 0;
                if ( newstatement < 0 ) newstatement = 0;
            }
            else if ( c == ',' ) {
                if ( newstatement > 0 ) {
                    newstatement = -1;
                    AP.eat = -2;
                }
/*              else if ( AP.eat == -2 ) { s--; } */
                else if ( AP.eat == -2 ) { AP.eat = 1; continue; }
                else { goto doall; }
            }
            else {
doall:;         if ( AP.eat < 0 ) {
                    if ( newstatement == 0 ) s--;
                    else { newstatement = 0; }
                }
                else if ( newstatement == 1 ) newstatement = 0;
                AP.eat = 1;
                if ( c == '*' && s > AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel] && s[-1] == '*' ) {
                    s[-1] = '^';
                    continue;
                }
            }
        }
        if ( s >= AC.iStop ) {
            if ( !AP.iBufError ) {
                LONG position = s - AC.iBuffer;
                LONG position2 = AC.iPointer - AC.iBuffer;
                UBYTE **ppp = &(AC.iBuffer); /* to avoid a compiler warning */
                if ( DoubleLList((VOID ***)ppp,&AC.iBufferSize
                ,sizeof(UBYTE),"statement buffer") ) {
                    *s = 0; retval = -1; AP.iBufError = 1;
                }
                AC.iPointer = AC.iBuffer + position2;
                AC.iStop = AC.iBuffer + AC.iBufferSize-2;
                s = AC.iBuffer + position;
            }
            if ( AP.iBufError ) {
                for(;;){
                    c = GetChar(0);
                    if ( c == ENDOFINPUT ) { retval = -1; break; }
                    if ( c == '"' ) {
                        if ( stringlevel > 0 ) stringlevel = 0;
                        else stringlevel = 1;
                    }
                    else if ( c == LINEFEED && !stringlevel ) { retval = -2; break; }
                    else if ( c == ';' && !stringlevel ) {
                        while ( ( c = GetChar(0) ) == ' ' || c == '\t' ) {}
                        if ( c != LINEFEED ) UngetChar(c);
                        retval = -1;
                        break;
                    }
                    else if ( c == '\\' ) c = GetChar(0);
                }
                break;
            }
        }
        *s++ = c;
    }
    AC.iPointer = s;
    *s = 0;
    if ( stringlevel > 0 ) {
        MesPrint("@Unbalanced \". Runaway string");
        retval = -1;
    }
    if ( retval == 1 ) {
        if ( ExpandTripleDots(0) < 0 ) retval = -1;
    }
    return(retval);
}

/*
        #] LoadStatement :
```

```
        #[ ExpandTripleDots :
*/

static inline int IsSignChar(UBYTE c)
{
    return c == '+' || c == '-';
}

static inline int IsAlphanumericChar(UBYTE c)
{
    return FG.cTable[c] == 0 || FG.cTable[c] == 1;
}

static inline int CanParseSignedNumber(const UBYTE *s)
{
    while ( IsSignChar(*s) ) s++;
    return FG.cTable[*s] == 1;
}

int ExpandTripleDots(int par)
{
    UBYTE *s, *s1, *s2, *n1, *n2, *t1, *t2, *startp, operator1, operator2, c, cc;
    UBYTE *nBuffer, *strngs, *Buffer, *Stop;
    LONG withquestion, x1, x2, y1, y2, number, inc, newsize, pow, fullsize;
    int i, error = 0, i1 ,i2, ii, *nums = 0;

    if ( par == 0 ) {
        Buffer = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; Stop = AC.iStop;
    }
    else {
        Buffer = AP.preStart; Stop = AP.preStop;
    }
    s = Buffer; while ( *s ) s++;
    fullsize = s - Buffer;
    if ( fullsize < 7 ) return(error);

    s = Buffer+2;
    while ( *s ) {
        if ( *s != '.' || ( s[-1] != ',' && FG.cTable[s[-1]] != 5 ) )
            { s++; continue; }
        if ( s[-1] == '%' || s[-1] == '^' || s[1] != '.' || s[2] != '.' )
            { s++; continue; }
        s1 = s - 2;
        s += 3;
        if ( *s != s[-4] && ( *s != '+' || s[-4] != '-' )
         && ( *s != '-' || s[-4] != '+' ) ) {
            MesPrint("&Improper operators for ...");
            error = -1;
        }
        operator1 = s[-4];
        operator2 = *s++;
        if ( operator1 == ':' ) operator1 = '.';
        if ( operator2 == ':' ) operator2 = '.';
/*
        We have now O1...O2 (O stands for operator)
        Full syntax is
        [str]#1[?]O1...O2[str]#2[?]   (Special case)
        in which both strings are identical and if one ? then also the other.
        <pattern1>O1...O2<pattern2>   (General case)
        in which the difference in the patterns is just numerical.
*/
        s2 = s;         /* the beginning of the second string */
        if ( *s2 != '<' || *s1 != '>' ) {   /* Special case */
            startp = s1+1;
            withquestion = ( *s1 == '?' ); s1--;
            while ( FG.cTable[*s1] == 1 && s1 >= Buffer ) s1--;
            n1 = s1+1;      /* Beginning of first number */
            if ( FG.cTable[*n1] != 1 ) {
                MesPrint("&No first number in ... operator");
                error = -1;
            }
            while ( FG.cTable[*s1] <= 1 && s1 >= Buffer ) s1--;
            s1++;
/*
            We have now the first string from s1 to n1, number from n1
*/
            t1 = s1; t2 = s2;
            while ( t1 < n1 && *t1 == *t2 ) { t1++; t2++; }
            n2 = t2;
            if ( FG.cTable[*t2] != 1 ) {
                MesPrint("&No second number in ... operator");
                error = -1;
            }
            x2 = 0;
            while ( FG.cTable[*t2] == 1 ) x2 = 10*x2 + *t2++ - '0';
            x1 = 0;
            while ( FG.cTable[*t1] == 1 ) x1 = 10*x1 + *t1++ - '0';
```

```
            if ( withquestion != ( *t2 == '?' ) ) {
                MesPrint("&Improper use of ? in ... operator");
                if ( *t2 == '?' ) t2++;
                error = -1;
            }
            else if ( withquestion ) t2++;
            if ( FG.cTable[*t2] <= 2 ) {
                MesPrint("&Illegal object after ... construction");
                error = -1;
            }
            c = *n1; *n1 = 0; s = t2;
            if ( error ) continue;
/*
            At this point the syntax has been fulfilled. We have
            str in s1.
            x1,x2 are #1,#2
            operator1,operator2 are the two operators.
            s points at whatever comes after.
            Expansion will have to be computed.
*/
            if ( x2 < x1 ) { number = x1-x2; inc = -1; y1 = x2; y2 = x1; }
            else           { number = x2-x1; inc =  1; y1 = x1; y2 = x2; }
            newsize = (number+1)*(n1-s1)            /* the strings   */
                     + number                       /* the operators */
                     +(number+1)*(withquestion?1:0) /* questionmarks */
                     +(number+1);                   /* last digits   */
            pow = 10;
            for ( i = 1; i < 10; i++, pow *= 10 ) {
                if ( y1 >= pow )      newsize += number+1;
                else if ( y2 >= pow ) newsize += y2-pow+1;
                else break;
            }
            while ( Buffer+(fullsize+newsize-(s-s1)) >= Stop ) {
                LONG strpos = s1-Buffer;
                LONG endstr = n1-Buffer;
                LONG startq = startp - Buffer;
                LONG position = s - Buffer;
                UBYTE **ppp;
                if ( par == 0 ) {
                    LONG position2 = AC.iPointer - AC.iBuffer;
                    ppp = &(AC.iBuffer); /* to avoid a compiler warning */
                    if ( DoubleLList((VOID ***)ppp,&AC.iBufferSize
                        ,sizeof(UBYTE),"statement buffer") ) {
                            Terminate(-1);
                    }
                    AC.iPointer = AC.iBuffer + position2;
                    AC.iStop = AC.iBuffer + AC.iBufferSize-2;
                    Buffer = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; Stop = AC.iStop;
                }
                else {
                    LONG fillpos = 0;
                    if ( AP.preFill ) fillpos = AP.preFill - AP.preStart;
                    ppp = &(AP.preStart); /* to avoid a compiler warning */
                    if ( DoubleLList((VOID ***)ppp,&AP.pSize,sizeof(UBYTE),
                        "instruction buffer") ) {
                            Terminate(-1);
                    }
                    AP.preStop = AP.preStart + AP.pSize-3;
                    if ( AP.preFill ) AP.preFill = fillpos + AP.preStart;
                    Buffer = AP.preStart; Stop = AP.preStop;
                }
                s  = Buffer + position;
                n1 = Buffer + endstr;
                s1 = Buffer + strpos;
                startp = Buffer + startq;
            }
/*
            We have space for the expansion in the buffer.
            There are two cases: new size >  old size
                                 old size >= new size
            Note that whereever we move things, it will be at least startp.
*/
            if ( newsize > (s-s1) ) {
                t2 = Buffer + fullsize;
                t1 = t2 + (newsize - (s-s1));
                *t1 = 0;
                while ( t2 > s ) { *--t1 = *--t2; }
            }
            else if ( newsize < (s-s1) ) {
                t1 = s1 + newsize; t2 = s; s = t1;
                while ( *t2 ) *t1++ = *t2++;
                *t1 = 0;
            }
            for ( x1 += inc, t1 = startp; number > 0; number--, x1 += inc ) {
                *t1++ = operator1;
                cc = operator1; operator1 = operator2; operator2 = cc;
                t2 = s1; while ( *t2 ) *t1++ = *t2++;
```

```
                x2 = x1; n2 = t1;
                do {
                    *t1++ = '0' + x2 % 10;
                    x2 /= 10;
                } while ( x2 );
                s2 = t1 - 1;
                while ( s2 > n2 ) { cc = *s2; *s2 = *n2; *n2++ = cc; s2--; }
                if ( withquestion ) *t1++ = '?';
            }
            fullsize += newsize - ( s - s1 );
            *n1 = c;
        }
        else {      /* General case. Find the patterns first */
            t1 = s1; s1--;
            while ( s1 > Buffer ) {
                if ( *s1 == '<' ) break;
                s1--;
            }
            t2 = s2;
            while ( *t2 ) {
                if ( *t2 == '>' ) break;
                t2++;
            }
            if ( *s1 != '<' || *t2 != '>' ) {
                MesPrint("&Illegal attempt to use ... operator");
                return(-1);
            }
            s1++; s2++;     /* Pointers to the patterns */
            nums = (int *)Malloc1((t1-s1)*2*(sizeof(int)+sizeof(UBYTE))
                        ,"Expand ...");
            strngs = (UBYTE *)(nums + 2*(t1-s1));
            n1 = s1; n2 = s2; ii = -1; i = 0;
            s = strngs;
            while ( n1 < t1 || n2 < t2 ) {
                /* Check the next characters can be parsed as numbers including signs. */
                if ( CanParseSignedNumber(n1) && CanParseSignedNumber(n2) ) {
                    /*
                     * Don't allow the cases that one has the sign and the other doesn't,
                     * and the meaning changes without the sign. For example,
                     *   <f(1)>+...+<f(3)>       Allowed
                     *   <f(-2)>+...+<f(2)>      Allowed
                     *   <f(x-2)>+...+<f(x+2)>   Allowed
                     *   <f(x-2)>+...+<f(x2)>    Not allowed
                     */
                    int sign1 = IsSignChar(*n1);
                    int sign2 = IsSignChar(*n2);
                    int inword1 = s1 < n1 && IsAlphanumericChar(n1[-1]);
                    int inword2 = s2 < n2 && IsAlphanumericChar(n2[-1]);
                    if ( ( sign1 ^ sign2 ) && ( inword1 || inword2 ) ) break;  /* Not allowed. */
                    if ( sign1 || sign2 ) {
                        *s++ = '+';  /* Marker indicating we need the sign. */
                    }
                } else {
                    /* If they are not numbers, they should be same. */
                    if ( *n1 == *n2 ) { *s++ = *n1++; n2++; continue; }
                    else break;
                }
                ParseSignedNumber(x1,n1)
                ParseSignedNumber(x2,n2)
                if ( x1 == x2 ) {
                    if ( s != strngs && ( s[-1] == '+' || s[-1] == '-' ) ) {
                        /* We need the sign. */
                        s--;
                        if ( x1 >= 0 ) {
                            *s++ = '+';
                        }
                    }
                    s = NumCopy(x1, s);
                }
                else {
                    nums[2*i] = x1; nums[2*i+1] = x2;
                    i++; *s++ = 0;
                }
            }
            if ( n1 < t1 || n2 < t2 ) {
                MesPrint("&Improper use of ... operator.");
theend:         M_free(nums,"Expand ...");
                return(-1);
            }
            *s = 0;
            if ( i == 0 ) ii = 0;
            else {
                ii = nums[0] - nums[1];
                if ( ii < 0 ) ii = -ii;
                for ( x1 = 1; x1 < i; x1++ ) {
                    x2 = nums[2*x1]-nums[2*x1+1];
                    if ( x2 < 0 ) x2 = -x2;
```

```
                    if ( x2 != ii ) {
                        MesPrint("&Improper synchronization of numbers in ... operator");
                        goto theend;
                    }
                }
            }
            ii++;
/*
            We have now proper syntax.
            There are i+1 strings in strngs and i pairs of numbers
            in nums. Each time a start value and a finish value.
            We have ii steps. If ii <= 2, it will fit in the existing
            allocation. But this is hardly useful.
            We make a new allocation and copy from the old.
            Compute space.
*/
            x2 = s - strngs - i;  /* -1 for eond-of-string and +1 for the operator*/
            for ( i1 = 0; i1 < i; i1++ ) {
                i2 = nums[2*i1];
                x1 = nums[2*i1+1];
                if ( i2 < 0 ) i2 = -i2;
                if ( x1 < 0 ) x1 = -x1;
                if ( x1 > i2 ) i2 = x1;
                x1 = 2;
                while ( i2 > 0 ) { i2 /= 10; x1++; }
                x2 += x1;
            }
            x2 *= ii;   /* Space for the expanded string (a bit more) */
            x2 += fullsize;
            x2 += 5;        /* This will definitely hold everything */
            x2 += sizeof(UBYTE *);
            x2 = x2 - (x2 & (sizeof(UBYTE *)-1));

            nBuffer = (UBYTE *)Malloc1(x2,"input buffer");
            n1 = nBuffer; s = Buffer; s1--;
            while ( s < s1 ) *n1++ = *s++;
/*
            Solution of the special case that no comma was generated
            due to the presence of < to start the pattern.
            We get a comma when the word before ends in an alphanumeric
            character, a _ or a ] and the word inside starts with an
            alphanumeric character, a [ (or an _ (for future considerations))
*/
            if ( ( ( n1 > nBuffer ) && ( ( FG.cTable[n1[-1]] <= 1 )
            || ( n1[-1] == '_' ) || ( n1[-1] == ']' ) ) ) &&
            ( ( FG.cTable[strngs[0]] <= 1 ) || ( strngs[0] == '[' )
             || ( strngs[0] == '_' ) ) ) *n1++ = ',';

            for ( i1 = 0; i1 < ii; i1++ ) {
                s = strngs; while ( *s ) *n1++ = *s++;
                for ( i2 = 0; i2 < i; i2++ ) {
                    if ( n1 > nBuffer && IsSignChar(n1[-1]) ) {
                        /* We need the sign of counters. */
                        n1--;
                        if ( nums[2*i2] >= 0 ) {
                            *n1++ = '+';
                        }
                    }
                    n1 = NumCopy((WORD)(nums[2*i2]),n1);
                    if ( nums[2*i2] > nums[2*i2+1] ) nums[2*i2]--;
                    else nums[2*i2]++;
                    s++; while ( *s ) *n1++ = *s++;
                }
                if ( ( i1 & 1 ) == 0 ) *n1++ = operator1;
                else *n1++ = operator2;
            }
            n1--;   /* drop the trailing operator */
            s = t2 + 1; n2 = n1;
/*
            Similar extra comma
*/
            if ( ( ( ( FG.cTable[n1[-1]] <= 1 )
            || ( n1[-1] == '_' ) || ( n1[-1] == ']' ) ) ) &&
            ( ( FG.cTable[s[0]] <= 1 ) || ( s[0] == '[' )
             || ( s[0] == '_' ) ) ) *n1++ = ',';

            while ( *s ) *n1++ = *s++;
            *n1 = 0;
            if ( par == 0 ) {
                LONG nnn1 = n1-nBuffer;
                LONG nnn2 = n2-nBuffer;
                LONG nnn3;
                while ( AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel] + x2 >= AC.iStop ) {
                    LONG position = s-Buffer;
                    LONG position2 = AC.iPointer - AC.iBuffer;
                    UBYTE **ppp;
                    ppp = &(AC.iBuffer); /* to avoid a compiler warning */
```

```
                    if ( DoubleLList((VOID ***)ppp,&AC.iBufferSize
                        ,sizeof(UBYTE),"statement buffer") ) {
                            Terminate(-1);
                    }
                    AC.iPointer = AC.iBuffer + position2;
                    AC.iStop = AC.iBuffer + AC.iBufferSize-2;
                    Buffer = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; Stop = AC.iStop;
                    s  = Buffer + position;
                }
/*
                This can be improved. We only have to start from the first term.
*/
                for ( nnn3 = 0; nnn3 < nnn1; nnn3++ ) Buffer[nnn3] = nBuffer[nnn3];
                Buffer[nnn3] = 0;
                n1 = Buffer + nnn1;
                n2 = Buffer + nnn2;
                M_free(nBuffer,"input buffer");
                M_free(nums,"Expand ...");
            }
            else { /* Comes here only inside a real preprocessor instruction */
                AP.preStop = nBuffer + x2 - 2;
                AP.pSize = x2;
                M_free(AP.preStart,"input buffer");
                M_free(nums,"Expand ...");
                AP.preStart = nBuffer;
                Buffer = AP.preStart; Stop = AP.preStop;
            }
            fullsize = n1 - Buffer;
            s = n2;
        }
    }
    return(error);
}

/*
        #] ExpandTripleDots :
        #[ FindKeyWord :
*/

KEYWORD *FindKeyWord(UBYTE *theword, KEYWORD *table, int size)
{
    int low,med,hi;
    UBYTE *s1, *s2;
    low = 0;
    hi = size-1;
    while ( hi >= low ) {
        med = (hi+low)/2;
        s1 = (UBYTE *)(table[med].name);
        s2 = theword;
        while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
        if ( *s1 == 0 &&
/*[30apr2004 mt]:*/
/* The bug!:
                FG.cTable[*s2] != 1 && FG.cTable[*s2] != 2
*/
                FG.cTable[*s2] != 0 && FG.cTable[*s2] != 1
/*      ( *s2 == ' ' || *s2 == '\t' || *s2 == 0 || *s2 == ',' || *s2 == '(' ) */
            )
            return(table+med);
        if ( tolower(*s2) > tolower(*s1) ) low = med+1;
        else hi = med - 1;
    }
    return(0);
}

/*
        #] FindKeyWord :
        #[ FindInKeyWord :
*/

KEYWORD *FindInKeyWord(UBYTE *theword, KEYWORD *table, int size)
{
    int i;
    UBYTE *s1, *s2;
    for ( i = 0; i < size; i++ ) {
        s1 = (UBYTE *)(table[i].name);
        s2 = theword;
        while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
        if ( *s2 == 0 || *s2 == ' ' || *s2 == ',' || *s2 == '\t' )
            return(table+i);
    }
    return(0);
}

/*
        #] FindInKeyWord :
        #[ TheDefine :
```

```
*/

int TheDefine(UBYTE *s, int mode)
{
    UBYTE *name, *value, *valpoin, *args = 0, c;
    if ( ( mode & 2 ) == 0 ) {
        if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
        if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    }
    else { mode &= ~2; }
    name = s;
    if ( chartype[*s] != 0 ) goto illname;
    s++;
    while ( chartype[*s] <= 1 ) s++;
    value = s;
    while ( *s == ' ' || *s == '\t' ) s++;
    c = *s; *value = 0;
    if ( c == 0 ) {
        if ( PutPreVar(name,(UBYTE *)"1",0,mode) < 0 ) return(-1);
        return(0);
    }
    if ( c == '(' ) {   /* arguments. scan for correctness */
        s++; args = s;
        for (;;) {
            if ( chartype[*s] != 0 ) goto illarg;
            s++;
            while ( chartype[*s] <= 1 ) s++;
            while ( *s == ' ' || *s == '\t' ) s++;
            if ( *s == ')' ) break;
            if ( *s != ',' ) goto illargs;
            s++;
            while ( *s == ' ' || *s == '\t' ) s++;
        }
        *s++ = 0;
        while ( *s == ' ' || *s == '\t' ) s++;
        c = *s;
    }
    if ( c == '"' ) {
        s++; valpoin = value = s;
        while ( *s != '"' ) {
            if ( *s == '\\' ) {
                if ( s[1] == 'n' ) { *valpoin++ = LINEFEED; s += 2; }
                else if ( s[1] == '"' ) { *valpoin++ = '"'; s += 2; }
                else if ( s[1] == 0 ) goto illval;
                else { *valpoin++ = *s++; *valpoin++ = *s++; }
            }
            else *valpoin++ = *s++;
        }
        *valpoin = 0;
        if ( PutPreVar(name,value,args,mode) < 0 ) return(-1);
    }
    else {
        MesPrint("@Illegal string for preprocessor variable %s. Forgotten double quotes (\") ?",name);
        return(-1);
    }
    return(0);
illname:;
    MesPrint("@Illegally formed name of preprocessor variable");
    return(-1);
illarg:;
    MesPrint("@Illegally formed name of argument of preprocessor definition");
    return(-1);
illargs:;
    MesPrint("@Illegally formed arguments of preprocessor definition");
    return(-1);
illval:;
    MesPrint("@Illegal valpoin for preprocessor variable %s",name);
    return(-1);
}

/*
        #] TheDefine :
        #[ DoCommentChar :
*/

int DoCommentChar(UBYTE *s)
{
    UBYTE c;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == 0 || *s == '\n' ) {
        MesPrint("@No valid comment character specified");
        return(-1);
    }
    c = *s++;
    while ( *s == ' ' || *s == '\t' ) s++;
```

```
    if ( *s != 0 && *s != '\n' ) {
        MesPrint("@Comment character should be a single valid character");
        return(-1);
    }
    AP.ComChar = c;
    return(0);
}

/*
        #] DoCommentChar :
        #[ DoPreAssign :

        Routine assigns a 'value' to a $variable.
        Syntax: #assign
            next line(s) a statement of the type
            $name = expression;
        Note: at the moment of the assign there cannot be an 'open' statement.
*/

int DoPreAssign(UBYTE *s)
{
    int error = 0;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) {
        return(0);
    }
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) {
        return(0);
    }
    if ( *s ) {
        MesPrint("@Illegal characters in %#assign instruction");
        error = 1;
    }
    PUSHPREASSIGNLEVEL;
    AP.PreAssignFlag = 1;
/*
    if ( AP.PreContinuation ) {
        MesPrint("@Assign instructions cannot occur inside statements");
        MesPrint("@Missing ; ?");
        AP.PreContinuation = 0;
        error = 1;
    }
*/
    return(error);
}

/*
        #] DoPreAssign :
        #[ DoDefine :
*/

int DoDefine(UBYTE *s)
{
    return(TheDefine(s,0));
}

/*
        #] DoDefine :
        #[ DoRedefine :
*/

int DoRedefine(UBYTE *s)
{
    return(TheDefine(s,1));
}

/*
        #] DoRedefine :
        #[ ClearMacro :

        Undefines the arguments of a macro after its use.
*/

int ClearMacro(UBYTE *name)
{
    int i;
    PREVAR *p;
    UBYTE *s;
    for ( i = NumPre-1, p = &(PreVar[NumPre-1]); i >= 0; i--, p-- ) {
        if ( StrCmp(name,p->name) == 0 ) break;
    }
    if ( i < 0 ) return(-1);
    if ( p->nargs <= 0 ) return(0);
    s = p->argnames;
    for ( i = 0; i < p->nargs; i++ ) {
        TheUndefine(s);
        while ( *s ) s++;
        s++;
```

```
    }
    return(0);
}

/*
        #] ClearMacro :
        #[ TheUndefine :

        There is a complication here. If there are redefine statements
        they will be pointing at the wrong variable if their number is
        greater than the number of the variable we pop.
*/

int TheUndefine(UBYTE *name)
{
    int i, inum, error = 0;
    PREVAR *p;
    for ( i = NumPre-1, p = &(PreVar[NumPre-1]); i >= 0; i--, p-- ) {
        if ( StrCmp(name,p->name) == 0 ) {
            M_free(p->name,"undefining PreVar");
            NumPre--;
            inum = i;
            while ( i < NumPre ) {
                p->name  = p[1].name;
                p->value = p[1].value;
                p++; i++;
            }
            p->name = 0; p->value = 0;
            {
                CBUF *CC = cbuf + AC.cbufnum;
                int j, k;
                for ( j = 1; j <= CC->numlhs; j++ ) {
                    if ( CC->lhs[j][0] == TYPEREDEFPRE ) {
                        if ( CC->lhs[j][2] > inum ) CC->lhs[j][2]--;
                        else if ( CC->lhs[j][2] == inum ) {
                            for ( k = inum - 1; k >= 0; k-- )
                                if ( StrCmp(name, PreVar[k].name) == 0 ) break;
                            if ( k >= 0 ) CC->lhs[j][2] = k;
                            else {
                                MesPrint("@Conflict between undefining a preprocessor variable and a redefine statement");
                                error = 1;
                            }
                        }
                    }
                }
#ifdef PARALLELCODE
                for ( j = 0; j < AC.numpfirstnum; j++ ) {
                    if ( AC.pfirstnum[j] > inum ) AC.pfirstnum[j]--;
                    else if ( AC.pfirstnum[j] == inum ) {
                        for ( k = inum - 1; k >= 0; k-- )
                            if ( StrCmp(name, PreVar[k].name) == 0 ) break;
                        if ( k >= 0 ) AC.pfirstnum[j] = k;
                    }
                }
#endif
            }
            break;
        }
    }
    return(error);
}

/*
        #] TheUndefine :
        #[ DoUndefine :
*/

int DoUndefine(UBYTE *s)
{
    UBYTE *name, *t;
    int error = 0, retval;
/*
    int i;
    PREVAR *p;
*/
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    name = s;
    if ( chartype[*s] != 0 ) goto illname;
    s++;
    while ( chartype[*s] <= 1 ) s++;
    t = s;
    if ( *s && *s != ' ' && *s != '\t' ) goto illname;
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s ) {
        MesPrint("@Undefine should just have a variable name");
```

```
        error = -1;
    }
    *t = 0;
    if ( ( retval = TheUndefine(name) ) != 0 ) {
        if ( error == 0 ) return(retval);
        if ( error > 0 ) error = retval;
    }
/*
    for ( i = NumPre-1, p = &(PreVar[NumPre-1]); i >= 0; i--, p-- ) {
        if ( StrCmp(name,p->name) == 0 ) {
            M_free(p->name,"undefining PreVar");
            NumPre--;
            while ( i < NumPre ) {
                p->name  = p[1].name;
                p->value = p[1].value;
                p++; i++;
            }
            p->name = 0; p->value = 0;
            break;
        }
    }
*/
    return(error);
illname:;
    MesPrint("@Illegally formed name of preprocessor variable");
    return(-1);
}

/*
        #] DoUndefine :
        #[ DoInclude :
*/

int DoInclude(UBYTE *s) { return(Include(s,FILESTREAM)); }

/*
        #] DoInclude :
        #[ DoReverseInclude :
*/

int DoReverseInclude(UBYTE *s) { return(Include(s,REVERSEFILESTREAM)); }

/*
        #] DoReverseInclude :
        #[ Include :
*/

int Include(UBYTE *s, int type)
{
    UBYTE *name = s, *fold, *t, c, c1 = 0, c2 = 0, c3 = 0;
    int str1offset, withnolist = AC.NoShowInput;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( *s == '-' || *s == '+' ) {
        if ( *s == '-' ) withnolist = 1;
        else             withnolist = 0;
        s++;
        while ( *s == ' ' || *s == '\t' ) s++;
        name = s;
    }
    if ( *s == '"' ) {
        while ( *s && *s != '"' ) {
            if ( *s == '\\' ) s++;
            s++;
        }
        t = s++;
    }
    else {
        while ( *s && *s != ' ' && *s != '\t' ) {
            if ( *s == '\\' ) s++;
            s++;
        }
        t = s;
    }
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == '#' ) {
        *t = 0;
        s++;
        while ( *s == ' ' || *s == '\t' ) s++;
        fold = s;
        if ( *s == 0 ) {
            MesPrint("@Empty fold name");
            return(-1);
        }
continue_fold:
        while ( *s && *s != ' ' && *s != '\t' ) {
            if ( *s == '\\' ) s++;
```

```
            s++;
        }
        t = s;
        while ( *s == ' ' || *s == '\t' ) s++;
        if ( *s ) {
            /*
             * A non-whitespace character is found. Continue parsing the fold.
             */
            goto continue_fold;
        }
    }
    else if ( *s == 0 ) {
        fold = 0;
    }
    else {
        MesPrint("@Improper syntax for file name");
        return(-1);
    }
    *t = 0;
    if ( fold ) {
        fold = strDup1(fold,"foldname");
    }
/*
    We have the name of the file in 'name' and the fold in 'fold' (or NULL)
*/
    if ( OpenStream(name,type,0,PRENOACTION) == 0 ) {
        if ( fold ) { M_free(fold,"foldname"); fold = 0; }
        return(-1);
    }
    if ( fold ) {
        LONG position = -1;
        int foldopen = 0;
        LONG linenum = 0, prevline = 0;
        name = strDup1(name,"name of include file");
        AC.CurrentStream->FoldName = strDup1(fold,"name of fold");
        AC.NoShowInput++;
        for(;;) {
            c = GetFromStream(AC.CurrentStream);
            if ( c == ENDOFSTREAM ) {
                AC.CurrentStream = CloseStream(AC.CurrentStream);
                goto nofold;
            }
            if ( c == AP.ComChar ) {
                str1offset = AC.CurrentStream-AC.Streams;
                LoadInstruction(1);
                if ( AC.CurrentStream != str1offset+AC.Streams ) {
                    c = ENDOFSTREAM;
                }
                else {
                    t = AP.preStart;
                    if ( t[2] == '#' && ( ( t[3] == '[' && !foldopen )
                    || ( t[3] == ']' && foldopen ) ) ) {
                        t += 4;
                        while ( *t == ' ' || *t == '\t' ) t++;
                        s = AC.CurrentStream->FoldName;
                        while ( *s == *t ) { s++; t++; }
                        if ( *s == 0 && ( *t == ' ' || *t == '\t'
                        || *t == ':' ) ) {
                            while ( *t == ' ' || *t == '\t' ) t++;
                            if ( *t == ':' ) {
                                if ( foldopen == 0 ) {
                                    foldopen = 1;
                                    position = GetStreamPosition(AC.CurrentStream);
                                    linenum = AC.CurrentStream->linenumber;
                                    prevline = AC.CurrentStream->prevline;
                                    c3 = AC.CurrentStream->isnextchar;
                                    c1 = AC.CurrentStream->nextchar[0];
                                    c2 = AC.CurrentStream->nextchar[1];
                                }
                                else {
                                    foldopen = 0;
                                    PositionStream(AC.CurrentStream,position);
                                    AC.CurrentStream->linenumber = linenum;
                                    AC.CurrentStream->prevline = prevline;
                                    AC.CurrentStream->eqnum = 1;
                                    AC.NoShowInput--;
                                    AC.CurrentStream->isnextchar = c3;
                                    AC.CurrentStream->nextchar[0] = c1;
                                    AC.CurrentStream->nextchar[1] = c2;
                                    break;
                                }
                            }
                        }
                    }
                }
            }
            else {
```

```
                while ( c != LINEFEED && c != ENDOFSTREAM ) {
                    c = GetFromStream(AC.CurrentStream);
                    if ( c == ENDOFSTREAM ) {
                        AC.CurrentStream = CloseStream(AC.CurrentStream);
                        break;
                    }
                }
            }
            if ( c == ENDOFSTREAM ) {
nofold:
                MesPrint("@Cannot find fold %s in file %s",fold,name);
                UngetChar(c);
                AC.NoShowInput--;
                M_free(name,"name of include file");
                Terminate(-1);
            }
        }
        M_free(name,"name of include file");
    }
    AC.NoShowInput = withnolist;
    if ( fold ) { M_free(fold,"foldname"); fold = 0; }
    return(0);
}

/*
        #] Include :
        #[ DoPreExchange :

        Exchanges the names of expressions or the contents of dollars
        Syntax:
            #exchange expr1,expr2
            #exchange $var1,$var2
*/

int DoPreExchange(UBYTE *s)
{
    int error = 0;
    UBYTE *s1, *s2;
    WORD num1, num2;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
    if ( *s == '$' ) {
        s++; s1 = s; while ( FG.cTable[*s] <= 1 ) s++;
        if ( *s != ',' && *s != ' ' && *s != '\t' ) goto syntax;
        *s++ = 0;
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
        if ( *s != '$' ) goto syntax;
        s++; s2 = s; while ( FG.cTable[*s] <= 1 ) s++;
        if ( *s != 0 && *s != ';' ) goto syntax;
        *s = 0;
        if ( ( num1 = GetDollar(s1) ) <= 0 ) {
            MesPrint("@$%s has not been defined (yet)",s1);
            error = 1;
        }
        if ( ( num2 = GetDollar(s2) ) <= 0 ) {
            MesPrint("@$%s has not been defined (yet)",s2);
            error = 1;
        }
        if ( error == 0 ) {
            ExchangeDollars((int)num1,(int)num2);
        }
    }
    else {
        s1 = s; s = SkipAName(s);
        if ( *s != ',' && *s != ' ' && *s != '\t' ) goto syntax;
        *s++ = 0;
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
        if ( FG.cTable[*s] != 0 && *s != '[' ) goto syntax;
        s2 = s; s = SkipAName(s);
        if ( *s != 0 && *s != ';' ) goto syntax;
        *s = 0;
        if ( GetName(AC.exprnames,s1,&num1,NOAUTO) != CEXPRESSION ) {
            MesPrint("@%s is not an expression",s1);
            error = 1;
        }
        if ( GetName(AC.exprnames,s2,&num2,NOAUTO) != CEXPRESSION ) {
            MesPrint("@%s is not an expression",s2);
            error = 1;
        }
        if ( error == 0 ) {
            ExchangeExpressions((int)num1,(int)num2);
        }
    }
    return(error);
syntax:
    MesPrint("@Proper syntax: %#exchange expr1,expr2 or %#exchange $var1,$var2");
```

```
    return(1);
}

/*
        #] DoPreExchange :
        #[ DoCall :
*/

int DoCall(UBYTE *s)
{
    UBYTE *t, *u, *v, *name, c, cp, *args1, *args2, *t1, *t2, *wild = 0;
    int bratype = 0, wildargs = 0, inwildargs = 0, nwildargs = 0;
    PROCEDURE *p;
    int streamoffset;
    int i, namesize, narg1, narg2, bralevel, numpre;
    LONG i1, i2;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
/*
    1:  Get the name of the procedure.
    2:  Locate the procedure.
*/
    name = s; s = EndOfToken(s); c = *s; *s = 0;
    for ( i = NumProcedures-1; i >= 0; i-- ) {
        if ( StrCmp(Procedures[i].name,name) == 0 ) break;
    }
    p = (PROCEDURE *)FromList(&AP.ProcList);
    if ( i < 0 ) {  /* Try to find a file */
        namesize = 0;
        t = name;
        while ( *t ) { t++; namesize++; }
        t = AP.procedureExtension;
        while ( *t ) { t++; namesize++; }
        t = p->name = (UBYTE *)Malloc1(namesize+2,"procedure");
        u = name;
        while ( *u ) *t++ = *u++;
        *t++ = '.';
        v = AP.procedureExtension;
        while ( *v ) *t++ = *v++;
        *t = 0;
        p->loadmode = 0;    /* buffer should be freed at end */
        p->p.buffer = LoadInputFile(p->name,PROCEDUREFILE);
        if ( p->p.buffer == 0 ) return(-1);
        t[-4] = 0;
    }
    else {
        p->p.buffer = Procedures[i].p.buffer;
        p->name = Procedures[i].name;
        p->loadmode = 1;
    }
    t = p->p.buffer;
    SKIPBLANKS(t)
    if ( *t++ != '#' ) goto wrongfile;
    SKIPBLANKS(t)
    t += 9;
    SKIPBLANKS(t)
    u = EndOfToken(t);
    cp = *u; *u = 0;
    if ( StrCmp(t,name) != 0 ) goto wrongfile;
    *u = cp;
    *s = c;
/*
    The pointer p points to the contents of the procedure (in memory)
    Now we have to match the arguments. u points to after the name
    in the 'file', s to after the name in the call statement.
*/
    bralevel = narg1 = narg2 = 0; args2 = u;
    SKIPBLANKS(u)
    if ( *u == '(' ) {
        u++; SKIPBLANKS(u)
        args2 = u;
        while ( *u != ')' ) {
            if ( *u == '?' ) { wildargs++; u++; nwildargs = narg2+1; }
            narg2++; u = EndOfToken(u); SKIPBLANKS(u)
            if ( *u == ',' ) { u++; SKIPBLANKS(u) }
            else if ( *u != ')' || ( wildargs > 1 ) ) {
                MesPrint("@Illegal argument field in procedure %s",p->name);
                return(-1);
            }
        }
    }
    while ( *u != LINEFEED ) u++;
    SKIPBLANKS(s)
    args1 = s+1;
    if ( *s == '(' ) bratype = 1;
    do {
        if ( *s == '{' && bratype == 0 ) bralevel++;
```

```
        else if ( *s == '(' && bratype == 1 ) bralevel++;
        else if ( *s == '}' && bratype == 0 ) {
            bralevel--;
            if ( bralevel == 0 ) {
                *s = 0; narg1++;
                if ( wildargs && narg1 == nwildargs ) wild = s;
            }
        }
        else if ( *s == ')' && bratype == 1 ) {
            bralevel--;
            if ( bralevel == 0 ) {
                *s = 0; narg1++;
                if ( wildargs && narg1 == nwildargs ) wild = s;
            }
        }
        /*[12dec2003 mt]:*/
        /*else if ( *s == ',' || *s == '|' ) {*/
        else if (set_in(*s,AC.separators)) {/*Function set_in see in
                                                        file tools.c*/
        /*:[12dec2003 mt]*/
            *s = 0; narg1++;
            if ( wildargs && narg1 == nwildargs ) wild = s;
        }
        else if ( *s == '\\' ) s++;
        s++;
    } while ( bralevel > 0 );
    if ( wildargs && narg1 >= narg2-1 ) {
        inwildargs = narg1-narg2+1;
        if ( inwildargs == 0 ) nwildargs = 0;
        else {
            while ( inwildargs > 1 ) {
                *wild = ',';
                while ( *wild ) wild++;
                inwildargs--;
            }
        }
    }
    else if ( narg1 != narg2 && ( narg2 != 0 || narg1 != 1 || *args1 != 0 ) ) {
        MesPrint("@Arguments of procedure %s are not matching",p->name);
        return(-1);
    }
    numpre = -NumPre-1; /* For the stream */
    for ( i = 0; i < narg2; i++ ) {
        t = args2;
        if ( *t == '?' ) {
            args2++;
        }
        if ( *t == '?' && inwildargs == 0 ) {
            args2 = EndOfToken(args2); c = *args2; *args2 = 0;
            if ( PutPreVar(t,(UBYTE *)"",0,0) < 0 ) return(-1);
        }
        else {
            args2 = EndOfToken(args2); c = *args2; *args2 = 0;
            t1 = t2 = args1;
            while ( *t1 ) {
                if ( *t1 == '\\' ) t1++;
                if ( t1 != t2 ) *t2 = *t1;
                t2++; t1++;
            }
            *t2 = 0;
            if ( PutPreVar(t,args1,0,0) < 0 ) return(-1);
            args1 = t1+1;                   /* Next argument */
        }
        *args2 = c; SKIPBLANKS(args2)  /* skip to next name */
        args2++; SKIPBLANKS(args2)
    }
    streamoffset = AC.CurrentStream - AC.Streams;
    args1 = AC.CurrentStream->name;
    AC.CurrentStream->name = p->name;
    i1 = AC.CurrentStream->linenumber;
    i2 = AC.CurrentStream->prevline;
    AC.CurrentStream->prevline   =
    AC.CurrentStream->linenumber = 2;
    OpenStream(u+1,PREREADSTREAM3,numpre,PRENOACTION);
    AC.Streams[streamoffset].name = args1;
    AC.Streams[streamoffset].linenumber = i1;
    AC.Streams[streamoffset].prevline   = i2;
    AddToPreTypes(PRETYPEPROCEDURE);
    return(0);
wrongfile:;
    if ( i < 0 ) MesPrint("@File %s is not a proper procedure",p->name);
    else MesPrint("!!!Internal error with procedure names: %s",name);
    return(-1);
}

/*
        #] DoCall :
```

```
        #[ DoDebug :
*/

int DoDebug(UBYTE *s)
{
    int x;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    NeedNumber(x,s,nonumber)
    if ( x < 0 || x >(PREPROONLY
                    | DUMPTOCOMPILER
                    | DUMPOUTTERMS
                    | DUMPINTERMS
                    | DUMPTOSORT
                    | DUMPTOPARALLEL
#ifdef WITHPTHREADS
                    | THREADSDEBUG
#endif
             ) ) goto nonumber;
    AP.PreDebug = 0;
    if ( ( x & PREPROONLY     ) != 0 ) AP.PreDebug |= PREPROONLY;     /* 1  */
    if ( ( x & DUMPTOCOMPILER ) != 0 ) AP.PreDebug |= DUMPTOCOMPILER; /* 2  */
    if ( ( x & DUMPOUTTERMS   ) != 0 ) AP.PreDebug |= DUMPOUTTERMS;   /* 4  */
    if ( ( x & DUMPINTERMS    ) != 0 ) AP.PreDebug |= DUMPINTERMS;    /* 8  */
    if ( ( x & DUMPTOSORT     ) != 0 ) AP.PreDebug |= DUMPTOSORT;     /* 16 */
    if ( ( x & DUMPTOPARALLEL ) != 0 ) AP.PreDebug |= DUMPTOPARALLEL; /* 32 */
#ifdef WITHPTHREADS
    if ( ( x & THREADSDEBUG   ) != 0 ) AP.PreDebug |= THREADSDEBUG;   /* 64 */
#endif
    return(0);
nonumber:
    MesPrint("@Illegal argument for debug instruction");
    return(1);
}

/*
        #] DoDebug :
        #[ DoTerminate :
*/

int DoTerminate(UBYTE *s)
{
    int x;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( *s ) {
        NeedNumber(x,s,nonumber)
        Terminate(x);
    }
    else {
        Terminate(-1);
    }
    return(0);
nonumber:
    MesPrint("@Illegal argument for terminate instruction");
    return(1);
}

/*
        #] DoTerminate :
        #[ DoDo :

        The do loop has three varieties:
        #do i = num1,num2 [,num3]
        #do i = {string1,string2,....,stringn}
            The | as separator is also allowed for backwards compatibility
        #do i = expression      One by one all terms of the expression
*/

int DoDo(UBYTE *s)
{
    GETIDENTITY
    UBYTE *t, c, *u, *uu;
    DOLOOP *loop;
    WORD expnum;
    LONG linenum  = AC.CurrentStream->linenumber;
    int oldNoShowInput = AC.NoShowInput, i, oldpreassignflag;

    if ( ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH )
    || ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ) {
        if ( PreSkip((UBYTE *)"do",(UBYTE *)"enddo",1) ) return(-1);
        return(0);
    }

/*
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
```

```
*/
    AddToPreTypes(PRETYPEDO);

    loop = (DOLOOP *)FromList(&AP.LoopList);
    loop->firstdollar = loop->lastdollar = loop->incdollar = -1;
    loop->NumPreTypes = AP.NumPreTypes-1;
    loop->PreIfLevel = AP.PreIfLevel;
    loop->PreSwitchLevel = AP.PreSwitchLevel;
    AC.NoShowInput = 1;
    if ( PreLoad(&(loop->p),(UBYTE *)"do",(UBYTE *)"enddo",1,"doloop") ) return(-1);
    AC.NoShowInput = oldNoShowInput;
    loop->NoShowInput = AC.NoShowInput;
/*
    Get now the name. We have to take great care when the name is terminated!
*/
    s = loop->p.buffer + (s - AP.preStart);
    SKIPBLANKS(s)
    loop->name = s;
    if ( chartype[*s] != 0 ) goto illname;
    s++;
    while ( chartype[*s] <= 1 ) s++;
    t = s;
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s != '=' ) goto illdo;
    s++;
    while ( *s == ' ' || *s == '\t' ) s++;
    *t = 0;

    if ( *s == '{' ) {
        loop->type = LISTEDLOOP;
        s++; loop->vars = s;
        loop->lastnum = 0;
        while ( *s != '}' && *s != 0 ) {
            if ( set_in(*s,AC.separators) ) { *s = 0; loop->lastnum++; }
            else if ( *s == '\\' ) s++;
            s++;
        }
        if ( *s == 0 ) goto illdo;
        *s++ = 0;
        loop->lastnum++;
        loop->firstnum = 0;
        loop->contents = s;
    }
    else if ( *s == '-' || *s == '+' || chartype[*s] == 1 || *s == '$' ) {
        loop->type = NUMERICALLOOP;
        t = s;
        while ( *s && *s != ',' ) s++;
        if ( *s == 0 ) goto illdo;
        if ( *t == '$' ) {
            c = *s; *s = 0;
            if ( GetName(AC.dollarnames,t+1,&loop->firstdollar,NOAUTO) != CDOLLAR ) {
                MesPrint("@%s is undefined in first parameter in %#do instruction",t);
                return(-1);
            }
            loop->firstnum = DolToLong(BHEAD loop->firstdollar);
            if ( AN.ErrorInDollar ) {
                MesPrint("@%s does not evaluate into a valid loop parameter",t);
                return(-1);
            }
            *s++ = c;
        }
        else {
            *s = '}';
            if ( PreEval(t,&loop->firstnum) == 0 ) goto illdo;
            *s++ = ',';
        }
        t = s;
        while ( *s && *s != ',' && *s != ';' && *s != LINEFEED ) s++;
        c = *s;
        if ( *t == '$' ) {
            *s = 0;
            if ( GetName(AC.dollarnames,t+1,&loop->lastdollar,NOAUTO) != CDOLLAR ) {
                MesPrint("@%s is undefined in second parameter in %#do instruction",t);
                return(-1);
            }
            loop->lastnum = DolToLong(BHEAD loop->lastdollar);
            if ( AN.ErrorInDollar ) {
                MesPrint("@%s does not evaluate into a valid loop parameter",t);
                return(-1);
            }
            *s++ = c;
        }
        else {
            *s = '}';
            if ( PreEval(t,&loop->lastnum) == 0 ) goto illdo;
            *s++ = c;
        }
```

```
        if ( c == ',' ) {
            t = s;
            while ( *s && *s != ';' && *s != LINEFEED ) s++;
            if ( *t == '$' ) {
                c = *s; *s = 0;
                if ( GetName(AC.dollarnames,t+1,&loop->incdollar,NOAUTO) != CDOLLAR ) {
                    MesPrint("@%s is undefined in third parameter in %#do instruction",t);
                    return(-1);
                }
                loop->incnum = DolToLong(BHEAD loop->incdollar);
                if ( AN.ErrorInDollar ) {
                    MesPrint("@%s does not evaluate into a valid loop parameter",t);
                    return(-1);
                }
                *s++ = c;
            }
            else {
                c = *s; *s = '}';
                if ( PreEval(t,&loop->incnum) == 0 ) goto illdo;
                *s++ = c;
            }
        }
        else loop->incnum = 1;
        loop->contents = s;
    }
    else if ( ( chartype[*s] == 0 ) || ( *s == '[' ) ) {
        int oldNumPotModdollars = NumPotModdollars;
#ifdef WITHMPI
        WORD oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
        AC.RhsExprInModuleFlag = 0;
#endif
        t = s;
        if ( ( s = SkipAName(s) ) == 0 ) goto illdo;
        c = *s; *s = 0;
        if ( GetName(AC.exprnames,t,&expnum,NOAUTO) == CEXPRESSION ) {
            loop->type = ONEEXPRESSION;
/*
            We should remember the expression by name for when it gets
            renumbered!!! If it gets deleted there will be a crash or at
            least the loop terminates.
*/
            loop->vars = t;
        }
        else goto illdo;
        if ( c == ',' || c == '\t' || c == ';' ) { s++; }
        else if ( c != 0 && c != '\n' ) goto illdo;
        while ( *s == ',' || *s == '\t' || *s == ';' ) s++;
        if ( *s != 0 && *s != '\n' ) goto illdo;
        loop->firstnum = 0;
        s++;
        loop->contents = s;
        loop->incnum = 0;
/*
        Next determine size of statement and allocate space
*/
        while ( *t ) t++;
        i = t - loop->vars;
        t = loop->name;
        while ( *t ) { t++; i++; }
        i += 4;
        loop->dollarname = Malloc1((LONG)i,"do-loop instruction");
/*
        Construct the statement
*/
        u = loop->dollarname;
        *u++ = '$'; t = loop->name; while ( *t ) *u++ = *t++;
        *u++ = '_'; uu = u; *u++ = '='; t = loop->vars;
        while ( *t ) *u++ = *t++; *t = 0; *u = 0;
/*
        Compile and put in dollar variable.
        Note that we remember the dollar by name and that this name ends in _
*/
        oldpreassignflag = AP.PreAssignFlag;
        AP.PreAssignFlag = 2;
        CompileStatement(loop->dollarname);
        if ( CatchDollar(0) ) {
            MesPrint("@Cannot load expression in do loop");
            return(-1);
        }
        AP.PreAssignFlag = oldpreassignflag;
        NumPotModdollars = oldNumPotModdollars;
#ifdef WITHMPI
        AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
#endif
        *uu = 0;
    }
    else goto illdo; /* Syntax problems */
```

```
    loop->errorsinloop = 0;
/*  loop->startlinenumber = linenum+1; 5-oct-2000 One too much? */
    loop->startlinenumber = linenum;
    PutPreVar(loop->name,(UBYTE *)"0",0,0);
    loop->firstloopcall = 1;
    return(DoEnddo(s));
illname:;
    MesPrint("@Improper name for do loop variable");
    return(-1);
illdo:;
    MesPrint("@Improper syntax in do loop instruction");
    return(-1);
}

/*
        #] DoDo :
        #[ DoBreakDo :

        #dobreak [num]
        jumps out of num #do-loops (if there are that many) (default is 1)
*/

int DoBreakDo(UBYTE *s)
{
    DOLOOP *loop;
    WORD levels;

    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);

    if ( NumDoLoops <= 0 ) {
        MesPrint("@%#dobreak without %#do");
        return(1);
    }
/*
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEDO ) { MessPreNesting(4); return(-1); }
*/
    while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
    if ( *s == 0 ) {
        levels = 1;
    }
    else if ( FG.cTable[*s] == 1 ) {
        levels = 0;
        while ( *s >= '0' && *s <= '9' ) { levels = 10*levels + *s++ - '0'; }
        if ( *s != 0 ) goto improper;
    }
    else {
improper:
        MesPrint("@Improper syntax of %#dobreak instruction");
        return(1);
    }
    if ( levels > NumDoLoops ) {
        MesPrint("@Too many loop levels requested in %#breakdo instruction");
        Terminate(-1);
    }
    while ( levels > 0 ) {
        while ( AC.CurrentStream->type != PREREADSTREAM
          && AC.CurrentStream->type != PREREADSTREAM2
          && AC.CurrentStream->type != PREREADSTREAM3 ) {
            AC.CurrentStream = CloseStream(AC.CurrentStream);
        }
        while ( AP.PreTypes[AP.NumPreTypes] != PRETYPEDO
        && AP.PreTypes[AP.NumPreTypes] != PRETYPEPROCEDURE ) AP.NumPreTypes--;
        if ( AC.CurrentStream->type == PREREADSTREAM3
        || AP.PreTypes[AP.NumPreTypes] == PRETYPEPROCEDURE ) {
            MesPrint("@Trying to jump out of a procedure with a %#breakdo instruction");
            Terminate(-1);
        }
        loop = &(DoLoops[NumDoLoops-1]);
        AP.NumPreTypes = loop->NumPreTypes;
        AP.PreIfLevel = loop->PreIfLevel;
        AP.PreSwitchLevel = loop->PreSwitchLevel;
/*
        AP.NumPreTypes--;
*/
        NumDoLoops--;
        DoUndefine(loop->name);
        M_free(loop->p.buffer,"loop->p.buffer");
        loop->firstloopcall = 0;

        AC.CurrentStream = CloseStream(AC.CurrentStream);
        levels--;
    }
    return(0);
}

/*
```

```
        #] DoBreakDo :
        #[ DoElse :
*/

int DoElse(UBYTE *s)
{
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEIF ) {
        if ( AP.PreIfLevel <= 0 ) MesPrint("@%#else without corresponding %#if");
        else MessPreNesting(1);
        return(-1);
    }
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    while ( *s == ' ' ) s++;
    if ( tolower(*s) == 'i' && tolower(s[1]) == 'f' && s[2]
        && FG.cTable[s[2]] > 1 && s[2] != '_' ) {
        s += 2;
        while ( *s == ' ' ) s++;
        return(DoElseif(s));
    }
    if ( AP.PreIfLevel <= 0 ) {
        MesPrint("@%#else without corresponding %#if");
        return(-1);
    }
    switch ( AP.PreIfStack[AP.PreIfLevel] ) {
        case EXECUTINGIF:
            AP.PreIfStack[AP.PreIfLevel] = LOOKINGFORENDIF;
            break;
        case LOOKINGFORELSE:
            AP.PreIfStack[AP.PreIfLevel] = EXECUTINGIF;
            break;
        case LOOKINGFORENDIF:
            break;
    }
    return(0);
}

/*
        #] DoElse :
        #[ DoElseif :
*/

int DoElseif(UBYTE *s)
{
    int condition;
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEIF ) {
        if ( AP.PreIfLevel <= 0 ) MesPrint("@%#elseif without corresponding %#if");
        else MessPreNesting(2);
        return(-1);
    }
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfLevel <= 0 ) {
        MesPrint("@%#elseif without corresponding %#if");
        return(-1);
    }
    switch ( AP.PreIfStack[AP.PreIfLevel] ) {
        case EXECUTINGIF:
            AP.PreIfStack[AP.PreIfLevel] = LOOKINGFORENDIF;
            break;
        case LOOKINGFORELSE:
            if ( ( condition = EvalPreIf(s) ) < 0 ) return(-1);
            AP.PreIfStack[AP.PreIfLevel] = condition;
            break;
        case LOOKINGFORENDIF:
            break;
    }
    return(0);
}

/*
        #] DoElseif :
        #[ DoEnddo :

        At the first call there is no stream yet.
        After that we have to close the stream and start a new one.
*/

int DoEnddo(UBYTE *s)
{
    GETIDENTITY
    DOLOOP *loop;
    UBYTE *t, *tt, *value, numstr[16];
    LONG xval;
    int xsign, retval;
    DUMMYUSE(s);
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
/*
```

```
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ||
        AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) {
        if ( AP.PreTypes[AP.NumPreTypes] == PRETYPEDO ) AP.NumPreTypes--;
        else { MessPreNesting(3); return(-1); }
        return(0);
    }
*/
    if ( NumDoLoops <= 0 ) {
        MesPrint("@%#enddo without %#do");
        return(1);
    }
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEDO ) { MessPreNesting(4); return(-1); }
    loop = &(DoLoops[NumDoLoops-1]);
    if ( !loop->firstloopcall ) AC.CurrentStream = CloseStream(AC.CurrentStream);

    if ( loop->errorsinloop ) {
        MesPrint("++++Errors in Loop");
        goto finish;
    }
    if ( loop->type == LISTEDLOOP ) {
        if ( loop->firstnum >= loop->lastnum ) goto finish;
        loop->firstnum++;
        t = value = loop->vars;
        while ( *value ) value++;
        value++;
        loop->vars = value;
        value = tt = t;
        while ( *value ) {
            if ( *value == '\\' ) value++;
            *tt++ = *value++;
        }
        *tt = 0;
        PutPreVar(loop->name,t,0,1);   /* We overwrite the definition */
    }
    else if ( loop->type == NUMERICALLOOP ) {

        if ( !loop->firstloopcall ) {
/*
            Test whether the variable was changed inside the loop into
            a different numerical value. If so, adjust.
*/
            t = GetPreVar(loop->name,WITHOUTERROR);
            if ( t ) {
                value = t;
                xsign = 1;
                while ( *value && ( *value == ' '
                        || *value == '-' || *value == '+' ) ) {
                    if ( *value == '-' ) xsign = -xsign;
                    value++;
                }
                t = value; xval = 0;
                while ( *value >= '0' && *value <= '9' ) xval = 10*xval + *value++ - '0';
                while ( *value && *value == ' ' ) value++;
                if ( *value == 0 ) {
/*
                    Now we may substitute the loopvalue.
*/
                    if ( xsign < 0 ) xval = -xval;
                    if ( loop->incdollar >= 0 ) {
                        loop->incnum = DolToLong(BHEAD loop->incdollar);
                        if ( AN.ErrorInDollar ) {
                            MesPrint("@%s does not evaluate into a valid third loop parameter",DOLLARNAME(Dollars,loop->incdollar));
                            return(-1);
                        }
                    }
                    loop->firstnum = xval + loop->incnum;
                }
            }
            if ( loop->lastdollar >= 0 ) {
                loop->lastnum = DolToLong(BHEAD loop->lastdollar);
                if ( AN.ErrorInDollar ) {
                    MesPrint("@%s does not evaluate into a valid second loop parameter",DOLLARNAME(Dollars,loop->lastdollar));
                    return(-1);
                }
            }
        }
        if ( ( loop->incnum > 0 && loop->firstnum > loop->lastnum )
        || ( loop->incnum < 0 && loop->firstnum < loop->lastnum ) ) goto finish;
        NumToStr(numstr,loop->firstnum);
        t = numstr;
        loop->firstnum += loop->incnum;
        PutPreVar(loop->name,t,0,1);   /* We overwrite the definition */
    }
    else if ( loop->type == ONEEXPRESSION ) {
/*
```

```
        Find the dollar expression
*/
        WORD numdollar = GetDollar(loop->dollarname+1);
        DOLLARS d = Dollars + numdollar;
        WORD *w, *dw, v, *ww;
        if ( (d->where) == 0 ) {
            d->type = DOLUNDEFINED;
            M_free(loop->dollarname,"do-loop instruction");
            goto finish;
        }
        w = d->where + loop->incnum;
        if ( *w == 0 ) {
            M_free(d->where,"dollar");
            d->where = 0;
            d->type = DOLUNDEFINED;
            M_free(loop->dollarname,"do-loop instruction");
            goto finish;
        }
        loop->incnum += *w;
/*
        Now the term has to be converted to text.
*/
        ww = w + *w; v = *ww; *ww = 0;
        dw = d->where; d->where = w;
        t = WriteDollarToBuffer(numdollar,1);
        d->where = dw; *ww = v;
        PutPreVar(loop->name,t,0,1);   /* We overwrite the definition */
        M_free(t,"dollar");
    }
    if ( loop->firstloopcall ) OpenStream(loop->contents,PREREADSTREAM2,0,PRENOACTION);
    else OpenStream(loop->contents,PREREADSTREAM,0,PRENOACTION);
    AC.CurrentStream->prevline   =
    AC.CurrentStream->linenumber = loop->startlinenumber;
    AC.CurrentStream->eqnum = 0;
    loop->firstloopcall = 0;
    return(0);
finish:;
    NumDoLoops--;
    retval = DoUndefine(loop->name);
    M_free(loop->p.buffer,"loop->p.buffer");
    loop->firstloopcall = 0;
    AP.NumPreTypes--;
    return(retval);
}

/*
        #] DoEnddo :
        #[ DoEndif :
*/

int DoEndif(UBYTE *s)
{
    DUMMYUSE(s);
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEIF ) {
        if ( AP.PreIfLevel <= 0 ) MesPrint("@%#endif without corresponding %#if");
        else MessPreNesting(5);
        return(-1);
    }
    AP.NumPreTypes--;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfLevel <= 0 ) {
        MesPrint("@%#endif without corresponding %#if");
        return(-1);
    }
    AP.PreIfLevel--;
    return(0);
}

/*
        #] DoEndif :
        #[ DoEndprocedure :

        Action is simple: close the current stream if it is still
        the stream from which the statement came.
        Then pop the current procedure and all its local derivatives.
        if loadmode > 1 the procedure was defined locally.
*/

int DoEndprocedure(UBYTE *s)
{
    DUMMYUSE(s);
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEPROCEDURE ) {
        MessPreNesting(6);
        return(-1);
    }
    AP.NumPreTypes--;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
```

```
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    AC.CurrentStream = CloseStream(AC.CurrentStream);
    do {
        NumProcedures--;
        if ( Procedures[NumProcedures].loadmode == 0 ) {
            M_free(Procedures[NumProcedures].p.buffer,"procedures buffer");
            M_free(Procedures[NumProcedures].name,"procedures name");
        }
    } while ( Procedures[NumProcedures].loadmode > 1 );
    return(0);
}

/*
        #] DoEndprocedure :
        #[ DoIf :
*/

int DoIf(UBYTE *s)
{
    int condition;
    AddToPreTypes(PRETYPEIF);
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF ) {
        condition = EvalPreIf(s);
        if ( condition < 0 ) return(-1);
    }
    else condition = LOOKINGFORENDIF;
    if ( AP.PreIfLevel+1 >= AP.MaxPreIfLevel ) {
        int **ppp = &AP.PreIfStack; /* To avoid a compiler warning */
        if ( DoubleList((VOID ***)ppp,&AP.MaxPreIfLevel,sizeof(int),
            "PreIfLevels") ) return(-1);
    }
    AP.PreIfStack[++AP.PreIfLevel] = condition;
    return(0);
}

/*
        #] DoIf :
        #[ DoIfdef :
*/

int DoIfdef(UBYTE *s, int par)
{
    int condition;
    AddToPreTypes(PRETYPEIF);
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF ) {
        while ( *s == ' ' || *s == '\t' ) s++;
        if ( ( *s == 0 ) == ( par == 1 ) ) condition = LOOKINGFORELSE;
        else                               condition = EXECUTINGIF;
    }
    else condition = LOOKINGFORENDIF;
    if ( AP.PreIfLevel+1 >= AP.MaxPreIfLevel ) {
        int **ppp = &AP.PreIfStack; /* to avoid a compiler warning */
        if ( DoubleList((VOID ***)ppp,&AP.MaxPreIfLevel,sizeof(int),
            "PreIfLevels") ) return(-1);
    }
    AP.PreIfStack[++AP.PreIfLevel] = condition;
    return(0);
}

/*
        #] DoIfdef :
        #[ DoInside :

    #inside $var1,...,$varn
        statements without .sort
    #endinside

    executes the statements on the contents of the $ variables as if they
    are a module. The results are put back in the dollar variables.
    To do this right we need a struct with
        old compiler buffer
        list of numbers of dollars
        length of the list
        length of the array containing the list
    Because we need to compose statements, the statement buffer must be
    empty. This means that we have to test for that. Same at the end. We
    must have a completed statement.
*/

int DoInside(UBYTE *s)
{
    GETIDENTITY
    int numdol, error = 0;
    WORD *nb, newsize, i;
    UBYTE *name, c;
```

```
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.PreInsideLevel != 0 ) {
        MesPrint("@Illegal nesting of %#inside/%#endinside instructions");
        return(-1);
    }
/*
    if ( AP.PreContinuation ) {
        error = -1;
        MesPrint("@%#inside cannot be inside a regular statement");
    }
*/
    PUSHPREASSIGNLEVEL
/*
    Now the dollars to do
*/
    AP.inside.numdollars = 0;
    for(;;) {
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
        if ( *s == 0 ) break;
        if ( *s != '$' ) {
            MesPrint("@%#inside instruction can have only $ variables for parameters");
            return(-1);
        }
        s++;
        name = s;
        while (chartype[*s] <= 1 ) s++;
        c = *s; *s = 0;
        if ( ( numdol = GetDollar(name) ) < 0 ) {
            MesPrint("@%#inside: $%s has not (yet) been defined",name);
            *s = c;
            error = -1;
        }
        else {
            *s = c;
            if ( AP.inside.numdollars >= AP.inside.size ) {
                if ( AP.inside.buffer == 0 ) newsize = 20;
                else                         newsize = 2*AP.inside.size;
                nb = (WORD *)Malloc1(newsize*sizeof(WORD),"insidebuffer");
                if ( AP.inside.buffer ) {
                    for ( i = 0; i < AP.inside.size; i++ ) nb[i] = AP.inside.buffer[i];
                    M_free(AP.inside.buffer,"insidebuffer");
                }
                AP.inside.buffer = nb;
                AP.inside.size = newsize;
            }
            AP.inside.buffer[AP.inside.numdollars++] = numdol;
        }
    }
/*
    We have to store the configuration of the compiler buffer, so that
    we know where to start executing and how to reset the buffer.
*/
    AP.inside.oldcompiletype = AC.compiletype;
    AP.inside.oldparallelflag = AC.mparallelflag;
    AP.inside.oldnumpotmoddollars = NumPotModdollars;
    AP.inside.oldcbuf = AC.cbufnum;
    AP.inside.oldrbuf = AM.rbufnum;
    AP.inside.oldcnumlhs = AR.Cnumlhs,
    AddToPreTypes(PRETYPEINSIDE);
    AP.PreInsideLevel = 1;
    AC.cbufnum = AP.inside.inscbuf;
    AM.rbufnum = AP.inside.inscbuf;
    clearcbuf(AC.cbufnum);
    AC.compiletype = 0;
    AC.mparallelflag = PARALLELFLAG;
#ifdef WITHMPI
    /*
     * We use AC.RhsExprInModuleFlag, PotModdollars, and AC.pfirstnum
     * in order to check (1) whether there are expression names in RHS,
     * (2) which dollar variables can be modified, and (3) which
     * preprocessor variables can be redefined, in #inside.
     * We store the current values of them, and then reset them.
     */
    PF_StoreInsideInfo();
    AC.RhsExprInModuleFlag = 0;
    NumPotModdollars = 0;
    AC.numpfirstnum = 0;
#endif
    return(error);
}

/*
        #] DoInside :
        #[ DoEndInside :
*/

```

```
int DoEndInside(UBYTE *s)
{
    GETIDENTITY
    WORD numdol, *oldworkpointer = AT.WorkPointer, *term, *t, j, i;
    DOLLARS d, nd;
    WORD oldbracketon = AR.BracketOn;
    WORD *oldcompresspointer = AR.CompressPointer;
    int oldmultithreaded = AS.MultiThreaded;
    /* int oldmparallelflag = AC.mparallelflag; */
    FILEHANDLE *f;
#ifdef WITHMPI
    int error = 0;
#endif
    DUMMYUSE(s);
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEINSIDE ) {
        if ( AP.PreInsideLevel != 1 ) MesPrint("@%#endinside without corresponding %#inside");
        else MessPreNesting(11);
        return(-1);
    }
    AP.NumPreTypes--;
    if ( AP.PreInsideLevel != 1 ) {
        MesPrint("@%#endinside without corresponding %#inside");
        return(-1);
    }
    if ( AP.PreContinuation ) {
        MesPrint("@%#endinside: previous statement not terminated.");
        Terminate(-1);
    }
    AC.compiletype = AP.inside.oldcompiletype;
    AR.Cnumlhs = cbuf[AM.rbufnum].numlhs;
#ifdef WITHMPI
    /*
     * If the #inside...#endinside contains expressions in RHS, only the master executes it
     * and then broadcasts the result to the all slaves. If not, the all processes execute
     * it and in this case no MPI interactions are needed.
     */
    if ( PF.me == MASTER || !AC.RhsExprInModuleFlag ) {
#endif
    AR.BracketOn = 0;
    AS.MultiThreaded = 0;
    /* AC.mparallelflag = PARALLELFLAG; */
    if ( AR.CompressPointer == 0 ) AR.CompressPointer = AR.CompressBuffer;
    f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
/*
    Now we have to execute the statements on the proper dollars.
*/
    for ( i = 0; i < AP.inside.numdollars; i++ ) {
        numdol = AP.inside.buffer[i];
        nd = d = Dollars + numdol;
        if ( d->type != DOLZERO ) {
            if ( d->type != DOLTERMS ) nd = DolToTerms(BHEAD numdol);
            term = nd->where;
            NewSort(BHEAD0);
            NewSort(BHEAD0);
            AR.MaxDum = AM.IndDum;
            while ( *term ) {
                t = oldworkpointer; j = *term;
                NCOPY(t,term,j);
                AT.WorkPointer = t;
                AN.IndDum = AM.IndDum;
                AR.CurDum = ReNumber(BHEAD term);
                if ( Generator(BHEAD oldworkpointer,0) ) {
                    MesPrint("@Called from %#endinside");
                    MesPrint("@Evaluating variable $%s",DOLLARNAME(Dollars,numdol));
                    Terminate(-1);
                }
            }
            AT.WorkPointer = oldworkpointer;
            CleanDollarFactors(d);
            if ( d->where ) { M_free(d->where,"dollar contents"); d->where = 0; }
            EndSort(BHEAD (WORD *)((VOID *)(&(d->where))),2);
            LowerSortLevel();
            term = d->where; while ( *term ) term += *term;
            d->size = term - d->where;
            if ( nd != d ) M_free(nd,"Copy of dollar variable");
            if ( d->where[0] == 0 ) {
                M_free(d->where,"dollar contents"); d->where = 0;
                d->type = DOLZERO;
            }
        }
    }
#ifdef WITHMPI
    }
    if ( AC.RhsExprInModuleFlag ) {
        /*
```

```
         * The only master executed the statements in #inside.
         * We need to broadcast the result to the all slaves.
         */
        for ( i = 0; i < AP.inside.numdollars; i++ ) {
            /*
             * Mark $-variables specified in the #inside instruction as modified
             * such that they will be broadcast.
             */
            AddPotModdollar(AP.inside.buffer[i]);
        }
        /* Now actual broadcast of modified variables. */
        if ( NumPotModdollars > 0 ) {
            error = PF_BroadcastModifiedDollars();
            if ( error ) goto cleanup;
        }
        if ( AC.numpfirstnum > 0 ) {
            error = PF_BroadcastRedefinedPreVars();
            if ( error ) goto cleanup;
        }
    }
cleanup:
#endif
    f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
    AC.cbufnum = AP.inside.oldcbuf;
    AM.rbufnum = AP.inside.oldrbuf;
    AR.Cnumlhs = AP.inside.oldcnumlhs;
    AR.BracketOn = oldbracketon;
    AP.PreInsideLevel = 0;
    AR.CompressPointer = oldcompresspointer;
    AS.MultiThreaded = oldmultithreaded;
    AC.mparallelflag = AP.inside.oldparallelflag;
    NumPotModdollars = AP.inside.oldnumpotmoddollars;
    POPPREASSIGNLEVEL
#ifdef WITHMPI
    PF_RestoreInsideInfo();
    if ( error ) return error;
#endif
    return(0);
}

/*
        #] DoEndInside :
        #[ DoMessage :
*/

int DoMessage(UBYTE *s)
{
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
    MesPrint("~~~%s",s);
    return(0);
}

/*
        #] DoMessage :
        #[ DoPipe :
*/

int DoPipe(UBYTE *s)
{
#ifndef WITHPIPE
    DUMMYUSE(s);
#endif
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
#ifdef WITHPIPE
    FLUSHCONSOLE;
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( OpenStream(s,PIPESTREAM,0,PRENOACTION) == 0 ) return(-1);
    return(0);
#else
    Error0("Pipes not implemented on this computer/system");
    return(-1);
#endif
}

/*
        #] DoPipe :
        #[ DoPrcExtension :
*/

int DoPrcExtension(UBYTE *s)
{
    UBYTE *t, *u, c;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
```

```
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == 0 || *s == '\n' ) {
        MesPrint("@No valid procedure extension specified");
        return(-1);
    }
    if ( FG.cTable[*s] != 0 ) {
        MesPrint("@Procedure extension should be a string starting with an alphabetic character. No whitespace.");
        return(-1);
    }
    t = s;
    while ( *s && *s != '\n' && *s != ' ' && *s != '\t' ) s++;
    u = s;
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s != 0 && *s != '\n' ) {
        MesPrint("@Too many parameters in ProcedureExtension instruction");
        return(-1);
    }
    c = *u; *u = 0;
    if ( AP.procedureExtension ) M_free(AP.procedureExtension,"ProcedureExtension");
    AP.procedureExtension = strDup1(t,"ProcedureExtension");
    *u = c;
    return(0);
}

/*
        #] DoPrcExtension :
        #[ DoPreOut :
*/

int DoPreOut(UBYTE *s)
{
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( tolower(*s) == 'o' ) {
        if ( tolower(s[1]) == 'n' && s[2] == 0 ) {
            AP.PreOut = 1;
            return(0);
        }
        if ( tolower(s[1]) == 'f' && tolower(s[2]) == 'f' && s[3] == 0 ) {
            AP.PreOut = 0;
            return(0);
        }
    }
    MesPrint("@Illegal option in PreOut instruction");
    return(-1);
}

/*
        #] DoPreOut :
        #[ DoPrePrintTimes :
*/

int DoPrePrintTimes(UBYTE *s)
{
    DUMMYUSE(s);
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    PrintRunningTime();
    return(0);
}

/*
        #] DoPrePrintTimes :
        #[ DoPreAppend :

        Syntax:
        #append <filename>
*/

int DoPreAppend(UBYTE *s)
{
    UBYTE *name, *to;

    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
/*
    Determine where to write
*/
    if ( *s == '<' ) {
        s++;
        name = to = s;
        while ( *s && *s != '>' ) {
            if ( *s == '\\' ) s++;
            *to++ = *s++;
```

```
        }
        if ( *s == 0 ) {
            MesPrint("@Improper termination of filename");
            return(-1);
        }
        s++;
        *to = 0;
        if ( *name ) { GetAppendChannel((char *)name); }
        else goto improper;
    }
    else {
improper:
        MesPrint("@Proper syntax is: %#append <filename>");
        return(-1);
    }
    return(0);
}

/*
        #] DoPreAppend :
        #[ DoPreCreate :

        Syntax:
        #create <filename>
*/

int DoPreCreate(UBYTE *s)
{
    UBYTE *name, *to;

    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
/*
    Determine where to write
*/
    if ( *s == '<' ) {
        s++;
        name = to = s;
        while ( *s && *s != '>' ) {
            if ( *s == '\\' ) s++;
            *to++ = *s++;
        }
        if ( *s == 0 ) {
            MesPrint("@Improper termination of filename");
            return(-1);
        }
        s++;
        *to = 0;
        if ( *name ) { GetChannel((char *)name); }
        else goto improper;
    }
    else {
improper:
        MesPrint("@Proper syntax is: %#create <filename>");
        return(-1);
    }
    return(0);
}

/*
        #] DoPreCreate :
        #[ DoPreRemove :
*/

int DoPreRemove(UBYTE *s)
{
    UBYTE *name, *to;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == '<' ) { s++; }
    else {
        MesPrint("@Proper syntax is: %#remove <filename>");
        return(-1);
    }
    name = to = s;
    while ( *s && *s != '>' ) {
        if ( *s == '\\' ) s++;
        *to++ = *s++;
    }
    if ( *s == 0 ) {
        MesPrint("@Improper filename");
        return(-1);
    }
```

```
    s++;
    *to = 0;
    CloseChannel((char *)name);
    remove((char *)name);
    return(0);
}

/*
        #] DoPreRemove :
        #[ DoPreClose :
*/

int DoPreClose(UBYTE *s)
{
    UBYTE *name, *to;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == '<' ) { s++; }
    else {
        MesPrint("@Proper syntax is: %#close <filename>");
        return(-1);
    }
    name = to = s;
    while ( *s && *s != '>' ) {
        if ( *s == '\\' ) s++;
        *to++ = *s++;
    }
    if ( *s == 0 ) {
        MesPrint("@Improper filename");
        return(-1);
    }
    s++;
    *to = 0;
    return(CloseChannel((char *)name));
}

/*
        #] DoPreClose :
        #[ DoPreWrite :

        Syntax:
        #write [<filename>] "formatstring" [,objects]
        The format string can contain the following special objects/codes
        \n  newline
        \t  tab
        \!  if last entry in string: no linefeed at end
        \b  put \ in output
        %$  $-variable (to be found among the objects)
        %e  expression (name to be found among the objects)
        %E  expression without ; (name to be found among the objects)
        %s  string (to be found among the objects) (with or without "")
        %S  subterms (see PrintSubtermList)
*/

int DoPreWrite(UBYTE *s)
{
    HANDLERS h;

    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);

#ifdef WITHMPI
    if ( PF.me != MASTER ) return 0;
#endif

    h.oldsilent    = AM.silent;
    h.newlogonly   = h.oldlogonly = AM.FileOnlyFlag;
    h.newhandle    = h.oldhandle  = AC.LogHandle;
    h.oldprinttype = AO.PrintType;

    while ( *s == ' ' || *s == '\t' ) s++;
/*
    Determine where to write
*/
    if( (s=defineChannel(s,&h))==0 ) return(-1);

    return(writeToChannel(WRITEOUT,s,&h));
}

/*
        #] DoPreWrite :
        #[ DoProcedure :

        We have to read this procedure into a buffer.
```

```
        The only complications are:
        1: we have to seek through the file to do this efficiently
           the file operations under VMS cannot do this properly
           (unless we use the proper ANSI structs?)
           This is the reason why we read whole input files under VMS.
        2: what to do when the same name is used twice.
        Note that we have to do the reading without substitution of
        preprocessor variables.
*/

int DoProcedure(UBYTE *s)
{
    UBYTE c;
    PROCEDURE *p;
    LONG i;
    if ( ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH )
    || ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ) {
        if ( PreSkip((UBYTE *)"procedure",(UBYTE *)"endprocedure",1) ) return(-1);
        return(0);
    }
    p = (PROCEDURE *)FromList(&AP.ProcList);
    if ( PreLoad(&(p->p),(UBYTE *)"procedure",(UBYTE *)"endprocedure"
        ,1,(char *)"procedure") ) return(-1);

    p->loadmode = 2;
    s = p->p.buffer + 10;
    while ( *s == ' ' || *s == LINEFEED ) s++;
    if ( chartype[*s] ) {
        MesPrint("@Illegal name for procedure");
        return(-1);
    }
    p->name = s++;
    while ( chartype[*s] == 0 || chartype[*s] == 1 ) s++;
    c = *s; *s = 0;
    p->name = strDup1(p->name,"procedure");
    *s = c;
/*
    Check for double names
*/
    for ( i = NumProcedures-2; i >= 0; i-- ) {
        if ( StrCmp(Procedures[i].name,p->name) == 0 ) {
            Error1("Multiple occurrence of procedure name ",p->name);
        }
    }
    return(0);
}

/*
        #] DoProcedure :
        #[ DoPreBreak :
*/

int DoPreBreak(UBYTE *s)
{
    DUMMYUSE(s);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
        if ( AP.PreSwitchLevel <= 0 )
             MesPrint("@Break without corresponding Switch");
        else MessPreNesting(7);
        return(-1);
    }
    if ( AP.PreSwitchLevel <= 0 ) {
        MesPrint("@Break without corresponding Switch");
        return(-1);
    }
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] == EXECUTINGPRESWITCH )
        AP.PreSwitchModes[AP.PreSwitchLevel] = SEARCHINGPREENDSWITCH;
    return(0);
}

/*
        #] DoPreBreak :
        #[ DoPreCase :
*/

int DoPreCase(UBYTE *s)
{
    UBYTE *t;
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
        if ( AP.PreSwitchLevel <= 0 )
             MesPrint("@Case without corresponding Switch");
        else MessPreNesting(8);
        return(-1);
    }
    if ( AP.PreSwitchLevel <= 0 ) {
```

```
        MesPrint("@Case without corresponding Switch");
        return(-1);
    }
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != SEARCHINGPRECASE ) return(0);

    SKIPBLANKS(s)
    t = s;
    while ( *s ) { if ( *s == '\\' ) s++; s++; }
    while ( s > t && ( s[-1] == ' ' || s[-1] == '\t' ) && s[-2] != '\\' ) {
        if ( s[-2] == '\\' ) s--;
        s--;
    }
    if ( *t == '"' && s > t+1 && s[-1] == '"' && s[-2] != '\\' ) {
        t++; s--; *s = 0;
    }
    else *s = 0;
    s = AP.PreSwitchStrings[AP.PreSwitchLevel];
    while ( *t == *s && *t ) { s++; t++; }
    if ( *t || *s ) return(0);  /* case did not match */
    AP.PreSwitchModes[AP.PreSwitchLevel] = EXECUTINGPRESWITCH;
    return(0);
}

/*
        #] DoPreCase :
        #[ DoPreDefault :
*/

int DoPreDefault(UBYTE *s)
{
    DUMMYUSE(s);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
        if ( AP.PreSwitchLevel <= 0 )
             MesPrint("@Default without corresponding Switch");
        else MessPreNesting(9);
        return(-1);
    }
    if ( AP.PreSwitchLevel <= 0 ) {
        MesPrint("@Default without corresponding Switch");
        return(-1);
    }
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != SEARCHINGPRECASE ) return(0);
    AP.PreSwitchModes[AP.PreSwitchLevel] = EXECUTINGPRESWITCH;
    return(0);
}

/*
        #] DoPreDefault :
        #[ DoPreEndSwitch :
*/

int DoPreEndSwitch(UBYTE *s)
{
    DUMMYUSE(s);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
        if ( AP.PreSwitchLevel <= 0 )
             MesPrint("@EndSwitch without corresponding Switch");
        else MessPreNesting(10);
        return(-1);
    }
    AP.NumPreTypes--;
    if ( AP.PreSwitchLevel <= 0 ) {
        MesPrint("@EndSwitch without corresponding Switch");
        return(-1);
    }
    M_free(AP.PreSwitchStrings[AP.PreSwitchLevel--],"pre switch string");
    return(0);
}

/*
        #] DoPreEndSwitch :
        #[ DoPreSwitch :

        There should be a string after this.
        We have to store it somewhere.
*/

int DoPreSwitch(UBYTE *s)
{
    UBYTE *t, *switchstring, **newstrings;
    int newnum, i, *newmodes;
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    SKIPBLANKS(s)
    t = s;
    while ( *s ) { if ( *s == '\\' ) s++; s++; }
```

```
    while ( s > t && ( s[-1] == ' ' || s[-1] == '\t' ) && s[-2] != '\\' ) {
        if ( s[-2] == '\\' ) s--;
        s--;
    }
    if ( *t == '"' && s > t+1 && s[-1] == '"' && s[-2] != '\\' ) {
        t++; s--; *s = 0;
    }
    else *s = 0;
    switchstring = (UBYTE *)Malloc1((s-t)+1,"case string");
    s = switchstring;
    while ( *t ) {
        if ( *t == '\\' ) t++;
        *s++ = *t++;
    }
    *s = 0;
    if ( AP.PreSwitchLevel >= AP.NumPreSwitchStrings ) {
        newnum = 2*AP.NumPreSwitchStrings;
        newstrings = (UBYTE **)Malloc1(sizeof(UBYTE *)*(newnum+1),"case strings");
        newmodes   = (int *)Malloc1(sizeof(int)*(newnum+1),"case strings");
        for ( i = 0; i < AP.NumPreSwitchStrings; i++ )
            newstrings[i] = AP.PreSwitchStrings[i];
        M_free(AP.PreSwitchStrings,"AP.PreSwitchStrings");
        for ( i = 0; i <= AP.NumPreSwitchStrings; i++ )
            newmodes[i] = AP.PreSwitchModes[i];
        M_free(AP.PreSwitchModes,"AP.PreSwitchModes");
        AP.PreSwitchStrings = newstrings;
        AP.PreSwitchModes   = newmodes;
        AP.NumPreSwitchStrings = newnum;
    }
    AP.PreSwitchStrings[++AP.PreSwitchLevel] = switchstring;
    if ( ( AP.PreSwitchLevel > 1 )
     && ( AP.PreSwitchModes[AP.PreSwitchLevel-1] != EXECUTINGPRESWITCH ) )
        AP.PreSwitchModes[AP.PreSwitchLevel] = SEARCHINGPREENDSWITCH;
    else
        AP.PreSwitchModes[AP.PreSwitchLevel] = SEARCHINGPRECASE;
    AddToPreTypes(PRETYPESWITCH);
    return(0);
}

/*
        #] DoPreSwitch :
        #[ DoPreShow :

        Print the contents of the preprocessor variables
*/

int DoPreShow(UBYTE *s)
{
    int i;
    UBYTE *name, c;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == 0 ) {
        MesPrint("%#The preprocessor variables:");
        for ( i = 0; i < NumPre; i++ ) {
            MesPrint("%d: %s = \"%s\"",i,PreVar[i].name,PreVar[i].value);
        }
    }
    else {
        while ( *s ) {
            name = s; while ( *s && *s != ' ' && *s != '\t' && *s != ',' ) s++;
            c = *s; *s = 0;
            for ( i = 0; i < NumPre; i++ ) {
                if ( StrCmp(PreVar[i].name,name) == 0 )
                    MesPrint("%d: %s = \"%s\"",i,PreVar[i].name,PreVar[i].value);
            }
            *s = c;
            while ( *s == ' ' || *s == '\t' ) s++;
        }
    }
    return(0);
}

/*
        #] DoPreShow :
        #[ DoSystem :
*/

int DoSystem(UBYTE *s)
{
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);
#ifdef WITHSYSTEM
    FLUSHCONSOLE;
    while ( *s == ' ' || *s == '\t' ) s++;
```

```
    if ( system((char *)s) ) {
        MesPrint("@System call returned with error condition");
        Terminate(-1);
    }
    return(0);
#else
    Error0("External programs not implemented on this computer/system");
    return(-1);
#endif
}

/*
        #] DoSystem :
        #[ PreLoad :

        Loads a loop or procedure into a special buffer.
        Note: The current instruction is already in the preStart buffer
*/

int PreLoad(PRELOAD *p, UBYTE *start, UBYTE *stop, int mode, char *message)
{
    UBYTE *s, *t, *top, *newbuffer, c;
    LONG i, ppsize, linenum = AC.CurrentStream->linenumber;
    int size1, size2, level, com=0, last=1, strng = 0;
    p->size = AP.pSize;
    p->buffer = (UBYTE *)Malloc1(p->size+1,message);
    top = p->buffer + p->size - 2;
    t = p->buffer; *t++ = '#';
    s = start; size1 = size2 = 0;
    while ( *s ) { s++; size1++; }
    s = stop; while ( *s ) { s++; size2++; }
    s = AP.preStart; while ( *s ) *t++ = *s++; *t++ = LINEFEED;
    level = 1;
    i = 100;
    for (;;) {
        c = GetInput();
        if ( c == ENDOFINPUT ) {
            MesPrint("@Missing %#%s, Should match line %l",stop,linenum);
            return(-1);
        }
        if ( c == AP.ComChar && last == 1 ) com = 1;
        if ( c == LINEFEED ) { last = 1; com = 0; }
        else last = 0;

        if ( ( c == '"' ) && ( com == 0 ) ) { strng ^= 1; }

        if ( ( c == '#' ) && ( com == 0 ) ) i = 0;
        else i++;

        if ( t >= top ) {
            ppsize = t - p->buffer;
            p->size *= 2;
            newbuffer = (UBYTE *)Malloc1(p->size,message);
            t = newbuffer; s = p->buffer;
            while ( --ppsize >= 0 ) *t++ = *s++;
            M_free(p->buffer,"loading do loop");
            p->buffer = newbuffer;
            top = p->buffer + p->size - 2;
        }
        *t++ = c;
        if ( strng == 0 ) {
          if ( ( i == size2 ) && ( com == 0 ) ) {
            *t = 0;
            if ( StrICmp(t-size2,(UBYTE *)(stop)) == 0 ) {
                while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
                level--;
                if ( level <= 0 ) break;
                if ( c == ENDOFINPUT ) Error1("Missing #",stop);
                *t++ = LINEFEED; *t = 0; last = 1;
            }
          }
          if ( ( i == size1 ) && mode && ( com == 0 ) ) {
            *t = 0;
            if ( StrICmp(t-size1,(UBYTE *)(start)) == 0 ) {
/*
                while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
                if ( c == ENDOFINPUT ) Error1("Missing #",stop);
*/
                level++;
            }
          }
          if ( i == 1 && t[-2] == LINEFEED ) {
            if ( c == '-' )       AC.NoShowInput = 1;
            else if ( c == '+' ) AC.NoShowInput = 0;
          }
        }
    }
```

```
    *t++ = LINEFEED;
    *t = 0;
    return(0);
}

/*
        #] PreLoad :
        #[ PreSkip :

        Skips a loop or procedure.
        Note: The current instruction is already in the preStart buffer
*/

#define SKIPBUFSIZE 20

int PreSkip(UBYTE *start, UBYTE *stop, int mode)
{
    UBYTE *s, *t, buffer[SKIPBUFSIZE+2], c;
    LONG i, linenum = AC.CurrentStream->linenumber;
    int size1, size2, level, com=0, last=1;

    t = buffer; *t++ = '#';
    s = start; size1 = size2 = 0;
    while ( *s ) { s++; size1++; }
    s = stop; while ( *s ) { s++; size2++; }
    level = 1;
    i = 0;
    for (;;) {
        c = GetInput();
        if ( c == ENDOFINPUT ) {
            MesPrint("@Missing %#%s, Should match line %l",stop,linenum);
            return(-1);
        }
        if ( c == AP.ComChar && last == 1 ) com = 1;
        if ( c == LINEFEED ) { last = 1; com = 0; i = 0; t = buffer; }
        else last = 0;
        if ( ( c == '#' ) && ( com == 0 ) ) { i = 0; t = buffer; }
        else i++;

        if ( i < SKIPBUFSIZE ) *t++ = c;
        if ( ( i == size2 ) && ( com == 0 ) ) {
            *t = 0;
            if ( StrICmp(t-size2,(UBYTE *)(stop)) == 0 ) {
                while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
                level--;
                if ( level <= 0 ) {
                    pushbackchar = LINEFEED;
                    break;
                }
                if ( c == ENDOFINPUT ) Error1("Missing #",stop);
                i = 0; t = buffer;
            }
        }
        if ( ( i == size1 ) && mode && ( com == 0 ) ) {
            *t = 0;
            if ( StrICmp(t-size1,(UBYTE *)(start)) == 0 ) {
                while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
                level++;
                i = 0; t = buffer;
            }
        }
    }
    return(0);
}

/*
        #] PreSkip :
        #[ StartPrepro :
*/

VOID StartPrepro()
{
    int **ppp;
    AP.MaxPreIfLevel = 2;
    ppp = &AP.PreIfStack;
    if ( DoubleList((VOID ***)ppp,&AP.MaxPreIfLevel,sizeof(int),
            "PreIfLevels") ) Terminate(-1);
    AP.PreIfLevel = 0; AP.PreIfStack[0] = EXECUTINGIF;

    AP.NumPreSwitchStrings = 10;
    AP.PreSwitchStrings = (UBYTE **)Malloc1(sizeof(UBYTE *)*
                            (AP.NumPreSwitchStrings+1),"case strings");
    AP.PreSwitchModes   = (int *)Malloc1(sizeof(int)*
                            (AP.NumPreSwitchStrings+1),"case strings");
    AP.PreSwitchModes[0] = EXECUTINGPRESWITCH;
    AP.PreSwitchLevel = 0;
}
```

```

/*
        #] StartPrepro :
        #[ EvalPreIf :

        Evaluates the condition in an if instruction.
        The return value is EXECUTINGIF if the condition is true.
        If it is false the returnvalue is LOOKINGFORELSE.
        An error gives a return value of -1
*/

int EvalPreIf(UBYTE *s)
{
    UBYTE *t, *u;
    int val;
    t = s;
    while ( *t ) t++;
    *t++ = ')';
    *t = 0;
    if ( ( u = PreIfEval(s,&val) ) == 0 ) return(-1);
    if ( u < t ) {
        MesPrint("@Unmatched parentheses in condition");
        return(-1);
    }
    if ( val ) return(EXECUTINGIF);
    else       return(LOOKINGFORELSE);
}

/*
        #] EvalPreIf :
        #[ PreIfEval :

        Used for recursions in the evaluation of a preprocessor if-condition.
        It determines whether the contents of () is true or false
        (or in error).
        The return value is the address of the first character after the
        closing parenthesis or null if there is an error.
        In value we find true(1) or false(0)
        We enter after the opening parenthesis.
        There are levels:
            0: orlevel:  a || b
            1: andlevel: a && b
            2: eqlevel:  a == b or a != b or a = b
            3: cmplevel: a > b or a >= b or a < b or a <= b or a >~ b etc
*/

UBYTE *PreIfEval(UBYTE *s, int *value)
{
    int orlevel = 0, andlevel = 0, eqlevel = 0, cmplevel = 0;
    int type, val;
    LONG val2;
    int ortype, orval, cmptype, cmpval, eqtype, eqval, andtype, andval;
    UBYTE *t, *eqt, *cmpt, c;
    int eqop, cmpop;
    ortype = orval = cmptype = cmpval = eqtype = eqval = andtype = andval = 0;
    eqop = cmpop = 0;
    eqt = cmpt = 0;
    *value = 0;
    while ( *s != ')' ) {
        while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
        t = s;
        s = pParseObject(s,&type,&val2);
        if ( s == 0 ) return(0);
        val = val2;
        c = *s;
        *s++ = 0;    /* in case the object is a string without " */
        while ( c == ' ' || c == '\t' || c == '\n' || c == '\r' ) {
            c = *s; *s++ = 0;
        }
        if ( *t == '"' ) t++;
        switch(c) {
            case '|':
                if ( *s != '|' ) goto illoper;
                s++;
            case ')':
                if ( cmplevel ) {
                    if ( type == 0 || cmptype == 0 ) goto illobject;
                    val = PreCmp(type,val,t,cmptype,cmpval,cmpt,cmpop);
                    type = 0;
                    cmplevel = 0;
                }
                if ( eqlevel ) {
                    val = PreEq(type,val,t,eqtype,eqval,eqt,eqop);
                    type = 0;
                    eqlevel = 0;
                }
                if ( andlevel ) {
```

```
                    if ( andtype != 0 || type != 0 ) goto illobject;
                    val &= andval;
                    andlevel = 0;
                }
                if ( orlevel ) {
                    if ( ortype != 0 || type != 0 ) goto illobject;
                    val |= orval;
                }
                if ( c == ')' ) {
                    *value = val;
                    return(s);
                }
                orlevel = 1;
                orval = val;
                ortype = type;
                break;
            case '&':
                if ( *s != '&' ) goto illoper;
                s++;
                if ( cmplevel ) {
                    if ( type == 0 || cmptype == 0 ) goto illobject;
                    val = PreCmp(type,val,t,cmptype,cmpval,cmpt,cmpop);
                    type = 0;
                    cmplevel = 0;
                }
                if ( eqlevel ) {
                    val = PreEq(type,val,t,eqtype,eqval,eqt,eqop);
                    type = 0;
                    eqlevel = 0;
                }
                if ( andlevel ) {
                    if ( andtype != 0 || type != 0 ) goto illobject;
                    val &= andval;
                }
                andlevel = 1;
                andval = val;
                andtype = type;
                break;
            case '!':
            case '=':
                if ( eqlevel ) goto illorder;
                if ( cmplevel ) {
                    if ( type == 0 || cmptype == 0 ) goto illobject;
                    val = PreCmp(type,val,t,cmptype,cmpval,cmpt,cmpop);
                    type = 0;
                    cmplevel = 0;
                }
                if ( c == '!' && *s != '=' ) goto illoper;
                if ( *s == '=' ) s++;
                if ( c == '!' ) eqop = 1;
                else eqop = 0;
                eqlevel = 1; eqt = t; eqval = val; eqtype = type;
                break;
            case '>':
            case '<':
                if ( cmplevel ) goto illorder;
                if ( c == '<' ) cmpop = -1;
                else            cmpop = 1;
                cmplevel = 1; cmpt = t; cmpval = val; cmptype = type;
                if ( *s == '=' ) {
                    s++;
                    if ( *s == '~' ) { s++; cmpop *= 4; }
                    else cmpop *= 2;
                }
                else if ( *s == '~' ) { s++; cmpop *= 3; }
                break;
            default:
                goto illoper;
        }
    }
    return(s);
illorder:
    MesPrint("@illegal order of operators");
    return(0);
illobject:
    MesPrint("@illegal object for this operator");
    return(0);
illoper:
    MesPrint("@illegal operator");
    return(0);
}

/*
        #] PreIfEval :
        #[ PreCmp :
*/

```

```
int PreCmp(int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int cmpop)
{
    if ( type == 2 || type2 == 2 || cmpop < -2 || cmpop > 2 ) {
        if ( cmpop < 0 && cmpop > -3 ) cmpop -= 2;
        if ( cmpop > 0 && cmpop <  3 ) cmpop += 2;
             if ( cmpop ==  3 ) val = StrCmp(t2,t) >  0;
        else if ( cmpop ==  4 ) val = StrCmp(t2,t) >= 0;
        else if ( cmpop == -3 ) val = StrCmp(t2,t) <  0;
        else if ( cmpop == -4 ) val = StrCmp(t2,t) <= 0;
    }
    else {
             if ( cmpop ==  1 ) val = ( val2 >  val );
        else if ( cmpop ==  2 ) val = ( val2 >= val );
        else if ( cmpop == -1 ) val = ( val2 <  val );
        else if ( cmpop == -2 ) val = ( val2 <= val );
    }
    return(val);
}

/*
        #] PreCmp :
        #[ PreEq :
*/

int PreEq(int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int eqop)
{
    UBYTE str[20];
    if ( type == 2 || type2 == 2 ) {
        if ( type  != 2 ) { NumToStr(str,val ); t  = str; }
        if ( type2 != 2 ) { NumToStr(str,val2); t2 = str; }
        if ( eqop == 1 ) val = StrCmp(t,t2) != 0;
        else             val = StrCmp(t,t2) == 0;
    }
    else {
        if ( eqop ) val = val != val2;
        else        val = val == val2;
    }
    return(val);
}

/*
        #] PreEq :
        #[ pParseObject :

        Parses a preprocessor object. We can have:
        1: a number  (type = 1)
        2: a string  (type = 2)
        3: an expression between parentheses (type = 0)
        4: a special function (type = 3)
        If the object is not a number, an expression or a special operator
        we try to interprete it as a string.
*/

UBYTE *pParseObject(UBYTE *s, int *type, LONG *val2)
{
    UBYTE *t, c;
    int sign, val = 0;
    LONG x;
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == '(' ) {
        s++;
        while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
        s = PreIfEval(s,&val);
        *type = 0;
        *val2 = val;
        return(s);
    }
    else if ( *s == '$' && s[1] == '(' ) {
        s += 2;
        while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
        s = PreIfDollarEval(s,&val);
        *type = 0; *val2 = val;
        return(s);
    }
    if ( *s == 0 ) {
illend:
        MesPrint("@illegal end of condition");
        return(0);
    }
    if ( *s == '"' ) {
        s++;
        while ( *s && *s != '"' ) {
            if ( *s == '\\' ) s++;
            s++;
        }
        if ( *s == 0 ) goto illend;
        else *s = 0;
```

```
        *type = 2;
        s++;

        while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;

        return(s);
    }
    t = s; sign = 1; x = 0;
    if ( chartype[*t] == 0 ) {  /* Special operators and strings without "" */
        do { t++; } while ( chartype[*t] <= 1 );
        if ( *t == '(' ) {
            c = *t; *t = 0;
            if ( StrICmp(s,(UBYTE *)"termsin") == 0 ) {
                UBYTE *tt;
                WORD numdol, numexp;
                *t++ = c;
                while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
                if ( *t == '$' ) {
                    t++; tt = t; while (chartype[*tt] <= 1 ) tt++;
                    c = *tt; *tt = 0;
                    if ( ( numdol = GetDollar(t) ) > 0 ) {
                        *tt = c;
                        x = TermsInDollar(numdol);
                    }
                    else {
                        MesPrint("@$%s has not (yet) been defined",t);
                        *tt = c;
                        Terminate(-1);
                    }
                }
                else {
                    tt = SkipAName(t);
                    c = *tt; *tt = 0;
                    if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
                        MesPrint("@%s has not (yet) been defined",t);
                        *tt = c;
                        Terminate(-1);
                    }
                    else {
                        *tt = c;
                        x = TermsInExpression(numexp);
                    }
                }
                while ( *tt == ' ' || *tt == '\t'
                         || *tt == '\n' || *tt == '\r' ) tt++;
                if ( *tt != ')' ) {
                    MesPrint("@Improper use of terms($var) or terms(expr)");
                    Terminate(-1);
                }
                *type = 3;
                s = tt+1;
                *val2 = x;
                return(s);
            }
            else if ( StrICmp(s,(UBYTE *)"exists") == 0 ) {
                UBYTE *tt;
                WORD numdol, numexp;
                *t++ = c;
                while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
                if ( *t == '$' ) {
                    t++; tt = t; while (chartype[*tt] <= 1 ) tt++;
                    c = *tt; *tt = 0;
                    if ( ( numdol = GetDollar(t) ) >= 0 ) { x = 1; }
                    else                                  { x = 0; }
                    *tt = c;
                }
                else {
                    tt = SkipAName(t);
                    c = *tt; *tt = 0;
                    if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) { x = 0; }
                    else { x = 1; }
                    *tt = c;
                }
                while ( *tt == ' ' || *tt == '\t'
                         || *tt == '\n' || *tt == '\r' ) tt++;
                if ( *tt != ')' ) {
                    MesPrint("@Improper use of exists($var) or exists(expr)");
                    Terminate(-1);
                }
                *type = 3;
                s = tt+1;
                *val2 = x;
                return(s);
            }
            else if ( StrICmp(s,(UBYTE *)"isnumerical") == 0 ) {
                GETIDENTITY
                UBYTE *tt;
```

```
                WORD numdol, numexp;
                *t++ = c;
                while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
                if ( *t == '$' ) {
                    t++; tt = t; while (chartype[*tt] <= 1 ) tt++;
                    c = *tt; *tt = 0;
                    if ( ( numdol = GetDollar(t) ) < 0 ) {
                        MesPrint("@$ variable in isnumerical(%s) does not exist",t);
                        Terminate(-1);
                    }
                    x = DolToLong(BHEAD numdol);
                    if ( AN.ErrorInDollar ) {
                        DOLLARS d = Dollars + numdol;
                        x = 0;
                        if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
                            if ( d->where[0] == 0 ) x = 1;
                            else if ( d->where[d->where[0]] == 0 ) {
                                if ( ABS(d->where[d->where[0]-1]) == d->where[0]-1 )
                                    x = 1;
                            }
                        }
                    }
                    else x = 1;
                    *tt = c;
                }
                else {
                    tt = SkipAName(t);
                    c = *tt; *tt = 0;
                    if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
                        MesPrint("@expression in isnumerical(%s) does not exist",t);
                        Terminate(-1);
                    }
                    x = TermsInExpression(numexp);
                    if ( x != 1 ) x = 0;
                    else {
                        WORD *term = AT.WorkPointer;
                        if ( GetFirstTerm(term,numexp) < 0 ) {
                            MesPrint("@error reading expression in isnumerical(%s)",t);
                            Terminate(-1);
                        }
                        if ( *term == ABS(term[*term-1])+1 ) x = 1;
                        else                                 x = 0;
                    }
                    *tt = c;
                }
                while ( *tt == ' ' || *tt == '\t'
                         || *tt == '\n' || *tt == '\r' ) tt++;
                if ( *tt != ')' ) {
                    MesPrint("@Improper use of isnumerical($var) or numerical(expr)");
                    Terminate(-1);
                }
                *type = 3;
                s = tt+1;
                *val2 = x;
                return(s);
            }
            else if ( StrICmp(s,(UBYTE *)("maxpowerof")) == 0 ) {
                UBYTE *tt;
                WORD numsym;
                int stype;
                *t++ = c;
                while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
                tt = SkipAName(t);
                c = *tt; *tt = 0;
                if ( ( stype = GetName(AC.varnames,t,&numsym,NOAUTO) ) == NAMENOTFOUND ) {
                    MesPrint("@%s has not (yet) been defined",t);
                    *tt = c;
                    Terminate(-1);
                }
                else if ( stype != CSYMBOL ) {
                    MesPrint("@%s should be a symbol",t);
                    *tt = c;
                    Terminate(-1);
                }
                else {
                    *tt = c;
                    x = symbols[numsym].maxpower;
                }
                while ( *tt == ' ' || *tt == '\t'
                         || *tt == '\n' || *tt == '\r' ) tt++;
                if ( *tt != ')' ) {
                    MesPrint("@Improper use of maxpowerof(symbol)");
                    Terminate(-1);
                }
                *type = 3;
                s = tt+1;
                *val2 = x;
```

```
                return(s);
            }
            else if ( StrICmp(s,(UBYTE *)("minpowerof")) == 0 ) {
                UBYTE *tt;
                WORD numsym;
                int stype;
                *t++ = c;
                while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
                tt = SkipAName(t);
                c = *tt; *tt = 0;
                if ( ( stype = GetName(AC.varnames,t,&numsym,NOAUTO) ) == NAMENOTFOUND ) {
                    MesPrint("@%s has not (yet) been defined",t);
                    *tt = c;
                    Terminate(-1);
                }
                else if ( stype != CSYMBOL ) {
                    MesPrint("@%s should be a symbol",t);
                    *tt = c;
                    Terminate(-1);
                }
                else {
                    *tt = c;
                    x = symbols[numsym].minpower;
                }
                while ( *tt == ' ' || *tt == '\t'
                         || *tt == '\n' || *tt == '\r' ) tt++;
                if ( *tt != ')' ) {
                    MesPrint("@Improper use of minpowerof(symbol)");
                    Terminate(-1);
                }
                *type = 3;
                s = tt+1;
                *val2 = x;
                return(s);
            }
            else if ( StrICmp(s,(UBYTE *)"isfactorized") == 0 ) {
                UBYTE *tt;
                WORD numdol, numexp;
                *t++ = c;
                while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
                if ( *t == '$' ) {
                    t++; tt = t; while (chartype[*tt] <= 1 ) tt++;
                    c = *tt; *tt = 0;
                    if ( ( numdol = GetDollar(t) ) > 0 ) {
                        if ( Dollars[numdol].factors != 0 ) x = 1;
                        else x = 0;
                    }
                    else {
                        MesPrint("@ %s should be the name of an expression or a $ variable",t-1);
                        Terminate(-1);
                    }
                    *tt = c;
                }
                else {
                    tt = SkipAName(t);
                    c = *tt; *tt = 0;
                    if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
                        MesPrint("@ %s should be the name of an expression or a $ variable",t);
                        Terminate(-1);
                    }
                    else {
                        if ( ( Expressions[numexp].vflags & ISFACTORIZED ) != 0 ) x = 1;
                        else x = 0;
                    }
                    *tt = c;
                }
                while ( *tt == ' ' || *tt == '\t'
                         || *tt == '\n' || *tt == '\r' ) tt++;
                if ( *tt != ')' ) {
                    MesPrint("@Improper use of isfactorized($var) or isfactorized(expr)");
                    Terminate(-1);
                }
                *type = 3;
                s = tt+1;
                *val2 = x;
                return(s);
            }
            else if ( StrICmp(s,(UBYTE *)"isdefined") == 0 ) {
                UBYTE *tt;
                *t++ = c;
                while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
                tt = SkipAName(t);
                c = *tt; *tt = 0;
                if ( GetPreVar(t,WITHOUTERROR) != 0 ) x = 1;
                else x = 0;
                *tt = c;
                while ( *tt == ' ' || *tt == '\t'
```

```
                        || *tt == '\n' || *tt == '\r' ) tt++;
                if ( *tt != ')' ) {
                    MesPrint("@Improper use of isdefined(var)");
                    Terminate(-1);
                }
                *type = 3;
                s = tt+1;
                *val2 = x;
                return(s);
            }
            else *t = c;
        }
        else if ( *t == '=' || *t == '<' || *t == '>' || *t == '!'
        || *t == ')' || *t == ' ' || *t == '\t' || *t == 0 || *t == '\n' ) {
            *val2 = 0;
            *type = 2;
            return(t);
        }
        else {
            MesPrint("@Illegal use of string in preprocessor condition: %s",s);
            Terminate(-1);
        }
    }
    while ( *t == '-' || *t == '+' || *t == ' ' || *t == '\t' ) {
        if ( *t == '-' ) sign = -sign;
        t++;
    }
    while ( chartype[*t] == 1 ) { x = 10*x + *t++ - '0'; }
    while ( *t == ' ' || *t == '\t' ) t++;
    if ( chartype[*t] == 8 || *t == ')' || *t == '=' || *t == 0 ) {
        *val2 = sign > 0 ? x: -x;
        *type = 1;
        return(t);
    }
    while ( chartype[*t] != 8 && *t != ')' && *t != '=' && *t ) t++;
    while ( ( t > s ) && ( t[-1] == ' ' || t[-1] == '\t' ) ) t--;
    *type = 2;
    *val2 = val;
    return(t);
}

/*
        #] pParseObject :
        #[ PreCalc :

        To be called when a { is encountered.
        Action: read first till matching }. This is to be stored.
        Next we look whether this is a set or whether it can be
        evaluated. If it is a set we consider it as a new stream.
        The stream will have to be deallocated when read completely.
        If it is to be evaluated we do that and put the result in
        a stream.
*/

UBYTE *PreCalc()
{
    UBYTE *buff, *s = 0, *t, *newb, c;
    int size, i, n, parlevel = 0, bralevel = 0;
    LONG answer;
    size = n = 0;
    buff = 0; c = '{';
    for (;;) {
        if ( n >= size ) {
            if ( size == 0 ) size = 72;
            else size *= 2;
            if ( ( newb = (UBYTE *)Malloc1(size+2,"{}") ) == 0 ) return(0);
            s = newb;
            if ( buff ) {
                i = n;
                t = buff;
                NCOPYB(s,t,i);
                M_free(buff,"pre calc buffer");
            }
            else s = newb;
            buff = newb;
        }
        *s++ = c; n++;
        c = GetChar(0);
        if ( c == 0 ) {
            Error0("Unmatched {}");
            M_free(buff,"precalc buffer");
            return(0);
        }
        else if ( c == '{' ) { bralevel++; }
        else if ( c == '}' ) {
            if ( --bralevel < 0 ) { *s++ = c; *s = 0; break; }
        }
```

```
        else if ( c == '(' ) { parlevel++; }
        else if ( c == ')' ) {
            if ( --parlevel < 0 ) { *s++ = c; *s = 0; goto setstring; }
        }
        else if ( chartype[c] != 1 && chartype[c] != 5
        && chartype[c] != 6 && c != '!' && c != '&'
        && c != '|' && c != '\\' ) { *s++ = c; *s = 0; goto setstring; }
    }
    if ( parlevel > 0 ) goto setstring;
/*
    Try now to evaluate the string.
    If it works, copy the resulting value back into buff as a string.
*/
    answer = 0;
    if ( PreEval(buff+1,&answer) == 0 ) goto setstring;
    t = buff + size;
    s = buff;
    if ( answer < 0 ) { *s++ = '-'; answer = -answer; }
    n = 0;
    do {
        *--t = ( answer % 10 ) + '0';
        answer /= 10;
        n++;
    } while ( answer > 0 );
    NCOPYB(s,t,n);
    *s = 0;
setstring:;
/*
    Open a stream that contains the current string.
    Mark it to be removed after termination.
*/
    if ( OpenStream(buff,PRECALCSTREAM,0,PRENOACTION) == 0 ) return(0);
    return(buff);
}

/*
        #] PreCalc :
        #[ PreEval :

        Operations are:
        +, -, *, /, %, &, |, ^, !,  ^% (postfix 2log), ^/ (postfix sqrt)
*/

UBYTE *PreEval(UBYTE *s, LONG *x)
{
    LONG y, z, a;
    int tobemultiplied, tobeadded = 1, expsign, i;
    UBYTE *t;
    *x = 0; a = 1;
    while ( *s == ' ' || *s == '\t' ) s++;
    for(;;){
        if ( *s == '+' || *s == '-' ) {
            if ( *s == '-' ) tobeadded = -1;
            else tobeadded = 1;
            s++;
            while ( *s == '-' || *s == '+' || *s == ' ' || *s == '\t' ) {
                if ( *s == '-' ) tobeadded = -tobeadded;
                s++;
            }
        }
        tobemultiplied = 0;
        for(;;){
            while ( *s == ' ' || *s == '\t' ) s++;
            if ( *s <= '9' && *s >= '0' ) {
                ParseNumber(y,s)
            }
            else if ( *s == '(' || *s == '{' ) {
                if ( ( t = PreEval(s+1,&y) ) == 0 ) return(0);
                s = t;
            }
            else return(0);
            while ( *s == ' ' || *s == '\t' ) s++;
            expsign = 1;
            while ( *s == '^' || *s == '!' ) {
                s++;
                if ( s[-1] == '!' ) {   /* factorial of course */
                    while ( *s == ' ' || *s == '\t' ) s++;
                    if ( y < 0 ) {
                        MesPrint("@Negative value in preprocessor factorial: %l",y);
                        return(0);
                    }
                    else if ( y == 0 ) y = 1;
                    else if ( y > 1 ) {
                        z = y-1;
                        while ( z > 0 ) { y = y*z; z--; }
                    }
                    continue;
```

```
                }
                else if ( *s == '%' ) { /* ^% is postfix 2log */
                    s++;
                    while ( *s == ' ' || *s == '\t' ) s++;
                    z = y;
                    if ( z <= 0 ) {
                        MesPrint("@Illegal value in preprocessor logarithm: %l",z);
                        return(0);
                    }
                    y = 0; z »= 1;
                    while ( z ) { y++; z »= 1; }
                    continue;
                }
                else if ( *s == '/' ) { /* ^/ is postfix sqrt */
                    LONG yy, zz;
                    s++;
                    while ( *s == ' ' || *s == '\t' ) s++;
                    z = y;
                    if ( z <= 0 ) {
                        MesPrint("@Illegal value in preprocessor square root: %l",z);
                        return(0);
                    }
                    if ( z > 8 ) {      /* Very crude integer square root */
                        zz = z;
                        yy = 0; zz »= 1;
                        while ( zz ) { yy++; zz »= 1; }
                        zz = z » (yy/2); i = 10; y = 0;
                        do {
                            yy = zz/2 + z/(2*zz); i--;
                            if ( y == yy ) break;
                            y = zz; zz = yy;
                        } while ( y != yy && i > 0 );
                        while ( y*y < z ) y++;
                        while ( y*y > z ) y--;
                    }
                    else if ( z >= 4 ) y = 2;
                    else if ( z == 0 ) y = 0;
                    else y = 1;
                    continue;
                }
                while ( *s == ' ' || *s == '\t' ) s++;
                while ( *s == '-' || *s == '+' || *s == ' ' || *s == '\t' ) {
                    if ( *s == '-' ) expsign = -expsign;
                }
                if ( *s <= '9' && *s >= '0' ) {
                    ParseNumber(z,s)
                }
                else if ( *s == '(' || *s == '{' ) {
                    if ( ( t = PreEval(s+1,&z) ) == 0 ) return(0);
                    s = t;
                }
                else return(0);
                while ( *s == ' ' || *s == '\t' ) s++;
                y = iexp(y,(int)z);
            }
            if ( tobemultiplied == 0 ) {
                if ( expsign < 0 ) a = 1/y;
                else a = y;
            }
            else {
                if ( tobemultiplied > 2 && expsign != 1 ) {
                    MesPrint("&Incorrect use of ^ with & or |. Use brackets!");
                    Terminate(-1);
                }
                tobemultiplied *= expsign;
                if ( tobemultiplied == 1 ) a *= y;
                else if ( tobemultiplied == 3 ) a &= y;
                else if ( tobemultiplied == 4 ) a |= y;
                else {
                    if ( y == 0 || tobemultiplied == -2 ) {
                        MesPrint("@Division by zero in preprocessor calculator");
                        Terminate(-1);
                    }
                    if ( tobemultiplied == 2 ) a %= y;
                    else a /= y;
                }
            }
            if ( *s == '%' ) tobemultiplied = 2;
            else if ( *s == '*' ) tobemultiplied = 1;
            else if ( *s == '/' ) tobemultiplied = -1;
            else if ( *s == '&' ) tobemultiplied = 3;
            else if ( *s == '|' ) tobemultiplied = 4;
            else {
                if ( tobeadded >= 0 ) *x += a;
                else *x -= a;
                if ( *s == ')' || *s == '}' ) return(s+1);
                else if ( *s == '-' || *s == '+' ) { tobeadded = 1; break; }
```

```
                else return(0);
            }
            s++;
        }
    }
/*  return(0); */
}

/*
        #] PreEval :
        #[ AddToPreTypes :
*/

void AddToPreTypes(int type)
{
    if ( AP.NumPreTypes >= AP.MaxPreTypes ) {
        int i, *newlist = (int *)Malloc1(sizeof(int)*(2*AP.MaxPreTypes+1)
                        ,"preprocessor type lists");
        for ( i = 0; i <= AP.MaxPreTypes; i++ ) newlist[i] = AP.PreTypes[i];
        M_free(AP.PreTypes,"preprocessor type lists");
        AP.PreTypes = newlist;
        AP.MaxPreTypes = 2*AP.MaxPreTypes;
    }
    AP.PreTypes[++AP.NumPreTypes] = type;
}

/*
        #] AddToPreTypes :
        #[ MessPreNesting :
*/

void MessPreNesting(int par)
{
    MesPrint("@(%d)Illegal nesting of %#if, %#do, %#procedure and/or %#switch",par);
}

/*
        #] MessPreNesting :
        #[ DoPreAddSeparator :

        Preprocessor directives "addseparator" and "rmseparator" add/remove
        separator characters used to separate function arguments.
        Example:

            #define QQ "a|g|a"
            #addseparator %
            *Comma must be quoted!:
            #rmseparator ","
            #rmseparator |
            #call H(a,a%`QQ')

        Characters ' ', '\t' and '"' are ignored!
*/

int DoPreAddSeparator(UBYTE *s)
{
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    for(;*s != '\0';s++){
        while ( *s == ' ' || *s == '\t' || *s == '"') s++;
        /* Todo:
        if ( set_in(*s,invalidseparators) ) {
            MesPrint("@Invalid separator specified");
            return(-1);
        }
        */
        set_set(*s,AC.separators);
    }
    return(0);
}

/*
        #] DoPreAddSeparator :
        #[ DoPreRmSeparator :

        See commentary with DoPreAddSeparator

        Characters ' ', '\t' and '"' are ignored!
*/
int DoPreRmSeparator(UBYTE *s)
{
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    for(;*s != '\0';s++){
        while ( *s == ' ' || *s == '\t' || *s == '"') s++;
        set_del(*s,AC.separators);
    }
```

```
    return(0);
}

/*
        #] DoPreRmSeparator :
        #[ DoExternal:

        #external ["prevar"] command
*/
int DoExternal(UBYTE *s)
{
#ifdef WITHEXTERNALCHANNEL
    UBYTE *prevar=0;
    int externalD= 0;
#else
    DUMMYUSE(s);
#endif
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);

#ifdef WITHEXTERNALCHANNEL
    while ( *s == ' ' || *s == '\t' ) s++;
    if(*s == '"'){/*prevar to store the descriptor is defined*/
        prevar=++s;

        if ( chartype[*s] == 0 )for(;*s != '"'; s++)switch(chartype[*s]){
            case 10:/*'\0' fits here*/
                MesPrint("@Can't finde closing \"");
                Terminate(-1);
            case 0:case 1: continue;
            default:
                break;
        }
        if(*s != '"'){
                MesPrint("@Illegal name of preprocessor variable to store external channel");
                return(-1);
      }
      *s='\0';
        for(s++; *s == ' ' || *s == '\t'; s++);
    }

    if(*s == '\0'){
        MesPrint("@Illegal external command");
        return(-1);
    }
    /*here s is a command*/
   /*See the file extcmd.c*/
    /*[08may2006 mt]:*/
    externalD=openExternalChannel(
                s,
                AX.daemonize,
                AX.shellname,
                AX.stderrname);
    /*:[08may2006 mt]*/
    if(externalD<1){/*error?*/
        /*Not quite correct - terminate the program on error:*/
        Error1("Can't start external program",s);
        return(-1);
    }
   /*Now external command runs.*/

   if(prevar){/*Store the external channel descriptor in the provided variable:*/
        UBYTE buf[21];/* 64/Log_2[10] = 19.3, so this is enough forever...*/
        NumToStr(buf,externalD);
        if ( PutPreVar(prevar,buf,0,1) < 0 ) return(-1);
    }

    AX.currentExternalChannel=externalD;
    /*[08may2006 mt]:*/
    if(AX.currentPrompt!=0){/*Change default terminator*/
        if(setTerminatorForExternalChannel(  (char *)AX.currentPrompt)){
            MesPrint("@Prompt is too long");
            return(-1);
        }
    }
    setKillModeForExternalChannel(AX.killSignal,AX.killWholeGroup);
    /*:[08may2006 mt]*/
    return(0);
#else /*ifdef WITHEXTERNALCHANNEL*/
    Error0("External channel: not implemented on this computer/system");
    return(-1);
#endif /*ifdef WITHEXTERNALCHANNEL ... else*/
}

/*
        #] DoExternal:
```

```
        #[ DoPrompt:
            #prompt string
*/

int DoPrompt(UBYTE *s)
{
#ifndef WITHEXTERNALCHANNEL
    DUMMYUSE(s);
#endif
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);

#ifdef WITHEXTERNALCHANNEL
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( AX.currentPrompt )
            M_free(AX.currentPrompt,"external channel prompt");
    if ( *s == '\0' )
        AX.currentPrompt = (UBYTE *)strDupl((UBYTE *)"","external channel prompt");
    else
        AX.currentPrompt = strDupl(s,"external channel prompt");
    if(  setTerminatorForExternalChannel( (char *)AX.currentPrompt) > 0  ){
        MesPrint("@Prompt is too long");
        return(-1);
    }
    /*else: if 0, ok; if -1, there is no current channel-ok, just prompt is stored.*/
    return(0);
#else /*ifdef WITHEXTERNALCHANNEL*/
    Error0("External channel: not implemented on this computer/system");
    return(-1);
#endif /*ifdef WITHEXTERNALCHANNEL ... else*/
}
/*
        #] DoPrompt:
        #[ DoSetExternal:
            #setexternal n
*/

int DoSetExternal(UBYTE *s)
{
#ifdef WITHEXTERNALCHANNEL
    int n=0;
#else
    DUMMYUSE(s);
#endif
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);

#ifdef WITHEXTERNALCHANNEL
    while ( *s == ' ' || *s == '\t' ) s++;
    while ( chartype[*s] == 1 ) { n = 10*n + *s++ - '0'; }
    while ( *s == ' ' || *s == '\t' ) s++;
    if(*s!='\0'){
        MesPrint("@setexternal: number expected");
        return(-1);
    }
    if(selectExternalChannel(n)<0){
        MesPrint("@setexternal: invalid number");
        return(-1);
    }
    AX.currentExternalChannel=n;
    return(0);
#else /*ifdef WITHEXTERNALCHANNEL*/
    Error0("External channel: not implemented on this computer/system");
    return(-1);
#endif /*ifdef WITHEXTERNALCHANNEL ... else*/
}
/*
        #] DoSetExternal:
        #[ DoSetExternalAttr:
*/

static FORM_INLINE UBYTE *pickupword(UBYTE *s)
{

    for(;*s>' ';s++)switch(*s){
        case '=':
        case ',':
        case ';':
            return(s);
    }/*for(;*s>' ';s++)switch(*s)*/
    return(s);
}
/*Returns 0 if the first string (case insensitively) equal to
  the beginning of the second string (of length n):
*/
static inline int strINCmp(UBYTE *a, UBYTE *b, int n)
```

```
{
    for(;n>0;n--)if(tolower(*a++)!=tolower(*b++))
        return(1);
    return(*a != '\0');
}

#define KILL "kill"
#define KILLALL "killall"
#define DAEMON "daemon"
#define SHELL "shell"
#define STDERR "stderr"

#define TRUE_EXPR "true"
#define FALSE_EXPR "false"
#define NOSHELL "noshell"
#define TERMINAL "terminal"

/*
    Expects comma-separated list of pairs name=value
*/
int DoSetExternalAttr(UBYTE *s)
{
#ifdef WITHEXTERNALCHANNEL
    int lnam,lval;
    UBYTE *nam,*val;
#else
    DUMMYUSE(s);
#endif
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);

#ifdef WITHEXTERNALCHANNEL
    do{
        /*Read the name:*/
        while ( *s == ' ' || *s == '\t' ) s++;
        s=pickupword(nam=s);
        lnam=s-nam;
        while ( *s == ' ' || *s == '\t' ) s++;
        if(*s++!='='){
            MesPrint("@External channel:'=' expected instead of %s",s-1);
            return(-1);
        }
        /*Read the value:*/
        while ( *s == ' ' || *s == '\t' ) s++;
        val=s;

        for(;;){
            UBYTE *m;
            s=pickupword(s);
            m=s;
            while ( *s == ' ' || *s == '\t' ) s++;
         if( (*s == ',')||(*s == '\n')||(*s == ';')||(*s == '\0') ){
                s=m;
                break;
            }
        }/*for(;;)*/

        lval=s-val;
        while ( *s == ' ' || *s == '\t' ) s++;

        if(strINCmp((UBYTE *)SHELL,nam,lnam)==0){
            if(AX.shellname!=NULL)
                M_free(AX.shellname,"external channel shellname");
            if(strINCmp((UBYTE *)NOSHELL,val,lval)==0)
                AX.shellname=NULL;
            else{
                UBYTE *ch,*b;
                b=ch=AX.shellname=Malloc1(lval+1,"external channel shellname");
                while(ch-b<lval)
                    *ch++=*val++;
                *ch='\0';
            }
        }else if(strINCmp((UBYTE *)DAEMON,nam,lnam)==0){
            if(strINCmp((UBYTE *)TRUE_EXPR,val,lval)==0)
                AX.daemonize = 1;
            else if(strINCmp((UBYTE *)FALSE_EXPR,val,lval)==0)
                AX.daemonize = 0;
            else{
                MesPrint("@External channel:true or false expected for %s",DAEMON);
                return(-1);
            }
        }else   if(strINCmp((UBYTE *)KILLALL,nam,lnam)==0){
            if(strINCmp((UBYTE *)TRUE_EXPR,val,lval)==0)
                AX.killWholeGroup = 1;
            else if(strINCmp((UBYTE *)FALSE_EXPR,val,lval)==0)
                AX.killWholeGroup = 0;
```

```
            else{
                MesPrint("@External channel: true or false expected for %s",KILLALL);
                return(-1);
            }
        }else   if(strINCmp((UBYTE *)KILL,nam,lnam)==0){
            int i,n=0;
            for(i=0;i<lval;i++)
                if( *val>='0' && *val<= '9' )
                    n = 10*n + *val++  - '0';
                else{
                    MesPrint("@External channel: number expected for %s",KILL);
                    return(-1);
                }
                AX.killSignal=n;
        }else   if(strINCmp((UBYTE *)STDERR,nam,lnam)==0){
            if( AX.stderrname != NULL ) {
                M_free(AX.stderrname,"external channel stderrname");
            }
            if(strINCmp((UBYTE *)TERMINAL,val,lval)==0)
                AX.stderrname = NULL;
            else{
                UBYTE *ch,*b;
                b=ch=AX.stderrname=Malloc1(lval+1,"external channel stderrname");
                while(ch-b<lval)
                    *ch++=*val++;
                *ch='\0';
            }
        }else{
            nam[lnam+1]='\0';
            MesPrint("@External channel: unrecognized attribute",nam);
            return(-1);
        }
    }while(*s++ == ',');
    if(  (*(s-1)>' ')&&(*(s-1)!=';')  ){
        MesPrint("@External channel: syntax error: %s",s-1);
        return(-1);
    }
   return(0);
#else /*ifdef WITHEXTERNALCHANNEL*/
    Error0("External channel: not implemented on this computer/system");
    return(-1);
#endif /*ifdef WITHEXTERNALCHANNEL ... else*/
}
/*
        #] DoSetExternalAttr:
        #[ DoRmExternal:
            #rmexternal [n] (if 0, close all)
*/

int DoRmExternal(UBYTE *s)
{
#ifdef WITHEXTERNALCHANNEL
    int n = -1;
#else
    DUMMYUSE(s);
#endif
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);

#ifdef WITHEXTERNALCHANNEL
    while ( *s == ' ' || *s == '\t' ) s++;
    if( chartype[*s] == 1 ){
        for(n=0; chartype[*s] == 1 ; s++) { n = 10*n + *s - '0'; }
        while ( *s == ' ' || *s == '\t' ) s++;
    }
    if(*s!='\0'){
        MesPrint("@rmexternal: invalid number");
        return(-1);
    }
    switch(n){
        case 0:/*Close all opened channels*/
            closeAllExternalChannels();
            AX.currentExternalChannel=0;
            /*Do not clean AX.currentPrompt!*/
            return(0);
        case -1:/*number is not specified - try current*/
            n=AX.currentExternalChannel;
            /*No break!*/
        default:
            closeExternalChannel(n);/*No reaction for possible error*/
    }
    if (n == AX.currentExternalChannel)/*cleaned up by closeExternalChannel()*/
        AX.currentExternalChannel=0;
    return(0);
#else /*ifdef WITHEXTERNALCHANNEL*/
    Error0("External channel: not implemented on this computer/system");
```

```
    return(-1);
#endif /*ifdef WITHEXTERNALCHANNEL ... else*/

}
/*
        #] DoRmExternal:
        #[ DoFromExternal :
                #fromexternal
                    is used to read the text from the running external
                    program, the synthax is similar to the #include
                    directive.
                #fromexternal "varname"
                    is used to read the text from the running external
                    program into the preprocessor variable varname.
                    directive.
                #fromexternal "varname" maxlength
                    is used to read the text from the running external
                    program into the preprocessor variable varname.
                    directive. Only first maxlength characters are
                    stored.

                    FORM continues to read the running external
                    program output until the extrenal program outputs a
                    prompt.

*/

int DoFromExternal(UBYTE *s)
{
#ifdef WITHEXTERNALCHANNEL
    UBYTE *prevar=0;
    int lbuf=-1;
    int withNoList=AC.NoShowInput;
    int oldpreassignflag;
#else
    DUMMYUSE(s);
#endif
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);
#ifdef WITHEXTERNALCHANNEL

    FLUSHCONSOLE;

    while ( *s == ' ' || *s == '\t' ) s++;
    /*[17may2006 mt]:*/
    if ( *s == '-' || *s == '+' ) {
        if ( *s == '-' )
            withNoList = 1;
        else
            withNoList = 0;
        s++;
        while ( *s == ' ' || *s == '\t' ) s++;
    }/*if ( *s == '-' || *s == '+' )*/
    /*:[17may2006 mt]*/
    /*[02feb2006 mt]:*/
    if(*s == '"'){/*prevar to store the output is defined*/
        prevar=++s;

        if ( *s=='$' || chartype[*s] == 0 )for(;*s != '"'; s++)switch(chartype[*s]){
            case 10:/*'\0' fits here*/
                MesPrint("@Can't finde closing \"");
                Terminate(-1);
            case 0:case 1: continue;
            default:
                break;
        }
        if(*s != '"'){
                MesPrint("@Illegal name to store output of external channel");
                return(-1);
      }
      *s='\0';
        for(s++; *s == ' ' || *s == '\t'; s++);
    }/*if(*s == '"')*/

    if(*s != '\0'){
        if( chartype[*s] == 1 ){
            for(lbuf=0; chartype[*s] == 1 ; s++) { lbuf = 10*lbuf + *s - '0'; }
            while ( *s == ' ' || *s == '\t' ) s++;
        }
        if( (*s!='\0')||(lbuf<0) ){
            MesPrint("@Illegal buffer length in fromexternal");
            return(-1);
        }
    }/*if(*s != '\0')*/
    /*:[02feb20006 mt]*/
    if(getCurrentExternalChannel()!=AX.currentExternalChannel)
```

```
        /*[08may20006 mt]:*/
        /*selectExternalChannel(AX.currentExternalChannel);*/
        if(selectExternalChannel(AX.currentExternalChannel)){
            MesPrint("@No current external channel");
            return(-1);
        }
        /*:[08may20006 mt]*/

    /*[02feb2006 mt]:*/
    if(prevar!=0){/*The result must be stored into preprovar*/
      UBYTE *buf;
        int cc = 0;
        if(lbuf == -1){/*Unlimited buffer, everything must be stored*/
            int i;
            buf=Malloc1( (lbuf=255)+1,"Fromexternal");
            /*[18may20006 mt]:*/
            /*for(i=0;(cc=getcFromExtChannel())!=EOF;i++){*/
            /* May 2006: now getcFromExtChannelOk returns EOF while
                getcFromExtChannelFailure returns -2 (see comments in
                exctcmd.c):*/
            for(i=0;(cc=getcFromExtChannel())>0;i++){
            /*:[18may20006 mt]*/
                if(i==lbuf){
                    int j;
                    UBYTE *tmp=Malloc1( (lbuf*=2)+1,"Fromexternal");
                    for(j=0;j<i;j++)tmp[j]=buf[j];
                    M_free(buf,"Fromexternal");
                    buf=tmp;
                }
                buf[i]=(UBYTE)(cc);
            }/*for(i=0;(cc=getcFromExtChannel())>0;i++)*/
            /*[18may20006 mt]:*/
         if(cc == -2){
                MesPrint("@No current external channel");
                return(-1);
            }
            lbuf=i;
            /*:[18may20006 mt]*/
            buf[i]='\0';
        }else{/*Fixed buffer, only lbuf chars must be stored*/
            int i;
            buf=Malloc1(lbuf+1,"Fromexternal");
            for(i=0; i<lbuf;i++){
            /*[18may20006 mt]:*/
                /*if( (cc=getcFromExtChannel())==EOF )*/
                /* May 2006: now getcFromExtChannelOk returns EOF while
                    getcFromExtChannelFailure returns -2 (see comments in
                    exctcmd.c):*/
                if( (cc=getcFromExtChannel())<1 )
            /*:[18may20006 mt]*/
                    break;
                buf[i]=(UBYTE)(cc);
            }
            buf[i]='\0';
            /*[18may20006 mt]:*/
            /*if(cc!=EOF)
                while(getcFromExtChannel()!=EOF);*//*Eat the rest*/
            /* May 2006: now getcFromExtChannelOk returns EOF while
                getcFromExtChannelFailure returns -2 (see comments in
                exctcmd.c):*/
            if(cc>0)
                while(getcFromExtChannel()>0);/*Eat the rest*/
            else if(cc == -2){
                MesPrint("@No current external channel");
                return(-1);
            }
            /*:[18may20006 mt]*/
        }
        /*[18may20006 mt]:*/
        if(*prevar == '$'){/*Put the answer to the dollar variable*/
            int oldNumPotModdollars = NumPotModdollars;
#ifdef WITHMPI
            WORD oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
            AC.RhsExprInModuleFlag = 0;
#endif
            /*Here lbuf is the actual length of buf!*/
            /*"prevar=buf'\0'":*/
            UBYTE *pbuf=Malloc1(StrLen(prevar)+1+lbuf+1,"Fromexternal to dollar");
            UBYTE *c=pbuf;
            UBYTE *b=prevar;
            while(*b!='\0'){*c++ = *b++;}
            *c++='=';
            b=buf;
            while(  (*c++=*b++)!='\0'  );
            oldpreassignflag = AP.PreAssignFlag;
            AP.PreAssignFlag = 1;
            if ( ( cc = CompileStatement(pbuf) ) || ( cc = CatchDollar(0) ) ) {
```

```
                Error1("External channel: can't asign output to dollar variable ",prevar);
            }
            AP.PreAssignFlag = oldpreassignflag;
            NumPotModdollars = oldNumPotModdollars;
#ifdef WITHMPI
            AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
#endif
            M_free(pbuf,"Fromexternal to dollar");
        }else{
            cc = PutPreVar(prevar, buf, 0, 1) < 0;
        }
        /*:[18may20006 mt]*/
        M_free(buf,"Fromexternal");
        if ( cc ) return(-1);
        return(0);
    }
    /*:[02feb2006 mt]*/
    if ( OpenStream(s,EXTERNALCHANNELSTREAM,0,PRENOACTION) == 0 ) return(-1);
    /*[17may2006 mt]:*/
    AC.NoShowInput = withNoList;
    /*:[17may2006 mt]*/
    return(0);
#else
    Error0("External channel: not implemented on this computer/system");
    return(-1);
#endif
}

/*
        #] DoFromExternal :
        #[ DoToExternal :
            #toexetrnal
*/

#ifdef WITHEXTERNALCHANNEL

/*A wrapper to writeBufToExtChannel, see the file extcmd.c:*/
LONG WriteToExternalChannel(int handle, UBYTE *buffer, LONG size)
{
    /*ATT! handle is not used! Actual output is performed to
        the current external channel, see extcmd.c!*/
    DUMMYUSE(handle);
    if(writeBufToExtChannel((char*)buffer,size))
        return(-1);
    return(size);
}
#endif /*ifdef WITHEXTERNALCHANNEL*/

int DoToExternal(UBYTE *s)
{
#ifdef WITHEXTERNALCHANNEL
   HANDLERS h;
   LONG (*OldWrite)(int handle, UBYTE *buffer, LONG size) = WriteFile;
    int ret=-1;
#else
    DUMMYUSE(s);
#endif
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    if ( AP.preError ) return(0);
#ifdef WITHEXTERNALCHANNEL

    h.oldsilent=AM.silent;
    h.newlogonly = h.oldlogonly = AM.FileOnlyFlag;
    h.newhandle = h.oldhandle = AC.LogHandle;
    h.oldprinttype = AO.PrintType;

    WriteFile=&WriteToExternalChannel;

    while ( *s == ' ' || *s == '\t' ) s++;

    if(AX.currentExternalChannel==0){
        MesPrint("@No current external channel");
        goto DoToExternalReady;
    }

    if(getCurrentExternalChannel()!=AX.currentExternalChannel)
        selectExternalChannel(AX.currentExternalChannel);

    ret=writeToChannel(EXTERNALCHANNELOUT,s,&h);
    DoToExternalReady:
        WriteFile=OldWrite;
        return(ret);
#else /*ifdef WITHEXTERNALCHANNEL*/
    Error0("External channel: not implemented on this computer/system");
    return(-1);
#endif /*ifdef WITHEXTERNALCHANNEL ... else*/
```

```

}

/*
        #] DoToExternal :
        #[ defineChannel :
*/

UBYTE *defineChannel(UBYTE *s, HANDLERS *h)
{
    UBYTE *name,*to;

    if ( *s != '<' )
        return(s);

    s++;
    name = to = s;
    while ( *s && *s != '>' ) {
        if ( *s == '\\' ) s++;
        *to++ = *s++;
    }
    if ( *s == 0 ) {
        MesPrint("@Improper termination of filename");
        return(0);
    }
    s++;
    *to = 0;
    if ( *name ) {
        h->newhandle = GetChannel((char *)name);
        h->newlogonly = 1;
    }
    else if ( AC.LogHandle >= 0 ) {
        h->newhandle = AC.LogHandle;
        h->newlogonly = 1;
    }
    return(s);
}

/*
        #] defineChannel :
        #[ writeToChannel :
*/

int writeToChannel(int wtype, UBYTE *s, HANDLERS *h)
{
    UBYTE *to, *fstring, *ss, *sss, *s1, c, c1;
    WORD  num, number, nfac;
    UBYTE Out[MAXLINELENGTH+14], *stopper;
    int nosemi, i;

/*
    Now determine the format string
*/
    while ( *s == ',' || *s == ' ' ) s++;
    if ( *s != '"' ) {
        MesPrint("@No format string present");
        return(-1);
    }
    s++; fstring = to = s;
    while ( *s ) {
        if ( *s == '\\' ) {
            s++;
            if ( *s == '\\' ) {
                *to++ = *s++;
                if ( *s == '\\' ) *to++ = *s++;
            }
            else if ( *s == '"' ) *to++ = *s++;
            else { *to++ = '\\'; *to++ = *s++; }
        }
        else if ( *s == '"' ) break;
        else *to++ = *s++;
    }
    if ( *s != '"' ) {
        MesPrint("@No closing \" in format string");
        return(-1);
    }
    *to = 0; s++;
    if ( AC.LineLength > 20 && AC.LineLength <= MAXLINELENGTH ) stopper = Out + AC.LineLength;
    else stopper = Out + MAXLINELENGTH;
    to = Out;
/*
    s points now at the list of objects (if any)
    we can start executing the format string.
*/
    AM.silent = 0;
    AC.LogHandle = h->newhandle;
    AM.FileOnlyFlag = h->newlogonly;
```

```
    if ( h->newhandle >= 0 ) {
        AO.PrintType |= PRINTLFILE;
    }
    while ( *fstring ) {
        if ( to >= stopper ) {
            if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
                *to++ = '&';
            }
            num = to - Out;
            WriteString(wtype,Out,num);
            to = Out;
            if ( AC.OutputMode == FORTRANMODE
             || AC.OutputMode == PFORTRANMODE ) {
                number = 7;
                for ( i = 0; i < number; i++ ) *to++ = ' ';
                to[-2] = '&';
            }
        }
        if ( *fstring == '\\' ) {
            fstring++;
            if ( *fstring == 'n' ) {
                num = to - Out;
                WriteString(wtype,Out,num);
                to = Out;
                fstring++;
            }
            else if ( *fstring == 't' ) { *to++ = '\t'; fstring++; }
            else if ( *fstring == 'b' ) { *to++ = '\\'; fstring++; }
            else *to++ = *fstring++;
        }
        else if ( *fstring == '%' ) {
            fstring++;
            if ( *fstring == 'd' ) {
                int sign,dig;
                number = -1;
donumber:
                while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
                sign = 1;
                while ( *s == '+' || *s == '-' ) {
                    if ( *s == '-' ) sign = -sign;
                    s++;
                }
                dig = 0; ss = s; if ( sign < 0 ) { ss--; *ss = '-'; dig++; }
                while ( *s >= '0' && *s <= '9' ) { s++; dig++; }
                if ( number < 0 ) {
                    while ( ss < s ) {
                        if ( to >= stopper ) {
                            num = to - Out;
                            WriteString(wtype,Out,num);
                            to = Out;
                        }
                        if ( *ss == '\\' ) ss++;
                        *to++ = *ss++;
                    }
                }
                else {
                    if ( number < dig ) { dig = number; ss = s - dig; }
                    while ( number > dig ) {
                        if ( to >= stopper ) {
                            num = to - Out;
                            WriteString(wtype,Out,num);
                            to = Out;
                        }
                        *to++ = ' '; number--;
                    }
                    while ( ss < s ) {
                        if ( to >= stopper ) {
                            num = to - Out;
                            WriteString(wtype,Out,num);
                            to = Out;
                        }
                        if ( *ss == '\\' ) ss++;
                        *to++ = *ss++;
                    }
                }
                fstring++;
            }
            else if ( *fstring == '$' ) {
                UBYTE *dolalloc;
                number = AO.OutSkip;
dodollar:
                while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
                if ( AC.OutputMode == FORTRANMODE
                 || AC.OutputMode == PFORTRANMODE ) {
                    number = 7;
                }
                if ( *s != '$' ) {
```

```
nodollar:           MesPrint("@$-variable expected in #write instruction");
                    AM.FileOnlyFlag = h->oldlogonly;
                    AC.LogHandle = h->oldhandle;
                    AO.PrintType = h->oldprinttype;
                    AM.silent = h->oldsilent;
                    return(-1);
                }
                s++; ss = s;
                while ( chartype[*s] <= 1 ) s++;
                if ( s == ss ) goto nodollar;
                c = *s; *s = 0;
                num = GetDollar(ss);
                if ( num < 0 ) {
                    MesPrint("@#write instruction: $%s has not been defined",ss);
                    AM.FileOnlyFlag = h->oldlogonly;
                    AC.LogHandle = h->oldhandle;
                    AO.PrintType = h->oldprinttype;
                    AM.silent = h->oldsilent;
                    return(-1);
                }
                *s = c;
                if ( *s == '[' ) {
                    if ( Dollars[num].nfactors <= 0 ) {
                        *s = 0;
                        MesPrint("@#write instruction: $%s has not been factorized",ss);
                        AM.FileOnlyFlag = h->oldlogonly;
                        AC.LogHandle = h->oldhandle;
                        AO.PrintType = h->oldprinttype;
                        AM.silent = h->oldsilent;
                        return(-1);
                    }
/*
                    Now get the number between the []
*/
                    nfac = GetDollarNumber(&s,Dollars+num);

                    if ( Dollars[num].nfactors == 1 && nfac == 1 ) goto writewhole;

                    if ( ( dolalloc = WriteDollarFactorToBuffer(num,nfac,0) ) == 0 ) {
                        AM.FileOnlyFlag = h->oldlogonly;
                        AC.LogHandle = h->oldhandle;
                        AO.PrintType = h->oldprinttype;
                        AM.silent = h->oldsilent;
                        return(-1);
                    }
                    goto writealloc;
                }
                else if ( *s && *s != ' ' && *s != ',' && *s != '\t' ) {
                    MesPrint("@#write instruction: illegal characters after $-variable");
                    AM.FileOnlyFlag = h->oldlogonly;
                    AC.LogHandle = h->oldhandle;
                    AO.PrintType = h->oldprinttype;
                    AM.silent = h->oldsilent;
                    return(-1);
                }
                else {
writewhole:
                  if ( ( dolalloc = WriteDollarToBuffer(num,0) ) == 0 ) {
                    AM.FileOnlyFlag = h->oldlogonly;
                    AC.LogHandle = h->oldhandle;
                    AO.PrintType = h->oldprinttype;
                    AM.silent = h->oldsilent;
                    return(-1);
                  }
                  else {
writealloc:
                    ss = dolalloc;
                    while ( *ss ) {
                        if ( to >= stopper ) {
                            if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
                                *to++ = '&';
                            }
                            num = to - Out;
                            WriteString(wtype,Out,num);
                            to = Out;
                            for ( i = 0; i < number; i++ ) *to++ = ' ';
                            if ( AC.OutputMode == FORTRANMODE
                             || AC.OutputMode == PFORTRANMODE ) to[-2] = '&';
                        }
                        if ( chartype[*ss] > 3 ) { *to++ = *ss++; }
                        else {
                            sss = ss; while ( chartype[*ss] <= 3 ) ss++;
                            if ( ( to + (ss-sss) ) >= stopper ) {
                                if ( (ss-sss) >= (stopper-Out) ) {
                                    if ( ( to - stopper ) < 10 ) {
                                        if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 ==
ISFORTRAN90 ) {
```

```
                                        *to++ = '&';
                                    }
                                    num = to - Out;
                                    WriteString(wtype,Out,num);
                                    to = Out;
                                    for ( i = 0; i < number; i++ ) *to++ = ' ';
                                    if ( AC.OutputMode == FORTRANMODE
                                     || AC.OutputMode == PFORTRANMODE ) to[-2] = '&';
                                }
                                while ( (ss-sss) >= (stopper-Out) ) {
                                    while ( to < stopper-1 ) {
                                        *to++ = *sss++;
                                    }
                                    if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 ==
    ISFORTRAN90 ) {
                                        *to++ = '&';
                                    }
                                    else {
                                        *to++ = '\\';
                                    }
                                    num = to - Out;
                                    WriteString(wtype,Out,num);
                                    to = Out;
                                    if ( AC.OutputMode == FORTRANMODE
                                     || AC.OutputMode == PFORTRANMODE ) {
                                        for ( i = 0; i < number; i++ ) *to++ = ' ';
                                        to[-2] = '&';
                                    }
                                }
                            }
                            else {
                                if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90
    ) {
                                    *to++ = '&';
                                }
                                num = to - Out;
                                WriteString(wtype,Out,num);
                                to = Out;
                                for ( i = 0; i < number; i++ ) *to++ = ' ';
                                if ( AC.OutputMode == FORTRANMODE
                                 || AC.OutputMode == PFORTRANMODE ) to[-2] = '&';
                            }
                        }
                        while ( sss < ss ) *to++ = *sss++;
                    }
                }
            }
            M_free(dolalloc,"written dollar");
            fstring++;
        }
    }
    else if ( *fstring == 's' ) {
        fstring++;
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
        if ( *s == '"' ) {
            s++; ss = s;
            while ( *s ) {
                if ( *s == '\\' ) s++;
                else if ( *s == '"' ) break;
                s++;
            }
            if ( *s == 0 ) {
                MesPrint("@#write instruction: Missing \" in string");
                AM.FileOnlyFlag = h->oldlogonly;
                AC.LogHandle = h->oldhandle;
                AO.PrintType = h->oldprinttype;
                AM.silent = h->oldsilent;
                return(-1);
            }
            while ( ss < s ) {
                if ( to >= stopper ) {
                    num = to - Out;
                    WriteString(wtype,Out,num);
                    to = Out;
                }
                if ( *ss == '\\' ) ss++;
                *to++ = *ss++;
            }
            s++;
        }
        else {
            sss = ss = s;
            while ( *s && *s != ',' ) {
                if ( *s == '\\' ) { s++; sss = s+1; }
                s++;
            }
            while ( s > sss+1 && ( s[-1] == ' ' || s[-1] == '\t' ) ) s--;
```

```
                    while ( ss < s ) {
                        if ( to >= stopper ) {
                            num = to - Out;
                            WriteString(wtype,Out,num);
                            to = Out;
                        }
                        if ( *ss == '\\' ) ss++;
                        *to++ = *ss++;
                    }
                }
            }
            else if ( *fstring == 'X' ) {
                fstring++;
                if ( cbuf[AM.sbufnum].numrhs > 0 ) {
/*
                    This should be only to the value of AM.oldnumextrasymbols
*/
                    UBYTE *s = GetPreVar(AM.oldnumextrasymbols,0);
                    WORD x = 0;
                    while ( *s >= '0' && *s <= '9' ) x = 10*x + *s++ - '0';
                    if ( x > 0 )
                        PrintSubtermList(1,x);
                    else
                        PrintSubtermList(1,cbuf[AM.sbufnum].numrhs);
                }
            }
            else if ( *fstring == 'O' ) {
                number = AO.OutSkip;
dooptim:
                fstring++;
/*
                First test whether there is an optimization buffer
*/
                if ( AO.OptimizeResult.code == NULL && AO.OptimizationLevel != 0 ) {
                    MesPrint("@In #write instruction: no optimization results available!");
                    return(-1);
                }
                num = to - Out;
                WriteString(wtype,Out,num);
                to = Out;
                if ( AO.OptimizationLevel != 0 ) {
                    WORD oldoutskip = AO.OutSkip;
                    AO.OutSkip = number;
                    optimize_print_code(0);
                    AO.OutSkip = oldoutskip;
                }
            }
            else if ( *fstring == 'e' || *fstring == 'E' ) {
                if ( *fstring == 'E'
                    || AC.OutputMode == FORTRANMODE
                    || AC.OutputMode == PFORTRANMODE ) nosemi = 1;
                else nosemi = 0;
                fstring++;
                while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
                if ( chartype[*s] != 0 && *s != '[' ) {
noexpr:             MesPrint("@expression name expected in #write instruction");
                    AM.FileOnlyFlag = h->oldlogonly;
                    AC.LogHandle = h->oldhandle;
                    AO.PrintType = h->oldprinttype;
                    AM.silent = h->oldsilent;
                    return(-1);
                }
                ss = s;
                if ( ( s = SkipAName(ss) ) == 0 || s[-1] == '_' ) goto noexpr;
                s1 = s; c = c1 = *s1;
                if ( c1 == '(' ) {
                    SKIPBRA3(s)
                    if ( *s == ')' ) {
                        AO.CurBufWrt = s1+1;
                        c = *s; *s = 0;
                    }
                    else {
                        MesPrint("@Illegal () specifier in expression name in #write");
                        AM.FileOnlyFlag = h->oldlogonly;
                        AC.LogHandle = h->oldhandle;
                        AO.PrintType = h->oldprinttype;
                        AM.silent = h->oldsilent;
                        return(-1);
                    }
                }
                else AO.CurBufWrt = (UBYTE *)underscore;
                *s1 = 0;
                num = to - Out;
                if ( num > 0 ) WriteUnfinString(wtype,Out,num);
                to = Out;
                WORD oldOptimizationLevel = AO.OptimizationLevel;
                AO.OptimizationLevel = 0;
```

```
                if ( WriteOne(ss,(int)num,nosemi) < 0 ) {
                    AM.FileOnlyFlag = h->oldlogonly;
                    AC.LogHandle = h->oldhandle;
                    AO.PrintType = h->oldprinttype;
                    AM.silent = h->oldsilent;
                    return(-1);
                }
                AO.OptimizationLevel = oldOptimizationLevel;
                *s1 = c1;
                if ( s > s1 ) *s++ = c;
            }
/*
            File content
*/
            else if ( ( *fstring == 'f' ) || ( *fstring == 'F' ) ) {
                LONG n;
                while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
                ss = s;
                while ( *s && *s != ',' ) {
                    if ( *s == '\\' ) s++;
                    s++;
                }
                c = *s; *s = 0;
                s1 = LoadInputFile(ss,HEADERFILE);
                *s = c;
/*
                There should have been a way to pass the file size.
                Also there should be conversions for \r\n etc.
*/
                if ( s1 ) {
                    ss = s1; while ( *ss ) ss++;
                    n = ss-s1;
                    WriteString(wtype,s1,n);
                    M_free(s1,"copy file");
                }
                else if ( *fstring == 'F' ) {
                    *s = 0;
                    MesPrint("@Error in #write: could not open file %s",ss);
                    *s = c;
                    goto ReturnWithError;
                }
                fstring++;
            }
            else if ( *fstring == '%' ) {
                *to++ = *fstring++;
            }
            else if ( FG.cTable[*fstring] == 1 ) {  /* %#S */
                number = 0;
                while ( FG.cTable[*fstring] == 1 ) {
                    number = 10*number + *fstring++ - '0';
                }
                if ( *fstring == 'O' ) goto dooptim;
                else if ( *fstring == 'd' ) goto donumber;
                else if ( *fstring == '$' ) goto dodollar;
                else if ( *fstring == 'X' || *fstring == 'x' ) {
                    if ( number > 0 && number <= cbuf[AM.sbufnum].numrhs ) {
                        UBYTE buffer[80], *out, *old1, *old2, *old3;
                        WORD *term, first;
                        if ( *fstring == 'X' ) {
                            out = StrCopy((UBYTE *)AC.extrasym,buffer);
                            if ( AC.extrasymbols == 0 ) {
                                out = NumCopy(number,out);
                                out = StrCopy((UBYTE *)"_",out);
                            }
                            else if ( AC.extrasymbols == 1 ) {
                                if ( AC.OutputMode == CMODE ) {
                                    out = StrCopy((UBYTE *)"[",out);
                                    out = NumCopy(number,out);
                                    out = StrCopy((UBYTE *)"]",out);
                                }
                                else {
                                    out = StrCopy((UBYTE *)"(",out);
                                    out = NumCopy(number,out);
                                    out = StrCopy((UBYTE *)")",out);
                                }
                            }
                            out = StrCopy((UBYTE *)"=",out);
                            ss = buffer;
                            while ( ss < out ) {
                                if ( to >= stopper ) {
                                    num = to - Out;
                                    WriteString(wtype,Out,num);
                                    to = Out;
                                }
                                *to++ = *ss++;
                            }
                        }
```

```
                        term = cbuf[AM.sbufnum].rhs[number];
                        first = 1;
                        if ( *term == 0 ) {
                            *to++ = '0';
                        }
                        else {
                            old1 = AO.OutFill;
                            old2 = AO.OutputLine;
                            old3 = AO.OutStop;
                            AO.OutFill = to;
                            AO.OutputLine = Out;
                            AO.OutStop = Out + AC.LineLength;
                            while ( *term ) {
                                if ( WriteInnerTerm(term,first) ) Terminate(-1);
                                term += *term;
                                first = 0;
                            }
                            to = Out + (AO.OutFill-AO.OutputLine);
                            AO.OutFill = old1;
                            AO.OutputLine = old2;
                            AO.OutStop = old3;
                        }
                    }
                    fstring++;
                }
                else {
                    goto IllegControlSequence;
                }
            }
            else if ( *fstring == 0 ) {
                *to++ = 0;
            }
            else {
IllegControlSequence:
                MesPrint("@Illegal control sequence in format string in #write instruction");
ReturnWithError:
                AM.FileOnlyFlag = h->oldlogonly;
                AC.LogHandle = h->oldhandle;
                AO.PrintType = h->oldprinttype;
                AM.silent = h->oldsilent;
                return(-1);
            }
        }
        else {
            *to++ = *fstring++;
        }
    }
/*
    Now flush the output
*/
    num = to - Out;
    /*[15apr2004 mt]:*/
    if(wtype==EXTERNALCHANNELOUT){
        if(num!=0)
            WriteUnfinString(wtype,Out,num);
    }else
    /*:[15apr2004 mt]*/
    WriteString(wtype,Out,num);
/*
    and restore original parameters
*/
    AM.FileOnlyFlag = h->oldlogonly;
    AC.LogHandle = h->oldhandle;
    AO.PrintType = h->oldprinttype;
    AM.silent = h->oldsilent;
    return(0);
}

/*
        #] writeToChannel :
        #[ DoFactDollar :

        Executes the #factdollar $var
              instruction
*/

int DoFactDollar(UBYTE *s)
{
    GETIDENTITY
    WORD numdollar, *oldworkpointer;

    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == '$' ) {
        if ( GetName(AC.dollarnames,s+1,&numdollar,NOAUTO) != CDOLLAR ) {
            MesPrint("@%s is undefined",s);
```

```
            return(-1);
        }
        s = SkipAName(s+1);
        if ( *s != 0 ) {
            MesPrint("@#FactDollar should have a single $variable for its argument");
            return(-1);
        }
        NewSort(BHEAD0);
        oldworkpointer = AT.WorkPointer;
        if ( DollarFactorize(BHEAD numdollar) ) return(-1);
        AT.WorkPointer = oldworkpointer;
        LowerSortLevel();
        return(0);
    }
    else if ( ParenthesesTest(s) ) return(-1);
    else {
        MesPrint("@#FactDollar should have a single $variable for its argument");
        return -1;
    }
}

/*
        #] DoFactDollar :
        #[ GetDollarNumber :
*/

WORD GetDollarNumber(UBYTE **inp, DOLLARS d)
{
    UBYTE *s = *inp, c, *name;
    WORD number, nfac, *w;
    DOLLARS dd;
    s++;
    if ( *s == '$' ) {
        s++; name = s;
        while ( FG.cTable[*s] < 2 ) s++;
        c = *s; *s = 0;
        if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
            MesPrint("@dollar in #write should have been defined previously");
            Terminate(-1);
        }
        *s = c;
        dd = Dollars + number;
        if ( c == '[' ) {
            *inp = s;
            nfac = GetDollarNumber(inp,dd);
            s = *inp;
            if ( *s != ']' ) {
                MesPrint("@Illegal factor for dollar variable");
                Terminate(-1);
            }
            *inp = s+1;
            if ( nfac == 0 ) {
                if ( dd->nfactors > d->nfactors ) {
TooBig:
                    MesPrint("@Factor number for dollar variable too large");
                    Terminate(-1);
                }
                return(dd->nfactors);
            }
            w = dd->factors[nfac-1].where;
            if ( w == 0 ) {
                if ( dd->factors[nfac-1].value > d->nfactors ||
                     dd->factors[nfac-1].value < 0 ) goto TooBig;
                return(dd->factors[nfac-1].value);
            }
            if ( *w == 4 && w[4] == 0 && w[3] == 3 && w[2] == 1
              && w[1] <= d->nfactors ) return(w[1]);
            if ( w[*w] == 0 && w[*w-1] == *w-1 ) goto TooBig;
IllNum:
            MesPrint("@Illegal factor number for dollar variable");
            Terminate(-1);
        }
        else {  /* The dollar should be a number */
            if ( dd->type == DOLZERO ) {
                return(0);
            }
            else if ( dd->type == DOLTERMS || dd->type == DOLNUMBER ) {
                w = dd->where;
                if ( *w == 4 && w[4] == 0 && w[3] == 3 && w[2] == 1
                  && w[1] <= d->nfactors ) return(w[1]);
                if ( w[*w] == 0 && w[*w-1] == *w-1 ) goto TooBig;
                goto IllNum;
            }
            else goto IllNum;
        }
    }
    else if ( FG.cTable[*s] == 1 ) {
```

```
        WORD x = *s++ - '0';
        while ( FG.cTable[*s] == 1 ) {
            x = 10*x + *s++ - '0';
            if ( x > d->nfactors ) {
                MesPrint("@Factor number %d for dollar variable too large",x);
                Terminate(-1);
            }
        }
        if ( *s != ']' ) {
            MesPrint("@Illegal factor number for dollar variable");
            Terminate(-1);
        }
        s++; *inp = s;
        return(x);
    }
    else {
        MesPrint("@Illegal factor indicator for dollar variable");
        Terminate(-1);
    }
    return(-1);
}

/*
        #] GetDollarNumber :
        #[ DoSetRandom :

        Executes the #SetRandom number
*/

int DoSetRandom(UBYTE *s)
{
    ULONG x;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
    x = 0;
    while ( FG.cTable[*s] == 1 ) {
        x = 10*x + (*s++-'0');
    }
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s  == 0 ) {
#ifdef WITHPTHREADS
#ifdef WITHSORTBOTS
        int id, totnum = MaX(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
#else
        int id, totnum = AM.totalnumberofthreads;
#endif
        for ( id = 0; id < totnum; id++ ) {
            AB[id]->R.wranfseed = x;
            if ( AB[id]->R.wranfia ) M_free(AB[id]->R.wranfia,"wranf");
            AB[id]->R.wranfia = 0;
        }
#else
        AR.wranfseed = x;
        if ( AR.wranfia ) M_free(AR.wranfia,"wranf");
        AR.wranfia = 0;
#endif
        return(0);
    }
    else {
        MesPrint("@proper syntax is #SetRandom number");
        return(-1);
    }
}

/*
        #] DoSetRandom :
        #[ DoOptimize :

        Executes the #Optimize(expr) instruction.
*/

int DoOptimize(UBYTE *s)
{
    GETIDENTITY
    UBYTE *exprname;
    WORD numexpr;
    int error = 0, i;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    DUMMYUSE(*s)
    exprname = s; s = SkipAName(s);
    if ( *s != 0 && *s != ';' ) {
        MesPrint("@proper syntax is #Optimize,expression");
        return(-1);
    }
    *s = 0;
```

```
    if ( GetName(AC.exprnames,exprname,&numexpr,NOAUTO) != CEXPRESSION ) {
        MesPrint("@%s is not an expression",exprname);
        error = 1;
    }
    else if ( AP.preError == 0 ) {
        EXPRESSIONS e = Expressions + numexpr;
        POSITION position;
        int firstterm;
        WORD *term = AT.WorkPointer;
        ClearOptimize();
        if ( AO.OptimizationLevel == 0 ) return(0);
        switch ( e->status ) {
            case LOCALEXPRESSION:
            case GLOBALEXPRESSION:
                break;
            default:
                MesPrint("@Expression %s is not an active unhidden local or global expression.",exprname);
                Terminate(-1);
                break;
        }
#ifdef WITHMPI
        if ( PF.me == MASTER )
#endif
        RevertScratch();
        for ( i = NumExpressions-1; i >= 0; i-- ) {
            AS.OldOnFile[i] = Expressions[i].onfile;
            AS.OldNumFactors[i] = Expressions[i].numfactors;
            AS.Oldvflags[i] = Expressions[i].vflags;
            Expressions[i].vflags &= ~(ISUNMODIFIED|ISZERO);
        }
        for ( i = 0; i < NumExpressions; i++ ) {
            if ( i == numexpr ) {
                PutPreVar(AM.oldnumextrasymbols,
                        GetPreVar((UBYTE *)"EXTRASYMBOLS_",0),0,1);
                Optimize(numexpr, 0);
                AO.OptimizeResult.nameofexpr = strDup1(exprname,"optimize expression name");
                continue;
            }
#ifdef WITHMPI
            if ( PF.me == MASTER ) {
#endif
            e = Expressions + i;
            switch ( e->status ) {
                case LOCALEXPRESSION:
                case SKIPLEXPRESSION:
                case DROPLEXPRESSION:
                case DROPPEDEXPRESSION:
                case GLOBALEXPRESSION:
                case SKIPGEXPRESSION:
                case DROPGEXPRESSION:
                case HIDELEXPRESSION:
                case HIDEGEXPRESSION:
                case DROPHLEXPRESSION:
                case DROPHGEXPRESSION:
                case INTOHIDELEXPRESSION:
                case INTOHIDEGEXPRESSION:
                    break;
                default:
                    continue;
            }
            AR.GetFile = 0;
            SetScratch(AR.infile,&(e->onfile));
            if ( GetTerm(BHEAD term) <= 0 ) {
                MesPrint("@Expression %d has problems reading from scratchfile",i);
                Terminate(-1);
            }
            term[3] = i;
            AR.DeferFlag = 0;
            SeekScratch(AR.outfile,&position);
            e->onfile = position;
            *AM.S0->sBuffer = 0; firstterm = -1;
            do {
                WORD *oldipointer = AR.CompressPointer;
                WORD *comprtop = AR.ComprTop;
                AR.ComprTop = AM.S0->sTop;
                AR.CompressPointer = AM.S0->sBuffer;
                if ( firstterm > 0 ) {
                    if ( PutOut(BHEAD term,&position,AR.outfile,1) < 0 ) goto DoSerr;
                }
                else if ( firstterm < 0 ) {
                    if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto DoSerr;
                    firstterm++;
                }
                else {
                    if ( PutOut(BHEAD term,&position,AR.outfile,-1) < 0 ) goto DoSerr;
                    firstterm++;
```

```
                }
                AR.CompressPointer = oldipointer;
                AR.ComprTop = comprtop;
            } while ( GetTerm(BHEAD term) );
            if ( FlushOut(&position,AR.outfile,1) ) {
DoSerr:
                MesPrint("@Expression %d has problems writing to scratchfile",i);
                Terminate(-1);
            }
#ifdef WITHMPI
            }
#endif
        }
/*
        Now some administration and we are done
*/
        UpdateMaxSize();
    }
    else {
        ClearOptimize();
    }
    return(error);

}

/*
        #] DoOptimize :
        #[ DoClearOptimize :

        Clears all relevant buffers of the output optimization
*/

int DoClearOptimize(UBYTE *s)
{
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    DUMMYUSE(*s);
    return(ClearOptimize());
}

/*
        #] DoClearOptimize :
        #[ DoSkipExtraSymbols :

        Adds the intermediate variables of the previous optimization
        to the list of extra symbols, provided it has not yet been erased
        by a #clearoptimize
        To remove them again one needs to use the 'delete extrasymbols;'
        or the 'delete extrasymbols>num;' statement in which num is the
        old number of extra symbols.
*/

int DoSkipExtraSymbols(UBYTE *s)
{
    CBUF *C = cbuf + AM.sbufnum;
    WORD tt = 0, j = 0, oldval = AO.OptimizeResult.minvar;
    if ( AO.OptimizeResult.code == NULL ) return(0);
    if ( AO.OptimizationLevel == 0 ) return(0);
    while ( *s == ',' ) s++;
    if ( *s == 0 ) {
        AO.OptimizeResult.minvar = AO.OptimizeResult.maxvar+1;
    }
    else {
        while ( *s <= '9' && *s >= '0' ) j = 10*j + *s++ - '0';
        if ( *s ) {
            MesPrint("@Illegal use of #SkipExtraSymbols instruction");
            Terminate(-1);
        }
        AO.OptimizeResult.minvar += j;
        if ( AO.OptimizeResult.minvar > AO.OptimizeResult.maxvar )
            AO.OptimizeResult.minvar = AO.OptimizeResult.maxvar+1;
    }
    j = AO.OptimizeResult.minvar - oldval;
    while ( j > 0 ) {
        AddRHS(AM.sbufnum,1);
        AddNtoC(AM.sbufnum,1,&tt,16);
        AddToCB(C,0)
        InsTree(AM.sbufnum,C->numrhs);
        j--;
    }
    return(0);
}

/*
        #] DoSkipExtraSymbols :
        #[ DoPreReset :

```

```
        Does a reset of variables.
        Currently only the timer (stopwatch) of `timer_'
*/

int DoPreReset(UBYTE *s)
{
    UBYTE *ss, c;
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    while ( *s == ' ' || *s == '\t' ) s++;
    if ( *s == 0 ) {
        MesPrint("@proper syntax is #Reset variable");
        return(-1);
    }
    ss = s;
    while ( FG.cTable[*s] == 0 ) s++;
    c = *s; *s = 0;
    if ( ( StrICmp(ss,(UBYTE *)"timer") == 0 )
      || ( StrICmp(ss,(UBYTE *)"stopwatch") == 0 ) ) {
        *s = c;
        AP.StopWatchZero = GetRunningTime();
        return(0);
    }
    else {
        *s = c;
        MesPrint("@proper syntax is #Reset variable");
        return(-1);
    }
}

/*
        #] DoPreReset :
        #[ DoPreAppendPath :
*/

static int DoAddPath(UBYTE *s, int bPrepend)
{
    /* NOTE: this doesn't support some file systems, e.g., 0x5c with CP932. */

    UBYTE *path, *path_end, *current_dir, *current_dir_end, *NewPath, *t;
    int bRelative, n;

    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);

    /* Parse the path in the input. */
    while ( *s == ' ' || *s == '\t' ) s++;  /* skip spaces */
    if ( *s == '"' ) {  /* the path is given by "..." */
        path = ++s;
        while ( *s && *s != '"' ) {
            if ( SEPARATOR != '\\' && *s == '\\' ) {  /* escape character, e.g., "\\"" */
                if ( !s[1] ) goto ImproperPath;
                s++;
            }
            s++;
        }
        if ( *s != '"' ) goto ImproperPath;
        path_end = s++;
    }
    else {
        path = s;
        while ( *s && *s != ' ' && *s != '\t' ) {
            if ( SEPARATOR != '\\' && *s == '\\' ) {  /* escape character, e.g., "\\ " */
                if ( !s[1] ) goto ImproperPath;
                s++;
            }
            s++;
        }
        path_end = s;
    }
    if ( path == path_end ) goto ImproperPath;  /* empty path */
    while ( *s == ' ' || *s == '\t' ) s++;  /* skip spaces */
    if ( *s ) goto ImproperPath;  /* extra tokens found */

    /* Check if the path is an absolute path. */
    bRelative = 1;
    if ( path[0] == SEPARATOR ) {  /* starts with the directory separator */
        bRelative = 0;
    }
#ifdef WINDOWS
    else if ( chartype[path[0]] == 0 && path[1] == ':' ) {  /* starts with (drive letter): */
        bRelative = 0;
    }
#endif

    /* Get the current file directory when a relative path is given. */
    if ( bRelative ) {
```

```
        if ( !AC.CurrentStream ) goto FileNameUnavailable;
        if ( AC.CurrentStream->type != FILESTREAM && AC.CurrentStream->type != REVERSEFILESTREAM ) goto FileNameUnavailable;
        if ( !AC.CurrentStream->name ) goto FileNameUnavailable;
        s = current_dir = current_dir_end = AC.CurrentStream->name;
        while ( *s ) {
            if ( SEPARATOR != '\\' && *s == '\\' && s[1] ) {  /* escape character, e.g., "\\\"" */
                s += 2;
                continue;
            }
            if ( *s == SEPARATOR ) {
                current_dir_end = s;
            }
            s++;
        }
    }
    else {
        current_dir = current_dir_end = NULL;
    }

    /* Allocate a buffer for new AM.Path. */
    n = path_end - path;
    if ( AM.Path ) n += StrLen(AM.Path) + 1;
    if ( current_dir != current_dir_end ) n+= current_dir_end - current_dir + 1;
    s = NewPath = (UBYTE *)Malloc1(n + 1,"add path");

    /* Construct new FORM path. */
    if ( bPrepend ) {
        if ( current_dir != current_dir_end ) {
            t = current_dir;
            while ( t != current_dir_end ) *s++ = *t++;
            *s++ = SEPARATOR;
        }
        t = path;
        while ( t != path_end ) *s++ = *t++;
        if ( AM.Path ) *s++ = PATHSEPARATOR;
    }
    if ( AM.Path ) {
        t = AM.Path;
        while ( *t ) *s++ = *t++;
    }
    if ( !bPrepend ) {
        if ( AM.Path ) *s++ = PATHSEPARATOR;
        if ( current_dir != current_dir_end ) {
            t = current_dir;
            while ( t != current_dir_end ) *s++ = *t++;
            *s++ = SEPARATOR;
        }
        t = path;
        while ( t != path_end ) *s++ = *t++;
    }
    *s = '\0';

    /* Update AM.Path. */
    if ( AM.Path ) M_free(AM.Path,"add path");
    AM.Path = NewPath;

    return(0);

ImproperPath:
    MesPrint("@Improper syntax for %#%sPath", bPrepend ? "Prepend" : "Append");
    return(-1);

FileNameUnavailable:
    /* This may be improved in future. */
    MesPrint("@Sorry, %#%sPath can't resolve the current file name from here", bPrepend ? "Prepend" : "Append");
    return(-1);
}

int DoPreAppendPath(UBYTE *s)
{
    return DoAddPath(s, 0);
}

/*
        #] DoPreAppendPath :
        #[ DoPrePrependPath :
*/

int DoPrePrependPath(UBYTE *s)
{
    return DoAddPath(s, 1);
}

/*
        #] DoPrePrependPath :
```

```
06899     # ] PreProcessor :
06900 */
```

4.86 proces.c File Reference

`#include "form3.h"`

Include dependency graph for proces.c:

Functions

- WORD Processor ()
- WORD TestSub (PHEAD WORD ∗term, WORD level)
- WORD InFunction (PHEAD WORD ∗term, WORD ∗termout)
- WORD InsertTerm (PHEAD WORD ∗term, WORD replac, WORD extractbuff, WORD ∗position, WORD ∗termout, WORD tepos)
- LONG PasteFile (PHEAD WORD number, WORD ∗accum, POSITION ∗position, WORD ∗∗accfill, RENUMBER renumber, WORD ∗freeze, WORD nexpr)
- WORD ∗ PasteTerm (PHEAD WORD number, WORD ∗accum, WORD ∗position, WORD times, WORD divby)
- WORD FiniTerm (PHEAD WORD ∗term, WORD ∗accum, WORD ∗termout, WORD number, WORD tepos)
- WORD Generator (PHEAD WORD ∗term, WORD level)
- WORD DoOnePow (PHEAD WORD ∗term, WORD power, WORD nexp, WORD ∗accum, WORD ∗aa, WORD level, WORD ∗freeze)
- WORD Deferred (PHEAD WORD ∗term, WORD level)
- WORD PrepPoly (PHEAD WORD ∗term, WORD par)
- WORD PolyFunMul (PHEAD WORD ∗term)

Variables

- WORD printscratch [2]

4.86.1 Detailed Description

Contains the central terms processor routines. This is the core of the virtual machine. All other files are to help these routines.

Definition in file proces.c.

4.86.2 Function Documentation

4.86.2.1 Processor()

```
WORD Processor ( )
```

This is the central processor. It accepts a stream of Expressions which is accessed by calls to GetTerm. The expressions reside either in AR.infile or AR.hidefile The definitions of an expression are seen as an id-statement, so the primary Expressions should be written to the system of scratch files as single terms with an expression pointer. Each expression is terminated with a zero and the whole is terminated by two zeroes.

The routine DoExecute should determine whether results are to be printed, should revert the scratch I/O directions etc. In principle it is DoExecute that calls Processor.

Returns

if everything OK: 0. Otherwise error. The preprocessor may continue with compilation though. Really fatal errors should return on the spot by calling Terminate.

Definition at line 64 of file proces.c.

References CbUf::Buffer, EndSort(), FlushOut(), Generator(), CbUf::lhs, LowerSortLevel(), NewSort(), PF_BroadcastRHS(), PF_InParallelProcessor(), PF_Processor(), CbUf::Pointer, PutOut(), CbUf::rhs, and StoreTerm().

Here is the call graph for this function:

4.86.2.2 TestSub()

```
WORD TestSub (
            PHEAD WORD * term,
            WORD level )
```

TestSub hunts for subexpression pointers. If one is found its power is given in AN.TeSuOut. and the returnvalue is 'expressionnumber'. If the expression number is negative it is an expression on disk.

In addition this routine tries to locate subexpression pointers in functions. It also notices that action must be taken with any of the special functions.

Parameters

term	The term in which TestSub hunts for potential action
level	The number of the 'level' in the compiler buffer.

Returns

The number of the (sub)expression that was encountered.

Other values that are returned are in AN.TeSuOut, AR.TePos, AT.TMbuff, AN.TeInFun, AN.Frozen, AT.TMaddr

The level in the compiler buffer is more or less the number of the statement in the module. Hence it refers to the element in the lhs array.

This routine is one of the most important routines in FORM.

Definition at line 680 of file proces.c.

References AddRHS(), TaBlEs::bounds, TaBlEs::bufnum, CbUf::CanCommu, CompareSymbols(), DoubleCbuffer(), EndSort(), Generator(), LowerSortLevel(), MiNmAx::maxi, MiNmAx::mini, TaBlEs::mm, NewSort(), TaBlEs::numind, TaBlEs::pattern, CbUf::Pointer, PrepPoly(), TaBlEs::prototype, CbUf::rhs, MiNmAx::size, TaBlEs::sparse, StoreTerm(), TaBlEs::strict, TaBlEs::tablepointers, TestSub(), and CbUf::Top.

Referenced by Generator(), and TestSub().

Here is the call graph for this function:

4.86.2.3 InFunction()

```
WORD InFunction (
            PHEAD WORD * term,
            WORD * termout )
```

Makes the replacement of the subexpression with the number 'replac' in a function argument. Additional information is passed in some of the AR, AN, AT variables.

Parameters

term	The input term
termout	The output term

Returns

0: everything is fine, Negative: fatal, Positive: error.

Special attention should be given to nested functions!

Definition at line 1969 of file proces.c.

References Compare1(), CompareSymbols(), EndSort(), Generator(), LowerSortLevel(), NewSort(), StoreTerm(), and SymbolNormalize().

Referenced by Generator().

Here is the call graph for this function:

4.86.2.4 InsertTerm()

```
WORD InsertTerm (
            PHEAD WORD * term,
            WORD replac,
            WORD extractbuff,
            WORD * position,
            WORD * termout,
            WORD tepos )
```

Puts the terms 'term' and 'position' together into a single legal term in termout. replac is the number of the subexpression that should be replaced. It must be a positive term. When action is needed in the argument of a function all terms in that argument are dealt with recursively. The subexpression is sorted. Only one subexpression is done at a time this way.

Parameters

term	the input term
replac	number of the subexpression pointer to replace

Parameters

extractbuff	number of the compiler buffer replac refers to
position	position from where to take the term in the compiler buffer
termout	the output term
tepos	offset in term where the subexpression is.

Returns

Normal conventions (OK = 0).

Definition at line 2514 of file proces.c.

Referenced by Deferred(), Generator(), and PF_Deferred().

4.86.2.5 PasteFile()

```
LONG PasteFile (
            PHEAD WORD number,
            WORD * accum,
            POSITION * position,
            WORD ** accfill,
            RENUMBER renumber,
            WORD * freeze,
            WORD nexpr )
```

Gets a term from stored expression expr and puts it in the accumulator at position number. It returns the length of the term that came from file.

Parameters

number	number of partial terms to skip in accum
accum	the accumulator
position	file position from where to get the stored term
accfill	returns tail position in accum
renumber	the renumber struct for the variables in the stored expression
freeze	information about if we need only the contents of a bracket
nexpr	the number of the stored expression

Returns

Normal conventions (OK = 0).

Definition at line 2650 of file proces.c.

Referenced by Generator().

4.86.2.6 PasteTerm()

```
WORD * PasteTerm (
            PHEAD WORD number,
            WORD * accum,
            WORD * position,
            WORD times,
            WORD divby )
```

Puts the term at position in the accumulator accum at position 'number+1'. if times $>$ 0 the coefficient of this term is multiplied by times/divby.

Parameters

number	The number of term fragments in accum that should be skipped
accum	The accumulator of term fragments
position	A position in (typically) a compiler buffer from where a (piece of a) term comes.
times	Multiply the result by this
divby	Divide the result by this.

This routine is typically used when we have to replace a (sub)expression pointer by a power of a (sub)expression. This uses mostly a binomial expansion and the new term is the old term multiplied one by one by terms of the new expression. The factors times and divby keep track of the binomial coefficient. Once this is complete, the routine FiniTerm will make the contents of the accumulator into a proper term that still needs to be normalized.

Definition at line 2772 of file proces.c.

Referenced by Generator().

4.86.2.7 FiniTerm()

```
WORD FiniTerm (
            PHEAD WORD * term,
            WORD * accum,
            WORD * termout,
            WORD number,
            WORD tepos )
```

Concatenates the contents of the accumulator into a single legal term, which replaces the subexpression pointer

Parameters

term	the input term with the (sub)expression subterm
accum	the accumulator with the term fragments
termout	the location where the output should be written
number	the number of term fragments in the accumulator
tepos	the position of the subterm in term to be replaced

Definition at line 2837 of file proces.c.

Referenced by DoOnePow(), and Generator().

4.86.2.8 Generator()

```
WORD Generator (
            PHEAD WORD * term,
            WORD level )
```

The heart of the program. Here the expansion tree is set up in one giant recursion

Parameters

term	the input term. may be overwritten
level	the level in the compiler buffer (number of statement)

Returns

Normal conventions (OK = 0).

The routine looks first whether there are unsubstituted (sub)expressions. If so, one of them gets inserted term by term and the new term is used in a renewed call to Generator. If there are no (sub)expressions, the term is normalized, the compiler level is raised (next statement) and the program looks what type of statement this is. If this is a special statement it is either treated on the spot or the appropriate routine is called. If it is a substitution, the pattern matcher is called (TestMatch) which tells whether there was a match. If so we need to call TestSub again to test for (sub)expressions. If we run out of levels, the term receives a final treatment for modulus calculus and/or brackets and is then sent off to the sorting routines.

Definition at line 3034 of file proces.c.

References Deferred(), DoOnePow(), FiniTerm(), Generator(), InFunction(), InsertTerm(), CbUf::lhs, VaRrEnUm::lo, PasteFile(), PasteTerm(), PF_Deferred(), CbUf::Pointer, PolyFunMul(), PrepPoly(), PutPreVar(), CbUf::rhs, StoreTerm(), ReNuMbEr::symb, TestMatch(), and TestSub().

Referenced by Deferred(), DoOnePow(), Generator(), InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Deferred(), PF_Processor(), Processor(), TestMatch(), and TestSub().

Here is the call graph for this function:

4.86.2.9 DoOnePow()

```
WORD DoOnePow (
            PHEAD WORD * term,
            WORD power,
            WORD nexp,
            WORD * accum,
            WORD * aa,
            WORD level,
            WORD * freeze )
```

Routine gets one power of an expression in the scratch system. If there are more powers needed there will be a recursion.

No attempt is made to use binomials because we have no information about commutating properties.

There is a searching for the contents of brackets if needed. This searching may be rather slow because of the single links.

Parameters

term	is the term we are adding to.
power	is the power of the expression that we need.
nexp	is the number of the expression.
accum	is the accumulator of terms. It accepts the termfragments that are made into a proper term in FiniTerm
aa	points to the start of the entire accumulator. In ∗aa we store the number of term fragments that are in the accumulator.
level	is the current depth in the tree of statements. It is needed to continue to the next operation/substitution with each generated term
freeze	is the pointer to the bracket information that should be matched.

Definition at line 4304 of file proces.c.

References DoOnePow(), FiniTerm(), Generator(), and FiLe::handle.

Referenced by DoOnePow(), and Generator().

Here is the call graph for this function:

4.86.2.10 Deferred()

```
WORD Deferred (
            PHEAD WORD * term,
            WORD level )
```

Picks up the deferred brackets. These are the bracket contents of which we postpone the reading when we use the 'Keep Brackets' statement. These contents are multiplying the terms just before they are sent to the sorting system. Special attention goes to having it thread-safe We have to lock positioning the file and reading it in a thread specific buffer.

Parameters

term	The term that must be multiplied by the contents of the current bracket
level	The compiler level. This is needed because after multiplying term by term we call Generator again.

Definition at line 4525 of file proces.c.

References Generator(), and InsertTerm().

Referenced by Generator().

Here is the call graph for this function:

4.86.2.11 PrepPoly()

```
WORD PrepPoly (
            PHEAD WORD * term,
            WORD par )
```

Routine checks whether the count of function AR.PolyFun is zero or one. If it is one and it has one scalarlike argument the coefficient of the term is pulled inside the argument. If the count is zero a new function is made with the coefficient as its only argument. The function should be placed at its proper position.

When this function is active it places the PolyFun as last object before the coefficient. This is needed because otherwise the compress algorithm has problems in MergePatches.

The bracket routine should also place the PolyFun at a comparable spot. The compression should then stop at the PolyFun. It doesn't really have to stop when writing the final result but this may be too complicated.

The parameter par tells whether we are at groundlevel or inside a function or dollar variable.

Definition at line 4653 of file proces.c.

Referenced by Generator(), and TestSub().

4.86.2.12 PolyFunMul()

```
WORD PolyFunMul (
            PHEAD WORD * term )
```

Multiplies the arguments of multiple occurrences of the polyfun. In this routine we do the original PolyFun with one argument only. The PolyRatFun (PolyFunType = 2) is done in a dedicated routine in the file polywrap.cc The new result is written over the old result.

Parameters

term	It contains the input term and later the output.

Returns

Normal conventions (OK = 0).

Definition at line 5035 of file proces.c.

References EndSort(), LowerSortLevel(), NewSort(), and StoreTerm().

Referenced by Generator().

Here is the call graph for this function:

4.86.3 Variable Documentation

4.86.3.1 printscratch

```
WORD printscratch[2]
```

Definition at line 39 of file proces.c.

4.87 proces.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
#define HIDEDEBUG
    #[ Includes : proces.c
*/

#include "form3.h"

WORD printscratch[2];

/*
    #] Includes :
    #[ Processor :
        #[ Processor :          WORD Processor()
*/
WORD Processor()
{
    GETIDENTITY
    WORD *term, *t, i, retval = 0, size;
    EXPRESSIONS e;
    POSITION position;
    WORD last, LastExpression, fromspectator;
    LONG dd = 0;
    CBUF *C = cbuf+AC.cbufnum;
    int firstterm;
    CBUF *CC = cbuf+AT.ebufnum;
    WORD **w, *cpo, *cbo;
    FILEHANDLE *curfile, *oldoutfile = AR.outfile;
    WORD oldBracketOn = AR.BracketOn;
    WORD *oldBrackBuf = AT.BrackBuf;
    WORD oldbracketindexflag = AT.bracketindexflag;
#ifdef WITHPTHREADS
    int OldMultiThreaded = AS.MultiThreaded, Oldmparallelflag = AC.mparallelflag;
#endif
    if ( CC->numrhs > 0 || CC->numlhs > 0 ) {
        if ( CC->rhs ) {
            w = CC->rhs; i = CC->numrhs;
            do { *w++ = 0; } while ( --i > 0 );
        }
        if ( CC->lhs ) {
            w = CC->lhs; i = CC->numlhs;
            do { *w++ = 0; } while ( --i > 0 );
        }
        CC->numlhs = CC->numrhs = 0;
        ClearTree(AT.ebufnum);
        CC->Pointer = CC->Buffer;
    }

    if ( NumExpressions == 0 ) return(0);
    AR.expflags = 0;
    AR.CompressPointer = AR.CompressBuffer;
    AR.NoCompress = AC.NoCompress;
    term = AT.WorkPointer;
    if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer) ) > AT.WorkTop ) return(MesWork());
    UpdatePositions();
    C->rhs[C->numrhs+1] = C->Pointer;
```

```
    AR.KeptInHold = 0;
    if ( AC.CollectFun ) AR.DeferFlag = 0;
    AR.outtohide = 0;
    AN.PolyFunTodo = 0;
#ifdef HIDEDEBUG
    MesPrint("Status at the start of Processor (HideLevel = %d)",AC.HideLevel);
    for ( i = 0; i < NumExpressions; i++ ) {
        e = Expressions+i;
        ExprStatus(e);
    }
#endif
/*
        Next determine the last expression. This is used for removing the
        input file when the final stage of the sort of this expression is
        reached. That can save up to 1/3 in disk space.
*/
    for ( i = NumExpressions-1; i >= 0; i-- ) {
        e = Expressions+i;
        if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
        || e->status == HIDELEXPRESSION || e->status == HIDEGEXPRESSION
        || e->status == SKIPLEXPRESSION || e->status == SKIPGEXPRESSION
        || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
        || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION
        ) break;
    }
    last = i;
    for ( i = NumExpressions-1; i >= 0; i-- ) {
        AS.OldOnFile[i] = Expressions[i].onfile;
        AS.OldNumFactors[i] = Expressions[i].numfactors;
/*      AS.Oldvflags[i] = e[i].vflags; */
        AS.Oldvflags[i] = Expressions[i].vflags;
        Expressions[i].vflags &= ~(ISUNMODIFIED|ISZERO);
    }
#ifdef WITHPTHREADS
/*
        When we run with threads we have to make sure that all local input
        buffers are pointed correctly. Of course this isn't needed if we
        run on a single thread only.
*/
    if ( AC.partodoflag && AM.totalnumberofthreads > 1 ) {
        AS.MultiThreaded = 1; AC.mparallelflag = PARALLELFLAG;
    }
    if ( AS.MultiThreaded && AC.mparallelflag == PARALLELFLAG ) {
        SetWorkerFiles();
    }
/*
        We start with running the expressions with expr->partodo in parallel.
        The current model is: give each worker an expression. Wait for
        workers to finish and tell them where to write.
        Then give them a new expression. Workers may have to wait for each
        other. This is also the case with the last one.
*/
    if ( AS.MultiThreaded && AC.mparallelflag == PARALLELFLAG ) {
        if ( InParallelProcessor() ) {
            retval = 1;
        }
        AS.MultiThreaded = OldMultiThreaded;
        AC.mparallelflag = Oldmparallelflag;
    }
#endif
#ifdef WITHMPI
    if ( AC.RhsExprInModuleFlag && PF.rhsInParallel && (AC.mparallelflag == PARALLELFLAG ||
AC.partodoflag) ) {
        if ( PF_BroadcastRHS() ) {
            retval = -1;
        }
    }
    PF.exprtodo = -1; /* This means, the slave does not perform inparallel */
    if ( AC.partodoflag > 0 ) {
        if ( PF_InParallelProcessor() ) {
            retval = -1;
        }
    }
#endif
    for ( i = 0; i < NumExpressions; i++ ) {
#ifdef INNERTEST
        if ( AC.InnerTest ) {
            if ( StrCmp(AC.TestValue,(UBYTE *)INNERTEST) == 0 ) {
                MesPrint("Testing(Processor): value = %s",AC.TestValue);
            }
        }
#endif
        e = Expressions+i;
#ifdef WITHPTHREADS
        if ( AC.partodoflag > 0 && e->partodo > 0 && AM.totalnumberofthreads > 2 ) {
            e->partodo = 0;
            continue;
```

```
        }
#endif
#ifdef WITHMPI
        if ( AC.partodoflag > 0 && e->partodo > 0 && PF.numtasks > 2 ) {
            e->partodo = 0;
            continue;
        }
#endif
        AS.CollectOverFlag = 0;
        AR.expchanged = 0;
        if ( i == last ) LastExpression = 1;
        else             LastExpression = 0;
        if ( e->inmem ) {
/*
            #[ in memory : Memory allocated by poly.c only thusfar.
                           Here GetTerm cannot work.
                           For the moment we ignore this for parallelization.
*/
            WORD j;

            AR.GetFile = 0;
            SetScratch(AR.infile,&(e->onfile));
            if ( GetTerm(BHEAD term) <= 0 ) {
                MesPrint("(1) Expression %d has problems in scratchfile",i);
                retval = -1;
                break;
            }
            term[3] = i;
            AR.CurExpr = i;
            SeekScratch(AR.outfile,&position);
            e->onfile = position;
            if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto ProcErr;
            AR.DeferFlag = AC.ComDefer;
            NewSort(BHEAD0);
            AN.ninterms = 0;
            t = e->inmem;
            while ( *t ) {
                for ( j = 0; j < *t; j++ ) term[j] = t[j];
                t += *t;
                AN.ninterms++; dd = AN.deferskipped;
                if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)(sizeof(WORD)))) ) {
                    if ( GetMoreFromMem(term,&t) ) {
                        LowerSortLevel(); goto ProcErr;
                    }
                }
                AT.WorkPointer = term + *term;
                AN.RepPoint = AT.RepCount + 1;
                AN.IndDum = AM.IndDum;
                AR.CurDum = ReNumber(BHEAD term);
                if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
                if ( AN.ncmod ) {
                    if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
                    else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
                }
                else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
                if ( Generator(BHEAD term,0) ) {
                    LowerSortLevel(); goto ProcErr;
                }
                AN.ninterms += dd;
            }
            AN.ninterms += dd;
            if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) goto ProcErr;
            if ( AM.S0->TermsLeft ) e->vflags &= ~ISZERO;
            else e->vflags |= ISZERO;
            if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
                if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
            if ( AR.expchanged ) AR.expflags |= ISUNMODIFIED;
            AR.GetFile = 0;
/*
            #] in memory :
*/
        }
        else {
            AR.CurExpr = i;
            switch ( e->status ) {
            case UNHIDELEXPRESSION:
            case UNHIDEGEXPRESSION:
                AR.GetFile = 2;
#ifdef WITHMPI
                if ( PF.me == MASTER ) SetScratch(AR.hidefile,&(e->onfile));
#else
                SetScratch(AR.hidefile,&(e->onfile));
                AR.InHiBuf = AR.hidefile->POfull-AR.hidefile->POfill;
#ifdef HIDEDEBUG
                MesPrint("Hidefile: onfile: %15p, POposition: %15p, filesize: %15p",&(e->onfile)
                ,&(AR.hidefile->POposition),&(AR.hidefile->filesize));
                MesPrint("Set hidefile to buffer position %l/%l; AR.InHiBuf = %l"
```

```
                    ,(AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD)
                    ,(AR.hidefile->POfull-AR.hidefile->PObuffer)*sizeof(WORD)
                    ,AR.InHiBuf
                );
#endif
#endif
                curfile = AR.hidefile;
                goto commonread;
            case INTOHIDELEXPRESSION:
            case INTOHIDEGEXPRESSION:
                AR.outtohide = 1;
/*
                BugFix 12-feb-2016
                This may not work when the file is open and we move around.
                AR.hidefile->POfill = AR.hidefile->POfull;
*/
                SetEndHScratch(AR.hidefile,&position);
            case LOCALEXPRESSION:
            case GLOBALEXPRESSION:
                AR.GetFile = 0;
/*[20oct2009 mt]:*/
#ifdef WITHMPI
                if( ( PF.me == MASTER ) || (PF.mkSlaveInfile) )
#endif
                SetScratch(AR.infile,&(e->onfile));
/*:[20oct2009 mt]*/
                curfile = AR.infile;
commonread:;
#ifdef WITHMPI
                if ( PF_Processor(e,i,LastExpression) ) {
                    MesPrint("Error in PF_Processor");
                    goto ProcErr;
                }
/*[20oct2009 mt]:*/
                if ( AC.mparallelflag != PARALLELFLAG ){
                    if(PF.me != MASTER)
                        break;
#endif
/*:[20oct2009 mt]*/
                if ( GetTerm(BHEAD term) <= 0 ) {
#ifdef HIDEDEBUG
                    MesPrint("Error condition 1a");
                    ExprStatus(e);
#endif
                    MesPrint("(2) Expression %d has problems in scratchfile(process)",i);
                    retval = -1;
                    break;
                }
                term[3] = i;
                if ( term[5] < 0 ) {    /* Fill with spectator */
                    fromspectator = -term[5];
                    PUTZERO(AM.SpectatorFiles[fromspectator-1].readpos);
                    term[5] = AC.cbufnum;
                }
                else fromspectator = 0;
                if ( AR.outtohide ) {
                    SeekScratch(AR.hidefile,&position);
                    e->onfile = position;
                    if ( PutOut(BHEAD term,&position,AR.hidefile,0) < 0 ) goto ProcErr;
                }
                else {
                    SeekScratch(AR.outfile,&position);
                    e->onfile = position;
                    if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto ProcErr;
                }
                AR.DeferFlag = AC.ComDefer;
                AR.Eside = RHSIDE;
                if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
                    AR.BracketOn = 1;
                    AT.BrackBuf = AM.BracketFactors;
                    AT.bracketindexflag = 1;
                }
                if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
#ifdef WITHPTHREADS
                if ( AS.MultiThreaded && AC.mparallelflag == PARALLELFLAG ) {
                    if ( ThreadsProcessor(e,LastExpression,fromspectator) ) {
                        MesPrint("Error in ThreadsProcessor");
                        goto ProcErr;
                    }
                    if ( AR.outtohide ) {
                        AR.outfile = oldoutfile;
                        AR.hidefile->POfull = AR.hidefile->POfill;
                    }
                }
                else
#endif
                {
```

```
                    NewSort(BHEAD0);
                    AR.MaxDum = AM.IndDum;
                    AN.ninterms = 0;
                    for(;;) {
                     if ( fromspectator ) size = GetFromSpectator(term,fromspectator-1);
                     else                 size = GetTerm(BHEAD term);
                     if ( size <= 0 ) break;
                     SeekScratch(curfile,&position);
                     if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
                      StoreTerm(BHEAD term);
                     }
                     else {
                      AN.ninterms++; dd = AN.deferskipped;
                      if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)(sizeof(WORD)))) ) {
                        if ( GetMoreTerms(term) < 0 ) {
                          LowerSortLevel(); goto ProcErr;
                        }
                        SeekScratch(curfile,&position);
                      }
                      AT.WorkPointer = term + *term;
                      AN.RepPoint = AT.RepCount + 1;
                      if ( AR.DeferFlag ) {
                        AN.IndDum = Expressions[AR.CurExpr].numdummies + AM.IndDum;
                        AR.CurDum = AN.IndDum;
                      }
                      else {
                        AN.IndDum = AM.IndDum;
                        AR.CurDum = ReNumber(BHEAD term);
                      }
                      if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
                      if ( AN.ncmod ) {
                        if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
                        else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
                      }
                      else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
                      if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
                        && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
                        PolyFunClean(BHEAD term);
                      }
                      if ( Generator(BHEAD term,0) ) {
                        LowerSortLevel(); goto ProcErr;
                      }
                      AN.ninterms += dd;
                     }
                     SetScratch(curfile,&position);
                     if ( AR.GetFile == 2 ) {
                        AR.InHiBuf = (curfile->POfull-curfile->PObuffer)
                            -DIFBASE(position,curfile->POposition)/sizeof(WORD);
                     }
                     else {
                        AR.InInBuf = (curfile->POfull-curfile->PObuffer)
                            -DIFBASE(position,curfile->POposition)/sizeof(WORD);
                     }
                    }
                    AN.ninterms += dd;
                    if ( LastExpression ) {
                        UpdateMaxSize();
                        if ( AR.infile->handle >= 0 ) {
                            CloseFile(AR.infile->handle);
                            AR.infile->handle = -1;
                            remove(AR.infile->name);
                            PUTZERO(AR.infile->POposition);
                        }
                        AR.infile->POfill = AR.infile->POfull = AR.infile->PObuffer;
                    }
                    if ( AR.outtohide ) AR.outfile = AR.hidefile;
                    if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) goto ProcErr;
                    if ( AR.outtohide ) {
                        AR.outfile = oldoutfile;
                        AR.hidefile->POfull = AR.hidefile->POfill;
                    }
                    e->numdummies = AR.MaxDum - AM.IndDum;
                    UpdateMaxSize();
                }
                AR.BracketOn = oldBracketOn;
                AT.BrackBuf = oldBrackBuf;
                if ( ( e->vflags & TOBEFACTORED ) != 0 ) {
                    poly_factorize_expression(e);
                }
                else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
                 && ( ( e->vflags & ISFACTORIZED ) != 0 ) ) {
                    poly_unfactorize_expression(e);
                }
                AT.bracketindexflag = oldbracketindexflag;
                if ( AM.S0->TermsLeft )   e->vflags &= ~ISZERO;
                else                      e->vflags |= ISZERO;
                if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
```

```
                if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
                if ( AR.expchanged )    AR.expflags |= ISUNMODIFIED;
                AR.GetFile = 0;
                AR.outtohide = 0;
/*[20oct2009 mt]:*/
#ifdef WITHMPI
                }
#endif
#ifdef WITHPTHREADS
                if ( e->status == INTOHIDELEXPRESSION ||
                     e->status == INTOHIDEGEXPRESSION ) {
                    SetHideFiles();
                }
#endif
                break;
            case SKIPLEXPRESSION:
            case SKIPGEXPRESSION:
/*
                This can be greatly improved of course by file-to-file copy.
*/
#ifdef WITHMPI
                if ( PF.me != MASTER ) break;
#endif
                AR.GetFile = 0;
                SetScratch(AR.infile,&(e->onfile));
                if ( GetTerm(BHEAD term) <= 0 ) {
#ifdef HIDEDEBUG
                    MesPrint("Error condition 1b");
                    ExprStatus(e);
#endif
                    MesPrint("(3) Expression %d has problems in scratchfile",i);
                    retval = -1;
                    break;
                }
                term[3] = i;
                AR.DeferFlag = 0;
                SeekScratch(AR.outfile,&position);
                e->onfile = position;
                *AM.S0->sBuffer = 0; firstterm = -1;
                do {
                    WORD *oldipointer = AR.CompressPointer;
                    WORD *comprtop = AR.ComprTop;
                    AR.ComprTop = AM.S0->sTop;
                    AR.CompressPointer = AM.S0->sBuffer;
                    if ( firstterm > 0 ) {
                        if ( PutOut(BHEAD term,&position,AR.outfile,1) < 0 ) goto ProcErr;
                    }
                    else if ( firstterm < 0 ) {
                        if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto ProcErr;
                        firstterm++;
                    }
                    else {
                        if ( PutOut(BHEAD term,&position,AR.outfile,-1) < 0 ) goto ProcErr;
                        firstterm++;
                    }
                    AR.CompressPointer = oldipointer;
                    AR.ComprTop = comprtop;
                } while ( GetTerm(BHEAD term) );
                if ( FlushOut(&position,AR.outfile,1) ) goto ProcErr;
                UpdateMaxSize();
                break;
            case HIDELEXPRESSION:
            case HIDEGEXPRESSION:
#ifdef WITHMPI
                if ( PF.me != MASTER ) break;
#endif
                AR.GetFile = 0;
                SetScratch(AR.infile,&(e->onfile));
                if ( GetTerm(BHEAD term) <= 0 ) {
#ifdef HIDEDEBUG
                    MesPrint("Error condition 1c");
                    ExprStatus(e);
#endif
                    MesPrint("(4) Expression %d has problems in scratchfile",i);
                    retval = -1;
                    break;
                }
                term[3] = i;
                AR.DeferFlag = 0;
                SetEndHScratch(AR.hidefile,&position);
                e->onfile = position;
#ifdef HIDEDEBUG
                if ( AR.hidefile->handle >= 0 ) {
                    POSITION possize,pos;
                    PUTZERO(possize);
                    PUTZERO(pos);
                    SeekFile(AR.hidefile->handle,&pos,SEEK_CUR);
```

```
                    SeekFile(AR.hidefile->handle,&possize,SEEK_END);
                    SeekFile(AR.hidefile->handle,&pos,SEEK_SET);
                    MesPrint("Processor Hide1: filesize(th) = %12p, filesize(ex) = %12p",&(position),
                            &(possize));
                    MesPrint("    in buffer:
%l",(AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD));
                }
#endif
                *AM.S0->sBuffer = 0; firstterm = -1;
                cbo = cpo = AM.S0->sBuffer;
                do {
                    WORD *oldipointer = AR.CompressPointer;
                    WORD *oldibuffer = AR.CompressBuffer;
                    WORD *comprtop = AR.ComprTop;
                    AR.ComprTop = AM.S0->sTop;
                    AR.CompressPointer = cpo;
                    AR.CompressBuffer = cbo;
                    if ( firstterm > 0 ) {
                        if ( PutOut(BHEAD term,&position,AR.hidefile,1) < 0 ) goto ProcErr;
                    }
                    else if ( firstterm < 0 ) {
                        if ( PutOut(BHEAD term,&position,AR.hidefile,0) < 0 ) goto ProcErr;
                        firstterm++;
                    }
                    else {
                        if ( PutOut(BHEAD term,&position,AR.hidefile,-1) < 0 ) goto ProcErr;
                        firstterm++;
                    }
                    cpo = AR.CompressPointer;
                    cbo = AR.CompressBuffer;
                    AR.CompressPointer = oldipointer;
                    AR.CompressBuffer = oldibuffer;
                    AR.ComprTop = comprtop;
                } while ( GetTerm(BHEAD term) );
#ifdef HIDEDEBUG
                if ( AR.hidefile->handle >= 0 ) {
                    POSITION possize,pos;
                    PUTZERO(possize);
                    PUTZERO(pos);
                    SeekFile(AR.hidefile->handle,&pos,SEEK_CUR);
                    SeekFile(AR.hidefile->handle,&possize,SEEK_END);
                    SeekFile(AR.hidefile->handle,&pos,SEEK_SET);
                    MesPrint("Processor Hide2: filesize(th) = %12p, filesize(ex) = %12p",&(position),
                            &(possize));
                    MesPrint("    in buffer:
%l",(AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD));
                }
#endif
                if ( FlushOut(&position,AR.hidefile,1) ) goto ProcErr;
                AR.hidefile->POfull = AR.hidefile->POfill;
#ifdef HIDEDEBUG
                if ( AR.hidefile->handle >= 0 ) {
                    POSITION possize,pos;
                    PUTZERO(possize);
                    PUTZERO(pos);
                    SeekFile(AR.hidefile->handle,&pos,SEEK_CUR);
                    SeekFile(AR.hidefile->handle,&possize,SEEK_END);
                    SeekFile(AR.hidefile->handle,&pos,SEEK_SET);
                    MesPrint("Processor Hide3: filesize(th) = %12p, filesize(ex) = %12p",&(position),
                            &(possize));
                    MesPrint("    in buffer:
%l",(AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD));
                }
#endif
/*
                Because we direct the e->onfile already to the hide file, we
                need to change the status of the expression. Otherwise the use
                of parts (or the whole) of the expression looks in the infile
                while the position is that of the hide file.
                We choose to get everything from the hide file. On average that
                should give least file activity.
*/
                if ( e->status == HIDELEXPRESSION ) {
                    e->status = HIDDENLEXPRESSION;
                    AS.OldOnFile[i] = e->onfile;
                    AS.OldNumFactors[i] = Expressions[i].numfactors;
                }
                if ( e->status == HIDEGEXPRESSION ) {
                    e->status = HIDDENGEXPRESSION;
                    AS.OldOnFile[i] = e->onfile;
                    AS.OldNumFactors[i] = Expressions[i].numfactors;
                }
#ifdef WITHPTHREADS
                SetHideFiles();
#endif
                UpdateMaxSize();
                break;
```

```
            case DROPPEDEXPRESSION:
            case DROPLEXPRESSION:
            case DROPGEXPRESSION:
            case DROPHLEXPRESSION:
            case DROPHGEXPRESSION:
            case STOREDEXPRESSION:
            case HIDDENLEXPRESSION:
            case HIDDENGEXPRESSION:
            case SPECTATOREXPRESSION:
            default:
                break;
        }
        }
        AR.KeptInHold = 0;
    }
    AR.DeferFlag = 0;
    AT.WorkPointer = term;
#ifdef HIDEDEBUG
    MesPrint("Status at the end of Processor (HideLevel = %d)",AC.HideLevel);
    for ( i = 0; i < NumExpressions; i++ ) {
        e = Expressions+i;
        ExprStatus(e);
    }
#endif
    return(retval);
ProcErr:
    AT.WorkPointer = term;
    if ( AM.tracebackflag ) MesCall("Processor");
    return(-1);
}
/*
        #] Processor :
        #[ TestSub :            WORD TestSub(term,level)
*/
WORD TestSub(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    WORD *m, *t, *r, retvalue, funflag, j, oldncmod, nexpr;
    WORD *stop, *t1, *t2, funnum, wilds, tbufnum, stilldirty = 0;
    NESTING n;
    CBUF *C = cbuf+AT.ebufnum;
    LONG isp, i;
    TABLES T;
    VOID *oldcompareroutine = AR.CompareRoutine;
    WORD oldsorttype = AR.SortType;
ReStart:
    tbufnum = 0; i = 0;
    AT.TMbuff = AM.rbufnum;
    funflag = 0;
    t = term;
    r = t + *t - 1;
    m = r - ABS(*r) + 1;
    t++;
    if ( t < m ) do {
        if ( *t == SUBEXPRESSION ) {
            /*
                Subexpression encountered
                There may be more than one.
                The old strategy was to take the last.
                A newer strategy was to take the lowest power first.
                The current strategy is that we compute the number of terms
                generated by this subexpression and take the minimum of that.
            */

#ifdef WHICHSUBEXPRESSION

            WORD *tmin = t, AN.nbino;
/*          LONG minval = MAXLONG; */
            LONG minval = -1;
            LONG mm, mnum1 = 1;
            if ( AN.BinoScrat == 0 ) {
                AN.BinoScrat = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"GetBinoScrat");
            }
#endif
            if ( t[3] ) {
                r = t + t[1];
                while ( AN.subsubveto == 0 &&
                            *r == SUBEXPRESSION && r < m && r[3] ) {
#ifdef WHICHSUBEXPRESSION
                    mnum1++;
#endif
                    if ( r[1] == t[1] && r[2] == t[2] && r[4] == t[4] ) {
                        j = t[1] - SUBEXPSIZE;
                        t1 = t + SUBEXPSIZE;
                        t2 = r + SUBEXPSIZE;
                        while ( j > 0 && *t1++ == *t2++ ) j--;
                        if ( j <= 0 ) {
```

```
                            t[3] += r[3];
                            if ( t[3] == 0 ) {
                                t1 = r + r[1];
                                t2 = term + *term;
                                *term -= r[1]+t[1];
                                r = t;
                                while ( t1 < t2 ) *r++ = *t1++;
                                goto ReStart;
                            }
                            else {
                                t1 = r + r[1];
                                t2 = term + *term;
                                *term -= r[1];
                                m -= r[1];
                                while ( t1 < t2 ) *r++ = *t1++;
                                r = t;
                            }
                        }
                    }
#ifdef WHICHSUBEXPRESSION

                    else if ( t[2] >= 0 ) {
/*
                        Compute Binom(numterms+power-1,power-1)
                        We need potentially long arrithmetic.
                        That is why we had to allocate AN.BinoScrat
*/
                        if ( AN.last1 == t[3] && AN.last2 == cbuf[t[4]].NumTerms[t[2]] + t[3] - 1 ) {
                            if ( AN.last3 > minval ) {
                                minval = AN.last3; tmin = t;
                            }
                        }
                        else {
                        AN.last1 = t[3]; mm = AN.last2 = cbuf[t[4]].NumTerms[t[2]] + t[3] - 1;
                        if ( t[3] == 1 ) {
                            if ( mm > minval ) {
                                minval = mm; tmin = t;
                            }
                        }
                        else if ( t[3] > 0 ) {
                            if ( mm > MAXPOSITIVE ) goto TooMuch;
                            GetBinom(AN.BinoScrat,&AN.nbino,(WORD)mm,t[3]);
                            if ( AN.nbino > 2 ) goto TooMuch;
                            if ( AN.nbino == 2 ) {
                                mm = AN.BinoScrat[1];
                                mm = ( mm « BITSINWORD ) + AN.BinoScrat[0];
                            }
                            else if ( AN.nbino == 1 ) mm = AN.BinoScrat[0];
                            else mm = 0;
                            if ( mm > minval ) {
                                minval = mm; tmin = t;
                            }
                        }
                        AN.last3 = mm;
                        }
                    }
#endif
                    t = r;
                    r += r[1];
                }
#ifdef WHICHSUBEXPRESSION
                if ( mnum1 > 1 && t[2] >= 0 ) {
/*
                    To keep the flowcontrol simple we duplicate some code here
*/
                    if ( AN.last1 == t[3] && AN.last2 == cbuf[t[4]].NumTerms[t[2]] + t[3] - 1 ) {
                        if ( AN.last3 > minval ) {
                            minval = AN.last3; tmin = t;
                        }
                    }
                    else {
                    AN.last1 = t[3]; mm = AN.last2 = cbuf[t[4]].NumTerms[t[2]] + t[3] - 1;
                    if ( t[3] == 1 ) {
                        if ( mm > minval ) {
                            minval = mm; tmin = t;
                        }
                    }
                    else if ( t[3] > 0 ) {
                        if ( mm > MAXPOSITIVE ) {
/*
                            We will generate more terms than we can count
*/
TooMuch:;
                            MLOCK(ErrorMessageLock);
                            MesPrint("Attempt to generate more terms than FORM can count");
                            MUNLOCK(ErrorMessageLock);
                            Terminate(-1);
```

```
                        }
                        GetBinom(AN.BinoScrat,&AN.nbino,(WORD)mm,t[3]);
                        if ( AN.nbino > 2 ) goto TooMuch;
                        if ( AN.nbino == 2 ) {
                            mm = AN.BinoScrat[1];
                            mm = ( mm « BITSINWORD ) + AN.BinoScrat[0];
                        }
                        else if ( AN.nbino == 1 ) mm = AN.BinoScrat[0];
                        else mm = 0;
                        if ( mm > minval ) {
                            minval = mm; tmin = t;
                        }
                    }
                    AN.last3 = mm;
                    }
                }
                t = tmin;
#endif
/*              AR.TePos = 0;  */
                AR.TePos = WORDDIF(t,term);
                AT.TMbuff = t[4];
                if ( t[4] == AM.dbufnum && (t+t[1]) < m && t[t[1]] == DOLLAREXPR2 ) {
                    if ( t[t[1]+2] < 0 ) AT.TMdolfac = -t[t[1]+2];
                    else {  /* resolve the element number */
                        AT.TMdolfac = GetDolNum(BHEAD t+t[1],m)+1;
                    }
                }
                else AT.TMdolfac = 0;
                if ( t[3] < 0 ) {
                    AN.TeInFun = 1;
                    AR.TePos = WORDDIF(t,term);
                    return(t[2]);
                }
                else {
                    AN.TeInFun = 0;
                    AN.TeSuOut = t[3];
                }
                if ( t[2] < 0 ) {
                    AN.TeSuOut = -t[3];
                    return(-t[2]);
                }
                return(t[2]);
            }
        }
        else if ( *t == EXPRESSION ) {
            WORD *toTMaddr;
            i = -t[2] - 1;
            if ( t[3] < 0 ) {
                AN.TeInFun = 1;
                AR.TePos = WORDDIF(t,term);
                return(i);
            }
            nexpr = t[3];
            toTMaddr = m = AT.WorkPointer;
            AN.Frozen = 0;
/*
            We have to be very careful with respect to setting variables
            like AN.TeInFun, because we may still call Generator and that
            may change those variables. That is why we set them at the
            last moment only.
*/
            j = t[1];
            AT.WorkPointer += j;
            r = t;
            NCOPY(m,r,j);
            r = t + t[1];
            t += SUBEXPSIZE;
            while ( t < r ) {
                if ( *t == FROMBRAC ) {
                    WORD *ttstop,*tttstop;
/*
                    Note: Convention is that wildcards are done
                    after the expression has been picked up. So
                    no wildcard substitutions are needed here.
*/
                    t += 2;
                    AN.Frozen = m = AT.WorkPointer;
/*
                    We should check now for subexpressions and if necessary
                    we substitute them. Keep in mind: only one term allowed!

                    In retrospect (26-jan-2010): take also functions that
                    have a dirty flag on
*/
                    j = *t; tttstop = t + j;
                    GETSTOP(t,ttstop);
                    *m++ = j; t++;
```

```
                    while ( t < ttstop ) {
                        if ( *t == SUBEXPRESSION ) break;
                        if ( *t >= FUNCTION && ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) ) break;
                        j = t[1]; NCOPY(m,t,j);
                    }
                    if ( t < ttstop ) {
/*
                        We ran into a subexpression or a function with a
                        'dirty' argument. It could also be a $ or
                        just e[(a^2)*b]. In all cases we should evaluate
*/
                        while ( t < tttstop ) *m++ = *t++;
                        *AT.WorkPointer = m-AT.WorkPointer;
                        m = AT.WorkPointer;
                        AT.WorkPointer = m + *m;
                        NewSort(BHEAD0);
                        if ( Generator(BHEAD m,AR.Cnumlhs) ) {
                            LowerSortLevel(); goto EndTest;
                        }
                        if ( EndSort(BHEAD m,0) < 0 ) goto EndTest;
                        AN.Frozen = m;
                        if ( *m == 0 ) {
                            *m++ = 4; *m++ = 1; *m++ = 1; *m++ = 3;
                        }
                        else if ( m[*m] != 0 ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Bracket specification in expression should be one single term");
                            MUNLOCK(ErrorMessageLock);
                            Terminate(-1);
                        }
                        else {
                            m += *m;
                            m -= ABS(m[-1]);
                            *m++ = 1; *m++ = 1; *m++ = 3;
                            *AN.Frozen = m - AN.Frozen;
                        }
                    }
                    else {
                        while ( t < tttstop ) *m++ = *t++;
                        *AT.WorkPointer = m-AT.WorkPointer;
                        m = AT.WorkPointer;
                        AT.WorkPointer = m + *m;
                        if ( Normalize(BHEAD m) ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Error while picking up contents of bracket");
                            MUNLOCK(ErrorMessageLock);
                            Terminate(-1);
                        }
                        if ( !*m ) {
                            *m++ = 4; *m++ = 1; *m++ = 1; *m++ = 3;
                        }
                        else m += *m;
                    }
                    AT.WorkPointer = m;
                    break;
                }
                t += t[1];
            }
            AN.TeInFun = 0;
            AR.TePos = 0;
            AN.TeSuOut = nexpr;
            AT.TMaddr = toTMaddr;
            return(i);
        }
        else if ( *t >= FUNCTION ) {
            if ( t[0] == EXPONENT ) {
                if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SYMBOL &&
                t[FUNHEAD+2] == -SNUMBER && t[FUNHEAD+3] < MAXPOWER
                && t[FUNHEAD+3] > -MAXPOWER ) {
                    t[0] = SYMBOL;
                    t[1] = 4;
                    t[2] = t[FUNHEAD+1];
                    t[3] = t[FUNHEAD+3];
                    r = term + *term;
                    m = t + FUNHEAD+4;
                    t += 4;
                    while ( m < r ) *t++ = *m++;
                    *term = WORDDIF(t,term);
                    goto ReStart;
                }
                else if ( t[1] == FUNHEAD+ARGHEAD+11 && t[FUNHEAD] == ARGHEAD+9
                 && t[FUNHEAD+ARGHEAD] == 9 && t[FUNHEAD+ARGHEAD+1] == DOTPRODUCT
                 && t[FUNHEAD+ARGHEAD+8] == 3
                 && t[FUNHEAD+ARGHEAD+7] == 1
                 && t[FUNHEAD+ARGHEAD+6] == 1
                 && t[FUNHEAD+ARGHEAD+5] == 1
                 && t[FUNHEAD+ARGHEAD+9] == -SNUMBER
```

```
                 && t[FUNHEAD+ARGHEAD+10] < MAXPOWER
                 && t[FUNHEAD+ARGHEAD+10] > -MAXPOWER ) {
                    t[0] = DOTPRODUCT;
                    t[1] = 5;
                    t[2] = t[FUNHEAD+ARGHEAD+3];
                    t[3] = t[FUNHEAD+ARGHEAD+4];
                    t[4] = t[FUNHEAD+ARGHEAD+10];
                    r = term + *term;
                    m = t + FUNHEAD+ARGHEAD+11;
                    t += 5;
                    while ( m < r ) *t++ = *m++;
                    *term = WORDDIF(t,term);
                    goto ReStart;
                }
            }
            funnum = *t;
            if ( *t >= FUNCTION + WILDOFFSET ) funnum -= WILDOFFSET;
            if ( *t == EXPONENT ) {
/*
                Test whether the second argument is an integer
*/
                r = t+FUNHEAD;
                NEXTARG(r)
                if ( *r == -SNUMBER && r[1] < MAXPOWER && r+2 == t+t[1] &&
                t[FUNHEAD] > -FUNCTION && ( t[FUNHEAD] != -SNUMBER
                || t[FUNHEAD+1] != 0 ) && t[FUNHEAD] != ARGHEAD ) {
                  if ( r[1] == 0 ) {
                    if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Encountered 0^0. Fatal error.");
                        MUNLOCK(ErrorMessageLock);
                        SETERROR(-1);
                    }
                    *t = DUMMYFUN;
/*
                        Now mark it clean to avoid further interference.
                        Normalize will remove this object.
*/
                    t[2] = 0;
                  }
                  else {
                    /* Note that the case 0^ is treated in Normalize */

                    t1 = AddRHS(AT.ebufnum,1);
                    m = t + FUNHEAD;
                    if ( *m > 0 ) {
                        m += ARGHEAD;
                        i = t[FUNHEAD] - ARGHEAD;
                        while ( (t1 + i + 10) > C->Top )
                            t1 = DoubleCbuffer(AT.ebufnum,t1,9);
                        while ( --i >= 0 ) *t1++ = *m++;
                    }
                    else {
                        if ( (t1 + 20) > C->Top )
                            t1 = DoubleCbuffer(AT.ebufnum,t1,10);
                        ToGeneral(m,t1,1);
                        t1 += *t1;
                    }
                    *t1++ = 0;
                    C->rhs[C->numrhs+1] = t1;
                    C->Pointer = t1;

                    /* No provisions yet for commuting objects */

                    C->CanCommu[C->numrhs] = 1;
                    *t++ = SUBEXPRESSION;
                    *t++ = SUBEXPSIZE;
                    *t++ = C->numrhs;
                    *t++ = r[1];
                    *t++ = AT.ebufnum;
#if SUBEXPSIZE > 5
Important: we may not have enough spots here
#endif
                    FILLSUB(t)  /* Important: We have maybe only 5 spots! */
                    r += 2;
                    m = term + *term;
                    do { *t++ = *r++; } while ( r < m );
                    *term -= WORDDIF(r,t);
                    goto ReStart;
                  }
                }
            }
            else if ( *t == SUMF1 || *t == SUMF2 ) {
/*
                What we are looking for is:
                1-st argument:  Single symbol or index.
                2-nd argument:  Number.
```

```
                3-rd argument:  Number.
                (4-th argument):Number.
                One more argument.
                This would activate the summation procedure.
                Note that the initiated recursion here can be done
                without upsetting the regular procedures.
*/
                WORD *tstop, lcounter, lcmin, lcmax, lcinc;
                tstop = t + t[1];
                r = t+FUNHEAD;
                if ( r+6 < tstop && r[2] == -SNUMBER && r[4] == -SNUMBER
                && ( ( r[0] == -SYMBOL )
                || ( r[0] == -INDEX && r[1] >= AM.OffsetIndex
                && r[3] >= 0 && r[3] < AM.OffsetIndex
                && r[5] >= 0 && r[5] < AM.OffsetIndex ) ) ) {
                    lcounter = r[0] == -INDEX ? -r[1]: r[1]; /* The loop counter */
                    lcmin = r[3];
                    lcmax = r[5];
                    r += 6;
                    if ( *r == -SNUMBER && r+2 < tstop ) {
                        lcinc = r[1];
                        r += 2;
                    }
                    else lcinc = 1;
                    if ( r < tstop && ( ( *r > 0 && (r+*r) == tstop )
                    || ( *r <= -FUNCTION && r+1 == tstop )
                    || ( *r > -FUNCTION && *r < 0 && r+2 == tstop ) ) ) {
                        m = AddRHS(AT.ebufnum,1);
                        if ( *r > 0 ) {
                            i = *r - ARGHEAD;
                            r += ARGHEAD;
                            while ( (m + i + 10) > C->Top )
                                m = DoubleCbuffer(AT.ebufnum,m,11);
                            while ( --i >= 0 ) *m++ = *r++;
                        }
                        else {
                            while ( (m + 20) > C->Top )
                                m = DoubleCbuffer(AT.ebufnum,m,12);
                            ToGeneral(r,m,1);
                            m += *m;
                        }
                        *m++ = 0;
                        C->rhs[C->numrhs+1] = m;
                        C->Pointer = m;
                        m = AT.TMout;
                        *m++ = 6;
                        if ( *t == SUMF1 ) *m++ = SUMNUM1;
                        else               *m++ = SUMNUM2;
                        *m++ = lcounter;
                        *m++ = lcmin;
                        *m++ = lcmax;
                        *m++ = lcinc;
                        m = t + t[1];
                        r = C->rhs[C->numrhs];
/*
                        Test now if the argument was already evaluated.
                        In that case it needs a new subexpression prototype.
                        In either case we replace the function now by a
                        subexpression prototype.
*/
                        if ( *r >= (SUBEXPSIZE+4)
                        && ABS(*(r+*r-1)) < (*r - 1)
                        && r[1] == SUBEXPRESSION ) {
                            r++;
                            i = r[1] - 5;
                            *t++ = *r++; *t++ = *r++; *t++ = C->numrhs;
                            r++; *t++ = *r++; *t++ = AT.ebufnum; r++;
                            while ( --i >= 0 ) *t++ = *r++;
                        }
                        else {
                            *t++ = SUBEXPRESSION;
                            *t++ = 4+SUBEXPSIZE;
                            *t++ = C->numrhs;
                            *t++ = 1;
                            *t++ = AT.ebufnum;
                            FILLSUB(t)
                            if ( lcounter < 0 ) {
                                *t++ = INDTOIND;
                                *t++ = 4;
                                *t++ = -lcounter;
                            }
                            else {
                                *t++ = SYMTONUM;
                                *t++ = 4;
                                *t++ = lcounter;
                            }
                            *t++ = lcmin;
```

```
                        }
                        t2 = term + *term;
                        while ( m < t2 ) *t++ = *m++;
                        *term = WORDDIF(t,term);
                        AN.TeInFun = -C->numrhs;
                        AR.TePos = 0;
                        AN.TeSuOut = 0;
                        AT.TMbuff = AT.ebufnum;
                        return(C->numrhs);
                    }
                }
            }
            if ( functions[funnum-FUNCTION].spec == 0
                || ( t[2] & (DIRTYFLAG|MUSTCLEANPRF) ) != 0 ) { funflag = 1; }
            if ( *t <= MAXBUILTINFUNCTION ) {
              if ( *t <= DELTAP && *t >= THETA ) { /* Speeds up by 2 or 3 compares */
              if ( *t == THETA || *t == THETA2 ) {
                WORD *tstop, *tt2, kk;
                tstop = t + t[1];
                tt2 = t + FUNHEAD;
                while ( tt2 < tstop ) {
                    if ( *tt2 > 0 && tt2[1] != 0 ) goto DoSpec;
                    NEXTARG(tt2)
                }
                if ( !AT.RecFlag ) {
                    if ( ( kk = DoTheta(BHEAD t) ) == 0 ) {
                        *term = 0;
                        return(0);
                    }
                    else if ( kk > 0 ) {
                        m = t + t[1];
                        r = term + *term;
                        while ( m < r ) *t++ = *m++;
                        *term = WORDDIF(t,term);
                        goto ReStart;
                    }
                }
              }
              else if ( *t == DELTA2 || *t == DELTAP ) {
                WORD *tstop, *tt2, kk;
                tstop = t + t[1];
                tt2 = t + FUNHEAD;
                while ( tt2 < tstop ) {
                    if ( *tt2 > 0 && tt2[1] != 0 ) goto DoSpec;
                    NEXTARG(tt2)
                }
                if ( !AT.RecFlag ) {
                    if ( ( kk = DoDelta(t) ) == 0 ) {
                        *term = 0;
                        return(0);
                    }
                    else if ( kk > 0 ) {
                        m = t + t[1];
                        r = term + *term;
                        while ( m < r ) *t++ = *m++;
                        *term = WORDDIF(t,term);
                        goto ReStart;
                    }
                }
              } }
              else if ( *t == DISTRIBUTION && t[FUNHEAD] == -SNUMBER
              && t[FUNHEAD+1] >= -2 && t[FUNHEAD+1] <= 2
              && t[FUNHEAD+2] == -SNUMBER
              && t[FUNHEAD+4] <= -FUNCTION
              && t[FUNHEAD+5] <= -FUNCTION ) {
                WORD *ttt = t+FUNHEAD+6, *tttstop = t+t[1];
                while ( ttt < tttstop ) {
                    if ( *ttt == -DOLLAREXPRESSION ) break;
                    NEXTARG(ttt);
                }
                if ( ttt >= tttstop ) {
                    AN.TeInFun = -1;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
              }
              else if ( *t == DELTA3 && ((t[1]-FUNHEAD) & 1 ) == 0 ) {
                AN.TeInFun = -2;
                AN.TeSuOut = 0;
                AR.TePos = -1;
                return(1);
              }
              else if ( ( *t == TABLEFUNCTION ) && ( t[FUNHEAD] <= -FUNCTION )
              && ( T = functions[-t[FUNHEAD]-FUNCTION].tabl ) != 0
              && ( t[1] >= FUNHEAD+1+2*T->numind )
              && ( t[FUNHEAD+1] == -SYMBOL ) ) {
```

```
/*
                The case of table_(tab,sym1,...,symn)
*/
                for ( isp = 0; isp < T->numind; isp++ ) {
                    if ( t[FUNHEAD+1+2*isp] != -SYMBOL ) break;
                }
                if ( isp >= T->numind ) {
                    AN.TeInFun = -3;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
              }
              else if ( *t == TABLEFUNCTION && t[FUNHEAD] <= -FUNCTION
              && ( T = functions[-t[FUNHEAD]-FUNCTION].tabl ) != 0
              && ( t[1] == FUNHEAD+2 )
              && ( t[FUNHEAD+1] <= -FUNCTION ) ) {
/*
                The case of table_(tab,fun)
*/
                AN.TeInFun = -3;
                AN.TeSuOut = 0;
                AR.TePos = -1;
                return(1);
              }
              else if ( *t == FACTORIN ) {
                if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -DOLLAREXPRESSION ) {
                    AN.TeInFun = -4;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
                else if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -EXPRESSION ) {
                    AN.TeInFun = -5;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
              }
              else if ( *t == TERMSINBRACKET ) {
                if ( t[1] == FUNHEAD || (
                     t[1] == FUNHEAD+2
                    && t[FUNHEAD] == -SNUMBER
                    && t[FUNHEAD+1] == 0
                 ) ) {
                    AN.TeInFun = -6;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
/*
                The other cases have not yet been implemented
                We still have to add the case of short arguments
                First the different bracket in same expression

                else if ( t[1] > FUNHEAD+ARGHEAD
                        && t[FUNHEAD] == t[1]-FUNHEAD
                        && t[FUNHEAD+ARGHEAD] == t[1]-FUNHEAD-ARGHEAD
                        && t[t[1]-1] == 3
                        && t[t[1]-2] == 1
                        && t[t[1]-3] == 1 ) {
                    AN.TeInFun = -6;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }

                Next the bracket in an other expression

                else if ( t[1] > FUNHEAD+ARGHEAD+2
                        && t[FUNHEAD] == -EXPRESSION
                        && t[FUNHEAD+2] == t[1]-FUNHEAD-2
                        && t[FUNHEAD+ARGHEAD+2] == t[1]-FUNHEAD-ARGHEAD-2
                        && t[t[1]-1] == 3
                        && t[t[1]-2] == 1
                        && t[t[1]-3] == 1 ) {
                    AN.TeInFun = -6;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
*/
              }
              else if ( *t == EXTRASYMFUN ) {
                if ( t[1] == FUNHEAD+2 && (
                    ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] <= cbuf[AM.sbufnum].numrhs
                            && t[FUNHEAD+1] > 0 ) ||
```

```
                    ( t[FUNHEAD] == -SYMBOL && t[FUNHEAD+1] < MAXVARIABLES
                            && t[FUNHEAD+1] >= MAXVARIABLES-cbuf[AM.sbufnum].numrhs ) ) ) {
                    AN.TeInFun = -7;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
                else if ( t[1] == FUNHEAD ) {
                    AN.TeInFun = -7;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
            }
            else if ( *t == DIVFUNCTION || *t == REMFUNCTION
                      || *t == INVERSEFUNCTION || *t == MULFUNCTION
                      || *t == GCDFUNCTION ) {
                WORD *tf;
                int todo = 1, numargs = 0;
                tf = t + FUNHEAD;
                while ( tf < t + t[1] ) {
                    DOLLARS d;
                    if ( *tf == -DOLLAREXPRESSION ) {
                        d = Dollars + tf[1];
                        if ( d->type == DOLWILDARGS ) {
                            WORD *tterm = AT.WorkPointer, *tw;
                            WORD *ta = term, *tb = tterm, *tc, *td = term + *term;
                            while ( ta < t ) *tb++ = *ta++;
                            tc = tb;
                            while ( ta < tf ) *tb++ = *ta++;
                            tw = d->where+1;
                            while ( *tw ) {
                                if ( *tw < 0 ) {
                                    if ( *tw > -FUNCTION ) *tb++ = *tw++;
                                    *tb++ = *tw++;
                                }
                                else {
                                    int ia;
                                    for ( ia = 0; ia < *tw; ia++ ) *tb++ = *tw++;
                                }
                            }
                            NEXTARG(ta)
                            while ( ta < t+t[1] ) *tb++ = *ta++;
                            tc[1] = tb-tc;
                            while ( ta < td ) *tb++ = *ta++;
                            *tterm = tb - tterm;
                            {
                                int ia, na = *tterm;
                                ta = tterm; tb  = term;
                                for ( ia = 0; ia < na; ia++ ) *tb++ = *ta++;
                            }
                            if ( tb > AT.WorkTop ) {
                                MLOCK(ErrorMessageLock);
                                MesWork();
                                goto EndTest2;
                            }
                            AT.WorkPointer = tb;
                            goto ReStart;
                        }
                    }
                    NEXTARG(tf);
                }
                tf = t + FUNHEAD;
                while ( tf < t + t[1] ) {
                    numargs++;
                    if ( *tf > 0 && tf[1] != 0 ) todo = 0;
                    NEXTARG(tf);
                }
                if ( todo && numargs == 2 ) {
                    if ( *t == DIVFUNCTION ) AN.TeInFun = -9;
                    else if ( *t == REMFUNCTION ) AN.TeInFun = -10;
                    else if ( *t == INVERSEFUNCTION ) AN.TeInFun = -11;
                    else if ( *t == MULFUNCTION ) AN.TeInFun = -14;
                    else if ( *t == GCDFUNCTION ) AN.TeInFun = -8;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
                else if ( todo && *t == GCDFUNCTION ) {
                    AN.TeInFun = -8;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
            }
            else if ( *t == PERMUTATIONS && ( ( t[1] >= FUNHEAD+1
              && t[FUNHEAD] <= -FUNCTION ) || ( t[1] >= FUNHEAD+3
```

```
                && t[FUNHEAD] == -SNUMBER && t[FUNHEAD+2] <= -FUNCTION ) ) ) {
                AN.TeInFun = -12;
                AN.TeSuOut = 0;
                AR.TePos = -1;
                return(1);
              }
              else if ( *t == PARTITIONS ) {
                if ( TestPartitions(BHEAD t,&(AT.partitions)) ) {
                    AT.partitions.where = t-term;
                    AN.TeInFun = -13;
                    AN.TeSuOut = 0;
                    AR.TePos = -1;
                    return(1);
                }
              }
            }
        }
        t += t[1];
    } while ( t < m );
    if ( funflag ) {    /* Search in functions */
DoSpec:
        t = term;
        AT.NestPoin->termsize = t;
        if ( AT.NestPoin == AT.Nest ) AN.EndNest = t + *t;
        t++;
        oldncmod = AN.ncmod;
        if ( t < m ) do {
            if ( *t < FUNCTION ) {
                t += t[1]; continue;
            }
            if ( AN.ncmod && ( ( AC.modmode & ALSOFUNARGS ) == 0 ) ) {
                if ( *t != AR.PolyFun ) AN.ncmod = 0;
                else                    AN.ncmod = oldncmod;
            }
            r = t + t[1];
            funnum = *t;
            if ( *t >= FUNCTION + WILDOFFSET ) funnum -= WILDOFFSET;
            if ( ( *t == NUMFACTORS || *t == FIRSTTERM || *t == CONTENTTERM )
             && t[1] == FUNHEAD+2 &&
            ( t[FUNHEAD] == -EXPRESSION || t[FUNHEAD] == -DOLLAREXPRESSION ) ) {
/*
                if ( *t == NUMFACTORS ) {
                    This we leave for Normalize
                }
*/
            }
            else if ( functions[funnum-FUNCTION].spec == 0 ) {
                AT.NestPoin->funsize = t + 1;
                t1 = t;
                t += FUNHEAD;
                while ( t < r ) {   /* Sum over arguments */
                    if ( *t > 0 && t[1] ) { /* Argument is dirty  */
                        AT.NestPoin->argsize = t;
                        AT.NestPoin++;
/*                      stop = t + *t; */
                        t2 = t;
                        t += ARGHEAD;
                        while ( t < AT.NestPoin[-1].argsize+*(AT.NestPoin[-1].argsize) ) {
                                                /* Sum over terms */
                            AT.RecFlag++;
                            i = *t;
                            AN.subsubveto = 1;
/*
                            AN.subsubveto repairs a bug that became apparent
                            in an example by York Schroeder:
                                f(k1.k1)*replace_(k1,2*k2)
                            Is it possible to repair the counting of the various
                            length indicators? (JV 1-jun-2010)
*/
                            if ( ( retvalue = TestSub(BHEAD t,level) ) != 0 ) {
/*
                                Possible size changes:
                                Note defs at 471,467,460,400,425,328
*/
redosize:
                                if ( i > *t ) {
                                    i -= *t;
                                    *t2 -= i;
                                    t1[1] -= i;
                                    t += *t;
                                    r = t + i;
                                    m = term + *term;
                                    while ( r < m ) *t++ = *r++;
                                    *term -= i;
                                }
                                AN.subsubveto = 0;
                                t1[2] = 1;
```

```
                                if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 )
                                    t1[2] |= MUSTCLEANPRF;
                                AT.RecFlag--;
                                AT.NestPoin--;
                                AN.TeInFun++;
                                AR.TePos = 0;
                                AN.ncmod = oldncmod;
                                return(retvalue);
                            }
                            else {
                                /*
                                 * Somehow the next line fixes Issue #106.
                                 */
                                i = *t;
                                Normalize(BHEAD t);
/*                              if ( i > *t ) { retvalue = 1; goto redosize; } */
                                /*
                                 * Experimentally, the next line fixes Issue #105.
                                 */
                                if ( *t == 0 ) { retvalue = 1; goto redosize; }
                                {
                                    WORD *tend = t + *t, *tt = t+1;
                                    stilldirty = 0;
                                    tend -= ABS(tend[-1]);
                                    while ( tt < tend ) {
                                        if ( *tt == SUBEXPRESSION ) {
                                            stilldirty = 1; break;
                                        }
                                        tt += tt[1];
                                    }
                                }
                                if ( i > *t ) {
                                    retvalue = 1;
                                    i -= *t;
                                    *t2 -= i;
                                    t1[1] -= i;
                                    t += *t;
                                    r = t + i;
                                    m = term + *term;
                                    while ( r < m ) *t++ = *r++;
                                    *term -= i;
                                    t = AT.NestPoin[-1].argsize + ARGHEAD;
                                }
                            }
                            AN.subsubveto = 0;
                            AT.RecFlag--;
                            t += *t;
                        }
                        AT.NestPoin--;
/*                      Argument contains no subexpressions.
                        It should be normalized and sorted.
                        The main problem is the storage.
*/
                        t = AT.NestPoin->argsize;
                        j = *t;
                        t += ARGHEAD;
                        NewSort(BHEAD0);
                        if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
                            AR.CompareRoutine = &CompareSymbols;
                            AR.SortType = SORTHIGHFIRST;
                        }
                        if ( AT.WorkPointer < term + *term )
                            AT.WorkPointer = term + *term;

                        while ( t < AT.NestPoin->argsize++*(AT.NestPoin->argsize) ) {
                            m = AT.WorkPointer;
                            r = t + *t;
                            do { *m++ = *t++; } while ( t < r );
                            r = AT.WorkPointer;
                            AT.WorkPointer = r + *r;
                            if ( Normalize(BHEAD r) ) {
                                if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
                                    AR.SortType = oldsorttype;
                                    AR.CompareRoutine = oldcompareroutine;
                                    t1[2] |= MUSTCLEANPRF;
                                }
                                LowerSortLevel(); goto EndTest;
                            }
                            if ( AN.ncmod != 0 ) {
                                if ( *r ) {
                                    if ( Modulus(r) ) {
                                        LowerSortLevel();
                                        AT.WorkPointer = r;
                                        if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
                                            AR.SortType = oldsorttype;
                                            AR.CompareRoutine = oldcompareroutine;
```

```
                                                t1[2] |= MUSTCLEANPRF;
                                            }
                                            goto EndTest;
                                        }
                                    }
                                }
                                if ( AR.PolyFun > 0 ) {
                                    if ( PrepPoly(BHEAD r,1) != 0 ) goto EndTest;
                                }
                                if ( *r ) StoreTerm(BHEAD r);
                                AT.WorkPointer = r;
                            }
/* the next call had parameter 0. That was wrong!!!!!) */
                            if ( EndSort(BHEAD AT.WorkPointer+ARGHEAD,1) < 0 ) goto EndTest;
                            m = AT.WorkPointer+ARGHEAD;
                            if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
                                AR.SortType = oldsorttype;
                                AR.CompareRoutine = oldcompareroutine;
                                t1[2] |= MUSTCLEANPRF;
                            }
                            while ( *m ) m += *m;
                            i = WORDDIF(m,AT.WorkPointer);
                            *AT.WorkPointer = i;
                            AT.WorkPointer[1] = stilldirty;
                            if ( ToFast(AT.WorkPointer,AT.WorkPointer) ) {
                                m = AT.WorkPointer;
                                if ( *m <= -FUNCTION ) { m++; i = 1; }
                                else { m += 2; i = 2; }
                            }
                            j = i - j;
                            if ( j > 0 ) {
                                r = m + j;
                                if ( r > AT.WorkTop ) {
                                    MLOCK(ErrorMessageLock);
                                    MesWork();
                                    goto EndTest2;
                                }
                                do { *--r = *--m; } while ( m > AT.WorkPointer );
                                AT.WorkPointer = r;
                                m = AN.EndNest;
                                r = m + j;
                                stop = AT.NestPoin->argsize+*(AT.NestPoin->argsize);
                                do { *--r = *--m; } while ( m >= stop );
                            }
                            else if ( j < 0 ) {
                                m = AT.NestPoin->argsize+*(AT.NestPoin->argsize);
                                r = m + j;
                                do { *r++ = *m++; } while ( m < AN.EndNest );
                            }
                            m = AT.NestPoin->argsize;
                            r = AT.WorkPointer;
                            while ( --i >= 0 ) *m++ = *r++;
                            n = AT.Nest;
                            while ( n <= AT.NestPoin ) {
                                if ( *(n->argsize) > 0 && n != AT.NestPoin )
                                    *(n->argsize) += j;
                                *(n->funsize) += j;
                                *(n->termsize) += j;
                                n++;
                            }
                            AN.EndNest += j;
/*                          (AT.NestPoin->argsize)[1] = 0;  */
                            if ( funnum == DENOMINATOR || funnum == EXPONENT ) {
                                if ( Normalize(BHEAD term) ) {
/*
                                    In this case something has been substituted
                                    Either a $ or a replace_?????
                                    Originally we had here:

                                    goto EndTest;

                                    It seems better to restart.
*/
                                    AN.ncmod = oldncmod;
                                    goto ReStart;
                                }
/*
                                And size changes here?????
*/
                            }
                            AN.ncmod = oldncmod;
                            goto ReStart;
                        }
                        else if ( *t == -DOLLAREXPRESSION ) {
                            if ( *t1 == TERMSINEXPR && t1[1] == FUNHEAD+2 ) {}
                            else {
                                if ( AR.Eside != LHSIDE ) {
```

```
                                AN.TeInFun = 1; AR.TePos = 0;
                                AT.TMbuff = AM.dbufnum; t1[2] |= DIRTYFLAG;
                                AN.ncmod = oldncmod;
                                return(1);
                            }
                            AC.lhdollarflag = 1;
                        }
                    }
                    else if ( *t == -TERMSINBRACKET ) {
                        if ( AR.Eside != LHSIDE ) {
                            AN.TeInFun = 1; AR.TePos = 0;
                            t1[2] |= DIRTYFLAG;
                            AN.ncmod = oldncmod;
                            return(1);
                        }
                    }
                    else if ( AN.ncmod != 0 && *t == -SNUMBER ) {
                        if ( AN.ncmod == 1 || AN.ncmod == -1 ) {
                            isp = (UWORD)(AC.cmod[0]);
                            isp = t[1] % isp;
                            if ( ( AC.modmode & POSNEG ) != 0 ) {
                                if ( isp > (UWORD)(AC.cmod[0])/2 ) isp = isp - (UWORD)(AC.cmod[0]);
                                else if ( -isp > (UWORD)(AC.cmod[0])/2 ) isp = isp +
    (UWORD)(AC.cmod[0]);
                            }
                            else {
                                if ( isp < 0 ) isp += (UWORD)(AC.cmod[0]);
                            }
                            if ( isp <= MAXPOSITIVE && isp >= -MAXPOSITIVE ) {
                                t[1] = isp;
                            }
                        }
                    }
                    NEXTARG(t)
                }
                if ( funnum >= FUNCTION && functions[funnum-FUNCTION].tabl ) {
/*
                    Test whether the table catches
                    Test 1: index arguments and range. i will be the number
                        of the element in the table.
*/
                    WORD rhsnumber, *oldwork = AT.WorkPointer, *Tpattern;
                    WORD ii, *p;
                    MINMAX *mm;
                    T = functions[funnum-FUNCTION].tabl;
/*
                    The next application of T->pattern isn't thread safe.
                    p = T->pattern + FUNHEAD+1;
                    The new code is in the next three lines and in the application
                    ii = T->pattern[1]; p = Tpattern; pp = T->pattern;
                    for ( i = 0; i < ii; i++ ) *p++ = *pp++;
                    AT.WorkPointer = p;
*/
#ifdef WITHPTHREADS
                    Tpattern = T->pattern[AT.identity];
#else
                    Tpattern = T->pattern;
#endif
                    p = Tpattern + FUNHEAD+1;

                    mm = T->mm;
                    if ( T->sparse ) {
                        t = t1+FUNHEAD;
                        if ( T->numind == 0 ) { isp = 0; }
                        else {
                          for ( i = 0; i < T->numind; i++, t += 2 ) {
                            if ( *t != -SNUMBER ) break;
                          }
                          if ( i < T->numind ) goto teststrict;

                          isp = FindTableTree(T,t1+FUNHEAD,2);
                        }
                        if ( isp < 0 ) {
teststrict:                 if ( T->strict == -2 ) {
                                rhsnumber = AM.zerorhs;
                                tbufnum = AM.zbufnum;
                            }
                            else if ( T->strict == -3 ) {
                                rhsnumber = AM.onerhs;
                                tbufnum = AM.zbufnum;
                            }
                            else if ( T->strict < 0 ) goto NextFun;
                            else {
                                MLOCK(ErrorMessageLock);
                                MesPrint("Element in table is undefined");
                                goto showtable;
                            }
```

```
/*
                            Copy the indices;
*/
                            t = t1+FUNHEAD+1;
                            for ( i = 0; i < T->numind; i++ ) {
                                *p = *t; p+=2; t+=2;
                            }
                        }
                        else {
                            rhsnumber = T->tablepointers[isp+T->numind];
#if ( TABLEEXTENSION == 2 )
                            tbufnum = T->bufnum;
#else
                            tbufnum = T->tablepointers[isp+T->numind+1];
#endif
                            t = t1+FUNHEAD+1;
                            ii = T->numind;
                            while ( --ii >= 0 ) {
                                *p = *t; t += 2; p += 2;
                            }
                        }
                        goto caughttable;
                    }
                    else {
                        i = 0;
                        t = t1 + FUNHEAD;
                        j = T->numind;
                        while ( --j >= 0 ) {
                            if ( *t != -SNUMBER ) goto NextFun;
                            t++;
                            if ( *t < mm->mini || *t > mm->maxi ) {
                                if ( T->bounds ) {
                                    MLOCK(ErrorMessageLock);
                                    MesPrint("Table boundary check. Argument %d",
                                    T->numind-j);
showtable:                          AO.OutFill = AO.OutputLine = (UBYTE *)m;
                                    AO.OutSkip = 8;
                                    IniLine(0);
                                    WriteSubTerm(t1,1);
                                    FiniLine();
                                    MUNLOCK(ErrorMessageLock);
                                    SETERROR(-1)
                                }
                                goto NextFun;
                            }
                            i += ( *t - mm->mini ) * (LONG)(mm->size);
                            *p = *t++;
                            p += 2;
                            mm++;
                        }
/*
                        Test now whether the entry exists.
*/
                        i *= TABLEEXTENSION;
                        if ( T->tablepointers[i] == -1 ) {
                            if ( T->strict == -2 ) {
                                rhsnumber = AM.zerorhs;
                                tbufnum = AM.zbufnum;
                            }
                            else if ( T->strict == -3 ) {
                                rhsnumber = AM.onerhs;
                                tbufnum = AM.zbufnum;
                            }
                            else if ( T->strict < 0 ) goto NextFun;
                            else {
                                MLOCK(ErrorMessageLock);
                                MesPrint("Element in table is undefined");
                                goto showtable;
                            }
                        }
                        else {
                            rhsnumber = T->tablepointers[i];
#if ( TABLEEXTENSION == 2 )
                            tbufnum = T->bufnum;
#else
                            tbufnum = T->tablepointers[i+1];
#endif
                        }
                    }
/*
                    If there are more arguments we have to do some
                    pattern matching. This should be easy. We addapted the
                    pattern, so that the array indices match already.
                    Note that if there is no match the program will become
                    very slow.
*/
caughttable:
```

```
#ifdef WITHPTHREADS
                    AN.FullProto = T->prototype[AT.identity];
#else
                    AN.FullProto = T->prototype;
#endif
                    AN.WildValue = AN.FullProto + SUBEXPSIZE;
                    AN.WildStop = AN.FullProto+AN.FullProto[1];
                    ClearWild(BHEAD0);
                    AN.RepFunNum = 0;
                    AN.RepFunList = AN.EndNest;
                    AT.WorkPointer = (WORD *)(((UBYTE *)(AN.EndNest)) + AM.MaxTer/2);
                    if ( AT.WorkPointer >= AT.WorkTop ) {
                        MLOCK(ErrorMessageLock);
                        MesWork();
                        MUNLOCK(ErrorMessageLock);
                    }
                    wilds = 0;
/*                  if ( MatchFunction(BHEAD T->pattern,t1,&wilds) > 0 ) {  } */
                    if ( MatchFunction(BHEAD Tpattern,t1,&wilds) > 0 ) {
                        AT.WorkPointer = oldwork;
                        if ( AT.NestPoin != AT.Nest ) {
                            AN.ncmod = oldncmod;
                            return(1);
                        }

                        m = AN.FullProto;
                        retvalue = m[2] = rhsnumber;
                        m[4] = tbufnum;
                        t = t1;
                        j = t[1];
                        i = m[1];
                        if ( j > i ) {
                            j = i - j;
                            NCOPY(t,m,i);
                            m = term + *term;
                            while ( r < m ) *t++ = *r++;
                            *term += j;
                        }
                        else if ( j < i ) {
                            j = i-j;
                            t = term + *term;
                            while ( t >= r ) { t[j] = *t; t--; }
                            t = t1;
                            NCOPY(t,m,i);
                            *term += j;
                        }
                        else {
                            NCOPY(t,m,j);
                        }
                        AN.TeInFun = 0;
                        AR.TePos = 0;
                        AN.TeSuOut = -1;
                        if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
                        AT.TMbuff = tbufnum;
                        AN.ncmod = oldncmod;
                        return(retvalue);
                    }
                    AT.WorkPointer = oldwork;
                }
NextFun:;
            }
            else if ( ( t[2] & DIRTYFLAG ) != 0 ) {
                t += FUNHEAD;
                while ( t < r ) {
                    if ( *t == FUNNYDOLLAR ) {
                        if ( AR.Eside != LHSIDE ) {
                            AN.TeInFun = 1;
                            AR.TePos = 0;
                            AT.TMbuff = AM.dbufnum;
                            AN.ncmod = oldncmod;
                            return(1);
                        }
                        AC.lhdollarflag = 1;
                    }
                    t++;
                }
            }
            t = r;
            AN.ncmod = oldncmod;
        } while ( t < m );
    }
    return(0);
EndTest:;
    MLOCK(ErrorMessageLock);
EndTest2:;
    MesCall("TestSub");
    MUNLOCK(ErrorMessageLock);
```

```
    SETERROR(-1)
}

/*
        #] TestSub :
        #[ InFunction :         WORD InFunction(term,termout)
*/
WORD InFunction(PHEAD WORD *term, WORD *termout)
{
    GETBIDENTITY
    WORD *m, *t, *r, *rr, sign = 1, oldncmod;
    WORD *u, *v, *w, *from, *to,
        ipp, olddefer = AR.DeferFlag, oldPolyFun = AR.PolyFun, i, j;
    LONG numterms;
    from = t = term;
    r = t + *t - 1;
    m = r - ABS(*r) + 1;
    t++;
    while ( t < m ) {
        if ( *t >= FUNCTION+WILDOFFSET ) ipp = *t - WILDOFFSET;
        else ipp = *t;
        if ( AR.TePos ) {
            if ( ( term + AR.TePos ) == t ) {
                m = termout;
                while ( from < t ) *m++ = *from++;
                *m++ = DENOMINATOR;
                *m++ = t[1] + 4 + FUNHEAD + ARGHEAD;
                *m++ = DIRTYFLAG;
                FILLFUN3(m)
                *m++ = t[1] + 4 + ARGHEAD;
                *m++ = 1;
                FILLARG(m)
                *m++ = t[1] + 4;
                t[3] = -t[3];
                v = t + t[1];
                while ( t < v ) *m++ = *t++;
                from[3] = -from[3];
                *m++ = 1;
                *m++ = 1;
                *m++ = 3;
                r = term + *term;
                while ( t < r ) *m++ = *t++;
                if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) goto TooLarge;
                *termout = WORDDIF(m,termout);
                return(0);
            }
        }
        else if ( ( *t >= FUNCTION && functions[ipp-FUNCTION].spec == 0 )
        && ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) {
            m = termout;
            r = t + t[1];
            u = t;
            t += FUNHEAD;
            oldncmod = AN.ncmod;
            while ( t < r ) {   /* t points at an argument */
                if ( *t > 0 && t[1] ) { /* Argument has been modified */
                    WORD oldsorttype = AR.SortType;
                    /* This whole argument must be redone */

                    if ( ( AN.ncmod != 0 )
                        && ( ( AC.modmode & ALSOFUNARGS ) == 0 )
                        && ( *u != AR.PolyFun ) ) { AN.ncmod = 0; }
                    AR.DeferFlag = 0;
                    v = t + *t;
                    t += ARGHEAD;       /* First term */
                    w = 0;  /* to appease the compilers warning devices */
                    while ( from < t ) {
                        if ( from == u ) w = m;
                        *m++ = *from++;
                    }
                    to = m;
                    NewSort(BHEAD0);
                    if ( *u == AR.PolyFun && AR.PolyFunType == 2 ) {
                        AR.CompareRoutine = &CompareSymbols;
                        AR.SortType = SORTHIGHFIRST;
                    }
/*
                    AR.PolyFun = 0;
*/
                    while ( t < v ) {
                        i = *t;
                        NCOPY(m,t,i);
                        m = to;
                        if ( AT.WorkPointer < m+*m ) AT.WorkPointer = m + *m;
                        if ( Generator(BHEAD m,AR.Cnumlhs) ) {
                            AN.ncmod = oldncmod;
                            LowerSortLevel(); goto InFunc;
```

```
                        }
                    }
                    /* w = the function */
                    /* v = the next argument */
                    /* u = the function */
                    /* to is new argument */

                    to -= ARGHEAD;
                    if ( EndSort(BHEAD m,1) < 0 ) {
                        AN.ncmod = oldncmod;
                        goto InFunc;
                    }
                    AR.PolyFun = oldPolyFun;
                    if ( *u == AR.PolyFun && AR.PolyFunType == 2 ) {
                        AR.CompareRoutine = &Compare1;
                        AR.SortType = oldsorttype;
                    }
                    while ( *m ) m += *m;
                    *to = WORDDIF(m,to);
                    to[1] = 1;  /* ??????? or rather 0?. 24-mar-2006 JV */
                    if ( ToFast(to,to) ) {
                        if ( *to <= -FUNCTION ) m = to+1;
                        else m = to+2;
                    }
                    w[1] = WORDDIF(m,w) + WORDDIF(r,v);
                    r = term + *term;
                    t = v;
                    while ( t < r ) *m++ = *t++;
                    if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) goto TooLarge;
                    *termout = WORDDIF(m,termout);
                    AR.DeferFlag = olddefer;
                    AN.ncmod = oldncmod;
                    return(0);
                }
                else if ( *t == -DOLLAREXPRESSION ) {
                    if ( AR.Eside == LHSIDE ) {
                        NEXTARG(t)
                        AC.lhdollarflag = 1;
                    }
                    else {
/*
                        This whole argument must be redone
*/
                    DOLLARS d = Dollars + t[1];
#ifdef WITHPTHREADS
                    int nummodopt, dtype = -1;
                    if ( AS.MultiThreaded ) {
                        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                            if ( t[1] == ModOptdollars[nummodopt].number ) break;
                        }
                        if ( nummodopt < NumModOptdollars ) {
                            dtype = ModOptdollars[nummodopt].type;
                            if ( dtype == MODLOCAL ) {
                                d = ModOptdollars[nummodopt].dstruct+AT.identity;
                            }
                            else {
                                LOCK(d->pthreadslockread);
                            }
                        }
                    }
#endif
                    oldncmod = AN.ncmod;
                    if ( ( AN.ncmod != 0 )
                        && ( ( AC.modmode & ALSOFUNARGS ) == 0 )
                        && ( *u != AR.PolyFun ) ) { AN.ncmod = 0; }
                    AR.DeferFlag = 0;
                    v = t + 2;
                    w = 0;  /* to appease the compilers warning devices */
                    while ( from < t ) {
                        if ( from == u ) w = m;
                        *m++ = *from++;
                    }
                    to = m;
                    switch ( d->type ) {
                        case DOLINDEX:
                            if ( d->index >= 0 && d->index < AM.OffsetIndex ) {
                                *m++ = -SNUMBER; *m++ = d->index;
                            }
                            else { *m++ = -INDEX; *m++ = d->index; }
                            break;
                        case DOLZERO:
                            *m++ = -SNUMBER; *m++ = 0; break;
                        case DOLNUMBER:
                            if ( d->where[0] == 4 &&
                            ( d->where[1] & MAXPOSITIVE ) == d->where[1] ) {
                                *m++ = -SNUMBER;
                                if ( d->where[3] >= 0 ) *m++ = d->where[1];
```

```
                                else *m++ = -d->where[1];
                                break;
                            }
                        case DOLTERMS:
/*
                            Here we have the special case of the PolyRatFun
                            That function may have a different sort of the
                            terms in the argument.
*/
                            to = m; r = d->where;
                            *m++ = 0; *m++ = 1;
                            FILLARG(m)
                            while ( *r ) {
                                i = *r; NCOPY(m,r,i)
                            }
                            *to = m-to;
                            if ( ToFast(to,to) ) {
                                if ( *to <= -FUNCTION ) m = to+1;
                                else m = to+2;
                            }
                            else if ( *u == AR.PolyFun && AR.PolyFunType == 2 ) {
                                AR.PolyFun = 0;
                                NewSort(BHEAD0);
                                AR.CompareRoutine = &CompareSymbols;
                                r = to + ARGHEAD;
                                while ( r < m ) {
                                    rr = r; r += *r;
                                    if ( SymbolNormalize(rr) ) goto InFunc;
                                    if ( StoreTerm(BHEAD rr) ) {
                                        AR.CompareRoutine = &Compare1;
                                        LowerSortLevel();
                                        Terminate(-1);
                                    }
                                }
                                if ( EndSort(BHEAD to+ARGHEAD,1) < 0 ) goto InFunc;
                                AR.PolyFun = oldPolyFun;
                                AR.CompareRoutine = &Compare1;
                                m = to+ARGHEAD;
                                if ( *m == 0 ) {
                                    *to = -SNUMBER;
                                    to[1] = 0;
                                    m = to + 2;
                                }
                                else {
                                    while ( *m ) m += *m;
                                    *t = m - to;
                                    if ( ToFast(to,to) ) {
                                        if ( *to <= -FUNCTION ) m = to+1;
                                        else m = to+2;
                                    }
                                }
                            }
                            w[1] = w[1] - 2 + (m-to);
                            break;
                        case DOLSUBTERM:
                            to = m; r = d->where;
                            i = r[1];
                            *m++ = i+4+ARGHEAD; *m++ = 1;
                            FILLARG(m)
                            *m++ = i+4;
                            while ( --i >= 0 ) *m++ = *r++;
                            *m++ = 1; *m++ = 1; *m++ = 3;
                            if ( ToFast(to,to) ) {
                                if ( *to <= -FUNCTION ) m = to+1;
                                else m = to+2;
                            }
                            w[1] = w[1] - 2 + (m-to);
                            break;
                        case DOLARGUMENT:
                            to = m; r = d->where;
                            if ( *r > 0 ) {
                                i = *r - 2;
                                *m++ = *r++; *m++ = 1; r++;
                                while ( --i >= 0 ) *m++ = *r++;
                            }
                            else if ( *r <= -FUNCTION ) *m++ = *r++;
                            else { *m++ = *r++; *m++ = *r++; }
                            w[1] = w[1] - 2 + (m-to);
                            break;
                        case DOLWILDARGS:
                            to = m; r = d->where;
                            if ( *r > 0 ) { /* Tensor arguments */
                                i = *r++;
                                while ( --i >= 0 ) {
                                    if ( *r < 0 ) {
                                        *m++ = -VECTOR; *m++ = *r++;
                                    }
```

```
                                    else if ( *r >= AM.OffsetIndex ) {
                                        *m++ = -INDEX; *m++ = *r++;
                                    }
                                    else { *m++ = -SNUMBER; *m++ = *r++; }
                                }
                            }
                            else { /* Regular arguments */
                                r++;
                                while ( *r ) {
                                    if ( *r > 0 ) {
                                        i = *r - 2;
                                        *m++ = *r++; *m++ = 1; r++;
                                        while ( --i >= 0 ) *m++ = *r++;
                                    }
                                    else if ( *r <= -FUNCTION ) *m++ = *r++;
                                    else { *m++ = *r++; *m++ = *r++; }
                                }
                            }
                            w[1] = w[1] - 2 + (m-to);
                            break;
                        case DOLUNDEFINED:
                        default:
                            MLOCK(ErrorMessageLock);
                            MesPrint("!!!Undefined $-variable: $%s!!!",
                            AC.dollarnames->namebuffer+d->name);
                            MUNLOCK(ErrorMessageLock);
#ifdef WITHPTHREADS
                            if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                            Terminate(-1);
                        }
#ifdef WITHPTHREADS
                    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                    r = term + *term;
                    t = v;
                    while ( t < r ) *m++ = *t++;
                    if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) goto TooLarge;
                    *termout = WORDDIF(m,termout);
                    AR.DeferFlag = olddefer;
                    AN.ncmod = oldncmod;
                    return(0);
                }
                }
                else if ( *t == -TERMSINBRACKET ) {
                    if ( AC.ComDefer ) numterms = CountTerms1(BHEAD0);
                    else               numterms = 1;
/*
                    Compose the output term
                    First copy the part till this function argument
                    m points at the output term space
                    u points at the start of the function
                    t points at the start of the argument
*/
                    w = 0;
                    while ( from < t ) {
                        if ( from == u ) w = m;
                        *m++ = *from++;
                    }
                    if ( ( numterms & MAXPOSITIVE ) == numterms ) {
                        *m++ = -SNUMBER; *m++ =  numterms & MAXPOSITIVE;
                        w[1] += 1;
                    }
                    else if ( ( i = numterms » BITSINWORD ) == 0 ) {
                        *m++ = ARGHEAD+4;
                        for ( j = 1; j < ARGHEAD; j++ ) *m++ = 0;
                        *m++ = 4; *m++ = numterms & WORDMASK; *m++ = 1; *m++ = 3;
                        w[1] += ARGHEAD+3;
                    }
                    else {
                        *m++ = ARGHEAD+6;
                        for ( j = 1; j < ARGHEAD; j++ ) *m++ = 0;
                        *m++ = 6; *m++ = numterms & WORDMASK;
                        *m++ = i; *m++ = 1; *m++ = 0; *m++ = 5;
                        w[1] += ARGHEAD+5;
                    }
                    from++;  /* Skip our function */
                    r = term + *term;
                    while ( from < r ) *m++ = *from++;
                    if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) goto TooLarge;
                    *termout = WORDDIF(m,termout);
                    return(0);
                }
                else { NEXTARG(t) }
            }
            t = u;
        }
```

```
        else if ( ( *t >= FUNCTION && functions[ipp-FUNCTION].spec )
        && ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) { /* Could be FUNNYDOLLAR */
            u = t; v = t + t[1];
            t += FUNHEAD;
            while ( t < v ) {
                if ( *t == FUNNYDOLLAR ) {
                    if ( AR.Eside != LHSIDE ) {
                    DOLLARS d = Dollars + t[1];
#ifdef WITHPTHREADS
                    int nummodopt, dtype = -1;
                    if ( AS.MultiThreaded ) {
                        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                            if ( t[1] == ModOptdollars[nummodopt].number ) break;
                        }
                        if ( nummodopt < NumModOptdollars ) {
                            dtype = ModOptdollars[nummodopt].type;
                            if ( dtype == MODLOCAL ) {
                                d = ModOptdollars[nummodopt].dstruct+AT.identity;
                            }
                            else {
                                LOCK(d->pthreadslockread);
                            }
                        }
                    }
#endif
                    oldncmod = AN.ncmod;
                    if ( ( AN.ncmod != 0 )
                        && ( ( AC.modmode & ALSOFUNARGS ) == 0 )
                        && ( *u != AR.PolyFun ) ) { AN.ncmod = 0; }
                    m = termout; w = 0;
                    while ( from < t ) {
                        if ( from == u ) w = m;
                        *m++ = *from++;
                    }
                    to = m;
                    switch ( d->type ) {
                        case DOLINDEX:
                            *m++ = d->index; break;
                        case DOLZERO:
                            *m++ = 0; break;
                        case DOLNUMBER:
                        case DOLTERMS:
                            if ( d->where[0] == 4 && d->where[4] == 0
                            && d->where[3] == 3 && d->where[2] == 1
                            && d->where[1] < AM.OffsetIndex ) {
                                *m++ = d->where[1];
                            }
                            else {
wrongtype:;
#ifdef WITHPTHREADS
                                if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                MLOCK(ErrorMessageLock);
                                MesPrint("$%s has wrong type for tensor substitution",
                                AC.dollarnames->namebuffer+d->name);
                                MUNLOCK(ErrorMessageLock);
                                AN.ncmod = oldncmod;
                                return(-1);
                            }
                            break;
                        case DOLARGUMENT:
                            if ( d->where[0] == -INDEX ) {
                                *m++ = d->where[1]; break;
                            }
                            else if ( d->where[0] == -VECTOR ) {
                                *m++ = d->where[1]; break;
                            }
                            else if ( d->where[0] == -MINVECTOR ) {
                                *m++ = d->where[1];
                                sign = -sign;
                                break;
                            }
                            else if ( d->where[0] == -SNUMBER ) {
                                if ( d->where[1] >= 0
                                && d->where[1] < AM.OffsetIndex ) {
                                    *m++ = d->where[1]; break;
                                }
                            }
                            goto wrongtype;
                        case DOLWILDARGS:
                            if ( d->where[0] > 0 ) {
                                r = d->where; i = *r++;
                                while ( --i >= 0 ) *m++ = *r++;
                            }
                            else {
                                r = d->where + 1;
                                while ( *r ) {
```

```
                                    if ( *r == -INDEX ) {
                                        *m++ = r[1]; r += 2; continue;
                                    }
                                    else if ( *r == -VECTOR ) {
                                        *m++ = r[1]; r += 2; continue;
                                    }
                                    else if ( *r == -MINVECTOR ) {
                                        *m++ = r[1]; r += 2;
                                        sign = -sign; continue;
                                    }
                                    else if ( *r == -SNUMBER ) {
                                        if ( r[1] >= 0
                                        && r[1] < AM.OffsetIndex ) {
                                            *m++ = r[1]; r += 2; continue;
                                        }
                                    }
                                    goto wrongtype;
                                }
                            }
                            break;
                        case DOLSUBTERM:
                            r = d->where;
                            if ( *r == INDEX && r[1] == 3 ) {
                                *m++ = r[2];
                            }
                            else goto wrongtype;
                            break;
                        case DOLUNDEFINED:
#ifdef WITHPTHREADS
                            if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                            MLOCK(ErrorMessageLock);
                            MesPrint("$%s is undefined in tensor substitution",
                            AC.dollarnames->namebuffer+d->name);
                            MUNLOCK(ErrorMessageLock);
                            AN.ncmod = oldncmod;
                            return(-1);
                    }
#ifdef WITHPTHREADS
                    if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                    w[1] = w[1] - 2 + (m-to);
                    from += 2;
                    term += *term;
                    while ( from < term ) *m++ = *from++;
                    if ( sign < 0 ) m[-1] = -m[-1];
                    if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) goto TooLarge;
                    *termout = m - termout;
                    AN.ncmod = oldncmod;
                    return(0);
                }
                else {
                    AC.lhdollarflag = 1;
                }
                }
                t++;
            }
            t = u;
        }
        t += t[1];
    }
    MLOCK(ErrorMessageLock);
    MesPrint("Internal error in InFunction: Function not encountered.");
    if ( AM.tracebackflag ) {
        MesPrint("%w: AR.TePos = %d",AR.TePos);
        MesPrint("%w: AN.TeInFun = %d",AN.TeInFun);
        termout = term;
        AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer + AM.MaxTer;
        AO.OutSkip = 3;
        FiniLine();
        i = *termout;
        while ( --i >= 0 ) {
            TalToLine((UWORD)(*termout++));
            TokenToLine((UBYTE *)"  ");
        }
        AO.OutSkip = 0;
        FiniLine();
        MesCall("InFunction");
    }
    MUNLOCK(ErrorMessageLock);
    return(1);

InFunc:
    MLOCK(ErrorMessageLock);
    MesCall("InFunction");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
```

```c

TooLarge:
    MLOCK(ErrorMessageLock);
    MesPrint("Output term too large. Try to increase MaxTermSize in the setup.");
    MesCall("InFunction");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] InFunction :
        #[ InsertTerm :         WORD InsertTerm(term,replac,extractbuff,position,termout)
*/
WORD InsertTerm(PHEAD WORD *term, WORD replac, WORD extractbuff, WORD *position, WORD *termout,
                WORD tepos)
{
    GETBIDENTITY
    WORD *m, *t, *r, i, l2, j;
    WORD *u, *v, l1, *coef;
    coef = AT.WorkPointer;
    if ( ( AT.WorkPointer = coef + 2*AM.MaxTal ) > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    t = term;
    r = t + *t;
    l1 = l2 = r[-1];
    m = r - ABS(l2);
    if ( tepos > 0 ) {
        t = term + tepos;
        goto foundit;
    }
    t++;
    while ( t < m ) {
        if ( *t == SUBEXPRESSION && t[2] == replac && t[3] && t[4] == extractbuff ) {
            r = t + t[1];
            while ( *r == SUBEXPRESSION && r[2] == replac && r[3] && r < m && r[4] == extractbuff ) {
                t = r; r += r[1];
            }
foundit:;
            u = m;
            r = term;
            m = termout;
            do { *m++ = *r++; } while ( r < t );
            if ( t[1] > SUBEXPSIZE ) {
/*
                if this is a dollar expression there are no wildcards
*/
                i = *--m;
                if ( ( l2 = WildFill(BHEAD m,position,t) ) < 0 ) goto InsCall;
                *m = i;
                m += l2-1;
                l2 = *m;
                i = ( j = ABS(l2) ) - 1;
                r = coef + i;
                do { *--r = *--m; } while ( --i > 0 );
            }
            else {
                v = t;
                t = position;
                r = t + *t;
                l2 = r[-1];
                r -= ( j = ABS(l2) );
                t++;
                if ( t < r ) do { *m++ = *t++; } while ( t < r );
                t = v;
            }
            t += t[1];
            while ( t < u && *t == DOLLAREXPR2 ) t += t[1];
ComAct:     if ( t < u ) do { *m++ = *t++; } while ( t < u );
            if ( *r == 1 && r[1] == 1 && j == 3 ) {
                if ( l2 < 0 ) l1 = -l1;
                i = ABS(l1)-1;
                NCOPY(m,t,i);
                *m++ = l1;
            }
            else {
                if ( MulRat(BHEAD (UWORD *)u,REDLENG(l1),(UWORD *)r,REDLENG(l2),
                (UWORD *)m,&l1) ) goto InsCall;
                l2 = l1;
                l2 «= 1;
                if ( l2 < 0 ) {
                    m -= l2;
                    *m++ = l2-1;
                }
```

```
                else {
                    m += l2;
                    *m++ = l2+1;
                }
            }
            *termout = WORDDIF(m,termout);
            if ( (*termout)*((LONG)sizeof(WORD)) > AM.MaxTer ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Term too complex during substitution. MaxTermSize of %l is too small",AM.MaxTer);
                goto InsCall2;
            }
            AT.WorkPointer = coef;
            return(0);
        }
        t += t[1];
    }
/*
    The next action is for when there is no subexpression pointer.
    We append the extra term. Effectively the routine becomes now a
    merge routine for two terms.
*/
    v = t;
    u = m;
    r = term;
    m = termout;
    do { *m++ = *r++; } while ( r < t );
    t = position;
    r = t + *t;
    l2 = r[-1];
    r -= ( j = ABS(l2) );
    t++;
    if ( t < r ) do { *m++ = *t++; } while ( t < r );
    t = v;
    goto ComAct;

InsCall:
    MLOCK(ErrorMessageLock);
InsCall2:
    MesCall("InsertTerm");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] InsertTerm :
        #[ PasteFile :          WORD PasteFile(num,acc,pos,accf,renum,freeze,nexpr)
*/
LONG PasteFile(PHEAD WORD number, WORD *accum, POSITION *position, WORD **accfill,
               RENUMBER renumber, WORD *freeze, WORD nexpr)
{
    GETBIDENTITY
    WORD *r, l, *m, i;
    WORD *stop, *s1, *s2;
/*  POSITION AccPos; bug 12-apr-2008 JV */
    WORD InCompState;
    WORD *oldipointer;
    LONG retlength;
    stop = (WORD *)(((UBYTE *)(accum)) + 2*AM.MaxTer);
    *accum++ = number;
    while ( --number >= 0 ) accum += *accum;
    if ( freeze ) {
/*      AccPos = *position; bug 12-apr-2008 JV */
        oldipointer = AR.CompressPointer;
        do {
            AR.CompressPointer = oldipointer;
/*          if ( ( l = GetFromStore(accum,&AccPos,renumber,&InCompState,nexpr) ) < 0 ) bug 12-apr-2008 JV */
            if ( ( l = GetFromStore(accum,position,renumber,&InCompState,nexpr) ) < 0 )
                goto PasErr;
            if ( !l ) { *accum = 0; return(0); }
            r = accum;
            m = r + *r;
            m -= ABS(m[-1]);
            r++;
            while ( r < m && *r != HAAKJE ) r += r[1];
            if ( r >= m ) {
                if ( *freeze != 4 ) l = -1;
            }
            else {
/*
                The algorithm for accepting terms with a given (freeze)
                representation outside brackets is rather crude. A refinement
                would be to store the part outside the bracket and skip the
                term when this part doesn't alter (and is unacceptable).
                Once accepting one can keep accepting till the bracket alters
                and then one may stop the generation. It is necessary to
```

```
                set up a struct to remember the bracket and the progress
                status.
*/
                m = AT.WorkPointer;
                s2 = r;
                r = accum;
                *m++ = WORDDIF(s2,r) + 3;
                r++;
                while ( r < s2 ) *m++ = *r++;
                *m++ = 1; *m++ = 1; *m++ = 3;
                m = AT.WorkPointer;
                if ( Normalize(BHEAD AT.WorkPointer) ) goto PasErr;
                r = freeze;
                i = *m;
                while ( --i >= 0 && *m++ == *r++ ) {}
                if ( i > 0 ) {
                    l = -1;
                }
                else {  /* Term to be accepted */
                    r = accum;
                    s1 = r + *r;
                    r++;
                    m = s2;
                    m += m[1];
                    do { *r++ = *m++; } while ( m < s1 );
                    *accum = l = WORDDIF(r,accum);
                }
            }
        } while ( l < 0 );
        retlength = InCompState;
/*      retlength = DIFBASE(AccPos,*position) / sizeof(WORD);  bug 12-apr-2008 JV */
    }
    else {
        if ( ( l = GetFromStore(accum,position,renumber,&InCompState,nexpr) ) < 0 ) {
            MLOCK(ErrorMessageLock);
            MesCall("PasteFile");
            MUNLOCK(ErrorMessageLock);
            SETERROR(-1)
        }
        if ( l == 0 ) { *accum = 0; return(0); }
        retlength = InCompState;
    }
    accum += l;
    if ( accum > stop ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Buffer too small in PasteFile");
        MUNLOCK(ErrorMessageLock);
        SETERROR(-1)
    }
    *accum = 0;
    *accfill = accum;
    return(retlength);
PasErr:
    MLOCK(ErrorMessageLock);
    MesCall("PasteFile");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] PasteFile :
        #[ PasteTerm :          WORD PasteTerm(number,accum,position,times,divby)
*/
WORD *PasteTerm(PHEAD WORD number, WORD *accum, WORD *position, WORD times, WORD divby)
{
    GETBIDENTITY
    WORD *t, *r, x, y, z;
    WORD *m, *u, l1, a[2];
    m = (WORD *)(((UBYTE *)(accum)) + AM.MaxTer);
/*    m = (WORD *)(((UBYTE *)(accum)) + 2*AM.MaxTer); */
    *accum++ = number;
    while ( --number >= 0 ) accum += *accum;
    if ( times == divby ) {
        t = position;
        r = t + *t;
        if ( t < r ) do { *accum++ = *t++; } while ( t < r );
    }
    else {
        u = accum;
        t = position;
        r = t + *t - 1;
        l1 = *r;
        r -= ABS(*r) - 1;
        if ( t < r ) do { *accum++ = *t++; } while ( t < r );
        if ( divby > times ) { x = divby; y = times; }
        else { x = times; y = divby; }
        z = x%y;
```

```
        while ( z ) { x = y; y = z; z = x%y; }
        if ( y != 1 ) { divby /= y; times /= y; }
        a[1] = divby;
        a[0] = times;
        if ( MulRat(BHEAD (UWORD *)t,REDLENG(l1),(UWORD *)a,1,(UWORD *)accum,&l1) ) {
            MLOCK(ErrorMessageLock);
            MesCall("PasteTerm");
            MUNLOCK(ErrorMessageLock);
            return(0);
        }
        x = l1;
        x «= 1;
        if ( x < 0 ) { accum -= x; *accum++ = x - 1; }
        else         { accum += x; *accum++ = x + 1; }
        *u = WORDDIF(accum,u);
    }
    if ( accum >= m ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Buffer too small in PasteTerm");
        MUNLOCK(ErrorMessageLock);
        return(0);
    }
    *accum = 0;
    return(accum);
}

/*
        #] PasteTerm :
        #[ FiniTerm :           WORD FiniTerm(term,accum,termout,number)
*/
WORD FiniTerm(PHEAD WORD *term, WORD *accum, WORD *termout, WORD number, WORD tepos)
{
    GETBIDENTITY
    WORD *m, *t, *r, i, numacc, l2, ipp;
    WORD *u, *v, l1, *coef = AT.WorkPointer, *oldaccum;
    if ( ( AT.WorkPointer = coef + 2*AM.MaxTal ) > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    oldaccum = accum;
    t = term;
    m = t + *t - 1;
    l1 = REDLENG(*m);
    i = ABS(*m) - 1;
    r = coef + i;
    do { *--r = *--m; } while ( --i > 0 ); /* Copies coefficient */
    if ( tepos > 0 ) {
        t = term + tepos;
        goto foundit;
    }
    t++;
    if ( t < m ) do {
        if ( ( ( *t == SUBEXPRESSION && ( *(r=t+t[1]) != SUBEXPRESSION
        || r >= m || !r[3] ) ) || *t == EXPRESSION ) && t[2] == number && t[3] ) {
foundit:;
            u = m;
            r = term;
            m = termout;
            if ( r < t ) do { *m++ = *r++; } while ( r < t );
            numacc = *accum++;
            if ( numacc >= 0 ) do {
                if ( *t == EXPRESSION ) {
                    v = t + t[1];
                    r = t + SUBEXPSIZE;
                    while ( r < v ) {
                        if ( *r == WILDCARDS ) {
                            r += 2;
                            i = *--m;
                            if ( ( l2 = WildFill(BHEAD m,accum,r) ) < 0 ) goto FiniCall;
                            goto AllWild;
                        }
                        r += r[1];
                    }
                    goto NoWild;
                }
                else if ( t[1] > SUBEXPSIZE && t[SUBEXPSIZE] != FROMBRAC ) {
                    i = *--m;
                    if ( ( l2 = WildFill(BHEAD m,accum,t) ) < 0 ) goto FiniCall;
AllWild:            *m = i;
                    m += l2-1;
                    l2 = *m;
                    m -= ABS(l2) - 1;
                    r = m;
                }
                else {
```

```
NoWild:             r = accum;
                    v = r + *r - 1;
                    l2 = *v;
                    v -= ABS(l2) - 1;
                    r++;
                    if ( r < v ) do { *m++ = *r++; } while ( r < v );
                }
                if ( *r == 1 && r[1] == 1 && ABS(l2) == 3 ) {
                    if ( l2 < 0 ) l1 = -l1;
                }
                else {
                    l2 = REDLENG(l2);
                    if ( l2 == 0 ) {
                        t = oldaccum;
                        numacc = *t++;
                        AO.OutSkip = 3;
                        FiniLine();
                        while ( --numacc >= 0 ) {
                            i = *t;
                            while ( --i >= 0 ) {
                                TalToLine((UWORD)(*t++));
                                TokenToLine((UBYTE *)"  ");
                            }
                        }
                        AO.OutSkip = 0;
                        FiniLine();
                        goto FiniCall;
                    }
                    if ( MulRat(BHEAD (UWORD *)coef,l1,(UWORD *)r,l2,(UWORD *)coef,&l1) ) goto FiniCall;
                    if ( AN.ncmod != 0 && TakeModulus((UWORD *)coef,&l1,AC.cmod,AN.ncmod,UNPACK|AC.modmode) ) goto FiniCall;
                }
                accum += *accum;
            } while ( --numacc >= 0 );
            if ( *t == SUBEXPRESSION ) {
                 while ( t+t[1] < u && t[t[1]] == DOLLAREXPR2 ) t += t[1];
            }
            t += t[1];
            if ( t < u ) do { *m++ = *t++; } while ( t < u );
            l2 = l1;
/*
            Code to economize when taking x = (a+b)/2
*/
            r = termout+1;
            while ( r < m ) {
                if ( *r == SUBEXPRESSION ) {
                    t = r + r[1];
                    l1 = (WORD)(cbuf[r[4]].CanCommu[r[2]]);
                    while ( t < m ) {
                        if ( *t == SUBEXPRESSION &&
                            t[1] == r[1] && t[2] == r[2] && t[4] == r[4] ) {
                                i = t[1] - SUBEXPSIZE;
                                u = r + SUBEXPSIZE; v = t + SUBEXPSIZE;
                                while ( i > 0 ) {
                                     if ( *v++ != *u++ ) break; i--; }
                                if ( i <= 0 ) {
                                    u = r;
                                    r[3] += t[3];
                                    r = t + t[1];
                                    while ( r < m ) *t++ = *r++;
                                    m = t;
                                    r = u;
                                    goto Nextr;
                                }
                                if ( l1 && cbuf[t[4]].CanCommu[t[2]] ) break;
                                while ( t+t[1] < m && t[t[1]] == DOLLAREXPR2 ) t += t[1];
                        }
                        else if ( l1 ) {
                            if ( *t == SUBEXPRESSION && cbuf[t[4]].CanCommu[t[2]] )
                                break;
                            if ( *t >= FUNCTION+WILDOFFSET )
                                ipp = *t - WILDOFFSET;
                            else ipp = *t;
                            if ( *t >= FUNCTION
                             && functions[ipp-FUNCTION].commute && l1 ) break;
                            if ( *t == EXPRESSION ) break;
                        }
                        t += t[1];
                    }
                    r += r[1];
                }
                else r += r[1];
Nextr:;
            }

            i = ABS(l2);
```

```
            i <<= 1;
            i++;
            l2 = ( l2 >= 0 ) ? i: -i;
            r = coef;
            while ( --i > 0 ) *m++ = *r++;
            *m++ = l2;
            *termout = WORDDIF(m,termout);
            AT.WorkPointer = coef;
            return(0);
        }
        t += t[1];
    } while ( t < m );
    AT.WorkPointer = coef;
    return(1);

FiniCall:
    MLOCK(ErrorMessageLock);
    MesCall("FiniTerm");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] FiniTerm :
        #[ Generator :          WORD Generator(BHEAD term,level)
*/

static WORD zeroDollar[] = { 0, 0 };
/*
static LONG debugcounter = 0;
*/

WORD Generator(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    WORD replac, *accum, *termout, *t, i, j, tepos, applyflag = 0, *StartBuf;
    WORD *a, power, power1, DumNow = AR.CurDum, oldtoprhs, oldatoprhs, retnorm, extractbuff;
    int *RepSto = AN.RepPoint, iscopy = 0;
    CBUF *C = cbuf+AM.rbufnum, *CC = cbuf + AT.ebufnum, *CCC = cbuf + AT.aebufnum;
    LONG posisub, oldcpointer, oldacpointer;
    DOLLARS d = 0;
    WORD numfac[5], idfunctionflag;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1, id;
#endif
    oldtoprhs = CC->numrhs;
    oldcpointer = CC->Pointer - CC->Buffer;
    oldatoprhs = CCC->numrhs;
    oldacpointer = CCC->Pointer - CCC->Buffer;
ReStart:
    if ( ( replac = TestSub(BHEAD term,level) ) == 0 ) {
        if ( applyflag ) { TableReset(); applyflag = 0; }
/*
        if ( AN.PolyNormFlag > 1 ) {
            if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
            AN.PolyNormFlag = 0;
            if ( !*term ) goto Return0;
        }
*/
Renormalize:
        AN.PolyNormFlag = 0;
        AN.idfunctionflag = 0;
        if ( ( retnorm = Normalize(BHEAD term) ) != 0 ) {
            if ( retnorm > 0 ) {
                if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
                goto ReStart;
            }
            goto GenCall;
        }
        idfunctionflag = AN.idfunctionflag;
        if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }

        if ( AN.PolyNormFlag ) {
            if ( AN.PolyFunTodo == 0 ) {
                if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
                if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
            }
            else {
                WORD oldPolyFunExp = AR.PolyFunExp;
                AR.PolyFunExp = 0;
                if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
                AT.WorkPointer = term+*term;
                AR.PolyFunExp = oldPolyFunExp;
                if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
                if ( Normalize(BHEAD term) < 0 ) goto GenCall;
                if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
                AT.WorkPointer = term+*term;
```

```
                if ( AN.PolyNormFlag ) {
                    if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
                    if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
                    AT.WorkPointer = term+*term;
                }
                AN.PolyFunTodo = 0;
            }
        }
        if ( idfunctionflag > 0 ) {
            if ( TakeIDfunction(BHEAD term) ) {
                AT.WorkPointer = term + *term;
                goto ReStart;
            }
        }
        if ( AT.WorkPointer < (WORD *)(((UBYTE *)(term)) + AM.MaxTer) )
             AT.WorkPointer = (WORD *)(((UBYTE *)(term)) + AM.MaxTer);
        do {
SkipCount:  level++;
            if ( level > AR.Cnumlhs ) {
                if ( AR.DeferFlag && AR.sLevel <= 0 ) {
#ifdef WITHMPI
                  if ( PF.me != MASTER && AC.mparallelflag == PARALLELFLAG && PF.exprtodo < 0 ) {
                    if ( PF_Deferred(term,level) ) goto GenCall;
                  }
                  else
#endif
                    if ( Deferred(BHEAD term,level) ) goto GenCall;
                    goto Return0;
                }
                if ( AN.ncmod != 0 ) {
                    if ( Modulus(term) ) goto GenCall;
                    if ( !*term ) goto Return0;
                }
                if ( AR.CurDum > AM.IndDum && AR.sLevel <= 0 ) {
                    WORD olddummies = AN.IndDum;
                    AN.IndDum = AM.IndDum;
                    ReNumber(BHEAD term); Normalize(BHEAD term);
                    AN.IndDum = olddummies;
                    if ( !*term ) goto Return0;
                    olddummies = DetCurDum(BHEAD term);
                    if ( olddummies > AR.MaxDum ) AR.MaxDum = olddummies;
                }
                if ( AR.PolyFun > 0 && ( AR.sLevel <= 0 || AN.FunSorts[AR.sLevel]->PolyFlag > 0 ) ) {
                    if ( PrepPoly(BHEAD term,0) != 0 ) goto Return0;
                }
                else if ( AR.PolyFun > 0 ) {
                    if ( PrepPoly(BHEAD term,1) != 0 ) goto Return0;
                }
                if ( AR.sLevel <= 0 && AR.BracketOn ) {
                    if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
                    termout = AT.WorkPointer;
                    if ( AT.WorkPointer + *term + 3 > AT.WorkTop ) goto OverWork;
                    if ( PutBracket(BHEAD term) ) return(-1);
                    AN.RepPoint = RepSto;
                    *AT.WorkPointer = 0;
                    i = StoreTerm(BHEAD termout);
                    AT.WorkPointer = termout;
                    CC->numrhs = oldtoprhs;
                    CC->Pointer = CC->Buffer + oldcpointer;
                    CCC->numrhs = oldatoprhs;
                    CCC->Pointer = CCC->Buffer + oldacpointer;
                    return(i);
                }
                else {
                    if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
                    if ( AT.WorkPointer >= AT.WorkTop ) goto OverWork;
                    *AT.WorkPointer = 0;
                    AN.RepPoint = RepSto;
                    i = StoreTerm(BHEAD term);
                    CC->numrhs = oldtoprhs;
                    CC->Pointer = CC->Buffer + oldcpointer;
                    CCC->numrhs = oldatoprhs;
                    CCC->Pointer = CCC->Buffer + oldacpointer;
                    return(i);
                }
            }
            i = C->lhs[level][0];
            if ( i >= TYPECOUNT ) {
/*
            #[ Special action :
*/
                switch ( i ) {
                  case TYPECOUNT:
                    if ( CountDo(term,C->lhs[level]) < C->lhs[level][2] ) {
                        AT.WorkPointer = term + *term;
                        goto Return0;
                    }
```

```
                    break;
                  case TYPEMULT:
                    if ( MultDo(BHEAD term,C->lhs[level]) ) goto GenCall;
                    goto ReStart;
                  case TYPEGOTO:
                    level = AC.Labels[C->lhs[level][2]];
                    break;
                  case TYPEDISCARD:
                    AT.WorkPointer = term + *term;
                    goto Return0;
                  case TYPEIF:
#ifdef WITHPTHREADS
                    {
/*
                        We may be writing in the space here when wildcards
                        are involved in a match(). Hence we have to make
                        a private copy here!!!!
*/
                        WORD ic, jc, *ifcode, *jfcode;
                        jfcode = C->lhs[level]; jc = jfcode[1];
                        ifcode = AT.WorkPointer; AT.WorkPointer += jc;
                        for ( ic = 0; ic < jc; ic++ ) ifcode[ic] = jfcode[ic];
                        while ( !DoIfStatement(BHEAD ifcode,term) ) {
                            level = C->lhs[level][2];
                            if ( C->lhs[level][0] != TYPEELIF ) break;
                        }
                        AT.WorkPointer = ifcode;
                    }
#else
                    while ( !DoIfStatement(BHEAD C->lhs[level],term) ) {
                        level = C->lhs[level][2];
                        if ( C->lhs[level][0] != TYPEELIF ) break;
                    }
#endif
                    break;
                  case TYPEELIF:
                    do {
                        level = C->lhs[level][2];
                    } while ( C->lhs[level][0] == TYPEELIF );
                    break;
                  case TYPEELSE:
                  case TYPEENDIF:
                    level = C->lhs[level][2];
                    break;
                  case TYPESUMFIX:
                    {
                        WORD *cp = AR.CompressPointer, *op = AR.CompressPointer;
                        WORD *tlhs = C->lhs[level] + 3, *m, jlhs;
                        WORD theindex = C->lhs[level][2];
                        if ( theindex < 0 ) {   /* $-variable */
#ifdef WITHPTHREADS
                            int ddtype = -1;
                            theindex = -theindex;
                            d = Dollars + theindex;
                            if ( AS.MultiThreaded ) {
                                for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                                    if ( theindex == ModOptdollars[nummodopt].number ) break;
                                }
                                if ( nummodopt < NumModOptdollars ) {
                                    ddtype = ModOptdollars[nummodopt].type;
                                    if ( ddtype == MODLOCAL ) {
                                        d = ModOptdollars[nummodopt].dstruct+AT.identity;
                                    }
                                    else {
                                        LOCK(d->pthreadslockread);
                                    }
                                }
                            }
#else
                            theindex = -theindex;
                            d = Dollars + theindex;
#endif

                            if ( d->type != DOLINDEX
                            || d->index < AM.OffsetIndex
                            || d->index >= AM.OffsetIndex + WILDOFFSET ) {
                                MLOCK(ErrorMessageLock);
                                MesPrint("$%s should have been an index"
                                ,AC.dollarnames->namebuffer+d->name);
                                AN.currentTerm = term;
                                MesPrint("Current term: %t");
                                AN.listinprint = printscratch;
                                printscratch[0] = DOLLAREXPRESSION;
                                printscratch[1] = theindex;
                                MesPrint("$%s = %$"
                                ,AC.dollarnames->namebuffer+d->name);
                                MUNLOCK(ErrorMessageLock);
```

```
#ifdef WITHPTHREADS
                            if ( ddtype > 0 && ddtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                goto GenCall;
                            }
                            theindex = d->index;
#ifdef WITHPTHREADS
                            if ( ddtype > 0 && ddtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                        }
                        cp[1] = SUBEXPSIZE+4;
                        cp += SUBEXPSIZE;
                        *cp++ = INDTOIND;
                        *cp++ = 4;
                        *cp++ = theindex;
                        i = C->lhs[level][1] - 3;
                        cp++;
                        AR.CompressPointer = cp;
                        while ( --i >= 0 ) {
                            cp[-1] = *tlhs++;
                            termout = AT.WorkPointer;
                            if ( ( jlhs = WildFill(BHEAD termout,term,op)) < 0 )
                                goto GenCall;
                            m = term;
                            jlhs = *m;
                            while ( --jlhs >= 0 ) {
                                if ( *m++ != *termout++ ) break;
                            }
                            if ( jlhs >= 0 ) {
                                termout = AT.WorkPointer;
                                AT.WorkPointer = termout + *termout;
                                if ( Generator(BHEAD termout,level) ) goto GenCall;
                                AT.WorkPointer = termout;
                            }
                            else {
                                AR.CompressPointer = op;
                                goto SkipCount;
                            }
                        }
                        AR.CompressPointer = op;
                        goto CommonEnd;
                    }
                    case TYPESUM:
                    {
                        WORD *wp, *cp = AR.CompressPointer, *op = AR.CompressPointer;
                        WORD theindex;
                        WORD *ow;
/*
                        At this point it is safest to determine CurDum
*/
                        AR.CurDum = DetCurDum(BHEAD term);
                        i = C->lhs[level][1]-2;
                        wp = C->lhs[level] + 2;
                        cp[1] = SUBEXPSIZE+4*i;
                        cp += SUBEXPSIZE;
                        while ( --i >= 0 ) {
                            theindex = *wp++;
                            if ( theindex < 0 ) {   /* $-variable */
#ifdef WITHPTHREADS
                                int ddtype = -1;
                                theindex = -theindex;
                                d = Dollars + theindex;
                                if ( AS.MultiThreaded ) {
                                    for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                                        if ( theindex == ModOptdollars[nummodopt].number ) break;
                                    }
                                    if ( nummodopt < NumModOptdollars ) {
                                        ddtype = ModOptdollars[nummodopt].type;
                                        if ( ddtype == MODLOCAL ) {
                                            d = ModOptdollars[nummodopt].dstruct+AT.identity;
                                        }
                                        else {
                                            LOCK(d->pthreadslockread);
                                        }
                                    }
                                }
#else
                                theindex = -theindex;
                                d = Dollars + theindex;
#endif
                                if ( d->type != DOLINDEX
                                || d->index < AM.OffsetIndex
                                || d->index >= AM.OffsetIndex + WILDOFFSET ) {
                                    MLOCK(ErrorMessageLock);
                                    MesPrint("$%s should have been an index"
                                    ,AC.dollarnames->namebuffer+d->name);
                                    AN.currentTerm = term;
```

```
                                    MesPrint("Current term: %t");
                                    AN.listinprint = printscratch;
                                    printscratch[0] = DOLLAREXPRESSION;
                                    printscratch[1] = theindex;
                                    MesPrint("$%s = %$"
                                    ,AC.dollarnames->namebuffer+d->name);
                                    MUNLOCK(ErrorMessageLock);
#ifdef WITHPTHREADS
                                    if ( ddtype > 0 && ddtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                                    goto GenCall;
                                }
                                theindex = d->index;
#ifdef WITHPTHREADS
                                if ( ddtype > 0 && ddtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
                            }
                            *cp++ = INDTOIND;
                            *cp++ = 4;
                            *cp++ = theindex;
                            *cp++ = ++AR.CurDum;
                        }
                        ow = AT.WorkPointer;
                        AR.CompressPointer = cp;
                        if ( WildFill(BHEAD ow,term,op) < 0 ) goto GenCall;
                        AR.CompressPointer = op;
                        i = ow[0];
                        for ( j = 0; j < i; j++ ) term[j] = ow[j];
                        AT.WorkPointer = ow;
                        ReNumber(BHEAD term);
                        goto Renormalize;
                    }
                    case TYPECHISHOLM:
                      if ( Chisholm(BHEAD term,level) ) goto GenCall;
CommonEnd:
                      AT.WorkPointer = term + *term;
                      goto Return0;
                    case TYPEARG:
                      if ( ( i = execarg(BHEAD term,level) ) < 0 ) goto GenCall;
                      level = C->lhs[level][2];
                      if ( i > 0 ) goto ReStart;
                      break;
                    case TYPENORM:
                    case TYPENORM2:
                    case TYPENORM3:
                    case TYPENORM4:
                    case TYPESPLITARG:
                    case TYPESPLITARG2:
                    case TYPESPLITFIRSTARG:
                    case TYPESPLITLASTARG:
                    case TYPEARGTOEXTRASYMBOL:
                      if ( execarg(BHEAD term,level) < 0 ) goto GenCall;
                      level = C->lhs[level][2];
                      break;
                    case TYPEFACTARG:
                    case TYPEFACTARG2:
                      { WORD jjj;
                      if ( ( jjj = execarg(BHEAD term,level) ) < 0 ) goto GenCall;
                      if ( jjj > 0 ) goto ReStart;
                      level = C->lhs[level][2];
                      break; }
                    case TYPEEXIT:
                      if ( C->lhs[level][2] > 0 ) {
                          MLOCK(ErrorMessageLock);
                          MesPrint("%s",C->lhs[level]+3);
                          MUNLOCK(ErrorMessageLock);
                      }
                      Terminate(-1);
                      goto GenCall;
                    case TYPESETEXIT:
                      AM.exitflag = 1; /* no danger of race conditions */
                      break;
                    case TYPEPRINT:
                      AN.currentTerm = term;
                      AN.numlistinprint = (C->lhs[level][1] - C->lhs[level][2] - 3)/2;
                      AN.listinprint = C->lhs[level]+3+C->lhs[level][2];
                      MLOCK(ErrorMessageLock);
                      AO.ErrorBlock = 1;
                      MesPrint((char *)(C->lhs[level]+3));
                      AO.ErrorBlock = 0;
                      MUNLOCK(ErrorMessageLock);
                      break;
                    case TYPEFPRINT:
                      {
                      int oldFOflag;
```

```
                    WORD oldPrintType;
                    MLOCK(ErrorMessageLock);
                    oldFOflag = AM.FileOnlyFlag;
                    oldPrintType = AO.PrintType;
                    if ( AC.LogHandle >= 0 ) {
                        AM.FileOnlyFlag = 1;
                        AO.PrintType |= PRINTLFILE;
                    }
                    AN.currentTerm = term;
                    AN.numlistinprint = (C->lhs[level][1] - C->lhs[level][2] - 3)/2;
                    AN.listinprint = C->lhs[level]+3+C->lhs[level][2];
                    MesPrint((char *)(C->lhs[level]+3));
                    AO.PrintType = oldPrintType;
                    AM.FileOnlyFlag = oldFOflag;
                    MUNLOCK(ErrorMessageLock);
                    }
                    break;
                  case TYPEREDEFPRE:
                    j = C->lhs[level][2];
#ifdef WITHMPI
                    {
                        /*
                         * Regardless of parallel/nonparallel switch, we need to set
                         * AC.inputnumbers[ii], which indicates that the corresponding
                         * preprocessor variable is redefined and so we need to
                         * send/broadcast it.
                         */
                        int ii;
                        for ( ii = 0; ii < AC.numpfirstnum; ii++ ) {
                            if ( AC.pfirstnum[ii] == j ) break;
                        }
                        AC.inputnumbers[ii] = AN.ninterms;
                    }
#endif
#ifdef WITHPTHREADS
                    if ( AS.MultiThreaded ) {
                        int ii;
                        for ( ii = 0; ii < AC.numpfirstnum; ii++ ) {
                            if ( AC.pfirstnum[ii] == j ) break;
                        }
                        if ( AN.inputnumber < AC.inputnumbers[ii] ) break;
                        LOCK(AP.PreVarLock);
                        if ( AN.inputnumber >= AC.inputnumbers[ii] ) {
                            a = C->lhs[level]+4;
                            if ( a[a[-1]] == 0 )
                                PutPreVar(PreVar[j].name,(UBYTE *)(a),0,1);
                            else
                                PutPreVar(PreVar[j].name,(UBYTE *)(a)
                                    ,(UBYTE *)(a+a[-1]+1),1);
/*
                            PutPreVar(PreVar[j].name,(UBYTE *)(C->lhs[level]+4),0,1);
*/
                            AC.inputnumbers[ii] = AN.inputnumber;
                        }
                        UNLOCK(AP.PreVarLock);
                    }
                    else
#endif
                    {
                        a = C->lhs[level]+4;
                        LOCK(AP.PreVarLock);
                        if ( a[a[-1]] == 0 )
                            PutPreVar(PreVar[j].name,(UBYTE *)(a),0,1);
                        else
                            PutPreVar(PreVar[j].name,(UBYTE *)(a)
                                ,(UBYTE *)(a+a[-1]+1),1);
                        UNLOCK(AP.PreVarLock);
                    }
                    break;
                  case TYPERENUMBER:
                    AT.WorkPointer = term + *term;
                    if ( FullRenumber(BHEAD term,C->lhs[level][2]) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    if ( *term == 0 ) goto Return0;
                    break;
                  case TYPETRY:
                    if ( TryDo(BHEAD term,C->lhs[level],level) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    goto Return0;
                  case TYPEASSIGN:
                    { WORD onc = AR.NoCompress, oldEside = AR.Eside;
                    WORD oldrepeat = *AN.RepPoint;
/*
                    Here we have to assign an expression to a $ variable.
*/
                    AR.Eside = RHSIDE;
                    AR.NoCompress = 1;
```

```
                    AN.cTerm = AN.currentTerm = term;
                    AT.WorkPointer = term + *term;
                    *AT.WorkPointer++ = 0;
                    if ( AssignDollar(BHEAD term,level) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    AN.cTerm = 0;
                    *AN.RepPoint = oldrepeat;
                    AR.NoCompress = onc;
                    AR.Eside = oldEside;
                    break;
                    }
                  case TYPEFINDLOOP:
                    if ( Lus(term,C->lhs[level][3],C->lhs[level][4],
                    C->lhs[level][5],C->lhs[level][6],C->lhs[level][2]) ) {
                        AT.WorkPointer = term + *term;
                        goto Renormalize;
                    }
                    break;
                  case TYPEINSIDE:
                    if ( InsideDollar(BHEAD C->lhs[level],level) < 0 ) goto GenCall;
                    level = C->lhs[level][2];
                    break;
                  case TYPETERM:
                    retnorm = execterm(BHEAD term,level);
                    AN.RepPoint = RepSto;
                    AR.CurDum = DumNow;
                    CC->numrhs = oldtoprhs;
                    CC->Pointer = CC->Buffer + oldcpointer;
                    CCC->numrhs = oldatoprhs;
                    CCC->Pointer = CCC->Buffer + oldacpointer;
                    return(retnorm);
                  case TYPEDETCURDUM:
                    AT.WorkPointer = term + *term;
                    AR.CurDum = DetCurDum(BHEAD term);
                    break;
                  case TYPEINEXPRESSION:
                    {WORD *ll = C->lhs[level];
                    int numexprs = (int)(ll[1]-3);
                    ll += 3;
                    while ( numexprs-- >= 0 ) {
                        if ( *ll == AR.CurExpr ) break;
                        ll++;
                    }
                    if ( numexprs < 0 ) level = C->lhs[level][2];
                    }
                    break;
                  case TYPEMERGE:
                    AT.WorkPointer = term + *term;
                    if ( DoShuffle(BHEAD term,level,C->lhs[level][2],C->lhs[level][3]) )
                        goto GenCall;
                    AT.WorkPointer = term + *term;
                    goto Return0;
                  case TYPESTUFFLE:
                    AT.WorkPointer = term + *term;
                    if ( DoStuffle(BHEAD term,level,C->lhs[level][2],C->lhs[level][3]) )
                        goto GenCall;
                    AT.WorkPointer = term + *term;
                    goto Return0;
                  case TYPETESTUSE:
                    AT.WorkPointer = term + *term;
                    if ( TestUse(term,level) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    break;
                  case TYPEAPPLY:
                    AT.WorkPointer = term + *term;
                    if ( ApplyExec(term,C->lhs[level][2],level) < C->lhs[level][2] ) {
                        AT.WorkPointer = term + *term;
                        *AN.RepPoint = 1;
                        goto ReStart;
                    }
                    AT.WorkPointer = term + *term;
                    break;
/*
                  case TYPEAPPLYRESET:
                    AT.WorkPointer = term + *term;
                    if ( ApplyReset(level) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    break;
*/
                  case TYPECHAININ:
                    AT.WorkPointer = term + *term;
                    if ( ChainIn(BHEAD term,C->lhs[level][2]) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    break;
                  case TYPECHAINOUT:
                    AT.WorkPointer = term + *term;
                    if ( ChainOut(BHEAD term,C->lhs[level][2]) ) goto GenCall;
```

```
                    AT.WorkPointer = term + *term;
                    break;
                  case TYPEFACTOR:
                    AT.WorkPointer = term + *term;
                    if ( DollarFactorize(BHEAD C->lhs[level][2]) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    break;
                  case TYPEARGIMPLODE:
                    AT.WorkPointer = term + *term;
                    if ( ArgumentImplode(BHEAD term,C->lhs[level]) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    break;
                  case TYPEARGEXPLODE:
                    AT.WorkPointer = term + *term;
                    if ( ArgumentExplode(BHEAD term,C->lhs[level]) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    break;
                  case TYPEDENOMINATORS:
                    DenToFunction(term,C->lhs[level][2]);
                    break;
                  case TYPEDROPCOEFFICIENT:
                    DropCoefficient(BHEAD term);
                    break;
                  case TYPETRANSFORM:
                    AT.WorkPointer = term + *term;
                    if ( RunTransform(BHEAD term,C->lhs[level]+2) ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    if ( *term == 0 ) goto Return0;
                    goto ReStart;
                  case TYPETOPOLYNOMIAL:
                    AT.WorkPointer = term + *term;
                    termout = AT.WorkPointer;
                    if ( ConvertToPoly(BHEAD term,termout,C->lhs[level],0) < 0 ) goto GenCall;
                    if ( *termout == 0 ) goto Return0;
                    i = termout[0]; t = term; NCOPY(t,termout,i);
                    AT.WorkPointer = term + *term;
                    break;
                  case TYPEFROMPOLYNOMIAL:
                    AT.WorkPointer = term + *term;
                    termout = AT.WorkPointer;
                    if ( ConvertFromPoly(BHEAD term,termout,0,numxsymbol,0,0) < 0 ) goto GenCall;
                    if ( *term == 0 ) goto Return0;
                    i = termout[0]; t = term; NCOPY(t,termout,i);
                    AT.WorkPointer = term + *term;
                    goto ReStart;
                  case TYPEDOLOOP:
                    level = TestDoLoop(BHEAD C->lhs[level],level);
                    if ( level < 0 ) goto GenCall;
                    break;
                  case TYPEENDDOLOOP:
                    level = TestEndDoLoop(BHEAD C->lhs[C->lhs[level][2]],C->lhs[level][2]);
                    if ( level < 0 ) goto GenCall;
                    break;
                  case TYPEDROPSYMBOLS:
                    DropSymbols(BHEAD term);
                    break;
                  case TYPEPUTINSIDE:
                    AT.WorkPointer = term + *term;
                    if ( PutInside(BHEAD term,C->lhs[level]) < 0 ) goto GenCall;
                    AT.WorkPointer = term + *term;
                    /*
                     * We need to call Generator() to convert slow notation to
                     * fast notation, which fixes Issue #30.
                     */
                    if ( Generator(BHEAD term,level) < 0 ) goto GenCall;
                    goto Return0;
                  case TYPETOSPECTATOR:
                    if ( PutInSpectator(term,C->lhs[level][2]) < 0 ) goto GenCall;
                    goto Return0;
                }
                goto SkipCount;
/*
            #] Special action :
*/
            }
        } while ( ( i = TestMatch(BHEAD term,&level) ) == 0 );
        if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
        if ( i > 0 ) replac = TestSub(BHEAD term,level);
        else replac = i;
        if ( replac >= 0 || AT.TMout[1] != SYMMETRIZE ) {
            *AN.RepPoint = 1;
            AR.expchanged = 1;
        }
        if ( replac < 0 ) {     /* Terms come from automatic generation */
AutoGen:    i = *AT.TMout;
            t = termout = AT.WorkPointer;
            if ( ( AT.WorkPointer += i ) > AT.WorkTop ) goto OverWork;
```

```
            accum = AT.TMout;
            while ( --i >= 0 ) *t++ = *accum++;
            if ( (*(FG.Operation[termout[1]]))(BHEAD term,termout,replac,level) ) goto GenCall;
            AT.WorkPointer = termout;
            goto Return0;
        }
    }
    if ( applyflag ) { TableReset(); applyflag = 0; }
/*  DumNow = AR.CurDum; */

    if ( AN.TeInFun ) { /* Match in function argument */
        if ( AN.TeInFun < 0 && !AN.TeSuOut ) {

            if ( AR.TePos >= 0 ) goto AutoGen;
            switch ( AN.TeInFun ) {
                case -1:
                    if ( DoDistrib(BHEAD term,level) ) goto GenCall;
                    break;
                case -2:
                    if ( DoDelta3(BHEAD term,level) ) goto GenCall;
                    break;
                case -3:
                    if ( DoTableExpansion(term,level) ) goto GenCall;
                    break;
                case -4:
                    if ( FactorIn(BHEAD term,level) ) goto GenCall;
                    break;
                case -5:
                    if ( FactorInExpr(BHEAD term,level) ) goto GenCall;
                    break;
                case -6:
                    if ( TermsInBracket(BHEAD term,level) < 0 ) goto GenCall;
                    break;
                case -7:
                    if ( ExtraSymFun(BHEAD term,level) < 0 ) goto GenCall;
                    break;
                case -8:
                    if ( GCDfunction(BHEAD term,level) < 0 ) goto GenCall;
                    break;
                case -9:
                    if ( DIVfunction(BHEAD term,level,0) < 0 ) goto GenCall;
                    break;
                case -10:
                    if ( DIVfunction(BHEAD term,level,1) < 0 ) goto GenCall;
                    break;
                case -11:
                    if ( DIVfunction(BHEAD term,level,2) < 0 ) goto GenCall;
                    break;
                case -12:
                    if ( DoPermutations(BHEAD term,level) ) goto GenCall;
                    break;
                case -13:
                    if ( DoPartitions(BHEAD term,level) ) goto GenCall;
                    break;
                case -14:
                    if ( DIVfunction(BHEAD term,level,3) < 0 ) goto GenCall;
                    break;
            }
        }
        else {
            termout = AT.WorkPointer;
            AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
            if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
            if ( InFunction(BHEAD term,termout) ) goto GenCall;
            AT.WorkPointer = termout + *termout;
            *AN.RepPoint = 1;
            AR.expchanged = 1;
            if ( *termout && Generator(BHEAD termout,level) < 0 ) goto GenCall;
            AT.WorkPointer = termout;
        }
    }
    else if ( replac > 0 ) {
        power = AN.TeSuOut;
        tepos = AR.TePos;
        if ( power < 0 ) {  /* Table expansion */
            power = -power; tepos = 0;
        }
        extractbuff = AT.TMbuff;
        if ( extractbuff == AM.dbufnum ) {
            d = DolToTerms(BHEAD replac);
            if ( d && d->where != 0 ) {
                iscopy = 1;
                if ( AT.TMdolfac > 0 ) {    /* We need a factor */
                  if ( AT.TMdolfac == 1 ) {
                    if ( d->nfactors ) {
                        numfac[0] = 4;
                        numfac[1] = d->nfactors;
```

```
                        numfac[2] = 1;
                        numfac[3] = 3;
                        numfac[4] = 0;
                    }
                    else {
                        numfac[0] = 0;
                    }
                    StartBuf = numfac;
                }
                else {
                    if ( (AT.TMdolfac-1) > d->nfactors && d->nfactors > 0 ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Attempt to use an nonexisting factor %d of a $-variable",(WORD)(AT.TMdolfac-1));
                        if ( d->nfactors == 1 )
                            MesPrint("There is only one factor");
                        else
                            MesPrint("There are only %d factors",(WORD)(d->nfactors));
                        MUNLOCK(ErrorMessageLock);
                        goto GenCall;
                    }
                    if ( d->nfactors > 1 ) {
                        DOLLARS dd;
                        LONG dsize;
                        WORD *td1, *td2;
                        dd = Dollars + replac;
#ifdef WITHPTHREADS
                        {
                            int nummodopt, dtype = -1;
                            if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
                                for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                                    if ( replac == ModOptdollars[nummodopt].number ) break;
                                }
                                if ( nummodopt < NumModOptdollars ) {
                                    dtype = ModOptdollars[nummodopt].type;
                                    if ( dtype == MODLOCAL ) {
                                        dd = ModOptdollars[nummodopt].dstruct+AT.identity;
                                    }
                                }
                            }
                        }
#endif
                        dsize = dd->factors[AT.TMdolfac-2].size;
/*
                        We copy only the factor we need
*/
                        if ( dsize == 0 ) {
                            numfac[0] = 4;
                            numfac[1] = d->factors[AT.TMdolfac-2].value;
                            numfac[2] = 1;
                            numfac[3] = 3;
                            numfac[4] = 0;
                            StartBuf = numfac;
                            if ( numfac[1] < 0 ) {
                                numfac[1] = -numfac[1];
                                numfac[3] = -numfac[3];
                            }
                        }
                        else {
                        d->factors[AT.TMdolfac-2].where = td2 = (WORD *)Malloc1(
                            (dsize+1)*sizeof(WORD),"Copy of factor");
                        td1 = dd->factors[AT.TMdolfac-2].where;
                        StartBuf = td2;
                        d->size = dsize; d->type = DOLTERMS;
                        NCOPY(td2,td1,dsize);
                        *td2 = 0;
                        }
                    }
                    else if ( d->nfactors == 1 ) {
                        StartBuf = d->where;
                    }
                    else {
                        MLOCK(ErrorMessageLock);
                        if ( d->nfactors == 0 ) {
                            MesPrint("Attempt to use factor %d of an unfactored $-variable",(WORD)(AT.TMdolfac-1));
                        }
                        else {
                            MesPrint("Internal error. Illegal number of factors for $-variable");
                        }
                        MUNLOCK(ErrorMessageLock);
                        goto GenCall;
                    }
                }
            }
            else StartBuf = d->where;
        }
```

```
            else {
                d = Dollars + replac;
                StartBuf = zeroDollar;
            }
            posisub = 0;
            i = DetCommu(d->where);
#ifdef WITHPTHREADS
            if ( AS.MultiThreaded ) {
                for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                    if ( replac == ModOptdollars[nummodopt].number ) break;
                }
                if ( nummodopt < NumModOptdollars ) {
                    dtype = ModOptdollars[nummodopt].type;
                    if ( dtype != MODLOCAL && dtype != MODSUM ) {
                        if ( StartBuf[0] && StartBuf[StartBuf[0]] ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("A dollar variable with modoption max or min can have only one term");
                            MUNLOCK(ErrorMessageLock);
                            goto GenCall;
                        }
                        LOCK(d->pthreadslockread);
                    }
                }
            }
#endif
        }
        else {
            StartBuf = cbuf[extractbuff].Buffer;
            posisub = cbuf[extractbuff].rhs[replac] - StartBuf;
            i = (WORD)cbuf[extractbuff].CanCommu[replac];
        }
        if ( power == 1 ) {     /* Just a single power */
            termout = AT.WorkPointer;
            AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
            if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
            while ( StartBuf[posisub] ) {
                if ( extractbuff == AT.allbufnum ) WildDollars(BHEAD &(StartBuf[posisub]));
                AT.WorkPointer = (WORD *)(((UBYTE *)(termout)) + AM.MaxTer);
                if ( InsertTerm(BHEAD term,replac,extractbuff,
                    &(StartBuf[posisub]),termout,tepos) < 0 ) goto GenCall;
                AT.WorkPointer = termout + *termout;
                *AN.RepPoint = 1;
                AR.expchanged = 1;
                posisub += StartBuf[posisub];
/*
                    For multiple table substitutions it may be better to
                    do modulus arithmetic right here
                    Turns out to be not very effective.

                if ( AN.ncmod != 0 ) {
                    if ( Modulus(termout) ) goto GenCall;
                    if ( !*termout ) goto Return0;
                }
*/
#ifdef WITHPTHREADS
                if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { UNLOCK(d->pthreadslockread); }
                if ( ( AS.Balancing && CC->numrhs == 0 ) && StartBuf[posisub] ) {
                    if ( ( id = ConditionalGetAvailableThread() ) >= 0 ) {
                        if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
                    }
                }
                else
#endif
                if ( Generator(BHEAD termout,level) < 0 ) goto GenCall;
#ifdef WITHPTHREADS
                if ( dtype > 0 && dtype != MODLOCAL ) { dtype = 0; break; }
#endif
                if ( iscopy == 0 && ( extractbuff != AM.dbufnum ) ) {
/*
                    There are cases in which a bigger buffer is created
                    on the fly, like with wildcard buffers.
                    We play it safe here. Maybe we can be more selective
                    in some distant future?
*/
                    StartBuf = cbuf[extractbuff].Buffer;
                }
            }
            if ( extractbuff == AT.allbufnum ) {
                CBUF *Ce = cbuf + extractbuff;
                Ce->Pointer = Ce->rhs[Ce->numrhs--];
            }
#ifdef WITHPTHREADS
            if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { UNLOCK(d->pthreadslockread); dtype = 0; }
#endif
```

```
            if ( iscopy ) {
                if ( d->nfactors > 1 ) {
                    int j;
                    for ( j = 0; j < d->nfactors; j++ ) {
                        if ( d->factors[j].where ) M_free(d->factors[j].where,"Copy of factor");
                    }
                    M_free(d->factors,"Dollar factors");
                }
                M_free(d,"Copy of dollar variable");
                d = 0; iscopy = 0;
            }
            AT.WorkPointer = termout;
        }
        else if ( i <= 1 ) {        /* Use binomials */
            LONG posit, olw;
            WORD *same, *ow = AT.WorkPointer;
            LONG olpw = AT.posWorkPointer;
            power1 = power+1;
            WantAddLongs(power1);
            olw = posit = AT.lWorkPointer; AT.lWorkPointer += power1;
            same = ++AT.WorkPointer;
            a = accum = ( AT.WorkPointer += power1+1 );
            AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
            if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
            AT.lWorkSpace[posit] = posisub;
            same[-1] = 0;
            *same = 1;
            *accum = 0;
            tepos = AR.TePos;
            i = 1;
            do {
                if ( StartBuf[AT.lWorkSpace[posit]] ) {
                    if ( ( a = PasteTerm(BHEAD i-1,accum,
                        &(StartBuf[AT.lWorkSpace[posit]]),i,*same) ) == 0 )
                        goto GenCall;
                    AT.lWorkSpace[posit+1] = AT.lWorkSpace[posit];
                    same[1] = *same + 1;
                    if ( i > 1 && AT.lWorkSpace[posit] < AT.lWorkSpace[posit-1] ) *same = 1;
                    AT.lWorkSpace[posit] += StartBuf[AT.lWorkSpace[posit]];
                    i++;
                    posit++;
                    same++;
                }
                else {
                    i--; posit--; same--;
                }
                if ( i > power ) {
                    termout = AT.WorkPointer = a;
                    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
                    if ( AT.WorkPointer > AT.WorkTop )
                        goto OverWork;
                    if ( FiniTerm(BHEAD term,accum,termout,replac,tepos) ) goto GenCall;
                    AT.WorkPointer = termout + *termout;
                    *AN.RepPoint = 1;
                    AR.expchanged = 1;
#ifdef WITHPTHREADS
                    if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { UNLOCK(d->pthreadslockread); }
                    if ( ( AS.Balancing && CC->numrhs == 0 ) && ( i > 0 )
                    && ( id = ConditionalGetAvailableThread() ) >= 0 ) {
                        if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
                    }
                    else
#endif
                    if ( Generator(BHEAD termout,level) ) goto GenCall;
#ifdef WITHPTHREADS
                    if ( dtype > 0 && dtype != MODLOCAL ) { dtype = 0; break; }
#endif
                    if ( iscopy == 0 && ( extractbuff != AM.dbufnum ) )
                            StartBuf = cbuf[extractbuff].Buffer;
                    i--; posit--; same--;
                }
            } while ( i > 0 );
#ifdef WITHPTHREADS
            if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { UNLOCK(d->pthreadslockread); dtype = 0; }
#endif
            if ( iscopy ) {
                if ( d->nfactors > 1 ) {
                    int j;
                    for ( j = 0; j < d->nfactors; j++ ) {
                        if ( d->factors[j].where ) M_free(d->factors[j].where,"Copy of factor");
                    }
                    M_free(d->factors,"Dollar factors");
                }
                M_free(d,"Copy of dollar variable");
                d = 0; iscopy = 0;
```

```
            }
            AT.WorkPointer = ow; AT.lWorkPointer = olw; AT.posWorkPointer = olpw;
        }
        else {                          /* No binomials */
            LONG posit, olw, olpw = AT.posWorkPointer;
            WantAddLongs(power);
            posit = olw = AT.lWorkPointer; AT.lWorkPointer += power;
            a = accum = AT.WorkPointer;
            AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
            if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
            for ( i = 0; i < power; i++ ) AT.lWorkSpace[posit++] = posisub;
            posit = olw;
            *accum = 0;
            tepos = AR.TePos;
            i = 0;
            while ( i >= 0 ) {
                if ( StartBuf[AT.lWorkSpace[posit]] ) {
                    if ( ( a = PasteTerm(BHEAD i,accum,
                        &(StartBuf[AT.lWorkSpace[posit]]),1,1) ) == 0 ) goto GenCall;
                    AT.lWorkSpace[posit] += StartBuf[AT.lWorkSpace[posit]];
                    i++; posit++;
                }
                else {
                    AT.lWorkSpace[posit--] = posisub;
                    i--;
                }
                if ( i >= power ) {
                    termout = AT.WorkPointer = a;
                    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
                    if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
                    if ( FiniTerm(BHEAD term,accum,termout,replac,tepos) ) goto GenCall;
                    AT.WorkPointer = termout + *termout;
                    *AN.RepPoint = 1;
                    AR.expchanged = 1;
#ifdef WITHPTHREADS
                    if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { UNLOCK(d->pthreadslockread); }
                    if ( ( AS.Balancing && CC->numrhs == 0 ) && ( i > 0 ) && ( id = ConditionalGetAvailableThread() ) >= 0 ) {
                        if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
                    }
                    else
#endif
                    if ( Generator(BHEAD termout,level) ) goto GenCall;
#ifdef WITHPTHREADS
                    if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { dtype = 0; break; }
#endif
                    if ( iscopy == 0 && ( extractbuff != AM.dbufnum ) )
                            StartBuf = cbuf[extractbuff].Buffer;
                    i--; posit--;
                }
            }
#ifdef WITHPTHREADS
            if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { UNLOCK(d->pthreadslockread); dtype = 0; }
#endif
            if ( iscopy ) {
                if ( d->nfactors > 1 ) {
                    int j;
                    for ( j = 0; j < d->nfactors; j++ ) {
                        if ( d->factors[j].where ) M_free(d->factors[j].where,"Copy of factor");
                    }
                    M_free(d->factors,"Dollar factors");
                }
                M_free(d,"Copy of dollar variable");
                d = 0; iscopy = 0;
            }
            AT.WorkPointer = accum;
            AT.lWorkPointer = olw;
            AT.posWorkPointer = olpw;
        }
    }
    else {                              /* Expression from disk */
        POSITION StartPos;
        LONG position, olpw, opw, comprev, extra;
        RENUMBER renumber;
        WORD *Freeze, *aa, *dummies;
        replac = -replac-1;
        power = AN.TeSuOut;
        Freeze = AN.Frozen;
        if ( Expressions[replac].status == STOREDEXPRESSION ) {
            POSITION firstpos;
            SETSTARTPOS(firstpos);

/*          Note that AT.TMaddr is needed for GetTable just once! */
/*
            We need space for the previous term in the compression
```

```
            This is made available in AR.CompressBuffer, although we may get
            problems with this sooner or later. Hence we need to keep
            a set of pointers in AR.CompressBuffer
            Note that after the last call there has been no use made
            of AR.CompressPointer, so it points automatically at its original
            position!
*/
            WantAddPointers(power+1);
            comprev = opw = AT.pWorkPointer;
            AT.pWorkPointer += power+1;
            WantAddPositions(power+1);
            position = olpw = AT.posWorkPointer;
            AT.posWorkPointer += power + 1;

            AT.pWorkSpace[comprev++] = AR.CompressPointer;

            for ( i = 0; i < power; i++ ) {
                PUTZERO(AT.posWorkSpace[position]); position++;
            }
            position = olpw;
            if ( ( renumber = GetTable(replac,&(AT.posWorkSpace[position]),1) ) == 0 ) goto GenCall;
            dummies = AT.WorkPointer;
            *dummies++ = AR.CurDum;
            AT.WorkPointer += power+2;
            accum = AT.WorkPointer;
            AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
            if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
            aa = AT.WorkPointer;
            *accum = 0;
            i = 0; StartPos = AT.posWorkSpace[position];
            dummies[i] = AR.CurDum;
            while ( i >= 0 ) {
skippedfirst:
                AR.CompressPointer = AT.pWorkSpace[comprev-1];
                if ( ( extra = PasteFile(BHEAD i,accum,&(AT.posWorkSpace[position])
                        ,&a,renumber,Freeze,replac) ) < 0 ) goto GenCall;
                if ( Expressions[replac].numdummies > 0 ) {
                    AR.CurDum = dummies[i] + Expressions[replac].numdummies;
                }
                if ( NOTSTARTPOS(firstpos) ) {
                    if ( ISMINPOS(firstpos) || ISEQUALPOS(firstpos,AT.posWorkSpace[position]) ) {
                        firstpos = AT.posWorkSpace[position];
/*
                        ADDPOS(AT.posWorkSpace[position],extra * sizeof(WORD));
*/
                        goto skippedfirst;
                    }
                }
                if ( extra ) {
/*
                    ADDPOS(AT.posWorkSpace[position],extra * sizeof(WORD));
*/
                    i++; AT.posWorkSpace[++position] = StartPos;
                    AT.pWorkSpace[comprev++] = AR.CompressPointer;
                    dummies[i] = AR.CurDum;
                }
                else {
                    PUTZERO(AT.posWorkSpace[position]); position--; i--;
                    AR.CurDum = dummies[i];
                    comprev--;
                }
                if ( i >= power ) {
                    termout = AT.WorkPointer = a;
                    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2*AM.MaxTer);
                    if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
                    if ( FiniTerm(BHEAD term,accum,termout,replac,0) ) goto GenCall;
                    if ( *termout ) {
                        AT.WorkPointer = termout + *termout;
                        *AN.RepPoint = 1;
                        AR.expchanged = 1;
#ifdef WITHPTHREADS
                        if ( ( AS.Balancing && CC->numrhs == 0 ) && ( i > 0 ) && ( id =
ConditionalGetAvailableThread() ) >= 0 ) {
                            if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;

                        }
                        else
#endif
                        if ( Generator(BHEAD termout,level) ) goto GenCall;
                    }
                    i--; position--;
                    AR.CurDum = dummies[i];
                    comprev--;
                }
                AT.WorkPointer = aa;
            }
            AT.WorkPointer = accum;
```

```
            AT.posWorkPointer = olpw;
            AT.pWorkPointer = opw;
/*
        Bug fix. See also GetTable
#ifdef WITHPTHREADS
            M_free(renumber->symb.lo,"VarSpace");
            M_free(renumber,"Renumber");
#endif
*/
            if ( renumber->symb.lo != AN.dummyrenumlist )
                M_free(renumber->symb.lo,"VarSpace");
            M_free(renumber,"Renumber");

        }
        else {          /* Active expression */
            aa = accum = AT.WorkPointer;
            if ( ( (WORD *)(((UBYTE *)(AT.WorkPointer)) + 2 * AM.MaxTer + sizeof(WORD)) ) > AT.WorkTop )
                    goto OverWork;
            *accum++ = -1; AT.WorkPointer++;
            if ( DoOnePow(BHEAD term,power,replac,accum,aa,level,Freeze) ) goto GenCall;
            AT.WorkPointer = aa;
        }
    }
Return0:
    AR.CurDum = DumNow;
    AN.RepPoint = RepSto;
    CC->numrhs = oldtoprhs;
    CC->Pointer = CC->Buffer + oldcpointer;
    CCC->numrhs = oldatoprhs;
    CCC->Pointer = CCC->Buffer + oldacpointer;
    return(0);

GenCall:
    if ( AM.tracebackflag ) {
        termout = term;
        MLOCK(ErrorMessageLock);
        AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
        AO.OutSkip = 3;
        FiniLine();
        i = *termout;
        while ( --i >= 0 ) {
            TalToLine((UWORD)(*termout++));
            TokenToLine((UBYTE *)"  ");
        }
        AO.OutSkip = 0;
        FiniLine();
        MesCall("Generator");
        MUNLOCK(ErrorMessageLock);
    }
    CC->numrhs = oldtoprhs;
    CC->Pointer = CC->Buffer + oldcpointer;
    CCC->numrhs = oldatoprhs;
    CCC->Pointer = CCC->Buffer + oldacpointer;
    return(-1);
OverWork:
    CC->numrhs = oldtoprhs;
    CC->Pointer = CC->Buffer + oldcpointer;
    CCC->numrhs = oldatoprhs;
    CCC->Pointer = CCC->Buffer + oldacpointer;
    MLOCK(ErrorMessageLock);
    MesWork();
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
        #] Generator :
        #[ DoOnePow :           WORD DoOnePow(term,power,nexp,accum,aa,level,freeze)
*/
#ifdef WITHPTHREADS
char freezestring[] = "freeze<-xxxx";
#endif

WORD DoOnePow(PHEAD WORD *term, WORD power, WORD nexp, WORD * accum,
              WORD *aa, WORD level, WORD *freeze)
{
    GETBIDENTITY
    POSITION oldposition, startposition;
    WORD *acc, *termout, fromfreeze = 0;
    WORD *oldipointer = AR.CompressPointer;
    FILEHANDLE *fi;
    WORD type, retval;
    WORD oldGetOneFile = AR.GetOneFile;
    WORD olddummies = AR.CurDum;
    WORD extradummies = Expressions[nexp].numdummies;
/*
```

```
    The next code is for some tricky debugging. (5-jan-2010 JV)
    Normally it should be disabled.
*/
/*
#ifdef WITHPTHREADS
    if ( freeze ) {
        MLOCK(ErrorMessageLock);
        if ( AT.identity < 10 ) {
            freezestring[8] = '0'+AT.identity;
            freezestring[9] = '>';
            freezestring[10] = 0;
        }
        else if ( AT.identity < 100 ) {
            freezestring[8] = '0'+AT.identity/10;
            freezestring[9] = '0'+AT.identity%10;
            freezestring[10] = '>';
            freezestring[11] = 0;
        }
        else {
            freezestring[8] = 0;
        }
        PrintTerm(freeze,freezestring);
        MUNLOCK(ErrorMessageLock);
    }
#else
    if ( freeze ) PrintTerm(freeze,"freeze");
#endif
*/
    type = Expressions[nexp].status;
    if ( type == HIDDENLEXPRESSION || type == HIDDENGEXPRESSION
     || type == DROPHLEXPRESSION || type == DROPHGEXPRESSION
     || type == UNHIDELEXPRESSION || type == UNHIDEGEXPRESSION ) {
        AR.GetOneFile = 2; fi = AR.hidefile;
    }
    else {
        AR.GetOneFile = 0; fi = AR.infile;
    }
    if ( fi->handle >= 0 ) {
        PUTZERO(oldposition);
#ifdef WITHSEEK
        LOCK(AS.inputslock);
        SeekFile(fi->handle,&oldposition,SEEK_CUR);
        UNLOCK(AS.inputslock);
#endif
    }
    else {
        SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
    }
    if ( freeze && ( Expressions[nexp].bracketinfo != 0 ) ) {
        POSITION *brapos;
/*
        There is a bracket index
        AR.CompressPointer = oldipointer;
*/
        (*aa)++;
        power--;
        if ( ( brapos = FindBracket(nexp,freeze) ) == 0 )
            goto EndExpr;
        startposition = *brapos;
        goto doterms;
    }
    startposition = AS.OldOnFile[nexp];
    retval = GetOneTerm(BHEAD accum,fi,&startposition,0);
    if ( retval > 0 ) {         /* Skip prototype */
        (*aa)++;
        power--;
doterms:
        AR.CompressPointer = oldipointer;
        for (;;) {
            retval = GetOneTerm(BHEAD accum,fi,&startposition,0);
            if ( retval <= 0 ) break;
/*
            Here should come the code to test for [].
*/
            if ( freeze ) {
                WORD *t, *m, *r, *mstop;
                WORD *tset;
                t = accum;
                m = freeze;
                m += *m;
                m -= ABS(m[-1]);
                mstop = m;
                m = freeze + 1;
                r = t;
                r += *t;
                r -= ABS(r[-1]);
                t++;
```

```
                tset = t;
                while ( t < r && *t != HAAKJE ) t += t[1];
                if ( t >= r ) {
                    if ( m < mstop ) {
                        if ( fromfreeze ) goto EndExpr;
                        goto NextTerm;
                    }
                    t = tset;
                }
                else {
                    r = tset;
                    while ( r < t && m < mstop ) {
                        if ( *r == *m ) { m++; r++; }
                        else {
                            if ( fromfreeze ) goto EndExpr;
                            goto NextTerm;
                        }
                    }
                    if ( r < t || m < mstop ) {
                        if ( fromfreeze ) goto EndExpr;
                        goto NextTerm;
                    }
                }
                fromfreeze = 1;
                r = tset;
                m = accum;
                m += *m;
                while ( t < m ) *r++ = *t++;
                *accum = WORDDIF(r,accum);
            }
            if ( extradummies > 0 ) {
                if ( olddummies > AM.IndDum ) {
                    MoveDummies(BHEAD accum,olddummies-AM.IndDum);
                }
                AR.CurDum = olddummies+extradummies;
            }
            acc = accum;
            acc += *acc;
            if ( power <= 0 ) {
                termout = acc;
                AT.WorkPointer = (WORD *)(((UBYTE *)(acc)) + 2*AM.MaxTer);
                if ( AT.WorkPointer > AT.WorkTop ) {
                    MLOCK(ErrorMessageLock);
                    MesWork();
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                if ( FiniTerm(BHEAD term,aa,termout,nexp,0) ) goto PowCall;
                if ( *termout ) {
                    MarkPolyRatFunDirty(termout)
/*                  PolyFunDirty(BHEAD termout); */
                    AT.WorkPointer = termout + *termout;
                    *AN.RepPoint = 1;
                    AR.expchanged = 1;
                    if ( Generator(BHEAD termout,level) ) goto PowCall;
                }
            }
            else {
                if ( acc > AT.WorkTop ) {
                    MLOCK(ErrorMessageLock);
                    MesWork();
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                if ( DoOnePow(BHEAD term,power,nexp,acc,aa,level,freeze) ) goto PowCall;
            }
NextTerm:;
            AR.CompressPointer = oldipointer;
        }
EndExpr:
        (*aa)--;
    }
    AR.CompressPointer = oldipointer;
    if ( fi->handle >= 0 ) {
#ifdef WITHSEEK
        LOCK(AS.inputslock);
        SeekFile(fi->handle,&oldposition,SEEK_SET);
        UNLOCK(AS.inputslock);
        if ( ISNEGPOS(oldposition) ) {
            MLOCK(ErrorMessageLock);
            MesPrint("File error");
            goto PowCall2;
        }
#endif
    }
    else {
        fi->POfill = fi->PObuffer + BASEPOSITION(oldposition);
```

```
    }
    AR.GetOneFile = oldGetOneFile;
    AR.CurDum = olddummies;
    return(0);
PowCall:;
    MLOCK(ErrorMessageLock);
#ifdef WITHSEEK
PowCall2:;
#endif
    MesCall("DoOnePow");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] DoOnePow :
        #[ Deferred :           WORD Deferred(term,level)
*/
WORD Deferred(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    POSITION startposition;
    WORD *t, *m, *mstop, *tstart, decr, oldb, *termout, i, *oldwork, retval;
    WORD *oldipointer = AR.CompressPointer, *oldPOfill = AR.infile->POfill;
    WORD oldGetOneFile = AR.GetOneFile;
    AR.GetOneFile = 1;
    oldwork = AT.WorkPointer;
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    termout = AT.WorkPointer;
    AR.DeferFlag = 0;
    startposition = AR.DefPosition;
/*
        Store old position
*/
    if ( AR.infile->handle >= 0 ) {
/*
        PUTZERO(oldposition);
        SeekFile(AR.infile->handle,&oldposition,SEEK_CUR);
*/
    }
    else {
/*
        SETBASEPOSITION(oldposition,AR.infile->POfill-AR.infile->PObuffer);
*/
        AR.infile->POfill = (WORD *)((UBYTE *)(AR.infile->PObuffer)
                    +BASEPOSITION(startposition));
    }
/*
        Look in the CompressBuffer where the bracket contents start
*/
    t = m = AR.CompressBuffer;
    t += *t;
    mstop = t - ABS(t[-1]);
    m++;
    while ( *m != HAAKJE && m < mstop ) m += m[1];
    if ( m >= mstop ) { /* No deferred action! */
        AT.WorkPointer = term + *term;
        if ( Generator(BHEAD term,level) ) goto DefCall;
        AR.DeferFlag = 1;
        AT.WorkPointer = oldwork;
        AR.GetOneFile = oldGetOneFile;
        return(0);
    }
    mstop = m + m[1];
    decr = WORDDIF(mstop,AR.CompressBuffer)-1;
    tstart = AR.CompressPointer + decr;

    m = AR.CompressBuffer;
    t = AR.CompressPointer;
    i = *m;
    NCOPY(t,m,i);
    oldb = *tstart;
    AR.TePos = 0;
    AN.TeSuOut = 0;
/*
        Status:
        First bracket content starts at mstop.
        Next term starts at startposition.
        Decompression information is in AR.CompressPointer.
        The outside of the bracket runs from AR.CompressBuffer+1 to mstop.
*/
    for(;;) {
        *tstart = *(AR.CompressPointer)-decr;
        AR.CompressPointer = AR.CompressPointer+AR.CompressPointer[0];
        if ( InsertTerm(BHEAD term,0,AM.rbufnum,tstart,termout,0) < 0 ) {
            goto DefCall;
        }
```

```
        *tstart = oldb;
        AT.WorkPointer = termout + *termout;
        if ( Generator(BHEAD termout,level) ) goto DefCall;
        AR.CompressPointer = oldipointer;
        AT.WorkPointer = termout;
        retval = GetOneTerm(BHEAD AT.WorkPointer,AR.infile,&startposition,0);
        if ( retval >= 0 ) AR.CompressPointer = oldipointer;
        if ( retval <= 0 ) break;
        t = AR.CompressPointer;
        if ( *t < (1 + decr + ABS(*(t+*t-1))) ) break;
        t++;
        m = AR.CompressBuffer+1;
        while ( m < mstop ) {
            if ( *m != *t ) goto Thatsit;
            m++; t++;
        }
    }
Thatsit:;
/*
        Finished. Reposition the file, restore information and return.
*/
    if ( AR.infile->handle < 0 ) AR.infile->POfill = oldPOfill;
    AR.DeferFlag = 1;
    AR.GetOneFile = oldGetOneFile;
    AT.WorkPointer = oldwork;
    return(0);
DefCall:;
    MLOCK(ErrorMessageLock);
    MesCall("Deferred");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] Deferred :
        #[ PrepPoly :           WORD PrepPoly(term,par)
*/
WORD PrepPoly(PHEAD WORD *term,WORD par)
{
    GETBIDENTITY
    WORD count = 0, i, jcoef, ncoef;
    WORD *t, *m, *r, *tstop, *poly = 0, *v, *w, *vv, *ww;
    WORD *oldworkpointer = AT.WorkPointer;
/*
    The problem here is that the function will be forced into 'long'
    notation. After this -SNUMBER,1 becomes 6,0,4,1,1,3 and the
    pattern matcher cannot match a short 1 with a long 1.
    But because this is an undocumented feature for very special
    purposes, we don't do anything about it. (30-aug-2011)
*/
    if ( AR.PolyFunType == 2 && AR.PolyFunExp != 2 ) {
        WORD oldtype = AR.SortType;
        AR.SortType = SORTHIGHFIRST;
        if ( poly_ratfun_normalize(BHEAD term) != 0 ) Terminate(-1);
/*      if ( ReadPolyRatFun(BHEAD term) != 0 ) Terminate(-1); */
        oldworkpointer = AT.WorkPointer;
        AR.SortType = oldtype;
    }
    AT.PolyAct = 0;
    t = term;
    GETSTOP(t,tstop);
    t++;
    while ( t < tstop ) {
        if ( *t == AR.PolyFun ) {
            if ( count > 0 ) return(0);
            poly = t;
            count++;
        }
        t += t[1];
    }
    r = m = term + *term;
    i = ABS(m[-1]);
    if ( par > 0 ) {
        if ( count == 0 ) return(0);
        else if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) )
            goto DoOne;
        else if ( AR.PolyFunType == 2 )
            goto DoTwo;
        else
            goto DoError;
    }
    else if ( count == 0 ) {
/*
        #[ Create a PolyFun :
*/
        poly = t = tstop;
        if ( i == 3 && m[-2] == 1 && (m[-3]&MAXPOSITIVE) == m[-3] ) {
```

```
            *m++ = AR.PolyFun;
            if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) ) {
                *m++ = FUNHEAD+2;
                FILLFUN(m)
                *m++ = -SNUMBER;
                *m = m[-2-FUNHEAD] < 0 ? -m[-4-FUNHEAD]: m[-4-FUNHEAD];
                m++;
            }
            else if ( AR.PolyFunType == 2 ) {
                *m++ = FUNHEAD+4;
                FILLFUN(m)
                *m++ = -SNUMBER;
                *m = m[-2-FUNHEAD] < 0 ? -m[-4-FUNHEAD]: m[-4-FUNHEAD];
                m++;
                *m++ = -SNUMBER;
                *m++ = 1;
            }
        }
        else {
            r = tstop;
            if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) ) {
                *m++ = AR.PolyFun;
                *m++ = FUNHEAD+ARGHEAD+i+1;
                FILLFUN(m)
                *m++ = ARGHEAD+i+1;
                *m++ = 0;
                FILLARG(m)
                *m++ = i+1;
                NCOPY(m,r,i);
            }
            else if ( AR.PolyFunType == 2 ) {
                WORD *num, *den, size, sign, sizenum, sizeden;
                if ( m[-1] < 0 ) { sign = -1; size = -m[-1]; }
                else             { sign =  1; size =  m[-1]; }
                num = m - size; size = (size-1)/2; den = num + size;
                sizenum = size; while ( num[sizenum-1] == 0 ) sizenum--;
                sizeden = size; while ( den[sizeden-1] == 0 ) sizeden--;
                v = m;
                AT.PolyAct = WORDDIF(v,term);
                *v++ = AR.PolyFun;
                *v++ = FUNHEAD + 2*(ARGHEAD+sizenum+sizeden+2);
                FILLFUN(v);
                *v++ = ARGHEAD+2*sizenum+2;
                *v++ = 0;
                FILLARG(v);
                *v++ = 2*sizenum+2;
                for ( i = 0; i < sizenum; i++ ) *v++ = num[i];
                *v++ = 1;
                for ( i = 1; i < sizenum; i++ ) *v++ = 0;
                *v++ = sign*(2*sizenum+1);
                *v++ = ARGHEAD+2*sizeden+2;
                *v++ = 0;
                FILLARG(v);
                *v++ = 2*sizeden+2;
                for ( i = 0; i < sizeden; i++ ) *v++ = den[i];
                *v++ = 1;
                for ( i = 1; i < sizeden; i++ ) *v++ = 0;
                *v++ = 2*sizeden+1;
                w = num;
                i = v - m;
                NCOPY(w,m,i);
                *w++ = 1; *w++ = 1; *w++ = 3; *term = w - term;
                return(0);
            }
        }
/*
        #] Create a PolyFun :
*/
    }
    else if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) ) {
        DoOne:;
/*
        #[ One argument :
*/
        m = term + *term;
        r = poly + poly[1];
        if ( ( poly[1] == FUNHEAD+2 && poly[FUNHEAD+1] == 0
        && poly[FUNHEAD] == -SNUMBER ) || poly[1] == FUNHEAD ) return(1);
        t = poly + FUNHEAD;
        if ( t >= r ) return(0);
        if ( m[-1] == 3 && *tstop == 1 && tstop[1] == 1 ) {
            i = poly[1];
            t = poly;
            NCOPY(m,t,i);
        }
        else if ( *t <= -FUNCTION ) {
            if ( t+1 < r ) return(0);   /* More than one argument */
```

```
            r = tstop;
            *m++ = AR.PolyFun;
            *m++ = FUNHEAD*2+ARGHEAD+i+1;
            FILLFUN(m)
            *m++ = FUNHEAD+ARGHEAD+i+1;
            *m++ = 0;
            FILLARG(m)
            *m++ = FUNHEAD+i+1;
            *m++ = -*t++;
            *m++ = FUNHEAD;
            FILLFUN(m)
            NCOPY(m,r,i);
        }
        else if ( *t < 0 ) {
            if ( t+2 < r ) return(0);   /* More than one argument */
            r = tstop;
            if ( *t == -SNUMBER ) {
                if ( t[1] == 0 ) return(1); /* Term should be zero now */
                *m = AR.PolyFun;
                w = m+1;
                m += FUNHEAD+ARGHEAD;
                v = m;
                *m++ = 5+i;
                *m++ = SNUMBER;
                *m++ = 4;
                *m++ = t[1];
                *m++ = 1;
                NCOPY(m,r,i);
                if ( m >= AT.WorkSpace && m < AT.WorkTop )
                    AT.WorkPointer = m;
                if ( Normalize(BHEAD v) ) Terminate(-1);
                AT.WorkPointer = oldworkpointer;
                m = w;
                if ( *v == 4 && v[2] == 1 && (v[1]&MAXPOSITIVE) == v[1] ) {
                    *m++ = FUNHEAD+2;
                    FILLFUN(m)
                    *m++ = -SNUMBER;
                    *m++ = v[3] < 0 ? -v[1] : v[1];
                }
                else if ( *v == 0 ) return(1);
                else {
                    *m++ = FUNHEAD+ARGHEAD+*v;
                    FILLFUN(m)
                    *m++ = ARGHEAD+*v;
                    *m++ = 0;
                    FILLARG(m)
                    m = v + *v;
                }
            }
            else if ( *t == -SYMBOL ) {
                *m++ = AR.PolyFun;
                *m++ = FUNHEAD+ARGHEAD+5+i;
                FILLFUN(m)
                *m++ = ARGHEAD+5+i;
                *m++ = 0;
                FILLARG(m)
                *m++ = 5+i;
                *m++ = SYMBOL;
                *m++ = 4;
                *m++ = t[1];
                *m++ = 1;
                NCOPY(m,r,i);
            }
            else return(0);         /* Not symbol-like */
        }
        else {
            if ( t + *t < r ) return(0); /* More than one argument */
            i = m[-1];
            *m++ = AR.PolyFun;
            w = m;
            m += ARGHEAD+FUNHEAD-1;
            t += ARGHEAD;
            jcoef = i < 0 ? (i+1)»1:(i-1)»1;
            v = t;
/*
            Test now the scalar nature of the argument.
            No indices allowed.
*/
            while ( t < r ) {
                WORD *vstop;
                vv = t + *t;
                vstop = vv - ABS(vv[-1]);
                t++;
                while( t < vstop ) {
                    if ( *t == INDEX ) return(0);
                    t += t[1];
                }
```

```
                t = vv;
            }
/*
            Now multiply each term by the coefficient.
*/
            t = v;
            while ( t < r ) {
                ww = m;
                v = t + *t;
                ncoef = v[-1];
                vv = v - ABS(ncoef);
                if ( ncoef < 0 ) ncoef++;
                else ncoef--;
                ncoef >>= 1;
                while ( t < vv ) *m++ = *t++;
                if ( MulRat(BHEAD (UWORD *)vv,ncoef,(UWORD *)tstop,jcoef,
                    (UWORD *)m,&ncoef) ) Terminate(-1);
                ncoef <<= 1;
                m += ABS(ncoef);
                if ( ncoef < 0 ) ncoef--;
                else ncoef++;
                *m++ = ncoef;
                *ww = WORDDIF(m,ww);
                if ( AN.ncmod != 0 ) {
                    if ( Modulus(ww) ) Terminate(-1);
                    if ( *ww == 0 ) return(1);
                    m = ww + *ww;
                }
                t = v;
            }
            *w = (WORDDIF(m,w))+1;
            w[FUNHEAD-1] = w[0] - FUNHEAD;
            w[FUNHEAD] = 0;
            w[1] = 0; /* omission survived for years. 23-mar-2006 JV */
            w += FUNHEAD-1;
            if ( ToFast(w,w) ) {
                if ( *w <= -FUNCTION ) { w[-FUNHEAD+1] = FUNHEAD+1; m = w+1; }
                else { w[-FUNHEAD+1] = FUNHEAD+2; m = w+2; }

            }
        }
        t = poly + poly[1];
        while ( t < tstop ) *poly++ = *t++;
/*
        #] One argument :
*/
    }
    else if ( AR.PolyFunType == 2 ) {
        DoTwo:;
/*
        #[ Two arguments :
*/
        WORD *num, *den, size, sign, sizenum, sizeden;
/*
        First make sure that the PolyFun is last
*/
        m = term + *term;
        if ( poly + poly[1] < tstop ) {
            for ( i = 0; i < poly[1]; i++ ) m[i] = poly[i];
            t = poly; v = poly + poly[1];
            while ( v < tstop ) *t++ = *v++;
            poly = t;
            for ( i = 0; i < m[1]; i++ ) t[i] = m[i];
            t += m[1];
        }
        AT.PolyAct = WORDDIF(poly,term);
/*
        If needed we convert the coefficient into a PolyRatFun and then
        we call poly_ratfun_normalize
*/
        if ( m[-1] == 3 && m[-2] == 1 && m[-3] == 1 ) return(0);
        if ( AR.PolyFunExp != 1 ) {
        if ( m[-1] < 0 ) { sign = -1; size = -m[-1]; } else { sign = 1; size = m[-1]; }
        num = m - size; size = (size-1)/2; den = num + size;
        sizenum = size; while ( num[sizenum-1] == 0 ) sizenum--;
        sizeden = size; while ( den[sizeden-1] == 0 ) sizeden--;
        v = m;
        *v++ = AR.PolyFun;
        *v++ = FUNHEAD + 2*(ARGHEAD+sizenum+sizeden+2);
/*      *v++ = MUSTCLEANPRF; */
        *v++ = 0;
        FILLFUN3(v);
        *v++ = ARGHEAD+2*sizenum+2;
        *v++ = 0;
        FILLARG(v);
        *v++ = 2*sizenum+2;
        for ( i = 0; i < sizenum; i++ ) *v++ = num[i];
```

```
        *v++ = 1;
        for ( i = 1; i < sizenum; i++ ) *v++ = 0;
        *v++ = sign*(2*sizenum+1);
        *v++ = ARGHEAD+2*sizeden+2;
        *v++ = 0;
        FILLARG(v);
        *v++ = 2*sizeden+2;
        for ( i = 0; i < sizeden; i++ ) *v++ = den[i];
        *v++ = 1;
        for ( i = 1; i < sizeden; i++ ) *v++ = 0;
        *v++ = 2*sizeden+1;
        w = num;
        i = v - m;
        NCOPY(w,m,i);
        }
        else {
            w = m-ABS(m[-1]);
        }
        *w++ = 1; *w++ = 1; *w++ = 3; *term = w - term;
        {
            WORD oldtype = AR.SortType;
            AR.SortType = SORTHIGHFIRST;
/*
            if ( count > 0 )
                poly_ratfun_normalize(BHEAD term);
            else
                ReadPolyRatFun(BHEAD term);
*/
            poly_ratfun_normalize(BHEAD term);

/*          oldworkpointer = AT.WorkPointer; */
            AR.SortType = oldtype;
        }
        goto endofit;
/*
        #] Two arguments :
*/
    }
    else {
        DoError:;
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal value for PolyFunType in PrepPoly");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    r = term + *term;
    AT.PolyAct = WORDDIF(poly,term);
    while ( r < m ) *poly++ = *r++;
    *poly++ = 1;
    *poly++ = 1;
    *poly++ = 3;
    *term = WORDDIF(poly,term);
endofit:;
    return(0);
}

/*
        #] PrepPoly :
        #[ PolyFunMul :         WORD PolyFunMul(term)
*/
WORD PolyFunMul(PHEAD WORD *term)
{
    GETBIDENTITY
    WORD *t, *fun1, *fun2, *t1, *t2, *m, *w, *ww, *tt1, *tt2, *tt4, *arg1, *arg2;
    WORD *tstop, i, dirty = 0, OldPolyFunPow = AR.PolyFunPow, minp1, minp2;
    WORD n1, n2, i1, i2, l1, l2, l3, l4, action = 0, noac = 0, retval = 0;
    if ( AR.PolyFunType == 2 && AR.PolyFunExp == 1 ) {
        WORD pow = 0, pow1;
        t = term + 1; t1 = term + *term; t1 -= ABS(t1[-1]);
        w = t;
        while ( t < t1 ) {
            if ( *t != AR.PolyFun ) {
SkipFun:
                if ( t == w ) { t += t[1]; w = t; }
                else { i = t[1]; NCOPY(w,t,i) }
                continue;
            }
            pow1 = 0;
            t2 = t + t[1]; t += FUNHEAD;
            if ( *t < 0 ) {
                if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) pow1++;
                else if ( *t != -SNUMBER ) goto NoLegal;
                t += 2;
            }
            else if ( t[0] == ARGHEAD+8 && t[ARGHEAD] == 8
             && t[ARGHEAD+1] == SYMBOL && t[ARGHEAD+3] == AR.PolyFunVar
             && t[ARGHEAD+5] == 1 && t[ARGHEAD+6] == 1 && t[ARGHEAD+7] == 3 ) {
```

```
                pow1 += t[ARGHEAD+4];
                t += *t;
            }
            else {
NoLegal:
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal term with divergence in PolyRatFun");
                MesCall("PolyFunMul");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            if ( *t < 0 ) {
                if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) pow1--;
                else if ( *t != -SNUMBER ) goto NoLegal;
                t += 2;
            }
            else if ( t[0] == ARGHEAD+8 && t[ARGHEAD] == 8
             && t[ARGHEAD+1] == SYMBOL && t[ARGHEAD+3] == AR.PolyFunVar
             && t[ARGHEAD+5] == 1 && t[ARGHEAD+6] == 1 && t[ARGHEAD+7] == 3 ) {
                pow1 -= t[ARGHEAD+4];
                t += *t;
            }
            else goto NoLegal;
            if ( t == t2 ) pow += pow1;
            else goto SkipFun;
        }
        m = w;
        *w++ = AR.PolyFun; *w++ = 0; FILLFUN(w);
        if ( pow > 1 ) {
            *w++ = 8+ARGHEAD; *w++ = 0; FILLARG(w);
            *w++ = 8; *w++ = SYMBOL; *w++ = 4; *w++ = AR.PolyFunVar; *w++ = pow;
            *w++ = 1; *w++ = 1; *w++ = 3; *w++ = -SNUMBER; *w++ = 1;
        }
        else if ( pow == 1 ) {
            *w++ = -SYMBOL; *w++ = AR.PolyFunVar; *w++ = -SNUMBER; *w++ = 1;
        }
        else if ( pow < -1 ) {
            *w++ = -SNUMBER; *w++ = 1; *w++ = 8+ARGHEAD; *w++ = 0; FILLARG(w);
            *w++ = 8; *w++ = SYMBOL; *w++ = 4; *w++ = AR.PolyFunVar; *w++ = -pow;
            *w++ = 1; *w++ = 1; *w++ = 3;
        }
        else if ( pow == -1 ) {
            *w++ = -SNUMBER; *w++ = 1; *w++ = -SYMBOL; *w++ = AR.PolyFunVar;
        }
        else {
            *w++ = -SNUMBER; *w++ = 1; *w++ = -SNUMBER; *w++ = 1;
        }
        m[1] = w - m;
        *w++ = 1; *w++ = 1; *w++ = 3;
        *term = w - term;
        if ( w > AT.WorkSpace && w < AT.WorkTop ) AT.WorkPointer = w;
        return(0);
    }
ReStart:
    if ( AR.PolyFunType == 2 && ( ( AR.PolyFunExp != 2 )
     || ( AR.PolyFunExp == 2 && AN.PolyNormFlag > 1 ) ) ) {
        WORD count1 = 0, count2 = 0, count3;
        WORD oldtype = AR.SortType;
        t = term + 1; t1 = term + *term; t1 -= ABS(t1[-1]);
        while ( t < t1 ) {
            if ( *t == AR.PolyFun ) {
              if ( t[2] && dirty == 0 ) { /* Any dirty flag on? */
                dirty = 1;
/*              ReadPolyRatFun(BHEAD term); */
/*              ToPolyFunGeneral(BHEAD term); */
                poly_ratfun_normalize(BHEAD term);
                if ( term[0] == 0 ) return(0);
                count1 = 0;
                action++;
                goto ReStart;
              }
              t2 = t + t[1]; tt2 = t+FUNHEAD; count3 = 0;
              while ( tt2 < t2 ) { count3++; NEXTARG(tt2); }
              if ( count3 == 2 ) {
                count1++;
                if ( ( t[2] & MUSTCLEANPRF ) != 0 ) {   /* Better civilize this guy */
                    action++;
                    w = AT.WorkPointer;
                    AR.SortType = SORTHIGHFIRST;
                    t2 = t + t[1]; tt2 = t+FUNHEAD;
                    while ( tt2 < t2 ) {
                        if ( *tt2 > 0 ) {
                            tt4 = tt2; tt1 = tt2 + ARGHEAD; tt2 += *tt2;
                            NewSort(BHEAD0);
                            while ( tt1 < tt2 ) {
                                i = *tt1; ww = w; NCOPY(ww,tt1,i);
                                AT.WorkPointer = ww;
```

```
                                Normalize(BHEAD w);
                                StoreTerm(BHEAD w);
                            }
                            EndSort(BHEAD w,1);
                            ww = w; while ( *ww ) ww += *ww;
                            if ( ww-w != *tt4-ARGHEAD ) { /* Little problem */
/*
                                Solution: brute force copy
                                Maybe it will never come here????
*/
                                WORD *r1 = TermMalloc("PolyFunMul");
                                WORD ii = (ww-w)-(*tt4-ARGHEAD); /* increment */
                                WORD *r2 = tt4+ARGHEAD, *r3, *r4 = r1;
                                i = r2 - term; r3 = term; NCOPY(r4,r3,i);
                                i = ww-w; ww = w; NCOPY(r4,ww,i);
                                r3 = tt2; i = term+*term-tt2; NCOPY(r4,r3,i);
                                *r1 = i = r4-r1; r4 = term; r3 = r1;
                                NCOPY(r4,r3,i);
                                t[1] += ii; t1 += ii; *tt4 += ii;
                                tt2 = tt4 + *tt4;
                                TermFree(r1,"PolyFunMul");
                            }
                            else {
                                i = ww-w; ww = w; tt1 = tt4+ARGHEAD;
                                NCOPY(tt1,ww,i);
                                AT.WorkPointer = w;
                            }
                        }
                        else if ( *tt2 <= -FUNCTION ) tt2++;
                        else tt2 += 2;
                    }
                    AR.SortType = oldtype;
                }
            }
        }
        t += t[1];
    }
    if ( count1 <= 1 ) { goto checkaction; }
    if ( AR.PolyFunExp == 1 ) {
        t = term + *term; t -= ABS(t[-1]);
        *t++ = 1; *t++ = 1; *t++ = 3; *term = t - term;
    }
    {
        AR.SortType = SORTHIGHFIRST;
/*      retval = ReadPolyRatFun(BHEAD term); */
/*      ToPolyFunGeneral(BHEAD term); */
        retval = poly_ratfun_normalize(BHEAD term);
        if ( *term == 0 ) return(retval);
        AR.SortType = oldtype;
    }

    t = term + 1; t1 = term + *term; t1 -= ABS(t1[-1]);
    while ( t < t1 ) {
        if ( *t == AR.PolyFun ) {
          t2 = t + t[1]; tt2 = t+FUNHEAD; count3 = 0;
          while ( tt2 < t2 ) { count3++; NEXTARG(tt2); }
          if ( count3 == 2 ) {
            count2++;
          }
        }
        t += t[1];
    }
    if ( count1 >= count2 ) {
        t = term + 1;
        while ( t < t1 ) {
            if ( *t == AR.PolyFun ) {
                t2 = t;
                t = t + t[1];
                t2[2] |= (DIRTYFLAG|MUSTCLEANPRF);
                t2 += FUNHEAD;
                while ( t2 < t ) {
                    if ( *t2 > 0 ) t2[1] = DIRTYFLAG;
                    NEXTARG(t2);
                }
            }
            else t += t[1];
        }
    }

    w = term + *term;
    if ( w > AT.WorkSpace && w < AT.WorkTop ) AT.WorkPointer = w;
checkaction:
    if ( action ) retval = action;
    return(retval);
  }
retry:
  if ( term >= AT.WorkSpace && term+*term < AT.WorkTop )
```

```c
        AT.WorkPointer = term + *term;
    GETSTOP(term,tstop);
    t = term+1;
    while ( *t != AR.PolyFun && t < tstop ) t += t[1];
    while ( t < tstop && *t == AR.PolyFun ) {
        if ( t[1] > FUNHEAD ) {
            if ( t[FUNHEAD] < 0 ) {
                if ( t[FUNHEAD] <= -FUNCTION && t[1] == FUNHEAD+1 ) break;
                if ( t[FUNHEAD]  > -FUNCTION && t[1] == FUNHEAD+2 ) {
                    if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) {
                        *term = 0;
                        return(0);
                    }
                    break;
                }
            }
            else if ( t[FUNHEAD] == t[1] - FUNHEAD ) break;
        }
        noac = 1;
        t += t[1];
    }
    if ( *t != AR.PolyFun || t >= tstop ) goto done;
    fun1 = t;
    t += t[1];
    while ( t < tstop && *t == AR.PolyFun ) {
        if ( t[1] > FUNHEAD ) {
            if ( t[FUNHEAD] < 0 ) {
                if ( t[FUNHEAD] <= -FUNCTION && t[1] == FUNHEAD+1 ) break;
                if ( t[FUNHEAD]  > -FUNCTION && t[1] == FUNHEAD+2 ) {
                    if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) {
                        *term = 0;
                        return(0);
                    }
                    break;
                }
            }
            else if ( t[FUNHEAD] == t[1] - FUNHEAD ) break;
        }
        noac = 1;
        t += t[1];
    }
    if ( *t != AR.PolyFun || t >= tstop ) goto done;
    fun2 = t;
/*
    We have two functions of the proper type.
    Count terms (needed for the specials)
*/
    t = fun1 + FUNHEAD;
    if ( *t < 0 ) {
        n1 = 1; arg1 = AT.WorkPointer;
        ToGeneral(t,arg1,1);
        AT.WorkPointer = arg1 + *arg1;
    }
    else {
        t += ARGHEAD;
        n1 = 0; t1 = fun1 + fun1[1]; arg1 = t;
        while ( t < t1 ) { n1++; t += *t; }
    }
    t = fun2 + FUNHEAD;
    if ( *t < 0 ) {
        n2 = 1; arg2 = AT.WorkPointer;
        ToGeneral(t,arg2,1);
        AT.WorkPointer = arg2 + *arg2;
    }
    else {
        t += ARGHEAD;
        n2 = 0; t2 = fun2 + fun2[1]; arg2 = t;
        while ( t < t2 ) { n2++; t += *t; }
    }
/*
    Now we can start the multiplications. We first multiply the terms
    without coefficients, then normalize, and finally put the coefficients
    in place. This is because one has often truncated series and the
    high powers may get killed, while their coefficients are the most
    expensive ones.
    Note: We may run into fun(-SNUMBER,value)
*/
    w = AT.WorkPointer;
    NewSort(BHEAD0);
    if ( AR.PolyFunType == 2 && AR.PolyFunExp == 2 ) {
        AT.TrimPower = 1;
/*
        We have to find the lowest power in both polynomials.
        This will be needed to temporarily correct the AR.PolyFunPow
*/
        minp1 = MAXPOWER;
        for ( t1 = arg1, i1 = 0; i1 < n1; i1++, t1 += *t1 ) {
```

```
            if ( *t1 == 4 ) {
                if ( minp1 > 0 ) minp1 = 0;
            }
            else if ( ABS(t1[*t1-1]) == (*t1-1) ) {
                if ( minp1 > 0 ) minp1 = 0;
            }
            else {
                if ( t1[1] == SYMBOL && t1[2] == 4 && t1[3] == AR.PolyFunVar ) {
                    if ( t1[4] < minp1 ) minp1 = t1[4];
                }
                else {
                    MesPrint("Illegal term in expanded polyratfun.");
                    goto PolyCall;
                }
            }
        }
        minp2 = MAXPOWER;
        for ( t2 = arg2, i2 = 0; i2 < n2; i2++, t2 += *t2 ) {
            if ( *t2 == 4 ) {
                if ( minp2 > 0 ) minp2 = 0;
            }
            else if ( ABS(t2[*t2-1]) == (*t2-1) ) {
                if ( minp2 > 0 ) minp2 = 0;
            }
            else {
                if ( t2[1] == SYMBOL && t2[2] == 4 && t2[3] == AR.PolyFunVar ) {
                    if ( t2[4] < minp2 ) minp2 = t2[4];
                }
                else {
                    MesPrint("Illegal term in expanded polyratfun.");
                    goto PolyCall;
                }
            }
        }
        AR.PolyFunPow += minp1+minp2;
    }
    for ( t1 = arg1, i1 = 0; i1 < n1; i1++, t1 += *t1 ) {
    for ( t2 = arg2, i2 = 0; i2 < n2; i2++, t2 += *t2 ) {
        m = w;
        m++;
        GETSTOP(t1,tt1);
        t = t1 + 1;
        while ( t < tt1 ) *m++ = *t++;
        GETSTOP(t2,tt2);
        t = t2+1;
        while ( t < tt2 ) *m++ = *t++;
        *m++ = 1; *m++ = 1; *m++ = 3; *w = WORDDIF(m,w);
        AT.WorkPointer = m;
        if ( Normalize(BHEAD w) ) { LowerSortLevel(); goto PolyCall; }
        if ( *w ) {
            m = w + *w;
            if ( m[-1] != 3 || m[-2] != 1 || m[-3] != 1 ) {
                l3 = REDLENG(m[-1]);
                m -= ABS(m[-1]);
                t = t1 + *t1 - 1;
                l1 = REDLENG(*t);
                if ( MulRat(BHEAD (UWORD *)m,l3,(UWORD *)tt1,l1,(UWORD *)m,&l4) ) {
                    LowerSortLevel(); goto PolyCall; }
                if ( AN.ncmod != 0 && TakeModulus((UWORD *)m,&l4,AC.cmod,AN.ncmod,UNPACK|AC.modmode) ) {
                    LowerSortLevel(); goto PolyCall; }
                if ( l4 == 0 ) continue;
                t = t2 + *t2 - 1;
                l2 = REDLENG(*t);
                if ( MulRat(BHEAD (UWORD *)m,l4,(UWORD *)tt2,l2,(UWORD *)m,&l3) ) {
                    LowerSortLevel(); goto PolyCall; }
                if ( AN.ncmod != 0 && TakeModulus((UWORD *)m,&l3,AC.cmod,AN.ncmod,UNPACK|AC.modmode) ) {
                    LowerSortLevel(); goto PolyCall; }
            }
            else {
                m -= 3;
                t = t1 + *t1 - 1;
                l1 = REDLENG(*t);
                t = t2 + *t2 - 1;
                l2 = REDLENG(*t);
                if ( MulRat(BHEAD (UWORD *)tt1,l1,(UWORD *)tt2,l2,(UWORD *)m,&l3) ) {
                    LowerSortLevel(); goto PolyCall; }
                if ( AN.ncmod != 0 && TakeModulus((UWORD *)m,&l3,AC.cmod,AN.ncmod,UNPACK|AC.modmode) ) {
                    LowerSortLevel(); goto PolyCall; }
            }
            if ( l3 == 0 ) continue;
            l3 = INCLENG(l3);
            m += ABS(l3);
            m[-1] = l3;
            *w = WORDDIF(m,w);
```

```
            AT.WorkPointer = m;
            if ( StoreTerm(BHEAD w) ) { LowerSortLevel(); goto PolyCall; }
        }
    }
    }
    if ( EndSort(BHEAD w,0) < 0 ) goto PolyCall;
    AR.PolyFunPow = OldPolyFunPow;
    AT.TrimPower = 0;
    if ( *w == 0 ) {
        *term = 0;
        return(0);
    }
    t = w;
    while ( *t ) t += *t;
    AT.WorkPointer = t;
    n1 = WORDDIF(t,w);
    t1 = term;
    while ( t1 < fun1 ) *t++ = *t1++;
    t2 = t;
    *t++ = AR.PolyFun;
    *t++ = FUNHEAD+ARGHEAD+n1;
    *t++ = 0;
    FILLFUN3(t)
    *t++ = ARGHEAD+n1;
    *t++ = 0;
    FILLARG(t)
    NCOPY(t,w,n1);
    if ( ToFast(t2+FUNHEAD,t2+FUNHEAD) ) {
        if ( t2[FUNHEAD] > -FUNCTION ) t2[1] = FUNHEAD+2;
        else t2[FUNHEAD] = FUNHEAD+1;
        t = t2 + t2[1];
    }
    t1 = fun1 + fun1[1];
    while ( t1 < fun2 ) *t++ = *t1++;
    t1 = fun2 + fun2[1];
    t2 = term + *term;
    while ( t1 < t2 ) *t++ = *t1++;
    *AT.WorkPointer = n1 = WORDDIF(t,AT.WorkPointer);
    if ( n1*((LONG)sizeof(WORD)) > AM.MaxTer ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Term too complex. Maybe increasing MaxTermSize can help");
        goto PolyCall2;
    }
    m = term; t = AT.WorkPointer;
    NCOPY(m,t,n1);
    action++;
    goto retry;
done:
    AT.WorkPointer = term + *term;
    if ( action && noac ) {
        if ( Normalize(BHEAD term) ) goto PolyCall;
        AT.WorkPointer = term + *term;
    }
    return(0);
PolyCall:;
    MLOCK(ErrorMessageLock);
PolyCall2:;
    AR.PolyFunPow = OldPolyFunPow;
    MesCall("PolyFunMul");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] PolyFunMul :
    #] Processor :
*/
```

4.88 ratio.c File Reference

`#include "form3.h"`
Include dependency graph for ratio.c:

Data Structures

- struct ARGBUFFER

Functions

- WORD RatioFind (PHEAD WORD ∗term, WORD ∗params)
- WORD RatioGen (PHEAD WORD ∗term, WORD ∗params, WORD num, WORD level)
- WORD BinomGen (PHEAD WORD ∗term, WORD level, WORD ∗∗tstops, WORD x1, WORD x2, WORD pow1, WORD pow2, WORD sign, UWORD ∗coef, WORD ncoef)
- WORD DoSumF1 (PHEAD WORD ∗term, WORD ∗params, WORD replac, WORD level)
- WORD Glue (PHEAD WORD ∗term1, WORD ∗term2, WORD ∗sub, WORD insert)
- WORD DoSumF2 (PHEAD WORD ∗term, WORD ∗params, WORD replac, WORD level)
- int GCDfunction (PHEAD WORD ∗term, WORD level)
- WORD ∗ GCDfunction3 (PHEAD WORD ∗in1, WORD ∗in2)
- WORD ∗ PutExtraSymbols (PHEAD WORD ∗in, WORD startebuf, int ∗actionflag)
- WORD ∗ TakeExtraSymbols (PHEAD WORD ∗in, WORD startebuf)
- WORD ∗ MultiplyWithTerm (PHEAD WORD ∗in, WORD ∗term, WORD par)
- WORD ∗ TakeContent (PHEAD WORD ∗in, WORD ∗term)
- int MergeSymbolLists (PHEAD WORD ∗old, WORD ∗extra, int par)
- int MergeDotproductLists (PHEAD WORD ∗old, WORD ∗extra, int par)
- WORD ∗ CreateExpression (PHEAD WORD nexp)
- int GCDterms (PHEAD WORD ∗term1, WORD ∗term2, WORD ∗termout)
- int ReadPolyRatFun (PHEAD WORD ∗term)
- int FromPolyRatFun (PHEAD WORD ∗fun, WORD ∗∗numout, WORD ∗∗denout)
- WORD ∗ TakeSymbolContent (PHEAD WORD ∗in, WORD ∗term)
- void GCDclean (PHEAD WORD ∗num, WORD ∗den)
- WORD ∗ PolyDiv (PHEAD WORD ∗a, WORD ∗b, char ∗text)
- int DIVfunction (PHEAD WORD ∗term, WORD level, int par)
- WORD ∗ MULfunc (PHEAD WORD ∗p1, WORD ∗p2)
- WORD ∗ ConvertArgument (PHEAD WORD ∗arg, int ∗type)
- int ExpandRat (PHEAD WORD ∗fun)
- int InvPoly (PHEAD WORD ∗inpoly, WORD maxpow, WORD sym)

Variables

- WORD divrem [4] = { DIVFUNCTION, REMFUNCTION, INVERSEFUNCTION, MULFUNCTION }
- char * TheErrorMessage [ ]

4.88.1 Detailed Description

A variety of routines: The ratio command for partial fractioning (rather old. Schoonschip inheritance) The sum routines.

Definition in file ratio.c.

4.88.2 Function Documentation

4.88.2.1 RatioFind()

```
WORD RatioFind (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 62 of file ratio.c.

4.88.2.2 RatioGen()

```
WORD RatioGen (
            PHEAD WORD * term,
            WORD * params,
            WORD num,
            WORD level )
```

Definition at line 170 of file ratio.c.

4.88.2.3 BinomGen()

```
WORD BinomGen (
            PHEAD WORD * term,
            WORD level,
            WORD ** tstops,
            WORD x1,
            WORD x2,
            WORD pow1,
            WORD pow2,
            WORD sign,
            UWORD * coef,
            WORD ncoef )
```

Definition at line 320 of file ratio.c.

4.88.2.4 DoSumF1()

```
WORD DoSumF1 (
            PHEAD WORD * term,
            WORD * params,
            WORD replac,
            WORD level )
```

Definition at line 395 of file ratio.c.

4.88.2.5 Glue()

```
WORD Glue (
            PHEAD WORD * term1,
            WORD * term2,
            WORD * sub,
            WORD insert )
```

Definition at line 461 of file ratio.c.

4.88.2.6 DoSumF2()

```
WORD DoSumF2 (
            PHEAD WORD * term,
            WORD * params,
            WORD replac,
            WORD level )
```

Definition at line 520 of file ratio.c.

4.88.2.7 GCDfunction()

```
int GCDfunction (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 609 of file ratio.c.

4.88.2.8 GCDfunction3()

```
WORD * GCDfunction3 (
            PHEAD WORD * in1,
            WORD * in2 )
```

Definition at line 1141 of file ratio.c.

4.88.2.9 PutExtraSymbols()

```
WORD * PutExtraSymbols (
            PHEAD WORD * in,
            WORD startebuf,
            int * actionflag )
```

Definition at line 1244 of file ratio.c.

4.88.2.10 TakeExtraSymbols()

```
WORD * TakeExtraSymbols (
            PHEAD WORD * in,
            WORD startebuf )
```

Definition at line 1273 of file ratio.c.

4.88.2.11 MultiplyWithTerm()

```
WORD * MultiplyWithTerm (
            PHEAD WORD * in,
            WORD * term,
            WORD par )
```

Definition at line 1312 of file ratio.c.

4.88.2.12 TakeContent()

```
WORD * TakeContent (
            PHEAD WORD * in,
            WORD * term )
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. Here the input is a sequence of terms in 'in' and the answer is a content-free sequence of terms. This sequence has been allocated by the Malloc1 routine in a call to EndSort, unless the expression was already content-free. In that case the input pointer is returned. The content is returned in term. This is supposed to be a separate allocation, made by TermMalloc in the calling routine.

Definition at line 1376 of file ratio.c.

References GetModInverses().

Here is the call graph for this function:

4.88.2.13 MergeSymbolLists()

```
int MergeSymbolLists (
            PHEAD WORD * old,
            WORD * extra,
            int par )
```

Definition at line 1808 of file ratio.c.

4.88.2.14 MergeDotproductLists()

```
int MergeDotproductLists (
            PHEAD WORD * old,
            WORD * extra,
            int par )
```

Definition at line 1927 of file ratio.c.

4.88.2.15 CreateExpression()

```
WORD * CreateExpression (
            PHEAD WORD nexp )
```

Definition at line 2020 of file ratio.c.

4.88.2.16 GCDterms()

```
int GCDterms (
            PHEAD WORD * term1,
            WORD * term2,
            WORD * termout )
```

Definition at line 2076 of file ratio.c.

4.88.2.17 ReadPolyRatFun()

```
int ReadPolyRatFun (
            PHEAD WORD * term )
```

Definition at line 2264 of file ratio.c.

4.88.2.18 FromPolyRatFun()

```
int FromPolyRatFun (
            PHEAD WORD * fun,
            WORD ** numout,
            WORD ** denout )
```

Definition at line 2383 of file ratio.c.

4.88.2.19 TakeSymbolContent()

```
WORD * TakeSymbolContent (
            PHEAD WORD * in,
            WORD * term )
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. We allow only symbols as this code is used for the polyratfun only. We have a special routine, because the generic TakeContent does too much work and speed is at a premium here. Input: in is the input expression as a sequence of terms. Output: term: the content return value: the contentfree expression. it is in new allocation, made by TermMalloc. (should be in a TermMalloc space?)

Definition at line 2434 of file ratio.c.

References GetModInverses().

Here is the call graph for this function:

4.88.2.20 GCDclean()

```
void GCDclean (
            PHEAD WORD * num,
            WORD * den )
```

Definition at line 2644 of file ratio.c.

4.88.2.21 PolyDiv()

```
WORD * PolyDiv (
            PHEAD WORD * a,
            WORD * b,
            char * text )
```

Definition at line 2741 of file ratio.c.

4.88.2.22 DIVfunction()

```
int DIVfunction (
            PHEAD WORD * term,
            WORD level,
            int par )
```

Definition at line 2788 of file ratio.c.

4.88.2.23 MULfunc()

```
WORD * MULfunc (
            PHEAD WORD * p1,
            WORD * p2 )
```

Definition at line 2941 of file ratio.c.

4.88.2.24 ConvertArgument()

```
WORD * ConvertArgument (
            PHEAD WORD * arg,
            int * type )
```

Definition at line 3002 of file ratio.c.

4.88.2.25 ExpandRat()

```
int ExpandRat (
            PHEAD WORD * fun )
```

Definition at line 3097 of file ratio.c.

4.88.2.26 InvPoly()

```
int InvPoly (
            PHEAD WORD * inpoly,
            WORD maxpow,
            WORD sym )
```

Definition at line 3576 of file ratio.c.

4.88.3 Variable Documentation

4.88.3.1 divrem

```
WORD divrem[4] = { DIVFUNCTION, REMFUNCTION, INVERSEFUNCTION, MULFUNCTION }
```

Definition at line 2786 of file ratio.c.

4.88.3.2 TheErrorMessage

```
char* TheErrorMessage[]
```

Initial value:

```
= {
     "PolyRatFun not of a type that FORM will expand: incorrect variable inside."
    ,"Division by zero in PolyRatFun encountered in ExpandRat."
    ,"Irregular code in PolyRatFun encountered in ExpandRat."
    ,"Called from ExpandRat."
    ,"WorkSpace overflow. Change parameter WorkSpace in setup file?"
    }
```

Definition at line 3089 of file ratio.c.

4.89 ratio.c

Go to the documentation of this file.

```c

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes : ratio.c
*/

#include "form3.h"

/*
    #] Includes :
    #[ Ratio :

    These are the special operations regarding simple polynomials.
    The first and most needed is the partial fractioning expansion.
    Ratio,x1,x2,x3

    The files belonging to the ratio command serve also as a good example
    of how to implement a new operation.

        #[ RatioFind :

        The routine that should locate the need for a ratio command.
        If located the corresponding symbols are removed and the
        operational parameters are loaded. A subexpression pointer
        is inserted and the code for success is returned.

        params points at the compiler output block defined in RatioComp.

*/

WORD RatioFind(PHEAD WORD *term, WORD *params)
{
    GETBIDENTITY
    WORD *t, *m, *r;
    WORD x1, x2, i;
    WORD *y1, *y2, n1 = 0, n2 = 0;
    x1 = params[3];
    x2 = params[4];
    m = t = term;
    m += *m;
    m -= ABS(m[-1]);
    t++;
    if ( t < m ) do {
        if ( *t == SYMBOL ) {
            y1 = 0;
            y2 = 0;
            r = t + t[1];
            m = t + 2;
            do {
                if ( *m == x1 ) { y1 = m; n1 = m[1]; }
                else if ( *m == x2 ) { y2 = m; n2 = m[1]; }
                m += 2;
            } while ( m < r );
            if ( !y1 || !y2 || ( n1 > 0 && n2 > 0 ) ) return(0);
            m -= 2;
            if ( y1 > y2 ) { r = y1; y1 = y2; y2 = r; }
            *y2 = *m; y2[1] = m[1];
```

```
            m -= 2;
            *y1 = *m; y1[1] = m[1];
            i = WORDDIF(m,t);
#if SUBEXPSIZE > 6
We have to revise the code for the second case.
#endif
            if ( i > 2 ) {      /* Subexpression fits exactly */
                t[1] = i;
                y1 = term+*term;
                y2 = y1+SUBEXPSIZE-4;
                r = m+4;
                while ( y1 > r ) *--y2 = *--y1;
                *m++ = SUBEXPRESSION;
                *m++ = SUBEXPSIZE;
                *m++ = -1;
                *m++ = 1;
                *m++ = DUMMYBUFFER;
                FILLSUB(m)
                *term += SUBEXPSIZE-4;
            }
            else {              /* All symbols are gone. Rest has to be moved */
                m -= 2;
                *m++ = SUBEXPRESSION;
                *m++ = SUBEXPSIZE;
                *m++ = -1;
                *m++ = 1;
                *m++ = DUMMYBUFFER;
                FILLSUB(m)
                t = term;
                t += *t;
                *term += SUBEXPSIZE-6;
                r = m + 6-SUBEXPSIZE;
                do { *m++ = *r++; } while ( r < t );
            }
            t = AT.TMout;           /* Load up the TM out array for the generator */
            *t++ = 7;
            *t++ = RATIO;
            *t++ = x1;
            *t++ = x2;
            *t++ = params[5];
            *t++ = n1;
            *t++ = n2;
            return(1);
        }
        t += t[1];
    } while ( t < m );
    return(0);
}

/*
        #] RatioFind :
        #[ RatioGen :

        The algoritm:
        x1^-n1*x2^n2    ==>  x2 --> x1 + x3
        x1^n1*x2^-n2    ==>  x1 --> x2 - x3
        x1^-n1*x2^-n2   ==>

           +sum(i=0,n1-1){(-1)^i*binom(n2-1+i,n2-1)
                    *x3^-(n2+i)*x1^-(n1-i)}
           +sum(i=0,n2-1){(-1)^(n1)*binom(n1-1+i,n1-1)
                    *x3^-(n1+i)*x2^-(n2-i)}

        Actually there is an amount of arbitrariness in the first two
        formulae and the replacement x2 -> x1 + x3 could be made 'by hand'.
        It is better to use the nontrivial 'minimal change' formula:

        x1^-n1*x2^n2:   if ( n1 >= n2 ) {
                           +sum(i=0,n2){x3^i*x1^-(n1-n2+i)*binom(n2,i)}
                        }
                        else {
                            sum(i=0,n2-n1){x2^(n2-n1-i)*x3^i*binom(n1-1+i,n1-1)}
                           +sum(i=0,n1-1){x3^(n2-i)*x1^-(n1-i)*binom(n2,i)}
                        }
        x1^n1*x2^-n2:   Same but x3 -> -x3.

        The contents of the AT.TMout/params array are:
        length,type,x1,x2,x3,n1,n2

*/

WORD RatioGen(PHEAD WORD *term, WORD *params, WORD num, WORD level)
{
    GETBIDENTITY
    WORD *t, *m;
    WORD *tstops[3];
    WORD n1, n2, i, j;
```

```
    WORD x1,x2,x3;
    UWORD *coef;
    WORD ncoef, sign = 0;
    coef = (UWORD *)AT.WorkPointer;
    t = term;
    tstops[2] = m = t + *t;
    m -= ABS(m[-1]);
    t++;
    do {
        if ( *t == SUBEXPRESSION && t[2] == num ) break;
        t += t[1];
    } while ( t < m );
    tstops[0] = t;
    tstops[1] = t + t[1];
/*
    Copying to termout will be from term to tstop1, then the induced part
    and finally from tstop2 to tstop3

    Now separate the various cases:

*/
    t = params + 2;
    x1 = *t++;
    x2 = *t++;
    x3 = *t++;
    n1 = *t++;
    n2 = *t++;
    if ( n1 > 0 ) {     /* Flip the variables and indicate -x3 */
        n2 = -n2;
        sign = 1;
        i = n1; n1 = n2; n2 = i;
        i = x1; x1 = x2; x2 = i;
        goto PosNeg;
    }
    else if ( n2 > 0 ) {
        n1 = -n1;
PosNeg:
        if ( n2 <= n1 ) {   /* x1 -> x2 + x3 */
            *coef = 1;
            ncoef = 1;
            AT.WorkPointer = (WORD *)(coef + 1);
            j = n2;
            for ( i = 0; i <= n2; i++ ) {
                if ( BinomGen(BHEAD term,level,tstops,x1,x3,n2-n1-i,i,sign&i
                ,coef,ncoef) ) goto RatioCall;
                if ( i < n2 ) {
                    if ( Product(coef,&ncoef,j) ) goto RatioCall;
                    if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
                    j--;
                    AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
                }
            }
            AT.WorkPointer = (WORD *)(coef);
            return(0);
        }
        else {
/*
            sum(i=0,n2-n1){x2^(n2-n1-i)*x3^i*binom(n1-1+i,n1-1)}
           +sum(i=0,n1-1){x3^(n2-i)*x1^-(n1-i)*binom(n2,i)}
*/
            *coef = 1;
            ncoef = 1;
            AT.WorkPointer = (WORD *)(coef + 1);
            j = n2 - n1;
            for ( i = 0; i <= j; i++ ) {
                if ( BinomGen(BHEAD term,level,tstops,x2,x3,n2-n1-i,i,sign&i
                ,coef,ncoef) ) goto RatioCall;
                if ( i < j ) {
                    if ( Product(coef,&ncoef,n1+i) ) goto RatioCall;
                    if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
                    AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
                }
            }
            *coef = 1;
            ncoef = 1;
            AT.WorkPointer = (WORD *)(coef + 1);
            j = n1-1;
            for ( i = 0; i <= j; i++ ) {
                if ( BinomGen(BHEAD term,level,tstops,x1,x3,i-n1,n2-i,sign&(n2-i)
                ,coef,ncoef) ) goto RatioCall;
                if ( i < j ) {
                    if ( Product(coef,&ncoef,n2-i) ) goto RatioCall;
                    if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
                    AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
                }
            }
            AT.WorkPointer = (WORD *)(coef);
```

```
            return(0);
        }
    }
    else {
        n2 = -n2;
        n1 = -n1;
/*
           +sum(i=0,n1-1){(-1)^i*binom(n2-1+i,n2-1)
                    *x3^-(n2+i)*x1^-(n1-i)}
           +sum(i=0,n2-1){(-1)^(n1)*binom(n1-1+i,n1-1)
                    *x3^-(n1+i)*x2^-(n2-i)}
*/
        *coef = 1;
        ncoef = 1;
        AT.WorkPointer = (WORD *)(coef + 1);
        j = n1-1;
        for ( i = 0; i <= j; i++ ) {
            if ( BinomGen(BHEAD term,level,tstops,x1,x3,i-n1,-n2-i,i&1
            ,coef,ncoef) ) goto RatioCall;
            if ( i < j ) {
                if ( Product(coef,&ncoef,n2+i) ) goto RatioCall;
                if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
                AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
            }
        }
        *coef = 1;
        ncoef = 1;
        AT.WorkPointer = (WORD *)(coef + 1);
        j = n2-1;
        for ( i = 0; i <= j; i++ ) {
            if ( BinomGen(BHEAD term,level,tstops,x2,x3,i-n2,-n1-i,n1&1
            ,coef,ncoef) ) goto RatioCall;
            if ( i < j ) {
                if ( Product(coef,&ncoef,n1+i) ) goto RatioCall;
                if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
                AT.WorkPointer = (WORD *)(coef + ABS(ncoef));
            }
        }
        AT.WorkPointer = (WORD *)(coef);
        return(0);
    }

RatioCall:
    MLOCK(ErrorMessageLock);
    MesCall("RatioGen");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] RatioGen :
        #[ BinomGen :

        Routine for the generation of terms in a binomialtype expansion.

*/

WORD BinomGen(PHEAD WORD *term, WORD level, WORD **tstops, WORD x1, WORD x2,
              WORD pow1, WORD pow2, WORD sign, UWORD *coef, WORD ncoef)
{
    GETBIDENTITY
    WORD *t, *r;
    WORD *termout;
    WORD k;
    termout = AT.WorkPointer;
    t = termout;
    r = term;
    do { *t++ = *r++; } while ( r < tstops[0] );
    *t++ = SYMBOL;
    if ( pow2 == 0 ) {
        if ( pow1 == 0 ) t--;
        else { *t++ = 4; *t++ = x1; *t++ = pow1; }
    }
    else if ( pow1 == 0 ) {
        *t++ = 4; *t++ = x2; *t++ = pow2;
    }
    else {
        *t++ = 6; *t++ = x1; *t++ = pow1; *t++ = x2; *t++ = pow2;
    }
    *t++ = LNUMBER;
    *t++ = ABS(ncoef) + 3;
    *t = ncoef;
    if ( sign ) *t = -*t;
    t++;
    ncoef = ABS(ncoef);
    for ( k = 0; k < ncoef; k++ ) *t++ = coef[k];
    r = tstops[1];
```

```
    do { *t++ = *r++; } while ( r < tstops[2] );
    *termout = WORDDIF(t,termout);
    AT.WorkPointer = t;
    if ( AT.WorkPointer > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    *AN.RepPoint = 1;
    AR.expchanged = 1;
    if ( Generator(BHEAD termout,level) ) {
        MLOCK(ErrorMessageLock);
        MesCall("BinomGen");
        MUNLOCK(ErrorMessageLock);
        SETERROR(-1)
    }
    AT.WorkPointer = termout;
    return(0);
}

/*
        #] BinomGen :
    #] Ratio :
    #[ Sum :
        #[ DoSumF1 :

        Routine expands a sum_ function.
        Its arguments are:
        The term in which the function occurs.
        The parameter list:
            length of parameter field
            function number (SUMNUM1)
            number of the symbol that is loop parameter
            min value
            max value
            increment
        the number of the subexpression to be removed
        the level in the generation tree.

        Note that the insertion of the loop parameter in the argument
        is done via the regular wildcard substitution mechanism.

*/

WORD DoSumF1(PHEAD WORD *term, WORD *params, WORD replac, WORD level)
{
    GETBIDENTITY
    WORD *termout, *t, extractbuff = AT.TMbuff;
    WORD isum, ival, iinc;
    LONG from;
    CBUF *C;
    ival = params[3];
    iinc = params[5];
    if ( ( iinc > 0 && params[4] >= ival )
      || ( iinc < 0 && params[4] <= ival ) ) {
        isum = (params[4] - ival)/iinc + 1;
    }
    else return(0);
    termout = AT.WorkPointer;
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    if ( AT.WorkPointer > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    t = term + 1;
    while ( *t != SUBEXPRESSION || t[2] != replac || t[4] != extractbuff )
                    t += t[1];
    C = cbuf+t[4];
    t += SUBEXPSIZE;
    if ( params[2] < 0 ) {
        while ( *t != INDTOIND || t[2] != -params[2] ) t += t[1];
        *t = INDTOIND;
    }
    else {
        while ( *t > SYMTOSUB || t[2] != params[2] ) t += t[1];
        *t = SYMTONUM;
    }
    do {
        t[3] = ival;
        from = C->rhs[replac] - C->Buffer;
        while ( C->Buffer[from] ) {
            if ( InsertTerm(BHEAD term,replac,extractbuff,C->Buffer+from,termout,0) < 0 ) goto
SumF1Call;
            AT.WorkPointer = termout + *termout;
```

```
            if ( Generator(BHEAD termout,level) < 0 ) goto SumF1Call;
            from += C->Buffer[from];
        }
        ival += iinc;
    } while ( --isum > 0 );
    AT.WorkPointer = termout;
    return(0);
SumF1Call:
    MLOCK(ErrorMessageLock);
    MesCall("DoSumF1");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] DoSumF1 :
        #[ Glue :

        Routine multiplies two terms. The second term is subject
        to the wildcard substitutions in sub.
        Output in the first term. This routine is a variation on
        the routine InsertTerm.

*/

WORD Glue(PHEAD WORD *term1, WORD *term2, WORD *sub, WORD insert)
{
    GETBIDENTITY
    UWORD *coef;
    WORD ncoef, *t, *t1, *t2, i, nc2, nc3, old, newer;
    coef = (UWORD *)(TermMalloc("Glue"));
    t = term1;
    t += *t;
    i = t[-1];
    t -= ABS(i);
    old = WORDDIF(t,term1);
    ncoef = REDLENG(i);
    if ( i < 0 ) i = -i;
    i--;
    t1 = t;
    t2 = (WORD *)coef;
    while ( --i >= 0 ) *t2++ = *t1++;
    i = *--t;
    nc2 = WildFill(BHEAD t,term2,sub);
    *t = i;
    t += nc2;
    nc2 = t[-1];
    t -= ABS(nc2);
    newer = WORDDIF(t,term1);
    if ( MulRat(BHEAD (UWORD *)t,REDLENG(nc2),coef,ncoef,(UWORD *)t,&nc3) ) {
        MLOCK(ErrorMessageLock);
        MesCall("Glue");
        MUNLOCK(ErrorMessageLock);
        TermFree(coef,"Glue");
        SETERROR(-1)
    }
    i = (ABS(nc3))«1;
    t += i++;
    *t++ = (nc3 >= 0)?i:-i;
    *term1 = WORDDIF(t,term1);
/*
    Switch the new piece with the old tail, so that noncommuting
    variables get into their proper spot.
*/
    i = old - insert;
    t1 = t;
    t2 = term1+insert;
    NCOPY(t1,t2,i);
    i = newer - old;
    t1 = term1+insert;
    t2 = term1+old;
    NCOPY(t1,t2,i);
    t2 = t;
    i = old - insert;
    NCOPY(t1,t2,i);
    TermFree(coef,"Glue");
    return(0);
}

/*
        #] Glue :
        #[ DoSumF2 :
*/

WORD DoSumF2(PHEAD WORD *term, WORD *params, WORD replac, WORD level)
{
    GETBIDENTITY
```

```
    WORD *termout, *t, *from, *sub, *to, extractbuff = AT.TMbuff;
    WORD isum, ival, iinc, insert, i;
    CBUF *C;
    ival = params[3];
    iinc = params[5];
    if ( ( iinc > 0 && params[4] >= ival )
      || ( iinc < 0 && params[4] <= ival ) ) {
        isum = (params[4] - ival)/iinc + 1;
    }
    else return(0);
    termout = AT.WorkPointer;
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    if ( AT.WorkPointer > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    t = term + 1;
    while ( *t != SUBEXPRESSION || t[2] != replac || t[4] != extractbuff ) t += t[1];
    insert = WORDDIF(t,term);

    from = term;
    to = termout;
    while ( from < t ) *to++ = *from++;
    from += t[1];
    sub = term + *term;
    while ( from < sub ) *to++ = *from++;
    *termout -= t[1];

    sub = t;
    C = cbuf+t[4];
    t += SUBEXPSIZE;
    if ( params[2] < 0 ) {
        while ( *t != INDTOIND || t[2] != -params[2] ) t += t[1];
        *t = INDTOIND;
    }
    else {
        while ( *t > SYMTOSUB || t[2] != params[2] ) t += t[1];
        *t = SYMTONUM;
    }
    t[3] = ival;
    for(;;) {
        AT.WorkPointer = termout + *termout;
        to = AT.WorkPointer;
        if ( ( to + *termout ) > AT.WorkTop ) {
            MLOCK(ErrorMessageLock);
            MesWork();
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        from = termout;
        i = *termout;
        NCOPY(to,from,i);
        from = AT.WorkPointer;
        AT.WorkPointer = to;
        if ( Generator(BHEAD from,level) < 0 ) goto SumF2Call;
        if ( --isum <= 0 ) break;
        ival += iinc;
        t[3] = ival;
        if ( Glue(BHEAD termout,C->rhs[replac],sub,insert) < 0 ) goto SumF2Call;
    }
    AT.WorkPointer = termout;
    return(0);
SumF2Call:
    MLOCK(ErrorMessageLock);
    MesCall("DoSumF2");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] DoSumF2 :
    #] Sum :
    #[ GCDfunction :
        #[ GCDfunction :
*/

typedef struct {
    WORD *buffer;
    DOLLARS dollar;
    LONG size;
    int type;
    int dummy;
} ARGBUFFER;

int GCDfunction(PHEAD WORD *term,WORD level)
```

```
{
    GETBIDENTITY
    WORD *t, *tstop, *tf, *termout, *tin, *tout, *m, *mnext, *mstop, *mm;
    int todo, i, ii, j, istart, sign = 1, action = 0;
    WORD firstshort = 0, firstvalue = 0, gcdisone = 0, mlength, tlength, newlength;
    WORD totargs = 0, numargs, argsdone = 0, *mh, oldval1, *g, *gcdout = 0;
    WORD *arg1, *arg2;
    UWORD x1,x2,x3;
    LONG args;
#if ( FUNHEAD > 4 )
    WORD sh[FUNHEAD+5];
#else
    WORD sh[9];
#endif
    DOLLARS d;
    ARGBUFFER *abuf = 0, ab;
/*
    #[ Find Function. Count arguments :

    First find the proper function
*/
    t = term + *term; tlength = t[-1];
    tstop = t - ABS(tlength);
    t = term + 1;
    while ( t < tstop ) {
        if ( *t != GCDFUNCTION ) { t += t[1]; continue; }
        todo = 1; totargs = 0;
        tf = t + FUNHEAD;
        while ( tf < t + t[1] ) {
            totargs++;
            if ( *tf > 0 && tf[1] != 0 ) todo = 0;
            NEXTARG(tf);
        }
        if ( todo ) break;
        t += t[1];
    }
    if ( t >= tstop ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Internal error. Indicated gcd_ function not encountered.");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    WantAddPointers(totargs);
    args = AT.pWorkPointer; AT.pWorkPointer += totargs;
/*
    #] Find Function. Count arguments :
    #[ Do short arguments :

    The function we need, in agreement with TestSub, is now in t
    Make first a compilation of the short arguments (except $-s and expressions)
    to see whether we need to do much work.
    This means that after this scan we can ignore all short arguments with
    the exception of unevaluated $-s and expressions.
*/
    numargs = 0;
    firstshort = 0;
    tf = t + FUNHEAD;
    while ( tf < t + t[1] ) {
        if ( *tf == -SNUMBER && tf[1] == 0 ) { NEXTARG(tf); continue; }
        if ( *tf > 0 || *tf == -DOLLAREXPRESSION || *tf == -EXPRESSION ) {
            AT.pWorkSpace[args+numargs++] = tf;
            NEXTARG(tf); continue;
        }
        if ( firstshort == 0 ) {
            firstshort = *tf;
            if ( *tf <= -FUNCTION ) { firstvalue = -(*tf); }
            else                    { firstvalue = tf[1]; }
            NEXTARG(tf);
            argsdone++;
            continue;
        }
        else if ( *tf != firstshort ) {
            if ( *tf != -INDEX && *tf != -VECTOR && *tf != -MINVECTOR ) {
                argsdone++; gcdisone = 1; break;
            }
            if ( firstshort != -INDEX && firstshort != -VECTOR && firstshort != -MINVECTOR ) {
                argsdone++; gcdisone = 1; break;
            }
            if ( tf[1] != firstvalue ) {
                argsdone++; gcdisone = 1; break;
            }
            if ( *t == -MINVECTOR ) { firstshort = -VECTOR; }
            if ( firstshort == -MINVECTOR ) { firstshort = -VECTOR; }
        }
        else if ( *tf > -FUNCTION && *tf != -SNUMBER && tf[1] != firstvalue ) {
            argsdone++; gcdisone = 1; break;
        }
```

```
        if ( *tf == -SNUMBER && firstvalue != tf[1] ) {
/*
            make a new firstvalue which is gcd_(firstvalue,tf[1])
*/
            if ( firstvalue == 1 || tf[1] == 1 ) { gcdisone = 1; break; }
            if ( firstvalue < 0 && tf[1] < 0 ) {
                x1 = -firstvalue; x2 = -tf[1]; sign = -1;
            }
            else {
                x1 = ABS(firstvalue); x2 = ABS(tf[1]); sign = 1;
            }
            while ( ( x3 = x1%x2 ) != 0 ) { x1 = x2; x2 = x3; }
            firstvalue = ((WORD)x2)*sign;
            argsdone++;
            if ( firstvalue == 1 ) { gcdisone = 1; break; }
        }
        NEXTARG(tf);
    }
    termout = AT.WorkPointer;
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    if ( AT.WorkPointer > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
/*
    #] Do short arguments :
    #[ Do trivial GCD :

    Copy head
*/
    i = t - term; tin = term; tout = termout;
    NCOPY(tout,tin,i);
    if ( gcdisone || ( firstshort == -SNUMBER && firstvalue == 1 ) ) {
        sign = 1;
gcdone:
        tin += t[1]; tstop = term + *term;
        while ( tin < tstop ) *tout++ = *tin++;
        *termout = tout - termout;
        if ( sign < 0 ) tout[-1] = -tout[-1];
        AT.WorkPointer = tout;
        if ( argsdone && Generator(BHEAD termout,level) < 0 ) goto CalledFrom;
        AT.WorkPointer = termout;
        AT.pWorkPointer = args;
        return(0);
    }
/*
    #] Do trivial GCD :
    #[ Do short argument GCD :
*/
    if ( numargs == 0 ) {   /* basically we are done */
doshort:
        sign = 1;
        if ( firstshort == 0 ) goto gcdone;
        if ( firstshort == -SNUMBER ) {
            *tout++ = SNUMBER; *tout++ = 4; *tout++ = firstvalue; *tout++ = 1;
            goto gcdone;
        }
        else if ( firstshort == -SYMBOL ) {
            *tout++ = SYMBOL; *tout++ = 4; *tout++ = firstvalue; *tout++ = 1;
            goto gcdone;
        }
        else if ( firstshort == -VECTOR || firstshort == -INDEX ) {
            *tout++ = INDEX; *tout++ = 3; *tout++ = firstvalue; goto gcdone;
        }
        else if ( firstshort == -MINVECTOR ) {
            sign = -1;
            *tout++ = INDEX; *tout++ = 3; *tout++ = firstvalue; goto gcdone;
        }
        else if ( firstshort <= -FUNCTION ) {
            *tout++ = firstvalue; *tout++ = FUNHEAD; FILLFUN(tout);
            goto gcdone;
        }
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("Internal error. Illegal short argument in GCDfunction.");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
    }
/*
    #] Do short argument GCD :
    #[ Convert short argument :

    Now we allocate space for the arguments in general notation.
    First the special one if there were short arguments
```

```
*/
    if ( firstshort ) {
      switch ( firstshort ) {
        case -SNUMBER:
            sh[0] = 4; sh[1] = ABS(firstvalue); sh[2] = 1;
            if ( firstvalue < 0 ) sh[3] = -3;
            else sh[3] = 3;
            sh[4] = 0;
            break;
        case -MINVECTOR:
        case -VECTOR:
        case -INDEX:
            sh[0] = 8; sh[1] = INDEX; sh[2] = 3; sh[3] = firstvalue;
            sh[4] = 1; sh[5] = 1;
            if ( firstshort == -MINVECTOR ) sh[6] = -3;
            else sh[6] = 3;
            sh[7] = 0;
            break;
        case -SYMBOL:
            sh[0] = 8; sh[1] = SYMBOL; sh[2] = 4; sh[3] = firstvalue; sh[4] = 1;
            sh[5] = 1; sh[6] = 1; sh[7] = 3; sh[8] = 0;
            break;
        default:
            sh[0] = FUNHEAD+4; sh[1] = firstshort; sh[2] = FUNHEAD;
            for ( i = 2; i < FUNHEAD; i++ ) sh[i+1] = 0;
            sh[FUNHEAD+1] = 1; sh[FUNHEAD+2] = 1; sh[FUNHEAD+3] = 3; sh[FUNHEAD+4] = 0;
        break;
      }
    }
/*
    #] Convert short argument :
    #[ Sort arguments :

    Now we should sort the arguments in a way that the dollars and the
    expressions come last. That way we may never need them.
*/
    for ( i = 1; i < numargs; i++ ) {
        for ( ii = i; ii > 0; ii-- ) {
            arg1 = AT.pWorkSpace[args+ii];
            arg2 = AT.pWorkSpace[args+ii-1];
            if ( *arg1 < 0 ) {
                if ( *arg2 < 0 ) {
                    if ( *arg1 == -EXPRESSION ) break;
                    if ( *arg2 == -DOLLAREXPRESSION ) break;
                    AT.pWorkSpace[args+ii] = arg2;
                    AT.pWorkSpace[args+ii-1] = arg1;
                }
                else break;
            }
            else if ( *arg2 < 0 ) {
                AT.pWorkSpace[args+ii] = arg2;
                AT.pWorkSpace[args+ii-1] = arg1;
            }
            else {
                if ( *arg1 > *arg2 ) {
                    AT.pWorkSpace[args+ii] = arg2;
                    AT.pWorkSpace[args+ii-1] = arg1;
                }
                else break;
            }
        }
    }
/*
    #] Sort arguments :
    #[ There is a single term argument :
*/
    if ( firstshort ) {
        mh = sh; istart = 0;
oneterm:;
        for ( i = istart; i < numargs; i++ ) {
            arg1 = AT.pWorkSpace[args+i];
            if ( *arg1 > 0 ) {
                oldval1 = arg1[*arg1]; arg1[*arg1] = 0;
                m = arg1+ARGHEAD;
                while ( *m ) {
                    GCDterms(BHEAD mh,m,mh); m += *m;
                    if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
                        gcdisone = 1; sign = 1; arg1[*arg1] = oldval1; goto gcdone;
                    }
                }
                arg1[*arg1] = oldval1;
            }
            else if ( *arg1 == -DOLLAREXPRESSION ) {
                if ( ( d = DolToTerms(BHEAD arg1[1]) ) != 0 ) {
                    m = d->where;
                    while ( *m ) {
                        GCDterms(BHEAD mh,m,mh); m += *m;
```

```
                        argsdone++;
                        if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
                            gcdisone = 1; sign = 1;
                            if ( d->factors ) M_free(d->factors,"Dollar factors");
                            M_free(d,"Copy of dollar variable"); goto gcdone;
                        }
                    }
                    if ( d->factors ) M_free(d->factors,"Dollar factors");
                    M_free(d,"Copy of dollar variable");
                }
            }
            else {
                mm = CreateExpression(BHEAD arg1[1]);
                m = mm;
                while ( *m ) {
                    GCDterms(BHEAD mh,m,mh); m += *m;
                    argsdone++;
                    if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
                        gcdisone = 1; sign = 1; M_free(mm,"CreateExpression"); goto gcdone;
                    }
                }
                M_free(mm,"CreateExpression");
            }
        }
        if ( firstshort ) {
            if ( mh[0] == 4 ) {
                firstshort = -SNUMBER; firstvalue = mh[1] * (mh[3]/3);
            }
            else if ( mh[1] == SYMBOL ) {
                firstshort = -SYMBOL; firstvalue = mh[3];
            }
            else if ( mh[1] == INDEX ) {
                firstshort = -INDEX; firstvalue = mh[3];
                if ( mh[6] == -3 ) firstshort = -MINVECTOR;
            }
            else if ( mh[1] >= FUNCTION ) {
                firstshort = -mh[1]; firstvalue = mh[1];
            }
            goto doshort;
        }
        else {
/*
            We have a GCD that is only a single term.
            Paste it in and combine the coefficients.
*/
            mh[mh[0]] = 0;
            mm = mh;
            ii = 0;
            goto multiterms;
        }
    }
/*
    Now we have only regular arguments.
    But some have not yet been expanded.
    Check whether there are proper long arguments and if so if there is
    one with just a single term
*/
    for ( i = 0; i < numargs; i++ ) {
        arg1 = AT.pWorkSpace[args+i];
        if ( *arg1 > 0 && arg1[ARGHEAD]+ARGHEAD == *arg1 ) {
/*
            We have an argument with a single term
*/
            if ( i != 0 ) {
                arg2 = AT.pWorkSpace[args];
                AT.pWorkSpace[args] = arg1;
                AT.pWorkSpace[args+1] = arg2;
            }
            m = mh = AT.WorkPointer;
            mm = arg1+ARGHEAD; i = *mm;
            NCOPY(m,mm,i);
            AT.WorkPointer = m;
            istart = 1;
            argsdone++;
            goto oneterm;
        }
    }
/*
    #] There is a single term argument :
    #[ Expand $ and expr :

    We have: 1: regular multiterm arguments
             2: dollars
             3: expressions.
    The sum of them is numargs. Their addresses are in args. The problem is
    that expansion will lead to allocations that we have to return and all
    these allocations are different in nature.
```

```
*/
    action = 1;
    abuf = (ARGBUFFER *)Malloc1(numargs*sizeof(ARGBUFFER),"argbuffer");
    for ( i = 0; i < numargs; i++ ) {
        arg1 = AT.pWorkSpace[args+i];
        if ( *arg1 > 0 ) {
            m = (WORD *)Malloc1(*arg1*sizeof(WORD),"argbuffer type 0");
            abuf[i].buffer = m;
            abuf[i].type = 0;
            mm = arg1+ARGHEAD;
            j = *arg1-ARGHEAD;
            abuf[i].size = j;
            if ( j ) argsdone++;
            NCOPY(m,mm,j);
            *m = 0;
        }
        else if ( *arg1 == -DOLLAREXPRESSION ) {
            d = DolToTerms(BHEAD arg1[1]);
            abuf[i].buffer = d->where;
            abuf[i].type = 1;
            abuf[i].dollar = d;
            m = abuf[i].buffer;
            if ( *m ) argsdone++;
            while ( *m ) m+= *m;
            abuf[i].size = m-abuf[i].buffer;
        }
        else if ( *arg1 == -EXPRESSION ) {
            abuf[i].buffer = CreateExpression(BHEAD arg1[1]);
            abuf[i].type = 2;
            m = abuf[i].buffer;
            if ( *m ) argsdone++;
            while ( *m ) m+= *m;
            abuf[i].size = m-abuf[i].buffer;
        }
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("What argument is this?");
            MUNLOCK(ErrorMessageLock);
            goto CalledFrom;
        }
    }
    for ( i = 0; i < numargs; i++ ) {
        arg1 = abuf[i].buffer;
        if ( *arg1 == 0 ) {}
        else if ( arg1[*arg1] == 0 ) {
/*
            After expansion there is an argument with a single term
*/
            ab = abuf[i]; abuf[i] = abuf[0]; abuf[0] = ab;
            mh = abuf[0].buffer;
            for ( j = 1; j < numargs; j++ ) {
                m = abuf[j].buffer;
                while ( *m ) {
                    GCDterms(BHEAD mh,m,mh); m += *m;
                    argsdone++;
                    if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
                        gcdisone = 1; sign = 1; break;
                    }
                }
                if ( *m ) break;
            }
            mm = mh + *mh; if ( mm[-1] < 0 ) { sign = -1; mm[-1] = -mm[-1]; }
            mstop = mm - mm[-1]; m = mh+1; mlength = mm[-1];
            while ( tin < t ) *tout++ = *tin++;
            while ( m < mstop ) *tout++ = *m++;
            tin += tin[1];
            while ( tin < tstop ) *tout++ = *tin++;
            tlength = REDLENG(tlength);
            mlength = REDLENG(mlength);
            if ( MulRat(BHEAD (UWORD *)tstop,tlength,(UWORD *)mstop,mlength,
                    (UWORD *)tout,&newlength) < 0 ) goto CalledFrom;
            mlength = INCLENG(newlength);
            tout += ABS(mlength);
            tout[-1] = mlength*sign;
            *termout = tout - termout;
            AT.WorkPointer = tout;
            if ( argsdone && Generator(BHEAD termout,level) < 0 ) goto CalledFrom;
            goto cleanup;
        }
    }
/*
    There are only arguments with more than one term.
    We order them by size to make the computations as easy as possible.
*/
    for ( i = 1; i < numargs; i++ ) {
        for ( ii = i; ii > 0; ii-- ) {
            if ( abuf[ii-1].size <= abuf[ii].size ) break;
```

```
            ab = abuf[ii-1]; abuf[ii-1] = abuf[ii]; abuf[ii] = ab;
        }
    }
/*
    #] Expand $ and expr :
    #[ Multiterm subexpressions :
*/
    ii = 0;
    gcdout = abuf[ii].buffer;
    for ( i = 0; i < numargs; i++ ) {
        if ( abuf[i].buffer[0] ) { gcdout = abuf[i].buffer; ii = i; i++; argsdone++; break; }
    }
    for ( ; i < numargs; i++ ) {
      if ( abuf[i].buffer[0] ) {
        g = GCDfunction3(BHEAD gcdout,abuf[i].buffer);
        argsdone++;
        if ( gcdout != abuf[ii].buffer ) M_free(gcdout,"gcdout");
        gcdout = g;
        if ( gcdout[*gcdout] == 0 && gcdout[0] == 4 && gcdout[1] == 1
        && gcdout[2] == 1 && gcdout[3] == 3 ) break;
      }
    }
    mm = gcdout;
multiterms:;
    tlength = REDLENG(tlength);
    while ( *mm ) {
        tin = term; tout = termout; while ( tin < t ) *tout++ = *tin++;
        tin += t[1];
        mnext = mm + *mm; mlength = mnext[-1]; mstop = mnext - ABS(mlength);
        mm++;
        while ( mm < mstop ) *tout++ = *mm++;
        while ( tin < tstop ) *tout++ = *tin++;
        mlength = REDLENG(mlength);
        if ( MulRat(BHEAD (UWORD *)tstop,tlength,(UWORD *)mm,mlength,
                (UWORD *)tout,&newlength) < 0 ) goto CalledFrom;
        mlength = INCLENG(newlength);
        tout += ABS(mlength);
        tout[-1] = mlength;
        *termout = tout - termout;
        AT.WorkPointer = tout;
        if ( argsdone && Generator(BHEAD termout,level) < 0 ) goto CalledFrom;
        mm = mnext; /* next term */
    }
    if ( action && ( gcdout != abuf[ii].buffer ) ) M_free(gcdout,"gcdout");
/*
    #] Multiterm subexpressions :
    #[ Cleanup :
*/
cleanup:;
    if ( action ) {
        for ( i = 0; i < numargs; i++ ) {
            if ( abuf[i].type == 0 ) { M_free(abuf[i].buffer,"argbuffer type 0"); }
            else if ( abuf[i].type == 1 ) {
                d = abuf[i].dollar;
                if ( d->factors ) M_free(d->factors,"Dollar factors");
                M_free(d,"Copy of dollar variable");
            }
            else if ( abuf[i].type == 2 ) { M_free(abuf[i].buffer,"CreateExpression"); }
        }
        M_free(abuf,"argbuffer");
    }
/*
    #] Cleanup :
*/
    AT.pWorkPointer = args;
    AT.WorkPointer = termout;
    return(0);

CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("GCDfunction");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
    return(-1);
}

/*
        #] GCDfunction :
        #[ GCDfunction3 :

    Finds the GCD of the two arguments which are buffers with terms.
    In principle the first buffer can have only one term.

    If both buffers have more than one term, we need to replace all
    non-symbolic objects by generated symbols and substitute that back
    afterwards. The rest we leave to the powerful routines.
    Philosophical problem: What do we do with GCD_(x/z+y,x+y*z) ?
```

```

    Method:
    If we have only negative powers of z and no positive powers we let
    the EXTRASYMBOLS do their job. When mixed, multiply the arguments with
    the negative powers with enough powers of z to eliminate the negative powers.
    The DENOMINATOR function is always eliminated with the mechanism as we
    cannot tell whether there are positive powers of its contents.
*/

WORD *GCDfunction3(PHEAD WORD *in1, WORD *in2)
{
    GETBIDENTITY
    WORD oldsorttype = AR.SortType, *ow = AT.WorkPointer;;
    WORD *t, *tt, *gcdout, *term1, *term2, *confree1, *confree2, *gcdout1, *proper1, *proper2;
    int i, actionflag1, actionflag2;
    WORD startebuf = cbuf[AT.ebufnum].numrhs;
    WORD tryterm1, tryterm2;
    if ( in2[*in2] == 0 ) { t = in1; in1 = in2; in2 = t; }
    if ( in1[*in1] == 0 ) { /* First input with only one term */
        gcdout = (WORD *)Malloc1((*in1+1)*sizeof(WORD),"gcdout");
        i = *in1; t = gcdout; tt = in1; NCOPY(t,tt,i); *t = 0;
        t = in2;
        while ( *t ) {
            GCDterms(BHEAD gcdout,t,gcdout);
            if ( gcdout[0] == 4 && gcdout[1] == 1
             && gcdout[2] == 1 && gcdout[3] == 3 ) break;
            t += *t;
        }
        AT.WorkPointer = ow;
        return(gcdout);
    }
/*
    We need to take out the content from the two expressions
    and determine their GCD. This plays with the negative powers!
*/
    AR.SortType = SORTHIGHFIRST;
    term1 = TermMalloc("GCDfunction3-a");
    term2 = TermMalloc("GCDfunction3-b");

    confree1 = TakeContent(BHEAD in1,term1);
    tryterm1 = AN.tryterm; AN.tryterm = 0;
    confree2 = TakeContent(BHEAD in2,term2);
    tryterm2 = AN.tryterm; AN.tryterm = 0;
/*
    confree1 = TakeSymbolContent(BHEAD in1,term1);
    confree2 = TakeSymbolContent(BHEAD in2,term2);
*/
    GCDterms(BHEAD term1,term2,term1);
    TermFree(term2,"GCDfunction3-b");
/*
    Now we have to replace all non-symbols and symbols to a negative power
    by extra symbols.
*/
    if ( ( proper1 = PutExtraSymbols(BHEAD confree1,startebuf,&actionflag1) ) == 0 ) goto CalledFrom;
    if ( confree1 != in1 ) {
        if ( tryterm1 ) { TermFree(confree1,"TakeContent"); }
        else { M_free(confree1,"TakeContent"); }
    }
/*
    TermFree(confree1,"TakeSymbolContent");
*/
    if ( ( proper2 = PutExtraSymbols(BHEAD confree2,startebuf,&actionflag2) ) == 0 ) goto CalledFrom;
    if ( confree2 != in2 ) {
        if ( tryterm2 ) { TermFree(confree2,"TakeContent"); }
        else { M_free(confree2,"TakeContent"); }
    }
/*
    TermFree(confree2,"TakeSymbolContent");
*/
/*
    And now the real work:
*/
    gcdout1 = poly_gcd(BHEAD proper1,proper2,0);
    M_free(proper1,"PutExtraSymbols");
    M_free(proper2,"PutExtraSymbols");

    AR.SortType = oldsorttype;
    if ( actionflag1 || actionflag2 ) {
        if ( ( gcdout = TakeExtraSymbols(BHEAD gcdout1,startebuf) ) == 0 ) goto CalledFrom;
        M_free(gcdout1,"gcdout");
    }
    else {
        gcdout = gcdout1;
    }

    cbuf[AT.ebufnum].numrhs = startebuf;
/*
```

```
    Now multiply gcdout by term1
*/
    if ( term1[0] != 4 || term1[3] != 3 || term1[1] != 1 || term1[2] != 1 ) {
        AN.tryterm = -1;
        if ( ( gcdout1 = MultiplyWithTerm(BHEAD gcdout,term1,2) ) == 0 ) goto CalledFrom;
        AN.tryterm = 0;
        M_free(gcdout,"gcdout");
        gcdout = gcdout1;
    }
    TermFree(term1,"GCDfunction3-a");
    AT.WorkPointer = ow;
    return(gcdout);
CalledFrom:
    AN.tryterm = 0;
    MLOCK(ErrorMessageLock);
    MesCall("GCDfunction3");
    MUNLOCK(ErrorMessageLock);
    return(0);
}

/*
        #] GCDfunction3 :
        #[ PutExtraSymbols :
*/

WORD *PutExtraSymbols(PHEAD WORD *in,WORD startebuf,int *actionflag)
{
    WORD *termout = AT.WorkPointer;
    int action;
    *actionflag = 0;
    NewSort(BHEAD0);
    while ( *in ) {
        if ( ( action = LocalConvertToPoly(BHEAD in,termout,startebuf,0) ) < 0 ) {
            LowerSortLevel();
            goto CalledFrom;
        }
        if ( action > 0 ) *actionflag = 1;
        StoreTerm(BHEAD termout);
        in += *in;
    }
    if ( EndSort(BHEAD (WORD *)((VOID *)(&termout)),2) < 0 ) goto CalledFrom;
    return(termout);
CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("PutExtraSymbols");
    MUNLOCK(ErrorMessageLock);
    return(0);
}

/*
        #] PutExtraSymbols :
        #[ TakeExtraSymbols :
*/

WORD *TakeExtraSymbols(PHEAD WORD *in,WORD startebuf)
{
    CBUF *C = cbuf+AC.cbufnum;
    CBUF *CC = cbuf+AT.ebufnum;
    WORD *oldworkpointer = AT.WorkPointer, *termout;

    termout = AT.WorkPointer;
    NewSort(BHEAD0);
    while ( *in ) {
        if ( ConvertFromPoly(BHEAD in,termout,numxsymbol,CC->numrhs-startebuf+numxsymbol,startebuf-numxsymbol,1) <= 0 ) {
            LowerSortLevel();
            goto CalledFrom;
        }
        in += *in;
        AT.WorkPointer = termout + *termout;
/*
        ConvertFromPoly leaves terms with subexpressions. Hence:
*/
        if ( Generator(BHEAD termout,C->numlhs) ) {
            LowerSortLevel();
            goto CalledFrom;
        }
    }
    AT.WorkPointer = oldworkpointer;
    if ( EndSort(BHEAD (WORD *)((VOID *)(&termout)),2) < 0 ) goto CalledFrom;
    return(termout);

CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("TakeExtraSymbols");
    MUNLOCK(ErrorMessageLock);
    return(0);
```

```
}

/*
        #] TakeExtraSymbols :
        #[ MultiplyWithTerm :
*/

WORD *MultiplyWithTerm(PHEAD WORD *in, WORD *term, WORD par)
{
    WORD *termout, *t, *tt, *tstop, *ttstop;
    WORD length, length1, length2;
    WORD oldsorttype = AR.SortType;
    void *oldcompareroutine = AR.CompareRoutine;
    AR.CompareRoutine = (void *)&CompareSymbols;

    if ( par == 0 || par == 2 ) AR.SortType = SORTHIGHFIRST;
    else            AR.SortType = SORTLOWFIRST;
    termout = AT.WorkPointer;
    NewSort(BHEAD0);
    while ( *in ) {
        tt = termout + 1;
        tstop = in + *in; tstop -= ABS(tstop[-1]); t = in + 1;
        while ( t < tstop ) *tt++ = *t++;
        ttstop = term + *term; ttstop -= ABS(ttstop[-1]); t = term + 1;
        while ( t < ttstop ) *tt++ = *t++;
        length1 = REDLENG(in[*in-1]); length2 = REDLENG(term[*term-1]);
        if ( MulRat(BHEAD (UWORD *)tstop,length1,
                (UWORD *)ttstop,length2,(UWORD *)tt,&length) ) goto CalledFrom;
        length = INCLENG(length);
        tt += ABS(length); tt[-1] = length;
        *termout = tt - termout;
        SymbolNormalize(termout);
        StoreTerm(BHEAD termout);
        in += *in;
    }
    if ( par == 2 ) {
/*      if ( AN.tryterm == 0 ) AN.tryterm = 1; */
        AN.tryterm = 0; /* For now */
        if ( EndSort(BHEAD (WORD *)((VOID *)(&termout)),2) < 0 ) goto CalledFrom;
    }
    else {
        if ( EndSort(BHEAD termout,1) < 0 ) goto CalledFrom;
    }

    AR.CompareRoutine = oldcompareroutine;

    AR.SortType = oldsorttype;
    return(termout);

CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("MultiplyWithTerm");
    MUNLOCK(ErrorMessageLock);
    return(0);
}

/*
        #] MultiplyWithTerm :
        #[ TakeContent :
*/
WORD *TakeContent(PHEAD WORD *in, WORD *term)
{
    GETBIDENTITY
    WORD *t, *tstop, *tcom, *tout, *tstore, *r, *rstop, *m, *mm, *w, *ww, *wterm;
    WORD *tnext, *tt, *tterm, code[2];
    WORD *inp, a, *den;
    int i, j, k, action = 0, sign;
    UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMbuffer2, *ap;
    WORD GCDlen, GCDlen2, LCMlen, LCMlen2, length, redlength, len1, len2;
    tout = tstore = term+1;
/*
    #[ INDEX :
*/
    t = in;
    tnext = t + *t;
    tstop = tnext-ABS(tnext[-1]);
    t++;
    while ( t < tstop ) {
        if ( *t == INDEX ) {
            i = t[1]; NCOPY(tout,t,i); break;
        }
        else t += t[1];
    }
    if ( tout > tstore ) { /* There are indices in the first term */
        t = tnext;
        while ( *t ) {
            tnext = t + *t;
```

```
            rstop = tnext - ABS(tnext[-1]);
            r = t+1;
            if ( r == rstop ) goto noindices;
            while ( r < rstop ) {
                if ( *r != INDEX ) { r += r[1]; continue; }
                m = tstore+2;
                while ( m < tout ) {
                    for ( i = 2; i < r[1]; i++ ) {
                        if ( *m == r[i] ) break;
                        if ( *m > r[i] ) continue;
                        mm = m+1;
                        while ( mm < tout ) { mm[-1] = mm[0]; mm++; }
                        tout--; tstore[1]--; m--;
                        break;
                    }
                    m++;
                }
            }
            if ( r >= rstop || tout <= tstore+2 ) {
                tout = tstore; break;
            }
        }
        if ( tout > tstore+2 ) { /* Now we have to take out what is in tstore */
            t = in; w = in;
            while ( *t ) {
                wterm = w;
                tnext = t + *t; t++; w++;
                while ( *t != INDEX ) { i = t[1]; NCOPY(w,t,i); }
                tt = t + t[1]; t += 2; r = tstore+2; ww = w; *w++ = INDEX; w++;
                while ( r < tout && t < tt ) {
                    if ( *r > *t ) { *w++ = *t++; }
                    else if ( *r == *t ) { r++; t++; }
                    else goto CalledFrom;
                }
                if ( r < tout ) goto CalledFrom;
                while (  t < tt ) *w++ = *t++;
                ww[1] = w - ww;
                if ( ww[1] == 2 ) w = ww;
                while ( t < tnext ) *w++ = *t++;
                *wterm = w - wterm;
            }
            *w = 0;
        }
noindices:
        tstore = tout;
    }
/*
    #] INDEX :
    #[ VECTOR/DELTA :
*/
    code[0] = VECTOR; code[1] = DELTA;
    for ( k = 0; k < 2; k++ ) {
      t = in;
      tnext = t + *t;
      tstop = tnext-ABS(tnext[-1]);
      t++;
      while ( t < tstop ) {
        if ( *t == code[k] ) {
            i = t[1]; NCOPY(tout,t,i); break;
        }
        else t += t[1];
      }
      if ( tout > tstore ) { /* There are vectors in the first term */
        t = tnext;
        while ( *t ) {
            tnext = t + *t;
            rstop = tnext - ABS(tnext[-1]);
            r = t+1;
            if ( r == rstop ) { tstore = tout; goto novectors; }
            while ( r < rstop ) {
                if ( *r != code[k] ) { r += r[1]; continue; }
                m = tstore+2;
                while ( m < tout ) {
                    for ( i = 2; i < r[1]; i += 2 ) {
                        if ( *m == r[i] && m[1] == r[i+1] ) break;
                        if ( *m > r[i] || ( *m == r[i] && m[1] > r[i+1] ) ) continue;
                        mm = m+2;
                        while ( mm < tout ) { mm[-2] = mm[0]; mm[-1] = mm[1]; mm += 2; }
                        tout -= 2; tstore[1] -= 2; m -= 2;
                        break;
                    }
                    m += 2;
                }
            }
            if ( r >= rstop || tout <= tstore+2 ) {
                tout = tstore; break;
            }
```

```
        }
        if ( tout > tstore+2 ) { /* Now we have to take out what is in tstore */
            t = in; w = in;
            while ( *t ) {
                wterm = w;
                tnext = t + *t; t++; w++;
                while ( *t != code[k] ) { i = t[1]; NCOPY(w,t,i); }
                tt = t + t[1]; t += 2; r = tstore+2; ww = w; *w++ = code[k]; w++;
                while ( r < tout && t < tt ) {
                    if ( ( *r > *t ) || ( *r == *t && r[1] > t[1] ) )
                        { *w++ = *t++; *w++ = *t++; }
                    else if ( *r == *t && r[1] == t[1] ) { r += 2; t += 2; }
                    else goto CalledFrom;
                }
                if ( r < tout ) goto CalledFrom;
                while (  t < tt ) *w++ = *t++;
                ww[1] = w - ww;
                if ( ww[1] == 2 ) w = ww;
                while ( t < tnext ) *w++ = *t++;
                *wterm = w - wterm;
            }
            *w = 0;
        }
        tstore = tout;
      }
    }
novectors:;
/*
    #] VECTOR/DELTA :
    #[ FUNCTIONS :
*/
    t = in;
    tnext = t + *t;
    tstop = tnext-ABS(tnext[-1]);
    t++;
    tcom = 0;
    while ( t < tstop ) {
        if ( *t >= FUNCTION ) {
            if ( functions[*t-FUNCTION].commute ) {
                if ( tcom == 0 ) { tcom = tstore; }
                else {
                    for ( i = 0; i < t[1]; i++ ) {
                        if ( t[i] != tcom[i] ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("GCD or factorization of more than one noncommuting object not allowed");
                            MUNLOCK(ErrorMessageLock);
                            goto CalledFrom;
                        }
                    }
                }
            }
            i = t[1]; NCOPY(tstore,t,i);
        }
        else t += t[1];
    }
    if ( tout > tstore ) {
        t = tnext;
        while ( *t ) {
            tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
            if ( t == tstop ) goto nofunctions;
            r = tstore;
            while ( r < tout ) {
                tt = t;
                while ( tt < tstop ) {
                    for ( i = 0; i < r[1]; i++ ) {
                        if ( r[i] != tt[i] ) break;
                    }
                    if ( i == r[1] ) { r += r[1]; goto nextr1; }
                }
/*
                Not encountered in this term. Scratch from list
*/
                m = r; mm = r + r[1];
                while ( mm < tout ) *m++ = *mm++;
                tout = m;
nextr1:;
            }
            if ( tout <= tstore ) break;
            t += *t;
        }
    }
    if ( tout > tstore ) {
/*
        Now we have one or more functions left that are common in all terms.
        Take them out. We do this one by one.
*/
```

```
        r = tstore;
        while ( r < tout ) {
            t = in; ww = in; w = ww+1;
            while ( *t ) {
                tnext = t + *t;
                t++;
                for(;;) {
                    for ( i = 0; i < r[1]; i++ ) {
                        if ( t[i] != r[i] ) {
                            j = t[1]; NCOPY(w,t,j);
                            break;
                        }
                    }
                    if ( i == r[1] ) {
                        t += t[1];
                        while ( t < tnext ) *w++ = *t++;
                        *ww = w - ww;
                        break;
                    }
                }
            }
            r += r[1];
            *w = 0;
        }
nofunctions:
        tstore = tout;
    }
/*
    #] FUNCTIONS :
    #[ SYMBOL :

    We make a list of symbols and their minimal powers.
    This includes negative powers. In the end we have to multiply by the
    inverse of this list. That takes out all negative powers and leaves
    things ready for further processing.
*/
    tterm = AT.WorkPointer; tt = tterm+1;
    tout[0] = SYMBOL; tout[1] = 2;
    t = in;
    tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
    while ( t < tstop ) {
        if ( *t == SYMBOL ) {
            for ( i = 0; i < t[1]; i++ ) tout[i] = t[i];
            break;
        }
        t += t[1];
    }
    t = tnext;
    while ( *t ) {
        tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
        if ( t == tstop ) {
            tout[1] = 2;
            break;
        }
        else {
            while ( t < tstop ) {
                if ( *t == SYMBOL ) {
                    MergeSymbolLists(BHEAD tout,t,-1);
                    break;
                }
                t += t[1];
            }
            t = tnext;
        }
    }
    if ( tout[1] > 2 ) {
        t = tout;
        tt[0] = t[0]; tt[1] = t[1];
        for ( i = 2; i < t[1]; i += 2 ) {
            tt[i] = t[i]; tt[i+1] = -t[i+1];
        }
        tt += tt[1];
        tout += tout[1];
        action++;
    }
/*
    #] SYMBOL :
    #[ DOTPRODUCT :

    We make a list of dotproducts and their minimal powers.
    This includes negative powers. In the end we have to multiply by the
    inverse of this list. That takes out all negative powers and leaves
    things ready for further processing.
*/
    tout[0] = DOTPRODUCT; tout[1] = 2;
    t = in;
    while ( *t ) {
```

```
        tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
        if ( t == tstop ) {
            tout[1] = 2;
            break;
        }
        while ( t < tstop ) {
            if ( *t == DOTPRODUCT ) {
                MergeDotproductLists(BHEAD tout,t,-1);
                break;
            }
            t += t[1];
        }
        t = tnext;
    }
    if ( tout[1] > 2 ) {
        t = tout;
        tt[0] = t[0]; tt[1] = t[1];
        for ( i = 2; i < t[1]; i += 3 ) {
            tt[i] = t[i]; tt[i+1] = t[i+1]; tt[i+2] = -t[i+2];
        }
        tt += tt[1];
        tout += tout[1];
        action++;
    }
/*
    #] DOTPRODUCT :
    #[ Coefficient :

    Now we have to collect the GCD of the numerators
    and the LCM of the denominators.
*/
    AT.WorkPointer = tt;
    if ( AN.cmod != 0 ) {
      WORD x, ix, ip;
      t = in; tnext = t + *t; tstop = tnext - ABS(tnext[-1]);
      x = tstop[0];
      if ( tnext[-1] < 0 ) x += AC.cmod[0];
      if ( GetModInverses(x,(WORD)(AN.cmod[0]),&ix,&ip) ) goto CalledFrom;
      *tout++ = x; *tout++ = 1; *tout++ = 3;
      *tt++ = ix; *tt++ = 1; *tt++ = 3;
    }
    else {
      GCDbuffer  = NumberMalloc("MakeInteger");
      GCDbuffer2 = NumberMalloc("MakeInteger");
      LCMbuffer  = NumberMalloc("MakeInteger");
      LCMbuffer2 = NumberMalloc("MakeInteger");
      t = in;
      tnext = t + *t; length = tnext[-1];
      if ( length < 0 ) { sign = -1; length = -length; }
      else              { sign = 1; }
      tstop = tnext - length;
      redlength = (length-1)/2;
      for ( i = 0; i < redlength; i++ ) {
        GCDbuffer[i] = (UWORD)(tstop[i]);
        LCMbuffer[i] = (UWORD)(tstop[redlength+i]);
      }
      GCDlen = LCMlen = redlength;
      while ( GCDbuffer[GCDlen-1] == 0 ) GCDlen--;
      while ( LCMbuffer[LCMlen-1] == 0 ) LCMlen--;
      t = tnext;
      while ( *t ) {
        tnext = t + *t; length = ABS(tnext[-1]);
        tstop = tnext - length; redlength = (length-1)/2;
        len1 = len2 = redlength;
        den = tstop + redlength;
        while ( tstop[len1-1] == 0 ) len1--;
        while ( den[len2-1] == 0 ) len2--;
        if ( GCDlen == 1 && GCDbuffer[0] == 1 ) {}
        else {
            GcdLong(BHEAD (UWORD *)tstop,len1,GCDbuffer,GCDlen,GCDbuffer2,&GCDlen2);
            ap = GCDbuffer; GCDbuffer = GCDbuffer2; GCDbuffer2 = ap;
            a = GCDlen; GCDlen = GCDlen2; GCDlen2 = a;
        }
        if ( len2 == 1 && den[0] == 1 ) {}
        else {
            GcdLong(BHEAD LCMbuffer,LCMlen,(UWORD *)den,len2,LCMbuffer2,&LCMlen2);
            DivLong((UWORD *)den,len2,LCMbuffer2,LCMlen2,
                GCDbuffer2,&GCDlen2,(UWORD *)AT.WorkPointer,&a);
            MulLong(LCMbuffer,LCMlen,GCDbuffer2,GCDlen2,LCMbuffer2,&LCMlen2);
            ap = LCMbuffer; LCMbuffer = LCMbuffer2; LCMbuffer2 = ap;
            a = LCMlen; LCMlen = LCMlen2; LCMlen2 = a;
        }
        t = tnext;
      }
      if ( GCDlen != 1 || GCDbuffer[0] != 1 || LCMlen != 1 || LCMbuffer[0] != 1 ) {
        redlength = GCDlen; if ( LCMlen > GCDlen ) redlength = LCMlen;
        for ( i = 0; i < GCDlen; i++ ) *tout++ = (WORD)(GCDbuffer[i]);
```

```
        for ( ; i < redlength; i++ ) *tout++ = 0;
        for ( i = 0; i < LCMlen; i++ ) *tout++ = (WORD)(LCMbuffer[i]);
        for ( ; i < redlength; i++ ) *tout++ = 0;
        *tout++ = (2*redlength+1)*sign;
        for ( i = 0; i < LCMlen; i++ ) *tt++ = (WORD)(LCMbuffer[i]);
        for ( ; i < redlength; i++ ) *tt++ = 0;
        for ( i = 0; i < GCDlen; i++ ) *tt++ = (WORD)(GCDbuffer[i]);
        for ( ; i < redlength; i++ ) *tt++ = 0;
        *tt++ = (2*redlength+1)*sign;
        action++;
      }
      else {
        *tout++ = 1; *tout++ = 1; *tout++ = 3*sign;
        *tt++ = 1; *tt++ = 1; *tt++ = 3*sign;
        if ( sign != 1 ) action++;
      }
      *tout = 0;
      NumberFree(LCMbuffer2,"MakeInteger");
      NumberFree(LCMbuffer ,"MakeInteger");
      NumberFree(GCDbuffer2,"MakeInteger");
      NumberFree(GCDbuffer ,"MakeInteger");
    }
/*
    #] Coefficient :
    #[ Multiply by the inverse content :
*/
    if ( action ) {
        *tterm = tt - tterm;
        AT.WorkPointer = tt;
        inp = MultiplyWithTerm(BHEAD in,tterm,2);
        AT.WorkPointer = tterm;
        in = inp;
    }
/*
    #] Multiply by the inverse content :
*/
    *term = tout - term;
    AT.WorkPointer = tterm;
    return(in);
CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("TakeContent");
    MUNLOCK(ErrorMessageLock);
    return(0);
}

/*
        #] TakeContent :
        #[ MergeSymbolLists :

        Merges the extra list into the old.
        If par == -1 we take minimum powers
        If par ==  1 we take maximum powers
        If par ==  0 we take minimum of the absolute value of the powers
                     if one is positive and the other negative we get zero.
        We assume that the symbols are in order in both lists
*/

int MergeSymbolLists(PHEAD WORD *old, WORD *extra, int par)
{
    GETBIDENTITY
    WORD *new = TermMalloc("MergeSymbolLists");
    WORD *t1, *t2, *fill;
    int i1,i2;
    fill = new + 2; *new = SYMBOL;
    i1 = old[1] - 2; i2 = extra[1] - 2;
    t1 = old + 2; t2 = extra + 2;
    switch ( par ) {
        case -1:
            while ( i1 > 0 && i2 > 0 ) {
                if ( *t1 > *t2 ) {
                    if ( t2[1] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
                    else t2 += 2;
                    i2 -= 2;
                }
                else if ( *t1 < *t2 ) {
                    if ( t1[1] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
                    else t1 += 2;
                    i1 -= 2;
                }
                else if ( t1[1] < t2[1] ) {
                    *fill++ = *t1++; *fill++ = *t1++; t2 += 2;
                    i1 -= 2; i2 -=2;
                }
                else {
                    *fill++ = *t2++; *fill++ = *t2++; t1 += 2;
                    i1 -= 2; i2 -=2;
```

```
                }
            }
            for ( ; i1 > 0; i1 -= 2 ) {
                if ( t1[1] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
                else t1 += 2;
            }
            for ( ; i2 > 0; i2 -= 2 ) {
                if ( t2[1] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
                else t2 += 2;
            }
            break;
        case 1:
            while ( i1 > 0 && i2 > 0 ) {
                if ( *t1 > *t2 ) {
                    if ( t2[1] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
                    else t2 += 2;
                    i2 -=2;
                }
                else if ( *t1 < *t2 ) {
                    if ( t1[1] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
                    else t1 += 2;
                    i1 -= 2;
                }
                else if ( t1[1] > t2[1] ) {
                    *fill++ = *t1++; *fill++ = *t1++; t2 += 2;
                    i1 -= 2; i2 -=2;
                }
                else {
                    *fill++ = *t2++; *fill++ = *t2++; t1 += 2;
                    i1 -= 2; i2 -=2;
                }
            }
            for ( ; i1 > 0; i1 -= 2 ) {
                if ( t1[1] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
                else t1 += 2;
            }
            for ( ; i2 > 0; i2 -= 2 ) {
                if ( t2[1] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
                else t2 += 2;
            }
            break;
        case 0:
            while ( i1 > 0 && i2 > 0 ) {
                if ( *t1 > *t2 ) {
                    t2 += 2; i2 -= 2;
                }
                else if ( *t1 < *t2 ) {
                    t1 += 2; i1 -= 2;
                }
                else if ( ( t1[1] > 0 ) && ( t2[1] < 0 ) ) { t1 += 2; t2 += 2; i1 -= 2; i2 -= 2; }
                else if ( ( t1[1] < 0 ) && ( t2[1] > 0 ) ) { t1 += 2; t2 += 2; i1 -= 2; i2 -= 2; }
                else if ( t1[1] > 0 ) {
                    if ( t1[1] < t2[1] ) {
                        *fill++ = *t1++; *fill++ = *t1++; t2 += 2; i2 -= 2;
                    }
                    else {
                        *fill++ = *t2++; *fill++ = *t2++; t1 += 2; i1 -= 2;
                    }
                }
                else {
                    if ( t2[1] < t1[1] ) {
                        *fill++ = *t2++; *fill++ = *t2++; t1 += 2; i1 -= 2; i2 -= 2;
                    }
                    else {
                        *fill++ = *t1++; *fill++ = *t1++; t2 += 2; i1 -= 2; i2 -= 2;
                    }
                }
            }
            for ( ; i1 > 0; i1-- ) *fill++ = *t1++;
            for ( ; i2 > 0; i2-- ) *fill++ = *t2++;
            break;
    }
    i1 = new[1] = fill - new;
    t2 = new; t1 = old; NCOPY(t1,t2,i1);
    TermFree(new,"MergeSymbolLists");
    return(0);
}

/*
        #] MergeSymbolLists :
        #[ MergeDotproductLists :

        Merges the extra list into the old.
        If par == -1 we take minimum powers
        If par ==  1 we take maximum powers
        If par ==  0 we take minimum of the absolute value of the powers
                     if one is positive and the other negative we get zero.
```

```
        We assume that the dotproducts are in order in both lists
*/

int MergeDotproductLists(PHEAD WORD *old, WORD *extra, int par)
{
    GETBIDENTITY
    WORD *new = TermMalloc("MergeDotproductLists");
    WORD *t1, *t2, *fill;
    int i1,i2;
    fill = new + 2;
    i1 = old[1] - 2; i2 = extra[1] - 2;
    t1 = old + 2; t2 = extra + 2;
    switch ( par ) {
        case -1:
            while ( i1 > 0 && i2 > 0 ) {
                if ( ( *t1 > *t2 ) || ( *t1 == *t2 && t1[1] > t2[1] ) ) {
                    if ( t2[2] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; }
                    else t2 += 3;
                }
                else if ( ( *t1 < *t2 ) || ( *t1 == *t2 && t1[1] < t2[1] ) ) {
                    if ( t1[2] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; }
                    else t1 += 3;
                }
                else if ( t1[2] < t2[2] ) {
                    *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
                }
                else {
                    *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
                }
            }
            break;
        case 1:
            while ( i1 > 0 && i2 > 0 ) {
                if ( ( *t1 > *t2 ) || ( *t1 == *t2 && t1[1] > t2[1] ) ) {
                    if ( t2[2] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; }
                    else t2 += 3;
                }
                else if ( ( *t1 < *t2 ) || ( *t1 == *t2 && t1[1] < t2[1] ) ) {
                    if ( t1[2] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; }
                    else t1 += 3;
                }
                else if ( t1[2] > t2[2] ) {
                    *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
                }
                else {
                    *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
                }
            }
            break;
        case 0:
            while ( i1 > 0 && i2 > 0 ) {
                if ( ( *t1 > *t2 ) || ( *t1 == *t2 && t1[1] > t2[1] ) ) {
                    t2 += 3;
                }
                else if ( ( *t1 < *t2 ) || ( *t1 == *t2 && t1[1] < t2[1] ) ) {
                    t1 += 3;
                }
                else if ( ( t1[2] > 0 ) && ( t2[2] < 0 ) ) { t1 += 3; t2 += 3; }
                else if ( ( t1[2] < 0 ) && ( t2[2] > 0 ) ) { t1 += 3; t2 += 3; }
                else if ( t1[2] > 0 ) {
                    if ( t1[2] < t2[2] ) {
                        *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
                    }
                    else {
                        *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
                    }
                }
                else {
                    if ( t2[2] < t1[2] ) {
                        *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
                    }
                    else {
                        *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
                    }
                }
            }
            break;
    }
    i1 = new[1] = fill - new;
    t2 = new; t1 = old; NCOPY(t1,t2,i1);
    TermFree(new,"MergeDotproductLists");
    return(0);
}

/*
        #] MergeDotproductLists :
        #[ CreateExpression :
```

```

        Looks for the expression in the argument, reads it and puts it
        in a buffer. Returns the address of the buffer.
        We send the expression through the Generator system, because there
        may be unsubstituted (sub)expressions as in
        Local F = (a+b);
        Local G = gcd_(F,...);
*/

WORD *CreateExpression(PHEAD WORD nexp)
{
    GETBIDENTITY
    CBUF *C = cbuf+AC.cbufnum;
    POSITION startposition, oldposition;
    FILEHANDLE *fi;
    WORD *term, *oldipointer = AR.CompressPointer;
;
    switch ( Expressions[nexp].status ) {
        case HIDDENLEXPRESSION:
        case HIDDENGEXPRESSION:
        case DROPHLEXPRESSION:
        case DROPHGEXPRESSION:
        case UNHIDELEXPRESSION:
        case UNHIDEGEXPRESSION:
            AR.GetOneFile = 2; fi = AR.hidefile;
            break;
        default:
            AR.GetOneFile = 0; fi = AR.infile;
            break;
    }
    SeekScratch(fi,&oldposition);
    startposition = AS.OldOnFile[nexp];
    term = AT.WorkPointer;
    if ( GetOneTerm(BHEAD term,fi,&startposition,0) <= 0 ) goto CalledFrom;
    NewSort(BHEAD0);
    AR.CompressPointer = oldipointer;
    while ( GetOneTerm(BHEAD term,fi,&startposition,0) > 0 ) {
        AT.WorkPointer = term + *term;
        if ( Generator(BHEAD term,C->numlhs) ) {
            LowerSortLevel();
            goto CalledFrom;
        }
        AR.CompressPointer = oldipointer;
    }
    AT.WorkPointer = term;
    if ( EndSort(BHEAD (WORD *)((VOID *)(&term)),2) < 0 ) goto CalledFrom;
    SetScratch(fi,&oldposition);
    return(term);
CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("CreateExpression");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(0);
}

/*
        #] CreateExpression :
        #[ GCDterms :      GCD of two terms

    Computes the GCD of two terms.
    Output in termout.
    termout may overlap with term1.
*/

int GCDterms(PHEAD WORD *term1, WORD *term2, WORD *termout)
{
    GETBIDENTITY
    WORD *t1, *t1stop, *t1next, *t2, *t2stop, *t2next, *tout, *tt1, *tt2;
    int count1, count2, i, ii, x1, sign;
    WORD length1, length2;
    t1 = term1 + *term1; t1stop = t1 - ABS(t1[-1]); t1 = term1+1;
    t2 = term2 + *term2; t2stop = t2 - ABS(t2[-1]); t2 = term2+1;
    tout = termout+1;
    while ( t1 < t1stop ) {
        t1next = t1 + t1[1];
        t2 = term2+1;
        if ( *t1 == SYMBOL ) {
            while ( t2 < t2stop && *t2 != SYMBOL ) t2 += t2[1];
            if ( *t2 == SYMBOL ) {
                t2next = t2+t2[1];
                tt1 = t1+2; tt2 = t2+2; count1 = 0;
                while ( tt1 < t1next && tt2 < t2next ) {
                    if ( *tt1 < *tt2 ) tt1 += 2;
                    else if ( *tt1 > *tt2 ) tt2 += 2;
                    else if ( ( tt1[1] > 0 && tt2[1] < 0 ) ||
                              ( tt2[1] > 0 && tt1[1] < 0 ) ) {
```

```
                        tt1 += 2; tt2 += 2;
                    }
                    else {
                        x1 = tt1[1];
                        if ( tt1[1] < 0 ) { if ( tt2[1] > x1 ) x1 = tt2[1]; }
                        else              { if ( tt2[1] < x1 ) x1 = tt2[1]; }
                        tout[count1+2] = *tt1;
                        tout[count1+3] = x1;
                        tt1 += 2; tt2 += 2;
                        count1 += 2;
                    }
                }
                if ( count1 > 0 ) {
                    *tout = SYMBOL; tout[1] = count1+2; tout += tout[1];
                }
            }
        }
        else if ( *t1 == DOTPRODUCT ) {
            while ( t2 < t2stop && *t2 != DOTPRODUCT ) t2 += t2[1];
            if ( *t2 == DOTPRODUCT ) {
                t2next = t2+t2[1];
                tt1 = t1+2; tt2 = t2+2; count1 = 0;
                while ( tt1 < t1next && tt2 < t2next ) {
                    if ( *tt1 < *tt2 || ( *tt1 == *tt2 && tt1[1] < tt2[1] ) ) tt1 += 3;
                    else if ( *tt1 > *tt2 || ( *tt1 == *tt2 && tt1[1] > tt2[1] ) ) tt2 += 3;
                    else if ( ( tt1[2] > 0 && tt2[2] < 0 ) ||
                              ( tt2[2] > 0 && tt1[2] < 0 ) ) {
                        tt1 += 3; tt2 += 3;
                    }
                    else {
                        x1 = tt1[2];
                        if ( tt1[2] < 0 ) { if ( tt2[2] > x1 ) x1 = tt2[2]; }
                        else              { if ( tt2[2] < x1 ) x1 = tt2[2]; }
                        tout[count1+2] = *tt1;
                        tout[count1+3] = tt1[1];
                        tout[count1+4] = x1;
                        tt1 += 3; tt2 += 3;
                        count1 += 3;
                    }
                }
                if ( count1 > 0 ) {
                    *tout = DOTPRODUCT; tout[1] = count1+2; tout += tout[1];
                }
            }
        }
        else if ( *t1 == VECTOR ) {
            while ( t2 < t2stop && *t2 != VECTOR ) t2 += t2[1];
            if ( *t2 == VECTOR ) {
                t2next = t2+t2[1];
                tt1 = t1+2; tt2 = t2+2; count1 = 0;
                while ( tt1 < t1next && tt2 < t2next ) {
                    if ( *tt1 < *tt2 || ( *tt1 == *tt2 && tt1[1] < tt2[1] ) ) tt1 += 2;
                    else if ( *tt1 > *tt2 || ( *tt1 == *tt2 && tt1[1] > tt2[1] ) ) tt2 += 2;
                    else {
                        tout[count1+2] = *tt1;
                        tout[count1+3] = tt1[1];
                        tt1 += 2; tt2 += 2;
                        count1 += 2;
                    }
                }
                if ( count1 > 0 ) {
                    *tout = VECTOR; tout[1] = count1+2; tout += tout[1];
                }
            }
        }
        else if ( *t1 == INDEX ) {
            while ( t2 < t2stop && *t2 != INDEX ) t2 += t2[1];
            if ( *t2 == INDEX ) {
                t2next = t2+t2[1];
                tt1 = t1+2; tt2 = t2+2; count1 = 0;
                while ( tt1 < t1next && tt2 < t2next ) {
                    if ( *tt1 < *tt2 ) tt1 += 1;
                    else if ( *tt1 > *tt2 ) tt2 += 1;
                    else {
                        tout[count1+2] = *tt1;
                        tt1 += 1; tt2 += 1;
                        count1 += 1;
                    }
                }
                if ( count1 > 0 ) {
                    *tout = INDEX; tout[1] = count1+2; tout += tout[1];
                }
            }
        }
        else if ( *t1 == DELTA ) {
            while ( t2 < t2stop && *t2 != DELTA ) t2 += t2[1];
            if ( *t2 == DELTA ) {
```

```
                t2next = t2+t2[1];
                tt1 = t1+2; tt2 = t2+2; count1 = 0;
                while ( tt1 < t1next && tt2 < t2next ) {
                    if ( *tt1 < *tt2 || ( *tt1 == *tt2 && tt1[1] < tt2[1] ) ) tt1 += 2;
                    else if ( *tt1 > *tt2 || ( *tt1 == *tt2 && tt1[1] > tt2[1] ) ) tt2 += 2;
                    else {
                        tout[count1+2] = *tt1;
                        tout[count1+3] = tt1[1];
                        tt1 += 2; tt2 += 2;
                        count1 += 2;
                    }
                }
                if ( count1 > 0 ) {
                    *tout = DELTA; tout[1] = count1+2; tout += tout[1];
                }
            }
        }
        else if ( *t1 >= FUNCTION ) { /* noncommuting functions? Forbidden! */
/*
            Count how many times this function occurs.
            Then count how many times it is in term2.
*/
            count1 = 1;
            while ( t1next < t1stop && *t1 == *t1next && t1[1] == t1next[1] ) {
                for ( i = 2; i < t1[1]; i++ ) {
                    if ( t1[i] != t1next[i] ) break;
                }
                if ( i < t1[1] ) break;
                count1++;
                t1next += t1next[1];
            }
            count2 = 0;
            while ( t2 < t2stop ) {
                if ( *t2 == *t1 && t2[1] == t1[1] ) {
                    for ( i = 2; i < t1[1]; i++ ) {
                        if ( t2[i] != t1[i] ) break;
                    }
                    if ( i >= t1[1] ) count2++;
                }
                t2 += t2[1];
            }
            if ( count1 < count2 ) count2 = count1; /* number of common occurrences */
            if ( count2 > 0 ) {
                if ( tout == t1 ) {
                    while ( count2 > 0 ) { tout += tout[1]; count2--; }
                }
                else {
                    i = t1[1]*count2;
                    NCOPY(tout,t1,i);
                }
            }
        }
        t1 = t1next;
    }
/*
        Now the coefficients. They are in t1stop and t2stop. Should go to tout.
*/
    sign = 1;
    length1 = term1[*term1-1]; ii = i = ABS(length1); t1 = t1stop;
    if ( t1 != tout ) { NCOPY(tout,t1,i); tout -= ii; }
    length2 = term2[*term2-1];
    if ( length1 < 0 && length2 < 0 ) sign = -1;
    if ( AccumGCD(BHEAD (UWORD *)tout,&length1,(UWORD *)t2stop,length2) ) {
        MLOCK(ErrorMessageLock);
        MesCall("GCDterms");
        MUNLOCK(ErrorMessageLock);
        SETERROR(-1)
    }
    if ( sign < 0 && length1 > 0 ) length1 = -length1;
    tout += ABS(length1); tout[-1] = length1;
    *termout = tout - termout; *tout = 0;
    return(0);
}

/*
        #] GCDterms :
        #[ ReadPolyRatFun :
*/

int ReadPolyRatFun(PHEAD WORD *term)
{
    WORD *oldworkpointer = AT.WorkPointer;
    int flag, i;
    WORD *t, *fun, *nextt, *num, *den, *t1, *t2, size, numsize, densize;
    WORD *term1, *term2, *confree1, *confree2, *gcd, *num1, *den1, move, *newnum, *newden;
    WORD *tstop, *m1, *m2;
    WORD oldsorttype = AR.SortType;
```

```
    void *oldcompareroutine = AR.CompareRoutine;
    AR.SortType = SORTHIGHFIRST;
    AR.CompareRoutine = (void *)&CompareSymbols;

    tstop = term + *term; tstop -= ABS(tstop[-1]);
    if ( term + *term == AT.WorkPointer ) flag = 1;
    else flag = 0;
    t = term+1;
    while ( t < tstop ) {
        if ( *t != AR.PolyFun ) { t += t[1]; continue; }
        if ( ( t[2] & MUSTCLEANPRF ) == 0 ) { t += t[1]; continue; }
        fun = t;
        nextt = t + t[1];
        if ( fun[1] > FUNHEAD && fun[FUNHEAD] == -SNUMBER && fun[FUNHEAD+1] == 0 )
            { *term = 0; break; }
        if ( FromPolyRatFun(BHEAD fun, &num, &den) > 0 ) { t = nextt; continue; }
        if ( *num == ARGHEAD ) { *term = 0; break; }
/*
        Now we have num and den. Both are in general argument notation,
        but can also be used as expressions as in num+ARGHEAD, den+ARGHEAD.
        We need the gcd. For this we have to take out the contents
        because PreGCD does not like contents.
*/
        term1 = TermMalloc("ReadPolyRatFun");
        term2 = TermMalloc("ReadPolyRatFun");
        confree1 = TakeSymbolContent(BHEAD num+ARGHEAD,term1);
        confree2 = TakeSymbolContent(BHEAD den+ARGHEAD,term2);
        GCDclean(BHEAD term1,term2);
/*      gcd = PreGCD(BHEAD confree1,confree2,1); */
        gcd = poly_gcd(BHEAD confree1,confree2,1);
        newnum = PolyDiv(BHEAD confree1,gcd,"ReadPolyRatFun");
        newden = PolyDiv(BHEAD confree2,gcd,"ReadPolyRatFun");
        TermFree(confree2,"ReadPolyRatFun");
        TermFree(confree1,"ReadPolyRatFun");
        num1 = MULfunc(BHEAD term1,newnum);
        den1 = MULfunc(BHEAD term2,newden);
        TermFree(newnum,"ReadPolyRatFun");
        TermFree(newden,"ReadPolyRatFun");
/*      M_free(gcd,"poly_gcd"); */
        TermFree(gcd,"poly_gcd");
        TermFree(term1,"ReadPolyRatFun");
        TermFree(term2,"ReadPolyRatFun");
/*
        Now we can put the function back together.
        Notice that we cannot use ToFast, because there is no reservation
        for the header of the argument. Fortunately there are only two
        types of fast arguments.
*/
        if ( num1[0] == 4 && num1[4] == 0 && num1[2] == 1 && num1[1] > 0 ) {
            numsize = 2; num1[0] = -SNUMBER;
            if ( num1[3] < 0 ) num1[1] = -num1[1];
        }
        else if ( num1[0] == 8 && num1[8] == 0 && num1[7] == 3 && num1[6] == 1
            && num1[5] == 1 && num1[1] == SYMBOL && num1[4] == 1 ) {
            numsize = 2; num1[0] = -SYMBOL; num1[1] = num1[3];
        }
        else { m1 = num1; while ( *m1 ) m1 += *m1; numsize = (m1-num1)+ARGHEAD; }
        if ( den1[0] == 4 && den1[4] == 0 && den1[2] == 1 && den1[1] > 0 ) {
            densize = 2; den1[0] = -SNUMBER;
            if ( den1[3] < 0 ) den1[1] = -den1[1];
        }
        else if ( den1[0] == 8 && den1[8] == 0 && den1[7] == 3 && den1[6] == 1
            && den1[5] == 1 && den1[1] == SYMBOL && den1[4] == 1 ) {
            densize = 2; den1[0] = -SYMBOL; den1[1] = den1[3];
        }
        else { m2 = den1; while ( *m2 ) m2 += *m2; densize = (m2-den1)+ARGHEAD; }
        size = FUNHEAD+numsize+densize;

        if ( size > fun[1] ) {
            move = size - fun[1];
            t1 = term+*term; t2 = t1+move;
            while ( t1 > nextt ) *--t2 = *--t1;
            tstop += move; nextt += move;
            *term += move;
        }
        else if ( size < fun[1] ) {
            move = fun[1]-size;
            t2 = fun+size; t1 = nextt;
            tstop -= move; nextt -= move;
            t = term+*term;
            while ( t1 < t ) *t2++ = *t1++;
            *term -= move;
        }
        else { /* no need to move anything */ }
        fun[1] = size; fun[2] = 0;
        t2 = fun+FUNHEAD; t1 = num1;
        if ( *num1 < 0 ) { *t2++ = num1[0]; *t2++ = num1[1]; }
```

```
        else { *t2++ = numsize; *t2++ = 0; FILLARG(t2);
            i = numsize-ARGHEAD; NCOPY(t2,t1,i) }
        t1 = den1;
        if ( *den1 < 0 ) { *t2++ = den1[0]; *t2++ = den1[1]; }
        else { *t2++ = densize; *t2++ = 0; FILLARG(t2);
            i = densize-ARGHEAD; NCOPY(t2,t1,i) }

        TermFree(num1,"MULfunc");
        TermFree(den1,"MULfunc");
        t = nextt;
    }

    if ( flag ) AT.WorkPointer = term ++term;
    else AT.WorkPointer = oldworkpointer;
    AR.CompareRoutine = oldcompareroutine;
    AR.SortType = oldsorttype;
    return(0);
}

/*
        #] ReadPolyRatFun :
        #[ FromPolyRatFun :
*/

int FromPolyRatFun(PHEAD WORD *fun, WORD **numout, WORD **denout)
{
    WORD *nextfun, *tt, *num, *den;
    int i;
    nextfun = fun + fun[1];
    fun += FUNHEAD;
    num = AT.WorkPointer;
    if ( *fun < 0 ) {
        if ( *fun != -SNUMBER && *fun != -SYMBOL ) goto Improper;
        ToGeneral(fun,num,0);
        tt = num + *num; *tt++ = 0;
        fun += 2;
    }
    else { i = *fun; tt = num; NCOPY(tt,fun,i); *tt++ = 0; }
    den = tt;
    if ( *fun < 0 ) {
        if ( *fun != -SNUMBER && *fun != -SYMBOL ) goto Improper;
        ToGeneral(fun,den,0);
        tt = den + *den; *tt++ = 0;
        fun += 2;
    }
    else { i = *fun; tt = den; NCOPY(tt,fun,i); *tt++ = 0; }
    *numout = num; *denout = den;
    if ( fun != nextfun ) { return(1); }
    AT.WorkPointer = tt;
    return(0);
Improper:
    MLOCK(ErrorMessageLock);
    MesPrint("Improper use of PolyRatFun");
    MesCall("FromPolyRatFun");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1);
}

/*
        #] FromPolyRatFun :
        #[ TakeSymbolContent :
*/
WORD *TakeSymbolContent(PHEAD WORD *in, WORD *term)
{
    GETBIDENTITY
    WORD *t, *tstop, *tout, *tstore;
    WORD *tnext, *tt, *tterm;
    WORD *inp, a, *den, *oldworkpointer = AT.WorkPointer;
    int i, action = 0, sign, first;
    UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMbuffer2, *ap;
    WORD GCDlen, GCDlen2, LCMlen, LCMlen2, length, redlength, len1, len2;
    LONG j;
    tout = tstore = term+1;
/*
    #[ SYMBOL :

    We make a list of symbols and their minimal powers.
    This includes negative powers. In the end we have to multiply by the
    inverse of this list. That takes out all negative powers and leaves
    things ready for further processing.
*/
    tterm = AT.WorkPointer; tt = tterm+1;
    tout[0] = SYMBOL; tout[1] = 2;
    t = in; first = 1;
    while ( *t ) {
        tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
        while ( t < tstop ) {
```

```
            if ( first ) {
                if ( *t == SYMBOL ) {
                    for ( i = 0; i < t[1]; i++ ) tout[i] = t[i];
                    goto didwork;
                }
                else {
                    MLOCK(ErrorMessageLock);
                    MesPrint ((char*)"ERROR: polynomials and polyratfuns must contain symbols only");
                    MUNLOCK(ErrorMessageLock);
                    Terminate(1);
                }
            }
            else if ( *t == SYMBOL ) {
                MergeSymbolLists(BHEAD tout,t,-1);
                goto didwork;
            }
            else {
                t += t[1];
            }
        }
/*
        Here we come when there were no symbols. Only keep the negative ones.
*/
        if ( first == 0 ) {
            int j = 2;
            for ( i = 2; i < tout[1]; i += 2 ) {
                if ( tout[i+1] < 0 ) {
                    if ( i == j ) { j += 2; }
                    else { tout[j] = tout[i]; tout[j+1] = tout[i+1]; j += 2; }
                }
            }
            tout[1] = j;
        }
didwork:;
        first = 0;
        t = tnext;
    }
    if ( tout[1] > 2 ) {
        t = tout;
        tt[0] = t[0]; tt[1] = t[1];
        for ( i = 2; i < t[1]; i += 2 ) {
            tt[i] = t[i]; tt[i+1] = -t[i+1];
        }
        tt += tt[1];
        tout += tout[1];
        action++;
    }
/*
    #] SYMBOL :
    #[ Coefficient :

    Now we have to collect the GCD of the numerators
    and the LCM of the denominators.
*/
    AT.WorkPointer = tt;
    if ( AN.cmod != 0 ) {
      WORD x, ix, ip;
      t = in; tnext = t + *t; tstop = tnext - ABS(tnext[-1]);
      x = tstop[0];
      if ( tnext[-1] < 0 ) x += AC.cmod[0];
      if ( GetModInverses(x,(WORD)(AN.cmod[0]),&ix,&ip) ) goto CalledFrom;
      *tout++ = x; *tout++ = 1; *tout++ = 3;
      *tt++ = ix; *tt++ = 1; *tt++ = 3;
    }
    else {
      GCDbuffer  = NumberMalloc("MakeInteger");
      GCDbuffer2 = NumberMalloc("MakeInteger");
      LCMbuffer  = NumberMalloc("MakeInteger");
      LCMbuffer2 = NumberMalloc("MakeInteger");
      t = in;
      tnext = t + *t; length = tnext[-1];
      if ( length < 0 ) { sign = -1; length = -length; }
      else              { sign = 1; }
      tstop = tnext - length;
      redlength = (length-1)/2;
      for ( i = 0; i < redlength; i++ ) {
        GCDbuffer[i] = (UWORD)(tstop[i]);
        LCMbuffer[i] = (UWORD)(tstop[redlength+i]);
      }
      GCDlen = LCMlen = redlength;
      while ( GCDbuffer[GCDlen-1] == 0 ) GCDlen--;
      while ( LCMbuffer[LCMlen-1] == 0 ) LCMlen--;
      t = tnext;
      while ( *t ) {
        tnext = t + *t; length = ABS(tnext[-1]);
        tstop = tnext - length; redlength = (length-1)/2;
        len1 = len2 = redlength;
```

```
        den = tstop + redlength;
        while ( tstop[len1-1] == 0 ) len1--;
        while ( den[len2-1] == 0 ) len2--;
        if ( GCDlen == 1 && GCDbuffer[0] == 1 ) {}
        else {
            GcdLong(BHEAD (UWORD *)tstop,len1,GCDbuffer,GCDlen,GCDbuffer2,&GCDlen2);
            ap = GCDbuffer; GCDbuffer = GCDbuffer2; GCDbuffer2 = ap;
            a = GCDlen; GCDlen = GCDlen2; GCDlen2 = a;
        }
        if ( len2 == 1 && den[0] == 1 ) {}
        else {
            GcdLong(BHEAD LCMbuffer,LCMlen,(UWORD *)den,len2,LCMbuffer2,&LCMlen2);
            DivLong((UWORD *)den,len2,LCMbuffer2,LCMlen2,
                GCDbuffer2,&GCDlen2,(UWORD *)AT.WorkPointer,&a);
            MulLong(LCMbuffer,LCMlen,GCDbuffer2,GCDlen2,LCMbuffer2,&LCMlen2);
            ap = LCMbuffer; LCMbuffer = LCMbuffer2; LCMbuffer2 = ap;
            a = LCMlen; LCMlen = LCMlen2; LCMlen2 = a;
        }
        t = tnext;
      }
      if ( GCDlen != 1 || GCDbuffer[0] != 1 || LCMlen != 1 || LCMbuffer[0] != 1 ) {
        redlength = GCDlen; if ( LCMlen > GCDlen ) redlength = LCMlen;
        for ( i = 0; i < GCDlen; i++ ) *tout++ = (WORD)(GCDbuffer[i]);
        for ( ; i < redlength; i++ ) *tout++ = 0;
        for ( i = 0; i < LCMlen; i++ ) *tout++ = (WORD)(LCMbuffer[i]);
        for ( ; i < redlength; i++ ) *tout++ = 0;
        *tout++ = (2*redlength+1)*sign;
        for ( i = 0; i < LCMlen; i++ ) *tt++ = (WORD)(LCMbuffer[i]);
        for ( ; i < redlength; i++ ) *tt++ = 0;
        for ( i = 0; i < GCDlen; i++ ) *tt++ = (WORD)(GCDbuffer[i]);
        for ( ; i < redlength; i++ ) *tt++ = 0;
        *tt++ = (2*redlength+1)*sign;
        action++;
      }
      else {
        *tout++ = 1; *tout++ = 1; *tout++ = 3*sign;
        *tt++ = 1; *tt++ = 1; *tt++ = 3*sign;
        if ( sign != 1 ) action++;
      }
      NumberFree(LCMbuffer2,"MakeInteger");
      NumberFree(LCMbuffer ,"MakeInteger");
      NumberFree(GCDbuffer2,"MakeInteger");
      NumberFree(GCDbuffer ,"MakeInteger");
    }
/*
    #] Coefficient :
    #[ Multiply by the inverse content :
*/
    if ( action ) {
        *term = tout - term; *tout = 0;
        *tterm = tt - tterm; *tt = 0;
        AT.WorkPointer = tt;
        inp = MultiplyWithTerm(BHEAD in,tterm,2);
        AT.WorkPointer = tterm;
        t = inp; while ( *t ) t += *t;
        j = (t-inp); t = inp;
        if ( j*sizeof(WORD) > (size_t)(AM.MaxTer) ) goto OverWork;
        in = tout = TermMalloc("TakeSymbolContent");
        NCOPY(tout,t,j); *tout = 0;
        if ( AN.tryterm > 0 ) { TermFree(inp,"MultiplyWithTerm"); AN.tryterm = 0; }
        else { M_free(inp,"MultiplyWithTerm"); }
    }
    else {
        t = in; while ( *t ) t += *t;
        j = (t-in); t = in;
        if ( j*sizeof(WORD) > (size_t)(AM.MaxTer) ) goto OverWork;
        in = tout = TermMalloc("TakeSymbolContent");
        NCOPY(tout,t,j); *tout = 0;
        term[0] = 4; term[1] = 1; term[2] = 1; term[3] = 3; term[4] = 0;
    }
/*
    #] Multiply by the inverse content :
    AT.WorkPointer = tterm + *tterm;
*/
    AT.WorkPointer = oldworkpointer;

    return(in);
OverWork:
    MLOCK(ErrorMessageLock);
    MesPrint("Term too complex. Maybe increasing MaxTermSize can help");
    MUNLOCK(ErrorMessageLock);
CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("TakeSymbolContent");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(0);
```

```
}

/*
        #] TakeSymbolContent :
        #[ GCDclean :

        Takes a numerator and a denominator that each consist of a
        single term with only a coefficient and symbols and makes them
        into a proper fraction. Output overwrites input.
*/

void GCDclean(PHEAD WORD *num, WORD *den)
{
    WORD *out1 = TermMalloc("GCDclean");
    WORD *out2 = TermMalloc("GCDclean");
    WORD *t1, *t2, *r1, *r2, *t1stop, *t2stop, csize1, csize2, csize3, pow, sign;
    int i;
    t1stop = num+*num; sign = ( t1stop[-1] < 0 ) ? -1 : 1;
    csize1 = ABS(t1stop[-1]); t1stop -= csize1;
    t2stop = den+*den; if ( t2stop[-1] < 0 ) sign = -sign;
    csize2 = ABS(t2stop[-1]); t2stop -= csize2;
    t1 = num+1; t2 = den+1;
    r1 = out1+3; r2 = out2+3;
    if ( t1 == t1stop ) {
        if ( t2 < t2stop ) {
            for ( i = 2; i < t2[1]; i += 2 ) {
                if ( t2[i+1] < 0 ) { *r1++ = t2[i]; *r1++ = -t2[i+1]; }
                else { *r2++ = t2[i]; *r2++ = t2[i+1]; }
            }
        }
    }
    else if ( t2 == t2stop ) {
        for ( i = 2; i < t1[1]; i += 2 ) {
            if ( t1[i+1] < 0 ) { *r2++ = t1[i]; *r2++ = -t1[i+1]; }
            else { *r1++ = t1[i]; *r1++ = t1[i+1]; }
        }
    }
    else {
        t1 += 2; t2 += 2;
        while ( t1 < t1stop && t2 < t2stop ) {
            if ( *t1 < *t2 ) {
                if ( t1[1] > 0 ) { *r1++ = *t1; *r1++ = t1[1]; t1 += 2; }
                else if ( t1[1] < 0 ) { *r2++ = *t1; *r2++ = -t1[1]; t1 += 2; }
            }
            else if ( *t1 > *t2 ) {
                if ( t2[1] > 0 ) { *r2++ = *t2; *r2++ = t2[1]; t2 += 2; }
                else if ( t2[1] < 0 ) { *r1++ = *t2; *r1++ = -t2[1]; t2 += 2; }
            }
            else {
                pow = t1[1]-t2[1];
                if ( pow > 0 ) { *r1++ = *t1; *r1++ = pow; }
                else if ( pow < 0 ) { *r2++ = *t1; *r2++ = -pow; }
                t1 += 2; t2 += 2;
            }
        }
        while ( t1 < t1stop ) {
            if ( t1[1] < 0 ) { *r2++ = *t1; *r2++ = -t1[1]; }
            else { *r1++ = *t1; *r1++ = t1[1]; }
            t1 += 2;
        }
        while ( t2 < t2stop ) {
            if ( t2[1] < 0 ) { *r1++ = *t2; *r1++ = -t2[1]; }
            else { *r2++ = *t2; *r2++ = t2[1]; }
            t2 += 2;
        }
    }
    if ( r1 > out1+3 ) { out1[1] = SYMBOL; out1[2] = r1 - out1 - 1; }
    else r1 = out1+1;
    if ( r2 > out2+3 ) { out2[1] = SYMBOL; out2[2] = r2 - out2 - 1; }
    else r2 = out2+1;
/*
    Now the coefficients.
*/
    csize1 = REDLENG(csize1);
    csize2 = REDLENG(csize2);
    if ( DivRat(BHEAD (UWORD *)t1stop,csize1,(UWORD *)t2stop,csize2,(UWORD *)r1,&csize3) ) {
        MLOCK(ErrorMessageLock);
        MesCall("GCDclean");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    UnPack((UWORD *)r1,csize3,&csize2,&csize1);
    t2 = r1+ABS(csize3);
    for ( i = 0; i < csize2; i++ ) r2[i] = t2[i];
    r2 += csize2; *r2++ = 1;
    for ( i = 1; i < csize2; i++ ) *r2++ = 0;
    csize2 = INCLENG(csize2); *r2++ = csize2; *out2 = r2-out2;
```

```
    r1 += ABS(csize1); *r1++ = 1;
    for ( i = 1; i < ABS(csize1); i++ ) *r1++ = 0;
    csize1 = INCLENG(csize1); *r1++ = csize1; *out1 = r1-out1;

    t1 = num; t2 = out1; i = *out1; NCOPY(t1,t2,i); *t1 = 0;
    if ( sign < 0 ) t1[-1] = -t1[-1];
    t1 = den; t2 = out2; i = *out2; NCOPY(t1,t2,i); *t1 = 0;

    TermFree(out2,"GCDclean");
    TermFree(out1,"GCDclean");
}

/*
        #] GCDclean :
        #[ PolyDiv :

        Special stub function for polynomials that should fit inside a term.
        We make sure that the space is allocated by TermMalloc.
        This makes things much easier on the calling routines.
*/

WORD *PolyDiv(PHEAD WORD *a,WORD *b,char *text)
{
/*
    Probably the following would work now
*/
    DUMMYUSE(text);
    return(poly_div(BHEAD a,b,1));
/*
    WORD *quo, *qq;
    WORD *x, *xx;
    LONG i;
    quo = poly_div(BHEAD a,b,1);
    x = TermMalloc(text);
    qq = quo; while ( *qq ) qq += *qq;
    i = (qq-quo+1);
    if ( i*sizeof(WORD) > (size_t)(AM.MaxTer) ) {
        DUMMYUSE(text);
        MLOCK(ErrorMessageLock);
        MesPrint("PolyDiv: Term too complex. Maybe increasing MaxTermSize can help");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    xx = x; qq = quo;
    NCOPY(xx,qq,i)
    TermFree(quo,"poly_div");
    return(x);
*/
}

/*
        #] PolyDiv :
    #] GCDfunction :
    #[ DIVfunction :

    Input: a div_ function that has two arguments inside a term.
    Action: Calculates [arg1/arg2] using polynomial techniques if needed.
    Output: The output result is combined with the remainder of the term
    and sent to Generator for further processing.
    Note that the output can be just a number or many terms.
    In case par == 0 the output is [arg1/arg2]
    In case par == 1 the output is [arg1%arg2]
    In case par == 2 the output is [inverse of arg1 modulus arg2]
    In case par == 3 the output is [arg1*arg2]
*/

WORD divrem[4] = { DIVFUNCTION, REMFUNCTION, INVERSEFUNCTION, MULFUNCTION };

int DIVfunction(PHEAD WORD *term,WORD level,int par)
{
    GETBIDENTITY
    WORD *t, *tt, *r, *arg1 = 0, *arg2 = 0, *arg3 = 0, *termout;
    WORD *tstop, *tend, *r3, *rr, *rstop, tlength, rlength, newlength;
    WORD *proper1, *proper2, *proper3 = 0;
    int numargs = 0, type1, type2, actionflag1, actionflag2;
    WORD startebuf = cbuf[AT.ebufnum].numrhs;
    if ( par < 0 || par > 3 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Internal error. Illegal parameter %d in DIVfunction.",par);
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
/*
    Find the function
*/
    tend = term + *term; tstop = tend - ABS(tend[-1]);
    t = term+1;
```

```
    while ( t < tstop ) {
        if ( *t != divrem[par] ) { t += t[1]; continue; }
        r = t + FUNHEAD;
        tt = t + t[1]; numargs = 0;
        while ( r < tt ) {
            if ( numargs == 0 ) { arg1 = r; }
            if ( numargs == 1 ) { arg2 = r; }
            numargs++;
            NEXTARG(r);
        }
        if ( numargs == 2 ) break;
        t = tt;
    }
    if ( t >= tstop ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Internal error. Indicated div_ or rem_ function not encountered.");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
/*
    We have two arguments in arg1 and arg2.
*/
    if ( *arg1 == -SNUMBER && arg1[1] == 0 ) {
        if ( *arg2 == -SNUMBER && arg2[1] == 0 ) {
zerozero:;
            MLOCK(ErrorMessageLock);
            MesPrint("0/0 in either div_ or rem_ function.");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        return(0);
    }
    if ( *arg2 == -SNUMBER && arg2[1] == 0 ) {
divzero:;
        MLOCK(ErrorMessageLock);
        MesPrint("Division by zero in either div_ or rem_ function.");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    if ( ( arg1 = ConvertArgument(BHEAD arg1, &type1) ) == 0 ) goto CalledFrom;
    if ( ( arg2 = ConvertArgument(BHEAD arg2, &type2) ) == 0 ) goto CalledFrom;
    if ( *arg1 == 0 ) {
        if ( *arg2 == 0 ) {
            M_free(arg2,"DIVfunction");
            M_free(arg1,"DIVfunction");
            goto zerozero;
        }
        M_free(arg2,"DIVfunction");
        M_free(arg1,"DIVfunction");
        return(0);
    }
    if ( *arg2 == 0 ) {
        M_free(arg2,"DIVfunction");
        M_free(arg1,"DIVfunction");
        goto divzero;
    }
    if ( ( proper1 = PutExtraSymbols(BHEAD arg1,startebuf,&actionflag1) ) == 0 ) goto CalledFrom;
    if ( ( proper2 = PutExtraSymbols(BHEAD arg2,startebuf,&actionflag2) ) == 0 ) goto CalledFrom;
/*
    if ( type2 == 0 ) M_free(arg2,"DIVfunction");
    else {
        DOLLARS d = ((DOLLARS)arg2)-1;
        if ( d->factors ) M_free(d->factors,"Dollar factors");
        M_free(d,"Copy of dollar variable");
    }
*/
    M_free(arg2,"DIVfunction");
/*
    if ( type1 == 0 ) M_free(arg1,"DIVfunction");
    else {
        DOLLARS d = ((DOLLARS)arg1)-1;
        if ( d->factors ) M_free(d->factors,"Dollar factors");
        M_free(d,"Copy of dollar variable");
    }
*/
    M_free(arg1,"DIVfunction");
    if ( par == 0 )      proper3 = poly_div(BHEAD proper1, proper2,0);
    else if ( par == 1 ) proper3 = poly_rem(BHEAD proper1, proper2,0);
    else if ( par == 2 ) proper3 = poly_inverse(BHEAD proper1, proper2);
    else if ( par == 3 ) proper3 = poly_mul(BHEAD proper1, proper2);
    if ( proper3 == 0 ) goto CalledFrom;
    if ( actionflag1 || actionflag2 ) {
        if ( ( arg3 = TakeExtraSymbols(BHEAD proper3,startebuf) ) == 0 ) goto CalledFrom;
        M_free(proper3,"DIVfunction");
    }
    else {
        arg3 = proper3;
```

```
    }
    M_free(proper2,"DIVfunction");
    M_free(proper1,"DIVfunction");
    cbuf[AT.ebufnum].numrhs = startebuf;
    if ( *arg3 ) {
        termout = AT.WorkPointer;
        tlength = tend[-1];
        tlength = REDLENG(tlength);
        r3 = arg3;
        while ( *r3 ) {
            tt = term + 1; rr = termout + 1;
            while ( tt < t ) *rr++ = *tt++;
            r = r3 + 1;
            r3 = r3 + *r3;
            rstop = r3 - ABS(r3[-1]);
            while ( r < rstop ) *rr++ = *r++;
            tt += t[1];
            while ( tt < tstop ) *rr++ = *tt++;
            rlength = r3[-1];
            rlength = REDLENG(rlength);
            if ( MulRat(BHEAD (UWORD *)tstop,tlength,(UWORD *)rstop,rlength,
                    (UWORD *)rr,&newlength) < 0 ) goto CalledFrom;
            rlength = INCLENG(newlength);
            rr += ABS(rlength);
            rr[-1] = rlength;
            *termout = rr - termout;
            AT.WorkPointer = rr;
            if ( Generator(BHEAD termout,level) ) goto CalledFrom;
        }
        AT.WorkPointer = termout;
    }
    M_free(arg3,"DIVfunction");
    return(0);
CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("DIVfunction");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
    #] DIVfunction :
    #[ MULfunc :

    Multiplies two polynomials and puts the results in TermMalloc space.
*/

WORD *MULfunc(PHEAD WORD *p1, WORD *p2)
{
    WORD *prod,size1,size2,size3,*t,*tfill,*ps1,*ps2,sign1,sign2, error, *p3;
    UWORD *num1, *num2, *num3;
    int i;
    WORD oldsorttype = AR.SortType;
    void *oldcompareroutine = AR.CompareRoutine;
    AR.SortType = SORTHIGHFIRST;
    AR.CompareRoutine = (void *)&CompareSymbols;
    num3 = NumberMalloc("MULfunc");
    prod = TermMalloc("MULfunc");
    NewSort(BHEAD0);
    while ( *p1 ) {
        ps1 = p1+*p1; num1 = (UWORD *)(ps1 - ABS(ps1[-1])); size1 = ps1[-1];
        if ( size1 < 0 ) { sign1 = -1; size1 = -size1; }
        else               sign1 = 1;
        size1 = (size1-1)/2;
        p3 = p2;
        while ( *p3 ) {
            ps2 = p3+*p3; num2 = (UWORD *)(ps2 - ABS(ps2[-1])); size2 = ps2[-1];
            if ( size2 < 0 ) { sign2 = -1; size2 = -size2; }
            else               sign2 = 1;
            size2 = (size2-1)/2;
            if ( MulLong(num1,size1,num2,size2,num3,&size3) ) {
                error = 1;
CalledFrom:
                MLOCK(ErrorMessageLock);
                MesPrint(" Error %d",error);
                MesCall("MulFunc");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            tfill = prod+1;
            t = p1+1; while ( t < (WORD *)num1 ) *tfill++ = *t++;
            t = p3+1; while ( t < (WORD *)num2 ) *tfill++ = *t++;
            t = (WORD *)num3;
            for ( i = 0; i < size3; i++ ) *tfill++ = *t++;
            *tfill++ = 1;
            for ( i = 1; i < size3; i++ ) *tfill++ = 0;
            *tfill++ = (2*size3+1)*sign1*sign2;
```

```
            prod[0] = tfill - prod;
            if ( SymbolNormalize(prod) ) { error = 2; goto CalledFrom; }
            if ( StoreTerm(BHEAD prod) ) { error = 3; goto CalledFrom; }
            p3 += *p3;
        }
        p1 += *p1;
    }
    NumberFree(num3,"MULfunc");
    EndSort(BHEAD prod,1);
    AR.CompareRoutine = oldcompareroutine;
    AR.SortType = oldsorttype;
    return(prod);
}

/*
    #] MULfunc :
    #[ ConvertArgument :

    Converts an argument to a general notation in allocated space.
*/

WORD *ConvertArgument(PHEAD WORD *arg, int *type)
{
    WORD *output, *t, *r;
    int i, size;
    if ( *arg > 0 ) {
        output = (WORD *)Malloc1((*arg)*sizeof(WORD),"ConvertArgument");
        i = *arg - ARGHEAD; t = arg + ARGHEAD; r = output;
        NCOPY(r,t,i);
        *r = 0; *type = 0;
        return(output);
    }
    if ( *arg == -EXPRESSION ) {
        *type = 0;
        return(CreateExpression(BHEAD arg[1]));
    }
    if ( *arg == -DOLLAREXPRESSION ) {
        DOLLARS d;
        *type = 1;
        d = DolToTerms(BHEAD arg[1]);
/*
        The problem is that DolToTerms creates a copy of the dollar variable.
        If we just return d->where we create a memory leak. Hence we have to
        copy the contents of d->where to a new buffer
*/
        output = (WORD *)Malloc1((d->size+1)*sizeof(WORD),"Copy of dollar content");
        WCOPY(output,d->where,d->size+1);
        if ( d->factors ) { M_free(d->factors,"Dollar factors"); d->factors = 0; }
        M_free(d,"Copy of dollar variable");
        return(output);
    }
#if ( FUNHEAD > 4 )
    size = FUNHEAD+5;
#else
    size = 9;
#endif
    output = (WORD *)Malloc1(size*sizeof(WORD),"ConvertArgument");
    switch(*arg) {
        case -SYMBOL:
            output[0] = 8; output[1] = SYMBOL; output[2] = 4; output[3] = arg[1];
            output[4] = 1; output[5] = 1; output[6] = 1; output[7] = 3;
            output[8] = 0;
            break;
        case -INDEX:
        case -VECTOR:
        case -MINVECTOR:
            output[0] = 7; output[1] = INDEX; output[2] = 3; output[3] = arg[1];
            output[4] = 1; output[5] = 1;
            if ( *arg == -MINVECTOR ) output[6] = -3;
            else output[6] = 3;
            output[7] = 0;
            break;
        case -SNUMBER:
            output[0] = 4;
            if ( arg[1] < 0 ) {
                output[1] = -arg[1]; output[2] = 1; output[3] = -3;
            }
            else {
                output[1] =  arg[1]; output[2] = 1; output[3] =  3;
            }
            output[4] = 0;
            break;
        default:
            output[0] = FUNHEAD+4;
            output[1] = -*arg;
            output[2] = FUNHEAD;
            for ( i = 3; i <= FUNHEAD; i++ ) output[i] = 0;
```

```
            output[FUNHEAD+1] = 1;
            output[FUNHEAD+2] = 1;
            output[FUNHEAD+3] = 3;
            output[FUNHEAD+4] = 0;
            break;
    }
    *type = 0;
    return(output);
}

/*
    #] ConvertArgument :
    #[ ExpandRat :

    Expands the denominator of a PolyRatFun in the variable PolyFunVar.
    The output is a polyratfun with a single argument.
    In the case that there is a polyratfun with more than one argument
    or the dirtyflag is on, the argument(s) is/are normalized.
    The output overwrites the input.
*/

char *TheErrorMessage[] = {
     "PolyRatFun not of a type that FORM will expand: incorrect variable inside."
    ,"Division by zero in PolyRatFun encountered in ExpandRat."
    ,"Irregular code in PolyRatFun encountered in ExpandRat."
    ,"Called from ExpandRat."
    ,"WorkSpace overflow. Change parameter WorkSpace in setup file?"
    };

int ExpandRat(PHEAD WORD *fun)
{
    WORD *r, *rr, *rrr, *tt, *tnext, *arg1, *arg2, *rmin = 0, *rmininv;
    WORD *rcoef, rsize, rcopy, *ow = AT.WorkPointer;
    WORD *numerator, *denominator, *rnext;
    WORD *thecopy, *rc, ncoef, newcoef, *m, *mm, nco, *outarg = 0;
    UWORD co[2], co1[2], co2[2];
    WORD OldPolyFunPow = AR.PolyFunPow;
    int i, j, minpow = 0, eppow, first, error = 0, ipoly;
    if ( fun[1] == FUNHEAD ) { return(0); }
    tnext = fun + fun[1];
    if ( fun[1] == fun[FUNHEAD]+FUNHEAD ) { /* Single argument */
        if ( fun[2] == 0 ) { goto done; }
/*
        We have to normalize the argument. This could make it shorter.
*/
NormArg:;
        if ( outarg == 0 ) outarg = TermMalloc("ExpandRat")+ARGHEAD;
        AT.TrimPower = 1;
        NewSort(BHEAD0);
        r = fun+FUNHEAD+ARGHEAD;
        if ( AR.PolyFunExp ==  2 ) {    /* Find minimum power */
            WORD minpow2 = MAXPOWER, *rrm;
            rrm = r;
            while ( rrm < tnext ) {
                if ( *rrm == 4 ) {
                    if ( minpow2 > 0 ) minpow2 = 0;
                }
                else if ( ABS(rrm[*rrm-1]) == (*rrm-1) ) {
                    if ( minpow2 > 0 ) minpow2 = 0;
                }
                else {
                    if ( rrm[1] == SYMBOL && rrm[2] == 4 && rrm[3] == AR.PolyFunVar ) {
                        if ( rrm[4] < minpow2 ) minpow2 = rrm[4];
                    }
                    else {
                        MesPrint("Illegal term in expanded polyratfun.");
                        goto onerror;
                    }
                }
                rrm += *rrm;
            }
            AR.PolyFunPow += minpow2;
        }
        while ( r < tnext ) {
            rr = r + *r;
            i = *r; rrr = outarg; NCOPY(rrr,r,i);
            Normalize(BHEAD outarg);
            if ( *outarg > 0 ) StoreTerm(BHEAD outarg);
        }
        r = fun+FUNHEAD+ARGHEAD;
        EndSort(BHEAD r,1);
        AT.TrimPower = 0;
        if ( *r == 0 ) {
            fun[FUNHEAD] = -SNUMBER; fun[FUNHEAD+1] = 0;
            fun[1] = FUNHEAD+2;
        }
        else {
```

```
            rr = fun+FUNHEAD;
            if ( ToFast(rr,rr) ) {
                NEXTARG(rr); fun[1] = rr - fun;
            }
            else {
                while ( *r ) r += *r;
                *rr = r-rr; rr[1] = CLEANFLAG;
                fun[1] = r - fun;
            }
        }
        fun[2] = CLEANFLAG;
        goto done;
    }
/*
    First test whether we have only AR.PolyFunVar in the denominator
*/
    tt = fun + FUNHEAD;
    arg1 = arg2 = 0;
    if ( tt < tnext ) {
        arg1 = tt; NEXTARG(tt);
        if ( tt < tnext ) {
            arg2 = tt; NEXTARG(tt);
            if ( tt != tnext ) { arg1 = arg2 = 0; } /* more than two arguments */
        }
    }
    if ( arg2 == 0 ) {
        if ( *arg1 < 0 ) { fun[2] = CLEANFLAG; goto done; }
        if ( fun[2] == CLEANFLAG ) goto done;
        goto NormArg;   /* Note: should not come here */
    }
/*
    Produce the output argument in outarg
*/
    if ( outarg == 0 ) outarg = TermMalloc("ExpandRat")+ARGHEAD;

    if ( *arg2 <= 0 ) {
/*
        These cases are trivial.
        We try as much as possible to write the output directly into the
        function. We just have to be extremely careful not to overwrite
        relevant information before we are finished with it.
*/
        if ( *arg2 == -SYMBOL && arg2[1] == AR.PolyFunVar ) {
            rr = r = fun+FUNHEAD+ARGHEAD;
            if ( *arg1 < 0 ) {
                if ( *arg1 == -SYMBOL ) {
                    if ( arg1[1] == AR.PolyFunVar ) {
                        *r++ = 4; *r++ = 1; *r++ = 1; *r++ = 3; *r = 0;
                    }
                    else {
                        *r++ = 10; *r++ = SYMBOL; *r++ = 6;
                        *r++ = arg1[1]; *r++ = 1;
                        *r++ = AR.PolyFunVar; *r++ = -1;
                        *r++ = 1; *r++ = 1; *r++ = 3; *r = 0;
                        Normalize(BHEAD rr);
                    }
                }
                else if ( *arg1 == -SNUMBER ) {
                    nco = arg1[1];
                    if ( nco == 0 ) { *r++ = 0; }
                    else {
                        *r++ = 8; *r++ = SYMBOL; *r++ = 4;
                        *r++ = AR.PolyFunVar; *r++ = -1;
                        *r++ = ABS(nco); *r++ = 1;
                        if ( nco < 0 ) *r++ = -3;
                        else *r++ = 3;
                        *r = 0;
                    }
                }
                else { error = 2; goto onerror; }  /* should not happen! */
            }
            else {  /* Multi-term numerator. */
                m = arg1+ARGHEAD;
                NewSort(BHEAD0); /* Technically maybe not needed */
                if ( AR.PolyFunExp ==  2 ) {    /* Find minimum power */
                    WORD minpow2 = MAXPOWER, *rrm;
                    rrm = m;
                    while ( rrm < arg2 ) {
                        if ( *rrm == 4 ) {
                            if ( minpow2 > 0 ) minpow2 = 0;
                        }
                        else if ( ABS(rrm[*rrm-1]) == (*rrm-1) ) {
                            if ( minpow2 > 0 ) minpow2 = 0;
                        }
                        else {
                            if ( rrm[1] == SYMBOL && rrm[2] == 4 && rrm[3] == AR.PolyFunVar ) {
                                if ( rrm[4] < minpow2 ) minpow2 = rrm[4];
```

```
                        }
                        else {
                            MesPrint("Illegal term in expanded polyratfun.");
                            goto onerror;
                        }
                    }
                    rrm += *rrm;
                }
                AR.PolyFunPow += minpow2-1;
            }
            while ( m < arg2 ) {
                r = outarg;
                rrr = r++; mm = m + *m;
                *r++ = SYMBOL; *r++ = 4; *r++ = AR.PolyFunVar; *r++ = -1;
                m++; while ( m < mm ) *r++ = *m++;
                *rrr = r-rrr;
                Normalize(BHEAD rrr);
                StoreTerm(BHEAD rrr);
            }
            EndSort(BHEAD rr,1);
            r = rr; while ( *r ) r += *r;
        }
        if ( *rr == 0 ) {
            fun[FUNHEAD] = -SNUMBER; fun[FUNHEAD+1] = CLEANFLAG;
            fun[1] = FUNHEAD+2;
        }
        else {
            rr = fun+FUNHEAD;
            *rr = r-rr;
            rr[1] = CLEANFLAG;
            if ( ToFast(rr,rr) ) {
                NEXTARG(rr);
                fun[1] = rr - fun;
            }
            else { fun[1] = r - fun; }
        }
        fun[2] = CLEANFLAG;
        goto done;
    }
    else if ( *arg2 == -SNUMBER ) {
        rr = r = outarg;
        if ( arg2[1] == 0 ) { error = 1; goto onerror; }
        if ( *arg1  == -SNUMBER ) { /* Things may not be normalized */
            if ( arg1[1] == 0 ) { *r++ = 0; }
            else {
                co1[0] = ABS(arg1[1]); co1[1] = 1;
                co2[0] = 1; co2[1] = ABS(arg2[1]);
                MulRat(BHEAD co1,1,co2,1,co,&nco);
                *r++ = 4; *r++ = (WORD)(co[0]); *r++ = (WORD)(co[1]);
                if ( ( arg1[1] < 0 && arg2[1] > 0 ) ||
                     ( arg1[1] > 0 && arg2[1] < 0 ) ) *r++ = -3;
                else *r++ = 3;
                *r = 0;
            }
        }
        else if ( *arg1 == -SYMBOL ) {
            *r++ = 8; *r++ = SYMBOL; *r++ = 4;
            *r++ = arg1[1]; *r++ = 1;
            *r++ = 1; *r++ = ABS(arg2[1]);
            if ( arg2[1] < 0 ) *r++ = -3;
            else *r++ = 3;
            *r = 0;
        }
        else if ( *arg1 < 0 ) { error = 2; goto onerror; }
        else {  /* Multi-term numerator. */
            m = arg1+ARGHEAD;
            NewSort(BHEAD0); /* Technically maybe not needed */
            if ( AR.PolyFunExp ==  2 ) {    /* Find minimum power */
                WORD minpow2 = MAXPOWER, *rrm;
                rrm = m;
                while ( rrm < arg2 ) {
                    if ( *rrm == 4 ) {
                        if ( minpow2 > 0 ) minpow2 = 0;
                    }
                    else if ( ABS(rrm[*rrm-1]) == (*rrm-1) ) {
                        if ( minpow2 > 0 ) minpow2 = 0;
                    }
                    else {
                        if ( rrm[1] == SYMBOL && rrm[2] == 4 && rrm[3] == AR.PolyFunVar ) {
                            if ( rrm[4] < minpow2 ) minpow2 = rrm[4];
                        }
                        else {
                            MesPrint("Illegal term in expanded polyratfun.");
                            goto onerror;
                        }
                    }
                    rrm += *rrm;
```

```
                    }
                    AR.PolyFunPow += minpow2;
                }
                while ( m < arg2 ) {
                    r = rr;
                    rrr = r++; mm = m + *m;
                    *r++ = DENOMINATOR; *r++ = FUNHEAD + 2; *r++ = DIRTYFLAG;
                    FILLFUN3(r);
                    *r++ = arg2[0]; *r++ = arg2[1];
                    m++; while ( m < mm ) *r++ = *m++;
                    *rrr = r-rrr;
                    if ( r < AT.WorkTop && r >= AT.WorkSpace )
                                AT.WorkPointer = r;
                    Normalize(BHEAD rrr);
                    if ( ABS(rrr[*rrr-1]) == *rrr-1 ) {
                        if ( AR.PolyFunPow >= 0 ) {
                            StoreTerm(BHEAD rrr);
                        }
                    }
                    else if ( rrr[1] == SYMBOL && rrr[2] == 4 &&
                    rrr[3] == AR.PolyFunVar && rrr[4] <= AR.PolyFunPow ) {
                        StoreTerm(BHEAD rrr);
                    }
                }
                EndSort(BHEAD rr,1);
            }
            r = rr; while ( *r ) r += *r;
            i = r-rr;
            r = fun + FUNHEAD + ARGHEAD;
            NCOPY(r,rr,i);
            rr = fun + FUNHEAD;
            *rr = r - rr; rr[1] = CLEANFLAG;
            if ( ToFast(rr,rr) ) {
                NEXTARG(rr);
                fun[1] = rr - fun;
            }
            else { fun[1] = r - fun; }
            fun[2] = CLEANFLAG;
            goto done;
        }
        else { error = 0; goto onerror; }
    }
    else {
        r = arg2+ARGHEAD; /* The argument ends at tnext */
        first = 1;
        while ( r < tnext ) {
            rr = r + *r; rr -= ABS(rr[-1]);
            if ( r+1 == rr ) {
                if ( first ) { minpow = 0; first = 0; rmin = r; }
                else if ( minpow > 0 ) { minpow = 0; rmin = r; }
            }
            else if ( r[1] != SYMBOL || r[2] != 4 || r[3] != AR.PolyFunVar
                || r[4] > MAXPOWER ) { error = 0; goto onerror; }
            else if ( first ) { minpow = r[4]; first = 0; rmin = r; }
            else if ( r[4] < minpow ) { minpow = r[4]; rmin = r; }
            r += *r;
        }
/*
        We have now:
        1: a numerator in arg1 which can contain several variables.
        2: a denominator in arg2 with at most only AR.PolyFunVar (ep).
        3: the minimum power in the denominator is minpow and the
           term with that minimum power is in rmin.
        Divide numerator and denominator by this minimum power.
        Determine the power range in the numerator.
        Call InvPoly.
        Multiply by the inverse in such a way that we never take more
        powers of ep than necessary.
*/
/*
        One: put 1/rmin in AT.WorkPointer -> rmininv
*/
        AT.WorkPointer += AM.MaxTer/sizeof(WORD);
        if ( AT.WorkPointer + (AM.MaxTer/sizeof(WORD)) >= AT.WorkTop ) {
            error = 4; goto onerror;
        }
        rmininv = r = AT.WorkPointer;
        rr = rmin; i = *rmin; NCOPY(r,rr,i)
        if ( minpow != 0 ) { rmininv[4] = -rmininv[4]; }
        rsize = ABS(r[-1]);
        rcoef = r - rsize;
        rsize = (rsize-1)/2; rr = rcoef + rsize;
        for ( i = 0; i < rsize; i++ ) {
            rcopy = rcoef[i]; rcoef[i] = rr[i]; rr[i] = rcopy;
        }
        AT.WorkPointer = r;
        if ( *arg1 < 0 ) {
```

```
            ToGeneral(arg1,r,0);
            arg1 = r; r += *r; *r++ = 0; rcopy = 0;
            AT.WorkPointer = r;
        }
        else {
            r = arg1 + *arg1;
            rcopy = *r; *r++ = 0;
        }
/*
        We can use MultiplyWithTerm.
*/
        AT.LeaveNegative = 1;
        numerator = MultiplyWithTerm(BHEAD arg1+ARGHEAD,rmininv,0);
        AT.LeaveNegative = 0;
        r[-1] = rcopy;
        r = numerator; while ( *r ) r += *r;
        AT.WorkPointer = r+1;
        rcopy = arg2[*arg2]; arg2[*arg2] = 0;
        denominator = MultiplyWithTerm(BHEAD arg2+ARGHEAD,rmininv,1);
        arg2[*arg2] = rcopy;
        r = denominator; while ( *r ) r += *r;
        AT.WorkPointer = r+1;
/*
        Now find the minimum power of ep in the numerator.
*/
        r = numerator;
        first = 1;
        while ( *r ) {
            rr = r + *r; rr -= ABS(rr[-1]);
            if ( r+1 == rr ) {
                if ( first ) { minpow = 0; first = 0; }
                else if ( minpow > 0 ) { minpow = 0; }
            }
            else if ( r[1] != SYMBOL ) { error = 0; goto onerror; }
            else {
                for ( i = 3; i < r[2]; i += 2 ) {
                    if ( r[i] == AR.PolyFunVar ) {
                        if ( first ) { minpow = r[i+1]; first = 0; }
                        else if ( r[i+1] < minpow ) minpow = r[i+1];
                        break;
                    }
                }
                if ( i >= r[2] ) {
                    if ( first ) { minpow = 0; first = 0; }
                    else if ( minpow > 0 ) minpow = 0;
                }
            }
            r += *r;
        }
/*
        We can invert the denominator.
        Note that the return value is an offset in AT.pWorkSpace.
        Hence there is no need to free memory afterwards.
*/
        if ( AR.PolyFunExp == 3 ) {
            ipoly = InvPoly(BHEAD denominator,AR.PolyFunPow-minpow,AR.PolyFunVar);
        }
        else {
            ipoly = InvPoly(BHEAD denominator,AR.PolyFunPow,AR.PolyFunVar);
        }
/*
        Now we start the multiplying
*/
        NewSort(BHEAD0);
        r = numerator;
        while ( *r ) {
/*
            1: Find power of ep.
*/
            rnext = r + *r;
            rrr = rnext - ABS(rnext[-1]);
            rr = r+1;
            eppow = 0;
            if ( rr < rrr ) {
                j = rr[1] - 2; rr += 2;
                while ( j > 0 ) {
                    if ( *rr == AR.PolyFunVar ) { eppow = rr[1]; break; }
                    j -= 2; rr += 2;
                }
            }
/*
            2: Multiply by the proper terms in ipoly
*/
            for ( i = 0; i <= AR.PolyFunPow-eppow+minpow; i++ ) {
                if ( AT.pWorkSpace[ipoly+i] == 0 ) continue;
/*
                Copy the term, add i to the power of ep and multiply coef.
```

```
*/
                rc = r;
                rr = thecopy = AT.WorkPointer;
                while ( rc < rrr ) *rr++ = *rc++;
                if ( i != 0 ) {
                    *rr++ = SYMBOL; *rr++ = 4; *rr++ = AR.PolyFunVar; *rr++ = i;
                }
                ncoef = REDLENG(rnext[-1]);
                MulRat(BHEAD (UWORD *)rrr,ncoef,
                    (UWORD *)(AT.pWorkSpace[ipoly+i])+1,AT.pWorkSpace[ipoly+i][0]
                    ,(UWORD *)rr,&newcoef);
                ncoef = ABS(newcoef); rr += 2*ncoef;
                newcoef = INCLENG(newcoef);
                *rr++ = newcoef;
                *thecopy = rr - thecopy;
                AT.WorkPointer = rr;
                Normalize(BHEAD thecopy);
                if ( *thecopy > 0 ) StoreTerm(BHEAD thecopy);
                AT.WorkPointer = thecopy;
            }
            r = rnext;
        }
/*
        Now we have all.
*/
        rr = fun + FUNHEAD; r = rr + ARGHEAD;
        EndSort(BHEAD r,1);
        if ( *r == 0 ) {
            fun[1] = FUNHEAD+2; fun[2] = CLEANFLAG;
            fun[FUNHEAD] = -SNUMBER; fun[FUNHEAD+1] = 0;
        }
        else {
            while ( *r ) r += *r;
            rr[0] = r-rr; rr[1] = CLEANFLAG;
            if ( ToFast(rr,rr) ) { NEXTARG(rr); fun[1] = rr-fun; }
            else { fun[1] = r-fun; }
            fun[2] = CLEANFLAG;
        }
    }
done:
    if ( outarg ) TermFree(outarg-ARGHEAD,"ExpandRat");
    AR.PolyFunPow = OldPolyFunPow;
    AT.WorkPointer = ow;
    AN.PolyNormFlag = 1;
    return(0);
onerror:
    if ( outarg ) TermFree(outarg-ARGHEAD,"ExpandRat");
    AR.PolyFunPow = OldPolyFunPow;
    AT.WorkPointer = ow;
    MLOCK(ErrorMessageLock);
    MesPrint(TheErrorMessage[error]);
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(-1);
}

/*
    #] ExpandRat :
    #[ InvPoly :

    The input polynomial is represented as a sequence of terms in ascending
    power. The first coefficient is 1. If we call this 1-a and
    a = sum_(j,1,n,x^j*a(j)), and b = 1/(1-a) we can find the coefficients
    of b with the recursion
        b(0) = 1, b(n) = sum_(j,1,n,a(j)*b(n-j))
    The variable is the symbol sym and we need maxpow powers in the answer.
    The answer is an array of pointers to the coefficients of the various
    powers as rational numbers in the notation signedsize,numerator,denominator
    We put these powers in the workspace and the answer is in AT.pWorkSpace.
    Hence the return value is an offset in the pWorkSpace.
    A zero pointer indicates that this coefficient is zero.
*/

int InvPoly(PHEAD WORD *inpoly, WORD maxpow, WORD sym)
{
    int needed, inpointers, outpointers, maxinput = 0, i, j;
    WORD *t, *tt, *ttt, *w, *c, *cc, *ccc, lenc, lenc1, lenc2, rc, *c1, *c2;
/*
    Step 0: allocate the space
*/
    needed = (maxpow+1)*2;
    WantAddPointers(needed);
    inpointers = AT.pWorkPointer;
    outpointers = AT.pWorkPointer+maxpow+1;
    for ( i = 0; i < needed; i++ ) AT.pWorkSpace[inpointers+i] = 0;
/*
    Step 1: determine the coefficients in inpoly
```

```
            often there is a maximum power that is much smaller than maxpow.
            keeping track of this can speed up things.
*/
    t = inpoly;
    w = AT.WorkPointer;
    while ( *t ) {
        if ( *t == 4 ) {
            if ( t[1] != 1 || t[2] != 1 || t[3] != 3 ) goto onerror;
            AT.pWorkSpace[inpointers] = 0;
        }
        else if ( t[1] != SYMBOL || t[2] != 4 || t[3] != sym || t[4] < 0 ) goto onerror;
        else if ( t[4] > maxpow ) {} /* power outside useful range */
        else {
            if ( t[4] > maxinput ) maxinput = t[4];
            AT.pWorkSpace[inpointers+t[4]] = w;
            tt = t + *t; rc = -*--tt; /* we need - the coefficient! */
            rc = REDLENG(rc); *w++ = rc;
            ttt = t+5;
            while ( ttt < tt ) *w++ = *ttt++;
        }
        t += *t;
    }
/*
    Step 2: compute the output. b(0) = 1.
    then the recursion starts.
*/
    AT.pWorkSpace[outpointers] = w;
    *w++ = 1; *w++ = 1; *w++ = 1;
    c  = TermMalloc("InvPoly");
    c1 = TermMalloc("InvPoly");
    c2 = TermMalloc("InvPoly");
    for ( j = 1; j <= maxpow; j++ ) {
/*
        Start at c = a(j)*b(0) = a(j)
*/
        if ( ( cc = AT.pWorkSpace[inpointers+j] ) != 0 ) {
            lenc = *cc++; /* reduced length */
            i = 2*ABS(lenc); ccc = c;
            NCOPY(ccc,cc,i);
        }
        else { lenc = 0; }
        for ( i = MiN(j-1,maxinput); i > 0; i-- ) {
/*
            c -> c + a(i)*b(j-i)
*/
            if ( AT.pWorkSpace[inpointers+i] == 0
                 || AT.pWorkSpace[outpointers+j-i] == 0 ) {
            }
            else {
                if ( MulRat(BHEAD (UWORD *)(AT.pWorkSpace[inpointers+i]+1),AT.pWorkSpace[inpointers+i][0],
                 (UWORD *)(AT.pWorkSpace[outpointers+j-i]+1),AT.pWorkSpace[outpointers+j-i][0],
                 (UWORD *)c1,&lenc1) ) goto calcerror;
                if ( lenc == 0 ) {
                    cc = c; c = c1; c1 = cc;
                    lenc = lenc1;
                }
                else {
                    if ( AddRat(BHEAD (UWORD *)c,lenc,(UWORD *)c1,lenc1,(UWORD *)c2,&lenc2) )
                        goto calcerror;
                    cc = c; c = c2; c2 = cc;
                    lenc = lenc2;
                }
            }
        }
/*
        Copy c to the proper location
*/
        if ( lenc == 0 ) AT.pWorkSpace[outpointers+j] = 0;
        else {
            AT.pWorkSpace[outpointers+j] = w;
            *w++ = lenc;
            i = 2*ABS(lenc); ccc = c;
            NCOPY(w,ccc,i);
        }
    }
    AT.WorkPointer = w;
    TermFree(c2,"InvPoly");
    TermFree(c1,"InvPoly");
    TermFree(c ,"InvPoly");

    return(outpointers);
onerror:
    MLOCK(ErrorMessageLock);
    MesPrint("Incorrect symbol field in InvPoly.");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
```

```
    return(-1);
calcerror:
    MLOCK(ErrorMessageLock);
    MesPrint("Called from InvPoly.");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(-1);
}

/*
    #] InvPoly :
*/
```

4.90 reken.c File Reference

```
#include "form3.h"
#include <math.h>
```

Include dependency graph for reken.c:

Macros

- #define GCDMAX 3
- #define NEWTRICK 1
- #define COPYLONG(x1, nx1, x2, nx2) { int i; for(i=0;i<ABS(nx2);i++)x1[i]=x2[i];nx1=nx2; }
- #define WARMUP 6

Functions

- VOID Pack (UWORD *a, WORD *na, UWORD *b, WORD nb)
- VOID UnPack (UWORD *a, WORD na, WORD *denom, WORD *numer)
- WORD Mully (PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
- WORD Divvy (PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
- WORD AddRat (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- WORD MulRat (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- WORD DivRat (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- WORD Simplify (PHEAD UWORD *a, WORD *na, UWORD *b, WORD *nb)
- WORD AccumGCD (PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
- int TakeRatRoot (UWORD *a, WORD *n, WORD power)
- WORD AddLong (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- WORD AddPLon (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- VOID SubPLon (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- WORD MulLong (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- WORD BigLong (UWORD *a, WORD na, UWORD *b, WORD nb)

- WORD DivLong (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc, UWORD *d, WORD *nd)
- WORD RaisPow (PHEAD UWORD *a, WORD *na, UWORD b)
- VOID RaisPowCached (PHEAD WORD x, WORD n, UWORD **c, WORD *nc)
- WORD RaisPowMod (WORD x, WORD n, WORD m)
- int NormalModulus (UWORD *a, WORD *na)
- int MakeInverses ()
- int GetModInverses (WORD m1, WORD m2, WORD *im1, WORD *im2)
- int GetLongModInverses (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *ia, WORD *nia, UWORD *ib, WORD *nib)
- WORD Product (UWORD *a, WORD *na, WORD b)
- UWORD Quotient (UWORD *a, WORD *na, WORD b)
- WORD Remain10 (UWORD *a, WORD *na)
- WORD Remain4 (UWORD *a, WORD *na)
- VOID PrtLong (UWORD *a, WORD na, UBYTE *s)
- WORD GetLong (UBYTE *s, UWORD *a, WORD *na)
- WORD GcdLong (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- WORD GetBinom (UWORD *a, WORD *na, WORD i1, WORD i2)
- WORD LcmLong (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int TakeLongRoot (UWORD *a, WORD *n, WORD power)
- int MakeRational (WORD a, WORD m, WORD *b, WORD *c)
- int MakeLongRational (PHEAD UWORD *a, WORD na, UWORD *m, WORD nm, UWORD *b, WORD *nb)
- WORD CompCoef (WORD *term1, WORD *term2)
- WORD Modulus (WORD *term)
- WORD TakeModulus (UWORD *a, WORD *na, UWORD *cmodvec, WORD ncmod, WORD par)
- WORD TakeNormalModulus (UWORD *a, WORD *na, UWORD *c, WORD nc, WORD par)
- WORD MakeModTable ()
- int Factorial (PHEAD WORD n, UWORD *a, WORD *na)
- int Bernoulli (WORD n, UWORD *a, WORD *na)
- WORD NextPrime (PHEAD WORD num)
- void iniwranf (PHEAD0)
- UWORD wranf (PHEAD0)
- UWORD iranf (PHEAD UWORD imax)
- UBYTE * PreRandom (UBYTE *s)

4.90.1 Detailed Description

This file contains the numerical routines. The arithmetic in FORM is normally over the rational numbers. Hence there are routines for dealing with integers and with rational of 'arbitrary precision' (within limits) There are also routines for that calculus modulus an integer. In addition there are the routines for factorials and bernoulli numbers. The random number function is currently only for internal purposes.

Definition in file reken.c.

4.90.2 Macro Definition Documentation

4.90.2.1 GCDMAX

```
#define GCDMAX 3
```

Definition at line 49 of file reken.c.

4.90.2.2 NEWTRICK

```
#define NEWTRICK 1
```

Definition at line 51 of file reken.c.

4.90.2.3 COPYLONG

```
#define COPYLONG(
              x1,
              nx1,
              x2,
              nx2 ) { int i; for(i=0;i<ABS(nx2);i++)x1[i]=x2[i];nx1=nx2; }
```

Definition at line 2891 of file reken.c.

4.90.2.4 WARMUP

```
#define WARMUP 6
```

Definition at line 3726 of file reken.c.

4.90.3 Function Documentation

4.90.3.1 Pack()

```
VOID Pack (
            UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )
```

Definition at line 62 of file reken.c.

4.90.3.2 UnPack()

```
VOID UnPack (
            UWORD * a,
            WORD na,
            WORD * denom,
            WORD * numer )
```

Definition at line 100 of file reken.c.

4.90.3.3 Mully()

```
WORD Mully (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )
```

Definition at line 126 of file reken.c.

4.90.3.4 Divvy()

```
WORD Divvy (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )
```

Definition at line 172 of file reken.c.

4.90.3.5 AddRat()

```
WORD AddRat (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )
```

Definition at line 210 of file reken.c.

4.90.3.6 MulRat()

```
WORD MulRat (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )
```

Definition at line 378 of file reken.c.

4.90.3.7 DivRat()

```
WORD DivRat (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )
```

Definition at line 500 of file reken.c.

4.90.3.8 Simplify()

```
WORD Simplify (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD * nb )
```

Definition at line 530 of file reken.c.

4.90.3.9 AccumGCD()

```
WORD AccumGCD (
            PHEAD UWORD * a,
            WORD * na,
            UWORD * b,
            WORD nb )
```

Definition at line 647 of file reken.c.

4.90.3.10 TakeRatRoot()

```
int TakeRatRoot (
            UWORD * a,
            WORD * n,
            WORD power )
```

Definition at line 680 of file reken.c.

4.90.3.11 AddLong()

```
WORD AddLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )
```

Definition at line 705 of file reken.c.

4.90.3.12 AddPLon()

```
WORD AddPLon (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )
```

Definition at line 750 of file reken.c.

4.90.3.13 SubPLon()

```
VOID SubPLon (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )
```

Definition at line 803 of file reken.c.

4.90.3.14 MulLong()

```
WORD MulLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )
```

Definition at line 841 of file reken.c.

4.90.3.15 BigLong()

```
WORD BigLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb )
```

Definition at line 944 of file reken.c.

4.90.3.16 DivLong()

```
WORD DivLong (
            UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc,
            UWORD * d,
            WORD * nd )
```

Definition at line 972 of file reken.c.

4.90.3.17 RaisPow()

```
WORD RaisPow (
            PHEAD UWORD * a,
            WORD * na,
            UWORD b )
```

Definition at line 1218 of file reken.c.

4.90.3.18 RaisPowCached()

```
VOID RaisPowCached (
            PHEAD WORD x,
            WORD n,
            UWORD ** c,
            WORD * nc )
```

Computes power x^n and caches the value

4.90.4 Description

Calculates the power x^n and stores the results for caching purposes. The pointer c (i.e., the pointer, and not what it points to) is overwritten. What it points to should not be overwritten in the calling function.

4.90.5 Notes

- Caching is done in AT.small_power[]. This array is extended if necessary.

Definition at line 1286 of file reken.c.

4.90.5.1 RaisPowMod()

```
WORD RaisPowMod (
            WORD x,
            WORD n,
            WORD m )
```

Definition at line 1373 of file reken.c.

4.90.5.2 NormalModulus()

```
int NormalModulus (
            UWORD * a,
            WORD * na )
```

Brings a modular representation in the range -p/2 to +p/2 The return value tells whether anything was done. Routine made in the general modulus revamp of July 2008 (JV).

Definition at line 1393 of file reken.c.

Referenced by AddCoef(), and MergePatches().

4.90.5.3 MakeInverses()

```
int MakeInverses ( )
```

Makes a table of inverses in modular calculus The modulus is in AC.cmod and AC.ncmod One should notice that the table of inverses can only be made if the modulus fits inside a single FORM word. Otherwise the table lookup becomes too difficult and the table too long.

Definition at line 1430 of file reken.c.

References GetModInverses().

Here is the call graph for this function:

4.90.5.4 GetModInverses()

```
int GetModInverses (
            WORD m1,
            WORD m2,
            WORD * im1,
            WORD * im2 )
```

Input m1 and m2, which are relative prime. determines a∗m1+b∗m2 = 1 (and 1 is the gcd of m1 and m2) then a∗m1 = 1 mod m2 and hence im1 = a. and b∗m2 = 1 mod m1 and hence im2 = b. Set m1 = 0∗m1+1∗m2 = a1∗m1+b1∗m2 m2 = 1∗m1+0∗m2 = a2∗m1+b2∗m2 If everything is OK, the return value is zero

Definition at line 1466 of file reken.c.

Referenced by MakeDollarMod(), MakeInverses(), MakeMod(), TakeContent(), and TakeSymbolContent().

4.90.5.5 GetLongModInverses()

```
int GetLongModInverses (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * ia,
            WORD * nia,
            UWORD * ib,
            WORD * nib )
```

Definition at line 1498 of file reken.c.

4.90.5.6 Product()

```
WORD Product (
            UWORD * a,
            WORD * na,
            WORD b )
```

Definition at line 1575 of file reken.c.

4.90.5.7 Quotient()

```
UWORD Quotient (
            UWORD * a,
            WORD * na,
            WORD b )
```

Definition at line 1610 of file reken.c.

4.90.5.8 Remain10()

```
WORD Remain10 (
            UWORD * a,
            WORD * na )
```

Definition at line 1649 of file reken.c.

4.90.5.9 Remain4()

```
WORD Remain4 (
            UWORD * a,
            WORD * na )
```

Definition at line 1676 of file reken.c.

4.90.5.10 PrtLong()

```
VOID PrtLong (
            UWORD * a,
            WORD na,
            UBYTE * s )
```

Definition at line 1702 of file reken.c.

4.90.5.11 GetLong()

```
WORD GetLong (
            UBYTE * s,
            UWORD * a,
            WORD * na )
```

Definition at line 1764 of file reken.c.

4.90.5.12 GcdLong()

```
WORD GcdLong (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )
```

Definition at line 2266 of file reken.c.

4.90.5.13 GetBinom()

```
WORD GetBinom (
            UWORD * a,
            WORD * na,
            WORD i1,
            WORD i2 )
```

Definition at line 2657 of file reken.c.

4.90.5.14 LcmLong()

```
WORD LcmLong (
            PHEAD UWORD * a,
            WORD na,
            UWORD * b,
            WORD nb,
            UWORD * c,
            WORD * nc )
```

Definition at line 2691 of file reken.c.

4.90.5.15 TakeLongRoot()

```
int TakeLongRoot (
            UWORD * a,
            WORD * n,
            WORD power )
```

Definition at line 2725 of file reken.c.

4.90.5.16 MakeRational()

```
int MakeRational (
            WORD a,
            WORD m,
            WORD * b,
            WORD * c )
```

Definition at line 2845 of file reken.c.

4.90.5.17 MakeLongRational()

```
int MakeLongRational (
            PHEAD UWORD * a,
            WORD na,
            UWORD * m,
            WORD nm,
            UWORD * b,
            WORD * nb )
```

Definition at line 2893 of file reken.c.

4.90.5.18 CompCoef()

```
WORD CompCoef (
            WORD * term1,
            WORD * term2 )
```

```
Routine takes a1 mod m1 and a2 mod m2 and returns a mod m1*m2 with
a mod m1 = a1 and a mod m2 = a2
```

Chinese remainder: a%(m1∗m2) = q1∗m1+a1 a%(m1∗m2) = q2∗m2+a2 Compute n1 such that (n1∗m1)m2 is one Compute n2 such that (n2∗m2)m1 is one Then (a1∗n2∗m2+a2∗n1∗m1)%(m1∗m2) is a%(m1∗m2)

Definition at line 3037 of file reken.c.

Referenced by Compare1().

4.90.5.19 Modulus()

```
WORD Modulus (
            WORD * term )
```

Definition at line 3092 of file reken.c.

4.90.5.20 TakeModulus()

```
WORD TakeModulus (
            UWORD * a,
            WORD * na,
            UWORD * cmodvec,
            WORD ncmod,
            WORD par )
```

Definition at line 3139 of file reken.c.

4.90.5.21 TakeNormalModulus()

```
WORD TakeNormalModulus (
            UWORD * a,
            WORD * na,
            UWORD * c,
            WORD nc,
            WORD par )
```

Definition at line 3311 of file reken.c.

4.90.5.22 MakeModTable()

```
WORD MakeModTable ( )
```

Definition at line 3353 of file reken.c.

4.90.5.23 Factorial()

```
int Factorial (
            PHEAD WORD n,
            UWORD * a,
            WORD * na )
```

Definition at line 3439 of file reken.c.

4.90.5.24 Bernoulli()

```
int Bernoulli (
            WORD n,
            UWORD * a,
            WORD * na )
```

Definition at line 3519 of file reken.c.

4.90.5.25 NextPrime()

```
WORD NextPrime (
            PHEAD WORD num )
```

Gives the next prime number in the list of prime numbers.

If the list isn't long enough we expand it. For ease in ParForm and because these lists shouldn't be very big we let each worker keep its own list.

The list is cut off at MAXPOWER, because we don't want to get into trouble that the power of a variable gets larger than the prime number.

Definition at line 3654 of file reken.c.

4.90.5.26 iniwranf()

```
void iniwranf (
            PHEAD0  )
```

Definition at line 3744 of file reken.c.

4.90.5.27 wranf()

```
UWORD wranf (
            PHEAD0  )
```

Definition at line 3793 of file reken.c.

4.90.5.28 iranf()

```
UWORD iranf (
            PHEAD UWORD imax )
```

Definition at line 3812 of file reken.c.

4.90.5.29 PreRandom()

```
UBYTE * PreRandom (
            UBYTE * s )
```

Definition at line 3834 of file reken.c.

4.91 reken.c

Go to the documentation of this file.

```
00001
00011 /* #[ License : */
00012 /*
00013  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00014  *   When using this file you are requested to refer to the publication
00015  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016  *   This is considered a matter of courtesy as the development was paid
00017  *   for by FOM the Dutch physics granting agency and we would like to
00018  *   be able to track its scientific use to convince FOM of its value
00019  *   for the community.
00020  *
00021  *   This file is part of FORM.
00022  *
00023  *   FORM is free software: you can redistribute it and/or modify it under the
00024  *   terms of the GNU General Public License as published by the Free Software
00025  *   Foundation, either version 3 of the License, or (at your option) any later
00026  *   version.
00027  *
00028  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00031  *   details.
00032  *
00033  *   You should have received a copy of the GNU General Public License along
00034  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00035  */
00036 /* #] License : */
00037 /*
00038     #[ Includes : reken.c
```

```
*/

#include "form3.h"
#include <math.h>

#ifdef WITHGMP
#include <gmp.h>
#define GMPSPREAD (GMP_LIMB_BITS/BITSINWORD)
#endif

#define GCDMAX 3

#define NEWTRICK 1
/*
    #] Includes :
    #[ RekenRational :
        #[ Pack :           VOID Pack(a,na,b,nb)

    Packs the contents of the numerator a and the denominator b into
    one normalized fraction a.

*/

VOID Pack(UWORD *a, WORD *na, UWORD *b, WORD nb)
{
    WORD c, sgn = 1, i;
    UWORD *to,*from;
    if ( (c = *na) == 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Caught a zero in Pack");
        MUNLOCK(ErrorMessageLock);
        return;
    }
    if ( nb == 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Division by zero in Pack");
        MUNLOCK(ErrorMessageLock);
        return;
    }
    if ( *na < 0 ) { sgn = -sgn; c = -c; }
    if ( nb < 0 ) { sgn = -sgn; nb = -nb; }
    *na = MaX(c,nb);
    to = a + c;
    i = *na - c;
    while ( --i >= 0 ) *to++ = 0;
    i = *na - nb;
    from = b;
    NCOPY(to,from,nb);
    while ( --i >= 0 ) *to++ = 0;
    if ( sgn < 0 ) *na = -*na;
}

/*
        #] Pack :
        #[ UnPack :         VOID UnPack(a,na,denom,numer)

    Determines the sizes of the numerator and the denominator in the
    normalized fraction a with length na.

*/

VOID UnPack(UWORD *a, WORD na, WORD *denom, WORD *numer)
{
    UWORD *pos;
    WORD i, sgn = na;
    if ( na < 0 ) { na = -na; }
    i = na;
    if ( i > 1 ) {      /* Find the respective leading words */
        a += i;
        a--;
        pos = a + i;
        while ( !(*a) ) { i--; a--; }
        while ( !(*pos) ) { na--; pos--; }
    }
    *denom = na;
    if ( sgn < 0 ) i = -i;
    *numer = i;
}

/*
        #] UnPack :
        #[ Mully :          WORD Mully(a,na,b,nb)

    Multiplies the rational a by the Long b.

*/

```

```
WORD Mully(PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
{
    GETBIDENTITY
    UWORD *d, *e;
    WORD i, sgn = 1;
    WORD nd, ne, adenom, anumer;
    if ( !nb ) { *na = 0; return(0); }
    else if ( *b == 1 ) {
        if ( nb == 1 ) return(0);
        else if ( nb == -1 ) { *na = -*na; return(0); }
    }
    if ( *na < 0 ) { sgn = -sgn; *na = -*na; }
    if ( nb < 0 ) { sgn = -sgn; nb = -nb; }
    UnPack(a,*na,&adenom,&anumer);
    d = NumberMalloc("Mully"); e = NumberMalloc("Mully");
    for ( i = 0; i < nb; i++ ) { e[i] = *b++; }
    ne = nb;
    if ( Simplify(BHEAD a+*na,&adenom,e,&ne) ) goto MullyEr;
    if ( MulLong(a,anumer,e,ne,d,&nd) ) goto MullyEr;
    b = a+*na;
    for ( i = 0; i < *na; i++ ) { e[i] = *b++; }
    ne = adenom;
    *na = nd;
    b = d;
    *na = nd;
    for ( i = 0; i < *na; i++ ) { a[i] = *b++; }
    Pack(a,na,e,ne);
    if ( sgn < 0 ) *na = -*na;
    NumberFree(d,"Mully"); NumberFree(e,"Mully");
    return(0);
MullyEr:
    MLOCK(ErrorMessageLock);
    MesCall("Mully");
    MUNLOCK(ErrorMessageLock);
    NumberFree(d,"Mully"); NumberFree(e,"Mully");
    SETERROR(-1)
}

/*
        #] Mully :
        #[ Divvy :          WORD Divvy(a,na,b,nb)

    Divides the rational a by the Long b.

*/

WORD Divvy(PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
{
    GETBIDENTITY
    UWORD *d,*e;
    WORD i, sgn = 1;
    WORD nd, ne, adenom, anumer;
    if ( !nb ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Division by zero in Divvy");
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    d = NumberMalloc("Divvy"); e = NumberMalloc("Divvy");
    if ( nb < 0 ) { sgn = -sgn; nb = -nb; }
    if ( *na < 0 ) { sgn = -sgn; *na = -*na; }
    UnPack(a,*na,&adenom,&anumer);
    for ( i = 0; i < nb; i++ ) { e[i] = *b++; }
    ne = nb;
    if ( Simplify(BHEAD a,&anumer,e,&ne) ) goto DivvyEr;
    if ( MulLong(a+*na,adenom,e,ne,d,&nd) ) goto DivvyEr;
    *na = anumer;
    Pack(a,na,d,nd);
    if ( sgn < 0 ) *na = -*na;
    NumberFree(d,"Divvy"); NumberFree(e,"Divvy");
    return(0);
DivvyEr:
    MLOCK(ErrorMessageLock);
    MesCall("Divvy");
    MUNLOCK(ErrorMessageLock);
    NumberFree(d,"Divvy"); NumberFree(e,"Divvy");
    SETERROR(-1)
}

/*
        #] Divvy :
        #[ AddRat :         WORD AddRat(a,na,b,nb,c,nc)
*/

WORD AddRat(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    GETBIDENTITY
```

```
    UWORD *d, *e, *f, *g;
    WORD nd, ne, nf, ng, adenom, anumer, bdenom, bnumer;
    if ( !na ) {
        WORD i;
        *nc = nb;
        if ( nb < 0 ) nb = -nb;
        nb <<= 1;
        for ( i = 0; i < nb; i++ ) *c++ = *b++;
        return(0);
    }
    else if ( !nb ) {
        WORD i;
        *nc = na;
        if ( na < 0 ) na = -na;
        na <<= 1;
        for ( i = 0; i < na; i++ ) *c++ = *a++;
        return(0);
    }
    else if ( b[1] == 1 && a[1] == 1 ) {
        if ( na == 1 ) {
            if ( nb == 1 ) {
                *c = *a + *b;
                c[1] = 1;
                if ( *c < *a ) { c[2] = 1; c[3] = 0; *nc = 2; }
                else { *nc = 1; }
                return(0);
            }
            else if ( nb == -1 ) {
                if ( *b > *a ) {
                    *c = *b - *a; *nc = -1;
                }
                else if ( *b < *a ) {
                    *c = *a - *b; *nc = 1;
                }
                else *nc = 0;
                c[1] = 1;
                return(0);
            }
        }
        else if ( na == -1 ){
            if ( nb == -1 ) {
                c[1] = 1;
                *c = *a + *b;
                if ( *c < *a ) { c[2] = 1; c[3] = 0; *nc = -2; }
                else { *nc = -1; }
                return(0);
            }
            else if ( nb == 1 ) {
                if ( *b > *a ) {
                    *c = *b - *a; *nc = 1;
                }
                else if ( *b < *a ) {
                    *c = *a - *b; *nc = -1;
                }
                else *nc = 0;
                c[1] = 1;
                return(0);
            }
        }
    }
    UnPack(a,na,&adenom,&anumer);
    UnPack(b,nb,&bdenom,&bnumer);
    if ( na < 0 ) na = -na;
    if ( nb < 0 ) nb = -nb;
    if ( na == 1 && nb == 1 ) {
        RLONG t1, t2, t3;
        t3 = ((RLONG)a[1])*((RLONG)b[1]);
        t1 = ((RLONG)a[0])*((RLONG)b[1]);
        t2 = ((RLONG)a[1])*((RLONG)b[0]);
        if ( ( anumer > 0 && bnumer > 0 ) || ( anumer < 0 && bnumer < 0 ) ) {
            if ( ( t1 = t1 + t2 ) < t2 ) {
                c[2] = 1;
                c[0] = (UWORD)t1;
                c[1] = (UWORD)(t1 >> BITSINWORD);
                *nc = 3;
            }
            else {
                c[0] = (UWORD)t1;
                if ( ( c[1] = (UWORD)(t1 >> BITSINWORD) ) != 0 ) *nc = 2;
                else *nc = 1;
            }
        }
        else {
            if ( t1 == t2 ) { *nc = 0; return(0); }
            if ( t1 > t2 ) {
                t1 -= t2;
            }
```

```
            else {
                t1 = t2 - t1;
                anumer = -anumer;
            }
            c[0] = (UWORD)t1;
            if ( ( c[1] = (UWORD)(t1 » BITSINWORD) ) != 0 ) *nc = 2;
            else *nc = 1;
        }
        if ( anumer < 0 ) *nc = -*nc;
        d = NumberMalloc("AddRat");
        d[0] = (UWORD)t3;
        if ( ( d[1] = (UWORD)(t3 » BITSINWORD) ) != 0 ) nd = 2;
        else nd = 1;
        if ( Simplify(BHEAD c,nc,d,&nd) ) goto AddRer1;
    }
/*
    else if ( a[na] == 1 && b[nb] == 1 && adenom == 1 && bdenom == 1 ) {
        if ( AddLong(a,na,b,nb,c,&nc) ) goto AddRer2;
        i = ABS(nc); d = c + i; *d++ = 1;
        while ( --i > 0 ) *d++ = 0 ;
        return(0);
    }
*/
    else {
        d = NumberMalloc("AddRat"); e = NumberMalloc("AddRat");
        f = NumberMalloc("AddRat"); g = NumberMalloc("AddRat");
        if ( GcdLong(BHEAD a+na,adenom,b+nb,bdenom,d,&nd) ) goto AddRer;
        if ( *d == 1 && nd == 1 ) nd = 0;
        if ( nd ) {
            if ( DivLong(a+na,adenom,d,nd,e,&ne,c,nc) ) goto AddRer;
            if ( DivLong(b+nb,bdenom,d,nd,f,&nf,c,nc) ) goto AddRer;
            if ( MulLong(a,anumer,f,nf,c,nc) ) goto AddRer;
            if ( MulLong(b,bnumer,e,ne,g,&ng) ) goto AddRer;
        }
        else {
            if ( MulLong(a+na,adenom,b,bnumer,c,nc) ) goto AddRer;
            if ( MulLong(b+nb,bdenom,a,anumer,g,&ng) ) goto AddRer;
        }
        if ( AddLong(c,*nc,g,ng,c,nc) ) goto AddRer;
        if ( !*nc ) {
            NumberFree(g,"AddRat"); NumberFree(f,"AddRat");
            NumberFree(e,"AddRat"); NumberFree(d,"AddRat");
            return(0);
        }
        if ( nd ) {
            if ( Simplify(BHEAD c,nc,d,&nd) ) goto AddRer;
            if ( MulLong(e,ne,d,nd,g,&ng) ) goto AddRer;
            if ( MulLong(g,ng,f,nf,d,&nd) ) goto AddRer;
        }
        else {
            if ( MulLong(a+na,adenom,b+nb,bdenom,d,&nd) ) goto AddRer;
        }
        NumberFree(g,"AddRat"); NumberFree(f,"AddRat"); NumberFree(e,"AddRat");
    }
    Pack(c,nc,d,nd);
    NumberFree(d,"AddRat");
    return(0);
AddRer:
    NumberFree(g,"AddRat"); NumberFree(f,"AddRat"); NumberFree(e,"AddRat");
AddRer1:
    NumberFree(d,"AddRat");
/* AddRer2: */
    MLOCK(ErrorMessageLock);
    MesCall("AddRat");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] AddRat :
        #[ MulRat :         WORD MulRat(a,na,b,nb,c,nc)

    Multiplies the rationals a and b. The Gcd of the individual
    pieces is divided out first to minimize the chances of spurious
    overflows.

*/

WORD MulRat(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    WORD i;
    WORD sgn = 1;
    if ( *b == 1 && b[1] == 1 ) {
        if ( nb == 1 ) {
            *nc = na;
            i = ABS(na); i «= 1;
            while ( --i >= 0 ) *c++ = *a++;
```

```
            return(0);
        }
        else if ( nb == -1 ) {
            *nc = - na;
            i = ABS(na); i <<= 1;
            while ( --i >= 0 ) *c++ = *a++;
            return(0);
        }
    }
    if ( *a == 1 && a[1] == 1 ) {
        if ( na == 1 ) {
            *nc = nb;
            i = ABS(nb); i <<= 1;
            while ( --i >= 0 ) *c++ = *b++;
            return(0);
        }
        else if ( na == -1 ) {
            *nc = - nb;
            i = ABS(nb); i <<= 1;
            while ( --i >= 0 ) *c++ = *b++;
            return(0);
        }
    }
    if ( na < 0 ) { na = -na; sgn = -sgn; }
    if ( nb < 0 ) { nb = -nb; sgn = -sgn; }
    if ( !na || !nb ) { *nc = 0; return(0); }
    if ( na != 1 || nb != 1 ) {
        GETBIDENTITY
        UWORD *xd,*xe, *xf,*xg;
        WORD dden, dnumr, eden, enumr;
        UnPack(a,na,&dden,&dnumr);
        UnPack(b,nb,&eden,&enumr);
        xd = NumberMalloc("MulRat"); xf = NumberMalloc("MulRat");
        for ( i = 0; i < dnumr; i++ ) xd[i] = a[i];
        a += na;
        for ( i = 0; i < dden; i++ ) xf[i] = a[i];
        xe = NumberMalloc("MulRat"); xg = NumberMalloc("MulRat");
        for ( i = 0; i < enumr; i++ ) xe[i] = b[i];
        b += nb;
        for ( i = 0; i < eden; i++ ) xg[i] = b[i];
        if ( Simplify(BHEAD xd,&dnumr,xg,&eden) ||
             Simplify(BHEAD xe,&enumr,xf,&dden) ||
             MulLong(xd,dnumr,xe,enumr,c,nc) ||
             MulLong(xf,dden,xg,eden,xd,&dnumr) ) {
            MLOCK(ErrorMessageLock);
            MesCall("MulRat");
            MUNLOCK(ErrorMessageLock);
            NumberFree(xd,"MulRat"); NumberFree(xe,"MulRat"); NumberFree(xf,"MulRat");
NumberFree(xg,"MulRat");
            SETERROR(-1)
        }
        Pack(c,nc,xd,dnumr);
        NumberFree(xd,"MulRat"); NumberFree(xe,"MulRat"); NumberFree(xf,"MulRat");
NumberFree(xg,"MulRat");
    }
    else {
        UWORD y;
        UWORD a0,a1,b0,b1;
        RLONG xx;
        y = a[0]; b1=b[1];
        do { a0 = y % b1; y = b1; } while ( ( b1 = a0 ) != 0 );
        if ( y != 1 ) {
            a0 = a[0] / y;
            b1 = b[1] / y;
        }
        else {
            a0 = a[0];
            b1 = b[1];
        }
        y=b[0]; a1=a[1];
        do { b0 = y % a1; y = a1; } while ( ( a1 = b0 ) != 0 );
        if ( y != 1 ) {
            a1 = a[1] / y;
            b0 = b[0] / y;
        }
        else {
            a1 = a[1];
            b0 = b[0];
        }
        xx = ((RLONG)a0)*b0;
        if ( xx & AWORDMASK ) {
            *nc = 2;
            c[0] = (UWORD)xx;
            c[1] = (UWORD)(xx >> BITSINWORD);
            xx = ((RLONG)a1)*b1;
            c[2] = (UWORD)xx;
            c[3] = (UWORD)(xx >> BITSINWORD);
```

```
        }
        else {
            c[0] = (UWORD)xx;
            xx = ((RLONG)a1)*b1;
            if ( xx & AWORDMASK ) {
                c[1] = 0;
                c[2] = (UWORD)xx;
                c[3] = (UWORD)(xx » BITSINWORD);
                *nc = 2;
            }
            else {
                c[1] = (UWORD)xx;
                *nc = 1;
            }
        }
    }
    if ( sgn < 0 ) *nc = -*nc;
    return(0);
}

/*
        #] MulRat :
        #[ DivRat :         WORD DivRat(a,na,b,nb,c,nc)

    Divides the rational a by the rational b.

*/

WORD DivRat(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    GETBIDENTITY
    WORD i, j;
    UWORD *xd,*xe,xx;
    if ( !nb ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Rational division by zero");
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    j = i = (nb >= 0)? nb: -nb;
    xd = b; xe = b + i;
    do { xx = *xd; *xd++ = *xe; *xe++ = xx; } while ( --j > 0 );
    j = MulRat(BHEAD a,na,b,nb,c,nc);
    xd = b; xe = b + i;
    do { xx = *xd; *xd++ = *xe; *xe++ = xx; } while ( --i > 0 );
    return(j);
}

/*
        #] DivRat :
        #[ Simplify :       WORD Simplify(a,na,b,nb)

    Determines the greatest common denominator of a and b and
    devides both by it. A possible sign is put in a. This is
    the simplification of the fraction a/b.

*/

WORD Simplify(PHEAD UWORD *a, WORD *na, UWORD *b, WORD *nb)
{
    GETBIDENTITY
    UWORD *x1,*x2,*x3;
    UWORD *x4;
    WORD n1,n2,n3,n4,sgn = 1;
    WORD i;
    UWORD *Siscrat5, *Siscrat6, *Siscrat7, *Siscrat8;
    if ( *na < 0 ) { *na = -*na; sgn = -sgn; }
    if ( *nb < 0 ) { *nb = -*nb; sgn = -sgn; }
    Siscrat5 = NumberMalloc("Simplify"); Siscrat6 = NumberMalloc("Simplify");
    Siscrat7 = NumberMalloc("Simplify"); Siscrat8 = NumberMalloc("Simplify");
    x1 = Siscrat8; x2 = Siscrat7;
    if ( *nb == 1 ) {
        x3 = Siscrat6;
        if ( DivLong(a,*na,b,*nb,x1,&n1,x2,&n2) ) goto SimpErr;
        if ( !n2 ) {
            for ( i = 0; i < n1; i++ ) *a++ = *x1++;
            *na = n1;
            *b = 1;
        }
        else {
            UWORD y1, y2, y3;
            y2 = *b;
            y3 = *x2;
            do { y1 = y2 % y3; y2 = y3; } while ( ( y3 = y1 ) != 0 );
            if ( ( *x2 = y2 ) != 1 ) {
                *b /= y2;
                if ( DivLong(a,*na,x2,(WORD)1,x1,&n1,x3,&n3) ) goto SimpErr;
```

```
                for ( i = 0; i < n1; i++ ) *a++ = *x1++;
                *na = n1;
            }
        }
    }
#ifdef NEWTRICK
    else if ( *na >= GCDMAX && *nb >= GCDMAX ) {
        n1 = i = *na; x3 = a;
        NCOPY(x1,x3,i);
        x3 = b; n2 = i = *nb;
        NCOPY(x2,x3,i);
        x4 = Siscrat5;
        x2 = Siscrat6;
        x3 = Siscrat7;
        if ( GcdLong(BHEAD Siscrat8,n1,Siscrat7,n2,x2,&n3) ) goto SimpErr;
        n2 = n3;
        if ( *x2 != 1 || n2 != 1 ) {
            DivLong(a,*na,x2,n2,x1,&n1,x4,&n4);
            *na = i = n1;
            NCOPY(a,x1,i);
            DivLong(b,*nb,x2,n2,x3,&n3,x4,&n4);
            *nb = i = n3;
            NCOPY(b,x3,i);
        }
    }
#endif
    else {
        x4 = Siscrat5;
        n1 = i = *na; x3 = a;
        NCOPY(x1,x3,i);
        x3 = b; n2 = i = *nb;
        NCOPY(x2,x3,i);
        x1 = Siscrat8; x2 = Siscrat7; x3 = Siscrat6;
        for(;;){
            if ( DivLong(x1,n1,x2,n2,x4,&n4,x3,&n3) ) goto SimpErr;
            if ( !n3 ) break;
            if ( n2 == 1 ) {
                while ( ( *x1 = (*x2) % (*x3) ) != 0 ) { *x2 = *x3; *x3 = *x1; }
                *x2 = *x3;
                break;
            }
            if ( DivLong(x2,n2,x3,n3,x4,&n4,x1,&n1) ) goto SimpErr;
            if ( !n1 ) { x2 = x3; n2 = n3; x3 = Siscrat7; break; }
            if ( n3 == 1 ) {
                while ( ( *x2 = (*x3) % (*x1) ) != 0 ) { *x3 = *x1; *x1 = *x2; }
                *x2 = *x1;
                n2 = 1;
                break;
            }
            if ( DivLong(x3,n3,x1,n1,x4,&n4,x2,&n2) ) goto SimpErr;
            if ( !n2 ) { x2 = x1; n2 = n1; x1 = Siscrat7; break; }
            if ( n1 == 1 ) {
                while ( ( *x3 = (*x1) % (*x2) ) != 0 ) { *x1 = *x2; *x2 = *x3; }
                break;
            }
        }
        if ( *x2 != 1 || n2 != 1 ) {
            DivLong(a,*na,x2,n2,x1,&n1,x4,&n4);
            *na = i = n1;
            NCOPY(a,x1,i);
            DivLong(b,*nb,x2,n2,x3,&n3,x4,&n4);
            *nb = i = n3;
            NCOPY(b,x3,i);
        }
    }
    if ( sgn < 0 ) *na = -*na;
    NumberFree(Siscrat5,"Simplify"); NumberFree(Siscrat6,"Simplify");
    NumberFree(Siscrat7,"Simplify"); NumberFree(Siscrat8,"Simplify");
    return(0);
SimpErr:
    MLOCK(ErrorMessageLock);
    MesCall("Simplify");
    MUNLOCK(ErrorMessageLock);
    NumberFree(Siscrat5,"Simplify"); NumberFree(Siscrat6,"Simplify");
    NumberFree(Siscrat7,"Simplify"); NumberFree(Siscrat8,"Simplify");
    SETERROR(-1)
}

/*
        #] Simplify :
        #[ AccumGCD :       WORD AccumGCD(PHEAD a,na,b,nb)

        Routine takes the rational GCD of the fractions in a and b and
        replaces a by the GCD of the two.
        The rational GCD is defined as the rational that consists of
        the GCD of the numerators divided by the GCD of the denominators
*/
```

```

WORD AccumGCD(PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
{
    GETBIDENTITY
    WORD nna,nnb,numa,numb,dena,denb,numc,denc;
    UWORD *GCDbuffer = NumberMalloc("AccumGCD");
    int i;
    nna = *na; if ( nna < 0 ) nna = -nna; nna = (nna-1)/2;
    nnb = nb;  if ( nnb < 0 ) nnb = -nnb; nnb = (nnb-1)/2;
    UnPack(a,nna,&dena,&numa);
    UnPack(b,nnb,&denb,&numb);
    if ( GcdLong(BHEAD a,numa,b,numb,GCDbuffer,&numc) ) goto AccErr;
    numa = numc;
    for ( i = 0; i < numa; i++ ) a[i] = GCDbuffer[i];
    if ( GcdLong(BHEAD a+nna,dena,b+nnb,denb,GCDbuffer,&denc) ) goto AccErr;
    dena = denc;
    for ( i = 0; i < dena; i++ ) a[i+nna] = GCDbuffer[i];
    Pack(a,&numa,a+nna,dena);
    *na = INCLENG(numa);
    NumberFree(GCDbuffer,"AccumGCD");
    return(0);
AccErr:
    MLOCK(ErrorMessageLock);
    MesCall("AccumGCD");
    MUNLOCK(ErrorMessageLock);
    NumberFree(GCDbuffer,"AccumGCD");
    SETERROR(-1)
}

/*
        #] AccumGCD :
        #[ TakeRatRoot:
*/

int TakeRatRoot(UWORD *a, WORD *n, WORD power)
{
    WORD numer,denom, nn;
    if ( ( power & 1 ) == 0 && *n < 0 ) return(1);
    if ( ABS(*n) == 1 && a[0] == 1 && a[1] == 1 ) return(0);
    nn = ABS(*n);
    UnPack(a,nn,&denom,&numer);
    if ( TakeLongRoot(a+nn,&denom,power) ) return(1);
    if ( TakeLongRoot(a,&numer,power) ) return(1);
    Pack(a,&numer,a+nn,denom);
    if ( *n < 0 ) *n = -numer;
    else          *n = numer;
    return(0);
}

/*
        #] TakeRatRoot:
    #] RekenRational :
    #[ RekenLong :
        #[ AddLong :        WORD AddLong(a,na,b,nb,c,nc)

    Long addition. Uses addition and subtraction of positive numbers.

*/

WORD AddLong(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    WORD sgn, res;
    if ( na < 0 ) {
        if ( nb < 0 ) {
            if ( AddPLon(a,-na,b,-nb,c,nc) ) return(-1);
            *nc = -*nc;
            return(0);
        }
        else {
            na = -na;
            sgn = -1;
        }
    }
    else {
        if ( nb < 0 ) {
            nb = -nb;
            sgn = 1;
        }
        else { return( AddPLon(a,na,b,nb,c,nc) ); }
    }
    if ( ( res = BigLong(a,na,b,nb) ) > 0 ) {
        SubPLon(a,na,b,nb,c,nc);
        if ( sgn < 0 ) *nc = -*nc;
    }
    else if ( res < 0 ) {
        SubPLon(b,nb,a,na,c,nc);
        if ( sgn > 0 ) *nc = -*nc;
```

```
    }
    else {
        *nc = 0;
        *c = 0;
    }
    return(0);
}

/*
        #] AddLong :
        #[ AddPLon :        WORD AddPLon(a,na,b,nb,c,nc)

    Adds two long integers a and b and puts the result in c.
    The length of a and b are na and nb. The length of c is returned in nc.
    c can be a or b.
*/

WORD AddPLon(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    UWORD carry = 0, e, nd = 0;
    while ( na && nb ) {
        e = *a;
        *c = e + *b + carry;
        if ( carry ) {
            if ( e < *c ) carry = 0;
        }
        else {
            if ( e > *c ) carry = 1;
        }
        a++; b++; c++; nd++; na--; nb--;
    }
    while ( na ) {
        if ( carry ) {
            *c = *a++ + carry;
            if ( *c++ ) carry = 0;
        }
        else *c++ = *a++;
        nd++; na--;
    }
    while ( nb ) {
        if ( carry ) {
            *c = *b++ + carry;
            if ( *c++ ) carry = 0;
        }
        else *c++ = *b++;
        nd++; nb--;
    }
    if ( carry ) {
        nd++;
        if ( nd > (UWORD)AM.MaxTal ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Overflow in addition");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        *c++ = carry;
    }
    *nc = nd;
    return(0);
}

/*
        #] AddPLon :
        #[ SubPLon :        VOID SubPLon(a,na,b,nb,c,nc)

    Subtracts b from a. Assumes that a > b. Result in c.
    c can be a or b.

*/

VOID SubPLon(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    UWORD borrow = 0, e, nd = 0;
    while ( nb ) {
        e = *a;
        if ( borrow ) {
            *c = e - *b - borrow;
            if ( *c < e ) borrow = 0;
        }
        else {
            *c = e - *b;
            if ( *c > e ) borrow = 1;
        }
        a++; b++; c++; na--; nb--; nd++;
    }
    while ( na ) {
        if ( borrow ) {
```

```
            if ( *a ) { *c++ = *a++ - 1; borrow = 0; }
            else { *c++ = (UWORD)(-1); a++; }
        }
        else *c++ = *a++;
        na--; nd++;
    }
    while ( nd && !*--c ) { nd--; }
    *nc = (WORD)nd;
}

/*
        #] SubPLon :
        #[ MulLong :        WORD MulLong(a,na,b,nb,c,nc)

    Does a Long multiplication. Assumes that WORD is half the size
    of a LONG to work out the scheme! The number of operations is
    the canonical na*nm multiplications.
    c should not overlap with a or b.

*/

WORD MulLong(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    WORD sgn = 1;
    UWORD i, *ic, *ia;
    RLONG t, bb;
    if ( !na || !nb ) { *nc = 0; return(0); }
    if ( na < 0 ) { na = -na; sgn = -sgn; }
    if ( nb < 0 ) { nb = -nb; sgn = -sgn; }
    *nc = i = na + nb;
    if ( i > (UWORD)(AM.MaxTal+1) ) goto MulLov;
    ic = c;
/*
    #[ GMP stuff :
*/
#ifdef WITHGMP
    if (na > 3 && nb > 3) {
/*      mp_limb_t res;  */
        UWORD *to, *from;
        int j;
        GETIDENTITY
        UWORD *DLscrat9 = NumberMalloc("MulLong"), *DLscratA = NumberMalloc("MulLong"), *DLscratB =
    NumberMalloc("MulLong");
#if ( GMPSPREAD != 1 )
        if ( na & 1 ) {
            from = a; a = to = DLscrat9; j = na; NCOPY(to, from, j);
            a[na++] = 0;
            ++*nc;
        } else
#endif
        if ( (LONG)a & (sizeof(mp_limb_t)-1) ) {
            from = a; a = to = DLscrat9; j = na; NCOPY(to, from, j);
        }

#if ( GMPSPREAD != 1 )
        if ( nb & 1 ) {
            from = b; b = to = DLscratA; j = nb; NCOPY(to, from, j);
            b[nb++] = 0;
            ++*nc;
        } else
#endif
        if ( (LONG)b & (sizeof(mp_limb_t)-1) ) {
            from = b; b = to = DLscratA; j = nb; NCOPY(to, from, j);
        }

        if ( ( *nc > (WORD)i ) || ( (LONG)c & (LONG)(sizeof(mp_limb_t)-1) ) ) {
            ic = DLscratB;
        }
        if ( na < nb ) {
            /* res = */
            mpn_mul((mp_ptr)ic, (mp_srcptr)b, nb/GMPSPREAD, (mp_srcptr)a, na/GMPSPREAD);
        } else {
            /* res = */
            mpn_mul((mp_ptr)ic, (mp_srcptr)a, na/GMPSPREAD, (mp_srcptr)b, nb/GMPSPREAD);
        }
        while ( ic[i-1] == 0 ) i--;
        *nc = i;
/*
        if ( res == 0 ) *nc -= GMPSPREAD;
        else if ( res <= WORDMASK ) --*nc;
*/
        if ( ic != c ) {
            j = *nc; NCOPY(c, ic, j);
        }
        if ( sgn < 0 ) *nc = -(*nc);
        NumberFree(DLscrat9,"MulLong"); NumberFree(DLscratA,"MulLong");
    NumberFree(DLscratB,"MulLong");
```

```
        return(0);
    }
#endif
/*
    #] GMP stuff :
*/
    do { *ic++ = 0; } while ( --i > 0 );
    do {
        ia = a;
        ic = c++;
        t = 0;
        i = na;
        bb = (RLONG)(*b++);
        do {
            t = (*ia++) * bb + t + *ic;
            *ic++ = (WORD)t;
            t »= BITSINWORD;       /* should actually be a swap */
        } while ( --i > 0 );
        if ( t ) *ic = (UWORD)t;
    } while ( --nb > 0 );
    if ( !*ic ) (*nc)--;
    if ( *nc > AM.MaxTal ) goto MulLov;
    if ( sgn < 0 ) *nc = -(*nc);
    return(0);
MulLov:
    MLOCK(ErrorMessageLock);
    MesPrint("Overflow in Multiplication");
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
        #] MulLong :
        #[ BigLong :        WORD BigLong(a,na,b,nb)

    Returns > 0 if a > b, < 0 if b > a and 0 if a == b

*/

WORD BigLong(UWORD *a, WORD na, UWORD *b, WORD nb)
{
    a += na;
    b += nb;
    while ( na && !*--a ) na--;
    while ( nb && !*--b ) nb--;
    if ( nb < na ) return(1);
    if ( nb > na ) return(-1);
    while ( --na >= 0 ) {
        if ( *a > *b ) return(1);
        else if ( *b > *a ) return(-1);
        a--; b--;
    }
    return(0);
}

/*
        #] BigLong :
        #[ DivLong :        WORD DivLong(a,na,b,nb,c,nc,d,nd)

    This is the long division which knows a couple of exceptions.
    It uses therefore a recursive call for the renormalization.
    The quotient comes in c and the remainder in d.
    d may be overlapping with b. It may also be identical to a.
    c should not overlap with a, but it can overlap with b.

*/

WORD DivLong(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c,
             WORD *nc, UWORD *d, WORD *nd)
{
    WORD sgna = 1, sgnb = 1, ne, nf, ng, nh;
    WORD i, ni;
    UWORD *w1, *w2;
    RLONG t, v;
    UWORD *e, *f, *ff, *g, norm, estim;
#ifdef WITHGMP
    UWORD *DLscrat9, *DLscratA, *DLscratB, *DLscratC;
#endif
    RLONG esthelp;
    if ( !nb ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Division by zero");
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    if ( !na ) { *nc = *nd = 0; return(0); }
    if ( na < 0 ) { sgna = -sgna; na = -na; }
```

```
    if ( nb < 0 ) { sgnb = -sgnb; nb = -nb; }
    if ( na < nb ) {
        for ( i = 0; i < na; i++ ) *d++ = *a++;
        *nd = na;
        *nc = 0;
    }
    else if ( nb == na && ( i = BigLong(b,nb,a,na) ) >= 0 ) {
        if ( i > 0 ) {
            for ( i = 0; i < na; i++ ) *d++ = *a++;
            *nd = na;
            *nc = 0;
        }
        else {
            *c = 1;
            *nc = 1;
            *nd = 0;
        }
    }
    else if ( nb == 1 ) {
        if ( *b == 1 ) {
            for ( i = 0; i < na; i++ ) *c++ = *a++;
            *nc = na;
            *nd = 0;
        }
        else {
            w1 = a+na;
            *nc = ni = na;
            *nd = 1;
            w2 = c+ni;
            v = (RLONG)(*b);
            t = (RLONG)(*--w1);
            while ( --ni >= 0 ) {
                *--w2 = t / v;
                t -= v * (*w2);
                if ( ni ) {
                    t <<= BITSINWORD;
                    t += *--w1;
                }
            }
            if ( ( *d = (UWORD)t ) == 0 ) *nd = 0;
            if ( !*(c+na-1) ) (*nc)--;
        }
    }
    else {
        GETIDENTITY
/*
        #[ GMP stuff :

        We start with copying a and b.
        Then we make space for c and d.
        Next we call mpn_tdiv_qr
        We adjust sizes and copy to c and d if needed.
        Finally the signs are settled.
*/
#ifdef WITHGMP
        if ( na > 4 && nb > 3 ) {
          UWORD *ic, *id, *to, *from;
          int j = na - nb;
          DLscrat9 = NumberMalloc("DivLong"); DLscratA = NumberMalloc("DivLong");
          DLscratB = NumberMalloc("DivLong"); DLscratC = NumberMalloc("DivLong");

#if ( GMPSPREAD != 1 )
          if ( na & 1 ) {
            from = a; a = to = DLscrat9; i = na; NCOPY(to, from, i);
            a[na++] = 0;
          } else
#endif
          if ( (LONG)a & (sizeof(mp_limb_t)-1) ) {
            from = a; a = to = DLscrat9; i = na; NCOPY(to, from, i);
          }

#if ( GMPSPREAD != 1 )
          if ( nb & 1 ) {
            from = b; b = to = DLscratA; i = nb; NCOPY(to, from, i);
            b[nb++] = 0;
          } else
#endif
          if ( ( (LONG)b & (sizeof(mp_limb_t)-1) ) != 0 ) {
            from = b; b = to = DLscratA; i = nb; NCOPY(to, from, i);
          }
          if ( ( (LONG)c & (sizeof(mp_limb_t)-1) ) != 0 ) ic = DLscratB;
          else                                            ic = c;

          if ( ( (LONG)d & (sizeof(mp_limb_t)-1) ) != 0 ) id = DLscratC;
          else                                            id = d;
          mpn_tdiv_qr((mp_limb_t *)ic,(mp_limb_t *)id,(mp_size_t)0,
            (const mp_limb_t *)a,(mp_size_t)(na/GMPSPREAD),
```

```
            (const mp_limb_t *)b,(mp_size_t)(nb/GMPSPREAD));
          while ( j >= 0 && ic[j] == 0 ) j--;
          j++; *nc = j;
          if ( c != ic ) { NCOPY(c,ic,j); }
          j = nb-1;
          while ( j >= 0 && id[j] == 0 ) j--;
          j++; *nd = j;
          if ( d != id ) { NCOPY(d,id,j); }
          if ( sgna < 0 ) { *nc = -(*nc); *nd = -(*nd); }
          if ( sgnb < 0 ) { *nc = -(*nc); }
          NumberFree(DLscrat9,"DivLong"); NumberFree(DLscratA,"DivLong");
          NumberFree(DLscratB,"DivLong"); NumberFree(DLscratC,"DivLong");
          return(0);
        }
#endif
/*
        #] GMP stuff :
*/
        /* Start with normalization operation */

        e = NumberMalloc("DivLong"); f = NumberMalloc("DivLong"); g = NumberMalloc("DivLong");
        if ( b[nb-1] == (FULLMAX-1) ) norm = 1;
        else {
            norm = (UWORD)(((ULONG)FULLMAX) / (ULONG)((b[nb-1]+1L)));
        }
        f[na] = 0;
        if ( MulLong(b,nb,&norm,1,e,&ne) ||
             MulLong(a,na,&norm,1,f,&nf) ) {
            NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
            return(-1);
        }
        if ( BigLong(f+nf-ne,ne,e,ne) >= 0 ) {
            SubPLon(f+nf-ne,ne,e,ne,f+nf-ne,&nh);
            w1 = c + (nf-ne);
            *nc = nf-ne+1;
        }
        else {
            nh = ne;
            *nc = nf-ne;
            w1 = 0;
        }
        w2 = c; i = *nc; do { *w2++ = 0; } while ( --i > 0 );
        nf = na;
        ni = nf-ne;
        esthelp = (RLONG)(e[ne-1]) + 1L;
        while ( nf >= ne ) {
            if ( (WORD)esthelp == 0 ) {
                estim = (WORD)(((((RLONG)(f[nf]))«BITSINWORD)+f[nf-1])»BITSINWORD);
            }
            else {
                estim = (WORD)(((((RLONG)(f[nf]))«BITSINWORD)+f[nf-1])/esthelp);
            }
            /* This estimate may be up to two too small */
            if ( estim ) {
                MulLong(e,ne,&estim,1,g,&ng);
                nh = ne + 1; if ( !f[ni+ne] ) nh--;
                SubPLon(f+ni,nh,g,ng,f+ni,&nh);
            }
            else {
                w2 = f+ni+ne; nh = ne+1;
                while ( ( nh > 0 ) && !*w2 ) { nh--; w2--; }
            }
            if ( BigLong(f+ni,nh,e,ne) >= 0 ) {
                estim++;
                SubPLon(f+ni,nh,e,ne,f+ni,&nh);
                if ( BigLong(f+ni,nh,e,ne) >= 0 ) {
                    estim++;
                    SubPLon(f+ni,nh,e,ne,f+ni,&nh);
                    if ( BigLong(f+ni,nh,e,ne) >= 0 ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Problems in DivLong");
                        AO.OutSkip = 3;
                        FiniLine();
                        i = na;
                        while ( --i >= 0 ) { TalToLine((UWORD)(*a++)); TokenToLine((UBYTE *)"  "); }
                        FiniLine();
                        i = nb;
                        while ( --i >= 0 ) { TalToLine((UWORD)(*b++)); TokenToLine((UBYTE *)"  "); }
                        AO.OutSkip = 0;
                        FiniLine();
                        MUNLOCK(ErrorMessageLock);
                        NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
                        return(-1);
                    }
                }
            }
            c[ni] = estim;
```

```
            nf--;
            ni--;
        }
        if ( w1 ) *w1 = 1;

        /* Finish with the renormalization operation */

        if ( nh > 0 ) {
            if ( norm == 1 ) {
                *nd = i = nh; ff = f;
                NCOPY(d,ff,i);
            }
            else {
                w1 = f+nh;
                *nd = ni = nh;
                w2 = d+ni;
                v = norm;
                t = (RLONG)(*--w1);
                while ( --ni >= 0 ) {
                    *--w2 = t / v;
                    t -= v * (*w2);
                    if ( ni ) {
                        t <<= BITSINWORD;
                        t += *--w1;
                    }
                }
                if ( t ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Error in DivLong");
                    MUNLOCK(ErrorMessageLock);
                    NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
                    return(-1);
                }
                if ( !*(d+nh-1) ) (*nd)--;
            }
        }
        else { *nd = 0; }
        NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
    }
    if ( sgna < 0 ) { *nc = -(*nc); *nd = -(*nd); }
    if ( sgnb < 0 ) { *nc = -(*nc); }
    return(0);
}

/*
        #] DivLong :
        #[ RaisPow :        WORD RaisPow(a,na,b)

    Raises a to the power b. a is a Long integer and b >= 0.
    The method that is used works with a bitdecomposition of b.
*/

WORD RaisPow(PHEAD UWORD *a, WORD *na, UWORD b)
{
    GETBIDENTITY
    WORD i, nu;
    UWORD *it, *iu, c;
    UWORD *is, *iss;
    WORD ns, nt, nmod;
    nmod = ABS(AN.ncmod);
    if ( !*na || ( ( *na == 1 ) && ( *a == 1 ) ) ) return(0);
    if ( !b ) { *na=1; *a=1; return(0); }
    is = NumberMalloc("RaisPow");
    it = NumberMalloc("RaisPow");
    for ( i = 0; i < ABS(*na); i++ ) is[i] = a[i];
    ns = *na;
    c = b;
    for ( i = 0; i < BITSINWORD; i++ ) {
        if ( !c ) break;
        c >>= 1;
    }
    i--;
    c = 1 << i;
    while ( --i >= 0 ) {
        c >>= 1;
        if(MulLong(is,ns,is,ns,it,&nt)) goto RaisOvl;
        if ( b & c ) {
            if ( MulLong(it,nt,a,*na,is,&ns) ) goto RaisOvl;
        }
        else {
            iu = is; is = it; it = iu;
            nu = ns; ns = nt; nt = nu;
        }
        if ( nmod != 0 ) {
            if ( DivLong(is,ns,(UWORD *)AC.cmod,nmod,it,&nt,is,&ns) ) goto RaisOvl;
        }
    }
```

```
    if ( ( nmod != 0 ) && ( ( AC.modmode & POSNEG ) != 0 ) ) {
        NormalModulus(is,&ns);
    }
    if ( ( *na = i = ns ) != 0 ) { iss = is; i=ABS(i); NCOPY(a,iss,i); }
    NumberFree(is,"RaisPow"); NumberFree(it,"RaisPow");
    return(0);
RaisOvl:
    MLOCK(ErrorMessageLock);
    MesCall("RaisPow");
    MUNLOCK(ErrorMessageLock);
    NumberFree(is,"RaisPow"); NumberFree(it,"RaisPow");
    SETERROR(-1)
}

/*
        #] RaisPow :
        #[ RaisPowCached :
*/

VOID RaisPowCached (PHEAD WORD x, WORD n, UWORD **c, WORD *nc) {

    int i,j;
    WORD new_small_power_maxx, new_small_power_maxn, ID;
    WORD *new_small_power_n;
    UWORD **new_small_power;

    /* check whether to extend the array */
    if (x>=AT.small_power_maxx || n>=AT.small_power_maxn) {

        new_small_power_maxx = AT.small_power_maxx;
        if (x>=AT.small_power_maxx)
            new_small_power_maxx = MaX(2*AT.small_power_maxx, x+1);

        new_small_power_maxn = AT.small_power_maxn;
        if (n>=AT.small_power_maxn)
            new_small_power_maxn = MaX(2*AT.small_power_maxn, n+1);

        new_small_power_n = (WORD*) Malloc1(new_small_power_maxx*new_small_power_maxn*sizeof(WORD),"RaisPowCached");
        new_small_power = (UWORD **) Malloc1(new_small_power_maxx*new_small_power_maxn*sizeof(UWORD *),"RaisPowCached");

        for (i=0; i<new_small_power_maxx * new_small_power_maxn; i++) {
            new_small_power_n[i] = 0;
            new_small_power  [i] = NULL;
        }

        for (i=0; i<AT.small_power_maxx; i++)
            for (j=0; j<AT.small_power_maxn; j++) {
                new_small_power_n[i*new_small_power_maxn+j] = AT.small_power_n[i*AT.small_power_maxn+j];
                new_small_power  [i*new_small_power_maxn+j] = AT.small_power[i*AT.small_power_maxn+j];
            }

        if (AT.small_power_n != NULL) {
            M_free(AT.small_power_n,"RaisPowCached");
            M_free(AT.small_power,"RaisPowCached");
        }

        AT.small_power_maxx = new_small_power_maxx;
        AT.small_power_maxn = new_small_power_maxn;
        AT.small_power_n    = new_small_power_n;
        AT.small_power      = new_small_power;
    }

    /* check whether the results is already calculated */
    ID = x * AT.small_power_maxn + n;

    if (AT.small_power[ID] == NULL) {
#ifdef OLDRAISPOWCACHED
        AT.small_power[ID] = NumberMalloc("RaisPowCached");
        AT.small_power_n[ID] = 1;
        AT.small_power[ID][0] = x;
        RaisPow(BHEAD AT.small_power[ID],&AT.small_power_n[ID],n);
#else
        UWORD *c = NumberMalloc("RaisPowCached");
        WORD i, k = 1;
        c[0] = x;
        RaisPow(BHEAD c,&k,n);
/*
        And now get the proper amount.
*/
        if ( AT.InNumMem < k ) {    /* We should start a new buffer */
            AT.InNumMem = 5*AM.MaxTal;
            AT.NumMem = (UWORD *)Malloc1(AT.InNumMem*sizeof(UWORD),"RaisPowCached");
/*
```

```
            MesPrint("  Got an extra %l UWORDS in RaisPowCached",AT.InNumMem);
*/
        }
        for ( i = 0; i < k; i++ ) AT.NumMem[i] = c[i];
        AT.small_power[ID] = AT.NumMem;
        AT.small_power_n[ID] = k;
        AT.NumMem += k;
        AT.InNumMem -= k;
        NumberFree(c,"RaisPowCached");
#endif
    }

    /* return the result */
    *c  = AT.small_power[ID];
    *nc = AT.small_power_n[ID];
}

/*
        #] RaisPowCached :
        #[ RaisPowMod :

   Computes the power x^n mod m
 */
WORD RaisPowMod (WORD x, WORD n, WORD m) {
    LONG y=1, z=x;
    while (n) {
        if (n&1) { y*=z; y%=m; }
        z*=z; z%=m;
        n»=1;
    }
    return (WORD)y;
}

/*
    #] RaisPowMod :
        #[ NormalModulus :  int NormalModulus(UWORD *a,WORD *na)
*/
int NormalModulus(UWORD *a,WORD *na)
{
    WORD n;
    if ( AC.halfmod == 0 ) {
      LOCK(AC.halfmodlock);
      if ( AC.halfmod == 0 ) {
        UWORD two[1],remain[1];
        WORD dummy;
        two[0] = 2;
        AC.halfmod = (UWORD *)Malloc1((ABS(AC.ncmod))*sizeof(UWORD),"halfmod");
        DivLong((UWORD *)AC.cmod,(ABS(AC.ncmod)),two,1
                ,(UWORD *)AC.halfmod,&(AC.nhalfmod),remain,&dummy);
      }
      UNLOCK(AC.halfmodlock);
    }
    n = ABS(*na);
    if ( BigLong(a,n,AC.halfmod,AC.nhalfmod) > 0 ) {
        SubPLon((UWORD *)AC.cmod,(ABS(AC.ncmod)),a,n,a,&n);
        if ( *na > 0 ) { *na = -n; }
        else { *na = n; }
        return(1);
    }
    return(0);
}

/*
        #] NormalModulus :
        #[ MakeInverses :
*/
int MakeInverses()
{
    WORD n = AC.cmod[0], i, inv2;
    if ( AC.ncmod != 1 ) return(1);
    if ( AC.modinverses == 0 ) {
      LOCK(AC.halfmodlock);
      if ( AC.modinverses == 0 ) {
        AC.modinverses = (UWORD *)Malloc1(n*sizeof(UWORD),"modinverses");
        AC.modinverses[0] = 0;
        AC.modinverses[1] = 1;
        for ( i = 2; i < n; i++ ) {
            if ( GetModInverses(i,n,
                        (WORD *)(&(AC.modinverses[i])),&inv2) ) {
                SETERROR(-1)
            }
        }
      }
      UNLOCK(AC.halfmodlock);
    }
    return(0);
}
```

```

/*
        #] MakeInverses :
        #[ GetModInverses :
*/
int GetModInverses(WORD m1, WORD m2, WORD *im1, WORD *im2)
{
    WORD a1, a2, a3;
    WORD b1, b2, b3;
    WORD x = m1, y, c, d = m2;
    if ( x < 1 || d <= 1 ) goto somethingwrong;
    a1 = 0; a2 = 1;
    b1 = 1; b2 = 0;
    for(;;) {
        c = d/x; y = d%x; /* a good compiler makes this faster than y=d-c*x */
        if ( y == 0 ) break;
        a3 = a1-c*a2; a1 = a2; a2 = a3;
        b3 = b1-c*b2; b1 = b2; b2 = b3;
        d = x; x = y;
    }
    if ( x != 1 ) goto somethingwrong;
    if ( a2 < 0 ) a2 += m2;
    if ( b2 < 0 ) b2 += m1;
    if (im1!=NULL) *im1 = a2;
    if (im2!=NULL) *im2 = b2;
    return(0);
somethingwrong:
    MLOCK(ErrorMessageLock);
    MesPrint("Error trying to determine inverses in GetModInverses");
    MUNLOCK(ErrorMessageLock);
    return(-1);
}
/*
        #] GetModInverses :
        #[ GetLongModInverses :
*/

int GetLongModInverses(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *ia, WORD *nia, UWORD *ib,
    WORD *nib) {

    UWORD *s, *t, *sa, *sb, *ta, *tb, *x, *y, *swap1;
    WORD ns, nt, nsa, nsb, nta, ntb, nx, ny, swap2;

    s = NumberMalloc("GetLongModInverses");
    ns = na;
    WCOPY(s, a, ABS(ns));

    t = NumberMalloc("GetLongModInverses");
    nt = nb;
    WCOPY(t, b, ABS(nt));

    sa = NumberMalloc("GetLongModInverses");
    nsa = 1;
    sa[0] = 1;

    sb = NumberMalloc("GetLongModInverses");
    nsb = 0;

    ta = NumberMalloc("GetLongModInverses");
    nta = 0;

    tb = NumberMalloc("GetLongModInverses");
    ntb = 1;
    tb[0] = 1;

    x = NumberMalloc("GetLongModInverses");
    y = NumberMalloc("GetLongModInverses");

    while (nt != 0) {
        DivLong(s,ns,t,nt,x,&nx,y,&ny);
        swap1=s; s=y; y=swap1;
        ns=ny;
        MulLong(x,nx,ta,nta,y,&ny);
        AddLong(sa,nsa,y,-ny,sa,&nsa);
        MulLong(x,nx,tb,ntb,y,&ny);
        AddLong(sb,nsb,y,-ny,sb,&nsb);

        swap1=s; s=t; t=swap1;
        swap2=ns; ns=nt; nt=swap2;
        swap1=sa; sa=ta; ta=swap1;
        swap2=nsa; nsa=nta; nta=swap2;
        swap1=sb; sb=tb; tb=swap1;
        swap2=nsb; nsb=ntb; ntb=swap2;
    }

    if (ia!=NULL) {
        *nia = nsa*ns;
```

```
        WCOPY(ia,sa,ABS(*nia));
    }

    if (ib!=NULL) {
        *nib = nsb*ns;
        WCOPY(ib,sb,ABS(*nib));
    }

    NumberFree(s,"GetLongModInverses");
    NumberFree(t,"GetLongModInverses");
    NumberFree(sa,"GetLongModInverses");
    NumberFree(sb,"GetLongModInverses");
    NumberFree(ta,"GetLongModInverses");
    NumberFree(tb,"GetLongModInverses");
    NumberFree(x,"GetLongModInverses");
    NumberFree(y,"GetLongModInverses");

    return 0;
}

/*
        #] GetLongModInverses :
        #[ Product :        WORD Product(a,na,b)

    Multiplies the Long number in a with the WORD b.

*/

WORD Product(UWORD *a, WORD *na, WORD b)
{
    WORD i, sgn = 1;
    RLONG t, u;
    if ( *na < 0 ) { *na = -(*na); sgn = -sgn; }
    if ( b < 0 ) { b = -b; sgn = -sgn; }
    t = 0;
    u = (RLONG)b;
    for ( i = 0; i < *na; i++ ) {
        t += *a * u;
        *a++ = (UWORD)t;
        t >>= BITSINWORD;
    }
    if ( t > 0 ) {
        if ( ++(*na) > AM.MaxTal ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Overflow in Product");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        *a = (UWORD)t;
    }
    if ( sgn < 0 ) *na = -(*na);
    return(0);
}

/*
        #] Product :
        #[ Quotient :       UWORD Quotient(a,na,b)

        Routine divides the long number a by b with the assumption that
        there is no remainder (like while computing binomials).

*/

UWORD Quotient(UWORD *a, WORD *na, WORD b)
{
    RLONG v, t;
    WORD i, j, sgn = 1;
    if ( ( i = *na ) < 0 ) { sgn = -1; i = -i; }
    if ( b < 0 ) { b = -b; sgn = -sgn; }
    if ( i == 1 ) {
        if ( ( *a /= (UWORD)b ) == 0 ) *na = 0;
        if ( sgn < 0 ) *na = -*na;
        return(0);
    }
    a += i;
    j = i;
    v = (RLONG)b;
    t = (RLONG)(*--a);
    while ( --i >= 0 ) {
        *a = t / v;
        t -= v * (*a);
        if ( i ) {
            t <<= BITSINWORD;
            t += *--a;
        }
    }
    a += j - 1;
```

```
    if ( !*a ) j--;
    if ( sgn < 0 ) j = -j;
    *na = j;
    return(0);
}

/*
        #] Quotient :
        #[ Remain10 :       WORD Remain10(a,na)

    Routine devides a by 10 and gives the remainder as return value.
    The value of a will be the quotient! a must be positive.

*/

WORD Remain10(UWORD *a, WORD *na)
{
    WORD i;
    RLONG t, u;
    UWORD *b;
    i = *na;
    t = 0;
    b = a + i - 1;
    while ( --i >= 0 ) {
        t += *b;
        *b-- = u = t / 10;
        t -= u * 10;
        if ( i > 0 ) t «= BITSINWORD;
    }
    if ( ( *na > 0 ) && !a[*na-1] ) (*na)--;
    return((WORD)t);
}

/*
        #] Remain10 :
        #[ Remain4 :        WORD Remain4(a,na)

    Routine devides a by 10000 and gives the remainder as return value.
    The value of a will be the quotient! a must be positive.

*/

WORD Remain4(UWORD *a, WORD *na)
{
    WORD i;
    RLONG t, u;
    UWORD *b;
    i = *na;
    t = 0;
    b = a + i - 1;
    while ( --i >= 0 ) {
        t += *b;
        *b-- = u = t / 10000;
        t -= u * 10000;
        if ( i > 0 ) t «= BITSINWORD;
    }
    if ( ( *na > 0 ) && !a[*na-1] ) (*na)--;
    return((WORD)t);
}

/*
        #] Remain4 :
        #[ PrtLong :        VOID PrtLong(a,na,s)

    Puts the long number a in string s.

*/

VOID PrtLong(UWORD *a, WORD na, UBYTE *s)
{
    GETIDENTITY
    WORD q, i;
    UBYTE *sa, *sb;
    UBYTE c;
    UWORD *bb, *b;

    if ( na < 0 ) {
        *s++ = '-';
        na = -na;
    }

    b = NumberMalloc("PrtLong");
    bb = b;
    i = na; while ( --i >= 0 ) *bb++ = *a++;
    a = b;
    if ( na > 2 ) {
        sa = s;
```

```
        do {
            q = Remain4(a,&na);
            *sa++ = (UBYTE)('0' + (q%10));
            q /= 10;
            *sa++ = (UBYTE)('0' + (q%10));
            q /= 10;
            *sa++ = (UBYTE)('0' + (q%10));
            q /= 10;
            *sa++ = (UBYTE)('0' + (q%10));
        } while ( na );
        while ( sa[-1] == '0' ) sa--;
        sb = s;
        s = sa;
        sa--;
        while ( sa > sb ) { c = *sa; *sa = *sb; *sb = c; sa--; sb++; }
    }
    else if ( na ) {
        sa = s;
        do {
            q = Remain10(a,&na);
            *sa++ = (UBYTE)('0' + q);
        } while ( na );
        sb = s;
        s = sa;
        sa--;
        while ( sa > sb ) { c = *sa; *sa = *sb; *sb = c; sa--; sb++; }
    }
    else *s++ = '0';
    *s = '\0';
    NumberFree(b,"PrtLong");
}

/*
        #] PrtLong :
        #[ GetLong :        WORD GetLong(s,a,na)

    Reads a long number from a string.
    The string is zero terminated and contains only digits!

    New algorithm: try to read 4 digits together before the result
    is accumulated.
*/

WORD GetLong(UBYTE *s, UWORD *a, WORD *na)
{
/*
    UWORD digit;
    *a = 0;
    *na = 0;
    while ( FG.cTable[*s] == 1 ) {
        digit = *s++ - '0';
        if ( *na && Product(a,na,(WORD)10) ) return(-1);
        if ( digit && AddLong(a,*na,&digit,(WORD)1,a,na) ) return(-1);
    }
    return(0);
*/
    UWORD digit, x = 0, y = 0;
    *a = 0;
    *na = 0;
    while ( FG.cTable[*s] == 1 ) {
        x = *s++ - '0';
        if ( FG.cTable[*s] != 1 ) { y = 10; break; }
        x = 10*x + *s++ - '0';
        if ( FG.cTable[*s] != 1 ) { y = 100; break; }
        x = 10*x + *s++ - '0';
        if ( FG.cTable[*s] != 1 ) { y = 1000; break; }
        x = 10*x + *s++ - '0';
        if ( *na && Product(a,na,(WORD)10000) ) return(-1);
        if ( ( digit = x ) != 0 && AddLong(a,*na,&digit,(WORD)1,a,na) )
            return(-1);
        y = 0;
    }
    if ( y ) {
        if ( *na && Product(a,na,(WORD)y) ) return(-1);
        if ( ( digit = x ) != 0 && AddLong(a,*na,&digit,(WORD)1,a,na) )
            return(-1);
    }
    return(0);
}

/*
        #] GetLong :
        #[ GCD :            WORD GCD(a,na,b,nb,c,nc)

    Algorithm to compute the GCD of two long numbers.
    See Knuth, sec 4.5.2 algorithm L.

```

```
    We assume that both numbers are positive

    NOTE!!!!!. NumberMalloc gets called and it may not be freed
*/

#ifdef EXTRAGCD

#define Convert(ia,aa,naa) \
    if ( (LONG)ia < 0 ) {          \
        ia = (ULONG)(-(LONG)ia); \
        aa[0] = ia;                \
        if ( ( aa[1] = ia » BITSINWORD ) != 0 ) naa = -2; \
        else naa = -1;             \
    }                              \
    else if ( ia == 0 ) { aa[0] = 0; naa = 0; } \
    else {                         \
        aa[0] = ia;                \
        if ( ( aa[1] = ia » BITSINWORD ) != 0 ) naa = 2; \
        else naa = 1;              \
    }

VOID GCD(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    int ja = 0, jb = 0, j;
    UWORD *r,*t;
    UWORD *x1, *x2, *x3;
    WORD nd,naa,nbb;
    ULONG ia,ib,ic,id,u,v,w,q,T;
    UWORD aa[2], bb[2];
/*
    First eliminate easy powers of 2^...
*/
    while ( a[0] == 0 ) { na--; ja++; a++; }
    while ( b[0] == 0 ) { nb--; jb++; b++; }
    if ( ja > jb ) ja = jb;
    if ( ja > 0 ) {
        j = ja;
        do { *c++ = 0; } while ( --j > 0 );
    }
/*
    Now arrange things such that a >= b
*/
    if ( na < nb ) {
        jb = na; na = nb; nb = jb;
exch:
        r = a; a = b; b = r;
    }
    else if ( na == nb ) {
        r = a+na;
        t = b+nb;
        j = na;
        while ( --j >= 0 ) {
            if ( *--r > *--t ) break;
            if ( *r < *t ) goto exch;
        }
        if ( j < 0 ) {
out:
            j = nb;
            NCOPY(c,b,j);
            *nc = nb+ja;
            return;
        }
    }
/*
    {
        MLOCK(ErrorMessageLock);
        MesPrint("Ordered input, ja = %d",(WORD)ja);
        AO.OutSkip = 3;
        FiniLine();
        j = na; r = a;
        while ( --j >= 0 ) { TalToLine((UWORD)(*r++)); TokenToLine((UBYTE *)"  "); }
        FiniLine();
        j = nb; r = b;
        while ( --j >= 0 ) { TalToLine((UWORD)(*r++)); TokenToLine((UBYTE *)"  "); }
        AO.OutSkip = 0;
        FiniLine();
        MUNLOCK(ErrorMessageLock);
    }
*/
/*
    We have now that A > B
    The loop recognizes the case that na-nb >= 1
    In that case we just have to divide!
*/
    r = x1 = NumberMalloc("GCD"); t = x2 = NumberMalloc("GCD"); x3 = NumberMalloc("GCD");
    j = na;
    NCOPY(r,a,j);
```

```
    j = nb;
    NCOPY(t,b,j);

    for(;;) {

        while ( na > nb ) {
toobad:
            DivLong(x1,na,x2,nb,c,nc,x3,&nd);
            if ( nd == 0 ) { b = x2; goto out; }
            t = x1; x1 = x2; x2 = x3; x3 = t; na = nb; nb = nd;
            if ( na == 2 ) break;
        }
/*
    Here we can use the shortcut.
*/
    if ( na == 2 ) {
        v = x1[0] + ( ((ULONG)x1[1]) « BITSINWORD );
        w = x2[0];
        if ( nb == 2 ) w += ((ULONG)x2[1]) « BITSINWORD;
#ifdef EXTRAGCD2
        v = GCD2(v,w);
#else
        do { u = v%w; v = w; w = u; } while ( w );
#endif
        c[0] = (UWORD)v;
        if ( ( c[1] = (UWORD)(v » BITSINWORD) ) != 0 ) *nc = 2+ja;
        else *nc = 1+ja;
        NumberFree(x1,"GCD"); NumberFree(x2,"GCD"); NumberFree(x3,"GCD");
        return;
    }
    if ( na == 1 ) {
        UWORD ui, uj;
        ui = x1[0]; uj = x2[0];
#ifdef EXTRAGCD2
        ui = (UWORD)GCD2((ULONG)ui,(ULONG)uj);
#else
        do { nd = ui%uj; ui = uj; uj = nd; } while ( nd );
#endif
        c[0] = ui;
        *nc = 1 + ja;
        NumberFree(x1,"GCD"); NumberFree(x2,"GCD"); NumberFree(x3,"GCD");
        return;
    }
    ia = 1; ib = 0; ic = 0; id = 1;
    u = ( ((ULONG)x1[na-1]) « BITSINWORD ) + x1[na-2];
    v = ( ((ULONG)x2[nb-1]) « BITSINWORD ) + x2[nb-2];

    while ( v+ic != 0 && v+id != 0 &&
         ( q = (u+ia)/(v+ic) ) == (u+ib)/(v+id) ) {
        T = ia-q*ic; ia = ic; ic = T;
        T = ib-q*id; ib = id; id = T;
        T = u - q*v; u = v; v = T;
    }
    if ( ib == 0 ) goto toobad;
    Convert(ia,aa,naa);
    Convert(ib,bb,nbb);
    MulLong(x1,na,aa,naa,x3,&nd);
    MulLong(x2,nb,bb,nbb,c,nc);
    AddLong(x3,nd,c,*nc,c,nc);
    Convert(ic,aa,naa);
    Convert(id,bb,nbb);
    MulLong(x1,na,aa,naa,x3,&nd);
    t = c; na = j = *nc; r = x1;
    NCOPY(r,t,j);
    MulLong(x2,nb,bb,nbb,c,nc);
    AddLong(x3,nd,c,*nc,x2,&nb);
    }
}

#endif

/*
        #] GCD :
        #[ GcdLong :        WORD GcdLong(a,na,b,nb,c,nc)

    Returns the Greatest Common Divider of a and b in c.
    If a and or b are zero an error message will be returned.
    The answer is always positive.
    In principle a and c can be the same.
*/

#ifndef NEWTRICK
/*
    #[ Old Routine :
*/

WORD GcdLong(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
```

```
{
    GETBIDENTITY
    if ( !na || !nb ) {
        if ( !na && !nb ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Cannot take gcd");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }

        if ( !na ) {
            *nc = abs(nb);
            NCOPY(c,b,*nc);
            *nc = abs(nb);
            return(0);
        }

        *nc = abs(na);
        NCOPY(c,a,*nc);
        *nc = abs(na);
        return(0);
    }
    if ( na < 0 ) na = -na;
    if ( nb < 0 ) nb = -nb;
    if ( na == 1 && nb == 1 ) {
#ifdef EXTRAGCD2
        *c = (UWORD)GCD2((ULONG)*a,(ULONG)*b);
#else
        UWORD x,y,z;
        x = *a;
        y = *b;
        do { z = x % y; x = y; } while ( ( y = z ) != 0 );
        *c = x;
#endif
        *nc = 1;
    }
    else if ( na <= 2 && nb <= 2 ) {
        RLONG lx,ly,lz;
        if ( na == 2 ) { lx = (((RLONG)(a[1]))«BITSINWORD) + *a; }
        else { lx = *a; }
        if ( nb == 2 ) { ly = (((RLONG)(b[1]))«BITSINWORD) + *b; }
        else { ly = *b; }
#ifdef LONGWORD
#ifdef EXTRAGCD2
        lx = GCD2(lx,ly);
#else
        do { lz = lx % ly; lx = ly; } while ( ( ly = lz ) != 0 );
#endif
#else
          if ( lx < ly ) { lz = lx; lx = ly; ly = lz; }
        do {
            lz = lx % ly; lx = ly;
        } while ( ( ly = lz ) != 0 && ( lx & AWORDMASK ) != 0 );
        if ( ly ) {
            do { *c = ((UWORD)lx)%((UWORD)ly); lx = ly; } while ( ( ly = *c ) != 0 );
            *c = (UWORD)lx;
            *nc = 1;
        }
        else
#endif
        {
            *c++ = (UWORD)lx;
            if ( ( *c = (UWORD)(lx » BITSINWORD) ) != 0 ) *nc = 2;
            else *nc = 1;
        }
    }
    else {
#ifdef EXTRAGCD
        GCD(a,na,b,nb,c,nc);
#else
#ifdef NEWGCD
        UWORD *x3,*x1,*x2, *GLscrat7, *GLscrat8;
        WORD n1,n2,n3,n4;
        WORD i, j;
        x1 = c; x3 = a; n1 = i = na;
        NCOPY(x1,x3,i);
        GLscrat7 = NumberMalloc("GcdLong"); GLscrat8 = NumberMalloc("GcdLong");
        x2 = GLscrat8; x3 = b; n2 = i = nb;
        NCOPY(x2,x3,i);
        x1 = c; i = 0;
        while ( x1[0] == 0 ) { i += BITSINWORD; x1++; n1--; }
        while ( ( x1[0] & 1 ) == 0 ) { i++; SCHUIF(x1,n1) }
        x2 = GLscrat8; j = 0;
        while ( x2[0] == 0 ) { j += BITSINWORD; x2++; n2--; }
        while ( ( x2[0] & 1 ) == 0 ) { j++; SCHUIF(x2,n2) }
        if ( j > i ) j = i;     /* powers of two in GCD */
        for(;;){
```

```
            if ( n1 > n2 ) {
firstbig:
                SubPLon(x1,n1,x2,n2,x1,&n3);
                n1 = n3;
                if ( n1 == 0 ) {
                    x1 = c;
                    n1 = i = n2; NCOPY(x1,x2,i);
                    break;
                }
                while ( ( x1[0] & 1 ) == 0 ) SCHUIF(x1,n1)
                if ( n1 == 1 ) {
                    if ( DivLong(x2,n2,x1,n1,GLscrat7,&n3,x2,&n4) ) goto GcdErr;
                    n2 = n4;
                    if ( n2 == 0 ) {
                        i = n1; x2 = c; NCOPY(x2,x1,i);
                        break;
                    }
#ifdef EXTRAGCD2
                    *c = (UWORD)GCD2((ULONG)x1[0],(ULONG)x2[0]);
#else
                    {
                        UWORD x,y,z;
                        x = x1[0];
                        y = x2[0];
                        do { z = x % y; x = y; } while ( ( y = z ) != 0 );
                        *c = x;
                    }
#endif
                    n1 = 1;
                    break;
                }
            }
            else if ( n1 < n2 ) {
lastbig:
                SubPLon(x2,n2,x1,n1,x2,&n3);
                n2 = n3;
                if ( n2 == 0 ) {
                    i = n1; x2 = c; NCOPY(x2,x1,i);
                    break;
                }
                while ( ( x2[0] & 1 ) == 0 ) SCHUIF(x2,n2)
                if ( n2 == 1 ) {
                    if ( DivLong(x1,n1,x2,n2,GLscrat7,&n3,x1,&n4) ) goto GcdErr;
                    n1 = n4;
                    if ( n1 == 0 ) {
                        x1 = c;
                        n1 = i = n2; NCOPY(x1,x2,i);
                        break;
                    }
#ifdef EXTRAGCD2
                    *c = (UWORD)GCD2((ULONG)x2[0],(ULONG)x1[0]);
#else
                    {
                        UWORD x,y,z;
                        x = x2[0];
                        y = x1[0];
                        do { z = x % y; x = y; } while ( ( y = z ) != 0 );
                        *c = x;
                    }
#endif
                    n1 = 1;
                    break;
                }
            }
            else {
                for ( i = n1-1; i >= 0; i-- ) {
                    if ( x1[i] > x2[i] ) goto firstbig;
                    else if ( x1[i] < x2[i] ) goto lastbig;
                }
                i = n1; x2 = c; NCOPY(x2,x1,i);
                break;
            }
        }
/*
        Now the GCD is in c but still needs j powers of 2.
*/
        x1 = c;
        while ( j >= BITSINWORD ) {
            for ( i = n1; i > 0; i-- ) x1[i] = x1[i-1];
            x1[0] = 0; n1++;
            j -= BITSINWORD;
        }
        if ( j > 0 ) {
            ULONG a1,a2 = 0;
            for ( i = 0; i < n1; i++ ) {
                a1 = x1[i]; a1 «= j;
                a2 += a1;
```

```
                x1[i] = a2;
                a2 »= BITSINWORD;
            }
            if ( a2 != 0 ) {
                x1[n1++] = a2;
            }
        }
        *nc = n1;
        NumberFree(GLscrat7,"GcdLong"); NumberFree(GLscrat8,"GcdLong");
#else
        UWORD *x1,*x2,*x3,*x4,*c1,*c2;
        WORD n1,n2,n3,n4,i;
        x1 = c; x3 = a; n1 = i = na;
        NCOPY(x1,x3,i);
        x1 = c; c1 = x2 = NumberMalloc("GcdLong"); x3 = NumberMalloc("GcdLong"); x4 = NumberMalloc("GcdLong");
        c2 = b; n2 = i = nb;
        NCOPY(c1,c2,i);
        for(;;){
            if ( DivLong(x1,n1,x2,n2,x4,&n4,x3,&n3) ) goto GcdErr;
            if ( !n3 ) { x1 = x2; n1 = n2; break; }
            if ( DivLong(x2,n2,x3,n3,x4,&n4,x1,&n1) ) goto GcdErr;
            if ( !n1 ) { x1 = x3; n1 = n3; break; }
            if ( DivLong(x3,n3,x1,n1,x4,&n4,x2,&n2) ) goto GcdErr;
            if ( !n2 ) {
                *nc = n1;
                NumberFree(x2,"GcdLong"); NumberFree(x3,"GcdLong"); NumberFree(x4,"GcdLong");
                return(0);
            }
        }
        *nc = i = n1;
        NCOPY(c,x1,i);
        NumberFree(x2,"GcdLong"); NumberFree(x3,"GcdLong"); NumberFree(x4,"GcdLong");
#endif
#endif
    }
    return(0);
GcdErr:
    MLOCK(ErrorMessageLock);
    MesCall("GcdLong");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}
/*
    #] Old Routine :
*/
#else

/*
    New routine for GcdLong that uses smart shortcuts.
    Algorithm by J. Vermaseren 15-nov-2006.
    It runs faster for very big numbers but only by a fixed factor.
    There is no improvement in the power behaviour.
    Improvement on the whole of hf9 (multiple zeta values at weight 9):
        Better than a factor 2 on a 32 bits architecture and 2.76 on a
        64 bits architecture.
    On hf10 (MZV's at weight 10), 64 bits architecture: factor 7.

    If we have two long numbers (na,nb > GCDMAX) we will work in a
    truncated way. At the moment of writing (15-nov-2006) it isn't
    clear whether this algorithm is an invention or a reinvention.
    A short search on the web didn't show anything.

    31-jul-2007:
    A better search shows that this is an adaptation of the Lehmer-Euclid
    algorithm, already described in Knuth. Here we can work without upper
    and lower limit because we are only interested in the GCD, not the
    extra numbers. Also it takes already some features of the double
    digit Lehmer-Euclid algorithm of Jebelean it seems.

    Maybe this can be programmed slightly better and we can get another
    few percent speed increase. Further improvements for the assymptotic
    case come from splitting the calculation as in Karatsuba and working
    with FFT divisions and multiplications etc. But this is when hundreds
    of words are involved at the least.

    Algorithm

    1: while ( na > nb || nb < GCDMAX ) {
        if ( nb == 0 ) { result in a }
        c = a % b;
        a = b;
        b = c;
       }
    2: Make the truncated values in which a and b are the combinations
       of the top two words of a and b. The whole numbers are aa and bb now.
    3: ma1 = 1; ma2 = 0; mb1 = 0; mb2 = 1;
```

```
    4: A = a; B = b; m = a/b; c = a - m*b;
       c = ma1*a+ma2*b-m*(mb1*a+mb2*b) = (ma1-m*mb1)*a+(ma2-m*mb2)*b
       mc1 = ma1-m*mb1; mc2 = ma2-m*mb2;
    5: a = b; ma1 = mb1; ma2 = mb2;
       b = c; mb1 = mc1; mb2 = mc2;
    6: if ( b != 0 && nb >= FULLMAX ) goto 4;
    7: Now construct the new quantities
        ma1*aa+ma2*bb and mb1*aa+mb2*bb
    8: goto 1;

    The essence of the above algorithm is that we do the divisions only
    on relatively short numbers. Also usually there are many steps 4&5
    for each step 7. This eliminates many operations.
    The termination at FULLMAX is that we make errors by not considering
    the tail of the number. If we run b down all the way, the errors combine
    in such a way that the new numbers may be of the same order as the old
    numbers. By stopping halfway we don't get the error beyond halfway
    either. Unfortunately this means that a >= FULLMAX and hence na > nb
    which means that next we will have a complete division. But just once.
    Running the steps 4-6 till a < FULLMAX runs already into problems.
    It may be necessary to experiment a bit to obtain the optimum value
    of GCDMAX.
*/

WORD GcdLong(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    GETBIDENTITY
    UWORD x,y,z;
    UWORD *x1,*x2,*x3,*x4,*x5,*d;
    UWORD *GLscrat6, *GLscrat7, *GLscrat8, *GLscrat9, *GLscrat10;
    WORD n1,n2,n3,n4,n5,i;
    RLONG lx,ly,lz;
    LONG ma1, ma2, mb1, mb2, mc1, mc2, m;
    if ( !na || !nb ) {
        if ( !na && !nb ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Cannot take gcd");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }

        if ( !na ) {
            *nc = abs(nb);
            NCOPY(c,b,*nc);
            *nc = abs(nb);
            return(0);
        }

        *nc = abs(na);
        NCOPY(c,a,*nc);
        *nc = abs(na);
        return(0);
    }
    if ( na < 0 ) na = -na;
    if ( nb < 0 ) nb = -nb;
/*
    #[ GMP stuff :
*/
#ifdef WITHGMP
    if ( na > 3 && nb > 3 ) {
        int ii;
        mp_limb_t *upa, *upb, *upc, xx;
        UWORD *uw, *u1, *u2;
        unsigned int tcounta, tcountb, tcounta1, tcountb1;
        mp_size_t ana, anb, anc;

        u1 = uw = NumberMalloc("GcdLong");
        upa = (mp_limb_t *)u1;
        ana = na; tcounta1 = 0;
        while ( a[0] == 0 ) { a++; ana--; tcounta1++; }
        for ( ii = 0; ii < ana; ii++ ) { *uw++ = *a++; }
        if ( ( ana & 1 ) != 0 ) { *uw = 0; ana++; }
        ana >>= 1;

        u2 = uw = NumberMalloc("GcdLong");
        upb = (mp_limb_t *)u2;
        anb = nb; tcountb1 = 0;
        while ( b[0] == 0 ) { b++; anb--; tcountb1++; }
        for ( ii = 0; ii < anb; ii++ ) { *uw++ = *b++; }
        if ( ( anb & 1 ) != 0 ) { *uw = 0; anb++; }
        anb >>= 1;

        xx = upa[0]; tcounta = 0;
        while ( ( xx & 15 ) == 0 ) { tcounta += 4; xx >>= 4; }
        while ( ( xx &  1 ) == 0 ) { tcounta += 1; xx >>= 1; }
        xx = upb[0]; tcountb = 0;
        while ( ( xx & 15 ) == 0 ) { tcountb += 4; xx >>= 4; }
```

```
        while ( ( xx &  1 ) == 0 ) { tcountb += 1; xx »= 1; }

        if ( tcounta ) {
            mpn_rshift(upa,upa,ana,tcounta);
            if ( upa[ana-1] == 0 ) ana--;
        }
        if ( tcountb ) {
            mpn_rshift(upb,upb,anb,tcountb);
            if ( upb[anb-1] == 0 ) anb--;
        }

        upc = (mp_limb_t *)(NumberMalloc("GcdLong"));
        if ( ( ana > anb ) || ( ( ana == anb ) && ( upa[ana-1] >= upb[ana-1] ) ) ) {
            anc = mpn_gcd(upc,upa,ana,upb,anb);
        }
        else {
            anc = mpn_gcd(upc,upb,anb,upa,ana);
        }

        tcounta = tcounta1*BITSINWORD + tcounta;
        tcountb = tcountb1*BITSINWORD + tcountb;
        if ( tcountb > tcounta ) tcountb = tcounta;
        tcounta = tcountb/BITSINWORD;
        tcountb = tcountb%BITSINWORD;

        if ( tcountb ) {
            xx = mpn_lshift(upc,upc,anc,tcountb);
            if ( xx ) { upc[anc] = xx; anc++; }
        }

        uw = (UWORD *)upc; anc *= 2;
        while ( uw[anc-1] == 0 ) anc--;
        for ( ii = 0; ii < (int)tcounta; ii++ ) *c++ = 0;
        for ( ii = 0; ii < anc; ii++ ) *c++ = *uw++;
        *nc = anc + tcounta;
        NumberFree(u1,"GcdLong"); NumberFree(u2,"GcdLong"); NumberFree((UWORD *)(upc),"GcdLong");
        return(0);
    }
#endif
/*
    #] GMP stuff :
*/
/*
    #[ Easy cases :
*/
    if ( na == 1 && nb == 1 ) {
        x = *a;
        y = *b;
        do { z = x % y; x = y; } while ( ( y = z ) != 0 );
        *c = x;
        *nc = 1;
        return(0);
    }
    else if ( na <= 2 && nb <= 2 ) {
        if ( na == 2 ) { lx = (((RLONG)(a[1]))«BITSINWORD) + *a; }
        else { lx = *a; }
        if ( nb == 2 ) { ly = (((RLONG)(b[1]))«BITSINWORD) + *b; }
        else { ly = *b; }
        if ( lx < ly ) { lz = lx; lx = ly; ly = lz; }
#if ( BITSINWORD == 16 )
        do {
            lz = lx % ly; lx = ly;
        } while ( ( ly = lz ) != 0 );
#else
        do {
            lz = lx % ly; lx = ly;
        } while ( ( ly = lz ) != 0 && ( lx & AWORDMASK ) != 0 );
        if ( ly ) {
            x = (UWORD)lx; y = (UWORD)ly;
            do { *c = x % y; x = y; } while ( ( y = *c ) != 0 );
            *c = x;
            *nc = 1;
        }
        else
#endif
        {
            *c++ = (UWORD)lx;
            if ( ( *c = (UWORD)(lx » BITSINWORD) ) != 0 ) *nc = 2;
            else *nc = 1;
        }
        return(0);
    }
/*
    #] Easy cases :
*/
    GLscrat6 = NumberMalloc("GcdLong"); GLscrat7 = NumberMalloc("GcdLong");
    GLscrat8 = NumberMalloc("GcdLong");
```

```
    GLscrat9 = NumberMalloc("GcdLong"); GLscrat10 = NumberMalloc("GcdLong");
restart:;
/*
    #[ Easy cases :
*/
    if ( na == 1 && nb == 1 ) {
        x = *a;
        y = *b;
        do { z = x % y; x = y; } while ( ( y = z ) != 0 );
        *c = x;
        *nc = 1;
    }
    else if ( na <= 2 && nb <= 2 ) {
        if ( na == 2 ) { lx = (((RLONG)(a[1]))«BITSINWORD) + *a; }
        else { lx = *a; }
        if ( nb == 2 ) { ly = (((RLONG)(b[1]))«BITSINWORD) + *b; }
        else { ly = *b; }
        if ( lx < ly ) { lz = lx; lx = ly; ly = lz; }
#if ( BITSINWORD == 16 )
        do {
            lz = lx % ly; lx = ly;
        } while ( ( ly = lz ) != 0 );
#else
        do {
            lz = lx % ly; lx = ly;
        } while ( ( ly = lz ) != 0 && ( lx & AWORDMASK ) != 0 );
        if ( ly ) {
            x = (UWORD)lx; y = (UWORD)ly;
            do { *c = x % y; x = y; } while ( ( y = *c ) != 0 );
            *c = x;
            *nc = 1;
        }
        else
#endif
        {
            *c++ = (UWORD)lx;
            if ( ( *c = (UWORD)(lx » BITSINWORD) ) != 0 ) *nc = 2;
            else *nc = 1;
        }
    }
/*
    #] Easy cases :
    #[ Original code :
*/
    else if ( na < GCDMAX || nb < GCDMAX || na != nb ) {
        if ( na < nb ) {
            x2 = GLscrat8; x3 = a; n2 = i = na;
            NCOPY(x2,x3,i);
            x1 = c; x3 = b; n1 = i = nb;
            NCOPY(x1,x3,i);
        }
        else {
            x1 = c; x3 = a; n1 = i = na;
            NCOPY(x1,x3,i);
            x2 = GLscrat8; x3 = b; n2 = i = nb;
            NCOPY(x2,x3,i);
        }
        x1 = c; x2 = GLscrat8; x3 = GLscrat7; x4 = GLscrat6;
        for(;;){
            if ( DivLong(x1,n1,x2,n2,x4,&n4,x3,&n3) ) goto GcdErr;
            if ( !n3 ) { x1 = x2; n1 = n2; break; }
            if ( n2 <= 2 ) { a = x2; b = x3; na = n2; nb = n3; goto restart; }
            if ( n3 >= GCDMAX && n2 == n3 ) {
                a = GLscrat9; b = GLscrat10; na = n2; nb = n3;
                for ( i = 0; i < na; i++ ) a[i] = x2[i];
                for ( i = 0; i < nb; i++ ) b[i] = x3[i];
                goto newtrick;
            }
            if ( DivLong(x2,n2,x3,n3,x4,&n4,x1,&n1) ) goto GcdErr;
            if ( !n1 ) { x1 = x3; n1 = n3; break; }
            if ( n3 <= 2 ) { a = x3; b = x1; na = n3; nb = n1; goto restart; }
            if ( n1 >= GCDMAX && n1 == n3 ) {
                a = GLscrat9; b = GLscrat10; na = n3; nb = n1;
                for ( i = 0; i < na; i++ ) a[i] = x3[i];
                for ( i = 0; i < nb; i++ ) b[i] = x1[i];
                goto newtrick;
            }
            if ( DivLong(x3,n3,x1,n1,x4,&n4,x2,&n2) ) goto GcdErr;
            if ( !n2 ) { *nc = n1; goto normalend; }
            if ( n1 <= 2 ) { a = x1; b = x2; na = n1; nb = n2; goto restart; }
            if ( n2 >= GCDMAX && n2 == n1 ) {
                a = GLscrat9; b = GLscrat10; na = n1; nb = n2;
                for ( i = 0; i < na; i++ ) a[i] = x1[i];
                for ( i = 0; i < nb; i++ ) b[i] = x2[i];
                goto newtrick;
            }
        }
```

```
        *nc = i = n1;
        NCOPY(c,x1,i);
    }
/*
    #] Original code :
    #[ New code :
*/
    else {
/*
        This is the new algorithm starting at step 3.

    3: ma1 = 1; ma2 = 0; mb1 = 0; mb2 = 1;
    4: A = a; B = b; m = a/b; c = a - m*b;
       c = ma1*a+ma2*b-m*(mb1*a+mb2*b) = (ma1-m*mb1)*a+(ma2-m*mb2)*b
       mc1 = ma1-m*mb1; mc2 = ma2-m*mb2;
    5: a = b; ma1 = mb1; ma2 = mb2;
       b = c; mb1 = mc1; mb2 = mc2;
    6: if ( b != 0 ) goto 4;
*/
newtrick:;
        ma1 = 1; ma2 = 0; mb1 = 0; mb2 = 1;
        lx = (((RLONG)(a[na-1]))«BITSINWORD) + a[na-2];
        ly = (((RLONG)(b[nb-1]))«BITSINWORD) + b[nb-2];
        if ( ly > lx ) { lz = lx; lx = ly; ly = lz; d = a; a = b; b = d; }
        do {
            m = lx/ly;
            mc1 = ma1-m*mb1; mc2 = ma2-m*mb2;
            ma1 = mb1; ma2 = mb2; mb1 = mc1; mb2 = mc2;
            lz = lx - m*ly; lx = ly; ly = lz;
        } while ( ly >= FULLMAX );
/*
        Next the construction of the two new numbers

    7: Now construct the new quantities
        a = ma1*aa+ma2*bb and b = mb1*aa+mb2*bb
*/
        x1 = GLscrat6;
        x2 = GLscrat7;
        x3 = GLscrat8;
        x5 = GLscrat10;
        if ( ma1 < 0 ) {
            ma1 = -ma1;
            x1[0] = (UWORD)ma1;
            x1[1] = (UWORD)(ma1 » BITSINWORD);
            if ( x1[1] ) n1 = -2;
            else         n1 = -1;
        }
        else {
            x1[0] = (UWORD)ma1;
            x1[1] = (UWORD)(ma1 » BITSINWORD);
            if ( x1[1] ) n1 = 2;
            else         n1 = 1;
        }
        if ( MulLong(a,na,x1,n1,x2,&n2) ) goto GcdErr;
        if ( ma2 < 0 ) {
            ma2 = -ma2;
            x1[0] = (UWORD)ma2;
            x1[1] = (UWORD)(ma2 » BITSINWORD);
            if ( x1[1] ) n1 = -2;
            else         n1 = -1;
        }
        else {
            x1[0] = (UWORD)ma2;
            x1[1] = (UWORD)(ma2 » BITSINWORD);
            if ( x1[1] ) n1 = 2;
            else         n1 = 1;
        }
        if ( MulLong(b,nb,x1,n1,x3,&n3) ) goto GcdErr;
        if ( AddLong(x2,n2,x3,n3,c,&n4) ) goto GcdErr;
        if ( mb1 < 0 ) {
            mb1 = -mb1;
            x1[0] = (UWORD)mb1;
            x1[1] = (UWORD)(mb1 » BITSINWORD);
            if ( x1[1] ) n1 = -2;
            else         n1 = -1;
        }
        else {
            x1[0] = (UWORD)mb1;
            x1[1] = (UWORD)(mb1 » BITSINWORD);
            if ( x1[1] ) n1 = 2;
            else         n1 = 1;
        }
        if ( MulLong(a,na,x1,n1,x2,&n2) ) goto GcdErr;
        if ( mb2 < 0 ) {
            mb2 = -mb2;
            x1[0] = (UWORD)mb2;
            x1[1] = (UWORD)(mb2 » BITSINWORD);
```

```
            if ( x1[1] ) n1 = -2;
            else         n1 = -1;
        }
        else {
            x1[0] = (UWORD)mb2;
            x1[1] = (UWORD)(mb2 » BITSINWORD);
            if ( x1[1] ) n1 = 2;
            else         n1 = 1;
        }
        if ( MulLong(b,nb,x1,n1,x3,&n3) ) goto GcdErr;
        if ( AddLong(x2,n2,x3,n3,x5,&n5) ) goto GcdErr;
        a = c; na = n4; b = x5; nb = n5;
        if ( nb == 0 ) { *nc = n4; goto normalend; }
        x4 = GLscrat9;
        for ( i = 0; i < na; i++ ) x4[i] = a[i];
        a = x4;
        if ( na < 0 ) na = -na;
        if ( nb < 0 ) nb = -nb;
/*
        The typical case now is that in a we have the last step to go
        to loose the leading word, while in b we have lost the leading word.
        We could go to DivLong now but we can also add an extra step that
        is less wasteful.
        In the case that the new leading word of b is extrememly short (like 1)
        we make a rather large error of course. In the worst case the whole
        will be intercepted by DivLong after all, but that is so rare that
        it shouldn't influence any timing in a measurable way.
*/
        if ( nb >= GCDMAX && na == nb+1 && b[nb-1] >= HALFMAX && b[nb-1] > a[na-1] ) {
            lx = (((RLONG)(a[na-1]))«BITSINWORD) + a[na-2];
            x1[0] = lx/b[nb-1]; n1 = 1;
            MulLong(b,nb,x1,n1,x2,&n2);
            n2 = -n2;
            AddLong(a,na,x2,n2,x4,&n4);
            if ( n4 == 0 ) {
                *nc = nb;
                for ( i = 0; i < nb; i++ ) c[i] = b[i];
                goto normalend;
            }
            if ( n4 < 0 ) n4 = -n4;
            a = b; na = nb; b = x4; nb = n4;
        }
        goto restart;
/*
    #] New code :
*/
    }
normalend:
    NumberFree(GLscrat6,"GcdLong"); NumberFree(GLscrat7,"GcdLong"); NumberFree(GLscrat8,"GcdLong");
    NumberFree(GLscrat9,"GcdLong"); NumberFree(GLscrat10,"GcdLong");
    return(0);
GcdErr:
    MLOCK(ErrorMessageLock);
    MesCall("GcdLong");
    MUNLOCK(ErrorMessageLock);
    NumberFree(GLscrat6,"GcdLong"); NumberFree(GLscrat7,"GcdLong"); NumberFree(GLscrat8,"GcdLong");
    NumberFree(GLscrat9,"GcdLong"); NumberFree(GLscrat10,"GcdLong");
    SETERROR(-1)
}

#endif

/*
        #] GcdLong :
        #[ GetBinom :       WORD GetBinom(a,na,i1,i2)
*/

WORD GetBinom(UWORD *a, WORD *na, WORD i1, WORD i2)
{
    GETIDENTITY
    WORD j, k, l;
    UWORD *GBscrat3, *GBscrat4;
    if ( i1-i2 < i2 ) i2 = i1-i2;
    if ( i2 == 0 ) { *a = 1; *na = 1; return(0); }
    if ( i2 > i1 ) { *a = 0; *na = 0; return(0); }
    *a = i1; *na = 1;
    GBscrat3 = NumberMalloc("GetBinom"); GBscrat4 = NumberMalloc("GetBinom");
    for ( j = 2; j <= i2; j++ ) {
        GBscrat3[0] = i1+1-j;
        if ( MulLong(a,*na,GBscrat3,(WORD)1,GBscrat4,&k) ) goto CalledFrom;
        GBscrat3[0] = j;
        if ( DivLong(GBscrat4,k,GBscrat3,(WORD)1,a,na,GBscrat3,&l) ) goto CalledFrom;
    }
    NumberFree(GBscrat3,"GetBinom"); NumberFree(GBscrat4,"GetBinom");
    return(0);
CalledFrom:
    MLOCK(ErrorMessageLock);
```

```
    MesCall("GetBinom");
    MUNLOCK(ErrorMessageLock);
    NumberFree(GBscrat3,"GetBinom"); NumberFree(GBscrat4,"GetBinom");
    SETERROR(-1)
}

/*
        #] GetBinom :
        #[ LcmLong :        WORD LcmLong(a,na,b,nb)

        Computes the LCM of the long numbers a and b and puts the result
        in c. c is allowed to be equal to a.
*/

WORD LcmLong(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
{
    WORD error = 0;
    UWORD *d = NumberMalloc("LcmLong");
    UWORD *e = NumberMalloc("LcmLong");
    UWORD *f = NumberMalloc("LcmLong");
    WORD nd, ne, nf;
    GcdLong(BHEAD a, na, b, nb, d, &nd);
    DivLong(a,na,d,nd,e,&ne,f,&nf);
    if ( MulLong(b,nb,e,ne,c,nc) ) {
        MLOCK(ErrorMessageLock);
        MesCall("LcmLong");
        MUNLOCK(ErrorMessageLock);
        error = -1;
    }
    NumberFree(f,"LcmLong");
    NumberFree(e,"LcmLong");
    NumberFree(d,"LcmLong");
    return(error);
}

/*
        #] LcmLong :
        #[ TakeLongRoot:    int TakeLongRoot(a,n,power)

    Takes the 'power'-root of the long number in a.
    If the root could be taken the return value is zero.
    If the root could not be taken, the return value is 1.
    The root will be in a if it could be taken, otherwise there will be garbage
    Algorithm: (assume b is guess of root, b' better guess)
        b' = (a-(power-1)*b^power)/(n*b^(power-1))
    Note: power should be positive!
*/

int TakeLongRoot(UWORD *a, WORD *n, WORD power)
{
    GETIDENTITY
    int numbits, guessbits, i, retval = 0;
    UWORD x, *b, *c, *d, *e;
    WORD na, nb, nc, nd, ne;
    if ( *n < 0 && ( power & 1 ) == 0 ) return(1);
    if ( power == 1 ) return(0);
    if ( *n < 0 ) { na = -*n; }
    else          { na =  *n; }
    if ( na == 1 ) {
/*          Special cases that are the most frequent */
        if ( a[0] == 1 ) return(0);
        if ( power < BITSINWORD && na == 1 && a[0] == (UWORD)(1«power) ) {
            a[0] = 2; return(0);
        }
        if ( 2*power < BITSINWORD && na == 1 && a[0] == (UWORD)(1«(2*power)) ) {
            a[0] = 4; return(0);
        }
    }
/*
    Step 1: make a guess. We count the number of bits.
    numbits will be the 1+2log(a)
*/
    numbits = BITSINWORD*(na-1);
    x = a[na-1];
    while ( ( x » 8 ) != 0 ) { numbits += 8; x »= 8; }
    if ( ( x » 4 ) != 0 ) { numbits += 4; x »= 4; }
    if ( ( x » 2 ) != 0 ) { numbits += 2; x »= 2; }
    if ( ( x » 1 ) != 0 ) numbits++;
    guessbits = numbits / power;
    if ( guessbits <= 0 ) return(1); /* root < 2 and 1 we did already */
    nb = guessbits/BITSINWORD;
/*
    The recursion is:
    (b'-b) = (a/b^(power-1)-b)/n
           = (a/c-b)/n
           = (d-b)/n    (remainder of a/c is e)
           = c/n  (we reuse the scratch array c)
```

```
    Termination can be tricky. When a/c has no remainder and = b we have a root.
    When d = b but the remainder of a/c != 0, there is definitely no root.
*/
    b = NumberMalloc("TakeLongRoot"); c = NumberMalloc("TakeLongRoot");
    d = NumberMalloc("TakeLongRoot"); e = NumberMalloc("TakeLongRoot");
    for ( i = 0; i < nb; i++ ) { b[i] = 0; }
    b[nb] = 1 « (guessbits%BITSINWORD);
    nb++;
    for(;;) {
        nc = nb;
        for ( i = 0; i < nb; i++ ) c[i] = b[i];
        if ( RaisPow(BHEAD c,&nc,power-1) ) goto TLcall;
        if ( DivLong(a,na,c,nc,d,&nd,e,&ne) ) goto TLcall;
        nb = -nb;
        if ( AddLong(d,nd,b,nb,c,&nc) ) goto TLcall;
        nb = -nb;
        if ( nc == 0 ) {
            if ( ne == 0 ) break;
            retval = 1; break;
/*
            else {
                NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
                NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
                return(1);
            }
*/
        }
        DivLong(c,nc,(UWORD *)(&power),1,d,&nd,e,&ne);
        if ( nd == 0 ) {
            retval = 1;
            break;
/*
            NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
            NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
            return(1);
*/
/*
            This code tries b+1 as a final possibility.
            We believe this is not needed
            UWORD one = 1;
            if ( AddLong(b,nb,&one,1,c,&nc) ) goto TLcall;
            if ( RaisPow(BHEAD c,&nc,power-1) ) goto TLcall;
            if ( DivLong(a,na,c,nc,d,&nd,e,&ne) ) goto TLcall;
            if ( ne != 0 ) return(1);
            nb = -nb;
            if ( SubLong(d,nd,b,nb,c,&nc) ) goto TLcall;
            nb = -nb;
            if ( nc != 0 ) {
                NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
                NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
                return(1);
            }
            break;
*/
        }
        if ( AddLong(b,nb,d,nd,b,&nb) ) goto TLcall;
    }
    for ( i = 0; i < nb; i++ ) a[i] = b[i];
    if ( *n < 0 ) *n = -nb;
    else          *n =  nb;
    NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
    NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
    return(retval);
TLcall:
    MLOCK(ErrorMessageLock);
    MesCall("TakeLongRoot");
    MUNLOCK(ErrorMessageLock);
    NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
    NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
    Terminate(-1);
    return(-1);
}

/*
        #] TakeLongRoot:
        #[ MakeRational:

        Makes the integer a mod m into a traction b/c with |b|,|c| < sqrt(m)
        For the algorithm, see MakeLongRational.
*/

int MakeRational(WORD a,WORD m, WORD *b, WORD *c)
{
    LONG x1,x2,x3,x4,y1,y2;
    if ( a < 0 ) { a = a+m; }
    if ( a <= 1 ) {
        if ( a > m/2 ) a = a-m;
```

```
        *b = a; *c = 1; return(0);
    }
    x1 = m; x2 = a;
    if ( x2*x2 >= m ) {
        y1 = x1/x2; y2 = x1%x2; x3 = 1; x4 = -y1; x1 = x2; x2 = y2;
        while ( x2*x2 >= m ) {
            y1 = x1/x2; y2 = x1%x2; x1 = x2; x2 = y2; y2 = x3-y1*x4; x3 = x4; x4 = y2;
        }
    }
    else x4 = 1;
    if ( x2 == 0 ) { return(1); }
    if ( x2 > m/2 ) *b = x2-m;
    else            *b = x2;
    if ( x4 > m/2 ) { *c = x4-m; *c = -*c; *b = -*b; }
    else if ( x4 <= -m/2 ) { x4 += m; *c = x4; }
    else if ( x4 < 0 ) { x4 = -x4; *c = x4; *b = -*b; }
    else            *c = x4;
    return(0);
}

/*
        #] MakeRational:
        #[ MakeLongRational:

        Converts the long number a mod m into the fraction b
        One of the properties of b is that num,den < sqrt(m)
        The algorithm: Start with:   m 0
                                     a 1
        Make now c=m%a, c1=m/a       c c2=0-c1*1
        Make now d=a%c  d1=a/c       d d2=1-d1*c2
        Make now e=c%d  e1=c/d       e e2=1-e1*d2
            etc till in the first column we get a number < sqrt(m)
            We have then f,f2 and the fraction is f/f2.
        If at any moment we get a zero, m contained an unlucky prime.

        Note that this can be made a lot faster when we make the same
        improvements as in the GCD routine. That is something for later.
#ifdef WITHMAKERATIONAL
*/

#define COPYLONG(x1,nx1,x2,nx2) { int i; for(i=0;i<ABS(nx2);i++)x1[i]=x2[i];nx1=nx2; }

int MakeLongRational(PHEAD UWORD *a, WORD na, UWORD *m, WORD nm, UWORD *b, WORD *nb)
{
    UWORD *root = NumberMalloc("MakeRational");
    UWORD *x1 = NumberMalloc("MakeRational");
    UWORD *x2 = NumberMalloc("MakeRational");
    UWORD *x3 = NumberMalloc("MakeRational");
    UWORD *x4 = NumberMalloc("MakeRational");
    UWORD *y1 = NumberMalloc("MakeRational");
    UWORD *y2 = NumberMalloc("MakeRational");
    WORD nroot,nx1,nx2,nx3,nx4,ny1,ny2,retval = 0;
    WORD sign = 1;
/*
    Step 1: Take the square root of m
*/
    COPYLONG(root,nroot,m,nm)
    TakeLongRoot(root,&nroot,2);
/*
    Step 2: Set the start values
*/
    if ( na < 0 ) { na = -na; sign = -sign; }
    COPYLONG(x1,nx1,m,nm)
    COPYLONG(x2,nx2,a,na)
/*
    x3[0] = 0, nx3 = 0;
    x4[0] = 1, nx4 = 1;
*/
/*
    The start operation needs some special attention because of the zero.
*/
    if ( BigLong(x2,nx2,root,nroot) <= 0 ) {
        x4[0] = 1, nx4 = 1;
        goto gottheanswer;
    }
    DivLong(x1,nx1,x2,nx2,y1,&ny1,y2,&ny2);
    if ( ny2 == 0 ) { retval = 1; goto cleanup; }
    COPYLONG(x1,nx1,x2,nx2)
    COPYLONG(x2,nx2,y2,ny2)
    x3[0] = 1; nx3 = 1;
    COPYLONG(x4,nx4,y1,ny1)
    nx4 = -nx4;
/*
    Now the loop.
*/
    while ( BigLong(x2,nx2,root,nroot) > 0 ) {
        DivLong(x1,nx1,x2,nx2,y1,&ny1,y2,&ny2);
```

```
        if ( ny2 == 0 ) { retval = 1; goto cleanup; }
        COPYLONG(x1,nx1,x2,nx2)
        COPYLONG(x2,nx2,y2,ny2)
        MulLong(y1,ny1,x4,nx4,y2,&ny2);
        ny2 = -ny2;
        AddLong(x3,nx3,y2,ny2,y1,&ny1);
        COPYLONG(x3,nx3,x4,nx4)
        COPYLONG(x4,nx4,y1,ny1)
    }
/*
    Now we have the answer. It is x2/x4. It has to be packed into b.
*/
gottheanswer:
    if ( nx4 < 0 ) { sign = -sign; nx4 = -nx4; }
    COPYLONG(b,*nb,x2,nx2)
    Pack(b,nb,x4,nx4);
    if ( sign < 0 ) *nb = -*nb;
cleanup:
    NumberFree(y2,"MakeRational");
    NumberFree(y1,"MakeRational");
    NumberFree(x4,"MakeRational");
    NumberFree(x3,"MakeRational");
    NumberFree(x2,"MakeRational");
    NumberFree(x1,"MakeRational");
    NumberFree(root,"MakeRational");
    return(retval);
}

/*
#endif
        #] MakeLongRational:
        #[ ChineseRemainder:
*/
#ifdef WITHCHINESEREMAINDER

int ChineseRemainder(PHEAD MODNUM *a1, MODNUM *a2, MODNUM *a)
{
    UWORD *inv1 = NumberMalloc("ChineseRemainder");
    UWORD *inv2 = NumberMalloc("ChineseRemainder");
    UWORD *fac1 = NumberMalloc("ChineseRemainder");
    UWORD *fac2 = NumberMalloc("ChineseRemainder");
    UWORD two[1];
    WORD ninv1, ninv2, nfac1, nfac2;
    if ( a1->na < 0 ) {
        AddLong(a1->a,a1->na,a1->m,a1->nm,a1->a,&(a1->na));
    }
    if ( a2->na < 0 ) {
        AddLong(a2->a,a2->na,a2->m,a2->nm,a2->a,&(a2->na));
    }
    MulLong(a1->m,a1->nm,a2->m,a2->nm,a->m,&(a->nm));

    GetLongModInverses(BHEAD a1->m,a1->nm,a2->m,a2->nm,inv1,&ninv1,inv2,&ninv2);
    MulLong(inv1,ninv1,a1->m,a1->nm,fac1,&nfac1);
    MulLong(inv2,ninv2,a2->m,a2->nm,fac2,&nfac2);

    MulLong(fac1,nfac1,a2->a,a2->na,inv1,&ninv1);
    MulLong(fac2,nfac2,a1->a,a1->na,inv2,&ninv2);
    AddLong(inv1,ninv1,inv2,ninv2,a->a,&(a->na));

    two[0] = 2;
    MulLong(a->a,a->na,two,1,fac1,&nfac1);
    if ( BigLong(fac1,nfac1,a->m,a->nm) > 0 ) {
        a->nm = -a->nm;
        AddLong(a->a,a->na,a->m,a->nm,a->a,&(a->na));
        a->nm = -a->nm;
    }
    NumberFree(fac2,"ChineseRemainder");
    NumberFree(fac1,"ChineseRemainder");
    NumberFree(inv2,"ChineseRemainder");
    NumberFree(inv1,"ChineseRemainder");
    return(0);
}

#endif

/*
        #] ChineseRemainder:
    #] RekenLong :
    #[ RekenTerms :
        #[ CompCoef :       WORD CompCoef(term1,term2)

    Compares the coefficients of term1 and term2 by subtracting them.
    This does more work than needed but this routine is only called
    when sorting functions and function arguments.
    (and comparing values
*/
/* #define 64SAVE */
```

```
WORD CompCoef(WORD *term1, WORD *term2)
{
    GETIDENTITY
    UWORD *c;
    WORD n1,n2,n3,*a;
    GETCOEF(term1,n1);
    GETCOEF(term2,n2);
    if ( term1[1] == 0 && n1 == 1 ) {
        if ( term2[1] == 0 && n2 == 1 ) return(0);
        if ( n2 < 0 ) return(1);
        return(-1);
    }
    else if ( term2[1] == 0 && n2 == 1 ) {
        if ( n1 < 0 ) return(-1);
        return(1);
    }
    if ( n1 > 0 ) {
        if ( n2 < 0 ) return(1);
    }
    else {
        if ( n2 > 0 ) return(-1);
        a = term1; term1 = term2; term2 = a;
        n3 = -n1; n1 = -n2; n2 = n3;
    }
    if ( term1[1] == 1 && term2[1] == 1 && n1 == 1 && n2 == 1 ) {
        if ( (UWORD)*term1 > (UWORD)*term2 ) return(1);
        else if ( (UWORD)*term1 < (UWORD)*term2 ) return(-1);
        else return(0);
    }

/*
    The next call should get dedicated code, as AddRat does more than
    strictly needed. Also more attention should be given to overflow.
*/
    c = NumberMalloc("CompCoef");
    if ( AddRat(BHEAD (UWORD *)term1,n1,(UWORD *)term2,-n2,c,&n3) ) {
        MLOCK(ErrorMessageLock);
        MesCall("CompCoef");
        MUNLOCK(ErrorMessageLock);
        NumberFree(c,"CompCoef");
        SETERROR(-1)
    }
    NumberFree(c,"CompCoef");
    return(n3);
}

/*
        #] CompCoef :
        #[ Modulus :        WORD Modulus(term)

    Routine takes the coefficient of term modulus b. The answer
    is in term again and the length of term is adjusted.

*/

WORD Modulus(WORD *term)
{
    WORD *t;
    WORD n1;
    t = term;
    GETCOEF(t,n1);
    if ( TakeModulus((UWORD *)t,&n1,AC.cmod,AC.ncmod,UNPACK) ) {
        MLOCK(ErrorMessageLock);
        MesCall("Modulus");
        MUNLOCK(ErrorMessageLock);
        SETERROR(-1)
    }
    if ( !n1 ) {
        *term = 0;
        return(0);
    }
    else if ( n1 > 0 ) {
        n1 «= 1;
        t += n1;        /* Note that n1 >= 0 */
        n1++;
    }
    else if ( n1 < 0 ) {
        n1 *= 2;
        t += -n1;
        n1--;
    }
    *t++ = n1;
    *term = WORDDIF(t,term);
    return(0);
}
```

```
/*
        #] Modulus :
        #[ TakeModulus :    WORD TakeModulus(a,na,cmodvec,ncmod,par)

        Routine gets the rational number in a with reduced length na.
        It is called when AC.ncmod != 0 and the number in AC.cmod is the
        number wrt which we need the modulus.
        The result is returned in a and na again.

        If par == NOUNPACK we only do a single number, not a fraction.
        In addition we don't do fancy. We want a positive number and
        the input was supposed to be positive.
        We don't pack the result. The calling routine is responsible for that.
        This may not be a good idea. To be checked.
*/

WORD TakeModulus(UWORD *a, WORD *na, UWORD *cmodvec, WORD ncmod, WORD par)
{
    GETIDENTITY
    UWORD *c, *d, *e, *f, *g, *h;
    UWORD *x4,*x2;
    UWORD *x3,*x1,*x5,*x6,*x7,*x8;
    WORD y3,y1,y5,y6;
    WORD n1, i, y2, y4;
    WORD nh, tdenom, tnumer, nmod;
    LONG x;
    if ( ncmod == 0 ) return(0);        /* No modulus operation */
    nmod = ABS(ncmod);
    n1 = *na;
    if ( ( par & UNPACK ) != 0 ) UnPack(a,n1,&tdenom,&tnumer);
    else { tnumer = n1; }
/*
    We fish out the special case that the coefficient is short as well.
    There is no need to make lots of calls etc
*/
    if ( ( ( par & UNPACK ) == 0 ) && nmod == 1 && ( n1 == 1 || n1 == -1 ) ) {
        goto simplecase;
    }
    else if ( nmod == 1 && ( n1 == 1 || n1 == -1 ) ) {
        if ( a[1] != 1 ) {
            a[1] = a[1] % cmodvec[0];
            if ( a[1] == 0 ) {
                MesPrint("Division by zero in short modulus arithmetic");
                return(-1);
            }
            y1 = 0;
            if ( ( AC.modinverses != 0 ) && ( ( par & NOINVERSES ) == 0 ) ) {
                y1 = AC.modinverses[a[1]];
            }
            else {
                GetModInverses(a[1],cmodvec[0],&y1,&y2);
            }
            x = a[0];
            a[0] = (x*y1) % cmodvec[0];
            a[1] = 1;
        }
        else {
simplecase:
            a[0] = a[0] % cmodvec[0];
        }
        if ( a[0] == 0 ) { *na = 0; return(0); }
        if ( ( AC.modmode & POSNEG ) != 0 ) {
            if ( a[0] > (UWORD)(cmodvec[0]/2) ) {
                a[0] =  cmodvec[0] - a[0];
                *na = -*na;
            }
        }
        else if ( *na < 0 ) {
            *na = 1; a[0] = cmodvec[0] - a[0];
        }
        return(0);
    }
    c = NumberMalloc("TakeModulus"); d = NumberMalloc("TakeModulus"); e = NumberMalloc("TakeModulus");
    f = NumberMalloc("TakeModulus"); g = NumberMalloc("TakeModulus"); h = NumberMalloc("TakeModulus");
    n1 = ABS(n1);
    if ( DivLong(a,tnumer,(UWORD *)cmodvec,nmod,
        c,&nh,a,&tnumer) ) goto ModErr;
    if ( tnumer == 0 ) { *na = 0; goto normalreturn; }
    if ( ( par & UNPACK ) == 0 ) {
        if ( ( AC.modmode & POSNEG ) != 0 ) {
            NormalModulus(a,&tnumer);
        }
        else if ( tnumer < 0 ) {
            SubPLon((UWORD *)cmodvec,nmod,a,-tnumer,a,&tnumer);
        }
        *na = tnumer;
        goto normalreturn;
```

```
    }
    if ( tdenom == 1 && a[n1] == 1 ) {
        if ( ( AC.modmode & POSNEG ) != 0 ) {
            NormalModulus(a,&tnumer);
        }
        else if ( tnumer < 0 ) {
            SubPLon((UWORD *)cmodvec,nmod,a,-tnumer,a,&tnumer);
        }
        *na = tnumer;
        i = ABS(tnumer);
        a += i;
        *a++ = 1;
        while ( --i > 0 ) *a++ = 0;
        goto normalreturn;
    }
    if ( DivLong(a+n1,tdenom,(UWORD *)cmodvec,nmod,c,&nh,a+n1,&tdenom) ) goto ModErr;
    if ( !tdenom ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Division by zero in modulus arithmetic");
        if ( AP.DebugFlag ) {
            AO.OutSkip = 3;
            FiniLine();
            i = *na;
            if ( i < 0 ) i = -i;
            while ( --i >= 0 ) { TalToLine((UWORD)(*a++)); TokenToLine((UBYTE *)"  "); }
            i = *na;
            if ( i < 0 ) i = -i;
            while ( --i >= 0 ) { TalToLine((UWORD)(*a++)); TokenToLine((UBYTE *)"  "); }
            TalToLine((UWORD)(*na));
            AO.OutSkip = 0;
            FiniLine();
        }
        MUNLOCK(ErrorMessageLock);
        NumberFree(c,"TakeModulus"); NumberFree(d,"TakeModulus"); NumberFree(e,"TakeModulus");
        NumberFree(f,"TakeModulus"); NumberFree(g,"TakeModulus"); NumberFree(h,"TakeModulus");
        return(-1);
    }
    if ( ( AC.modinverses != 0 ) && ( ( par & NOINVERSES ) == 0 )
    && ( tdenom == 1 || tdenom == -1 ) ) {
        *d = AC.modinverses[a[n1]]; y1 = 1; y2 = tdenom;
        if ( MulLong(a,tnumer,d,y1,c,&y3) ) goto ModErr;
        if ( DivLong(c,y3,(UWORD *)cmodvec,nmod,d,&y5,a,&tdenom) ) goto ModErr;
        if ( y2 < 0 ) tdenom = -tdenom;
    }
    else {
      x2 = (UWORD *)cmodvec; x1 = c; i = nmod; while ( --i >= 0 ) *x1++ = *x2++;
      x1 = c; x2 = a+n1; x3 = d; x4 = e; x5 = f; x6 = g;
      y1 = nmod; y2 = tdenom; y4 = 0; y5 = 1; *x5 = 1;
      for(;;) {
        if ( DivLong(x1,y1,x2,y2,h,&nh,x3,&y3) ) goto ModErr;
        if ( MulLong(x5,y5,h,nh,x6,&y6) ) goto ModErr;
        if ( AddLong(x4,y4,x6,-y6,x6,&y6) ) goto ModErr;
        if ( !y3 ) {
            if ( y2 != 1 || *x2 != 1 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Inverse in modulus arithmetic doesn't exist");
                MesPrint("Denominator and modulus are not relative prime");
                MUNLOCK(ErrorMessageLock);
                goto ModErr;
            }
            break;
        }
        x7 = x1; x1 = x2; y1 = y2; x2 = x3; y2 = y3; x3 = x7;
        x8 = x4; x4 = x5; y4 = y5; x5 = x6; y5 = y6; x6 = x8;
      }
      if ( y5 < 0 && AddLong((UWORD *)cmodvec,nmod,x5,y5,x5,&y5) ) goto ModErr;
      if ( MulLong(a,tnumer,x5,y5,c,&y3) ) goto ModErr;
      if ( DivLong(c,y3,(UWORD *)cmodvec,nmod,d,&y5,a,&tdenom) ) goto ModErr;
    }
    if ( !tdenom ) { *na = 0; goto normalreturn; }
    if ( ( ( AC.modmode & POSNEG ) != 0 ) && ( ( par & FROMFUNCTION ) == 0 ) ) {
        NormalModulus(a,&tdenom);
    }
    else if ( tdenom < 0 ) {
        SubPLon((UWORD *)cmodvec,nmod,a,-tdenom,a,&tdenom);
    }
    *na = tdenom;
    i = ABS(tdenom);
    a += i;
    *a++ = 1;
    while ( --i > 0 ) *a++ = 0;
normalreturn:
    NumberFree(c,"TakeModulus"); NumberFree(d,"TakeModulus"); NumberFree(e,"TakeModulus");
    NumberFree(f,"TakeModulus"); NumberFree(g,"TakeModulus"); NumberFree(h,"TakeModulus");
    return(0);
ModErr:
    MLOCK(ErrorMessageLock);
```

```
    MesCall("TakeModulus");
    MUNLOCK(ErrorMessageLock);
    NumberFree(c,"TakeModulus"); NumberFree(d,"TakeModulus"); NumberFree(e,"TakeModulus");
    NumberFree(f,"TakeModulus"); NumberFree(g,"TakeModulus"); NumberFree(h,"TakeModulus");
    SETERROR(-1)
}

/*
        #] TakeModulus :
        #[ TakeNormalModulus :  WORD TakeNormalModulus(a,na,par)

        added by Jan [01-09-2010]
*/

WORD TakeNormalModulus (UWORD *a, WORD *na, UWORD *c, WORD nc, WORD par)
{
    WORD n;
    WORD nhalfc;
    UWORD *halfc;

    GETIDENTITY;

    /* determine c/2 by right shifting */
    halfc = NumberMalloc("TakeNormalModulus");
    nhalfc=nc;
    WCOPY(halfc,c,nc);

    for (n=0; n<nhalfc; n++) {
        halfc[n] >>= 1;
        if (n+1<nc) halfc[n] |= c[n+1] << (BITSINWORD-1);
    }

    if (halfc[nhalfc-1]==0)
        nhalfc--;

    /* takes care of the number never expanding, e.g., -1(mod 100) -> 99 -> -1 */
    if (BigLong(a,ABS(*na),halfc,nhalfc) > 0) {

        TakeModulus(a,na,c,nc,par);

        n = ABS(*na);
        if (BigLong(a,n,halfc,nhalfc) > 0) {
            SubPLon(c,nc,a,n,a,&n);
            *na = (*na > 0 ? -n : n);
        }
    }

    NumberFree(halfc,"TakeNormalModulus");
    return(0);
}

/*
        #] TakeNormalModulus :
        #[ MakeModTable :   WORD MakeModTable()
*/

WORD MakeModTable()
{
    LONG size, i, j, n;
    n = ABS(AC.ncmod);
    if ( AC.modpowers ) {
        M_free(AC.modpowers,"AC.modpowers");
        AC.modpowers = NULL;
    }
    if ( n > 2 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("&No memory for modulus generator power table");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    if ( n == 0 ) return(0);
    size = (LONG)(*AC.cmod);
    if ( n == 2 ) size += (((LONG)AC.cmod[1])<<BITSINWORD);
    AC.modpowers = (UWORD *)Malloc1(size*n*sizeof(UWORD),"table for powers of modulus");
    if ( n == 1 ) {
        j = 1;
        for ( i = 0; i < size; i++ ) AC.modpowers[i] = 0;
        for ( i = 0; i < size; i++ ) {
            AC.modpowers[j] = (WORD)i;
            j *= *AC.powmod;
            j %= *AC.cmod;
        }
        for ( i = 2; i < size; i++ ) {
            if ( AC.modpowers[i] == 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("&improper generator for this modulus");
                MUNLOCK(ErrorMessageLock);
```

```
                M_free(AC.modpowers,"AC.modpowers");
                return(-1);
            }
        }
        AC.modpowers[1] = 0;
    }
    else {
        GETIDENTITY
        WORD nScrat, n2;
        UWORD *MMscrat7 = NumberMalloc("MakeModTable"), *MMscratC = NumberMalloc("MakeModTable");
        *MMscratC = 1;
        nScrat = 1;
        j = size « 1;
        for ( i = 0; i < j; i+=2 ) { AC.modpowers[i] = 0; AC.modpowers[i+1] = 0; }
        for ( i = 0; i < size; i++ ) {
            j = *MMscratC + (((LONG)MMscratC[1])«BITSINWORD);
            j «= 1;
            AC.modpowers[j] = (WORD)(i & WORDMASK);
            AC.modpowers[j+1] = (WORD)(i » BITSINWORD);
            MulLong((UWORD *)MMscratC,nScrat,(UWORD *)AC.powmod,
            AC.npowmod,(UWORD *)MMscrat7,&n2);
            TakeModulus(MMscrat7,&n2,AC.cmod,AC.ncmod,NOUNPACK);
            *MMscratC = *MMscrat7; MMscratC[1] = MMscrat7[1]; nScrat = n2;
        }
        NumberFree(MMscrat7,"MakeModTable"); NumberFree(MMscratC,"MakeModTable");
        j = size « 1;
        for ( i = 4; i < j; i+=2 ) {
            if ( AC.modpowers[i] == 0 && AC.modpowers[i+1] == 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("&improper generator for this modulus");
                MUNLOCK(ErrorMessageLock);
                M_free(AC.modpowers,"AC.modpowers");
                return(-1);
            }
        }
        AC.modpowers[2] = AC.modpowers[3] = 0;
    }
    return(0);
}

/*
        #] MakeModTable :
    #] RekenTerms :
    #[ Functions :
        #[ Factorial :      WORD Factorial(n,a,na)

    Starts with only the value of fac_(0).
    Builds up what is needed and remembers it for the next time.

    We have:
        AT.nfac: the number of the highest stored factorial
        AT.pfac: the array of locations in the array of stored factorials
        AT.factorials: the array with stored factorials
*/

int Factorial(PHEAD WORD n, UWORD *a, WORD *na)
{
    GETBIDENTITY
    UWORD *b, *c;
    WORD nc;
    int i, j;
    LONG ii;
    if ( n > AT.nfac ) {
        if ( AT.factorials == 0 ) {
            AT.nfac = 0; AT.mfac = 50; AT.sfact = 400;
            AT.pfac = (LONG *)Malloc1((AT.mfac+2)*sizeof(LONG),"factorials");
            AT.factorials = (UWORD *)Malloc1(AT.sfact*sizeof(UWORD),"factorials");
            AT.factorials[0] = 1; AT.pfac[0] = 0; AT.pfac[1] = 1;
        }
        b = a;
        c = AT.factorials+AT.pfac[AT.nfac];
        nc = i = AT.pfac[AT.nfac+1] - AT.pfac[AT.nfac];
        while ( --i >= 0 ) *b++ = *c++;
        for ( j = AT.nfac+1; j <= n; j++ ) {
            Product(a,&nc,j);
            if ( nc > AM.MaxTal ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Overflow in factorial. MaxTal = %d",AM.MaxTal);
                MesPrint("Increase MaxTerm in %s",setupfilename);
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
            if ( j > AT.mfac ) {  /* double the pfac buffer */
                LONG *p;
                p = (LONG *)Malloc1((AT.mfac*2+2)*sizeof(LONG),"factorials");
                i = AT.mfac;
                for ( i = AT.mfac+1; i >= 0; i-- ) p[i] = AT.pfac[i];
```

```
                M_free(AT.pfac,"factorial pointers"); AT.pfac = p; AT.mfac *= 2;
            }
            if ( AT.pfac[j] + nc >= AT.sfact ) { /* double the factorials buffer */
                UWORD *f;
                f = (UWORD *)Malloc1(AT.sfact*2*sizeof(UWORD),"factorials");
                ii = AT.sfact;
                c = AT.factorials; b = f;
                while ( --ii >= 0 ) *b++ = *c++;
                M_free(AT.factorials,"factorials");
                AT.factorials = f;
                AT.sfact *= 2;
            }
            b = a; c = AT.factorials + AT.pfac[j]; i = nc;
            while ( --i >= 0 ) *c++ = *b++;
            AT.pfac[j+1] = AT.pfac[j] + nc;
        }
        *na = nc;
        AT.nfac = n;
    }
    else if ( n == 0 ) {
        *a = 1; *na = 1;
    }
    else {
        *na = i = AT.pfac[n+1] - AT.pfac[n];
        b = AT.factorials + AT.pfac[n];
        while ( --i >= 0 ) *a++ = *b++;
    }
    return(0);
}

/*
        #] Factorial :
        #[ Bernoulli :      WORD Bernoulli(n,a,na)

    Starts with only the value of bernoulli_(0).
    Builds up what is needed and remembers it for the next time.
    b_0 = 1
    (n+1)*b_n = -b_{n-1}-sum_(i,1,n-1,b_i*b_{n-i})
    The n-1 playes only a role for b_2.
    We have hard coded b_0,b_1,b_2 and b_odd. After that:
    (2n+1)*b_2n = -sum_(i,1,n-1,b_2i*b_{2n-2i})

    We have:
        AT.nBer: the number of the highest stored Bernoulli number
        AT.pBer: the array of locations in the array of stored Bernoulli numbers
        AT.bernoullis: the array with stored Bernoulli numbers
*/

int Bernoulli(WORD n, UWORD *a, WORD *na)
{
    GETIDENTITY
    UWORD *b, *c, *scrib, *ntop, *ntop1;
    WORD i, i1, i2, nhalf, nqua, nscrib, nntop, nntop1, *oldworkpointer;
    UWORD twee = 2, twonplus1;
    int j;
    LONG ii;
    if ( n <= 1 ) {
        if ( n == 0 ) { a[0] = a[1] = 1; *na = 3; }
        else if ( n == 1 ) { a[0] = 1; a[1] = 2; *na = 3; }
        return(0);
    }
    if ( ( n & 1 ) != 0 ) { a[0] = a[1] = 0; *na = 0; return(0); }
    nhalf = n/2;
    if ( nhalf > AT.nBer ) {
        oldworkpointer = AT.WorkPointer;
        if ( AT.bernoullis == 0 ) {
            AT.nBer = 1; AT.mBer = 50; AT.sBer = 400;
            AT.pBer = (LONG *)Malloc1((AT.mBer+2)*sizeof(LONG),"bernoullis");
            AT.bernoullis = (UWORD *)Malloc1(AT.sBer*sizeof(UWORD),"bernoullis");
            AT.pBer[1] = 0; AT.pBer[2] = 3;
            AT.bernoullis[0] = 3; AT.bernoullis[1] = 1; AT.bernoullis[2] = 12;
            if ( nhalf == 1 ) {
                a[0] = 1; a[1] = 12; *na = 3; return(0);
            }
        }
        while ( nhalf > AT.mBer ) {
            LONG *p;
            p = (LONG *)Malloc1((AT.mBer*2+1)*sizeof(LONG),"bernoullis");
            i = AT.mBer;
            for ( i = AT.mBer; i >= 0; i-- ) p[i] = AT.pBer[i];
            M_free(AT.pBer,"factorial pointers"); AT.pBer = p; AT.mBer *= 2;
        }
        for ( n = AT.nBer+1; n <= nhalf; n++ ) {
            scrib = (UWORD *)(AT.WorkPointer);
            nqua = n/2;
            if ( ( n & 1 ) == 1 ) {
                nscrib = 0; ntop = scrib;
```

```
            }
            else {
                b = AT.bernoullis + AT.pBer[nqua];
                nscrib = *b++;
                i = (WORD)(REDLENG(nscrib));
                MulRat(BHEAD b,i,b,i,scrib,&nscrib);
                ntop = scrib + 2*nscrib;
                nqua--;
            }
            for ( j = 1; j <= nqua; j++ ) {
                b = AT.bernoullis + AT.pBer[j];
                c = AT.bernoullis + AT.pBer[n-j];
                i1 = (WORD)(*b); i2 = (WORD)(*c);
                i1 = REDLENG(i1);
                i2 = REDLENG(i2);
                MulRat(BHEAD b+1,i1,c+1,i2,ntop,&nntop);
                Mully(BHEAD ntop,&nntop,&twee,1);
                if ( nscrib ) {
                    i = (WORD)nntop; if ( i < 0 ) i = -i;
                    ntop1 = ntop + 2*i;
                    AddRat(BHEAD ntop,nntop,scrib,nscrib,ntop1,&nntop1);
                }
                else {
                    ntop1 = ntop; nntop1 = nntop;
                }
                nscrib = i1 = (WORD)nntop1;
                if ( i1 < 0 ) i1 = - i1;
                i1 = 2*i1;
                for ( i = 0; i < i1; i++ ) scrib[i] = ntop1[i];
                ntop = scrib + i1;
            }
            twonplus1 = 2*n+1;
            Divvy(BHEAD scrib,&nscrib,&twonplus1,-1);
            i1 = INCLENG(nscrib);
            i2 = i1; if ( i2 < 0 ) i2 = -i2;
            i = (WORD)(AT.bernoullis[AT.pBer[n-1]]);
            if ( i < 0 ) i = -i;
            AT.pBer[n] = AT.pBer[n-1]+i;
            if ( AT.pBer[n] + i2 >= AT.sBer ) {
                UWORD *f;
                f = (UWORD *)Malloc1(AT.sBer*2*sizeof(UWORD),"bernoullis");
                ii = AT.sBer;
                c = AT.bernoullis; b = f;
                while ( --ii >= 0 ) *b++ = *c++;
                M_free(AT.bernoullis,"bernoullis");
                AT.bernoullis = f;
                AT.sBer *= 2;
            }
            c = AT.bernoullis + AT.pBer[n]; b = scrib;
            *c++ = i1;
            for ( i = 1; i < i2; i++ ) *c++ = *b++;
        }
        AT.nBer = nhalf;
        AT.WorkPointer = oldworkpointer;
    }
    b = AT.bernoullis + AT.pBer[nhalf];
    *na = i = (WORD)(*b++);
    if ( i < 0 ) i = -i;
    i--;
    while ( --i >= 0 ) *a++ = *b++;
    return(0);
}

/*
        #] Bernoulli :
        #[ NextPrime :
*/
#if ( BITSINWORD == 32 )

void StartPrimeList(PHEAD0)
{
    int i, j;
    AR.PrimeList[AR.numinprimelist++] = 3;
    for ( i = 5; i < 46340; i += 2 ) {
        for ( j = 0; j < AR.numinprimelist && AR.PrimeList[j]*AR.PrimeList[j] <= i; j++ ) {
            if ( i % AR.PrimeList[j] == 0 ) goto nexti;
        }
        AR.PrimeList[AR.numinprimelist++] = i;
nexti:;
    }
    AR.notfirstprime = 1;
}

#endif

WORD NextPrime(PHEAD WORD num)
{
```

```c
    int i, j;
    WORD *newpl;
    LONG newsize, x;
#if ( BITSINWORD == 32 )
    if ( AR.notfirstprime == 0 ) StartPrimeList(BHEAD0);
#endif
    if ( num > AT.inprimelist ) {
        while ( AT.inprimelist < num ) {
            if ( num >= AT.sizeprimelist ) {
                if ( AT.sizeprimelist == 0 ) newsize = 32;
                else newsize = 2*AT.sizeprimelist;
                while ( num >= newsize ) newsize = newsize*2;
                newpl = (WORD *)Malloc1(newsize*sizeof(WORD),"NextPrime");
                for ( i = 0; i < AT.sizeprimelist; i++ ) {
                    newpl[i] = AT.primelist[i];
                }
                if ( AT.sizeprimelist > 0 ) {
                    M_free(AT.primelist,"NextPrime");
                }
                AT.sizeprimelist = newsize;
                AT.primelist = newpl;
            }
            if ( AT.inprimelist < 0 ) { i = MAXPOSITIVE; }
            else { i = AT.primelist[AT.inprimelist]; }
            while ( i > MAXPOWER ) {
                i -= 2; x = i;
#if ( BITSINWORD == 32 )
                for ( j = 0; j < AR.numinprimelist && AR.PrimeList[j]*(LONG)(AR.PrimeList[j]) <= x; j++ ) {
                    if ( x % AR.PrimeList[j] == 0 ) goto nexti;
                }
#else
                for ( j = 3; j*((LONG)j) <= x; j += 2 ) {
                    if ( x % j == 0 ) goto nexti;
                }
#endif
                AT.inprimelist++;
                AT.primelist[AT.inprimelist] = i;
                break;
nexti:;
            }
            if ( i < MAXPOWER ) {
                MLOCK(ErrorMessageLock);
                MesPrint("There are not enough short prime numbers for this calculation");
                MesPrint("Try to use a computer with a %d-bits architecture",
                    (int)(BITSINWORD*4));
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
        }
    }
    return(AT.primelist[num]);
}

/*
        #] NextPrime :
        #[ wranf :

        A random number generator that generates random WORDs with a very
        long sequence. It is based on the Knuth generator.

        We take some care that each thread can run its own, but each
        uses its own startup. Hence the seed includes the identity of
        the thread.

        For NPAIR1, NPAIR2 we can use any pair from the table on page 28.
        Candidates are 24,55 (the example on the pages 171,172)
        or (33,97) or (38,89)
        These values are defined in fsizes.h and used in startup.c and threads.c
*/

#define WARMUP 6

static void wranfnew(PHEAD0)
{
    int i;
    LONG j;
    for ( i = 0; i < AR.wranfnpair1; i++ ) {
        j = AR.wranfia[i] - AR.wranfia[i+(AR.wranfnpair2-AR.wranfnpair1)];
        if ( j < 0 ) j += (LONG)1 « (2*BITSINWORD-2);
        AR.wranfia[i] = j;
    }
    for ( i = AR.wranfnpair1; i < AR.wranfnpair2; i++ ) {
        j = AR.wranfia[i] - AR.wranfia[i-AR.wranfnpair1];
        if ( j < 0 ) j += (LONG)1 « (2*BITSINWORD-2);
        AR.wranfia[i] = j;
    }
```

```
}

void iniwranf(PHEAD0)
{
    int imax = AR.wranfnpair2-1;
    ULONG i, ii, seed = AR.wranfseed;
    LONG j, k;
    ULONG offset = 12345;
#ifdef PARALLELCODE
    int id;
#if defined(WITHPTHREADS)
    id = AT.identity;
#elif defined(WITHMPI)
    id = PF.me;
#endif
    seed += id;
    i = id + 1;
    if ( i > 1 ) {
        ULONG pow, accu;
        pow = offset; accu = 1;
        while ( i ) {
            if ( ( i & 1 ) != 0 ) accu *= pow;
            i »= 1; pow = pow*pow;
        }
        offset = accu;
    }
#endif
    if ( seed < ((LONG)1«(BITSINWORD-1)) ) {
        j = ( (seed+31459L) « (BITSINWORD-2))+offset;
    }
    else if ( seed < ((LONG)1«(BITSINWORD+10-1)) ) {
        j = ( (seed+31459L) « (BITSINWORD-10-2))+offset;
    }
    else {
        j = ( (seed+31459L) « 1)+offset;
    }
    if ( ( seed & 1 ) == 1 ) seed++;
    j += seed;
    AR.wranfia[imax] = j;
    k = 1;
    for ( i = 0; i <= (ULONG)(imax); i++ ) {
        ii = (AR.wranfnpair1*i)%AR.wranfnpair2;
        AR.wranfia[ii] = k;
        k = j - k;
        if ( k < 0 ) k += (LONG)1 « (2*BITSINWORD-2);
        j = AR.wranfia[ii];
    }
    for ( i = 0; i < WARMUP; i++ ) wranfnew(BHEAD0);
    AR.wranfcall = 0;
}

UWORD wranf(PHEAD0)
{
    UWORD wval;
    if ( AR.wranfia == 0 ) {
        AR.wranfia = (ULONG *)Malloc1(AR.wranfnpair2*sizeof(ULONG),"wranf");
        iniwranf(BHEAD0);
    }
    if ( AR.wranfcall >= AR.wranfnpair2) {
        wranfnew(BHEAD0);
        AR.wranfcall = 0;
    }
    wval = (UWORD)(AR.wranfia[AR.wranfcall++]»(BITSINWORD-1));
    return(wval);
}

/*
    Returns a random UWORD in the range (0,...,imax-1)
*/

UWORD iranf(PHEAD UWORD imax)
{
    UWORD i;
    ULONG x = (LONG)1 « BITSINWORD, xmax = x - x%imax;
    while ( ( i = wranf(BHEAD0) ) >= xmax ) {}
    return(i%imax);
}

/*
        #] wranf :
        #[ PreRandom :

        The random number generator of the preprocessor.
        This one is completely different from the execution time generator
        random_(number). In the preprocessor we generate a floating point
        number in a string according to a distribution.
        Currently allowed are:
```

```
            RANDOM_(log,min,max)
            RANDOM_(lin,min,max)
        The return value is a string with the floating point number.
*/

UBYTE *PreRandom(UBYTE *s)
{
    GETIDENTITY
    UBYTE *mode,*mins = 0,*maxs = 0, *outval;
    float num;
    double minval, maxval, value = 0;
    int linlog = -1;
    mode = s;
    while ( FG.cTable[*s] <= 1 ) s++;
    if ( *s == ',' ) { *s = 0; s++; }
    mins = s;
    while ( *s && *s != ',' ) s++;
    if ( *s == ',' ) { *s = 0; s++; }
    maxs = s;
    while ( *s && *s != ',' ) s++;
    if ( *s || *maxs == 0 || *mins == 0 ) {
        MesPrint("@Illegal arguments in macro RANDOM_");
        Terminate(-1);
    }
    if ( StrICmp(mode,(UBYTE *)"lin") == 0 ) {
        linlog = 0;
    }
    else if ( StrICmp(mode,(UBYTE *)"log") == 0 ) {
        linlog = 1;
    }
    else {
        MesPrint("@Illegal mode argument in macro RANDOM_");
        Terminate(-1);
    }

    sscanf((char *)mins,"%f",&num); minval = num;
    sscanf((char *)maxs,"%f",&num); maxval = num;

    /*
     * Note on ParFORM: we should use the same random number on all the
     * processes in the complication phase. The random number is generated
     * on the master and broadcast to the other processes.
     */
    {
        UWORD x;
        double xx;
#ifdef WITHMPI
        x = 0;
        if ( PF.me == MASTER ) {
            x = wranf(BHEAD0);
        }
        x = (UWORD)PF_BroadcastNumber((LONG)x);
#else
        x = wranf(BHEAD0);
#endif
        xx = x/pow(2.0,(double)(BITSINWORD-1));
        if ( linlog == 0 ) {
            value = minval + (maxval-minval)*xx;
        }
        else if ( linlog == 1 ) {
            value = minval * pow(maxval/minval,xx);
        }
    }

    outval = (UBYTE *)Malloc1(64,"PreRandom");
    if ( ABS(value) < 0.00001 || ABS(value) > 1000000. ) {
        sprintf((char *)outval,"%e",value);
    }
    else if ( ABS(value) < 0.0001 ) { sprintf((char *)outval,"%10f",value); }
    else if ( ABS(value) < 0.001 ) { sprintf((char *)outval,"%9f",value); }
    else if ( ABS(value) < 0.01 ) { sprintf((char *)outval,"%8f",value); }
    else if ( ABS(value) < 0.1 ) { sprintf((char *)outval,"%7f",value); }
    else if ( ABS(value) < 1. ) { sprintf((char *)outval,"%6f",value); }
    else if ( ABS(value) < 10. ) { sprintf((char *)outval,"%5f",value); }
    else if ( ABS(value) < 100. ) { sprintf((char *)outval,"%4f",value); }
    else if ( ABS(value) < 1000. ) { sprintf((char *)outval,"%3f",value); }
    else if ( ABS(value) < 10000. ) { sprintf((char *)outval,"%2f",value); }
    else { sprintf((char *)outval,"%1f",value); }
    return(outval);
}

/*
        #] PreRandom :
    #] Functions :
*/
```

4.92 reshuf.c File Reference

`#include "form3.h"`
Include dependency graph for reshuf.c:

Functions

- WORD ReNumber (PHEAD WORD ∗term)
- VOID FunLevel (PHEAD WORD ∗term)
- WORD DetCurDum (PHEAD WORD ∗t)
- int FullRenumber (PHEAD WORD ∗term, WORD par)
- VOID MoveDummies (PHEAD WORD ∗term, WORD shift)
- void AdjustRenumScratch (PHEAD0)
- WORD CountDo (WORD ∗term, WORD ∗instruct)
- WORD CountFun (WORD ∗term, WORD ∗countfun)
- WORD DimensionSubterm (WORD ∗subterm)
- WORD DimensionTerm (WORD ∗term)
- WORD DimensionExpression (PHEAD WORD ∗expr)
- WORD MultDo (PHEAD WORD ∗term, WORD ∗pattern)
- WORD TryDo (PHEAD WORD ∗term, WORD ∗pattern, WORD level)
- WORD DoDistrib (PHEAD WORD ∗term, WORD level)
- WORD EqualArg (WORD ∗parms, WORD num1, WORD num2)
- WORD DoDelta3 (PHEAD WORD ∗term, WORD level)
- WORD TestPartitions (PHEAD WORD ∗tfun, PARTI ∗parti)
- WORD DoPartitions (PHEAD WORD ∗term, WORD level)
- WORD DoPermutations (PHEAD WORD ∗term, WORD level)
- WORD DoShuffle (PHEAD WORD ∗term, WORD level, WORD fun, WORD option)
- int Shuffle (PHEAD WORD ∗from1, WORD ∗from2, WORD ∗to)
- int FinishShuffle (PHEAD WORD ∗fini)
- WORD DoStuffle (PHEAD WORD ∗term, WORD level, WORD fun, WORD option)
- int Stuffle (PHEAD WORD ∗from1, WORD ∗from2, WORD ∗to)
- int FinishStuffle (PHEAD WORD ∗fini)
- WORD ∗ StuffRootAdd (WORD ∗t1, WORD ∗t2, WORD ∗to)

4.92.1 Detailed Description

Mixed routines: Routines for relabelling dummy indices. The multiply command The distrib_ function The tryreplace statement

Definition in file reshuf.c.

4.92.2 Macro Definition Documentation

4.92.2.1 NEWCODE

```
#define NEWCODE
```

Definition at line 35 of file reshuf.c.

4.92.3 Function Documentation

4.92.3.1 ReNumber()

```
WORD ReNumber (
            PHEAD WORD * term )
```

Definition at line 80 of file reshuf.c.

4.92.3.2 FunLevel()

```
VOID FunLevel (
            PHEAD WORD * term )
```

Definition at line 130 of file reshuf.c.

4.92.3.3 DetCurDum()

```
WORD DetCurDum (
            PHEAD WORD * t )
```

Definition at line 251 of file reshuf.c.

4.92.3.4 FullRenumber()

```
int FullRenumber (
            PHEAD WORD * term,
            WORD par )
```

Definition at line 323 of file reshuf.c.

4.92.3.5 MoveDummies()

```
VOID MoveDummies (
            PHEAD WORD * term,
            WORD shift )
```

Definition at line 435 of file reshuf.c.

4.92.3.6 AdjustRenumScratch()

```
void AdjustRenumScratch (
            PHEAD0  )
```

Definition at line 505 of file reshuf.c.

4.92.3.7 CountDo()

```
WORD CountDo (
            WORD * term,
            WORD * instruct )
```

Definition at line 551 of file reshuf.c.

4.92.3.8 CountFun()

```
WORD CountFun (
            WORD * term,
            WORD * countfun )
```

Definition at line 705 of file reshuf.c.

4.92.3.9 DimensionSubterm()

```
WORD DimensionSubterm (
            WORD * subterm )
```

Definition at line 866 of file reshuf.c.

4.92.3.10 DimensionTerm()

```
WORD DimensionTerm (
            WORD * term )
```

Definition at line 967 of file reshuf.c.

4.92.3.11 DimensionExpression()

```
WORD DimensionExpression (
            PHEAD WORD * expr )
```

Definition at line 999 of file reshuf.c.

4.92.3.12 MultDo()

```
WORD MultDo (
            PHEAD WORD * term,
            WORD * pattern )
```

Definition at line 1043 of file reshuf.c.

4.92.3.13 TryDo()

```
WORD TryDo (
            PHEAD WORD * term,
            WORD * pattern,
            WORD level )
```

Definition at line 1071 of file reshuf.c.

4.92.3.14 DoDistrib()

```
WORD DoDistrib (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1118 of file reshuf.c.

4.92.3.15 EqualArg()

```
WORD EqualArg (
            WORD * parms,
            WORD num1,
            WORD num2 )
```

Definition at line 1378 of file reshuf.c.

4.92.3.16 DoDelta3()

```
WORD DoDelta3 (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1404 of file reshuf.c.

4.92.3.17 TestPartitions()

```
WORD TestPartitions (
            PHEAD WORD * tfun,
            PARTI * parti )
```

Definition at line 1606 of file reshuf.c.

4.92.3.18 DoPartitions()

```
WORD DoPartitions (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 1721 of file reshuf.c.

4.92.3.19 DoPermutations()

```
WORD DoPermutations (
            PHEAD WORD * term,
            WORD level )
```

Definition at line 2006 of file reshuf.c.

4.92.3.20 DoShuffle()

```
WORD DoShuffle (
            PHEAD WORD * term,
            WORD level,
            WORD fun,
            WORD option )
```

Definition at line 2094 of file reshuf.c.

4.92.3.21 Shuffle()

```
int Shuffle (
            PHEAD WORD * from1,
            WORD * from2,
            WORD * to )
```

Definition at line 2228 of file reshuf.c.

4.92.3.22 FinishShuffle()

```
int FinishShuffle (
            PHEAD WORD * fini )
```

Definition at line 2419 of file reshuf.c.

4.92.3.23 DoStuffle()

```
WORD DoStuffle (
            PHEAD WORD * term,
            WORD level,
            WORD fun,
            WORD option )
```

Definition at line 2484 of file reshuf.c.

4.92.3.24 Stuffle()

```
int Stuffle (
            PHEAD WORD * from1,
            WORD * from2,
            WORD * to )
```

Definition at line 2699 of file reshuf.c.

4.92.3.25 FinishStuffle()

```
int FinishStuffle (
            PHEAD WORD * fini )
```

Definition at line 2826 of file reshuf.c.

4.92.3.26 StuffRootAdd()

```
WORD * StuffRootAdd (
            WORD * t1,
            WORD * t2,
            WORD * to )
```

Definition at line 2873 of file reshuf.c.

4.93 reshuf.c

Go to the documentation of this file.

```
00001
00009 /* #[ License : */
00010 /*
00011  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00012  *   When using this file you are requested to refer to the publication
00013  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014  *   This is considered a matter of courtesy as the development was paid
00015  *   for by FOM the Dutch physics granting agency and we would like to
00016  *   be able to track its scientific use to convince FOM of its value
00017  *   for the community.
00018  *
00019  *   This file is part of FORM.
00020  *
00021  *   FORM is free software: you can redistribute it and/or modify it under the
00022  *   terms of the GNU General Public License as published by the Free Software
00023  *   Foundation, either version 3 of the License, or (at your option) any later
00024  *   version.
00025  *
00026  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00029  *   details.
00030  *
00031  *   You should have received a copy of the GNU General Public License along
00032  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00033  */
00034 /* #] License : */
00035 #define NEWCODE
00036 /*
00037     #[ Includes : reshuf.c
00038 */
00039
00040 #include "form3.h"
00041
00042 /*
00043     #] Includes :
```

```
    #[ Reshuf :

    Routines to rearrange dummy indices, so that
    a:  The notation becomes reasonably unique (the perfect job
        may consume very much time).
    b:  The value of AR.CurDum is reset.

    Also some routines are needed to aid in the reading of stored
    expressions. Also in those expressions there can be dummy
    indices, and there should be no conflict with the already
    existing dummies.

        #[ ReNumber :

        Reads the term, tests for dummies, and puts them in order.
        Note that this is kind of a first order approximation.
        There is quite some room to make this routine 'smart'
        First order:
            First index will be lowest, second will be next etc.
        Second order:
            Functions with more than one index and symmetry properties
            have some look ahead to see which index is the first to
            find its partner.
        Third order:
            Take the ordering of the functions into account.
        Fourth order:
            Try all permutations and see which one gives the 'minimal' term.
        Currently we use only the first order.

        We need a scratch array for the numbers we find, and one for
        the addresses at which these numbers are.
        We can use the space for the Scrat arrays. There are 13 of those
        and each is AM.MaxTal UWORDs long.

*/

WORD ReNumber(PHEAD WORD *term)
{
    GETBIDENTITY
    WORD *d, *e, **p, **f;
    WORD n, i, j, old;
    AN.DumFound = AN.RenumScratch;
    AN.DumPlace = AN.PoinScratch;
    AN.DumFunPlace = AN.FunScratch;
    AN.NumFound = 0;
    FunLevel(BHEAD term);
    d = AN.RenumScratch;
    p = AN.PoinScratch;
    f = AN.FunScratch;
/*
    Now the first level sorting.
*/
    i = AN.IndDum;
    n = AN.NumFound;
    while ( --n >= 0 ) {
        if ( *d > 0 ) {
            old = **p;
            **p = ++i;
            if ( *f ) **f |= DIRTYSYMFLAG;
            e = d;
            e++;
            for ( j = 1; j <= n; j++ ) {
                if ( *e && *(p[j]) == old ) {
                    *(p[j]) = i;
                    if ( f[j] ) *(f[j]) |= DIRTYSYMFLAG;
                    *e = 0;
                }
                e++;
            }
        }
        p++;
        d++;
        f++;
    }
    return(i);
}

/*
        #] ReNumber :
        #[ FunLevel :

        Does one term in determining where the dummies are.
        Made to work recursively for functions.

*/

VOID FunLevel(PHEAD WORD *term)
```

```
{
    GETBIDENTITY
    WORD *t, *tstop, *r, *fun;
    WORD *m;
    t = r = term;
    r += *r;
    tstop = r - ABS(r[-1]);
    t++;
    if ( t < tstop ) do {
        r = t + t[1];
        switch ( *t ) {
            case SYMBOL:
            case DOTPRODUCT:
                break;
            case VECTOR:
                t += 3;
                do {
                    if ( *t > AN.IndDum ) {
                        if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
                        AN.NumFound++;
                        *AN.DumFound++ = *t;
                        *AN.DumPlace++ = t;
                        *AN.DumFunPlace++ = 0;
                    }
                    t += 2;
                } while ( t < r );
                break;
            case SUBEXPRESSION:
/*
        Still must hunt down the wildcards(?)
*/
                break;
            case GAMMA:
                t += FUNHEAD-2;
            case DELTA:
            case INDEX:
                t += 2;
                while ( t < r ) {
                    if ( *t > AN.IndDum ) {
                        if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
                        AN.NumFound++;
                        *AN.DumFound++ = *t;
                        *AN.DumPlace++ = t;
                        *AN.DumFunPlace++ = 0;
                    }
                    t++;
                }
                break;
            case HAAKJE:
            case EXPRESSION:
            case SNUMBER:
            case LNUMBER:
                break;
            default:
                if ( *t < FUNCTION ) {
                  MLOCK(ErrorMessageLock);
                  MesPrint("Unexpected code in ReNumber");
                  MUNLOCK(ErrorMessageLock);
                  Terminate(-1);
                }
                fun = t+2;
                if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
                >= TENSORFUNCTION ) {
                    t += FUNHEAD;
                    while ( t < r ) {
                        if ( *t > AN.IndDum ) {
                            if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
                            AN.NumFound++;
                            *AN.DumFound++ = *t;
                            *AN.DumPlace++ = t;
                            *AN.DumFunPlace++ = fun;
                        }
                        t++;
                    }
                    break;
                }

                t += FUNHEAD;
                while ( t < r ) {
                    if ( *t > 0 ) {

                        /* General function. Enter 'recursion'. */

                        m = t + *t;
                        t += ARGHEAD;
                        while ( t < m ) {
                            FunLevel(BHEAD t);
```

```
                            t += *t;
                        }
                    }
                    else {
                        if ( *t == -INDEX ) {
                            t++;
                            if ( *t >= AN.IndDum ) {
                                if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
                                AN.NumFound++;
                                *AN.DumFound++ = *t;
                                *AN.DumPlace++ = t;
                                *AN.DumFunPlace++ = fun;
                            }
                            t++;
                        }
                        else if ( *t <= -FUNCTION ) t++;
                        else t += 2;
                    }
                }
                break;
        }
        t = r;
    } while ( t < tstop );
}

/*
        #] FunLevel :
        #[ DetCurDum :

        We look for indices in the range AM.IndDum to AM.IndDum+MAXDUMMIES.
        The maximum value is returned.
*/

WORD DetCurDum(PHEAD WORD *t)
{
    GETBIDENTITY
    WORD maxval = AN.IndDum;
    WORD maxtop = AM.IndDum + WILDOFFSET;
    WORD *tstop, *m, *r, i;
    tstop = t + *t - 1;
    tstop -= ABS(*tstop);
    t++;
    while ( t < tstop ) {
        if ( *t == VECTOR ) {
            m = t + 3;
            t += t[1];
            while ( m < t ) {
                if ( *m > maxval && *m < maxtop ) maxval = *m;
                m += 2;
            }
        }
        else if ( *t == DELTA || *t == INDEX ) {
            m = t + 2;
Singles:
            t += t[1];
            while ( m < t ) {
                if ( *m > maxval && *m < maxtop ) maxval = *m;
                m++;
            }
        }
        else if ( *t >= FUNCTION ) {
            if ( functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
                m = t + FUNHEAD;
                goto Singles;
            }
            r = t + FUNHEAD;
            t += t[1];
            while ( r < t ) {       /* The arguments */
                if ( *r < 0 ) {
                    if ( *r <= -FUNCTION ) r++;
                    else if ( *r == -INDEX ) {
                        if ( r[1] > maxval && r[1] < maxtop ) maxval = r[1];
                        r += 2;
                    }
                    else r += 2;
                }
                else {
                    m = r + ARGHEAD;
                    r += *r;
                    while ( m < r ) {   /* Terms in the argument */
                        i = DetCurDum(BHEAD m);
                        if ( i > maxval && i < maxtop ) maxval = i;
                        m += *m;
                    }
                }
            }
        }
```

```
        else {
            t += t[1];
        }
    }
    return(maxval);
}

/*
        #] DetCurDum :
        #[ FullRenumber :

        Does a full renumbering. May be slow if there are many indices.
        par = 1 Goes with a factorial!
        par = 0 All single exchanges only till there is no more improvement.
        Notice that there is a hole in the defense with respect to
        arguments inside functions inside functions.
*/

int FullRenumber(PHEAD WORD *term, WORD par)
{
    GETBIDENTITY
    WORD *d, **p, **f, *w, *t, *best, *stac, *perm, a, *termtry;
    WORD n, i, j, k, ii;
    WORD *oldworkpointer = AT.WorkPointer;
    n = ReNumber(BHEAD term) - AM.IndDum;
    if ( n <= 1 ) return(0);
    Normalize(BHEAD term);
    if ( *term == 0 ) return(0);
    n = ReNumber(BHEAD term) - AM.IndDum;
    d = AN.RenumScratch;
    p = AN.PoinScratch;
    f = AN.FunScratch;
    if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
    k = AN.NumFound;
    best = w = AT.WorkPointer; t = term;
    for ( i = *term; i > 0; i-- ) *w++ = *t++;
    AT.WorkPointer = w;
    Normalize(BHEAD best);
    AT.WorkPointer = w = best + *best;
    stac = w+100;
    perm = stac + n + 1;
    termtry = perm + n + 1;
    for ( i = 1; i <= n; i++ ) perm[i] = i + AM.IndDum;
    for ( i = 1; i <= n; i++ ) stac[i] = i;
    for ( i = 0; i < k; i++ ) d[i] = *(p[i]) - AM.IndDum;
    if ( par == 0 ) {   /* All single exchanges */
        for ( i = 1; i < n; i++ ) {
            for ( j = i+1; j <= n; j++ ) {
                a = perm[j]; perm[j] = perm[i]; perm[i] = a;
                for ( ii = 0; ii < k; ii++ ) {
                    *(p[ii]) = perm[d[ii]];
                    if ( f[ii] ) *(f[ii]) |= DIRTYSYMFLAG;
                }
                t = term; w = termtry;
                for ( ii = 0; ii < *term; ii++ ) *w++ = *t++;
                AT.WorkPointer = w;
                if ( Normalize(BHEAD termtry) == 0 ) {
                    if ( *termtry == 0 ) goto Return0;
                    if ( ( ii = CompareTerms(BHEAD termtry,best,0) ) > 0 ) {
                        t = termtry; w = best;
                        for ( ii = 0; ii < *termtry; ii++ ) *w++ = *t++;
                        i = 0; break; /* restart from beginning */
                    }
                    else if ( ii == 0 && CompCoef(termtry,best) != 0 )
                        goto Return0;
                }
                /* if no success, set back */
                a = perm[j]; perm[j] = perm[i]; perm[i] = a;
            }
        }
    }
    else if ( par == 1 ) {  /* all permutations */
        j = n-1;
        for(;;) {
            if ( stac[j] == n ) {
                a = perm[j]; perm[j] = perm[n]; perm[n] = a;
                stac[j] = j;
                j--;
                if ( j <= 0 ) break;
                continue;
            }
            if ( j != stac[j] ) {
                a = perm[j]; perm[j] = perm[stac[j]]; perm[stac[j]] = a;
            }
            (stac[j])++;
            a = perm[j]; perm[j] = perm[stac[j]]; perm[stac[j]] = a;
            {
```

```
                for ( i = 0; i < k; i++ ) {
                    *(p[i]) = perm[d[i]];
                    if ( f[i] ) *(f[i]) |= DIRTYSYMFLAG;
                }
                t = term; w = termtry;
                for ( i = 0; i < *term; i++ ) *w++ = *t++;
                AT.WorkPointer = w;
                if ( Normalize(BHEAD termtry) == 0 ) {
                    if ( *termtry == 0 ) goto Return0;
                    if ( ( ii = CompareTerms(BHEAD termtry,best,0) ) > 0 ) {
                        t = termtry; w = best;
                        for ( i = 0; i < *termtry; i++ ) *w++ = *t++;
                    }
                    else if ( ii == 0 && CompCoef(termtry,best) != 0 )
                        goto Return0;
                }
            }
            if ( j < n-1 ) { j = n-1; }
        }
    }
    t = term; w = best;
    n = *best;
    for ( i = 0; i < n; i++ ) *t++ = *w++;
    AT.WorkPointer = oldworkpointer;
    return(0);
Return0:
    *term = 0;
    AT.WorkPointer = oldworkpointer;
    return(0);
}

/*
        #] FullRenumber :
        #[ MoveDummies :

        Routine shifts the dummy indices by an amount 'shift'.
        It is an adaptation of DetCurDum.
        Needed for  = ...*expression^power*...
        in which expression contains dummy indices.
        Note that this code should have been in ver1 already and has
        always been missing. Routine made 29-jan-2007.
*/

VOID MoveDummies(PHEAD WORD *term, WORD shift)
{
    GETBIDENTITY
    WORD maxval = AN.IndDum;
    WORD maxtop = AM.IndDum + WILDOFFSET;
    WORD *tstop, *m, *r;
    tstop = term + *term - 1;
    tstop -= ABS(*tstop);
    term++;
    while ( term < tstop ) {
        if ( *term == VECTOR ) {
            m = term + 3;
            term += term[1];
            while ( m < term ) {
                if ( *m > maxval && *m < maxtop ) *m += shift;
                m += 2;
            }
        }
        else if ( *term == DELTA || *term == INDEX ) {
            m = term + 2;
Singles:
            term += term[1];
            while ( m < term ) {
                if ( *m > maxval && *m < maxtop ) *m += shift;
                m++;
            }
        }
        else if ( *term >= FUNCTION ) {
            if ( functions[*term-FUNCTION].spec >= TENSORFUNCTION ) {
                m = term + FUNHEAD;
                goto Singles;
            }
            r = term + FUNHEAD;
            term += term[1];
            while ( r < term ) {        /* The arguments */
                if ( *r < 0 ) {
                    if ( *r <= -FUNCTION ) r++;
                    else if ( *r == -INDEX ) {
                        if ( r[1] > maxval && r[1] < maxtop ) r[1] += shift;
                        r += 2;
                    }
                    else r += 2;
                }
                else {
```

```
                    m = r + ARGHEAD;
                    r += *r;
                    while ( m < r ) {   /* Terms in the argument */
                        MoveDummies(BHEAD m,shift);
                        m += *m;
                    }
                }
            }
        }
        else {
            term += term[1];
        }
    }
}

/*
        #] MoveDummies :
        #[ AdjustRenumScratch :

        Extends the buffer for number of dummies that can be found in
        a term. Originally we had a fixed buffer at size 300 in the AN
        struct. Thomas Hahn ran out of that. Hence we have now made it
        a dynamical buffer.
        Note that the pointers used in FunLevel need adjustment as well.
*/

void AdjustRenumScratch(PHEAD0)
{
    GETBIDENTITY
    WORD newsize;
    int i;
    WORD **newpoin, *newnum;
    if ( AN.MaxRenumScratch == 0 ) newsize = 100;
    else newsize = AN.MaxRenumScratch*2;
    if ( newsize > MAXPOSITIVE/2 ) newsize = MAXPOSITIVE/2+1;

    newpoin = (WORD **)Malloc1(newsize*sizeof(WORD *),"PoinScratch");
    for ( i = 0; i < AN.NumFound; i++ ) newpoin[i] = AN.PoinScratch[i];
    for ( ; i < newsize; i++ ) newpoin[i] = 0;
    if ( AN.PoinScratch ) M_free(AN.PoinScratch,"PoinScratch");
    AN.PoinScratch = newpoin;
    AN.DumPlace = newpoin + AN.NumFound;

    newpoin = (WORD **)Malloc1(newsize*sizeof(WORD *),"FunScratch");
    for ( i = 0; i < AN.NumFound; i++ ) newpoin[i] = AN.FunScratch[i];
    for ( ; i < newsize; i++ ) newpoin[i] = 0;
    if ( AN.FunScratch ) M_free(AN.FunScratch,"FunScratch");
    AN.FunScratch = newpoin;
    AN.DumFunPlace = newpoin + AN.NumFound;

    newnum = (WORD *)Malloc1(newsize*sizeof(WORD),"RenumScratch");
    for ( i = 0; i < AN.NumFound; i++ ) newnum[i] = AN.RenumScratch[i];
    for ( ; i < newsize; i++ ) newnum[i] = 0;
    if ( AN.RenumScratch ) M_free(AN.RenumScratch,"RenumScratch");
    AN.RenumScratch = newnum;
    AN.DumFound = newnum + AN.NumFound;

    AN.MaxRenumScratch = newsize;
}

/*
        #] AdjustRenumScratch :
    #] Reshuf :
    #[ Count :
        #[ CountDo :

        This function executes the counting action in a count
        operation. The return value is the count of the term.
        Input is the term and a pointer to the instruction.

*/

WORD CountDo(WORD *term, WORD *instruct)
{
    WORD *m, *r, i, j, count = 0;
    WORD *stopper, *tstop, *r1 = 0, *r2 = 0;
    m = instruct;
    stopper = m + m[1];
    instruct += 3;
    tstop = term + *term; tstop -= ABS(tstop[-1]); term++;
    while ( term < tstop ) {
        switch ( *term ) {
            case SYMBOL:
                i = term[1] - 2;
                term += 2;
                while ( i > 0 ) {
                    m = instruct;
```

```
                    while ( m < stopper ) {
                        if ( *m == SYMBOL && m[2] == *term ) {
                            count += m[3] * term[1];
                        }
                        m += m[1];
                    }
                    term += 2;
                    i -= 2;
                }
                break;
            case DOTPRODUCT:
                i = term[1] - 2;
                term += 2;
                while ( i > 0 ) {
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == DOTPRODUCT && (( m[2] == *term &&
                        m[3] == term[1]) || ( m[2] == term[1] &&
                        m[3] == *term )) ) {
                            count += m[4] * term[2];
                            break;
                        }
                        m += m[1];
                    }
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == VECTOR && m[2] == *term &&
                        ( m[3] & DOTPBIT ) != 0 ) {
                            count += m[m[1]-1] * term[2];
                        }
                        m += m[1];
                    }
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == VECTOR && m[2] == term[1] &&
                        ( m[3] & DOTPBIT ) != 0 ) {
                            count += m[m[1]-1] * term[2];
                        }
                        m += m[1];
                    }
                    term += 3;
                    i -= 3;
                }
                break;
            case INDEX:
                j = 1;
                goto VectInd;
            case VECTOR:
                j = 2;
VectInd:        i = term[1] - 2;
                term += 2;
                while ( i > 0 ) {
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == VECTOR && m[2] == *term &&
                        ( m[3] & VECTBIT ) != 0 ) {
                            count += m[m[1]-1];
                        }
                        m += m[1];
                    }
                    term += j;
                    i -= j;
                }
                break;
            default:
                if ( *term >= FUNCTION ) {
                    i = *term;
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == FUNCTION && m[2] == i ) count += m[3];
                        m += m[1];
                    }
                    if ( functions[i-FUNCTION].spec >= TENSORFUNCTION ) {
                        i = term[1] - FUNHEAD;
                        term += FUNHEAD;
                        while ( i > 0 ) {
                            if ( *term < 0 ) {
                                m = instruct;
                                while ( m < stopper ) {
                                    if ( *m == VECTOR && m[2] == *term &&
                                    ( m[3] & FUNBIT ) != 0 ) {
                                        count += m[m[1]-1];
                                    }
                                    m += m[1];
                                }
                            }
                            term++;
```

```
                            i--;
                        }
                    }
                    else {
                        r = term + term[1];
                        term += FUNHEAD;
                        while ( term < r ) {
                            if ( ( *term == -INDEX || *term == -VECTOR
                            || *term == -MINVECTOR ) && term[1] < MINSPEC ) {
                                m = instruct;
                                while ( m < stopper ) {
                                    if ( *m == VECTOR && term[1] == m[2]
                                    && ( m[3] & SETBIT ) != 0 ) {
                                        r1 = SetElements + Sets[m[4]].first;
                                        r2 = SetElements + Sets[m[4]].last;
                                        while ( r1 < r2 ) {
                                            if ( *r1 == i ) {
                                                count += m[m[1]-1];
                                                goto NextFF;
                                            }
                                            r1++;
                                        }
                                    }
                                    m += m[1];
                                }
NextFF:
                                term += 2;
                            }
                            else { NEXTARG(term) }
                        }
                    }
                    break;
                }
                else {
                    term += term[1];
                }
                break;
        }
    }
    return(count);
}

/*
        #] CountDo :
        #[ CountFun :

        This is the count function.
        The return value is the count of the term.
        Input is the term and a pointer to the count function.

*/

WORD CountFun(WORD *term, WORD *countfun)
{
    WORD *m, *r, i, j, count = 0, *instruct, *stopper, *tstop;
    m = countfun;
    stopper = m + m[1];
    instruct = countfun + FUNHEAD;
    tstop = term + *term; tstop -= ABS(tstop[-1]); term++;
    while ( term < tstop ) {
        switch ( *term ) {
            case SYMBOL:
                i = term[1] - 2;
                term += 2;
                while ( i > 0 ) {
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
                        if ( *m == -SYMBOL && m[1] == *term
                        && m[2] == -SNUMBER && ( m + 2 ) < stopper ) {
                            count += m[3] * term[1]; m += 4;
                        }
                        else { NEXTARG(m) }
                    }
                    term += 2;
                    i -= 2;
                }
                break;
            case DOTPRODUCT:
                i = term[1] - 2;
                term += 2;
                while ( i > 0 ) {
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
                        if ( *m == 9+ARGHEAD && m[ARGHEAD] == 9
                        && m[ARGHEAD+1] == DOTPRODUCT
```

```
                        && m[ARGHEAD+9] == -SNUMBER && ( m + ARGHEAD+9 ) < stopper
                        && (( m[ARGHEAD+3] == *term &&
                        m[ARGHEAD+4] == term[1]) ||
                         ( m[ARGHEAD+3] == term[1] &&
                        m[ARGHEAD+4] == *term )) ) {
                            count += m[ARGHEAD+10] * term[2];
                            m += ARGHEAD+11;
                        }
                        else { NEXTARG(m) }
                    }
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
                        if ( ( *m == -VECTOR || *m == -MINVECTOR )
                        && m[1] == *term &&
                        m[2] == -SNUMBER && ( m+2 ) < stopper ) {
                            count += m[3] * term[2]; m += 4;
                        }
                        NEXTARG(m)
                    }
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
                        if ( ( *m == -VECTOR || *m == -MINVECTOR )
                        && m[1] == term[1] &&
                        m[2] == -SNUMBER && ( m+2 ) < stopper ) {
                            count += m[3] * term[2];
                            m += 4;
                        }
                        NEXTARG(m)
                    }
                    term += 3;
                    i -= 3;
                }
                break;
            case INDEX:
                j = 1;
                goto VectInd;
            case VECTOR:
                j = 2;
VectInd:        i = term[1] - 2;
                term += 2;
                while ( i > 0 ) {
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
                        if ( ( *m == -VECTOR || *m == -MINVECTOR )
                        && m[1] == *term &&
                        m[2] == -SNUMBER && (m+2) < stopper ) {
                            count += m[3]; m += 4;
                        }
                        NEXTARG(m)
                    }
                    term += j;
                    i -= j;
                }
                break;
            default:
                if ( *term >= FUNCTION ) {
                    i = *term;
                    m = instruct;
                    while ( m < stopper ) {
                        if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
                        if ( *m == -i && m[1] == -SNUMBER && (m+1) < stopper ) {
                            count += m[2]; m += 3;
                        }
                        NEXTARG(m)
                    }
                    if ( functions[i-FUNCTION].spec >= TENSORFUNCTION ) {
                        i = term[1] - FUNHEAD;
                        term += FUNHEAD;
                        while ( i > 0 ) {
                            if ( *term < 0 ) {
                                m = instruct;
                                while ( m < stopper ) {
                                    if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
                                    if ( ( *m == -VECTOR || *m == -INDEX
                                    || *m == -MINVECTOR ) && m[1] == *term &&
                                    m[2] == -SNUMBER && (m+2) < stopper ) {
                                        count += m[3]; m += 4;
                                    }
                                    else { NEXTARG(m) }
                                }
                            }
                            term++;
                            i--;
                        }
```

```
                    }
                    else {
                        r = term + term[1];
                        term += FUNHEAD;
                        while ( term < r ) {
                            if ( ( *term == -INDEX || *term == -VECTOR
                            || *term == -MINVECTOR ) && term[1] < MINSPEC ) {
                                m = instruct;
                                while ( m < stopper ) {
                                    if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
                                    if ( *m == -VECTOR && m[1] == term[1]
                                    && m[2] == -SNUMBER && (m+2) < stopper ) {
                                        count += m[3];
                                        m += 4;
                                    }
                                    else { NEXTARG(m) }
                                }
                                term += 2;
                            }
                            else { NEXTARG(term) }
                        }
                    }
                    break;
                }
                else {
                    term += term[1];
                }
                break;
        }
    }
    return(count);
}

/*
        #] CountFun :
    #] Count :
    #[ DimensionSubterm :
*/

WORD DimensionSubterm(WORD *subterm)
{
    WORD *r, *rstop, dim, i;
    LONG x = 0;
    rstop = subterm + subterm[1];
    if ( *subterm == SYMBOL ) {
        r = subterm + 2;
        while ( r < rstop ) {
            if ( *r <= NumSymbols && *r > -MAXPOWER ) {
                dim = symbols[*r].dimension;
                if ( dim == MAXPOSITIVE ) goto undefined;
                x += dim * r[1];
                if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
                r += 2;
            }
            else if ( *r <= MAXVARIABLES ) {
/*
                Here we have an extra symbol. Store dimension in the compiler buffer
*/
                i = MAXVARIABLES - *r;
                dim = cbuf[AM.sbufnum].dimension[i];
                if ( dim ==  MAXPOSITIVE ) goto undefined;
                if ( dim == -MAXPOSITIVE ) goto outofrange;
                x += dim * r[1];
                if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
                r += 2;
            }
            else { r += 2; }
        }
    }
    else if ( *subterm == DOTPRODUCT ) {
        r = subterm + 2;
        while ( r < rstop ) {
            dim = vectors[*r-AM.OffsetVector].dimension;
            if ( dim == MAXPOSITIVE ) goto undefined;
            x += dim * r[2];
            if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
            dim = vectors[r[1]-AM.OffsetVector].dimension;
            if ( dim == MAXPOSITIVE ) goto undefined;
            x += dim * r[2];
            if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
            r += 3;
        }
    }
    else if ( *subterm == VECTOR ) {
        r = subterm + 2;
        while ( r < rstop ) {
            dim = vectors[*r-AM.OffsetVector].dimension;
```

```
            if ( dim == MAXPOSITIVE ) goto undefined;
            x += dim;
            if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
            r += 2;
        }
    }
    else if ( *subterm == INDEX ) {
        r = subterm + 2;
        while ( r < rstop ) {
            if ( *r < 0 ) {
                dim = vectors[*r-AM.OffsetVector].dimension;
                if ( dim == MAXPOSITIVE ) goto undefined;
                x += dim;
                if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
            }
            r++;
        }
    }
    else if ( *subterm >= FUNCTION ) {
        dim = functions[*subterm-FUNCTION].dimension;
        if ( dim == MAXPOSITIVE ) goto undefined;
        x += dim;
        if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
        if ( functions[*subterm-FUNCTION].spec > 0 ) {  /* tensor */
            r = subterm + FUNHEAD;
            while ( r < rstop ) {
                if ( *r < 0 ) {
                    dim = vectors[*r-AM.OffsetVector].dimension;
                    if ( dim == MAXPOSITIVE ) goto undefined;
                    x += dim;
                    if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
                }
                r++;
            }
        }
    }
    return((WORD)x);
undefined:
    return((WORD)MAXPOSITIVE);
outofrange:
    return(-(WORD)MAXPOSITIVE);
}

/*
    #] DimensionSubterm :
    #[ DimensionTerm :

    Returns the dimension of the given term.
    If there is any variable of which the dimension is not defined
    we return the code for undefined which is MAXPOSITIVE
    When the value is out of range we return -MAXPOSITIVE
*/

WORD DimensionTerm(WORD *term)
{
    WORD *t, *tstop, dim;
    LONG x = 0;
    tstop = term + *term; tstop -= ABS(tstop[-1]);
    t = term+1;
    while ( t < tstop ) {
        dim = DimensionSubterm(t);
        if ( dim ==  MAXPOSITIVE ) goto undefined;
        if ( dim == -MAXPOSITIVE ) goto outofrange;
        x += dim;
        if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
        t += t[1];
    }
    return((WORD)x);
undefined:
    return((WORD)MAXPOSITIVE);
outofrange:
    return(-(WORD)MAXPOSITIVE);
}

/*
    #] DimensionTerm :
    #[ DimensionExpression :

    Returns the dimension of the given expression.
    If there is any variable of which the dimension is not defined
    we return the code for undefined which is MAXPOSITIVE
    When the value is out of range we return -MAXPOSITIVE
    When the value is not consistent we return -MAXPOSITIVE.
*/

WORD DimensionExpression(PHEAD WORD *expr)
{
```

```
    WORD dim, *term, *old, x = 0;
    int first = 1;
    term = expr;
    while ( *term ) {
        dim = DimensionTerm(term);
        if ( dim ==  MAXPOSITIVE ) goto undefined;
        if ( dim == -MAXPOSITIVE ) goto outofrange;
        if ( first ) { x = dim; }
        else if ( x != dim ) {
            old  = AN.currentTerm;
            MLOCK(ErrorMessageLock);
            MesPrint("Dimension is not the same in the terms of the expression");
            term = expr;
            while ( *term ) {
                AN.currentTerm = term;
                MesPrint("   %T");
            }
            MUNLOCK(ErrorMessageLock);
            AN.currentTerm = old;
            return(-(WORD)MAXPOSITIVE);
        }
        term += *term;
    }
    return((WORD)x);
undefined:
    return((WORD)MAXPOSITIVE);
outofrange:
    old  = AN.currentTerm;
    AN.currentTerm = term;
    MLOCK(ErrorMessageLock);
    MesPrint("Dimension out of range in %t in subexpression");
    MUNLOCK(ErrorMessageLock);
    AN.currentTerm = old;
    return(-(WORD)MAXPOSITIVE);
}

/*
    #] DimensionExpression :
    #[ Multiply Term :
        #[ MultDo :
*/

WORD MultDo(PHEAD WORD *term, WORD *pattern)
{
    GETBIDENTITY
    WORD *t, *r, i;
    t = term + *term;
    if ( pattern[2] > 0 ) {         /* Left multiply */
        i = *term - 1;
    }
    else {                          /* Right multiply */
        i = ABS(t[-1]);
    }
    *term += SUBEXPSIZE;
    r = t + SUBEXPSIZE;
    do { *--r = *--t; } while ( --i > 0 );
    r = pattern + 3;
    i = r[1];
    while ( --i >= 0 ) *t++ = *r++;
    AT.WorkPointer = term + *term;
    return(0);
}

/*
        #] MultDo :
    #] Multiply Term :
    #[ Try Term(s) :
        #[ TryDo :
*/

WORD TryDo(PHEAD WORD *term, WORD *pattern, WORD level)
{
    GETBIDENTITY
    WORD *t, *r, *m, i, j;
    ReNumber(BHEAD term);
    Normalize(BHEAD term);
    m = r = term + *term;
    m++;
    i = pattern[2];
    t = pattern + 3;
    NCOPY(m,t,i)
    j = *term - 1;
    t = term + 1;
    NCOPY(m,t,j)
    *r = WORDDIF(m,r);
    AT.WorkPointer = m;
    if ( ( j = Normalize(BHEAD r) ) == 0 || j == 1 ) {
```

```
        if ( *r == 0 ) return(0);
        ReNumber(BHEAD r); Normalize(BHEAD r);
        if ( *r == 0 ) return(0);
        if ( ( i = CompareTerms(BHEAD term,r,0) ) < 0 ) {
            *AN.RepPoint = 1;
            AR.expchanged = 1;
            return(Generator(BHEAD r,level));
        }
        if ( i == 0 && CompCoef(term,r) != 0 ) { return(0); }
    }
    AT.WorkPointer = r;
    return(Generator(BHEAD term,level));
}

/*
        #] TryDo :
    #] Try Term(s) :
    #[ Distribute :
        #[ DoDistrib :

        The routine that generates the terms ordered by a distrib_ function.
        The presence of a replaceable distrib_ function has been sensed
        in the routine TestSub and has been passed on to Generator.
        It is then Generator that calls this function in a way that is
        similar to calling the trace routines, except for that for the
        trace routines and the Levi-Civita tensors the arguments are put
        in temporary storage and here we leave them inside the term,
        because there is no knowing how long the field will be.
*/

WORD DoDistrib(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    WORD *t, *m, *r = 0, *stop, *tstop, *termout, *endhead, *starttail, *parms;
    WORD i, j, k, n, nn, ntype, fun1 = 0, fun2 = 0, typ1 = 0, typ2 = 0;
    WORD *arg, *oldwork, *mf, ktype = 0, atype = 0;
    WORD sgn, dirtyflag;
    AN.TeInFun = AR.TePos = 0;
    t = term;
    tstop = t + *t;
    stop = tstop - ABS(tstop[-1]);
    t++;
    while ( t < stop ) {
        r = t + t[1];
        if ( *t == DISTRIBUTION && t[FUNHEAD] == -SNUMBER
        && t[FUNHEAD+1] >= -2 && t[FUNHEAD+1] <= 2
        && t[FUNHEAD+2] == -SNUMBER
        && t[FUNHEAD+4] <= -FUNCTION
        && t[FUNHEAD+5] <= -FUNCTION ) {
            WORD *ttt = t+FUNHEAD+6, *tttstop = t+t[1];
            while ( ttt < tttstop ) {
                if ( *ttt == -DOLLAREXPRESSION ) break;
                NEXTARG(ttt);
            }
            if ( ttt >= tttstop ) {
                fun1 = -t[FUNHEAD+4];
                fun2 = -t[FUNHEAD+5];
                typ1 = functions[fun1-FUNCTION].spec;
                typ2 = functions[fun2-FUNCTION].spec;
                if ( typ1 > 0 || typ2 > 0 ) {
                    m = t + FUNHEAD+6;
                    r = t + t[1];
                    while ( m < r ) {
                        if ( *m != -INDEX && *m != -VECTOR && *m != -MINVECTOR )
                            break;
                        m += 2;
                    }
                    if ( m < r ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Incompatible function types and arguments in distrib_");
                        MUNLOCK(ErrorMessageLock);
                        SETERROR(-1)
                    }
                }
                break;
            }
        }
        t = r;
    }
    dirtyflag = t[2];
    ntype = t[FUNHEAD+1];
    n = t[FUNHEAD+3];
/*
    t points at the distrib_ function to be expanded.
    fun1,fun2 and typ1,typ2 are the two functions and their types.
    ntype indicates the action:
        0:  Make all possible divisions:  2^nargs
```

```
        1:  fun1 should get n arguments:  nargs! / ( n! (nargs-n)! )
        2:  fun2 should get n arguments:  nargs! / ( n! (nargs-n)! )
    The distiction between 1 and two is for noncommuting objects.
        3:  fun1 should get n arguments. Super symmetric option.
        4:  fun2 idem
    The super symmetric option involves:
        a:  arguments get sorted
        b:  identical arguments are seen as such. Hence not all their
            distributions are taken into account. It is as if after the
            distrib there is a symmetrize fun1; symmetrize fun2;
        c:  Hence if the occurrence of each argument is a,b,c,...
            and their occurrence in fun1 is a1,b1,c1,... and in fun2
            is a2,b2,c2,... then each term is generated (a1+a2)!/a1!/a2!
            (b1+b2)!/b1!/b2! (c1+c2)!/c1!/c2! ... times.
        d:  We have to make an array of occurrences and positions.
        e:  Then we sort the arguments indirectly.
        f:  Next we generate the argument lists in the same way as we
            generate powers of expressions with binomials. Hence we need
            a third array to keep track of the `powers'
*/
    endhead = t;
    starttail = r;
    parms = m = t + FUNHEAD+6;
    i = 0;
    while ( m < r ) {   /* Count arguments */
        i++;
        NEXTARG(m);
    }
    oldwork = AT.WorkPointer;
    arg = AT.WorkPointer + 1;
    arg[-1] = 0;
    termout = arg + i;
    switch ( ntype ) {
        case  0: ktype = 1; atype = n < 0 ? 1: 0; n = 0; break;
        case  1: ktype = 1; atype = 0; break;
        case  2: ktype = 0; atype = 0; break;
        case -1: ktype = 1; atype = 1; break;
        case -2: ktype = 0; atype = 1; break;
    }
    do {
/*
        All distributions with n elements. We generate the array arg with
        all possible 1 and 0 patterns. 1 means in fun1 and 0 means in fun2.
*/
        if ( n > i ) return(0);     /* 0 elements */

        for ( j = 0; j < n; j++ ) arg[j] = 1;
        for ( j = n; j < i; j++ ) arg[j] = 0;
        for(;;) {
            sgn = 0;
            t = term;
            m = termout;
            while ( t < endhead ) *m++ = *t++;
            mf = m;
            *m++ = fun1;
            *m++ = FUNHEAD;
            *m++ = dirtyflag;
#if FUNHEAD > 3
            k = FUNHEAD -3;
            while ( k-- > 0 ) *m++ = 0;
#endif
            r = parms;
            for ( k = 0; k < i; k++ ) {
                if ( arg[k] == ktype ) {
                    if ( *r <= -FUNCTION ) *m++ = *r++;
                    else if ( *r < 0 ) {
                        if ( typl > 0 ) {
                            if ( *r == -MINVECTOR ) sgn ^= 1;
                            r++;
                            *m++ = *r++;
                        }
                        else { *m++ = *r++; *m++ = *r++; }
                    }
                    else {
                        nn = *r;
                        NCOPY(m,r,nn);
                    }
                }
                else { NEXTARG(r) }
            }
            mf[1] = WORDDIF(m,mf);
            mf = m;
            *m++ = fun2;
            *m++ = FUNHEAD;
            *m++ = dirtyflag;
#if FUNHEAD > 3
            k = FUNHEAD -3;
```

```
            while ( k-- > 0 ) *m++ = 0;
#endif
            r = parms;
            for ( k = 0; k < i; k++ ) {
                if ( arg[k] != ktype ) {
                    if ( *r <= -FUNCTION ) *m++ = *r++;
                    else if ( *r < 0 ) {
                        if ( typ2 > 0 ) {
                            if ( *r == -MINVECTOR ) sgn ^= 1;
                            r++;
                            *m++ = *r++;
                        }
                        else { *m++ = *r++; *m++ = *r++; }
                    }
                    else {
                        nn = *r;
                        NCOPY(m,r,nn);
                    }
                }
                else { NEXTARG(r) }
            }
            mf[1] = WORDDIF(m,mf);
#ifndef NUOVO
            if ( atype == 0 ) {
                WORD k1,k2;
                for ( k = 0; k < i-1; k++ ) {
                    if ( arg[k] == 0 ) continue;
                    k1 = 1; k2 = k;
                    while ( k < i-1 && EqualArg(parms,k,k+1) ) { k++; k1++; }
                    while ( k2 <= k && arg[k2] == 1 ) k2++;
                    k2 = k-k2+1;
/*
                    Now we need k1!/(k2! (k1-k2)!)
*/
                    if ( k2 != k1 && k2 != 0 ) {
                        if ( GetBinom((UWORD *)m+3,m+2,k1,k2) ) {
                            MLOCK(ErrorMessageLock);
                            MesCall("DoDistrib");
                            MUNLOCK(ErrorMessageLock);
                            SETERROR(-1)
                        }
                        m[1] = ( m[2] < 0 ? -m[2]: m[2] ) + 3;
                        *m = LNUMBER;
                        m += m[1];
                    }
                }
            }
#endif
            r = starttail;
            while ( r < tstop ) *m++ = *r++;

            if ( atype ) {      /* antisymmetric field */
                k = n;
                nn = 0;
                for ( j = 0; j < i && k > 0; j++ ) {
                    if ( arg[j] == 1 ) k--;
                    else nn += k;
                }
                sgn ^= nn & 1;
            }

            if ( sgn ) m[-1] = -m[-1];
            *termout = WORDDIF(m,termout);
            AT.WorkPointer = m;
            if ( AT.WorkPointer > AT.WorkTop ) {
                MLOCK(ErrorMessageLock);
                MesWork();
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
            *AN.RepPoint = 1;
            AR.expchanged = 1;
            if ( Generator(BHEAD termout,level) ) Terminate(-1);
#ifndef NUOVO
            {
                WORD k1;
            j = i - 1;
            k = 0;
redok:      while ( arg[j] == 1 && j >= 0 ) { j--; k++; }
            while ( arg[j] == 0 && j >= 0 ) j--;
            if ( j < 0 ) break;
            k1 = j;
            arg[j] = 0;
            while ( !atype && EqualArg(parms,j,j+1) ) {
                j++;
                if ( j >= i - k - 1 ) { j = k1; k++; goto redok; }
                arg[j] = 0;
```

```
            }
            while ( k >= 0 ) { j++; arg[j] = 1; k--; }
            j++;
            while ( j < i ) { arg[j] = 0; j++; }
            }
#else
            j = i - 1;
            k = 0;
            while ( arg[j] == 1 && j >= 0 ) { j--; k++; }
            while ( arg[j] == 0 && j >= 0 ) j--;
            if ( j < 0 ) break;
            arg[j] = 0;
            while ( k >= 0 ) { j++; arg[j] = 1; k--; }
            j++;
            while ( j < i ) { arg[j] = 0; j++; }
#endif
        }
    } while ( ntype == 0 && ++n <= i );
    AT.WorkPointer = oldwork;
    return(0);
}

/*
        #] DoDistrib :
        #[ EqualArg :

        Returns 1 if the arguments in the field are identical.
*/

WORD EqualArg(WORD *parms, WORD num1, WORD num2)
{
    WORD *t1, *t2;
    WORD i;
    t1 = parms;
    while ( --num1 >= 0 ) { NEXTARG(t1); }
    t2 = parms;
    while ( --num2 >= 0 ) { NEXTARG(t2); }
    if ( *t1 != *t2 ) return(0);
    if ( *t1 < 0 ) {
        if ( *t1 <= -FUNCTION || t1[1] == t2[1] ) return(1);
        return(0);
    }
    i = *t1;
    while ( --i >= 0 ) {
        if ( *t1 != *t2 ) return(0);
        t1++; t2++;
    }
    return(1);
}

/*
        #] EqualArg :
        #[ DoDelta3 :
*/

WORD DoDelta3(PHEAD WORD *term, WORD level)
{
    GETBIDENTITY
    WORD *t, *m, *m1, *m2, *stopper, *tstop, *termout, *dels, *taken;
    WORD *ic, *jc, *factors;
    WORD num, num2, i, j, k, knum, a;
    AN.TeInFun = AR.TePos = 0;
    tstop = term + *term;
    stopper = tstop - ABS(tstop[-1]);
    t = term+1;
    while ( ( *t != DELTA3 || ((t[1]-FUNHEAD) & 1 ) != 0 ) && t < stopper )
        t += t[1];
    if ( t >= stopper ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Internal error with dd_ function");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    m1 = t; m2 = t + t[1];
    num = t[1] - FUNHEAD;
    if ( num == 0 ) {
        termout = t = AT.WorkPointer;
        m = term;
        while ( m < m1 ) *t++ = *m++;
        m = m2; while ( m < tstop ) *t++ = *m++;
        *termout = WORDDIF(t,termout);
        AT.WorkPointer = t;
        *AN.RepPoint = 1;
        AR.expchanged = 1;
        if ( Generator(BHEAD termout,level) ) {
            MLOCK(ErrorMessageLock);
            MesCall("Do dd_");
```

```
            MUNLOCK(ErrorMessageLock);
            SETERROR(-1)
        }
        AT.WorkPointer = termout;
        return(0);
    }
    t += FUNHEAD;
/*
    Step 1: sort the arguments
*/
    for ( i = 1; i < num; i++ ) {
        if ( t[i] < t[i-1] ) {
            a = t[i]; t[i] = t[i-1]; t[i-1] = a;
            j = i - 1;
            while ( j > 0 ) {
                if ( t[j] >= t[j-1] ) break;
                a = t[j]; t[j] = t[j-1]; t[j-1] = a;
                j--;
            }
        }
    }
/*
    Step 2: Order them by occurrence
    In 'taken' we have the array with the number of occurrences.
    in 'dels' is the type of object.
*/
    m = taken = AT.WorkPointer;
    for ( i = 0; i < num; i++ ) *m++ = 0;
    dels = m; knum = 0;
    for ( i = 0; i < num; knum++ ) {
        *m++ = t[i]; i++; taken[knum] = 1;
        while ( i < num ) {
            if ( t[i] != t[i-1] ) break;
            i++; (taken[knum])++;
        }
    }
    for ( i = 0; i < knum; i++ ) *m++ = taken[i];
    ic = m; num2 = num/2;
    jc = ic + num2;
    factors = jc + num2;
    termout = factors + num2;
/*
    The recursion has num/2 steps
*/
    k = 0;
    while ( k >= 0 ) {
        if ( k >= num2 ) {
            t = termout; m = term;
            while ( m < m1 ) *t++ = *m++;
            *t++ = DELTA; *t++ = num+2;
            for ( i = 0; i < num2; i++ ) {
                *t++ = dels[ic[i]]; *t++ = dels[jc[i]];
            }
            for ( i = 0; i < num2; i++ ) {
                if ( ic[i] == jc[i] ) {
                    j = 1;
                    while ( i < num2-1 && ic[i] == ic[i+1] && ic[i] == jc[i+1] )
                        { i++; j++; }
                    for ( a = 1; a < j; a++ ) {
                        *t++ = SNUMBER; *t++ = 4; *t++ = 2*a+1; *t++ = 1;
                    }
                    for ( a = 0; a+1+i < num2; a++ ) {
                        if ( ic[a+i] != ic[a+i+1] ) break;
                    }
                    if ( a > 0 ) {
                        if ( GetBinom((UWORD *)(t+3),t+2,2*j+a,a) ) {
                            MLOCK(ErrorMessageLock);
                            MesCall("Do dd_");
                            MUNLOCK(ErrorMessageLock);
                            SETERROR(-1)
                        }
                        t[1] = ( t[2] < 0 ? -t[2]: t[2] ) + 3;
                        *t = LNUMBER;
                        t += t[1];
                    }
                }
                else if ( factors[i] != 1 ) {
                    *t++ = SNUMBER; *t++ = 4; *t++ = factors[i]; *t++ = 1;
                }
            }
            for ( i = 0; i < num2-1; i++ ) {
                if ( ic[i] == jc[i] ) continue;
                j = 1;
                while ( i < num2-1 && jc[i] == jc[i+1] && ic[i] == ic[i+1] ) {
                    i++; j++;
                }
                for ( a = 0; a+i < num2-1; a++ ) {
```

```
                    if ( ic[i+a] != ic[i+a+1] ) break;
                }
                if ( a > 0 ) {
                    if ( GetBinom((UWORD *)(t+3),t+2,j+a,a) ) {
                        MLOCK(ErrorMessageLock);
                        MesCall("Do dd_");
                        MUNLOCK(ErrorMessageLock);
                        SETERROR(-1)
                    }
                    t[1] = ( t[2] < 0 ? -t[2]: t[2] ) + 3;
                    *t = LNUMBER;
                    t += t[1];
                }
            }
            m = m2;
            while ( m < tstop ) *t++ = *m++;
            *termout = WORDDIF(t,termout);
            AT.WorkPointer = t;
            *AN.RepPoint = 1;
            AR.expchanged = 1;
            if ( Generator(BHEAD termout,level) ) {
                MLOCK(ErrorMessageLock);
                MesCall("Do dd_");
                MUNLOCK(ErrorMessageLock);
                SETERROR(-1)
            }
            k--;
            if ( k >= 0 ) goto nextj;
            else break;
        }
        for ( ic[k] = 0; ic[k] < knum; ic[k]++ ) {
            if ( taken[ic[k]] > 0 ) break;
        }
        if ( k > 0 && ic[k-1] == ic[k] ) jc[k] = jc[k-1];
        else jc[k] = ic[k];
        for ( ; jc[k] < knum; jc[k]++ ) {
            if ( taken[jc[k]] <= 0 ) continue;
            if ( ic[k] == jc[k] ) {
                if ( taken[jc[k]] <= 1 ) continue;
/*
                factors[k] = taken[ic[k]];
                if ( ( factors[k] & 1 ) == 0 ) (factors[k])--;
*/
                taken[ic[k]] -= 2;
            }
            else {
                factors[k] = taken[jc[k]];
                (taken[ic[k]])--; (taken[jc[k]])--;
            }
            k++;
            goto nextk;  /* This is the simulated recursion */
nextj:;
            (taken[ic[k]])++; (taken[jc[k]])++;
        }
        k--;
        if ( k >= 0 ) goto nextj;
nextk:;
    }
    AT.WorkPointer = taken;
    return(0);
}

/*
        #] DoDelta3 :
        #[ TestPartitions :

        Checks whether the function in tfun is a partitions_ function
        that can be expanded. If it can a number of relevant objects is
        inside the struct parti.
        This test is not entirely trivial because there are many restrictions
        w.r.t. the arguments.
        Syntax (still to be implemented)
        partitions_(number_of_partition_entries,[function,number,]^nope,arguments)
        [function,number,]: can be
            f,3     for a partition of 3 arguments
            f,0     for the remaining arguments (should be last)
            num1,f,num2 with num1 effectively a number of partitions but this
                    counts as num1 entries.
            0,f,num2: all partitions have num2 arguments. No number of partition
                    entries needed. If num2 does not divide the number of
                    arguments there will be no action.
*/

WORD TestPartitions(PHEAD WORD *tfun, PARTI *parti)
{
    WORD *tnext = tfun + tfun[1];
    WORD *t, *tt;
```

```
    WORD argcount = 0, sum = 0, i, ipart, argremain;
    WORD tensorflag = 0;
    parti->psize = parti->nfun = parti->args = parti->nargs = 0;
    parti->numargs = parti->numpart = parti->where = 0;
    tt = t = tfun + FUNHEAD;
    while ( t < tnext ) { argcount++; NEXTARG(t); }
    if ( argcount < 1 ) goto No;
    t = tt;
    if ( *t != -SNUMBER ) goto No;
    if ( t[1] == 0 ) {
        t += 2;
        if ( *t <= -FUNCTION && t[1] == -SNUMBER && t[2] > 0 ) {
            if ( functions[-*t-FUNCTION].spec > 0 ) tensorflag = 1;
            if ( argcount-3 < 0 ) goto No;
            if ( ( (argcount-3) % t[2] ) != 0 ) goto No;
        }
        else goto No;
        parti->numpart = (argcount-3)/t[2];
        parti->numargs = argcount - 3;
        parti->psize = (WORD *)Malloc1((parti->numpart*2+parti->numargs*2+2)
                            *sizeof(WORD),"partitions");
        parti->nfun = parti->psize + parti->numpart;
        parti->args = parti->nfun + parti->numpart;
        parti->nargs = parti->args + parti->numargs;
        for ( i = 0; i < parti->numpart; i++ ) {
            parti->psize[i] =  t[2];
            parti->nfun[i]  = -t[0];
        }
        t += 3;
    }
    else if ( t[1] > 0 ) { /* Number of partitions */
/*
        We can have sequences of function,number for one partition
        or number1,function,number2 for number1 partitions of size number2.
        The last partition can have number=0. It must be a single partition
        and it will take all remaining arguments.
        If any of the functions is a tensor, all arguments must be either
        vector or index.
*/
        parti->numpart = t[1]; t += 2;
        ipart = sum = 0; argremain = argcount - 1;
/*
        At this point is seems better to make an allocation already that
        may be too big. The alternative is having to pass this code twice.
*/
        parti->psize = (WORD *)Malloc1((argcount*4+2)*sizeof(WORD),"partitions");
        parti->nfun = parti->psize+argcount;
        parti->args = parti->nfun+argcount;
        parti->nargs = parti->args+argcount;
        while ( ipart < parti->numpart ) {
            if ( *t <= -FUNCTION && t[1] == -SNUMBER && t[2] >= 0 ) {
                if ( functions[-*t-FUNCTION].spec > 0 ) tensorflag = 1;
                if ( t[2] == 0 ) {
                    if ( ipart+1 != parti->numpart ) goto WhatAPity;
                    argremain -= 2;
                    parti->nfun[ipart] = -*t;
                    parti->psize[ipart++] = argremain-sum;
                    ipart++;
                    sum = argremain;
                }
                else {
                    parti->nfun[ipart] = -*t;
                    parti->psize[ipart++] = t[2];
                    argremain -= 2;
                    sum += t[2];
                }
                t += 3;
            }
            else if ( *t == -SNUMBER && t[1] > 0 && ipart+t[1] <= parti->numpart
            && t[2] <= -FUNCTION && t[3] == -SNUMBER && t[4] > 0 ) {
                if ( functions[-t[2]-FUNCTION].spec > 0 ) tensorflag = 1;
                argremain -= 3;
                for ( i = 0; i < t[1]; i++ ) {
                    parti->nfun[ipart] = -t[2];
                    parti->psize[ipart++] = t[4];
                    sum += t[4];
                }
                if ( sum > argremain ) goto WhatAPity;
                t += 5;
            }
            else goto WhatAPity;
        }
        if ( sum != argremain ) goto WhatAPity;
        parti->numargs = argremain;
    }
    else goto No;
/*
```

```
    Now load the offsets of the arguments and check if needed whether OK with tensor
*/
    for ( i = 0; i < parti->numargs; i++ ) {
        parti->args[i] = t - tfun;
        if ( tensorflag && ( *t != -VECTOR && *t != -INDEX ) ) goto WhatAPity;
        NEXTARG(t);
    }
    return(1);
WhatAPity:
    M_free(parti->psize,"partitions");
    parti->psize = parti->nfun = parti->args = parti->nargs = 0;
    parti->numargs = parti->numpart = parti->where = 0;
No:
    return(0);
}

/*
        #] TestPartitions :
        #[ DoPartitions :

    As we have only one AT.partitions we need to copy it locally
    if we keep needing it.
*/

WORD DoPartitions(PHEAD WORD *term, WORD level)
{
    WORD x, i, j, im, *fun, ndiff, siz, tensorflag = 0;
    PARTI part = AT.partitions;
    WORD *array, **j3, **j3fill, **j3where;
    WORD a, pfill, *j2, *j2fill, j3size, ncoeff, ncoeffnum, nfac, ncoeff2, ncoeff3, n;
    UWORD *coeff, *coeffnum, *cfac, *coeff2, *coeff3, *c;
    /* Make AT.partitions ready for future use (if there is another function) */
    AT.partitions.psize = AT.partitions.nfun = AT.partitions.args = AT.partitions.nargs = 0;
    AT.partitions.numargs = AT.partitions.numpart = AT.partitions.where = 0;
/*
    Start with bubble sorting the list of arguments. And the list of partitions.
*/
    fun = term + part.where;
    if ( functions[*fun-FUNCTION].spec ) tensorflag = 1;
    for ( i = 1; i < part.numargs; i++ ) {
        for ( j = i-1; j >= 0; j-- ) {
            if ( CompArg(fun+part.args[j+1],fun+part.args[j]) >= 0 ) break;
            x = part.args[j+1]; part.args[j+1] = part.args[j]; part.args[j] = x;
        }
    }
    for ( i = 1; i < part.numpart; i++ ) {
        for ( j = i-1; j >= 0; j-- ) {
            if ( part.psize[j+1] < part.psize[j] ) break;
            if ( part.psize[j+1] == part.psize[j] && part.nfun[j+1] <= part.nfun[j] ) break;
            x = part.psize[j+1]; part.psize[j+1] = part.psize[j]; part.psize[j] = x;
            x = part.nfun[j+1]; part.nfun[j+1] = part.nfun[j]; part.nfun[j] = x;
        }
    }
/*
    Now we have the partitions sorted from high to low and the arguments
    have been sorted the regular way arguments are sorted in a symmetrize.
    The important thing is that identical arguments are adjacent.
    Assign the numbers (identical arguments have identical numbers).
*/
    ndiff = 1; part.nargs[0] = ndiff;
    for ( i = 1; i < part.numargs; i++ ) {
        if ( CompArg(fun+part.args[i],fun+part.args[i-1]) != 0 ) ndiff++;
        part.nargs[i] = ndiff;
    }
    part.nargs[part.numargs] = 0;
    coeffnum = NumberMalloc("partitionsn");
    coeff = NumberMalloc("partitions");
    coeff2 = NumberMalloc("partitions2");
    coeff3 = NumberMalloc("partitions3");
    cfac = NumberMalloc("partitions!");
    ncoeffnum = 1; coeffnum[0] = 1;
/*
    The numerator of the coefficient will be n1!*n2!*...*n(ndiff)!
    We compute it only once.
*/
    j = 0;
    for ( i = 1; i <= ndiff; i++ ) {
        n = 0;
        while ( part.nargs[j] == i ) { n++; j++; }
        if ( n > 1 ) { /* 1! needs no attention */
            if ( Factorial(BHEAD n, cfac, &nfac) ) Terminate(-1);
            if ( MulLong(coeffnum,ncoeffnum,cfac,nfac,coeff2,&ncoeff2) ) Terminate(-1);
            c = coeffnum; coeffnum = coeff2; coeff2 = c;
            n = ncoeffnum; ncoeffnum = ncoeff2; ncoeff2 = n;
        }
    }
/*
```

```
    Now comes the part where we have to make sure that
    a: we generate all partitions.
    b: we generate only different partitions.
    c: we get the proper combinatorics factor.
    Method:
    Suppose the largest partition needs n objects and there are m partitions.
    We allocate m arrays of n 'digits'. Make in the smaller partitions the
    appropriate leading digits zero.
    Divide the largest numbers (of the arguments) over the partitions as
    leftmost digits (after possible zeroes). The arrays, seen as numbers,
    should be such that each is less or equal to its left neighbour. Take the
    next largest numbers, etc. This generates unique partitions and all of
    them. Because we have a formula for the multiplicity, this should do it.

    The general case. At a later stage we might put in a more economical
    version for special cases.
*/
    siz = part.psize[0];
    j3size = 2*(part.numpart+1)+2*(part.numargs+1);
    array = (WORD *)Malloc1((part.numpart+1)*siz*sizeof(WORD),"parts");
    j3 = (WORD **)Malloc1(j3size*sizeof(WORD *),"parts3");
    j2 = (WORD *)Malloc1((part.numpart+part.numargs+2)*sizeof(WORD),"parts2");
    j3fill = j3+(part.numpart+1);
    j3where = j3fill+(part.numpart+1);
    for ( i = 0; i < j3size; i++ ) j3[i] = 0;
    j2fill = j2+(part.numpart+1);
    for ( i = 0; i < part.numargs; i++ ) j2fill[i] = 0;
    for ( i = 0; i < part.numpart; i++ ) {
        j3[i] = array+i*siz;
        for ( j = 0; j < siz; j++ ) j3[i][j] = 0;
        j3fill[i] = j3[i]+(siz-part.psize[i]);
        j2[i] = part.psize[i];  /* Number of places still available */
    }
    j3[part.numpart] = array+part.numpart*siz;
    j2[part.numpart] = 0;
/*
    Now comes a complicated two-level recursion in a and pfill.
*/
    a = part.numargs-1;
    pfill = 0;
/*
    We start putting the last number in part.nargs in the first partition in array.
    For backtracking we need to know where we put this number. Hence j3where.
*/
    while ( a < part.numargs ) {
        while ( j2[pfill] <= 0 ) {
            pfill++;
            while ( pfill >= part.numpart ) { /* we have to pop */
                a++;
                if ( a >= part.numargs ) goto Done;
                pfill = j2fill[a];
                j2[pfill]++;
                j3where[a][0] = 0;
                j3fill[pfill]--;
                pfill++;
            }
        }
        j3where[a] = j3fill[pfill];
        *(j3fill[pfill])++ = part.nargs[a];
        j2[pfill]--; j2fill[a] = pfill;
/*
        Now test whether this is allowed.
*/
        if ( pfill > 0 && part.psize[pfill] == part.psize[pfill-1]
             && part.nfun[pfill] == part.nfun[pfill-1] ) { /* First check whether allowed */
            for ( im = 0; im < siz; im++ ) {
                if ( j3[pfill-1][im] < j3[pfill][im] ) break;
                if ( j3[pfill-1][im] > j3[pfill][im] ) im = siz;
            }
            if ( im < siz ) { /* not ordered. undo and raise pfill */
                pfill = j2fill[a];
                j2[pfill]++;
                j3where[a][0] = 0;
                j3fill[pfill]--;
                pfill++;
                continue; /* Note that j2[part.numpart] = 0 */
            }
        }
        a--;
        if ( a < 0 ) {  /* Solution */
/*
            #[ Solution :

            Now we compose the output term. The input term contains
            three parts: head, partitions_, tail.
            partitions_ starts at term+part.where.
            We first put the function parts and worry about the coefficient later.
```

```
*/
            WORD *t, *to, *twhere = term+part.where, *t2, *tend = term+*term, *termout;
            WORD num, jj, *targ, *tfun;
            t2 = twhere+twhere[1];
            to = termout = AT.WorkPointer;
            if ( termout + *term + part.numpart*FUNHEAD + AM.MaxTal >= AT.WorkTop ) {
                return(MesWork());
            }
            for ( i = 0; i < ncoeffnum; i++ ) coeff[i] = coeffnum[i];
            ncoeff = ncoeffnum;
            t = term; while ( t < twhere ) *to++ = *t++;
/*
            Now the partitions
*/
            for ( i = 0; i < part.numpart; i++ ) {
                tfun = to;
                *to++ = part.nfun[i]; to++; FILLFUN(to);
                for ( j = 1; j <= part.psize[i]; j++ ) {
                    num = j3[i][siz-j]; /* now we need an argument with this number */
                    for ( jj = num-1; jj < part.numargs; jj++ ) {
                        if ( part.nargs[jj] == num ) break;
                    }
                    targ = part.args[jj]+twhere;
                    if ( *targ < 0 ) {
                        if ( tensorflag ) targ++;
                        else if ( *targ > -FUNCTION ) *to++ = *targ++;
                        *to++ = *targ++;
                    }
                    else { jj = *targ; NCOPY(to,targ,jj); }
                }
                tfun[1] = to - tfun;
            }
/*
            Now the denominators of the coefficient
            First identical functions/partitions
*/
            j = 1; n = 1;
            while ( j < part.numpart ) {
                for ( im = 0; im < siz; im++ ) {
                    if ( part.nfun[j-1] != part.nfun[j] ) break;
                    if ( j3[j-1][im] < j3[j][im] ) break;
                    if ( j3[j-1][im] > j3[j][im] ) im = 2*siz+2;
                }
                if ( im == siz ) { n++; j++; continue; }
                if ( n > 1 ) {
div1:               if ( Factorial(BHEAD n, cfac, &nfac) ) Terminate(-1);
                    if ( DivLong(coeff,ncoeff,cfac,nfac,coeff2,&ncoeff2,coeff3,&ncoeff3) )
    Terminate(-1);
                    c = coeff; coeff = coeff2; coeff2 = c;
                    n = ncoeff; ncoeff = ncoeff2; ncoeff2 = n;
                }
                n = 1; j++;
            }
            if ( n > 1 ) goto div1;
/*
            Now identical elements inside the partitions
*/
            for ( i = 0; i < part.numpart; i++ ) {
                j = 0; while ( j3[i][j] == 0 ) j++;
                n = 1; j++;
                while ( j < siz ) {
                    if ( j3[i][j-1] == j3[i][j] ) { n++; j++; }
                    else {
                        if ( n > 1 ) {
div2:                       if ( Factorial(BHEAD n, cfac, &nfac) ) Terminate(-1);
                            if ( DivLong(coeff,ncoeff,cfac,nfac,coeff2,&ncoeff2,coeff3,&ncoeff3) )
    Terminate(-1);
                            c = coeff; coeff = coeff2; coeff2 = c;
                            n = ncoeff; ncoeff = ncoeff2; ncoeff2 = n;
                        }
                        n = 1; j++;
                    }
                }
                if ( n > 1 ) goto div2;
            }
/*
            And put this inside the term. Normalize will take care of it.
*/
            if ( ncoeff != 1 || coeff[0] > 1 ) {
                if ( ncoeff == 1 && coeff[0] <= MAXPOSITIVE ) {
                    *to++ = SNUMBER; *to++ = 4; *to++ = (WORD)(coeff[0]); *to++ = 1;
                }
                else {
                    *to++ = LNUMBER; *to++ = ncoeff+3; *to++ = ncoeff;
                    for ( i = 0; i < ncoeff; i++ ) *to++ = ((WORD *)coeff)[i];
                }
            }
```

```
/*
            And the tail
*/
            while ( t2 < tend ) *to++ = *t2++;
            *termout = to-termout;
            AT.WorkPointer = to;
            if ( Generator(BHEAD termout,level) ) Terminate(-1);
            AT.WorkPointer = termout;
/*
            #] Solution :

            Now we can pop all a with the lowest value and one more.
*/
            a = 0;
            while ( part.nargs[a] == 1 ) {
                pfill = j2fill[a]; j2[pfill]++; j3where[a][0] = 0; j3fill[pfill]--; a++;
            }
            if ( a < part.numargs ) {
                pfill = j2fill[a]; j2[pfill]++; j3where[a][0] = 0; j3fill[pfill]--; a++;
            }
            a--;
            pfill++;
        }
        else if ( part.nargs[a] == part.nargs[a+1] ) {}
        else { pfill = 0; }
    }
Done:
    M_free(j2,"parts2");
    M_free(j3,"parts3");
    M_free(array,"parts");
    NumberFree(cfac,"partitions!");
    NumberFree(coeff3,"partitions3");
    NumberFree(coeff2,"partitions2");
    NumberFree(coeff,"partitions");
    NumberFree(coeffnum,"partitionsn");
    M_free(part.psize,"partitions");
    part.psize = part.nfun = part.args = part.nargs = 0;
    part.numargs = part.numpart = part.where = 0;
    return(0);
}

/*
        #] DoPartitions :
    #] Distribute :
    #[ DoPermutations :

    Routine replaces the function perm_(f,args) by occurrences of f with
    all permutations of the args. This should always fit!
*/

WORD DoPermutations(PHEAD WORD *term, WORD level)
{
    PERMP perm;
    WORD *oldworkpointer = AT.WorkPointer, *termout = AT.WorkPointer;
    WORD *t, *tstop, *tt, *ttstop, odd = 0;
    WORD *args[MAXMATCH], nargs, i, first, skip, *to, *from;
/*
    Find function and count arguments. Check for odd/even
*/
    tstop = term+*term; tstop -= ABS(tstop[-1]);
    t = term+1;
    while ( t < tstop ) {
        if ( *t == PERMUTATIONS ) {
            if ( t[1] >= FUNHEAD+1 && t[FUNHEAD] <= -FUNCTION ) {
                odd = 0; skip = 1;
            }
            else if ( t[1] >= FUNHEAD+3 && t[FUNHEAD] == -SNUMBER && t[FUNHEAD+2] <= -FUNCTION ) {
                if ( t[FUNHEAD+1] % 2 == 1 ) odd = -1;
                else odd = 0;
                skip = 3;
            }
            else { t += t[1]; continue; }
            tt = t+FUNHEAD+skip; ttstop = t + t[1];
            nargs = 0;
            while ( tt < ttstop ) { NEXTARG(tt); nargs++; }
            tt = t+FUNHEAD+skip;
            if ( nargs > MAXMATCH ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Too many arguments in function perm_. %d! is way too big",(WORD)MAXMATCH);
                MUNLOCK(ErrorMessageLock);
                SETERROR(-1)
            }
            i = 0;
            while ( tt < ttstop ) { args[i++] = tt; NEXTARG(tt); }
            perm.n = nargs;
            perm.sign = 0;
            perm.objects = args;
```

```
            first = 1;
            while ( (first = PermuteP(&perm,first) ) == 0 ) {
/*
                Compose the output term
*/
                to = termout; from = term;
                while ( from < t ) *to++ = *from++;
                *to++ = -t[FUNHEAD+skip-1];
                *to++ = t[1] - skip;
                for ( i = 2; i < FUNHEAD; i++ ) *to++ = t[i];
                for ( i = 0; i < nargs; i++ ) {
                    from = args[i];
                    COPY1ARG(to,from);
                }
                from = t+t[1];
                tstop = term + *term;
                while ( from < tstop ) *to++ = *from++;
                if ( odd && ( ( perm.sign & 1 ) != 0 ) ) to[-1] = -to[-1];
                *termout = to - termout;
                AT.WorkPointer = to;
                if ( Generator(BHEAD termout,level) ) Terminate(-1);
                AT.WorkPointer = oldworkpointer;
            }
            return(0);
        }
        t += t[1];
    }
    return(0);
}

/*
    #] DoPermutations :
    #[ DoShuffle :

    Merges the arguments of all occurrences of function fun into a
    single occurrence of fun. The opposite of Distrib_
    Syntax:
        Shuffle[,once|all],fun;
        Shuffle[,once|all],$fun;
    The expansion of the dollar should give a single function.
    The dollar is indicated as usual with a negative value.
    option = 1 (once): generate identical results only once
    option = 0 (all): generate identical results with combinatorics (default)
*/

/*
    We use the Shuffle routine which has a large amount of combinatorics.
    It doesn't have grouped combinatorics as in (0,1,2)*(0,1,3) where the
    groups (0,1) also cause double terms.
*/

WORD DoShuffle(PHEAD WORD *term, WORD level, WORD fun, WORD option)
{
    GETBIDENTITY
    SHvariables SHback, *SH = &(AN.SHvar);
    WORD *t1, *t2, *tstop, ncoef, n = fun, *to, *from;
    int i, error;
    LONG k;
    UWORD *newcombi;

    if ( n < 0 ) {
        if ( ( n = DolToFunction(BHEAD -n) ) == 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("$-variable in merge statement did not evaluate to a function.");
            MUNLOCK(ErrorMessageLock);
            return(1);
        }
    }
    if ( AT.WorkPointer + 3*(*term) + AM.MaxTal > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }

    tstop = term + *term;
    ncoef = tstop[-1];
    tstop -= ABS(ncoef);
    t1 = term + 1;
    while ( t1 < tstop ) {
        if ( ( *t1 == n ) && ( t1+t1[1] < tstop ) && ( t1[1] > FUNHEAD ) ) {
            t2 = t1 + t1[1];
            if ( t2 >= tstop ) {
                return(Generator(BHEAD term,level));
            }
            while ( t2 < tstop ) {
                if ( ( *t2 == n ) && ( t2[1] > FUNHEAD ) ) break;
```

```
                t2 += t2[1];
            }
            if ( t2 < tstop ) break;
        }
        t1 += t1[1];
    }
    if ( t1 >= tstop ) {
        return(Generator(BHEAD term,level));
    }
    *AN.RepPoint = 1;
/*
    Now we have two occurrences of the function.
    Back up all relevant variables and load all the stuff that needs to be
    passed on.
*/
    SHback = AN.SHvar;
    SH->finishuf = &FinishShuffle;
    SH->do_uffle = &DoShuffle;
    SH->outterm = AT.WorkPointer;
    AT.WorkPointer += *term;
    SH->stop1 = t1 + t1[1];
    SH->stop2 = t2 + t2[1];
    SH->thefunction = n;
    SH->option = option;
    SH->level = level;
    SH->incoef = tstop;
    SH->nincoef = ncoef;

    if ( AN.SHcombi == 0 || AN.SHcombisize == 0 ) {
        AN.SHcombisize = 200;
        AN.SHcombi = (UWORD *)Malloc1(AN.SHcombisize*sizeof(UWORD),"AN.SHcombi");
        SH->combilast = 0;
        SHback.combilast = 0;
    }
    else {
        SH->combilast += AN.SHcombi[SH->combilast]+1;
        if ( SH->combilast >= AN.SHcombisize - 100 ) {
            newcombi = (UWORD *)Malloc1(2*AN.SHcombisize*sizeof(UWORD),"AN.SHcombi");
            for ( k = 0; k < AN.SHcombisize; k++ ) newcombi[k] = AN.SHcombi[k];
            M_free(AN.SHcombi,"AN.SHcombi");
            AN.SHcombi = newcombi;
            AN.SHcombisize *= 2;
        }
    }
    AN.SHcombi[SH->combilast] = 1;
    AN.SHcombi[SH->combilast+1] = 1;

    i = t1-term; to = SH->outterm; from = term;
    NCOPY(to,from,i)
    SH->outfun = to;
    for ( i = 0; i < FUNHEAD; i++ ) { *to++ = t1[i]; }

    error = Shuffle(BHEAD t1+FUNHEAD,t2+FUNHEAD,to);

    AT.WorkPointer = SH->outterm;
    AN.SHvar = SHback;
    if ( error ) {
        MesCall("DoShuffle");
        return(-1);
    }
    return(0);
}

/*
    #] DoShuffle :
    #[ Shuffle :

    How to make shuffles:

    We have two lists of arguments. We have to make a single
    shuffle of them. All combinations. Doubles should have as
    much as possible a combinatorics factor. Sometimes this is
    very difficult as in:
        (0,1,2)x(0,1,3) = -> (0,1) is a repeated pattern and the
        factor on that is difficult
    Simple way: (without combinatorics)
        repeat id  f0(?c)*f(x1?,?a)*f(x2?,?b) =
                        +f0(?c,x1)*f(?a)*f(x2,?b)
                        +f0(?c,x2)*f(x1,?a)*f(?b);
    Refinement:
            if ( x1 == x2 ) check how many more there are of the same.
            --> (n1,x) and (n2,x)
            id  f0(?c)*f1((n1,x),?b)*f2((n2,x),?c) =
                    +binom_(n1+n2,n1)*f0(?c,(n1+n2,x))*f1(?a)*f2(?b)
                    +sum_(j,0,n1-1,binom_(n2+j,j)*f0(?c,(j+n2,x))
                            *f1((n1-j),?a)*f2(?b))*force2
                    +sum_(j,0,n2-1,binom_(n1+j,j)*f0(?c,(j+n1,x))
```

```
                            *f1(?a)*f2((n2-j),?b))*force1
    The force operation can be executed directly

    The next question is how to program this: recursively or linearly
    which would require simulation of a recursion. Recursive is clearest
    but we need to pass a number of arguments from the calling routine
    to the final routine. This is done with AN.SHvar.

    We need space for the accumulation of the combinatoric factors.
*/

int Shuffle(PHEAD WORD *from1, WORD *from2, WORD *to)
{
    WORD *t, *fr, *next1, *next2, na, *fn1, *fn2, *tt;
    int i, n, n1, n2, j;
    LONG combilast;
    SHvariables *SH = &(AN.SHvar);
    if ( from1 == SH->stop1 && from2 == SH->stop2 ) {
        return(FiniShuffle(BHEAD to));
    }
    else if ( from1 == SH->stop1 ) {
        i = SH->stop2 - from2; t = to; tt = from2; NCOPY(t,tt,i)
        return(FiniShuffle(BHEAD t));
    }
    else if ( from2 == SH->stop2 ) {
        i = SH->stop1 - from1; t = to; tt = from1; NCOPY(t,tt,i)
        return(FiniShuffle(BHEAD t));
    }
/*
    Compare lead arguments
*/
    if ( AreArgsEqual(from1,from2) ) {
/*
        First find out how many of each
*/
        next1 = from1; n1 = 1; NEXTARG(next1)
        while ( ( next1 < SH->stop1 ) && AreArgsEqual(from1,next1) ) {
            n1++; NEXTARG(next1)
        }
        next2 = from2; n2 = 1; NEXTARG(next2)
        while ( ( next2 < SH->stop2 ) && AreArgsEqual(from2,next2) ) {
            n2++; NEXTARG(next2)
        }
        combilast = SH->combilast;
/*
            +binom_(n1+n2,n1)*f0(?c,(n1+n2,x))*f1(?a)*f2(?b)
*/
        t = to;
        n = n1 + n2;
        while ( --n >= 0 ) { fr = from1; CopyArg(t,fr) }
        if ( GetBinom((UWORD *)(t),&na,n1+n2,n1) ) goto shuffcall;
        if ( combilast + AN.SHcombi[combilast] + na + 2 >= AN.SHcombisize ) {
/*
            We need more memory in this stack. Fortunately this is the
            only place where we have to do this, because the other factors
            are definitely smaller.
            Layout:   size, LongInteger, size, LongInteger, .....
            We start pointing at the last one.
*/
            UWORD *combi = (UWORD *)Malloc1(2*AN.SHcombisize*2,"AN.SHcombi");
            LONG jj;
            for ( jj = 0; jj < AN.SHcombisize; jj++ ) combi[jj] = AN.SHcombi[jj];
            AN.SHcombisize *= 2;
            M_free(AN.SHcombi,"AN.SHcombi");
            AN.SHcombi = combi;
        }
        if ( MulLong((UWORD *)(AN.SHcombi+combilast+1),AN.SHcombi[combilast],
                     (UWORD *)(t),na,
                     (UWORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+2),
                     (WORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+1)) ) goto shuffcall;
        SH->combilast = combilast + AN.SHcombi[combilast] + 1;
        if ( next1 >= SH->stop1 ) {
            fr = next2; i = SH->stop2 - fr;
            NCOPY(t,fr,i)
            if ( FiniShuffle(BHEAD t) ) goto shuffcall;
        }
        else if ( next2 >= SH->stop2 ) {
            fr = next1; i = SH->stop1 - fr;
            NCOPY(t,fr,i)
            if ( FiniShuffle(BHEAD t) ) goto shuffcall;
        }
        else {
            if ( Shuffle(BHEAD next1,next2,t) ) goto shuffcall;
        }
        SH->combilast = combilast;
/*
            +sum_(j,0,n1-1,binom_(n2+j,j)*f0(?c,(j+n2,x))
```

```
                    *f1((n1-j),?a)*f2(?b))*force2
*/
        if ( next2 < SH->stop2 ) {
         t = to;
         n = n2;
         while ( --n >= 0 ) { fr = from1; CopyArg(t,fr) }
         for ( j = 0; j < n1; j++ ) {
          if ( GetBinom((UWORD *)(t),&na,n2+j,j) ) goto shuffcall;
          if ( MulLong((UWORD *)(AN.SHcombi+combilast+1),AN.SHcombi[combilast],
                       (UWORD *)(t),na,
                       (UWORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+2),
                       (WORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+1)) ) goto shuffcall;
          SH->combilast = combilast + AN.SHcombi[combilast] + 1;
          if ( j > 0 ) { fr = from1; CopyArg(t,fr) }
          fn2 = next2; tt = t;
          CopyArg(tt,fn2)

          if ( fn2 >= SH->stop2 ) {
            n = n1-j;
            while ( --n >= 0 ) { fr = from1; CopyArg(tt,fr) }
            fr = next1; i = SH->stop1 - fr;
            NCOPY(tt,fr,i)
            if ( FiniShuffle(BHEAD tt) ) goto shuffcall;
          }
          else {
            n = j; fn1 = from1; while ( --n >= 0 ) { NEXTARG(fn1) }
            if ( Shuffle(BHEAD fn1,fn2,tt) ) goto shuffcall;
          }
          SH->combilast = combilast;
         }
        }
/*
            +sum_(j,0,n2-1,binom_(n1+j,j)*f0(?c,(j+n1,x))
                    *f1(?a)*f2((n2-j),?b))*force1
*/
        if ( next1 < SH->stop1 ) {
         t = to;
         n = n1;
         while ( --n >= 0 ) { fr = from1; CopyArg(t,fr) }
         for ( j = 0; j < n2; j++ ) {
          if ( GetBinom((UWORD *)(t),&na,n1+j,j) ) goto shuffcall;
          if ( MulLong((UWORD *)(AN.SHcombi+combilast+1),AN.SHcombi[combilast],
                       (UWORD *)(t),na,
                       (UWORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+2),
                       (WORD *)(AN.SHcombi+combilast+AN.SHcombi[combilast]+1)) ) goto shuffcall;
          SH->combilast = combilast + AN.SHcombi[combilast] + 1;
          if ( j > 0 ) { fr = from1; CopyArg(t,fr) }
          fn1 = next1; tt = t;
          CopyArg(tt,fn1)

          if ( fn1 >= SH->stop1 ) {
            n = n2-j;
            while ( --n >= 0 ) { fr = from1; CopyArg(tt,fr) }
            fr = next2; i = SH->stop2 - fr;
            NCOPY(tt,fr,i)
            if ( FiniShuffle(BHEAD tt) ) goto shuffcall;
          }
          else {
            n = j; fn2 = from2; while ( --n >= 0 ) { NEXTARG(fn2) }
            if ( Shuffle(BHEAD fn1,fn2,tt) ) goto shuffcall;
          }
          SH->combilast = combilast;
         }
        }
    }
    else {
/*
        Argument from first list
*/
        t = to;
        fr = from1;
        CopyArg(t,fr)
        if ( fr >= SH->stop1 ) {
            fr = from2; i = SH->stop2 - fr;
            NCOPY(t,fr,i)
            if ( FiniShuffle(BHEAD t) ) goto shuffcall;
        }
        else {
            if ( Shuffle(BHEAD fr,from2,t) ) goto shuffcall;
        }
/*
        Argument from second list
*/
        t = to;
        fr = from2;
        CopyArg(t,fr)
        if ( fr >= SH->stop2 ) {
```

```
            fr = from1; i = SH->stop1 - fr;
            NCOPY(t,fr,i)
            if ( FiniShuffle(BHEAD t) ) goto shuffcall;
        }
        else {
            if ( Shuffle(BHEAD from1,fr,t) ) goto shuffcall;
        }
    }
    return(0);
shuffcall:
    MesCall("Shuffle");
    return(-1);
}

/*
    #] Shuffle :
    #[ FinishShuffle :

    The complications here are:
    1: We want to save space. We put the output term in 'out' straight
       on top of what we produced thusfar. We have to copy the early
       piece because once the term goes back to Generator, Normalize can
       change it in situ
    2: There can be other occurrence of the function between the two
       that we did. For shuffles that isn't likely, but we use this
       routine also for the stuffles and there it can happen.
*/

int FinishShuffle(PHEAD WORD *fini)
{
    WORD *t, *t1, *oldworkpointer = AT.WorkPointer, *tcoef, ntcoef, *out;
    int i;
    SHvariables *SH = &(AN.SHvar);
    SH->outfun[1] = fini - SH->outfun;
    if ( functions[SH->outfun[0]-FUNCTION].symmetric != 0 )
                    SH->outfun[2] |= DIRTYSYMFLAG;
    out = fini; i = fini - SH->outterm; t = SH->outterm;
    NCOPY(fini,t,i)
    t = SH->stop1;
    t1 = t + t[1];
    while ( t1 < SH->stop2 ) { t = t1; t1 = t + t[1]; }
    t1 = SH->stop1;
    while ( t1 < t ) *fini++ = *t1++;
    t = SH->stop2;
    while ( t < SH->incoef ) *fini++ = *t++;
    tcoef = fini;
    ntcoef = SH->nincoef;
    i = ABS(ntcoef);
    NCOPY(fini,t,i);
    ntcoef = REDLENG(ntcoef);
    Mully(BHEAD (UWORD *)tcoef,&ntcoef,
        (UWORD *)(AN.SHcombi+SH->combilast+1),AN.SHcombi[SH->combilast]);
    ntcoef = INCLENG(ntcoef);
    fini = tcoef + ABS(ntcoef);
    if ( ( ( SH->option & 2 ) != 0 ) && ( ( SH->option & 256 ) != 0 ) ) ntcoef = -ntcoef;
    fini[-1] = ntcoef;
    i = *out = fini - out;
/*
    Now check whether we have to do more
*/
    AT.WorkPointer = out + *out;
    if ( ( SH->option & 1 ) == 1 ) {
        if ( Generator(BHEAD out,SH->level) ) goto Finicall;
    }
    else {
        if ( DoShtuffle(BHEAD out,SH->level,SH->thefunction,SH->option) ) goto Finicall;
    }
    AT.WorkPointer = oldworkpointer;
    return(0);
Finicall:
    AT.WorkPointer = oldworkpointer;
    MesCall("FinishShuffle");
    return(-1);
}

/*
    #] FinishShuffle :
    #[ DoStuffle :

    Stuffling is a variation of shuffling.
    In the stuffling we insist that the arguments are (short) integers. nonzero.
    The stuffle sum is x st y = sig_(x)*sig_(y)*(abs(x)+abs(y))
    The way we do this is:
        1: count the arguments in each function: n1, n2
        2: take the minimum minval = min(n1,n2).
        3: for ( j = 0; j <= min; j++ ) take j elements in each of the lists.
        4: the j+1 groups of remaining arguments have to each be shuffled
```

```
        5: the j selected pairs have to be stuffle added.
    We can use many of the shuffle things.
    Considering the recursive nature of the generation we actually don't
    need to know n1, n2, minval.
*/

WORD DoStuffle(PHEAD WORD *term, WORD level, WORD fun, WORD option)
{
    GETBIDENTITY
    SHvariables SHback, *SH = &(AN.SHvar);
    WORD *t1, *t2, *tstop, *t1stop, *t2stop, ncoef, n = fun, *to, *from;
    WORD *r1, *r2;
    int i, error;
    LONG k;
    UWORD *newcombi;
#ifdef NEWCODE
    WORD *rr1, *rr2, i1, i2;
#endif
    if ( n < 0 ) {
        if ( ( n = DolToFunction(BHEAD -n) ) == 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("$-variable in merge statement did not evaluate to a function.");
            MUNLOCK(ErrorMessageLock);
            return(1);
        }
    }
    if ( AT.WorkPointer + 3*(*term) + AM.MaxTal > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }

    tstop = term + *term;
    ncoef = tstop[-1];
    tstop -= ABS(ncoef);
    t1 = term + 1;
retry1:;
    while ( t1 < tstop ) {
        if ( ( *t1 == n ) && ( t1+t1[1] < tstop ) && ( t1[1] > FUNHEAD ) ) {
            t2 = t1 + t1[1];
            if ( t2 >= tstop ) {
                return(Generator(BHEAD term,level));
            }
retry2:;
            while ( t2 < tstop ) {
                if ( ( *t2 == n ) && ( t2[1] > FUNHEAD ) ) break;
                t2 += t2[1];
            }
            if ( t2 < tstop ) break;
        }
        t1 += t1[1];
    }
    if ( t1 >= tstop ) {
        return(Generator(BHEAD term,level));
    }
/*
    Next we have to check that the arguments are of the correct type
    At the same time we can count them.
*/
#ifndef NEWCODE
    t1stop = t1 + t1[1];
    r1 = t1 + FUNHEAD;
    while ( r1 < t1stop ) {
        if ( *r1 != -SNUMBER ) break;
        if ( r1[1] == 0 ) break;
        r1 += 2;
    }
    if ( r1 < t1stop ) { t1 = t2; goto retry1; }
    t2stop = t2 + t2[1];
    r2 = t2 + FUNHEAD;
    while ( r2 < t2stop ) {
        if ( *r2 != -SNUMBER ) break;
        if ( r2[1] == 0 ) break;
        r2 += 2;
    }
    if ( r2 < t2stop ) { t2 = t2 + t2[1]; goto retry2; }
#else
    t1stop = t1 + t1[1];
    r1 = t1 + FUNHEAD;
    while ( r1 < t1stop ) {
        if ( *r1 == -SNUMBER ) {
            if ( r1[1] == 0 ) break;
            r1 += 2; continue;
        }
        else if ( *r1 == -SYMBOL ) {
            if ( ( symbols[r1[1]].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
```

```
                        break;
            r1 += 2; continue;
        }
        if ( *r1 > 0 && *r1 == r1[ARGHEAD]+ARGHEAD ) {
            if ( ABS(r1[r1[0]-1]) == r1[0]-ARGHEAD-1 ) {}
            else if ( r1[ARGHEAD+1] == SYMBOL ) {
                rr1 = r1 + ARGHEAD + 3;
                i1 = rr1[-1]-2;
                while ( i1 > 0 ) {
                    if ( ( symbols[*rr1].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
                        break;
                    i1 -= 2; rr1 += 2;
                }
                if ( i1 > 0 ) break;
            }
            else break;
            rr1 = r1+*r1-1;
            i1 = (ABS(*rr1)-1)/2;
            while ( i1 > 1 ) {
                if ( rr1[-1] ) break;
                i1--; rr1--;
            }
            if ( i1 > 1 || rr1[-1] != 1 ) break;
            r1 += *r1;
        }
        else break;
    }
    if ( r1 < t1stop ) { t1 = t2; goto retry1; }
    t2stop = t2 + t2[1];
    r2 = t2 + FUNHEAD;

    while ( r2 < t2stop ) {
        if ( *r2 == -SNUMBER ) {
            if ( r2[1] == 0 ) break;
            r2 += 2; continue;
        }
        else if ( *r2 == -SYMBOL ) {
            if ( ( symbols[r2[1]].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
                        break;
            r2 += 2; continue;
        }
        if ( *r2 > 0 && *r2 == r2[ARGHEAD]+ARGHEAD ) {
            if ( ABS(r2[r2[0]-1]) == r2[0]-ARGHEAD-1 ) {}
            else if ( r2[ARGHEAD+1] == SYMBOL ) {
                rr2 = r2 + ARGHEAD + 3;
                i2 = rr2[-1]-2;
                while ( i2 > 0 ) {
                    if ( ( symbols[*rr2].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
                        break;
                    i2 -= 2; rr2 += 2;
                }
                if ( i2 > 0 ) break;
            }
            else break;
            rr2 = r2+*r2-1;
            i2 = (ABS(*rr2)-1)/2;
            while ( i2 > 1 ) {
                if ( rr2[-1] ) break;
                i2--; rr2--;
            }
            if ( i2 > 1 || rr2[-1] != 1 ) break;
            r2 += *r2;
        }
        else break;
    }
    if ( r2 < t2stop ) { t2 = t2 + t2[1]; goto retry2; }
#endif
/*
    OK, now we got two objects that can be used.
*/
    *AN.RepPoint = 1;

    SHback = AN.SHvar;
    SH->finishuf = &FinishStuffle;
    SH->do_uffle = &DoStuffle;
    SH->outterm = AT.WorkPointer;
    AT.WorkPointer += *term;
    SH->ststop1 = t1 + t1[1];
    SH->ststop2 = t2 + t2[1];
    SH->thefunction = n;
    SH->option = option;
    SH->level = level;
    SH->incoef = tstop;
    SH->nincoef = ncoef;
    if ( AN.SHcombi == 0 || AN.SHcombisize == 0 ) {
        AN.SHcombisize = 200;
        AN.SHcombi = (UWORD *)Malloc1(AN.SHcombisize*sizeof(UWORD),"AN.SHcombi");
```

```
        SH->combilast = 0;
        SHback.combilast = 0;
    }
    else {
        SH->combilast += AN.SHcombi[SH->combilast]+1;
        if ( SH->combilast >= AN.SHcombisize - 100 ) {
            newcombi = (UWORD *)Malloc1(2*AN.SHcombisize*sizeof(UWORD),"AN.SHcombi");
            for ( k = 0; k < AN.SHcombisize; k++ ) newcombi[k] = AN.SHcombi[k];
            M_free(AN.SHcombi,"AN.SHcombi");
            AN.SHcombi = newcombi;
            AN.SHcombisize *= 2;
        }
    }
    AN.SHcombi[SH->combilast] = 1;
    AN.SHcombi[SH->combilast+1] = 1;

    i = t1-term; to = SH->outterm; from = term;
    NCOPY(to,from,i)
    SH->outfun = to;
    for ( i = 0; i < FUNHEAD; i++ ) { *to++ = t1[i]; }

    error = Stuffle(BHEAD t1+FUNHEAD,t2+FUNHEAD,to);

    AT.WorkPointer = SH->outterm;
    AN.SHvar = SHback;
    if ( error ) {
        MesCall("DoStuffle");
        return(-1);
    }
    return(0);
}

/*
    #] DoStuffle :
    #[ Stuffle :

    The way to generate the stuffles
    1: select an argument in the first  list (for(j1=0;j1<last;j1++))
    2: select an argument in the second list (for(j2=0;j2<last;j2++))
    3: put values for SH->ststop1 and SH->ststop2 at these arguments.
    4: generate all shuffles of the arguments in front.
    5: Then put the stuffle sum of arg(j1) and arg(j2)
    6: Then continue calling Stuffle
    7: Once one gets exhausted, we can clean up the list and call FinishShuffle
    8: if ( ( SH->option & 2 ) != 0 ) the stuffle sum is negative.
*/

int Stuffle(PHEAD WORD *from1, WORD *from2, WORD *to)
{
    GETBIDENTITY
    WORD *t, *tf, *next1, *next2, *st1, *st2, *save1, *save2;
    SHvariables *SH = &(AN.SHvar);
    int i, retval;
/*
    First the special cases (exhausted list(s)):
*/
    save1 = SH->stop1; save2 = SH->stop2;
    if ( from1 >= SH->ststop1 && from2 == SH->ststop2 ) {
        SH->stop1 = SH->ststop1;
        SH->stop2 = SH->ststop2;
        retval = FinishShuffle(BHEAD to);
        SH->stop1 = save1; SH->stop2 = save2;
        return(retval);
    }
    else if ( from1 >= SH->ststop1 ) {
        i = SH->ststop2 - from2; t = to; tf = from2; NCOPY(t,tf,i)
        SH->stop1 = SH->ststop1;
        SH->stop2 = SH->ststop2;
        retval = FinishShuffle(BHEAD t);
        SH->stop1 = save1; SH->stop2 = save2;
        return(retval);
    }
    else if ( from2 >= SH->ststop2 ) {
        i = SH->ststop1 - from1; t = to; tf = from1; NCOPY(t,tf,i)
        SH->stop1 = SH->ststop1;
        SH->stop2 = SH->ststop2;
        retval = FinishShuffle(BHEAD t);
        SH->stop1 = save1; SH->stop2 = save2;
        return(retval);
    }
/*
    Now the case that we have no stuffle sums.
*/
    SH->stop1 = SH->ststop1;
    SH->stop2 = SH->ststop2;
    SH->finishuf = &FinishShuffle;
    if ( Shuffle(BHEAD from1,from2,to) ) goto stuffcall;
```

```
    SH->finishuf = &FinishStuffle;
/*
    Now we have to select a pair, one from 1 and one from 2.
*/
#ifndef NEWCODE
    st1 = from1; next1 = st1+2;       /* <----- */
#else
    st1 = next1 = from1;
    NEXTARG(next1)
#endif
    while ( next1 <= SH->ststop1 ) {
#ifndef NEWCODE
        st2 = from2; next2 = st2+2;       /* <----- */
#else
        next2 = st2 = from2;
        NEXTARG(next2)
#endif
        while ( next2 <= SH->ststop2 ) {
            SH->stop1 = st1;
            SH->stop2 = st2;
            if ( st1 == from1 && st2 == from2 ) {
                t = to;
#ifndef NEWCODE
                *t++ = -SNUMBER; *t++ = StuffAdd(st1[1],st2[1]);
#else
                t = StuffRootAdd(st1,st2,t);
#endif
                SH->option ^= 256;
                if ( Stuffle(BHEAD next1,next2,t) ) goto stuffcall;
                SH->option ^= 256;
            }
            else if ( st1 == from1 ) {
                i = st2-from2;
                t = to; tf = from2; NCOPY(t,tf,i)
#ifndef NEWCODE
                *t++ = -SNUMBER; *t++ = StuffAdd(st1[1],st2[1]);
#else
                t = StuffRootAdd(st1,st2,t);
#endif
                SH->option ^= 256;
                if ( Stuffle(BHEAD next1,next2,t) ) goto stuffcall;
                SH->option ^= 256;
            }
            else if ( st2 == from2 ) {
                i = st1-from1;
                t = to; tf = from1; NCOPY(t,tf,i)
#ifndef NEWCODE
                *t++ = -SNUMBER; *t++ = StuffAdd(st1[1],st2[1]);
#else
                t = StuffRootAdd(st1,st2,t);
#endif
                SH->option ^= 256;
                if ( Stuffle(BHEAD next1,next2,t) ) goto stuffcall;
                SH->option ^= 256;
            }
            else {
                if ( Shuffle(BHEAD from1,from2,to) ) goto stuffcall;
            }
#ifndef NEWCODE
            st2 = next2; next2 += 2;       /* <----- */
#else
            st2 = next2;
            NEXTARG(next2)
#endif
        }
#ifndef NEWCODE
        st1 = next1; next1 += 2;       /* <----- */
#else
        st1 = next1;
        NEXTARG(next1)
#endif
    }
    SH->stop1 = save1; SH->stop2 = save2;
    return(0);
stuffcall:;
    MesCall("Stuffle");
    return(-1);
}

/*
    #] Stuffle :
    #[ FinishStuffle :

    The program only comes here from the Shuffle routine.
    It should add the stuffle sum and then call Stuffle again.
*/

```

```
int FinishStuffle(PHEAD WORD *fini)
{
    GETBIDENTITY
    SHvariables *SH = &(AN.SHvar);
#ifdef NEWCODE
    WORD *next1 = SH->stop1, *next2 = SH->stop2;
    fini = StuffRootAdd(next1,next2,fini);
#else
    *fini++ = -SNUMBER; *fini++ = StuffAdd(SH->stop1[1],SH->stop2[1]);
#endif
    SH->option ^= 256;
#ifdef NEWCODE
    NEXTARG(next1)
    NEXTARG(next2)
    if ( Stuffle(BHEAD next1,next2,fini) ) goto stuffcall;
#else
    if ( Stuffle(BHEAD SH->stop1+2,SH->stop2+2,fini) ) goto stuffcall;
#endif
    SH->option ^= 256;
    return(0);
stuffcall:;
    MesCall("FinishStuffle");
    return(-1);
}

/*
    #] FinishStuffle :
    #[ StuffRootAdd :

    Makes the stuffle sum of two arguments.
    The arguments can be of one of three types:
    1: -SNUMBER,num
    2: -SYMBOL,symbol
    3: Numerical (long) argument.
    4: Generic argument with (only) symbols that are roots of unity and
       a coefficient.
    We have excluded the case that both t1 and t2 are of type 1:
    The output should be written to 'to' and the new fill position should
    be the return value.
    `to' is inside the workspace.

    The stuffle sum is sig_(t2)*t1+sig_(t1)*t2
    or sig_(t1)*sig_(t2)*(abs_(t1)+abs_(t2))
*/

#ifdef NEWCODE

WORD *StuffRootAdd(WORD *t1, WORD *t2, WORD *to)
{
    int type1, type2, type3, sgn, sgn1, sgn2, sgn3, pow, root, nosymbols, i;
    WORD *tt1, *tt2, it1, it2, *t3, *r, size1, size2, size3;
    WORD scratch[2];
    LONG x;
    if ( *t1 == -SNUMBER ) { type1 = 1; if ( t1[1] < 0 ) sgn1 = -1; else sgn1 = 1; }
    else if ( *t1 == -SYMBOL ) { type1 = 2; sgn1 = 1; }
    else if ( ABS(t1[*t1-1]) == *t1-ARGHEAD-1 ) {
        type1 = 3; if ( t1[*t1-1] < 0 ) sgn1 = -1; else sgn1 = 1; }
    else { type1 = 4; if ( t1[*t1-1] < 0 ) sgn1 = -1; else sgn1 = 1; }
    if ( *t2 == -SNUMBER ) { type2 = 1; if ( t2[1] < 0 ) sgn2 = -1; else sgn2 = 1; }
    else if ( *t2 == -SYMBOL ) { type2 = 2; sgn2 = 1; }
    else if ( ABS(t2[*t2-1]) == *t2-ARGHEAD-1 ) {
        type2 = 3; if ( t2[*t2-1] < 0 ) sgn2 = -1; else sgn2 = 1; }
    else { type2 = 4; if ( t2[*t2-1] < 0 ) sgn2 = -1; else sgn2 = 1; }
    if ( type1 > type2 ) {
        t3 = t1; t1 = t2; t2 = t3;
        type3 = type1; type1 = type2; type2 = type3;
        sgn3 = sgn1; sgn1 = sgn2; sgn2 = sgn3;
    }
    nosymbols = 1; sgn3 = 1;
    switch ( type1 ) {
        case 1:
            if ( type2 == 1 ) {
                x  = sgn2 * t1[1];
                x += sgn1 * t2[1];
                if ( x > MAXPOSITIVE || x < -(MAXPOSITIVE+1) ) {
                    if ( x < 0 ) { sgn1 = -3; x = -x; }
                    else sgn1 = 3;
                    *to++ = ARGHEAD+4;
                    *to++ = 0;
                    FILLARG(to)
                    *to++ = 4; *to++ = (UWORD)x; *to++ = 1; *to++ = sgn1;
                }
                else { *to++ = -SNUMBER; *to++ = (WORD)x; }
            }
            else if ( type2 == 2 ) {
                *to++ = ARGHEAD+8; *to++ = 0; FILLARG(to)
                *to++ = 8; *to++ = SYMBOL; *to++ = 4; *to++ = t2[1]; *to++ = 1;
```

```
                *to++ = ABS(t1[1])+1;
                *to++ = 1;
                *to++ = 3*sgn1;
            }
            else if ( type2 == 3 ) {
                tt1 = (WORD *)scratch; tt1[0] = ABS(t1[1]); size1 = 1;
                tt2 = t2+ARGHEAD+1; size2 = (ABS(t2[*t2-1])-1)/2;
                t3 = to;
                *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
                goto DoCoeffi;
            }
            else {
/*
                t1 is (short) numeric, t2 has the symbol(s).
*/
                tt1 = (WORD *)scratch; tt1[0] = ABS(t1[1]); size1 = 1;
                tt2 = t2+ARGHEAD+1; tt2 += tt2[1]; size2 = (ABS(t2[*t2-1])-1)/2;
                t3 = to; i = tt2 - t2; r = t2;
                NCOPY(to,r,i)
                nosymbols = 0;
                goto DoCoeffi;
            }
        break;
        case 2:
            if ( type2 == 2 ) {
                if ( t1[1] == t2[1] ) {
                    if ( ( symbols[t1[1]].maxpower == 4 )
                    && ( ( symbols[t1[1]].complex & VARTYPEMINUS ) == VARTYPEMINUS ) ) {
                        *to++ = -SNUMBER; *to++ = -2;
                    }
                    else if ( symbols[t1[1]].maxpower == 2 ) {
                        *to++ = -SNUMBER; *to++ = 2;
                    }
                    else {
                        *to++ = ARGHEAD+8; *to++ = 0; FILLARG(to)
                        *to++ = 8; *to++ = SYMBOL; *to++ = 4;
                        *to++ = t1[1]; *to++ = 2;
                        *to++ = 2; *to++ = 1; *to++ = 3;
                    }
                }
                else {
                    *to++ = ARGHEAD+10; *to++ = 0; FILLARG(to)
                    *to++ = 10; *to++ = SYMBOL; *to++ = 6;
                    if ( t1[1] < t2[1] ) {
                        *to++ = t1[1]; *to++ = 1; *to++ = t2[1]; *to++ = 1;
                    }
                    else {
                        *to++ = t2[1]; *to++ = 1; *to++ = t1[1]; *to++ = 1;
                    }
                    *to++ = 2; *to++ = 1; *to++ = 3;
                }
            }
            else if ( type2 == 3 ) {
                t3 = to;
                *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
                *to++ = SYMBOL; *to++ = 4; *to++ = t1[1]; *to++ = 1;
                tt1 = scratch; tt1[1] = 1; size1 = 1;
                tt2 = t2+ARGHEAD+1; size2 = (ABS(t2[*t2-1])-1)/2;
                nosymbols = 0;
                goto DoCoeffi;
            }
            else {
                tt1 = scratch; tt1[0] = 1; size1 = 1;
                t3 = to;
                *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
                *to++ = SYMBOL; *to++ = 0;
                tt2 = t2 + ARGHEAD+3; it2 = tt2[-1]-2;
                while ( it2 > 0 ) {
                    if ( *tt2 == t1[1] ) {
                        pow = tt2[1]+1;
                        root = symbols[*tt2].maxpower;
                        if ( pow >= root ) pow -= root;
                        if ( ( symbols[*tt2].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
                            if ( ( root & 1 ) == 0 && pow >= root/2 ) {
                                pow -= root/2; sgn3 = -sgn3;
                            }
                        }
                        if ( pow != 0 ) {
                            *to++ = *tt2; *to++ = pow;
                        }
                        tt2 += 2; it2 -= 2;
                        break;
                    }
                    else if ( t1[1] < *tt2 ) {
                        *to++ = t1[1]; *to++ = 1; break;
                    }
                    else {
```

```
                        *to++ = *tt2++; *to++ = *tt2++; it2 -= 2;
                        if ( it2 <= 0 ) { *to++ = t1[1]; *to++ = 1; }
                    }
                }
                while ( it2 > 0 ) { *to++ = *tt2++; *to++ = *tt2++; it2 -= 2; }
                if ( (to - t3) > ARGHEAD+3 ) {
                    t3[ARGHEAD+2] = (to-t3)-ARGHEAD-1; /* size of the SYMBOL field */
                    nosymbols = 0;
                }
                else {
                    to = t3+ARGHEAD+1; /* no SYMBOL field */
                }
                size2 = (ABS(t2[*t2-1])-1)/2;
                goto DoCoeffi;
            }
        break;
        case 3:
            if ( type2 == 3 ) {
/*
                Both are numeric
*/
                tt1 = t1+ARGHEAD+1; size1 = (ABS(t1[*t1-1])-1)/2;
                tt2 = t2+ARGHEAD+1; size2 = (ABS(t2[*t2-1])-1)/2;
                t3 = to;
                *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
                goto DoCoeffi;
            }
            else {
/*
                t1 is (long) numeric, t2 has the symbol(s).
*/
                tt1 = t1+ARGHEAD+1; size1 = (ABS(t1[*t1-1])-1)/2;
                tt2 = t2+ARGHEAD+1; tt2 += tt2[1]; size2 = (ABS(t2[*t2-1])-1)/2;
                t3 = to; i = tt2 - t2; r = t2;
                NCOPY(to,r,i)
                nosymbols = 0;
                goto DoCoeffi;
            }
        break;
        case 4:
/*
            Both have roots of unity
            1: Merge the lists and simplify if possible
*/
            tt1 = t1+ARGHEAD+3; it1 = tt1[-1]-2;
            tt2 = t2+ARGHEAD+3; it2 = tt2[-1]-2;
            t3 = to;
            *to++ = 0; *to++ = 0; FILLARG(to)
            *to++ = 0; *to++ = SYMBOL; *to++ = 0;
            while ( it1 > 0 && it2 > 0 ) {
                if ( *tt1 == *tt2 ) {
                    pow = tt1[1]+tt2[1];
                    root = symbols[*tt1].maxpower;
                    if ( pow >= root ) pow -= root;
                    if ( ( symbols[*tt1].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
                        if ( ( root & 1 ) == 0 && pow >= root/2 ) {
                            pow -= root/2; sgn3 = -sgn3;
                        }
                    }
                    if ( pow != 0 ) {
                        *to++ = *tt1; *to++ = pow;
                    }
                    tt1 += 2; tt2 += 2; it1 -= 2; it2 -= 2;
                }
                else if ( *tt1 < *tt2 ) {
                    *to++ = *tt1++; *to++ = *tt1++; it1 -= 2;
                }
                else {
                    *to++ = *tt2++; *to++ = *tt2++; it2 -= 2;
                }
            }
            while ( it1 > 0 ) { *to++ = *tt1++; *to++ = *tt1++; it1 -= 2; }
            while ( it2 > 0 ) { *to++ = *tt2++; *to++ = *tt2++; it2 -= 2; }
            if ( (to - t3) > ARGHEAD+3 ) {
                t3[ARGHEAD+2] = (to-t3)-ARGHEAD-1; /* size of the SYMBOL field */
                nosymbols = 0;
            }
            else {
                to = t3+ARGHEAD+1; /* no SYMBOL field */
            }
            size1 = (ABS(t1[*t1-1])-1)/2;
            size2 = (ABS(t2[*t2-1])-1)/2;
/*
            Now tt1 and tt2 are pointing at their coefficients.
            sgn1 is the sign of 1, sgn2 is the sign of 2 and sgn3 is an extra
            overall sign.
*/
```

```
DoCoeffi:
            if ( AddLong((UWORD *)tt1,size1,(UWORD *)tt2,size2,(UWORD *)to,&size3) ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Called from StuffRootAdd");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            sgn = sgn1*sgn2*sgn3;
            if ( nosymbols && size3 == 1 ) {
                if ( (UWORD)(to[0]) <= MAXPOSITIVE && sgn > 0 ) {
                    sgn1 = to[0];
                    to = t3; *to++ = -SNUMBER; *to++ = sgn1;
                }
                else if ( (UWORD)(to[0]) <= (MAXPOSITIVE+1) && sgn < 0 ) {
                    sgn1 = to[0];
                    to = t3; *to++ = -SNUMBER; *to++ = -sgn1;
                }
                else goto genericcoef;
            }
            else {
genericcoef:
                to += size3;
                sgn = sgn*(2*size3+1);
                *to++ = 1;
                while ( size3 > 1 ) { *to++ = 0; size3--; }
                *to++ = sgn;
                t3[0] = to - t3;
                t3[ARGHEAD] = t3[0] - ARGHEAD;
            }
        break;
    }
    return(to);
}

#endif

/*
    #] StuffRootAdd :
*/
```

4.94 sch.c File Reference

`#include "form3.h"`

Include dependency graph for sch.c:

Macros

- #define va_dcl int va_alist;
- #define va_start(list) list = (UBYTE ∗) &va_alist
- #define va_end(list)
- #define va_arg(list, mode) (((mode ∗)(list += sizeof(mode)))[-1])

Typedefs

- typedef UBYTE ∗ va_list

Functions

- UBYTE * StrCopy (UBYTE *from, UBYTE *to)
- VOID AddToLine (UBYTE *s)
- VOID FiniLine ()
- VOID IniLine (WORD extrablank)
- VOID LongToLine (UWORD *a, WORD na)
- VOID RatToLine (UWORD *a, WORD na)
- VOID TalToLine (UWORD x)
- VOID TokenToLine (UBYTE *s)
- UBYTE * CodeToLine (WORD number, UBYTE *Out)
- void MultiplyToLine ()
- UBYTE * AddArrayIndex (WORD num, UBYTE *out)
- VOID PrtTerms ()
- UBYTE * WrtPower (UBYTE *Out, WORD Power)
- void PrintTime ()
- VOID WriteLists ()
- void WriteDictionary (DICTIONARY *dict)
- VOID WriteArgument (WORD *t)
- WORD WriteSubTerm (WORD *sterm, WORD first)
- WORD WriteInnerTerm (WORD *term, WORD first)
- WORD WriteTerm (WORD *term, WORD *lbrac, WORD first, WORD prtf, WORD br)
- WORD WriteExpression (WORD *terms, LONG ltot)
- WORD WriteAll ()
- WORD WriteOne (UBYTE *name, int alreadyinline, int nosemi)

4.94.1 Detailed Description

Contains the functions that deal with the writing of expressions/terms in a textual representation. (Dutch schrijven = to write)

Definition in file sch.c.

4.94.2 Macro Definition Documentation

4.94.2.1 va_dcl

```
#define va_dcl int va_alist;
```

Definition at line 48 of file sch.c.

4.94.2.2 va_start

```
#define va_start(
              list ) list = (UBYTE *) &va_alist
```

Definition at line 49 of file sch.c.

4.94.2.3 va_end

```
#define va_end(
            list )
```

Definition at line 50 of file sch.c.

4.94.2.4 va_arg

```
#define va_arg(
            list,
            mode ) (((mode *)(list += sizeof(mode)))[-1])
```

Definition at line 51 of file sch.c.

4.94.3 Typedef Documentation

4.94.3.1 va_list

```
typedef UBYTE* va_list
```

Definition at line 47 of file sch.c.

4.94.4 Function Documentation

4.94.4.1 StrCopy()

```
UBYTE * StrCopy (
            UBYTE * from,
            UBYTE * to )
```

Definition at line 67 of file sch.c.

4.94.4.2 AddToLine()

```
VOID AddToLine (
            UBYTE * s )
```

Definition at line 82 of file sch.c.

4.94.4.3 FiniLine()

```
VOID FiniLine ( )
```

Definition at line 182 of file sch.c.

4.94.4.4 IniLine()

```
VOID IniLine (
            WORD extrablank )
```

Definition at line 267 of file sch.c.

4.94.4.5 LongToLine()

```
VOID LongToLine (
            UWORD * a,
            WORD na )
```

Definition at line 303 of file sch.c.

4.94.4.6 RatToLine()

```
VOID RatToLine (
            UWORD * a,
            WORD na )
```

Definition at line 337 of file sch.c.

4.94.4.7 TalToLine()

```
VOID TalToLine (
            UWORD x )
```

Definition at line 533 of file sch.c.

4.94.4.8 TokenToLine()

```
VOID TokenToLine (
            UBYTE * s )
```

Definition at line 562 of file sch.c.

4.94.4.9 CodeToLine()

```
UBYTE * CodeToLine (
            WORD number,
            UBYTE * Out )
```

Definition at line 669 of file sch.c.

4.94.4.10 MultiplyToLine()

```
void MultiplyToLine ( )
```

Definition at line 682 of file sch.c.

4.94.4.11 AddArrayIndex()

```
UBYTE * AddArrayIndex (
            WORD num,
            UBYTE * out )
```

Definition at line 707 of file sch.c.

4.94.4.12 PrtTerms()

```
VOID PrtTerms ( )
```

Definition at line 727 of file sch.c.

4.94.4.13 WrtPower()

```
UBYTE * WrtPower (
            UBYTE * Out,
            WORD Power )
```

Definition at line 749 of file sch.c.

4.94.4.14 PrintTime()

```
void PrintTime ( )
```

Definition at line 787 of file sch.c.

4.94.4.15 WriteLists()

```
VOID WriteLists ( )
```

Definition at line 813 of file sch.c.

4.94.4.16 WriteDictionary()

```
void WriteDictionary (
            DICTIONARY * dict )
```

Definition at line 1278 of file sch.c.

4.94.4.17 WriteArgument()

```
VOID WriteArgument (
            WORD * t )
```

Definition at line 1395 of file sch.c.

4.94.4.18 WriteSubTerm()

```
WORD WriteSubTerm (
            WORD * sterm,
            WORD first )
```

Definition at line 1510 of file sch.c.

4.94.4.19 WriteInnerTerm()

```
WORD WriteInnerTerm (
            WORD * term,
            WORD first )
```

Definition at line 1903 of file sch.c.

4.94.4.20 WriteTerm()

```
WORD WriteTerm (
            WORD * term,
            WORD * lbrac,
            WORD first,
            WORD prtf,
            WORD br )
```

Definition at line 2061 of file sch.c.

4.94.4.21 WriteExpression()

```
WORD WriteExpression (
            WORD * terms,
            LONG ltot )
```

Definition at line 2383 of file sch.c.

4.94.4.22 WriteAll()

```
WORD WriteAll ( )
```

Definition at line 2414 of file sch.c.

4.94.4.23 WriteOne()

```
WORD WriteOne (
            UBYTE * name,
            int alreadyinline,
            int nosemi )
```

Definition at line 2634 of file sch.c.

4.95 sch.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ Includes : sch.c
00034 */
00035
00036 #include "form3.h"
00037
00038 #ifdef ANSI
00039 #include <stdarg.h>
00040 #else
00041 #ifdef mBSD
00042 #include <varargs.h>
00043 #else
00044 #ifdef VMS
00045 #include <varargs.h>
00046 #else
00047 typedef UBYTE *va_list;
00048 #define va_dcl int va_alist;
00049 #define va_start(list) list = (UBYTE *) &va_alist
00050 #define va_end(list)
00051 #define va_arg(list,mode) (((mode *)(list += sizeof(mode)))[-1])
00052 #endif
00053 #endif
00054 #endif
00055
00056 static int startinline = 0;
00057 static char fcontchar = '&';
00058 static int noextralinefeed = 0;
00059 static int lowestlevel = 1;
00060
00061 /*
00062     #] Includes :
00063     #[ schryf-Utilities :
00064         #[ StrCopy :            UBYTE *StrCopy(from,to)
00065 */
00066
00067 UBYTE *StrCopy(UBYTE *from, UBYTE *to)
00068 {
00069     while( ( *to++ = *from++ ) != 0 );
```

```
    return(to-1);
}

/*
        #] StrCopy :
        #[ AddToLine :          VOID AddToLine(s)

    Puts the characters of s in the outputline. If the line becomes
    filled it is written.

*/

VOID AddToLine(UBYTE *s)
{
    UBYTE *Out;
    LONG num;
    int i;
    if ( AO.OutInBuffer ) { AddToDollarBuffer(s); return; }
    Out = AO.OutFill;
    while ( *s ) {
        if ( Out >= AO.OutStop ) {
            if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
                *Out++ = fcontchar;
            }
#ifdef WITHRETURN
            *Out++ = CARRIAGERETURN;
#endif
            *Out++ = LINEFEED;
            AO.FortFirst = 0;
            num = Out - AO.OutputLine;

            if ( AC.LogHandle >= 0 ) {
                if ( WriteFile(AC.LogHandle,AO.OutputLine+startinline
                                ,num-startinline) != (num-startinline) ) {
/*
                    We cannot write to an otherwise open log file.
                    The disk could be full of course.
*/
#ifdef DEBUGGER
                    if ( BUG.logfileflag == 0 ) {
                        fprintf(stderr,"Panic: Cannot write to log file! Disk full?\n");
                        BUG.logfileflag = 1;
                    }
                    BUG.eflag = 1; BUG.printflag = 1;
#else
                    Terminate(-1);
#endif
                }
            }

            if ( ( AO.PrintType & PRINTLFILE ) == 0 ) {
#ifdef WITHRETURN
                if ( num > 1 && AO.OutputLine[num-2] == CARRIAGERETURN ) {
                    AO.OutputLine[num-2] = LINEFEED;
                    num--;
                }
#endif
                if ( WriteFile(AM.StdOut,AO.OutputLine+startinline
                        ,num-startinline) != (num-startinline) ) {
#ifdef DEBUGGER
                    if ( BUG.stdoutflag == 0 ) {
                        fprintf(stderr,"Panic: Cannot write to standard output!\n");
                        BUG.stdoutflag = 1;
                    }
                    BUG.eflag = 1; BUG.printflag = 1;
#else
                    Terminate(-1);
#endif
                }
            }
            /* thomasr 23/04/09: A continuation line has been started.
             * In Fortran90 we do not want a space after the initial
             * '&' character otherwise we might end up with something
             * like:
             *    ...  2.&
             *  & 0 ...
             */
            startinline = 0;
            for ( i = 0; i < AO.OutSkip; i++ ) AO.OutputLine[i] = ' ';
            Out = AO.OutputLine + AO.OutSkip;
            if ( ( AC.OutputMode == FORTRANMODE
             || AC.OutputMode == PFORTRANMODE ) && AO.OutSkip == 7 ) {
                /* thomasr 23/04/09: fix leading blank in fortran90 mode */
                if(AC.IsFortran90 == ISFORTRAN90) {
                    Out[-1] = fcontchar;
                }
                else {
```

```
                    Out[-2] = fcontchar;
                    Out[-1] = ' ';
                }
            }
            if ( AO.IsBracket ) { *Out++ = ' ';
                if ( AC.OutputSpaces == NORMALFORMAT ) {
                     *Out++ = ' '; *Out++ = ' '; }
            }
            *Out = '\0';
            if ( AC.OutputMode == FORTRANMODE
             || ( AC.OutputMode == CMODE && AO.FactorMode == 0 )
             || AC.OutputMode == PFORTRANMODE )
                AO.InFbrack++;
        }
        *Out++ = *s++;
    }
    *Out = '\0';
    AO.OutFill = Out;
}

/*
        #] AddToLine :
        #[ FiniLine :           VOID FiniLine()
*/

VOID FiniLine()
{
    UBYTE *Out;
    WORD i;
    LONG num;
    if ( AO.OutInBuffer ) return;
    Out = AO.OutFill;
    while ( Out > AO.OutputLine ) {
        if ( Out[-1] == ' ' ) Out--;
        else break;
    }
    i = (WORD)(Out-AO.OutputLine);
    if ( noextralinefeed == 0 ) {
        if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90
            && Out > AO.OutputLine ) {
/*
            *Out++ = fcontchar;
*/
        }
#ifdef WITHRETURN
        *Out++ = CARRIAGERETURN;
#endif
        *Out++ = LINEFEED;
        AO.FortFirst = 0;
    }
    num = Out - AO.OutputLine;

    if ( AC.LogHandle >= 0 ) {
        if ( WriteFile(AC.LogHandle,AO.OutputLine+startinline
                ,num-startinline) != (num-startinline) ) {
#ifdef DEBUGGER
            if ( BUG.logfileflag == 0 ) {
                fprintf(stderr,"Panic: Cannot write to log file! Disk full?\n");
                BUG.logfileflag = 1;
            }
            BUG.eflag = 1; BUG.printflag = 1;
#else
            Terminate(-1);
#endif
        }
    }

    if ( ( AO.PrintType & PRINTLFILE ) == 0 ) {
#ifdef WITHRETURN
        if ( num > 1 && AO.OutputLine[num-2] == CARRIAGERETURN ) {
            AO.OutputLine[num-2] = LINEFEED;
            num--;
        }
#endif
        if ( WriteFile(AM.StdOut,AO.OutputLine+startinline,
                num-startinline) != (num-startinline) ) {
#ifdef DEBUGGER
            if ( BUG.stdoutflag == 0 ) {
                fprintf(stderr,"Panic: Cannot write to standard output!\n");
                BUG.stdoutflag = 1;
            }
            BUG.eflag = 1; BUG.printflag = 1;
#else
            Terminate(-1);
#endif
        }
    }
```

```
    startinline = 0;
    if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
         || ( AC.OutputMode == CMODE && AO.FactorMode == 0 ) ) AO.InFbrack++;
    Out = AO.OutputLine;
    AO.OutStop = Out + AC.LineLength;
    i = AO.OutSkip;
    while ( --i >= 0 ) *Out++ = ' ';
    if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
     && AO.OutSkip == 7 ) {
        Out[-2] = fcontchar;
        Out[-1] = ' ';
    }
    AO.OutFill = Out;
}

/*
        #] FiniLine :
        #[ IniLine :            VOID IniLine(extrablank)

    Initializes the output line for the type of output

*/

VOID IniLine(WORD extrablank)
{
    UBYTE *Out;
    Out = AO.OutputLine;
    AO.OutStop = Out + AC.LineLength;
    *Out++ = ' ';
    *Out++ = ' ';
    *Out++ = ' ';
    *Out++ = ' ';
    *Out++ = ' ';
    if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
        *Out++ = fcontchar;
        AO.OutSkip = 7;
    }
    else
        AO.OutSkip = 6;
    *Out++ = ' ';
    while ( extrablank > 0 ) {
        *Out++ = ' ';
        extrablank--;
    }
    AO.OutFill = Out;
}

/*
        #] IniLine :
        #[ LongToLine :         VOID LongToLine(a,na)

    Puts a Long integer in the output line. If it is only a single
    word long it is put in the line as a single token.
    The sign of a is ignored.

*/

static UBYTE *LLscratch = 0;

VOID LongToLine(UWORD *a, WORD na)
{
    UBYTE *OutScratch;
    if ( LLscratch == 0 ) {
        LLscratch = (UBYTE *)Malloc1(4*(AM.MaxTal*sizeof(WORD)+2)*sizeof(UBYTE),"LongToLine");
    }
    OutScratch = LLscratch;
    if ( na < 0 ) na = -na;
    if ( na > 1 ) {
        PrtLong(a,na,OutScratch);
        if ( AO.NoSpacesInNumbers || AC.OutputMode == REDUCEMODE ) {
            AO.BlockSpaces = 1;
            TokenToLine(OutScratch);
            AO.BlockSpaces = 0;
        }
        else {
            TokenToLine(OutScratch);
        }
    }
    else if ( !na ) TokenToLine((UBYTE *)"0");
    else TalToLine(*a);
}

/*
        #] LongToLine :
        #[ RatToLine :          VOID RatToLine(a,na)

    Puts a rational number in the output line. The sign is ignored.
```

```c

*/

static UBYTE *RLscratch = 0;
static UWORD *RLscratE = 0;

VOID RatToLine(UWORD *a, WORD na)
{
    GETIDENTITY
    WORD adenom, anumer;
    if ( na < 0 ) na = -na;
    if ( AC.OutNumberType == RATIONALMODE ) {
/*
        We need some special provisions for the various Fortran modes.
        In PFORTRAN we use
                one     if denom = numerator = 1
                integer     if denom = 1
                (one/integer) if numerator = 1
                ((one*integer)/integer) in the general case
*/
        if ( AC.OutputMode == PFORTRANMODE ) {
          UnPack(a,na,&adenom,&anumer);
          if ( na == 1 && a[0] == 1 && a[1] == 1 ) {
            AddToLine((UBYTE *)"one");
            return;
          }
          if ( adenom == 1 && a[na] == 1 ) {
            LongToLine(a,anumer);
            if ( anumer > 1 ) {
                if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *)".Q0"); }
                else { AddToLine((UBYTE *)".D0"); }
            }
          }
          else if ( anumer == 1 && a[0] == 1 ) {
            a += na;
            AddToLine((UBYTE *)"(one/");
            LongToLine(a,adenom);
            if ( adenom > 1 ) {
                if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *)".Q0"); }
                else { AddToLine((UBYTE *)".D0"); }
            }
            AddToLine((UBYTE *)")");
          }
          else {
            if ( anumer > 1 || adenom > 1 ) {
                LongToLine(a,anumer);
                if ( anumer > 1 ) {
                    if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *)".Q0"); }
                    else { AddToLine((UBYTE *)".D0"); }
                }
                a += na;
                AddToLine((UBYTE *)"/");
                LongToLine(a,adenom);
                if ( adenom > 1 ) {
                    if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *)".Q0"); }
                    else { AddToLine((UBYTE *)".D0"); }
                }
            }
            else {
                AddToLine((UBYTE *)"((one*");
                LongToLine(a,anumer);
                a += na;
                AddToLine((UBYTE *)")/");
                LongToLine(a,adenom);
                AddToLine((UBYTE *)")");
            }
          }
        }
        else {
          UnPack(a,na,&adenom,&anumer);
          LongToLine(a,anumer);
          a += na;
          if ( anumer && !( adenom == 1 && *a == 1 ) ) {
            if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
                if ( AC.Fortran90Kind ) {
                    AddToLine(AC.Fortran90Kind);
                    AddToLine((UBYTE *)"/");
                }
                else {
                    AddToLine((UBYTE *)"./");
                }
            }
            else if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == CMODE ) {
                if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *)".Q0/"); }
                else if ( AO.DoubleFlag == 1 ) { AddToLine((UBYTE *)".D0/"); }
                else { AddToLine((UBYTE *)"./"); }
            }
```

```
            else AddToLine((UBYTE *)"/");
            LongToLine(a,adenom);
            if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
                if ( AC.Fortran90Kind ) {
                    AddToLine(AC.Fortran90Kind);
                }
                else {
                    AddToLine((UBYTE *)".");
                }
            }
            else if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == CMODE ) {
                if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *)".Q0"); }
                else if ( AO.DoubleFlag == 1 ) { AddToLine((UBYTE *)".D0"); }
                else { AddToLine((UBYTE *)"."); }
            }
          }
          else if ( anumer > 1 && ( AC.OutputMode == FORTRANMODE
          || AC.OutputMode == CMODE ) ) {
            if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
                if ( AC.Fortran90Kind ) {
                    AddToLine(AC.Fortran90Kind);
                }
                else {
                    AddToLine((UBYTE *)".");
                }
            }
            else if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *)".Q0"); }
            else if ( AO.DoubleFlag == 1 ) { AddToLine((UBYTE *)".D0"); }
            else { AddToLine((UBYTE *)"."); }
          }
          else if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
                if ( AC.Fortran90Kind ) {
                    AddToLine(AC.Fortran90Kind);
                }
                else {
                    AddToLine((UBYTE *)".");
                }
          }
          else if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == CMODE )
          && AO.DoubleFlag ) {
            if ( anumer == 1 && adenom == 1 && a[0] == 1 ) {}
            else if ( AO.DoubleFlag == 2 ) { AddToLine((UBYTE *)".Q0"); }
            else if ( AO.DoubleFlag == 1 ) { AddToLine((UBYTE *)".D0"); }
          }
        }
    }
    else {
/*
        This is the float mode
*/
        UBYTE *OutScratch;
        WORD exponent = 0, i, ndig, newl;
        UWORD *c, *den, b = 10, dig[10];
        UBYTE *o, *out, cc;
/*
        First we have to adjust the numerator and denominator
*/
        if ( RLscratch == 0 ) {
            RLscratch = (UBYTE *)Malloc1(4*(AM.MaxTal+2)*sizeof(UBYTE),"RatToLine");
            RLscratE = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"RatToLine");
        }
        out = OutScratch = RLscratch;
        c = RLscratE; for ( i = 0; i < 2*na; i++ ) c[i] = a[i];
        UnPack(c,na,&adenom,&anumer);
        while ( BigLong(c,anumer,c+na,adenom) >= 0 ) {
            Divvy(BHEAD c,&na,&b,1);
            UnPack(c,na,&adenom,&anumer);
            exponent++;
        }
        while ( BigLong(c,anumer,c+na,adenom) < 0 ) {
            Mully(BHEAD c,&na,&b,1);
            UnPack(c,na,&adenom,&anumer);
            exponent--;
        }
/*
        Now division will give a number between 1 and 9
*/
        den = c + na; i = 1;
        DivLong(c,anumer,den,adenom,dig,&ndig,c,&newl);
        *out++ = (UBYTE)(dig[0]+'0'); *out++ = '.';
        while ( newl && i < AC.OutNumberType ) {
            Pack(c,&newl,den,adenom);
            Mully(BHEAD c,&newl,&b,1);
            na = newl;
            UnPack(c,na,&adenom,&anumer);
            den = c + na;
            DivLong(c,anumer,den,adenom,dig,&ndig,c,&newl);
```

```
            if ( ndig == 0 ) *out++ = '0';
            else *out++ = (UBYTE)(dig[0]+'0');
            i++;
        }
        *out++ = 'E';
        if ( exponent < 0 ) { exponent = -exponent; *out++ = '-'; }
        else { *out++ = '+'; }
        o = out;
        do {
            *out++ = (UBYTE)((exponent % 10)+'0');
            exponent /= 10;
        } while ( exponent );
        *out = 0; out--;
        while ( o < out ) { cc = *o; *o = *out; *out = cc; o++; out--; }
        TokenToLine(OutScratch);
    }
}

/*
        #] RatToLine :
        #[ TalToLine :          VOID TalToLine(x)

    Writes the unsigned number x to the output as a single token.
    Par indicates the number of leading blanks in the line.
    This parameter is needed here for the WriteLists routine.

*/

VOID TalToLine(UWORD x)
{
    UBYTE t[BITSINWORD/3+1];
    UBYTE *s;
    WORD i = 0, j;
    s = t;
    do { *s++ = (UBYTE)((x % 10)+'0'); i++; } while ( ( x /= 10 ) != 0 );
    *s-- = '\0';
    j = ( i - 1 ) » 1;
    while ( j >= 0 ) {
        i = t[j]; t[j] = s[-j]; s[-j] = (UBYTE)i; j--;
    }
    TokenToLine(t);
}

/*
        #] TalToLine :
        #[ TokenToLine :        VOID TokenToLine(s)

    Puts s in the output buffer. If it doesn't fit the buffer is
    flushed first. This routine keeps tokens as one unit.
    Par indicates the number of leading blanks in the line.
    This parameter is needed here for the WriteLists routine.

    Remark (27-oct-2007): i and j must be longer than WORD!
    It can happen that a number is so long that it has more than 2^15 or 2^31
    digits!
*/

VOID TokenToLine(UBYTE *s)
{
    UBYTE *t, *Out;
    LONG num, i = 0, j;
    if ( AO.OutInBuffer ) { AddToDollarBuffer(s); return; }
    t = s; Out = AO.OutFill;
    while ( *t++ ) i++;
    while ( i > 0 ) {
        if ( ( Out + i ) >= AO.OutStop && ( ( i < ((AC.LineLength-AO.OutSkip)»1) )
        || ( (AO.OutStop-Out) < (i»2) ) ) ) {
            if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
                *Out++ = fcontchar;
            }
#ifdef WITHRETURN
            *Out++ = CARRIAGERETURN;
#endif
            *Out++ = LINEFEED;
            AO.FortFirst = 0;
            num = Out - AO.OutputLine;
            if ( AC.LogHandle >= 0 ) {
                if ( WriteFile(AC.LogHandle,AO.OutputLine+startinline,
                    num-startinline) != (num-startinline) ) {
#ifdef DEBUGGER
                    if ( BUG.logfileflag == 0 ) {
                        fprintf(stderr,"Panic: Cannot write to log file! Disk full?\n");
                        BUG.logfileflag = 1;
                    }
                    BUG.eflag = 1; BUG.printflag = 1;
#else
                    Terminate(-1);
```

```
#endif
                }
            }
            if ( ( AO.PrintType & PRINTLFILE ) == 0 ) {
#ifdef WITHRETURN
                if ( num > 1 && AO.OutputLine[num-2] == CARRIAGERETURN ) {
                    AO.OutputLine[num-2] = LINEFEED;
                    num--;
                }
#endif
                if ( WriteFile(AM.StdOut,AO.OutputLine+startinline,
                    num-startinline) != (num-startinline) ) {
#ifdef DEBUGGER
                    if ( BUG.stdoutflag == 0 ) {
                        fprintf(stderr,"Panic: Cannot write to standard output!\n");
                        BUG.stdoutflag = 1;
                    }
                    BUG.eflag = 1; BUG.printflag = 1;
#else
                    Terminate(-1);
#endif
                }
            }
            startinline = 0;
            Out = AO.OutputLine;
            if ( AO.BlockSpaces == 0 ) {
                for ( j = 0; j < AO.OutSkip; j++ ) { *Out++ = ' '; }
                if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) ) {
                  if ( AO.OutSkip == 7 ) {
                    Out[-2] = fcontchar;
                    Out[-1] = ' ';
                  }
                }
            }
/*
            Out = AO.OutputLine + AO.OutSkip;
            if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
             && AO.OutSkip == 7 ) {
                Out[-2] = fcontchar;
                Out[-1] = ' ';
            }
            else {
                for ( j = 0; j < AO.OutSkip; j++ ) { AO.OutputLine[j] = ' '; }
            }
*/
            if ( AO.IsBracket ) { *Out++ = ' '; *Out++ = ' '; *Out++ = ' '; }
            *Out = '\0';
            if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
             || ( AC.OutputMode == CMODE && AO.FactorMode == 0 ) ) AO.InFbrack++;
        }
        if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
                /* Very long numbers */
            if ( i > (WORD)(AO.OutStop-Out) ) j = (WORD)(AO.OutStop - Out);
            else                              j = i;
            i -= j;
            NCOPYB(Out,s,j);
        }
        else {
            if ( i > (WORD)(AO.OutStop-Out) ) j = (WORD)(AO.OutStop - Out - 1);
            else                              j = i;
            i -= j;
            NCOPYB(Out,s,j);
            if ( i > 0 ) *Out++ = '\\';
        }
    }
    *Out = '\0';
    AO.OutFill = Out;
}

/*
        #] TokenToLine :
        #[ CodeToLine :         VOID CodeToLine(name,number,mode)

    Writes a name and possibly its number to output as a single token.

*/

UBYTE *CodeToLine(WORD number, UBYTE *Out)
{
    Out = StrCopy((UBYTE *)"(",Out);
    Out = NumCopy(number,Out);
    Out = StrCopy((UBYTE *)")",Out);
    return(Out);
}

/*
        #] CodeToLine :
```

```
        #[ MultiplyToLine :
*/

void MultiplyToLine()
{
    int i;
    if ( AO.CurrentDictionary > 0 && AO.CurDictSpecials > 0
     && AO.CurDictSpecials == DICT_DOSPECIALS ) {
        DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
/*
        Find the star:
*/
        for ( i = 0; i < dict->numelements; i++ ) {
            if ( dict->elements[i]->type != DICT_SPECIALCHARACTER ) continue;
            if ( (UBYTE)dict->elements[i]->lhs[0] == (UBYTE)('*') ) {
                TokenToLine((UBYTE *)(dict->elements[i]->rhs));
                return;
            }
        }
    }
    TokenToLine((UBYTE *)"*");
}

/*
        #] MultiplyToLine :
        #[ AddArrayIndex :
*/

UBYTE *AddArrayIndex(WORD num,UBYTE *out)
{
    if ( AC.OutputMode == CMODE ) {
        out = StrCopy((UBYTE *)"[",out);
        out = NumCopy(num,out);
        out = StrCopy((UBYTE *)"]",out);
    }
    else {
        out = StrCopy((UBYTE *)"(",out);
        out = NumCopy(num,out);
        out = StrCopy((UBYTE *)")",out);
    }
    return(out);
}

/*
        #] AddArrayIndex :
        #[ PrtTerms :           VOID PrtTerms()
*/

VOID PrtTerms()
{
    UWORD a[2];
    WORD na;
    a[0] = (UWORD)AO.NumInBrack;
    a[1] = (UWORD)(AO.NumInBrack » BITSINWORD);
    if ( a[1] ) na = 2;
    else na = 1;
    TokenToLine((UBYTE *)" ");
    LongToLine(a,na);
    if ( a[0] == 1 && na == 1 ) {
        TokenToLine((UBYTE *)" term");
    }
    else TokenToLine((UBYTE *)" terms");
    AO.NumInBrack = 0;
}

/*
        #] PrtTerms :
        #[ WrtPower :
*/

UBYTE *WrtPower(UBYTE *Out, WORD Power)
{
    if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
         || AC.OutputMode == REDUCEMODE ) {
        *Out++ = '*'; *Out++ = '*';
    }
    else if ( AC.OutputMode == CMODE ) *Out++ = ',';
    else *Out++ = '^';
    if ( Power >= 0 ) {
        if ( Power < 2*MAXPOWER )
            Out = NumCopy(Power,Out);
        else
            Out = StrCopy(FindSymbol((WORD)((LONG)Power-2*MAXPOWER)),Out);
/*          Out = StrCopy(VARNAME(symbols,(LONG)Power-2*MAXPOWER),Out); */
        if ( AC.OutputMode == CMODE ) *Out++ = ')';
        *Out = 0;
    }
```

```
    else {
        if ( ( AC.OutputMode >= FORTRANMODE || AC.OutputMode >= PFORTRANMODE )
         && AC.OutputMode != CMODE )
            *Out++ = '(';
        *Out++ = '-';
        if ( Power > -2*MAXPOWER )
            Out = NumCopy(-Power,Out);
        else
            Out = StrCopy(FindSymbol((WORD)((LONG)Power-2*MAXPOWER)),Out);
/*          Out = StrCopy(VARNAME(symbols,(LONG)(-Power)-2*MAXPOWER),Out); */
        if ( AC.OutputMode >= FORTRANMODE || AC.OutputMode >= PFORTRANMODE ) *Out++ = ')';
        *Out = 0;
    }
    return(Out);
}

/*
        #] WrtPower :
        #[ PrintTime :
*/

void PrintTime()
{
    LONG millitime = TimeCPU(1);
    WORD timepart = (WORD)(millitime%1000);
    millitime /= 1000;
    timepart /= 10;
    MesPrint("Time = %7l.%2i sec",millitime,timepart);
}

/*
        #] PrintTime :
    #] schryf-Utilities :
    #[ schryf-Writes :
        #[ WriteLists :         VOID WriteLists()

    Writes the namelists. If mode > 0 also the internal codes are given.

*/

static UBYTE *symname[] = {
     (UBYTE *)"(cyclic)",(UBYTE *)"(reversecyclic)"
    ,(UBYTE *)"(symmetric)",(UBYTE *)"(antisymmetric)" };
static UBYTE *rsymname[] = {
     (UBYTE *)"(-cyclic)",(UBYTE *)"(-reversecyclic)"
    ,(UBYTE *)"(-symmetric)",(UBYTE *)"(-antisymmetric)" };

VOID WriteLists()
{
    GETIDENTITY
    WORD i, j, k, *skip;
    int first, startvalue;
    UBYTE *OutScr, *Out;
    EXPRESSIONS e;
    CBUF *C = cbuf+AC.cbufnum;
    int olddict = AO.CurrentDictionary;
    skip = &AO.OutSkip;
    *skip = 0;
    AO.OutputLine = AO.OutFill = (UBYTE *)AT.WorkPointer;
    AO.CurrentDictionary = 0;
    FiniLine();
    OutScr = (UBYTE *)AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) /2;
    if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
    else startvalue = FIRSTUSERSYMBOL;
    if ( ( j = NumSymbols ) > startvalue ) {
        TokenToLine((UBYTE *)" Symbols");
        *skip = 3;
        FiniLine();
        for ( i = startvalue; i < j; i++ ) {
            if ( i >= BUILTINSYMBOLS && i < FIRSTUSERSYMBOL ) continue;
            Out = StrCopy(VARNAME(symbols,i),OutScr);
            if ( symbols[i].minpower > -MAXPOWER || symbols[i].maxpower < MAXPOWER ) {
                Out = StrCopy((UBYTE *)"(",Out);
                if ( symbols[i].minpower > -MAXPOWER )
                    Out = NumCopy(symbols[i].minpower,Out);
                Out = StrCopy((UBYTE *)":",Out);
                if ( symbols[i].maxpower < MAXPOWER )
                    Out = NumCopy(symbols[i].maxpower,Out);
                Out = StrCopy((UBYTE *)")",Out);
            }
            if ( ( symbols[i].complex & VARTYPEIMAGINARY ) == VARTYPEIMAGINARY ) {
                Out = StrCopy((UBYTE *)"#i",Out);
            }
            else if ( ( symbols[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) {
                Out = StrCopy((UBYTE *)"#c",Out);
            }
            else if ( ( symbols[i].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
```

```
                Out = StrCopy((UBYTE *)"#",Out);
                if ( ( symbols[i].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
                    Out = StrCopy((UBYTE *)"-",Out);
                }
                else {
                    Out = StrCopy((UBYTE *)"+",Out);
                }
                Out = NumCopy(symbols[i].maxpower,Out);
            }
            if ( AC.CodesFlag ) Out = CodeToLine(i,Out);
            if ( ( symbols[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) i++;
            StrCopy((UBYTE *)" ",Out);
            TokenToLine(OutScr);
        }
        *skip = 0;
        FiniLine();
    }
    if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
    else startvalue = BUILTININDICES;
    if ( ( j = NumIndices ) > startvalue ) {
        TokenToLine((UBYTE *)" Indices");
        *skip = 3;
        FiniLine();
        for ( i = startvalue; i < j; i++ ) {
            Out = StrCopy(FindIndex(i+AM.OffsetIndex),OutScr);
            Out = StrCopy(VARNAME(indices,i),OutScr);
            if ( indices[i].dimension >= 0 ) {
                if ( indices[i].dimension != AC.lDefDim ) {
                    Out = StrCopy((UBYTE *)"=",Out);
                    Out = NumCopy(indices[i].dimension,Out);
                }
            }
            else if ( indices[i].dimension < 0 ) {
                Out = StrCopy((UBYTE *)"=",Out);
                Out = StrCopy(VARNAME(symbols,-indices[i].dimension),Out);
                if ( indices[i].nmin4 < -NMIN4SHIFT ) {
                    Out = StrCopy((UBYTE *)":",Out);
                    Out = StrCopy(VARNAME(symbols,-indices[i].nmin4-NMIN4SHIFT),Out);
                }
            }
            if ( AC.CodesFlag ) Out = CodeToLine(i+AM.OffsetIndex,Out);
            StrCopy((UBYTE *)" ",Out);
            TokenToLine(OutScr);
        }
        *skip = 0;
        FiniLine();
    }
    if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
    else startvalue = BUILTINVECTORS;
    if ( ( j = NumVectors ) > startvalue ) {
        TokenToLine((UBYTE *)" Vectors");
        *skip = 3;
        FiniLine();
        for ( i = startvalue; i < j; i++ ) {
            Out = StrCopy(VARNAME(vectors,i),OutScr);
            if ( AC.CodesFlag ) Out = CodeToLine(i+AM.OffsetVector,Out);
            StrCopy((UBYTE *)" ",Out);
            TokenToLine(OutScr);
        }
        *skip = 0;
        FiniLine();
    }

    if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
    else startvalue = AM.NumFixedFunctions;
    for ( k = 0; k < 2; k++ ) {
        first = 1;
        j = NumFunctions;
        for ( i = startvalue; i < j; i++ ) {
            if ( i > MAXBUILTINFUNCTION-FUNCTION
             && i < FIRSTUSERFUNCTION-FUNCTION ) continue;
            if ( ( k == 0 && functions[i].commute )
            || ( k != 0 && !functions[i].commute ) ) {
                if ( first ) {
                    TokenToLine((UBYTE *)(FG.FunNam[k]));
                    *skip = 3;
                    FiniLine();
                    first = 0;
                }
                Out = StrCopy(VARNAME(functions,i),OutScr);
                if ( ( functions[i].complex & VARTYPEIMAGINARY ) == VARTYPEIMAGINARY ) {
                    Out = StrCopy((UBYTE *)"#i",Out);
                }
                else if ( ( functions[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) {
                    Out = StrCopy((UBYTE *)"#c",Out);
                }
                if ( functions[i].spec >= TENSORFUNCTION ) {
```

```
                    Out = StrCopy((UBYTE *)"(Tensor)",Out);
                }
                if ( functions[i].symmetric > 0 ) {
                    if ( ( functions[i].symmetric & REVERSEORDER ) != 0 ) {
                        Out = StrCopy((UBYTE *)(rsymname[(functions[i].symmetric & ~REVERSEORDER)-1]),Out);
                    }
                    else {
                        Out = StrCopy((UBYTE *)(symname[functions[i].symmetric-1]),Out);
                    }
                }
                if ( AC.CodesFlag ) Out = CodeToLine(i+FUNCTION,Out);
                if ( ( functions[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) i++;
                StrCopy((UBYTE *)" ",Out);
                TokenToLine(OutScr);
            }
        }
        *skip = 0;
        if ( first == 0 ) FiniLine();
    }
    if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
    else startvalue = AM.NumFixedSets;
    if ( ( j = AC.SetList.num ) > startvalue ) {
        WORD element, LastElement, type, number;
        TokenToLine((UBYTE *)" Sets");
        for ( i = startvalue; i < j; i++ ) {
            *skip = 3;
            FiniLine();
            if ( Sets[i].name < 0 ) {
                Out = StrCopy((UBYTE *)"{}",OutScr);
            }
            else {
                Out = StrCopy(VARNAME(Sets,i),OutScr);
            }
            if ( AC.CodesFlag ) Out = CodeToLine(i,Out);
            StrCopy((UBYTE *)":",Out);
            TokenToLine(OutScr);
            if ( i < AM.NumFixedSets ) {
                TokenToLine((UBYTE *)" ");
                TokenToLine((UBYTE *)fixedsets[i].description);
            }
            else if ( Sets[i].type == CRANGE ) {
                int iflag = 0;
                if ( Sets[i].first == 3*MAXPOWER ) {
                }
                else if ( Sets[i].first >= MAXPOWER ) {
                    TokenToLine((UBYTE *)"<=");
                    NumCopy(Sets[i].first-2*MAXPOWER,OutScr);
                    TokenToLine(OutScr);
                    iflag = 1;
                }
                else {
                    TokenToLine((UBYTE *)"<");
                    NumCopy(Sets[i].first,OutScr);
                    TokenToLine(OutScr);
                    iflag = 1;
                }
                if ( Sets[i].last == -3*MAXPOWER ) {
                }
                else if ( Sets[i].last <= -MAXPOWER ) {
                    if ( iflag ) TokenToLine((UBYTE *)",");
                    TokenToLine((UBYTE *)">=");
                    NumCopy(Sets[i].last+2*MAXPOWER,OutScr);
                    TokenToLine(OutScr);
                }
                else {
                    if ( iflag ) TokenToLine((UBYTE *)",");
                    TokenToLine((UBYTE *)">");
                    NumCopy(Sets[i].last,OutScr);
                    TokenToLine(OutScr);
                }
            }
            else {
                element = Sets[i].first;
                LastElement = Sets[i].last;
                type = Sets[i].type;
                do {
                    TokenToLine((UBYTE *)" ");
                    number = SetElements[element++];
                    switch ( type ) {
                        case CSYMBOL:
                            if ( number < 0 ) {
                                StrCopy(VARNAME(symbols,-number),OutScr);
                                StrCopy((UBYTE *)"?",Out);
                                TokenToLine(OutScr);
                            }
                            else if ( number < MAXPOWER )
```

```
                                TokenToLine(VARNAME(symbols,number));
                            else {
                                NumCopy(number-2*MAXPOWER,OutScr);
                                TokenToLine(OutScr);
                            }
                            break;
                        case CINDEX:
                            if ( number >= AM.IndDum ) {
                                Out = StrCopy((UBYTE *)"N",OutScr);
                                Out = NumCopy(number-(AM.IndDum),Out);
                                StrCopy((UBYTE *)"_?",Out);
                                TokenToLine(OutScr);
                            }
                            else if ( number >= AM.OffsetIndex + (WORD)WILDMASK ) {
                                Out = StrCopy(VARNAME(indices,number
                                -AM.OffsetIndex-WILDMASK),OutScr);
                                StrCopy((UBYTE *)"?",Out);
                                TokenToLine(OutScr);
                            }
                            else if ( number >= AM.OffsetIndex ) {
                                TokenToLine(VARNAME(indices,number-AM.OffsetIndex));
                            }
                            else {
                                NumCopy(number,OutScr);
                                TokenToLine(OutScr);
                            }
                            break;
                        case CVECTOR:
                            Out = OutScr;
                            if ( number < AM.OffsetVector ) {
                                number += WILDMASK;
                                Out = StrCopy((UBYTE *)"-",Out);
                            }
                            if ( number >= AM.OffsetVector + WILDOFFSET ) {
                                Out = StrCopy(VARNAME(vectors,number
                                -AM.OffsetVector-WILDOFFSET),Out);
                                StrCopy((UBYTE *)"?",Out);
                            }
                            else {
                                Out = StrCopy(VARNAME(vectors,number-AM.OffsetVector),Out);
                            }
                            TokenToLine(OutScr);
                            break;
                        case CFUNCTION:
                            if ( number >= FUNCTION + (WORD)WILDMASK ) {
                                Out = StrCopy(VARNAME(functions,number
                                -FUNCTION-WILDMASK),OutScr);
                                StrCopy((UBYTE *)"?",Out);
                                TokenToLine(OutScr);
                            }
                            TokenToLine(VARNAME(functions,number-FUNCTION));
                            break;
                        default:
                            NumCopy(number,OutScr);
                            TokenToLine(OutScr);
                            break;
                    }
                } while ( element < LastElement );
            }
        }
        *skip = 0;
        FiniLine();
    }
    if ( AS.ExecMode ) {
        e = Expressions;
        j = NumExpressions;
        first = 1;
        for ( i = 0; i < j; i++, e++ ) {
            if ( e->status >= 0 ) {
                if ( first ) {
                    TokenToLine((UBYTE *)" Expressions");
                    *skip = 3;
                    FiniLine();
                    first = 0;
                }
                Out = StrCopy(AC.exprnames->namebuffer+e->name,OutScr);
                Out = StrCopy((UBYTE *)(FG.ExprStat[e->status]),Out);
                if ( AC.CodesFlag ) Out = CodeToLine(i,Out);
                StrCopy((UBYTE *)" ",Out);
                TokenToLine(OutScr);
            }
        }
        if ( !first ) {
            *skip = 0;
            FiniLine();
        }
    }
```

```
    e = Expressions;
    j = NumExpressions;
    first = 1;
    for ( i = 0; i < j; i++ ) {
        if ( e->printflag && ( e->status == LOCALEXPRESSION ||
        e->status == GLOBALEXPRESSION || e->status == UNHIDELEXPRESSION
        || e->status == UNHIDEGEXPRESSION ) ) {
            if ( first ) {
                TokenToLine((UBYTE *)" Expressions to be printed");
                *skip = 3;
                FiniLine();
                first = 0;
            }
            Out = StrCopy(AC.exprnames->namebuffer+e->name,OutScr);
            StrCopy((UBYTE *)" ",Out);
            TokenToLine(OutScr);
        }
        e++;
    }
    if ( !first ) {
        *skip = 0;
        FiniLine();
    }

    if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
    else startvalue = BUILTINDOLLARS;
    if ( ( j = NumDollars ) > startvalue ) {
        TokenToLine((UBYTE *)" Dollar variables");
        *skip = 3;
        FiniLine();
        for ( i = startvalue; i < j; i++ ) {
            Out = StrCopy((UBYTE *)"$", OutScr);
            Out = StrCopy(DOLLARNAME(Dollars, i), Out);
            if ( AC.CodesFlag ) Out = CodeToLine(i, Out);
            StrCopy((UBYTE *)" ", Out);
            TokenToLine(OutScr);
        }
        *skip = 0;
        FiniLine();
    }

    if ( ( j = NumPotModdollars ) > 0 ) {
        TokenToLine((UBYTE *)" Dollar variables to be modified");
        *skip = 3;
        FiniLine();
        for ( i = 0; i < j; i++ ) {
            Out = StrCopy((UBYTE *)"$", OutScr);
            Out = StrCopy(DOLLARNAME(Dollars, PotModdollars[i]), Out);
            for ( k = 0; k < NumModOptdollars; k++ )
                if ( ModOptdollars[k].number == PotModdollars[i] ) break;
            if ( k < NumModOptdollars ) {
                switch ( ModOptdollars[k].type ) {
                    case MODSUM:
                        Out = StrCopy((UBYTE *)"(sum)", Out);
                        break;
                    case MODMAX:
                        Out = StrCopy((UBYTE *)"(maximum)", Out);
                        break;
                    case MODMIN:
                        Out = StrCopy((UBYTE *)"(minimum)", Out);
                        break;
                    case MODLOCAL:
                        Out = StrCopy((UBYTE *)"(local)", Out);
                        break;
                    default:
                        Out = StrCopy((UBYTE *)"(?)", Out);
                        break;
                }
            }
            StrCopy((UBYTE *)" ", Out);
            TokenToLine(OutScr);
        }
        *skip = 0;
        FiniLine();
    }

    if ( AC.ncmod != 0 ) {
        TokenToLine((UBYTE *)"All arithmetic is modulus ");
        LongToLine((UWORD *)AC.cmod,ABS(AC.ncmod));
        if ( AC.ncmod > 0 ) TokenToLine((UBYTE *)" with powerreduction");
        else                TokenToLine((UBYTE *)" without powerreduction");
        if ( ( AC.modmode & POSNEG ) != 0 ) TokenToLine((UBYTE *)" centered around 0");
        else                                TokenToLine((UBYTE *)" positive numbers only");
        FiniLine();
    }
    if ( AC.lDefDim != 4 ) {
        TokenToLine((UBYTE *)"The default dimension is ");
```

```
        if ( AC.lDefDim >= 0 ) {
            NumCopy(AC.lDefDim,OutScr);
            TokenToLine(OutScr);
        }
        else {
            TokenToLine(VARNAME(symbols,-AC.lDefDim));
            if ( AC.lDefDim4 != -NMIN4SHIFT ) {
                TokenToLine((UBYTE *)":");
                if ( AC.lDefDim4 >= -NMIN4SHIFT ) {
                    NumCopy(AC.lDefDim4,OutScr);
                    TokenToLine(OutScr);
                }
                else {
                    TokenToLine(VARNAME(symbols,-AC.lDefDim4-NMIN4SHIFT));
                }
            }
        }
        FiniLine();
    }
    if ( AC.lUnitTrace != 4 ) {
        TokenToLine((UBYTE *)"The trace of the unit matrix is ");
        if ( AC.lUnitTrace >= 0 ) {
            NumCopy(AC.lUnitTrace,OutScr);
            TokenToLine(OutScr);
        }
        else {
            TokenToLine(VARNAME(symbols,-AC.lUnitTrace));
        }
        FiniLine();
    }
    if ( AO.NumDictionaries > 0 ) {
        for ( i = 0; i < AO.NumDictionaries; i++ ) {
            WriteDictionary(AO.Dictionaries[i]);
        }
        if ( olddict > 0 )
            MesPrint("\nCurrently dictionary %s is active\n",
                AO.Dictionaries[olddict-1]->name);
        else
            MesPrint("\nCurrently there is no actice dictionary\n");
    }
    if ( AC.CodesFlag ) {
        if ( C->numlhs > 0 ) {
            TokenToLine((UBYTE *)" Left Hand Sides:");
            AO.OutSkip = 3;
            for ( i = 1; i <= C->numlhs; i++ ) {
                FiniLine();
                skip = C->lhs[i];
                j = skip[1];
                while ( --j >= 0 ) { TalToLine((UWORD)(*skip++)); TokenToLine((UBYTE *)"  "); }
            }
            AO.OutSkip = 0;
            FiniLine();
        }
        if ( C->numrhs > 0 ) {
            TokenToLine((UBYTE *)" Right Hand Sides:");
            AO.OutSkip = 3;
            for ( i = 1; i <= C->numrhs; i++ ) {
                FiniLine();
                skip = C->rhs[i];
                while ( ( j = skip[0] ) != 0 ) {
                    while ( --j >= 0 ) { TalToLine((UWORD)(*skip++)); TokenToLine((UBYTE *)"  "); }
                }
                FiniLine();
            }
            AO.OutSkip = 0;
            FiniLine();
        }
    }
    AO.CurrentDictionary = olddict;
}

/*
        #] WriteLists :
        #[ WriteDictionary :

    This routine is part of WriteLists and should be called from there.
*/

void WriteDictionary(DICTIONARY *dict)
{
    GETIDENTITY
    int i, first;
    WORD *skip, na, *a, spec, *t, *tstop, j;
    UBYTE str[2], *OutScr, *Out;
    WORD oldoutputmode = AC.OutputMode, oldoutputspaces = AC.OutputSpaces;
    WORD oldoutskip = AO.OutSkip;
    AC.OutputMode = NORMALFORMAT;
```

```
    AC.OutputSpaces = NOSPACEFORMAT;
    MesPrint("===Contents of dictionary %s===",dict->name);
    skip = &AO.OutSkip;
    *skip = 3;
    AO.OutputLine = AO.OutFill = (UBYTE *)AT.WorkPointer;
    for ( j = 0; j < *skip; j++ ) *(AO.OutFill)++ = ' ';

    OutScr = (UBYTE *)AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) /2;
    for ( i = 0; i < dict->numelements; i++ ) {
        switch ( dict->elements[i]->type ) {
            case DICT_INTEGERNUMBER:
                LongToLine((UWORD *)(dict->elements[i]->lhs),dict->elements[i]->size);
                Out = OutScr; *Out = 0;
                break;
            case DICT_RATIONALNUMBER:
                a = dict->elements[i]->lhs;
                na = a[a[0]-1]; na = (ABS(na)-1)/2;
                RatToLine((UWORD *)(a+1),na);
                Out = OutScr; *Out = 0;
                break;
            case DICT_SYMBOL:
                na = dict->elements[i]->lhs[0];
                Out = StrCopy(VARNAME(symbols,na),OutScr);
                break;
            case DICT_VECTOR:
                na = dict->elements[i]->lhs[0]-AM.OffsetVector;
                Out = StrCopy(VARNAME(vectors,na),OutScr);
                break;
            case DICT_INDEX:
                na = dict->elements[i]->lhs[0]-AM.OffsetIndex;
                Out = StrCopy(VARNAME(indices,na),OutScr);
                break;
            case DICT_FUNCTION:
                na = dict->elements[i]->lhs[0]-FUNCTION;
                Out = StrCopy(VARNAME(functions,na),OutScr);
                break;
            case DICT_FUNCTION_WITH_ARGUMENTS:
                t = dict->elements[i]->lhs;
                na = *t-FUNCTION;
                Out = StrCopy(VARNAME(functions,na),OutScr);
                spec = functions[*t - FUNCTION].spec;
                tstop = t + t[1];
                first = 1;
                if ( t[1] <= FUNHEAD ) {}
                else if ( spec >= TENSORFUNCTION ) {
                    t += FUNHEAD; *Out++ = (UBYTE)'(';
                    while ( t < tstop ) {
                        if ( first == 0 ) *Out++ = (UBYTE)(',');
                        else first = 0;
                        j = *t++;
                        if ( j >= 0 ) {
                            if ( j < AM.OffsetIndex ) { Out = NumCopy(j,Out); }
                            else if ( j < AM.IndDum ) {
                                Out = StrCopy(VARNAME(indices,j-AM.OffsetIndex),Out);
                            }
                            else {
                                MesPrint("Currently wildcards are not allowed in dictionary elements");
                                Terminate(-1);
                            }
                        }
                        else {
                            Out = StrCopy(VARNAME(vectors,j-AM.OffsetVector),Out);
                        }
                    }
                    *Out++ = (UBYTE)')'; *Out = 0;
                }
                else {
                    t += FUNHEAD; *Out++ = (UBYTE)'('; *Out = 0;
                    TokenToLine(OutScr);
                    while ( t < tstop ) {
                        if ( !first ) TokenToLine((UBYTE *)",");
                        WriteArgument(t);
                        NEXTARG(t)
                        first = 0;
                    }
                    Out = OutScr;
                    *Out++ = (UBYTE)')'; *Out = 0;
                }
                break;
            case DICT_SPECIALCHARACTER:
                str[0] = (UBYTE)(dict->elements[i]->lhs[0]);
                str[1] = 0;
                Out = StrCopy(str,OutScr);
                break;
            default:
                Out = OutScr; *Out = 0;
```

```
                break;
        }
        Out = StrCopy((UBYTE *)": \"",Out);
        Out = StrCopy((UBYTE *)(dict->elements[i]->rhs),Out);
        Out = StrCopy((UBYTE *)"\"",Out);
        TokenToLine(OutScr);
        FiniLine();
    }
    MesPrint("========End of dictionary %s===",dict->name);
    AC.OutputMode = oldoutputmode;
    AC.OutputSpaces = oldoutputspaces;
    AO.OutSkip = oldoutskip;
}

/*
        #] WriteDictionary :
        #[ WriteArgument :      VOID WriteArgument(WORD *t)

        Write a single argument field. The general field goes to
        WriteExpression and the fast field is dealt with here.
*/

VOID WriteArgument(WORD *t)
{
    UBYTE buffer[180];
    UBYTE *Out;
    WORD i;
    int oldoutsidefun, oldlowestlevel = lowestlevel;
    lowestlevel = 0;
    if ( *t > 0 ) {
        oldoutsidefun = AC.outsidefun; AC.outsidefun = 0;
        WriteExpression(t+ARGHEAD,(LONG)(*t-ARGHEAD));
        AC.outsidefun = oldoutsidefun;
        goto CleanUp;
    }
    Out = buffer;
    if ( *t == -SNUMBER) {
        NumCopy(t[1],Out);
    }
    else if ( *t == -SYMBOL ) {
        if ( t[1] >= MAXVARIABLES-cbuf[AM.sbufnum].numrhs ) {
            Out = StrCopy(FindExtraSymbol(MAXVARIABLES-t[1]),Out);
/*
            Out = StrCopy((UBYTE *)AC.extrasym,Out);
            if ( AC.extrasymbols == 0 ) {
                Out = NumCopy((MAXVARIABLES-t[1]),Out);
                Out = StrCopy((UBYTE *)"_",Out);
            }
            else if ( AC.extrasymbols == 1 ) {
                Out = AddArrayIndex((MAXVARIABLES-t[1]),Out);
            }
*/
/*
            else if ( AC.extrasymbols == 2 ) {
                Out = NumCopy((MAXVARIABLES-t[1]),Out);
            }
*/
        }
        else {
            StrCopy(FindSymbol(t[1]),Out);
/*          StrCopy(VARNAME(symbols,t[1]),Out); */
        }
    }
    else if ( *t == -VECTOR ) {
        if ( t[1] == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
        else
            StrCopy(FindVector(t[1]),Out);
/*          StrCopy(VARNAME(vectors,t[1] - AM.OffsetVector),Out); */
    }
    else if ( *t == -MINVECTOR ) {
        *Out++ = '-';
        StrCopy(FindVector(t[1]),Out);
/*      StrCopy(VARNAME(vectors,t[1] - AM.OffsetVector),Out); */
    }
    else if ( *t == -INDEX ) {
        if ( t[1] >= 0 ) {
            if ( t[1] < AM.OffsetIndex ) { NumCopy(t[1],Out); }
            else {
                i = t[1];
                if ( i >= AM.IndDum ) {
                    i -= AM.IndDum;
                    *Out++ = 'N';
                    Out = NumCopy(i,Out);
                    *Out++ = '_';
                    *Out++ = '?';
                    *Out = 0;
                }
```

```
                else {
                    i -= AM.OffsetIndex;
                    Out = StrCopy(FindIndex(i%WILDOFFSET+AM.OffsetIndex),Out);
/*                  Out = StrCopy(VARNAME(indices,i%WILDOFFSET),Out); */
                    if ( i >= WILDOFFSET ) { *Out++ = '?'; *Out = 0; }
                }
            }
        }
        else if ( t[1] == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
        else
            StrCopy(FindVector(t[1]),Out);
/*          StrCopy(VARNAME(vectors,t[1] - AM.OffsetVector),Out); */
    }
    else if ( *t == -DOLLAREXPRESSION ) {
        DOLLARS d = Dollars + t[1];
        *Out++ = '$';
        StrCopy(AC.dollarnames->namebuffer+d->name,Out);
    }
    else if ( *t == -EXPRESSION ) {
        StrCopy(EXPRNAME(t[1]),Out);
    }
    else if ( *t <= -FUNCTION ) {
        StrCopy(FindFunction(-*t),Out);
/*      StrCopy(VARNAME(functions,-*t-FUNCTION),Out); */
    }
    else {
        MesPrint("Illegal function argument while writing");
        goto CleanUp;
    }
    TokenToLine(buffer);
CleanUp:
    lowestlevel = oldlowestlevel;
    return;
}

/*
        #] WriteArgument :
        #[ WriteSubTerm :       WORD WriteSubTerm(sterm,first)

    Writes a single subterm field to the output line.
    There is a recursion for functions.


#define NUMSPECS 8
UBYTE *specfunnames[NUMSPECS] = {
      (UBYTE *)"fac" , (UBYTE *)"nargs", (UBYTE *)"binom"
    , (UBYTE *)"sign", (UBYTE *)"mod", (UBYTE *)"min", (UBYTE *)"max"
    , (UBYTE *)"invfac" };
*/

WORD WriteSubTerm(WORD *sterm, WORD first)
{
    UBYTE buffer[80];
    UBYTE *Out, closepar[2] = { (UBYTE)')', 0};
    WORD *stopper, *t, *tt, i, j, po = 0;
    int oldoutsidefun;
    stopper = sterm + sterm[1];
    t = sterm + 2;
    switch ( *sterm ) {
        case SYMBOL :
            while ( t < stopper ) {
                if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
                    FiniLine();
                    if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
                        else IniLine(3);
                    if ( first ) TokenToLine((UBYTE *)" ");
                }
                if ( !first ) MultiplyToLine();
                if ( AC.OutputMode == CMODE && t[1] != 1 ) {
                    if ( AC.Cnumpows >= t[1] && t[1] > 0 ) {
                        po = t[1];
                        Out = StrCopy((UBYTE *)"POW",buffer);
                        Out = NumCopy(po,Out);
                        Out = StrCopy((UBYTE *)"(",Out);
                        TokenToLine(buffer);
                    }
                    else {
                        TokenToLine((UBYTE *)"pow(");
                    }
                }
                if ( *t < NumSymbols ) {
                    Out = StrCopy(FindSymbol(*t),buffer); t++;
/*                  Out = StrCopy(VARNAME(symbols,*t),buffer); t++; */
                }
                else {
/*
                    see also routine PrintSubtermList.
```

```
*/
                    Out = StrCopy(FindExtraSymbol(MAXVARIABLES-*t),buffer);
/*
                    Out = StrCopy((UBYTE *)AC.extrasym,buffer);
                    if ( AC.extrasymbols == 0 ) {
                        Out = NumCopy((MAXVARIABLES-*t),Out);
                        Out = StrCopy((UBYTE *)"_",Out);
                    }
                    else if ( AC.extrasymbols == 1 ) {
                        Out = AddArrayIndex((MAXVARIABLES-*t),Out);
                    }
*/
/*
                    else if ( AC.extrasymbols == 2 ) {
                        Out = NumCopy((MAXVARIABLES-*t),Out);
                    }
*/
                    t++;
                }
                if ( AC.OutputMode == CMODE && po > 1
                  && AC.Cnumpows >= po ) {
                    Out = StrCopy((UBYTE *)")",Out);
                    po = 0;
                }
                else if ( *t != 1 ) WrtPower(Out,*t);
                TokenToLine(buffer);
                t++;
                first = 0;
            }
            break;
        case VECTOR :
            while ( t < stopper ) {
                if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
                    FiniLine();
                    if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
                        else IniLine(3);
                    if ( first ) TokenToLine((UBYTE *)" ");
                }
                if ( !first ) MultiplyToLine();

                Out = StrCopy(FindVector(*t),buffer);
/*              Out = StrCopy(VARNAME(vectors,*t - AM.OffsetVector),buffer); */
                t++;
                if ( AC.OutputMode == MATHEMATICAMODE ) *Out++ = '[';
                else *Out++ = '(';
                if ( *t >= AM.OffsetIndex ) {
                    i = *t++;
                    if ( i >= AM.IndDum ) {
                        i -= AM.IndDum;
                        *Out++ = 'N';
                        Out = NumCopy(i,Out);
                        *Out++ = '_';
                        *Out++ = '?';
                        *Out = 0;
                    }
                    else
                        Out = StrCopy(FindIndex(i),Out);
/*                      Out = StrCopy(VARNAME(indices,i - AM.OffsetIndex),Out); */
                }
                else if ( *t == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
                else {
                    Out = NumCopy(*t++,Out);
                }
                if ( AC.OutputMode == MATHEMATICAMODE ) *Out++ = ']';
                else *Out++ = ')';
                *Out = 0;
                TokenToLine(buffer);
                first = 0;
            }
            break;
        case INDEX :
            while ( t < stopper ) {
                if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
                    FiniLine();
                    if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
                        else IniLine(3);
                    if ( first ) TokenToLine((UBYTE *)" ");
                }
                if ( !first ) MultiplyToLine();
            if ( *t >= 0 ) {
                if ( *t < AM.OffsetIndex ) {
                    TalToLine((UWORD)(*t++));
                }
                else {
                    i = *t++;
                    if ( i >= AM.IndDum ) {
                        i -= AM.IndDum;
```

```
                        Out = buffer;
                        *Out++ = 'N';
                        Out = NumCopy(i,Out);
                        *Out++ = '_';
                        *Out++ = '?';
                        *Out = 0;
                    }
                    else {
                        i -= AM.OffsetIndex;
                        Out = StrCopy(FindIndex(i%WILDOFFSET+AM.OffsetIndex),buffer);
/*                      Out = StrCopy(VARNAME(indices,i%WILDOFFSET),buffer); */
                        if ( i >= WILDOFFSET ) { *Out++ = '?'; *Out = 0; }
                    }
                    TokenToLine(buffer);
                }
            }
            else {
                TokenToLine(FindVector(*t)); t++;
/*              TokenToLine(VARNAME(vectors,*t - AM.OffsetVector)); t++; */
            }
            first = 0;
            }
            break;
        case DOLLAREXPRESSION:
            {
                DOLLARS d = Dollars + sterm[2];
                Out = StrCopy((UBYTE *)"$",buffer);
                Out = StrCopy(AC.dollarnames->namebuffer+d->name,Out);
                if ( sterm[3] != 1 ) WrtPower(Out,sterm[3]);
                TokenToLine(buffer);
            }
            first = 0;
            break;
        case DELTA :
            while ( t < stopper ) {
                if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
                    FiniLine();
                    if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
                        else IniLine(3);
                    if ( first ) TokenToLine((UBYTE *)" ");
                }
                if ( !first ) MultiplyToLine();
                Out = StrCopy((UBYTE *)"d_(",buffer);
                if ( *t >= AM.OffsetIndex ) {
                    if ( *t < AM.IndDum ) {
                        Out = StrCopy(FindIndex(*t),Out);
/*                      Out = StrCopy(VARNAME(indices,*t - AM.OffsetIndex),Out); */
                        t++;
                    }
                    else {
                        *Out++ = 'N';
                        Out = NumCopy( *t++ - AM.IndDum, Out);
                        *Out++ = '_';
                        *Out++ = '?';
                        *Out = 0;
                    }
                }
                else if ( *t == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
                else {
                    Out = NumCopy(*t++,Out);
                }
                *Out++ = ',';
                if ( *t >= AM.OffsetIndex ) {
                    if ( *t < AM.IndDum ) {
                        Out = StrCopy(FindIndex(*t),Out);
/*                      Out = StrCopy(VARNAME(indices,*t - AM.OffsetIndex),Out); */
                        t++;
                    }
                    else {
                        *Out++ = 'N';
                        Out = NumCopy(*t++ - AM.IndDum,Out);
                        *Out++ = '_';
                        *Out++ = '?';
                    }
                }
                else {
                    Out = NumCopy(*t++,Out);
                }
                *Out++ = ')';
                *Out = 0;
                TokenToLine(buffer);
                first = 0;
            }
            break;
        case DOTPRODUCT :
            while ( t < stopper ) {
                if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
```

```
                    FiniLine();
                    if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
                        else IniLine(3);
                    if ( first ) TokenToLine((UBYTE *)" ");
                }
                if ( !first ) MultiplyToLine();
                if ( AC.OutputMode == CMODE && t[2] != 1 )
                    TokenToLine((UBYTE *)"pow(");
                Out = StrCopy(FindVector(*t),buffer);
/*              Out = StrCopy(VARNAME(vectors,*t - AM.OffsetVector),buffer); */
                t++;
                if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
                 || AC.OutputMode == CMODE )
                    *Out++ = AO.FortDotChar;
                else *Out++ = '.';
                Out = StrCopy(FindVector(*t),Out);
/*              Out = StrCopy(VARNAME(vectors,*t - AM.OffsetVector),Out); */
                t++;
                if ( *t != 1 ) WrtPower(Out,*t);
                t++;
                TokenToLine(buffer);
                first = 0;
            }
            break;
        case EXPONENT :
#if FUNHEAD != 2
            t += FUNHEAD - 2;
#endif
            if ( !first ) MultiplyToLine();
            if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)"pow(");
            else TokenToLine((UBYTE *)"(");
            WriteArgument(t);
            if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
             || AC.OutputMode == REDUCEMODE )
                TokenToLine((UBYTE *)")**(");
            else if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)",");
            else TokenToLine((UBYTE *)")^(");
            NEXTARG(t)
            WriteArgument(t);
            TokenToLine((UBYTE *)")");
            break;
        case DENOMINATOR :
#if FUNHEAD != 2
            t += FUNHEAD - 2;
#endif
            if ( first ) TokenToLine((UBYTE *)"1/(");
            else TokenToLine((UBYTE *)"/(");
            WriteArgument(t);
            TokenToLine((UBYTE *)")");
            break;
        case SUBEXPRESSION:
            if ( !first ) MultiplyToLine();
            TokenToLine((UBYTE *)"(");
            t = cbuf[sterm[4]].rhs[sterm[2]];
            tt = t;
            while ( *tt ) tt += *tt;
            oldoutsidefun = AC.outsidefun; AC.outsidefun = 0;
            if ( *t ) {
                WriteExpression(t,(LONG)(tt-t));
            }
            else {
                TokenToLine((UBYTE *)"0");
            }
            AC.outsidefun = oldoutsidefun;
            TokenToLine((UBYTE *)")");
            if ( sterm[3] != 1 ) {
                TokenToLine((UBYTE *)"^");
                Out = buffer;
                NumCopy(sterm[3],Out);
                TokenToLine(buffer);
            }
            break;
        default :
            if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
                FiniLine();
                if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
                    else IniLine(3);
                if ( first ) TokenToLine((UBYTE *)" ");
            }
            if ( *sterm < FUNCTION ) {
            return(MesPrint("Illegal subterm while writing"));
            }
            if ( !first ) MultiplyToLine();
            first = 1;
            { UBYTE *tmp;
                if ( ( tmp = FindFunWithArgs(sterm) ) != 0 ) {
                    TokenToLine(tmp);
```

```
                    break;
                }
            }
            t += FUNHEAD-2;

            if ( *sterm == GAMMA && t[-FUNHEAD+1] == FUNHEAD+1 ) {
                TokenToLine((UBYTE *)"gi_(");
            }
            else {
                if ( *sterm != DUMFUN ) {
                    Out = StrCopy(FindFunction(*sterm),buffer);
/*                  Out = StrCopy(VARNAME(functions,*sterm - FUNCTION),buffer); */
                }
                else { Out = buffer; *Out = 0; }
                if ( t >= stopper ) {
                    TokenToLine(buffer);
                    break;
                }
                if ( AC.OutputMode == MATHEMATICAMODE ) { *Out++ = '['; closepar[0] = (UBYTE)']'; }
                else { *Out++ = '('; }
                *Out = 0;
                TokenToLine(buffer);
            }
            i = functions[*sterm - FUNCTION].spec;
            if ( i >= TENSORFUNCTION ) {
                int curdict = AO.CurrentDictionary;
                if ( AO.CurrentDictionary && AO.CurDictNotInFunctions > 0 )
                    AO.CurrentDictionary = 0;
                t = sterm + FUNHEAD;
                while ( t < stopper ) {
                    if ( !first ) TokenToLine((UBYTE *)",");
                    else first = 0;
                    j = *t++;
                    if ( j >= 0 ) {
                        if ( j < AM.OffsetIndex ) TalToLine((UWORD)(j));
                        else if ( j < AM.IndDum ) {
                            i = j - AM.OffsetIndex;
                            Out = StrCopy(FindIndex(i%WILDOFFSET+AM.OffsetIndex),buffer);
/*                          Out = StrCopy(VARNAME(indices,i%WILDOFFSET),buffer); */
                            if ( i >= WILDOFFSET ) { *Out++ = '?'; *Out = 0; }
                            TokenToLine(buffer);
                        }
                        else {
                            Out = buffer;
                            *Out++ = 'N';
                            Out = NumCopy(j - AM.IndDum,Out);
                            *Out++ = '_';
                            *Out++ = '?';
                            *Out = 0;
                            TokenToLine(buffer);
                        }
                    }
                    else if ( j == FUNNYVEC ) { TokenToLine((UBYTE *)"?"); }
                    else if ( j > -WILDOFFSET ) {
                        Out = buffer;
                        Out = NumCopy((UWORD)(-j + 4),Out);
                        *Out++ = '_';
                        *Out = 0;
                        TokenToLine(buffer);
                    }
                    else {
                        TokenToLine(FindVector(j));
/*                      TokenToLine(VARNAME(vectors,j - AM.OffsetVector)); */
                    }
                }
                AO.CurrentDictionary = curdict;
            }
            else {
                int curdict = AO.CurrentDictionary;
                if ( AO.CurrentDictionary && AO.CurDictNotInFunctions > 0 )
                    AO.CurrentDictionary = 0;
                while ( t < stopper ) {
                    if ( !first ) TokenToLine((UBYTE *)",");
                    WriteArgument(t);
                    NEXTARG(t)
                    first = 0;
                }
                AO.CurrentDictionary = curdict;
            }
            TokenToLine(closepar);
            closepar[0] = (UBYTE)')';
            break;
    }
    return(0);
}

/*
```

```
        #] WriteSubTerm :
        #[ WriteInnerTerm :     WORD WriteInnerTerm(term,first)

    Writes the contents of term to the output.
    Only the part that is inside parentheses is written.

*/

WORD WriteInnerTerm(WORD *term, WORD first)
{
    WORD *t, *s, *s1, *s2, n, i, pow;
    t = term;
    s = t+1;
    GETCOEF(t,n);
    while ( s < t ) {
        if ( *s == HAAKJE ) break;
        s += s[1];
    }
    if ( s < t ) { s += s[1]; }
    else { s = term+1; }

    if ( n < 0 || !first ) {
        if ( n > 0 ) { TOKENTOLINE(" + ","+") }
        else if ( n < 0 ) { n = -n; TOKENTOLINE(" - ","-") }
    }
    if ( AC.modpowers ) {
        if ( n == 1 && *t == 1 && t > s ) first = 1;
        else if ( ABS(AC.ncmod) == 1 ) {
            LongToLine((UWORD *)AC.powmod,AC.npowmod);
            TokenToLine((UBYTE *)"^");
            TalToLine(AC.modpowers[(LONG)((UWORD)*t)]);
            first = 0;
        }
        else {
            LONG jj;
            LongToLine((UWORD *)AC.powmod,AC.npowmod);
            TokenToLine((UBYTE *)"^");
            jj = (UWORD)*t;
            if ( n == 2 ) jj += ((LONG)t[1])«BITSINWORD;
            if ( AC.modpowers[jj+1] == 0 ) {
                TalToLine(AC.modpowers[jj]);
            }
            else {
                LongToLine(AC.modpowers+jj,2);
            }
            first = 0;
        }
    }
    else if ( n != 1 || *t != 1 || t[1] != 1 || t <= s ) {
        if ( lowestlevel && ( ( AO.PrintType & PRINTONEFUNCTION ) != 0 ) ) {
                FiniLine();
                if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
                else IniLine(3);
        }
        if ( AO.CurrentDictionary > 0 ) TransformRational((UWORD *)t,n);
        else                            RatToLine((UWORD *)t,n);
        first = 0;
    }
    else first = 1;
    while ( s < t ) {
        if ( lowestlevel && ( (AO.PrintType & (PRINTONEFUNCTION | PRINTALL)) == PRINTONEFUNCTION ) ) {
            FiniLine();
            if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
            else IniLine(3);
        }

/*
        #[ NEWGAMMA :
*/
#ifdef NEWGAMMA
        if ( *s == GAMMA ) {    /* String them up */
            WORD *tt,*ss;
            ss = AT.WorkPointer;
            *ss++ = GAMMA;
            *ss++ = s[1];
            FILLFUN(ss)
            *ss++ = s[FUNHEAD];
            tt = s + FUNHEAD + 1;
            n = s[1] - FUNHEAD-1;
            do {
                while ( --n >= 0 ) *ss++ = *tt++;
                tt = s + s[1];
                while ( *tt == GAMMA && tt[FUNHEAD] == s[FUNHEAD] && tt < t ) {
                    s = tt;
                    tt += FUNHEAD + 1;
                    n = s[1] - FUNHEAD-1;
                    if ( n > 0 ) break;
```

```
                }
            } while ( n > 0 );
            tt = AT.WorkPointer;
            AT.WorkPointer = ss;
            tt[1] = WORDDIF(ss,tt);
            if ( WriteSubTerm(tt,first) ) {
                MesCall("WriteInnerTerm");
                SETERROR(-1)
            }
            AT.WorkPointer = tt;
        }
        else
#endif
/*
        #] NEWGAMMA :
*/
        {
            if ( *s >= FUNCTION && AC.funpowers > 0
            && functions[*s-FUNCTION].spec == 0 && ( AC.funpowers == ALLFUNPOWERS ||
            ( AC.funpowers == COMFUNPOWERS && functions[*s-FUNCTION].commute == 0 ) ) ) {
                pow = 1;
                for(;;) {
                    s1 = s; s2 = s + s[1]; i = s[1];
                    if ( s2 < t ) {
                        while ( --i >= 0 && *s1 == *s2 ) { s1++; s2++; }
                        if ( i < 0 ) {
                            pow++; s = s+s[1];
                        }
                        else break;
                    }
                    else break;
                }
                if ( pow > 1 ) {
                    if ( AC.OutputMode == CMODE ) {
                        if ( !first ) MultiplyToLine();
                        TokenToLine((UBYTE *)"pow(");
                        first = 1;
                    }
                    if ( WriteSubTerm(s,first) ) {
                        MesCall("WriteInnerTerm");
                        SETERROR(-1)
                    }
                    if ( AC.OutputMode == FORTRANMODE
                     || AC.OutputMode == PFORTRANMODE ) { TokenToLine((UBYTE *)"**"); }
                    else if ( AC.OutputMode == CMODE ) { TokenToLine((UBYTE *)","); }
                    else { TokenToLine((UBYTE *)"^"); }
                    TalToLine(pow);
                    if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)")");
                }
                else if ( WriteSubTerm(s,first) ) {
                    MesCall("WriteInnerTerm");
                    SETERROR(-1)
                }
            }
            else if ( WriteSubTerm(s,first) ) {
                MesCall("WriteInnerTerm");
                SETERROR(-1)
            }
        }
        first = 0;
        s += s[1];
    }
    return(0);
}

/*
        #] WriteInnerTerm :
        #[ WriteTerm :          WORD WriteTerm(term,lbrac,first,prtf,br)

    Writes a term to output. It tests the bracket information first.
    If there are no brackets or the bracket is the same all is passed
    to WriteInnerTerm. If there are brackets and the bracket is not
    the same as for the predecessor the old bracket is closed and
    a new one is opened.
    br indicates whether we are in a subexpression, barring zeroing
    AO.IsBracket

*/

WORD WriteTerm(WORD *term, WORD *lbrac, WORD first, WORD prtf, WORD br)
{
    WORD *t, *stopper, *b, n;
    int oldIsFortran90 = AC.IsFortran90, i;
    if ( *lbrac >= 0 ) {
        t = term + 1;
        stopper = (term + *term - 1);
        stopper -= ABS(*stopper) - 1;
```

```
        while ( t < stopper ) {
            if ( *t == HAAKJE ) {
                stopper = t;
                t = term+1;
                if ( *lbrac == ( n = WORDDIF(stopper,t) ) ) {
                    b = AO.bracket + 1;
                    t = term + 1;
                    while ( n > 0 && ( *b++ == *t++ ) ) { n--; }
                    if ( n <= 0 && ( ( AO.InFbrack < AM.FortranCont )
                    || ( lowestlevel == 0 ) ) ) {
/*
                        We continue inside a bracket.
*/
                        AO.IsBracket = 1;
                        if ( ( prtf & PRINTCONTENTS ) != 0 ) {
                            AO.NumInBrack++;
                        }
                        else {
                            if ( WriteInnerTerm(term,0) ) goto WrtTmes;
                            if ( ( AO.PrintType & PRINTONETERM ) != 0 ) {
                                FiniLine();
                                TokenToLine((UBYTE *)"   ");
                            }
                        }
                        return(0);
                    }
                    t = term + 1;
                    n = WORDDIF(stopper,t);
                }
/*
                Close the bracket
*/
                if ( *lbrac ) {
                    if ( ( prtf & PRINTCONTENTS ) ) PrtTerms();
                    TOKENTOLINE(" )",")")
                    if ( AC.OutputMode == CMODE && AO.FactorMode == 0 )
                        TokenToLine((UBYTE *)";");
                    else if ( AO.FactorMode && ( n == 0 ) ) {
/*
                        This should not happen.
*/
                        return(0);
                    }
                    AC.IsFortran90 = ISNOTFORTRAN90;
                    FiniLine();
                    AC.IsFortran90 = oldIsFortran90;
                    if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE
                        && AC.OutputSpaces == NORMALFORMAT
                        && AO.FactorMode == 0 ) FiniLine();
                }
                else {
                    if ( AC.OutputMode == CMODE && AO.FactorMode == 0 )
                        TokenToLine((UBYTE *)";");
                    if ( AO.FortFirst == 0 ) {
                        if ( !first ) {
                            AC.IsFortran90 = ISNOTFORTRAN90;
                            FiniLine();
                            AC.IsFortran90 = oldIsFortran90;
                        }
                    }
                }
                if ( AO.FactorMode == 0 ) {
                  if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
                   && !first ) {
                    WORD oldmode = AC.OutputMode;
                    AC.OutputMode = 0;
                    IniLine(0);
                    AC.OutputMode = oldmode;
                    AO.OutSkip = 7;

                    if ( AO.FortFirst == 0 ) {
                        TokenToLine(AO.CurBufWrt);
                        TOKENTOLINE(" = ","=")
                        TokenToLine(AO.CurBufWrt);
                    }
                    else {
                        AO.FortFirst = 0;
                        TokenToLine(AO.CurBufWrt);
                        TOKENTOLINE(" = ","=")
                    }
                  }
                  else if ( AC.OutputMode == CMODE && !first ) {
                    IniLine(0);
                    if ( AO.FortFirst == 0 ) {
                        TokenToLine(AO.CurBufWrt);
                        TOKENTOLINE(" += ","+=")
                    }
```

```
                    else {
                        AO.FortFirst = 0;
                        TokenToLine(AO.CurBufWrt);
                        TOKENTOLINE(" = ","=")
                    }
                  }
                  else if ( startinline == 0 ) {
                    IniLine(0);
                  }
                  AO.InFbrack = 0;
                  if ( ( *lbrac = n ) > 0 ) {
                    b = AO.bracket;
                    *b++ = n + 4;
                    while ( --n >= 0 ) *b++ = *t++;
                    *b++ = 1; *b++ = 1; *b = 3;
                    AO.IsBracket = 0;
                    if ( WriteInnerTerm(AO.bracket,0) ) {
                        /* Error message */
                        WORD i;
WrtTmes:                t = term;
                        AO.OutSkip = 3;
                        FiniLine();
                        i = *t;
                        while ( --i >= 0 ) { TalToLine((UWORD)(*t++));
                            if ( AC.OutputSpaces == NORMALFORMAT )
                                        TokenToLine((UBYTE *)"  "); }
                        AO.OutSkip = 0;
                        FiniLine();
                        MesCall("WriteTerm");
                        SETERROR(-1)
                    }
                    TOKENTOLINE(" * ( ","*(")
                    AO.NumInBrack = 0;
                    AO.IsBracket = 1;
                    if ( ( prtf & PRINTONETERM ) != 0 ) {
                        first = 0;
                        FiniLine();
                        TokenToLine((UBYTE *)"   ");
                    }
                    else first = 1;
                  }
                  else {
                    AO.IsBracket = 0;
                    first = 0;
                  }
                }
                else {
/*
                    Here is the code that writes the glue between two factors.
                    We should not forget factors that are zero!
*/
                    if ( ( *lbrac = n ) > 0 ) {
                        b = AO.bracket;
                        *b++ = n + 4;
                        while ( --n >= 0 ) *b++ = *t++;
                        *b++ = 1; *b++ = 1; *b = 3;
                        for ( i = AO.FactorNum+1; i < AO.bracket[4]; i++ ) {
                            if ( first ) {
                                TOKENTOLINE("   ( 0 )"," (0)")
                                first = 0;
                            }
                            else {
                                TOKENTOLINE(" * ( 0 )","*(0)")
                            }
                            FiniLine();
                            IniLine(0);
                        }
                        AO.FactorNum = AO.bracket[4];
                    }
                    else {
                        AO.NumInBrack = 0;
                        return(0);
                    }
                    if ( first == 0 ) { TOKENTOLINE(" * ( ","*(") }
                    else              { TOKENTOLINE("   ( "," (") }
                    AO.NumInBrack = 0;
                    first = 1;
                }
                if ( ( prtf & PRINTCONTENTS ) != 0 ) AO.NumInBrack++;
                else if ( WriteInnerTerm(term,first) ) goto WrtTmes;
                if ( ( AO.PrintType & PRINTONETERM ) != 0 ) {
                    FiniLine();
                    TokenToLine((UBYTE *)"   ");
                }
                return(0);
            }
            else t += t[1];
```

```
        }
        if ( *lbrac > 0 ) {
            if ( ( prtf & PRINTCONTENTS ) != 0 ) PrtTerms();
            TokenToLine((UBYTE *)" )");
            if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)";");
            if ( AO.FortFirst == 0 ) {
                AC.IsFortran90 = ISNOTFORTRAN90;
                FiniLine();
                AC.IsFortran90 = oldIsFortran90;
            }
            if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE
                 && AC.OutputSpaces == NORMALFORMAT ) FiniLine();
            if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
             && !first ) {
                WORD oldmode = AC.OutputMode;
                AC.OutputMode = 0;
                IniLine(0);
                AC.OutputMode = oldmode;
                AO.OutSkip = 7;
                if ( AO.FortFirst == 0 ) {
                    TokenToLine(AO.CurBufWrt);
                    TOKENTOLINE(" = ","=")
                    TokenToLine(AO.CurBufWrt);
                }
                else {
                    AO.FortFirst = 0;
                    TokenToLine(AO.CurBufWrt);
                    TOKENTOLINE(" = ","=")
                }
/*
                TokenToLine(AO.CurBufWrt);
                TOKENTOLINE(" = ","=")
                if ( AO.FortFirst == 0 )
                    TokenToLine(AO.CurBufWrt);
                else AO.FortFirst = 0;
*/
            }
            else if ( AC.OutputMode == CMODE && !first ) {
                IniLine(0);
                if ( AO.FortFirst == 0 ) {
                    TokenToLine(AO.CurBufWrt);
                    TOKENTOLINE(" += ","+=")
                }
                else {
                    AO.FortFirst = 0;
                    TokenToLine(AO.CurBufWrt);
                    TOKENTOLINE(" = ","=")
                }
/*
                TokenToLine(AO.CurBufWrt);
                if ( AO.FortFirst == 0 ) { TOKENTOLINE(" += ","+=") }
                else {
                    TOKENTOLINE(" = ","=")
                    AO.FortFirst = 0;
                }
*/
            }
            else IniLine(0);
            *lbrac = 0;
            first = 1;
        }
    }
    if ( !br ) AO.IsBracket = 0;
    if ( ( AO.InFbrack >= AM.FortranCont ) && lowestlevel ) {
        if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)";");
        if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
         && !first ) {
            WORD oldmode = AC.OutputMode;
            if ( AO.FortFirst == 0 ) {
                AC.IsFortran90 = ISNOTFORTRAN90;
                FiniLine();
                AC.IsFortran90 = oldIsFortran90;
                AC.OutputMode = 0;
                IniLine(0);
                AC.OutputMode = oldmode;
                AO.OutSkip = 7;
                TokenToLine(AO.CurBufWrt);
                TOKENTOLINE(" = ","=")
                TokenToLine(AO.CurBufWrt);
            }
            else {
                AO.FortFirst = 0;
/*
                TokenToLine(AO.CurBufWrt);
                TOKENTOLINE(" = ","=")
*/
            }
```

```
/*
            TokenToLine(AO.CurBufWrt);
            TOKENTOLINE(" = ","=")
            if ( AO.FortFirst == 0 )
                TokenToLine(AO.CurBufWrt);
            else AO.FortFirst = 0;
*/
        }
        else if ( AC.OutputMode == CMODE && !first ) {
            FiniLine();
            IniLine(0);
            if ( AO.FortFirst == 0 ) {
                TokenToLine(AO.CurBufWrt);
                TOKENTOLINE(" += ","+=")
            }
            else {
                AO.FortFirst = 0;
                TokenToLine(AO.CurBufWrt);
                TOKENTOLINE(" = ","=")
            }
/*
            TokenToLine(AO.CurBufWrt);
            if ( AO.FortFirst == 0 ) { TOKENTOLINE(" += ","+=") }
            else {
                TOKENTOLINE(" = ","=")
                AO.FortFirst = 0;
            }
*/
        }
        else {
            FiniLine();
            IniLine(0);
        }
        AO.InFbrack = 0;
    }
    if ( WriteInnerTerm(term,first) ) goto WrtTmes;
    if ( ( AO.PrintType & PRINTONETERM ) != 0 ) {
        FiniLine();
        IniLine(0);
    }
    return(0);
}

/*
        #] WriteTerm :
        #[ WriteExpression :    WORD WriteExpression(terms,ltot)

    Writes a subexpression to output.
    The subexpression is in terms and contains ltot words.
    This is only used for function arguments.

*/

WORD WriteExpression(WORD *terms, LONG ltot)
{
    WORD *stopper;
    WORD first, btot;
    WORD OldIsBracket = AO.IsBracket, OldPrintType = AO.PrintType;
    if ( !AC.outsidefun ) { AO.PrintType &= ~PRINTONETERM; first = 1; }
    else first = 0;
    stopper = terms + ltot;
    btot = -1;
    while ( terms < stopper ) {
        AO.IsBracket = OldIsBracket;
        if ( WriteTerm(terms,&btot,first,0,1) ) {
            MesCall("WriteExpression");
            SETERROR(-1)
        }
        first = 0;
        terms += *terms;
    }
/*  AO.IsBracket = 0; */
    AO.IsBracket = OldIsBracket;
    AO.PrintType = OldPrintType;
    return(0);
}

/*
        #] WriteExpression :
        #[ WriteAll :           WORD WriteAll()

        Writes all expressions that should be written
*/

WORD WriteAll()
{
    GETIDENTITY
```

```
    WORD lbrac, first;
    WORD *t, *stopper, n, prtf;
    int oldIsFortran90 = AC.IsFortran90, i;
    POSITION pos;
    FILEHANDLE *f;
    EXPRESSIONS e;
    if ( AM.exitflag ) return(0);
#ifdef WITHMPI
    if ( PF.me != MASTER ) {
        /*
         * For the slaves, we need to call Optimize() the same number of times
         * as the master. The first argument doesn't have any important role.
         */
        for ( n = 0; n < NumExpressions; n++ ) {
            e = &Expressions[n];
            if ( !e->printflag & PRINTON ) continue;
            switch ( e->status ) {
                case LOCALEXPRESSION:
                case GLOBALEXPRESSION:
                case UNHIDELEXPRESSION:
                case UNHIDEGEXPRESSION:
                    break;
                default:
                    continue;
            }
            e->printflag = 0;
            PutPreVar(AM.oldnumextrasymbols, GetPreVar((UBYTE *)"EXTRASYMBOLS_", 0), 0, 1);
            if ( AO.OptimizationLevel > 0 ) {
                if ( Optimize(0, 1) ) return(-1);
            }
        }
        return(0);
    }
#endif
    SeekScratch(AR.outfile,&pos);
    if ( ResetScratch() ) {
        MesCall("WriteAll");
        SETERROR(-1)
    }
    AO.termbuf = AT.WorkPointer;
    AO.bracket = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer*2);
    AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
    AT.WorkPointer += 2*AC.LineLength;
    *(AR.CompressBuffer) = 0;
    first = 0;
    for ( n = 0; n < NumExpressions; n++ ) {
        if ( ( Expressions[n].printflag & PRINTON ) != 0 ) { first = 1; break; }
    }
    if ( !first ) goto EndWrite;
    AO.IsBracket = 0;
    AO.OutSkip = 3;
    AR.DeferFlag = 0;
    while ( GetTerm(BHEAD AO.termbuf) ) {
        t = AO.termbuf + 1;
        e = Expressions + AO.termbuf[3];
        n = e->status;
        if ( ( n == LOCALEXPRESSION || n == GLOBALEXPRESSION
        || n == UNHIDELEXPRESSION || n == UNHIDEGEXPRESSION ) &&
        ( ( prtf = e->printflag ) & PRINTON ) != 0 ) {
            e->printflag = 0;
            AO.NumInBrack = 0;
            PutPreVar(AM.oldnumextrasymbols,
                    GetPreVar((UBYTE *)"EXTRASYMBOLS_",0),0,1);
            if ( ( prtf & PRINTLFILE ) != 0 ) {
                if ( AC.LogHandle < 0 ) prtf &= ~PRINTLFILE;
            }
            AO.PrintType = prtf;
/*
            if ( AC.OutputMode == VORTRANMODE ) {
                UBYTE *oldOutFill = AO.OutFill, *oldOutputLine = AO.OutputLine;
                AO.OutSkip = 6;
                if ( Optimize(AO.termbuf[3], 1) ) goto AboWrite;
                AO.OutSkip = 3;
                AO.OutFill = oldOutFill; AO.OutputLine = oldOutputLine;
                FiniLine();
                continue;
            }
            else
*/
            if ( AO.OptimizationLevel > 0 ) {
                UBYTE *oldOutFill = AO.OutFill, *oldOutputLine = AO.OutputLine;
                AO.OutSkip = 6;
                if ( Optimize(AO.termbuf[3], 1) ) goto AboWrite;
                AO.OutSkip = 3;
                AO.OutFill = oldOutFill; AO.OutputLine = oldOutputLine;
                FiniLine();
```

```
                continue;
            }
            if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
                     AO.OutSkip = 6;
            FiniLine();
            AO.CurBufWrt = EXPRNAME(AO.termbuf[3]);
            TokenToLine(AO.CurBufWrt);
            stopper = t + t[1];
            t += SUBEXPSIZE;
            if ( t < stopper ) {
                TokenToLine((UBYTE *)"(");
                first = 1;
                while ( t < stopper ) {
                    n = *t;
                    if ( !first ) TokenToLine((UBYTE *)",");
                    switch ( n ) {
                        case SYMTOSYM :
                            TokenToLine(FindSymbol(t[2]));
/*                          TokenToLine(VARNAME(symbols,t[2])); */
                            break;
                        case VECTOVEC :
                            TokenToLine(FindVector(t[2]));
/*                          TokenToLine(VARNAME(vectors,t[2] - AM.OffsetVector)); */
                            break;
                        case INDTOIND :
                            TokenToLine(FindIndex(t[2]));
/*                          TokenToLine(VARNAME(indices,t[2] - AM.OffsetIndex)); */
                            break;
                        default :
                            TokenToLine(FindFunction(t[2]));
/*                          TokenToLine(VARNAME(functions,t[2] - FUNCTION)); */
                            break;
                    }
                    t += t[1];
                    first = 0;
                }
                TokenToLine((UBYTE *)")");
            }
            TOKENTOLINE(" =","=");
            lbrac = 0;
            AO.InFbrack = 0;
            if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
                AO.FortFirst = 1;
            else
                AO.FortFirst = 0;
            first = 1;
            if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
                AO.FactorMode = 1+e->numfactors;
                AO.FactorNum = 0; /* Which factor are we doing. For factors that are zero */
            }
            else {
                AO.FactorMode = 0;
            }
            while ( GetTerm(BHEAD AO.termbuf) ) {
                WORD *m;
                GETSTOP(AO.termbuf,m);
                if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
                && ( ( prtf & PRINTONETERM ) != 0 ) ) {}
                else {
                    if ( first ) {
                        FiniLine();
                        IniLine(0);
                    }
                }
                if ( ( prtf & PRINTONETERM ) != 0 ) first = 0;
                if ( WriteTerm(AO.termbuf,&lbrac,first,prtf,0) )
                    goto AboWrite;
                first = 0;
            }
            if ( AO.FactorMode ) {
                if ( first ) { AO.FactorNum = 1; TOKENTOLINE("   ( 0 )","  (0)") }
                else TOKENTOLINE(" )",")");
                for ( i = AO.FactorNum+1; i <= e->numfactors; i++ ) {
                    FiniLine();
                    IniLine(0);
                    TOKENTOLINE(" * ( 0 )","*(0)");
                }
                AO.FactorNum = e->numfactors;
                if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE )
                    TokenToLine((UBYTE *)";");
            }
            else if ( AO.FactorMode == 0 || first ) {
                if ( first ) { TOKENTOLINE(" 0","0") }
                else if ( lbrac ) {
                    if ( ( prtf & PRINTCONTENTS ) != 0 ) PrtTerms();
                    TOKENTOLINE(" )",")")
                }
```

```
                else if ( ( prtf & PRINTCONTENTS ) != 0 ) {
                    TOKENTOLINE(" + 1 * ( ","+1*(")
                    PrtTerms();
                    TOKENTOLINE(" )",")")
                }
                if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE )
                    TokenToLine((UBYTE *)";");
            }
            AO.OutSkip = 3;
            AC.IsFortran90 = ISNOTFORTRAN90;
            FiniLine();
            AC.IsFortran90 = oldIsFortran90;
            AO.FactorMode = 0;
        }
        else {
            do { } while ( GetTerm(BHEAD AO.termbuf) );
        }
    }
    if ( AC.OutputSpaces == NORMALFORMAT ) FiniLine();
EndWrite:
    if ( AR.infile->handle >= 0 ) {
        SeekFile(AR.infile->handle,&(AR.infile->filesize),SEEK_SET);
    }
    AO.IsBracket = 0;
    AT.WorkPointer = AO.termbuf;
    SetScratch(AR.infile,&pos);
    f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
    return(0);
AboWrite:
    SetScratch(AR.infile,&pos);
    f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
    MesCall("WriteAll");
    Terminate(-1);
    return(-1);
}

/*
        #] WriteAll :
        #[ WriteOne :           WORD WriteOne(name,alreadyinline)

        Writes one expression from the preprocessor
*/

WORD WriteOne(UBYTE *name, int alreadyinline, int nosemi)
{
    GETIDENTITY
    WORD number;
    WORD lbrac, first;
    POSITION pos;
    FILEHANDLE *f;

    if ( GetName(AC.exprnames,name,&number,NOAUTO) != CEXPRESSION ) {
        MesPrint("@%s is not an expression",name);
        return(-1);
    }
    switch ( Expressions[number].status ) {
        case HIDDENLEXPRESSION:
        case HIDDENGEXPRESSION:
        case HIDELEXPRESSION:
        case HIDEGEXPRESSION:
        case UNHIDELEXPRESSION:
        case UNHIDEGEXPRESSION:
/*
        case DROPHLEXPRESSION:
        case DROPHGEXPRESSION:
*/
            AR.GetFile = 2;
            break;
        case LOCALEXPRESSION:
        case GLOBALEXPRESSION:
        case SKIPLEXPRESSION:
        case SKIPGEXPRESSION:
/*
        case DROPLEXPRESSION:
        case DROPGEXPRESSION:
*/
            AR.GetFile = 0;
            break;
        default:
            MesPrint("@expressions %s is not active. It cannot be written",name);
            return(-1);
    }
    SeekScratch(AR.outfile,&pos);

    f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
/*
    if ( ResetScratch() ) {
```

```
        MesCall("WriteOne");
        SETERROR(-1)
    }
*/
    if ( AR.GetFile == 2 ) f = AR.hidefile;
    else f = AR.infile;
/*
        Now position the file
*/
    if ( f->handle >= 0 ) {
        SetScratch(f,&(Expressions[number].onfile));
    }
    else {
        f->POfill = (WORD *)((UBYTE *)(f->PObuffer)
                 + BASEPOSITION(Expressions[number].onfile));
    }
    AO.termbuf = AT.WorkPointer;
    AO.bracket = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer);
    AT.WorkPointer = (WORD *)(((UBYTE *)(AT.WorkPointer)) + AM.MaxTer*2);

    AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
    AT.WorkPointer += 2*AC.LineLength;
    *(AR.CompressBuffer) = 0;

    AO.IsBracket = 0;
    AO.OutSkip = 3;
    AR.DeferFlag = 0;

    if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
             AO.OutSkip = 6;
    if ( GetTerm(BHEAD AO.termbuf) <= 0 ) {
        MesPrint("@ReadError in expression %s",name);
        goto AboWrite;
    }
/*
    PutPreVar(AM.oldnumextrasymbols,
            GetPreVar((UBYTE *)"EXTRASYMBOLS_",0),0,1);
*/
    /*
     * Currently WriteOne() is called only from writeToChannel() with setting
     * AO.OptimizationLevel = 0, which means Optimize() is never called here.
     * So we don't need to think about how to ensure that the master and the
     * slaves call Optimize() at the same time. (TU 26 Jul 2013)
     */
    if ( AO.OptimizationLevel > 0 ) {
        AO.OutSkip = 6;
        if ( Optimize(AO.termbuf[3], 1) ) goto AboWrite;
        AO.OutSkip = 3;
        FiniLine();
    }
    else {
        lbrac = 0;
        AO.InFbrack = 0;
        AO.FortFirst = 0;
        first = 1;
        while ( GetTerm(BHEAD AO.termbuf) ) {
            WORD *m;
            GETSTOP(AO.termbuf,m);
            if ( first ) {
                IniLine(0);
                startinline = alreadyinline;
                AO.OutFill = AO.OutputLine + startinline;
            }
            if ( WriteTerm(AO.termbuf,&lbrac,first,0,0) )
                goto AboWrite;
            first = 0;
        }
        if ( first ) {
            IniLine(0);
            startinline = alreadyinline;
            AO.OutFill = AO.OutputLine + startinline;
            TOKENTOLINE(" 0","0");
        }
        else if ( lbrac ) {
            TOKENTOLINE(" )",")");
        }
        if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE
             && nosemi == 0 ) TokenToLine((UBYTE *)";");
        AO.OutSkip = 3;
        if ( AC.OutputSpaces == NORMALFORMAT && nosemi == 0 ) {
            FiniLine();
        }
        else {
            noextralinefeed = 1;
            FiniLine();
            noextralinefeed = 0;
        }
```

```
    }
    AO.IsBracket = 0;
    AT.WorkPointer = AO.termbuf;
    SetScratch(f,&pos);
    f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
    AO.InFbrack = 0;
    return(0);
AboWrite:
    SetScratch(AR.infile,&pos);
    f->POposition = pos;
    f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
    MesCall("WriteOne");
    Terminate(-1);
    return(-1);
}

/*
        #] WriteOne :
    #] schryf-Writes :
*/
```

4.96 setfile.c File Reference

`#include "form3.h"`

Include dependency graph for setfile.c:

Macros

- #define NUMERICALVALUE 0
- #define STRINGVALUE 1
- #define PATHVALUE 2
- #define ONOFFVALUE 3
- #define DEFINEVALUE 4
- #define SETBUFSIZE 257

Functions

- int DoSetups ()
- int ProcessOption (UBYTE *s1, UBYTE *s2, int filetype)
- SETUPPARAMETERS * GetSetupPar (UBYTE *s)
- int RecalcSetups ()
- int AllocSetups ()
- VOID WriteSetup ()
- SORTING * AllocSort (LONG LargeSize, LONG SmallSize, LONG SmallEsize, LONG TermsInSmall, int MaxPatches, int MaxFpatches, LONG IOsize)
- VOID AllocSortFileName (SORTING *sort)
- FILEHANDLE * AllocFileHandle (WORD par, char *name)
- void DeAllocFileHandle (FILEHANDLE *fh)
- int MakeSetupAllocs ()
- int TryFileSetups ()
- int TryEnvironment ()

Variables

- char curdirp [ ] = "."
- char cursortdirp [ ] = "."
- char commentchar [ ] = "*"
- char dotchar [ ] = "_"
- char highfirst [ ] = "highfirst"
- char lowfirst [ ] = "lowfirst"
- char procedureextension [ ] = "prc"
- SETUPPARAMETERS setupparameters [ ]

4.96.1 Detailed Description

The routines that deal with the setup parameters.

Definition in file setfile.c.

4.96.2 Macro Definition Documentation

4.96.2.1 NUMERICALVALUE

```
#define NUMERICALVALUE 0
```

Definition at line 47 of file setfile.c.

4.96.2.2 STRINGVALUE

```
#define STRINGVALUE 1
```

Definition at line 48 of file setfile.c.

4.96.2.3 PATHVALUE

```
#define PATHVALUE 2
```

Definition at line 49 of file setfile.c.

4.96.2.4 ONOFFVALUE

```
#define ONOFFVALUE 3
```

Definition at line 50 of file setfile.c.

4.96.2.5 DEFINEVALUE

```
#define DEFINEVALUE 4
```

Definition at line 51 of file setfile.c.

4.96.2.6 SETBUFSIZE

```
#define SETBUFSIZE 257
```

Definition at line 1088 of file setfile.c.

4.96.3 Function Documentation

4.96.3.1 DoSetups()

```
int DoSetups ( )
```

Definition at line 125 of file setfile.c.

4.96.3.2 ProcessOption()

```
int ProcessOption (
            UBYTE * s1,
            UBYTE * s2,
            int filetype )
```

Definition at line 175 of file setfile.c.

4.96.3.3 GetSetupPar()

```
SETUPPARAMETERS * GetSetupPar (
            UBYTE * s )
```

Definition at line 330 of file setfile.c.

4.96.3.4 RecalcSetups()

```
int RecalcSetups ( )
```

Definition at line 350 of file setfile.c.

4.96.3.5 AllocSetups()

```
int AllocSetups ( )
```

Definition at line 397 of file setfile.c.

4.96.3.6 WriteSetup()

```
VOID WriteSetup ( )
```

Definition at line 780 of file setfile.c.

4.96.3.7 AllocSort()

```
SORTING * AllocSort (
            LONG LargeSize,
            LONG SmallSize,
            LONG SmallEsize,
            LONG TermsInSmall,
            int MaxPatches,
            int MaxFpatches,
            LONG IOsize )
```

Definition at line 835 of file setfile.c.

4.96.3.8 AllocSortFileName()

```
VOID AllocSortFileName (
            SORTING * sort )
```

Definition at line 968 of file setfile.c.

4.96.3.9 AllocFileHandle()

```
FILEHANDLE * AllocFileHandle (
            WORD par,
            char * name )
```

Definition at line 993 of file setfile.c.

4.96.3.10 DeAllocFileHandle()

```
void DeAllocFileHandle (
            FILEHANDLE * fh )
```

Definition at line 1053 of file setfile.c.

4.96.3.11 MakeSetupAllocs()

```
int MakeSetupAllocs ( )
```

Definition at line 1070 of file setfile.c.

4.96.3.12 TryFileSetups()

```
int TryFileSetups ( )
```

Definition at line 1090 of file setfile.c.

4.96.3.13 TryEnvironment()

```
int TryEnvironment ( )
```

Definition at line 1174 of file setfile.c.

4.96.4 Variable Documentation

4.96.4.1 curdirp

```
char curdirp[] = "."
```

Definition at line 39 of file setfile.c.

4.96.4.2 cursortdirp

```
char cursortdirp[] = "."
```

Definition at line 40 of file setfile.c.

4.96.4.3 commentchar

```
char commentchar[] = "*"
```

Definition at line 41 of file setfile.c.

4.96.4.4 dotchar

```
char dotchar[] = "_"
```

Definition at line 42 of file setfile.c.

4.96.4.5 highfirst

```
char highfirst[] = "highfirst"
```

Definition at line 43 of file setfile.c.

4.96.4.6 lowfirst

```
char lowfirst[] = "lowfirst"
```

Definition at line 44 of file setfile.c.

4.96.4.7 procedureextension

```
char procedureextension[] = "prc"
```

Definition at line 45 of file setfile.c.

4.96.4.8 setupparameters

```
SETUPPARAMETERS setupparameters[]
```

Definition at line 53 of file setfile.c.

4.97 setfile.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes :
00033
00034     Routines that deal with settings and the setup file
00035 */
00036
00037 #include "form3.h"
00038
00039 char curdirp[] = ".";
00040 char cursortdirp[] = ".";
00041 char commentchar[] = "*";
00042 char dotchar[] = "_";
00043 char highfirst[] = "highfirst";
00044 char lowfirst[] = "lowfirst";
00045 char procedureextension[] = "prc";
00046
00047 #define NUMERICALVALUE 0
00048 #define STRINGVALUE 1
00049 #define PATHVALUE 2
00050 #define ONOFFVALUE 3
00051 #define DEFINEVALUE 4
00052
00053 SETUPPARAMETERS setupparameters[] =
00054 {
00055      {(UBYTE *)"bracketindexsize",      NUMERICALVALUE, 0, (LONG)MAXBRACKETBUFFERSIZE}
00056     ,{(UBYTE *)"commentchar",              STRINGVALUE, 0, (LONG)commentchar}
00057     ,{(UBYTE *)"compresssize",          NUMERICALVALUE, 0, (LONG)COMPRESSBUFFER}
00058     ,{(UBYTE *)"constindex",            NUMERICALVALUE, 0, (LONG)NUMFIXED}
00059     ,{(UBYTE *)"continuationlines",     NUMERICALVALUE, 0, (LONG)FORTRANCONTINUATIONLINES}
00060     ,{(UBYTE *)"define",                   DEFINEVALUE, 0, (LONG)0}
00061     ,{(UBYTE *)"dotchar",                  STRINGVALUE, 0, (LONG)dotchar}
```

```
    ,{(UBYTE *)"factorizationcache",    NUMERICALVALUE, 0, (LONG)FBUFFERSIZE}
    ,{(UBYTE *)"filepatches",           NUMERICALVALUE, 0, (LONG)MAXFPATCHES}
    ,{(UBYTE *)"functionlevels",        NUMERICALVALUE, 0, (LONG)MAXFLEVELS}
    ,{(UBYTE *)"hidesize",              NUMERICALVALUE, 0, (LONG)0}
    ,{(UBYTE *)"incdir",                     PATHVALUE, 0, (LONG)curdirp}
    ,{(UBYTE *)"indentspace",           NUMERICALVALUE, 0, (LONG)INDENTSPACE}
    ,{(UBYTE *)"insidefirst",               ONOFFVALUE, 0, (LONG)1}
    ,{(UBYTE *)"largepatches",          NUMERICALVALUE, 0, (LONG)MAXPATCHES}
    ,{(UBYTE *)"largesize",             NUMERICALVALUE, 0, (LONG)LARGEBUFFER}
    ,{(UBYTE *)"maxnumbersize",         NUMERICALVALUE, 0, (LONG)0}
/*  ,{(UBYTE *)"maxnumbersize",         NUMERICALVALUE, 0, (LONG)MAXNUMBERSIZE} */
    ,{(UBYTE *)"maxtermsize",           NUMERICALVALUE, 0, (LONG)MAXTER}
    ,{(UBYTE *)"maxwildcards",          NUMERICALVALUE, 0, (LONG)MAXWILDC}
    ,{(UBYTE *)"nospacesinnumbers",         ONOFFVALUE, 0, (LONG)0}
    ,{(UBYTE *)"numstorecaches",        NUMERICALVALUE, 0, (LONG)NUMSTORECACHES}
    ,{(UBYTE *)"nwritefinalstatistics",     ONOFFVALUE, 0, (LONG)0}
    ,{(UBYTE *)"nwriteprocessstatistics",   ONOFFVALUE, 0, (LONG)0}
    ,{(UBYTE *)"nwritestatistics",          ONOFFVALUE, 0, (LONG)0}
    ,{(UBYTE *)"nwritethreadstatistics",    ONOFFVALUE, 0, (LONG)0}
    ,{(UBYTE *)"oldfactarg",                ONOFFVALUE, 0, (LONG)NEWFACTARG}
    ,{(UBYTE *)"oldgcd",                    ONOFFVALUE, 0, (LONG)1}
    ,{(UBYTE *)"oldorder",                  ONOFFVALUE, 0, (LONG)0}
    ,{(UBYTE *)"oldparallelstatistics",     ONOFFVALUE, 0, (LONG)0}
    ,{(UBYTE *)"parentheses",           NUMERICALVALUE, 0, (LONG)MAXPARLEVEL}
    ,{(UBYTE *)"path",                       PATHVALUE, 0, (LONG)curdirp}
    ,{(UBYTE *)"procedureextension",       STRINGVALUE, 0, (LONG)procedureextension}
    ,{(UBYTE *)"processbucketsize",     NUMERICALVALUE, 0, (LONG)DEFAULTPROCESSBUCKETSIZE}
    ,{(UBYTE *)"resettimeonclear",          ONOFFVALUE, 0, (LONG)1}
    ,{(UBYTE *)"scratchsize",           NUMERICALVALUE, 0, (LONG)SCRATCHSIZE}
    ,{(UBYTE *)"shmwinsize",            NUMERICALVALUE, 0, (LONG)SHMWINSIZE}
    ,{(UBYTE *)"sizestorecache",        NUMERICALVALUE, 0, (LONG)SIZESTORECACHE}
    ,{(UBYTE *)"smallextension",        NUMERICALVALUE, 0, (LONG)SMALLOVERFLOW}
    ,{(UBYTE *)"smallsize",             NUMERICALVALUE, 0, (LONG)SMALLBUFFER}
    ,{(UBYTE *)"sortiosize",            NUMERICALVALUE, 0, (LONG)SORTIOSIZE}
    ,{(UBYTE *)"sorttype",                 STRINGVALUE, 0, (LONG)lowfirst}
    ,{(UBYTE *)"spectatorsize",         NUMERICALVALUE, 0, (LONG)SPECTATORSIZE}
    ,{(UBYTE *)"subfilepatches",        NUMERICALVALUE, 0, (LONG)SMAXFPATCHES}
    ,{(UBYTE *)"sublargepatches",       NUMERICALVALUE, 0, (LONG)SMAXPATCHES}
    ,{(UBYTE *)"sublargesize",          NUMERICALVALUE, 0, (LONG)SLARGEBUFFER}
    ,{(UBYTE *)"subsmallextension",     NUMERICALVALUE, 0, (LONG)SSMALLOVERFLOW}
    ,{(UBYTE *)"subsmallsize",          NUMERICALVALUE, 0, (LONG)SSMALLBUFFER}
    ,{(UBYTE *)"subsortiosize",         NUMERICALVALUE, 0, (LONG)SSORTIOSIZE}
    ,{(UBYTE *)"subtermsinsmall",       NUMERICALVALUE, 0, (LONG)STERMSSMALL}
    ,{(UBYTE *)"tempdir",                  STRINGVALUE, 0, (LONG)curdirp}
    ,{(UBYTE *)"tempsortdir",              STRINGVALUE, 0, (LONG)cursortdirp}
    ,{(UBYTE *)"termsinsmall",          NUMERICALVALUE, 0, (LONG)TERMSSMALL}
    ,{(UBYTE *)"threadbucketsize",      NUMERICALVALUE, 0, (LONG)DEFAULTTHREADBUCKETSIZE}
    ,{(UBYTE *)"threadloadbalancing",       ONOFFVALUE, 0, (LONG)DEFAULTTHREADLOADBALANCING}
    ,{(UBYTE *)"threads",               NUMERICALVALUE, 0, (LONG)DEFAULTTHREADS}
    ,{(UBYTE *)"threadscratchoutsize",  NUMERICALVALUE, 0, (LONG)THREADSCRATCHOUTSIZE}
    ,{(UBYTE *)"threadscratchsize",     NUMERICALVALUE, 0, (LONG)THREADSCRATCHSIZE}
    ,{(UBYTE *)"threadsortfilesynch",       ONOFFVALUE, 0, (LONG)0}
    ,{(UBYTE *)"totalsize",                 ONOFFVALUE, 0, (LONG)2}
    ,{(UBYTE *)"workspace",             NUMERICALVALUE, 0, (LONG)WORKBUFFER}
    ,{(UBYTE *)"wtimestats",                ONOFFVALUE, 0, (LONG)2}
};

/*
    #] Includes :
    #[ Setups :
        #[ DoSetups :
*/

int DoSetups()
{
    UBYTE *setbuffer, *s, *t, *u /*, c */;
    int errors = 0;
    setbuffer = LoadInputFile((UBYTE *)setupfilename,SETUPFILE);
    if ( setbuffer ) {
/*
        The contents of the file are now in setbuffer.
        Each line is commentary or a single command.
        The buffer is terminated with a zero.
*/
        s = setbuffer;
        while ( *s ) {
            if ( *s == ' ' || *s == '\t' || *s == '*' || *s == '#' || *s == '\n' ) {
                while ( *s && *s != '\n' ) s++;
            }
            else if ( tolower(*s) < 'a' || tolower(*s) > 'z' ) {
                t = s;
                while ( *s && *s != '\n' ) s++;
/*
                c = *s; *s = 0;
                Error1("Setup file: Illegal statement: ",t);
                errors++; *s = c;
*/
```

```
            }
            else {
                t = s; /* name of the option */
                while ( tolower(*s) >= 'a' && tolower(*s) <= 'z' ) s++;
                *s++ = 0;
                while ( *s == ' ' || *s == '\t' ) s++;
                u = s; /* 'value' of the option */
                while ( *s && *s != '\n' && *s != '\r' ) s++;
                if ( *s ) *s++ = 0;
                errors += ProcessOption(t,u,0);
            }
            while ( *s == '\n' || *s == '\r' ) s++;
        }
        M_free(setbuffer,"setup file buffer");
    }
    if ( errors ) return(1);
    else return(0);
}

/*
        #] DoSetups :
        #[ ProcessOption :
*/

static char *proop1[3] = { "Setup file", "Setups in .frm file", "Setup in environment" };

int ProcessOption(UBYTE *s1, UBYTE *s2, int filetype)
{
    SETUPPARAMETERS *sp;
    int n, giveback = 0, error = 0;
    UBYTE *s, *t, *s2ret;
    LONG x;
    sp = GetSetupPar(s1);
    if ( sp ) {
/*
        We check now whether there are `' variables to be looked up in the
        environment. This is new (30-may-2008). This is only allowed in s2.
*/
restart:;
        {
            UBYTE *s3,*s4,*s5,*s6, c, *start;
            int n1,n2,n3;
            s = s2;
            while ( *s ) {
                if ( *s == '\\' ) s += 2;
                else if ( *s == '`' ) {
                    start = s; s++;
                    while ( *s && *s != '\'' ) {
                        if ( *s == '\\' ) s++;
                        s++;
                    }
                    if ( *s == 0 ) {
                        MesPrint("%s: Illegal use of ` character for parameter %s"
                        ,proop1[filetype],s1);
                        return(1);
                    }
                    c = *s; *s = 0;
                    s3 = (UBYTE *)getenv((char *)(start+1));
                    if ( s3 == 0 ) {
                        MesPrint("%s: Cannot find environment variable %s for parameter %s"
                        ,proop1[filetype],start+1,s1);
                        return(1);

                    }
                    *s = c; s++;
                    n1 = start - s2; s4 = s3; n2 = 0;
                    while ( *s4 ) {
                        if ( *s4 == '\\' ) { s4++; n2++; }
                        s4++; n2++;
                    }
                    s4 = s; n3 = 0;
                    while ( *s4 ) {
                        if ( *s4 == '\\' ) { s4++; n3++; }
                        s4++; n3++;
                    }
                    s4 = (UBYTE *)Malloc1((n1+n2+n3+1)*sizeof(UBYTE),"environment in setup");
                    s5 = s2; s6 = s4;
                    while ( n1-- > 0 ) *s6++ = *s5++;
                    s5 = s3;
                    while ( n2-- > 0 ) *s6++ = *s5++;
                    s5 = s;
                    while ( n3-- > 0 ) *s6++ = *s5++;
                    *s6 = 0;
                    if ( giveback ) M_free(s2,"environment in setup");
                    s2 = s4;
                    giveback = 1;
                    goto restart;
```

```
                }
                else s++;
            }
        }
        n = sp->type;
        s2ret = s2;
        switch ( n ) {
            case NUMERICALVALUE:
                    ParseNumber(x,s2);
                if ( *s2 == 'K' ) { x = x * 1000; s2++; }
                else if ( *s2 == 'M' ) { x = x * 1000000; s2++; }
                else if ( *s2 == 'G' ) { x = x * 1000000000; s2++; }
                else if ( *s2 == 'T' ) { x = x * 1000000000000; s2++; }
                if ( *s2 && *s2 != ' ' && *s2 != '\t' ) {
                    MesPrint("%s: Numerical value expected for parameter %s"
                    ,proop1[filetype],s1);
                    error = 1; break;
                }
                sp->value = x;
                sp->flags = USEDFLAG;
                break;
            case STRINGVALUE:
                if ( StrICmp(s1,(UBYTE *)"tempsortdir") == 0 ) AM.havesortdir = 1;
                s = s2; t = s2;
                while ( *s ) {
                    if ( *s == ' ' || *s == '\t' ) break;
                    if ( *s == '\\' ) s++;
                    *t++ = *s++;
                }
                *t = 0;
                if ( sp->flags == USEDFLAG && sp->value != 0 )
                        M_free((VOID *)(sp->value),"Process option");
                sp->value = (LONG)strDup1(s2,"Process option");
                sp->flags = USEDFLAG;
                break;
            case PATHVALUE:
                if ( StrICmp(s1,(UBYTE *)"incdir") == 0 ) {
                    AM.IncDir = 0;
                }
                else if ( StrICmp(s1,(UBYTE *)"path") == 0 ) {
                    if ( AM.Path ) M_free(AM.Path,"path");
                    AM.Path = 0;
                }
                else {
                    MesPrint("Setups: %s not yet implemented",s1);
                    error = 1;
                    break;
                }
                if ( sp->flags == USEDFLAG && sp->value != 0 )
                    M_free((VOID *)(sp->value),"Process option");
                sp->value = (LONG)strDup1(s2,"Process option");
                sp->flags = USEDFLAG;
                break;
            case ONOFFVALUE:
                if ( tolower(*s2) == 'o' && tolower(s2[1]) == 'n'
                && ( s2[2] == 0 || s2[2] == ' ' || s2[2] == '\t' ) )
                    sp->value = 1;
                else if ( tolower(*s2) == 'o' && tolower(s2[1]) == 'f'
                && tolower(s2[2]) == 'f'
                && ( s2[3] == 0 || s2[3] == ' ' || s2[3] == '\t' ) )
                    sp->value = 0;
                else {
                    MesPrint("%s: Unrecognized option for parameter %s: %s"
                    ,proop1[filetype],s1,s2);
                    error = 1; break;
                }
                sp->flags = USEDFLAG;
                break;
            case DEFINEVALUE:
/*
                if ( sp->value ) M_free((UBYTE *)(sp->value),"Process option");
                sp->value = (LONG)strDup1(s2,"Process option");
*/
                if ( TheDefine(s2,2) ) error = 1;
                break;
            default:
                Error1("Error in setupparameter table for:",s1);
                error = 1;
                break;
        }
    }
    else {
        MesPrint("%s: Keyword not recognized: %s",proop1[filetype],s1);
        error = 1;
    }
    if ( giveback ) M_free(s2ret,"environment in setup");
    return(error);
```

```
}

/*
        #] ProcessOption :
        #[ GetSetupPar :
*/

SETUPPARAMETERS *GetSetupPar(UBYTE *s)
{
    int hi, med, lo, i;
    lo = 0;
    hi = sizeof(setupparameters)/sizeof(SETUPPARAMETERS);
    do {
        med = ( hi + lo ) / 2;
        i = StrICmp(s,(UBYTE *)setupparameters[med].parameter);
        if ( i == 0 ) return(setupparameters+med);
        if ( i < 0 ) hi = med-1;
        else         lo = med+1;
    } while ( hi >= lo );
    return(0);
}

/*
        #] GetSetupPar :
        #[ RecalcSetups :
*/

int RecalcSetups()
{
    SETUPPARAMETERS *sp, *sp1;

    sp1 = GetSetupPar((UBYTE *)"threads");
    if ( AM.totalnumberofthreads > 1 ) sp1->value = AM.totalnumberofthreads - 1;
    else sp1->value = 0;
/*
    if ( sp1->value > 0 ) AM.totalnumberofthreads = sp1->value+1;
    if ( AM.totalnumberofthreads == 0 ) AM.totalnumberofthreads = 1;
*/
    sp  = GetSetupPar((UBYTE *)"filepatches");
    if ( sp->value < AM.totalnumberofthreads-1 )
        sp->value = AM.totalnumberofthreads - 1;

    sp  = GetSetupPar((UBYTE *)"smallsize");
    sp1 = GetSetupPar((UBYTE *)"smallextension");
    if ( 6*sp1->value < 7*sp->value ) sp1->value = (7*sp->value)/6;
    sp = GetSetupPar((UBYTE *)"termsinsmall");
    sp->value = ( sp->value + 15 ) & (-16L);
#ifdef WITHPTHREADS
    {
    SETUPPARAMETERS *sp2;
    LONG totalsize, minimumsize;
    sp = GetSetupPar((UBYTE *)"largesize");
    totalsize = sp1->value+sp->value;
    sp2 = GetSetupPar((UBYTE *)"maxtermsize");
    AM.MaxTer = sp2->value*sizeof(WORD);
    if ( AM.MaxTer < 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = 200*(LONG)(sizeof(WORD));
    if ( AM.MaxTer > MAXPOSITIVE - 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = MAXPOSITIVE -
200*(LONG)(sizeof(WORD));
    AM.MaxTer /= sizeof(WORD);
    AM.MaxTer *= sizeof(WORD);
    minimumsize = (AM.totalnumberofthreads-1)*(AM.MaxTer+
        NUMBEROFBLOCKSINSORT*MINIMUMNUMBEROFTERMS*AM.MaxTer);
    if ( totalsize < minimumsize ) {
        sp->value = minimumsize - sp1->value;
    }
    }
#endif
    return(0);
}

/*
        #] RecalcSetups :
        #[ AllocSetups :
*/

int AllocSetups()
{
    SETUPPARAMETERS *sp;
    LONG LargeSize, SmallSize, SmallEsize, TermsInSmall, IOsize;
    int MaxPatches, MaxFpatches, error = 0, i, size;
    UBYTE *s;
#ifndef WITHPTHREADS
    int j;
#endif
    sp = GetSetupPar((UBYTE *)"threads");
    if ( sp->value > 0 ) AM.totalnumberofthreads = sp->value+1;

```

```
    AM.OutBuffer = (UBYTE *)Malloc1(AM.OutBufSize+1,"OutputBuffer");
    AP.PreAssignStack =(LONG *)Malloc1(AP.MaxPreAssignLevel*sizeof(LONG *),"PreAssignStack");
    for ( i = 0; i < AP.MaxPreAssignLevel; i++ ) AP.PreAssignStack[i] = 0;
    AC.iBuffer = (UBYTE *)Malloc1(AC.iBufferSize+1,"statement buffer");
    AC.iStop = AC.iBuffer + AC.iBufferSize-2;
    AP.preStart = (UBYTE *)Malloc1(AP.pSize,"instruction buffer");
    AP.preStop = AP.preStart + AP.pSize - 3;
    /* AP.PreIfStack is already allocated in StartPrepro(), but to be sure we
       "if" the freeing */
    if ( AP.PreIfStack ) M_free(AP.PreIfStack,"PreIfStack");
    AP.PreIfStack = (int *)Malloc1(AP.MaxPreIfLevel*sizeof(int),
                "Preprocessor if stack");
    AP.PreIfStack[0] = EXECUTINGIF;
    sp = GetSetupPar((UBYTE *)"insidefirst");
    AM.ginsidefirst = AC.minsidefirst = AC.insidefirst = sp->value;
/*
    We need to consider eliminating this variable
*/
    sp = GetSetupPar((UBYTE *)"maxtermsize");
    AM.MaxTer = sp->value*sizeof(WORD);
    if ( AM.MaxTer < 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = 200*(LONG)(sizeof(WORD));
    if ( AM.MaxTer > MAXPOSITIVE - 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = MAXPOSITIVE -
200*(LONG)(sizeof(WORD));
    AM.MaxTer /= (LONG)sizeof(WORD);
    AM.MaxTer *= (LONG)sizeof(WORD);
/*
    Allocate workspace.
*/
    sp = GetSetupPar((UBYTE *)"workspace");
    AM.WorkSize = sp->value;
#ifdef WITHPTHREADS
#else
    AT.WorkSpace = (WORD *)Malloc1(AM.WorkSize*sizeof(WORD),(char *)(sp->parameter));
    AT.WorkTop = AT.WorkSpace + AM.WorkSize;
    AT.WorkPointer = AT.WorkSpace;
#endif
/*
    Fixed indices
*/
    sp = GetSetupPar((UBYTE *)"constindex");
    if ( ( sp->value+100+5*WILDOFFSET ) > MAXPOSITIVE ) {
        MesPrint("Setting of %s in setupfile too large","constindex");
        AM.OffsetIndex = MAXPOSITIVE - 5*WILDOFFSET - 100;
        MesPrint("value corrected to maximum allowed: %d",AM.OffsetIndex);
    }
    else AM.OffsetIndex = sp->value + 1;
    AC.FixIndices = (WORD *)Malloc1((AM.OffsetIndex)*sizeof(WORD),(char *)(sp->parameter));
    AM.WilInd = AM.OffsetIndex + WILDOFFSET;
    AM.DumInd = AM.OffsetIndex + 2*WILDOFFSET;
    AM.IndDum = AM.DumInd + WILDOFFSET;
#ifndef WITHPTHREADS
    AR.CurDum = AN.IndDum = AM.IndDum;
#endif
    AM.mTraceDum = AM.IndDum + 2*WILDOFFSET;

    sp = GetSetupPar((UBYTE *)"parentheses");
    AM.MaxParLevel = sp->value+1;
    AC.tokenarglevel = (WORD *)Malloc1((sp->value+1)*sizeof(WORD),(char *)(sp->parameter));
/*
    Space during calculations
*/
    sp = GetSetupPar((UBYTE *)"maxnumbersize");
/*
    size = ( sp->value + 11 ) & (-4);
    AM.MaxTal = size - 2;
    if ( AM.MaxTal > (AM.MaxTer/sizeof(WORD)-2)/2 )
                AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/2;
    if ( AM.MaxTal < (AM.MaxTer/sizeof(WORD)-2)/4 )
                AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/4;
*/
/*
    There is too much confusion about MaxTal cq maxnumbersize.
    It seems better to fix it at its maximum value. This way we only worry
    about maxtermsize. This can be understood better by the 'innocent' user.
*/
    if ( sp->value == 0 ) {
        AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/2;
    }
    else {
        size = ( sp->value + 11 ) & (-4);
        AM.MaxTal = size - 2;
        if ( (size_t)AM.MaxTal > (size_t)((AM.MaxTer/sizeof(WORD)-2)/2) )
                    AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/2;
    }
    AM.MaxTal &= -sizeof(WORD)*2;

    sp->value = AM.MaxTal;
```

```
    AC.cmod = (UWORD *)Malloc1(AM.MaxTal*4*sizeof(UWORD),(char *)(sp->parameter));
    AM.gcmod = AC.cmod + AM.MaxTal;
    AC.powmod = AM.gcmod + AM.MaxTal;
    AM.gpowmod = AC.powmod + AM.MaxTal;
/*
    The IO buffers for the input and output expressions.
    Fscr[2] will be assigned in a later stage for hiding expressions from
    the regular action. That will make the program faster.
*/
    sp = GetSetupPar((UBYTE *)"scratchsize");
    AM.ScratSize = sp->value/sizeof(WORD);
    if ( AM.ScratSize < 4*AM.MaxTer ) AM.ScratSize = 4*AM.MaxTer;
    AM.HideSize = AM.ScratSize;
    sp = GetSetupPar((UBYTE *)"hidesize");
    if ( sp->value > 0 ) {
        AM.HideSize = sp->value/sizeof(WORD);
        if ( AM.HideSize < 4*AM.MaxTer ) AM.HideSize = 4*AM.MaxTer;
    }
    sp = GetSetupPar((UBYTE *)"factorizationcache");
    AM.fbuffersize = sp->value;
#ifdef WITHPTHREADS
    sp = GetSetupPar((UBYTE *)"threadscratchsize");
    AM.ThreadScratSize = sp->value/sizeof(WORD);
    sp = GetSetupPar((UBYTE *)"threadscratchoutsize");
    AM.ThreadScratOutSize = sp->value/sizeof(WORD);
#endif
#ifndef WITHPTHREADS
    for ( j = 0; j < 2; j++ ) {
        WORD *ScratchBuf;
        ScratchBuf = (WORD *)Malloc1(AM.ScratSize*sizeof(WORD),"scratchsize");
        AR.Fscr[j].POsize = AM.ScratSize * sizeof(WORD);
        AR.Fscr[j].POfull = AR.Fscr[j].POfill = AR.Fscr[j].PObuffer = ScratchBuf;
        AR.Fscr[j].POstop = AR.Fscr[j].PObuffer + AM.ScratSize;
        PUTZERO(AR.Fscr[j].POposition);
    }
    AR.Fscr[2].PObuffer = 0;
#endif
    sp = GetSetupPar((UBYTE *)"threadbucketsize");
    AC.ThreadBucketSize = AM.gThreadBucketSize = AM.ggThreadBucketSize = sp->value;
    sp = GetSetupPar((UBYTE *)"threadloadbalancing");
    AC.ThreadBalancing = AM.gThreadBalancing = AM.ggThreadBalancing = sp->value;
    sp = GetSetupPar((UBYTE *)"threadsortfilesynch");
    AC.ThreadSortFileSynch = AM.gThreadSortFileSynch = AM.ggThreadSortFileSynch = sp->value;
/*
     The size for shared memory window for oneside MPI2 communications
*/
    sp = GetSetupPar((UBYTE *)"shmwinsize");
    AM.shmWinSize = sp->value/sizeof(WORD);
    if ( AM.shmWinSize < 4*AM.MaxTer ) AM.shmWinSize = 4*AM.MaxTer;
/*
    The sort buffer
*/
    sp = GetSetupPar((UBYTE *)"smallsize");
    SmallSize = sp->value;
    sp = GetSetupPar((UBYTE *)"smallextension");
    SmallEsize = sp->value;
    sp = GetSetupPar((UBYTE *)"largesize");
    LargeSize = sp->value;
    sp = GetSetupPar((UBYTE *)"termsinsmall");
    TermsInSmall = sp->value;
    sp = GetSetupPar((UBYTE *)"largepatches");
    MaxPatches = sp->value;
    sp = GetSetupPar((UBYTE *)"filepatches");
    MaxFpatches = sp->value;
    sp = GetSetupPar((UBYTE *)"sortiosize");
    IOsize = sp->value;
    if ( IOsize < AM.MaxTer ) { IOsize = AM.MaxTer; sp->value = IOsize; }
#ifndef WITHPTHREADS
#ifdef WITHZLIB
    for ( j = 0; j < 2; j++ ) { AR.Fscr[j].ziosize = IOsize; }
#endif
#endif
    AM.S0 = 0;
    AM.S0 = AllocSort(LargeSize,SmallSize,SmallEsize,TermsInSmall
                      ,MaxPatches,MaxFpatches,IOsize);
#ifdef WITHZLIB
    AM.S0->file.ziosize = IOsize;
#ifndef WITHPTHREADS
    AR.FoStage4[0].ziosize = IOsize;
    AR.FoStage4[1].ziosize = IOsize;
    AT.S0 = AM.S0;
#endif
#else
#ifndef WITHPTHREADS
    AT.S0 = AM.S0;
#endif
#endif
```

```
#ifndef WITHPTHREADS
    AR.FoStage4[0].POsize   = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
    AR.FoStage4[1].POsize   = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
#endif
    sp = GetSetupPar((UBYTE *)"subsmallsize");
    AM.SSmallSize = sp->value;
    sp = GetSetupPar((UBYTE *)"subsmallextension");
    AM.SSmallEsize = sp->value;
    sp = GetSetupPar((UBYTE *)"sublargesize");
    AM.SLargeSize = sp->value;
    sp = GetSetupPar((UBYTE *)"subtermsinsmall");
    AM.STermsInSmall = sp->value;
    sp = GetSetupPar((UBYTE *)"sublargepatches");
    AM.SMaxPatches = sp->value;
    sp = GetSetupPar((UBYTE *)"subfilepatches");
    AM.SMaxFpatches = sp->value;
    sp = GetSetupPar((UBYTE *)"subsortiosize");
    AM.SIOsize = sp->value;
    sp = GetSetupPar((UBYTE *)"spectatorsize");
    AM.SpectatorSize = sp->value;
/*
    The next code is just for the moment (26-jan-1997) because we have
    the new parts combined with the old. Once the old parts are gone
    from the program, we can eliminate this code too.
*/
    sp = GetSetupPar((UBYTE *)"functionlevels");
    AM.maxFlevels = sp->value + 1;
#ifdef WITHPTHREADS
#else
    AT.Nest = (NESTING)Malloc1((LONG)sizeof(struct NeStInG)*AM.maxFlevels,"functionlevels");
    AT.NestStop = AT.Nest + AM.maxFlevels;
    AT.NestPoin = AT.Nest;
#endif

    sp = GetSetupPar((UBYTE *)"maxwildcards");
    AM.MaxWildcards = sp->value;
#ifdef WITHPTHREADS
#else
    AT.WildMask = (WORD *)Malloc1((LONG)AM.MaxWildcards*sizeof(WORD),"maxwildcards");
#endif

    sp = GetSetupPar((UBYTE *)"compresssize");
    if ( sp->value < 2*AM.MaxTer ) sp->value = 2*AM.MaxTer;
    AM.CompressSize = sp->value;
#ifndef WITHPTHREADS
    AR.CompressBuffer = (WORD *)Malloc1((AM.CompressSize+10)*sizeof(WORD),"compresssize");
    AR.CompressPointer = AR.CompressBuffer;
    AR.ComprTop = AR.CompressBuffer + AM.CompressSize;
#endif
    sp = GetSetupPar((UBYTE *)"bracketindexsize");
    if ( sp->value < 20*AM.MaxTer ) sp->value = 20*AM.MaxTer;
    AM.MaxBracketBufferSize = sp->value/sizeof(WORD);

    sp = GetSetupPar((UBYTE *)"dotchar");
    AO.FortDotChar = ((UBYTE *)(sp->value))[0];
    sp = GetSetupPar((UBYTE *)"commentchar");
    AP.cComChar = AP.ComChar = ((UBYTE *)(sp->value))[0];
    sp = GetSetupPar((UBYTE *)"procedureextension");
/*
    Check validity first.
*/
    s = (UBYTE *)(sp->value);
    if ( FG.cTable[*s] != 0 ) {
        MesPrint("  Illegal string for procedure extension %s",(UBYTE *)sp->value);
        error = -2;
    }
    else {
        s++;
        while ( *s ) {
            if ( *s == ' ' || *s == '\t' || *s == '\n' ) {
                MesPrint("  Illegal string for procedure extension %s",(UBYTE *)sp->value);
                error = -2;
                break;
            }
            s++;
        }
    }
    AP.cprocedureExtension = strDup1((UBYTE *)(sp->value),"procedureExtension");
    AP.procedureExtension = strDup1(AP.cprocedureExtension,"procedureExtension");

    sp = GetSetupPar((UBYTE *)"totalsize");
    if ( sp->value != 2 ) AM.PrintTotalSize = sp->value;

    sp = GetSetupPar((UBYTE *)"continuationlines");
    AM.FortranCont = sp->value;
    if ( AM.FortranCont <= 0 ) AM.FortranCont = 1;
    sp = GetSetupPar((UBYTE *)"oldorder");
```

```
    AM.OldOrderFlag = sp->value;
    sp = GetSetupPar((UBYTE *)"resettimeonclear");
    AM.resetTimeOnClear = sp->value;
    sp = GetSetupPar((UBYTE *)"nospacesinnumbers");
    AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers = AM.ggNoSpacesInNumbers = sp->value;
    sp = GetSetupPar((UBYTE *)"indentspace");
    AO.IndentSpace = AM.gIndentSpace = AM.ggIndentSpace = sp->value;
    sp = GetSetupPar((UBYTE *)"nwritestatistics");
    AC.StatsFlag = AM.gStatsFlag = AM.ggStatsFlag = 1-sp->value;
    sp = GetSetupPar((UBYTE *)"nwritefinalstatistics");
    AC.FinalStats = AM.gFinalStats = AM.ggFinalStats = 1-sp->value;
    sp = GetSetupPar((UBYTE *)"nwritethreadstatistics");
    AC.ThreadStats = AM.gThreadStats = AM.ggThreadStats = 1-sp->value;
    sp = GetSetupPar((UBYTE *)"nwriteprocessstatistics");
    AC.ProcessStats = AM.gProcessStats = AM.ggProcessStats = 1-sp->value;
    sp = GetSetupPar((UBYTE *)"oldparallelstatistics");
    AC.OldParallelStats = AM.gOldParallelStats = AM.ggOldParallelStats = sp->value;
    sp = GetSetupPar((UBYTE *)"oldfactarg");
    AC.OldFactArgFlag = AM.gOldFactArgFlag = AM.ggOldFactArgFlag = sp->value;
    sp = GetSetupPar((UBYTE *)"oldgcd");
    AC.OldGCDflag = AM.gOldGCDflag = AM.ggOldGCDflag = sp->value;
    sp = GetSetupPar((UBYTE *)"wtimestats");
    if ( sp->value == 2 ) sp->value = AM.ggWTimeStatsFlag;
    AC.WTimeStatsFlag = AM.gWTimeStatsFlag = AM.ggWTimeStatsFlag = sp->value;
    sp = GetSetupPar((UBYTE *)"sorttype");
    if ( StrICmp((UBYTE *)"lowfirst",(UBYTE *)sp->value) == 0 ) {
        AC.lSortType = SORTLOWFIRST;
    }
    else if ( StrICmp((UBYTE *)"highfirst",(UBYTE *)sp->value) == 0 ) {
        AC.lSortType = SORTHIGHFIRST;
    }
    else {
        MesPrint("  Illegal SortType specification: %s",(UBYTE *)sp->value);
        error = -2;
    }

    sp = GetSetupPar((UBYTE *)"processbucketsize");
    AM.hProcessBucketSize = AM.gProcessBucketSize =
    AC.ProcessBucketSize = AC.mProcessBucketSize = sp->value;
/*
    The store caches (code installed 15-aug-2006 JV)
*/
    sp = GetSetupPar((UBYTE *)"numstorecaches");
    AM.NumStoreCaches = sp->value;
    sp = GetSetupPar((UBYTE *)"sizestorecache");
    AM.SizeStoreCache = sp->value;
#ifndef WITHPTHREADS
/*
    Install the store caches (15-aug-2006 JV)
    Note that in the case of PTHREADS this is done in InitializeOneThread
*/
    AT.StoreCache = AT.StoreCacheAlloc = 0;
    if ( AM.NumStoreCaches > 0 ) {
        STORECACHE sa, sb;
        size = sizeof(struct StOrEcAcHe)+AM.SizeStoreCache;
        size = ((size-1)/sizeof(size_t)+1)*sizeof(size_t);
        AT.StoreCacheAlloc = (STORECACHE)Malloc1(size*AM.NumStoreCaches,"StoreCaches");
        AT.StoreCache = AT.StoreCacheAlloc;
        sa = AT.StoreCache;
        for ( j = 0; j < AM.NumStoreCaches; j++ ) {
            sb = (STORECACHE)(VOID *)((UBYTE *)sa+size);
            if ( j == AM.NumStoreCaches-1 ) {
                sa->next = 0;
            }
            else {
                sa->next = sb;
            }
            SETBASEPOSITION(sa->position,-1);
            SETBASEPOSITION(sa->toppos,-1);
            sa = sb;
        }
    }
#endif

/*
    And now some order sensitive things
*/
    if ( AM.Path == 0 ) {
        sp = GetSetupPar((UBYTE *)"path");
        AM.Path = strDup1((UBYTE *)(sp->value),"path");
    }
    if ( AM.IncDir == 0 ) {
        sp = GetSetupPar((UBYTE *)"incdir");
        AM.IncDir = strDup1((UBYTE *)(sp->value),"incdir");
    }
/*
    if ( AM.TempDir == 0 ) {
```

```
        sp = GetSetupPar((UBYTE *)"tempdir");
        AM.TempDir = strDup1((UBYTE *)(sp->value),"tempdir");
    }
*/
    return(error);
}

/*
        #] AllocSetups :
        #[ WriteSetup :

    The routine writes the values of the setup parameters.
    We should do this better. (JV, 21-may-2008)
    The way it should be done is:
        a: write the raw values.
        b: give readjusted values.
        c: give derived values.
    Because this is a difficult subject, it would be nice to have a LaTeX
    document that explains this all exactly. There should then be a
    mechanism to poke the values of the setup into the LaTeX document.
    probably the easiest way is to make a file with lots of \def definitions
    and have that included into the LaTeX file.
*/

VOID WriteSetup()
{
    int n = sizeof(setupparameters)/sizeof(SETUPPARAMETERS);
    SETUPPARAMETERS *sp;
    MesPrint(" The setup parameters are:");
    for ( sp = setupparameters; n > 0; n--, sp++ ) {
        switch(sp->type){
            case NUMERICALVALUE:
                MesPrint("   %s: %l",sp->parameter,sp->value);
                break;
            case PATHVALUE:
                if ( StrICmp(sp->parameter,(UBYTE *)"path") == 0 && AM.Path ) {
                    MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(AM.Path));
                    break;
                }
                if ( StrICmp(sp->parameter,(UBYTE *)"incdir") == 0 && AM.IncDir ) {
                    MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(AM.IncDir));
                    break;
                }
            case STRINGVALUE:
                if ( StrICmp(sp->parameter,(UBYTE *)"tempdir") == 0 && AM.TempDir ) {
                    MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(AM.TempDir));
                }
                else if ( StrICmp(sp->parameter,(UBYTE *)"tempsortdir") == 0 && AM.TempSortDir ) {
                    MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(AM.TempSortDir));
                }
                else {
                    MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(sp->value));
                }
                break;
            case ONOFFVALUE:
                if ( sp->value == 0 )
                    MesPrint("   %s: OFF",sp->parameter);
                else if ( sp->value == 1 )
                    MesPrint("   %s: ON",sp->parameter);
                break;
            case DEFINEVALUE:
/*
                MesPrint("   %s: '%s'",sp->parameter,(UBYTE *)(sp->value));
*/
                break;
        }
    }
    AC.SetupFlag = 0;
}

/*
        #] WriteSetup :
        #[ AllocSort :

        Routine allocates a complete struct for sorting.
        To be used for the main allocation of the sort buffers, and
        in a later stage for the function and subroutine sort buffers.
*/

SORTING *AllocSort(LONG LargeSize, LONG SmallSize, LONG SmallEsize, LONG TermsInSmall,
                   int MaxPatches, int MaxFpatches, LONG IOsize)
{
    LONG allocation,longer,terms2insmall,sortsize,longerp;
    LONG IObuffersize = IOsize;
    LONG IOtry;
    SORTING *sort;
    int i = 0, j = 0;
```

```
    char *s;
    if ( AM.S0 != 0 ) {
        s = FG.fname2; i = 0;
        while ( *s ) { s++; i++; }
        i += 16;
    }
    if ( MaxFpatches < 4 ) MaxFpatches = 4;
    longer = MaxPatches > MaxFpatches ? MaxPatches : MaxFpatches;
    longerp = longer;
    while ( (1 « j) < longerp ) j++;
    longerp = (1 « j) + 1;
    longerp += sizeof(WORD*) - (longerp%sizeof(WORD *));
    longer++;
    longer += sizeof(WORD*) - (longer%sizeof(WORD *));
    if ( SmallSize < 16*AM.MaxTer ) SmallSize = 16*AM.MaxTer+16;
    TermsInSmall = (TermsInSmall+15) & (-16L);
    terms2insmall = 2*TermsInSmall;  /* Used to be just + 100 rather than *2 */
    if ( SmallEsize < (SmallSize*3)/2 ) SmallEsize = (SmallSize*3)/2;
    if ( LargeSize > 0 && LargeSize < 2*SmallSize ) LargeSize = 2*SmallSize;
/*  if ( SmallEsize < 3*AM.MaxTer ) SmallEsize = 3*AM.MaxTer; */
    SmallEsize = (SmallEsize+15) & (-16L);
    if ( LargeSize < 0 ) LargeSize = 0;
    sortsize = sizeof(SORTING);
    sortsize = (sortsize+15)&(-16L);
    IObuffersize = (IObuffersize+sizeof(WORD)-1)/sizeof(WORD);
/*
    The next statement fixes a bug. In the rare case that we have a
    problem here, we expand the size of the large buffer or the
    small extension
*/
    if ( (ULONG)( LargeSize+SmallEsize ) < MaxFpatches*((IObuffersize
        +COMPINC)*sizeof(WORD)+2*AM.MaxTer) ) {
        if ( LargeSize == 0 )
            SmallEsize = MaxFpatches*((IObuffersize+COMPINC)*sizeof(WORD)+2*AM.MaxTer);
        else
            LargeSize  = MaxFpatches*((IObuffersize+COMPINC)*sizeof(WORD)+2*AM.MaxTer)
                - SmallEsize;
    }

    IOtry = ((LargeSize+SmallEsize)/MaxFpatches-2*AM.MaxTer)/sizeof(WORD)-COMPINC;

    if ( (LONG)(IObuffersize*sizeof(WORD)) < IOtry )
        IObuffersize = (IOtry+sizeof(WORD)-1)/sizeof(WORD);

    allocation =
         3*sizeof(POSITION)*(LONG)longer                /* Filepositions!! */
        +2*sizeof(WORD *)*longer
        +2*(longerp*(sizeof(WORD *)+sizeof(WORD)))
        +(3*longerp+2)*sizeof(WORD)
#ifdef WITHZLIB
        +(2*longerp+4)*sizeof(WORD)
#endif
        +terms2insmall*sizeof(WORD *)
        +terms2insmall*sizeof(WORD *)/2
        +LargeSize
        +SmallEsize
        +sortsize
        +IObuffersize*sizeof(WORD) + i + 16;
    sort = (SORTING *)Malloc1(allocation,"sort buffers");

    sort->LargeSize = LargeSize/sizeof(WORD);
    sort->SmallSize = SmallSize/sizeof(WORD);
    sort->SmallEsize = SmallEsize/sizeof(WORD);
    sort->MaxPatches = MaxPatches;
    sort->MaxFpatches = MaxFpatches;
    sort->TermsInSmall = TermsInSmall;
    sort->Terms2InSmall = terms2insmall;

    sort->sPointer = (WORD **)(sort+1);
    sort->SplitScratch = sort->sPointer + terms2insmall;
    sort->Patches = (WORD **)(sort->SplitScratch + terms2insmall/2);
    sort->pStop = sort->Patches+longer;
    sort->poina = sort->pStop+longer;
    sort->poin2a = sort->poina + longerp;
    sort->fPatches = (POSITION *)(sort->poin2a+longerp);
    sort->fPatchesStop = sort->fPatches + longer;
    sort->inPatches = sort->fPatchesStop + longer;
    sort->tree = (WORD *)(sort->inPatches + longer);
    sort->used = sort->tree+longerp;
#ifdef WITHZLIB
    sort->fpcompressed = sort->used+longerp;
    sort->fpincompressed = sort->fpcompressed+longerp+2;
    sort->ktoi = sort->fpincompressed+longerp+2;
    sort->zsparray = 0;
#else
    sort->ktoi = sort->used + longerp;
#endif
```

```
    sort->lBuffer = (WORD *)(sort->ktoi + longerp + 2);
    sort->lTop = sort->lBuffer+sort->LargeSize;
    sort->sBuffer = sort->lTop;
    if ( sort->LargeSize == 0 ) { sort->lBuffer = 0; sort->lTop = 0; }
    sort->sTop = sort->sBuffer + sort->SmallSize;
    sort->sTop2 = sort->sBuffer + sort->SmallEsize;
    sort->sHalf = sort->sBuffer + (LONG)((sort->SmallSize+sort->SmallEsize)»1);
    sort->file.PObuffer = (WORD *)(sort->sTop2);
    sort->file.POstop = sort->file.PObuffer+IObuffersize;
    sort->file.POsize = IObuffersize * sizeof(WORD);
    sort->file.POfill = sort->file.POfull = sort->file.PObuffer;
    sort->file.active = 0;
    sort->file.handle = -1;
    PUTZERO(sort->file.POposition);
#ifdef WITHPTHREADS
    sort->file.pthreadslock = dummylock;
#endif
#ifdef WITHZLIB
/*  sort->file.ziosize = IOsize; */
    sort->file.ziosize = IObuffersize*sizeof(WORD);
#endif
    if ( AM.S0 != 0 ) {
        sort->file.name = (char *)(sort->file.PObuffer + IObuffersize);
        AllocSortFileName(sort);
    }
    else sort->file.name = 0;
    sort->cBuffer = 0;
    sort->cBufferSize = 0;
    sort->f = 0;

    return(sort);
}

/*
        #] AllocSort :
        #[ AllocSortFileName :
*/

VOID AllocSortFileName(SORTING *sort)
{
    GETIDENTITY
    char *s, *t;
/*
        This is not the allocation before the tempfiles are determined.
        Hence we can use the name in FG.fname2 and modify the tail
*/
    s = FG.fname2; t = sort->file.name;
    while ( *s ) *t++ = *s++;
#ifdef WITHPTHREADS
    t[-2] = 'F';
    sprintf(t-1,"%d.%d",identity,AN.filenum);
#else
    t[-2] = 'f';
    sprintf(t-1,"%d",AN.filenum);
#endif
    AN.filenum++;
}

/*
        #] AllocSortFileName :
        #[ AllocFileHandle :
*/

FILEHANDLE *AllocFileHandle(WORD par,char *name)
{
    GETIDENTITY
    LONG allocation, Ssize;
    FILEHANDLE *fh;
    int i = 0;
    char *s, *t;

    s = FG.fname2; i = 0;
    while ( *s ) { s++; i++; }
    if ( par == 0 ) { i += 16; Ssize = AM.SIOsize; }
    else { s = name; while ( *s ) { i++; s++; } i+= 2; Ssize = AM.SpectatorSize; }

    allocation = sizeof(FILEHANDLE) + (Ssize+1)*sizeof(WORD) + i*sizeof(char);
    fh = (FILEHANDLE *)Malloc1(allocation,"FileHandle");

    fh->PObuffer = (WORD *)(fh+1);
    fh->POstop = fh->PObuffer+Ssize;
    fh->POsize = Ssize * sizeof(WORD);
    fh->active = 0;
    fh->handle = -1;
    PUTZERO(fh->POposition);
#ifdef WITHPTHREADS
    fh->pthreadslock = dummylock;
```

```
#endif
    if ( par == 0 ) {   /* sort file */
      if ( AM.S0 != 0 ) {
        fh->name = (char *)(fh->POstop + 1);
        s = FG.fname2; t = fh->name;
        while ( *s ) *t++ = *s++;
#ifdef WITHPTHREADS
        t[-2] = 'F';
        sprintf(t-1,"%d-%d",identity,AN.filenum);
#else
        t[-2] = 'f';
        sprintf(t-1,"%d",AN.filenum);
#endif
        AN.filenum++;
      }
      else fh->name = 0;
    }
    else {  /* Spectator file */
        fh->name = (char *)(fh->POstop + 1);
        s = FG.fname; t = fh->name;
        for ( i  = 0; i < FG.fnamebase; i++ ) *t++ = *s++;
        s = name;
        while ( *s ) *t++ = *s++;
        *t = 0;
    }
    fh->POfill = fh->POfull = fh->PObuffer;
    return(fh);
}

/*
        #] AllocFileHandle :
        #[ DeAllocFileHandle :

        Made to repair deallocation of AN.filenum. 21-sep-2000
*/

void DeAllocFileHandle(FILEHANDLE *fh)
{
    GETIDENTITY
    if ( fh->handle >= 0 ) {
        CloseFile(fh->handle);
        fh->handle = -1;
        remove(fh->name);
    }
    AN.filenum--; /* free namespace. was forgotten in first reading */
    M_free(fh,"Temporary FileHandle");
}

/*
        #] DeAllocFileHandle :
        #[ MakeSetupAllocs :
*/

int MakeSetupAllocs()
{
    if ( RecalcSetups() || AllocSetups() ) return(1);
    else return(0);
}

/*
        #] MakeSetupAllocs :
        #[ TryFileSetups :

        Routine looks in the input file for a start of the type
        [#-]
        #: setupparameter value
        It keeps looking until the first line that does not start with
        #-, #+ or #:
        Then it rewinds the input.
*/

#define SETBUFSIZE 257

int TryFileSetups()
{
    LONG oldstreamposition;
    int oldstream;
    int error = 0, eqnum;
    int oldNoShowInput = AC.NoShowInput;
    UBYTE buff[SETBUFSIZE+1], *s, *t, *u, *settop, c;
    LONG linenum, prevline;

    if ( AC.CurrentStream == 0 ) return(error);
    oldstream = AC.CurrentStream - AC.Streams;
    oldstreamposition = GetStreamPosition(AC.CurrentStream);
    linenum = AC.CurrentStream->linenumber;
    prevline = AC.CurrentStream->prevline;
```

```
    eqnum = AC.CurrentStream->eqnum;
    AC.NoShowInput = 1;
    settop = buff + SETBUFSIZE;
    for(;;) {
        c = GetInput();
        if ( c == '*' || c == '\n' ) {
            while ( c != '\n' && c != ENDOFINPUT ) c = GetInput();
            if ( c == ENDOFINPUT ) goto eoi;
            continue;
        }
        if ( c == ENDOFINPUT ) goto eoi;
        if ( c != '#' ) break;
        c = GetInput();
        if ( c == ENDOFINPUT ) goto eoi;
        if ( c != '-' && c != '+' && c != ':' ) break;
        if ( c != ':' ) {
            while ( c != '\n' && c != ENDOFINPUT ) c = GetInput();
            continue;
        }
        s = buff;
        while ( ( c = GetInput() ) == ' ' || c == '\t' || c == '\r' ) {}
        if ( c == ENDOFINPUT ) break;
        if ( c == LINEFEED ) continue;
        if ( c == 0 || c == ENDOFINPUT ) break;
        while ( c != LINEFEED ) {
            *s++ = c;
            c = GetInput();
            if ( c != LINEFEED && c != '\r' ) continue;
            if ( s >= settop ) {
                while ( c != '\n' && c != ENDOFINPUT ) c = GetInput();
                MesPrint("Setups in .frm file: Line too long. setup ignored");
                error++; goto nextline;
            }
        }
        *s++ = '\n';
        t = s = buff; /* name of the option */
        while ( tolower(*s) >= 'a' && tolower(*s) <= 'z' ) s++;
        if ( *s != '\n' ) {
            *s++ = 0;
            while ( *s == ' ' || *s == '\t' ) s++;
            u = s; /* 'value' of the option */
            while ( *s && *s != '\n' && *s != '\r' ) s++;
            if ( *s ) *s++ = 0;
        }
        else {
            /* The value is empty. */
            u = s;
            *s++ = 0;
        }
        error += ProcessOption(t,u,1);
nextline:;
    }
    AC.NoShowInput = oldNoShowInput;
    AC.CurrentStream = AC.Streams + oldstream;
    PositionStream(AC.CurrentStream,oldstreamposition);
    AC.CurrentStream->linenumber = linenum;
    AC.CurrentStream->prevline = prevline;
    AC.CurrentStream->eqnum = eqnum;
    ClearPushback();
    return(error);
eoi:
    MesPrint("Input file without a program.");
    return(-1);
}

/*
        #] TryFileSetups :
        #[ TryEnvironment :
*/

int TryEnvironment()
{
    char *s, *t, *u, varname[100];
    int i,imax = sizeof(setupparameters)/sizeof(SETUPPARAMETERS);
    int error = 0;
    varname[0] = 'F'; varname[1] = 'O'; varname[2] = 'R'; varname[3] = 'M';
    varname[4] = '_'; varname[5] = 0;
    for ( i = 0; i < imax; i++ ) {
        t = s = (char *)(setupparameters[i].parameter);
        u = varname+5;
        while ( *s ) { *u++ = (char)(toupper((unsigned char)*s)); s++; }
        *u = 0;
        s = (char *)(getenv(varname));
        if ( s ) {
            error += ProcessOption((UBYTE *)t,(UBYTE *)s,2);
        }
    }
```

```
    return(error);
}

/*
        #] TryEnvironment :
    #] Setups :
*/
```

4.98 smart.c File Reference

`#include "form3.h"`
Include dependency graph for smart.c:

Functions

- int StudyPattern (WORD *lhs)
- int MatchIsPossible (WORD *pattern, WORD *term)

4.98.1 Detailed Description

The functions for smart pattern searches in combinations of functions. When many wildcards are involved and the functions are (anti)symmetric an exhaustive search for all possibilities may take very much time (like factorial in the number of wildcards) while a human can often see immediately that there cannot be a match. The routines here try to make FORM a bit smarter in this respect.

This is just the beginning. It still needs lots of work!

Definition in file smart.c.

4.98.2 Function Documentation

4.98.2.1 StudyPattern()

```
int StudyPattern (
            WORD * lhs )
```

Definition at line 62 of file smart.c.

4.98.2.2 MatchIsPossible()

```
int MatchIsPossible (
            WORD * pattern,
            WORD * term )
```

Definition at line 299 of file smart.c.

4.99 smart.c

Go to the documentation of this file.

```
00001
00012 /* #[ License : */
00013 /*
00014  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00015  *   When using this file you are requested to refer to the publication
00016  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00017  *   This is considered a matter of courtesy as the development was paid
00018  *   for by FOM the Dutch physics granting agency and we would like to
00019  *   be able to track its scientific use to convince FOM of its value
00020  *   for the community.
00021  *
00022  *   This file is part of FORM.
00023  *
00024  *   FORM is free software: you can redistribute it and/or modify it under the
00025  *   terms of the GNU General Public License as published by the Free Software
00026  *   Foundation, either version 3 of the License, or (at your option) any later
00027  *   version.
00028  *
00029  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00030  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00031  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00032  *   details.
00033  *
00034  *   You should have received a copy of the GNU General Public License along
00035  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00036  */
00037 /* #] License : */
00038 /*
00039     #[ Includes : function.c
00040 */
00041
00042 #include "form3.h"
00043
00044 /*
00045     #] Includes :
00046     #[ StudyPattern :
00047
00048     Argument is a complete lhs of an id-statement
00049     Its last word is a zero (new convention(18-may-1997)) to indicate
00050     that no extra information is following. We can add there information
00051     about the pattern that will help during the actual pattern matching.
00052     Currently the WorkPointer points to just after this lhs.
00053
00054     Task of this routine: To study the functions, their symmetry properties
00055     and their wildcards to determine in which order the functions can
00056     be matched best. If the order should be different we can change it here.
00057
00058     Problem encountered 22-dec-2008 (JV): we don't take noncommuting
00059     functions into account.
00060 */
00061
00062 int StudyPattern(WORD *lhs)
00063 {
00064     GETIDENTITY
00065     WORD *fullproto, *pat, *p, *p1, *p2, *pstop, *info, f, nn;
00066     int numfun = 0, numsym = 0, allwilds = 0, i, j, k, nc;
00067     FUN_INFO *finf, *fmin, *f1, *f2, funscratch;
00068
00069     fullproto = lhs + IDHEAD;
00070 /*  if ( *lhs == TYPEIF ) fullproto--; */
00071     pat = fullproto + fullproto[1];
00072     info = pat + *pat;
00073
00074     p = pat + 1;
00075     while ( p < info ) {
00076         if ( *p >= FUNCTION ) {
```

```
            numfun++;
            nn = *p - FUNCTION;
            if ( nn >= WILDOFFSET ) nn -= WILDOFFSET;
/*
            We check here for cases that are not allowed like ?a inside
            symmetric functions or tensors.
*/
            if ( ( functions[nn].symmetric == SYMMETRIC ) ||
                 ( functions[nn].symmetric == ANTISYMMETRIC ) ) {
              p2 = p+p[1]; p1 = p+FUNHEAD;
              if ( functions[nn].spec ) {
                while ( p1 < p2 ) {
                    if ( *p1 == FUNNYWILD ) {
                        MesPrint("&Argument field wildcards are not allowed inside (anti)symmetric functions or tensors");
                        return(1);
                    }
                    p1++;
                }
              }
              else {
                while ( p1 < p2 ) {
                    if ( *p1 == -ARGWILD ) {
                        MesPrint("&Argument field wildcards are not allowed inside (anti)symmetric functions or tensors");
                        return(1);
                    }
                    NEXTARG(p1);
                }
              }
            }
        }
        p += p[1];
    }
    if ( numfun == 0 ) return(0);
    if ( ( lhs[2] & SUBMASK ) == SUBALL ) {
        p = pat + 1;
        while ( p < info ) {
            if ( *p == SYMBOL || *p == VECTOR || *p == DOTPRODUCT || *p == INDEX ) {
                MesPrint("&id,all can have only functions and/or tensors in the lhs.");
                return(1);
            }
            p += p[1];
        }
    }
/*
    We need now some room for the information about the functions
*/
    if ( numfun > AN.numfuninfo ) {
        if ( AN.FunInfo ) M_free(AN.FunInfo,"funinfo");
        AN.numfuninfo = numfun + 10;
        AN.FunInfo = (FUN_INFO *)Malloc1(AN.numfuninfo*sizeof(FUN_INFO),"funinfo");
    }
/*
    Now collect the information. First the locations.
*/
    p = pat + 1; i = 0;
    while ( p < info ) {
        if ( *p >= FUNCTION ) AN.FunInfo[i++].location = p;
        p += p[1];
    }
    for ( i = 0, finf = AN.FunInfo; i < numfun; i++, finf++ ) {
        p = finf->location;
        pstop = p + p[1];
        f = *p;
        if ( f > FUNCTION+WILDOFFSET ) f -= WILDOFFSET;
        finf->numargs = finf->numfunnies = finf->numwildcards = 0;
        finf->symmet = functions[f-FUNCTION].symmetric;
        finf->tensor = functions[f-FUNCTION].spec;
        finf->commute = functions[f-FUNCTION].commute;
        if ( finf->tensor >= TENSORFUNCTION ) {
            p += FUNHEAD;
            while ( p < pstop ) {
                if ( *p == FUNNYWILD ) {
                    finf->numfunnies++; p+= 2; continue;
                }
                else if ( *p < 0 ) {
                    if ( *p >= AM.OffsetVector + WILDOFFSET && *p < MINSPEC ) {
                        finf->numwildcards++;
                    }
                }
                else {
                    if ( *p >= AM.OffsetIndex + WILDOFFSET &&
                    *p <= AM.OffsetIndex + 2*WILDOFFSET ) finf->numwildcards++;
                }
                finf->numargs++;
                p++;
```

```
            }
        }
        else {
            p += FUNHEAD;
            while ( p < pstop ) {
                if ( *p > 0 ) { finf->numargs++; p += *p; continue; }
                if ( *p <= -FUNCTION ) {
                    if ( *p <= -FUNCTION - WILDOFFSET ) finf->numwildcards++;
                    p++;
                }
                else if ( *p == -SYMBOL ) {
                    if ( p[1] >= 2*MAXPOWER ) finf->numwildcards++;
                    p += 2;
                }
                else if ( *p == -INDEX ) {
                    if ( p[1] >= AM.OffsetIndex + WILDOFFSET &&
                    p[1] <= AM.OffsetIndex + 2*WILDOFFSET ) finf->numwildcards++;
                    p += 2;
                }
                else if ( *p == -VECTOR || *p == -MINVECTOR ) {
                    if ( p[1] >= AM.OffsetVector + WILDOFFSET && p[1] < MINSPEC ) {
                        finf->numwildcards++;
                    }
                    p += 2;
                }
                else if ( *p == -ARGWILD ) {
                    finf->numfunnies++;
                    p += 2;
                }
                else { p += 2; }
                finf->numargs++;
            }
        }
        if ( finf->symmet ) {
            numsym++;
            allwilds += finf->numwildcards + finf->numfunnies;
        }
    }
    if ( numsym == 0 ) return(0);
    if ( allwilds == 0 ) return(0);
/*
    We have the information in the array AN.FunInfo.
    We sort things and then write the sorted pattern.
    Of course we may not play with the order of the noncommuting functions.
    Of course we have to become even smarter in the future and look during
    the sorting which wildcards are asigned when.
    But for now this should do.
*/
    for ( nc = numfun-1; nc >= 0; nc-- ) { if ( AN.FunInfo[nc].commute ) break; }

    finf = AN.FunInfo;
    for ( i = nc+2; i < numfun; i++ ) {
        fmin = finf; finf++;
        if ( ( finf->symmet < fmin->symmet ) || (
        ( finf->symmet == fmin->symmet ) &&
        ( ( finf->numwildcards+finf->numfunnies < fmin->numwildcards+fmin->numfunnies )
        || ( ( finf->numwildcards+finf->numfunnies == fmin->numwildcards+fmin->numfunnies )
        && ( finf->numwildcards < fmin->numfunnies ) ) ) ) ) {
            funscratch = AN.FunInfo[i];
            AN.FunInfo[i] = AN.FunInfo[i-1];
            AN.FunInfo[i-1] = funscratch;
            for ( j = i-1; j > nc && j > 0; j-- ) {
                f1 = AN.FunInfo+j;
                f2 = f1-1;
                if ( ( f1->symmet < f2->symmet ) || (
                ( f1->symmet == f2->symmet ) &&
                ( ( f1->numwildcards+f1->numfunnies < f2->numwildcards+f2->numfunnies )
                || ( ( f1->numwildcards+f1->numfunnies == f2->numwildcards+f2->numfunnies )
                && ( f1->numwildcards < f2->numfunnies ) ) ) ) ) {
                    funscratch = AN.FunInfo[j];
                    AN.FunInfo[j] = AN.FunInfo[j-1];
                    AN.FunInfo[j-1] = funscratch;
                }
                else break;
            }
        }
    }
/*
    Now we rewrite the pattern. First into the space after it and then we
    copy it back. Be careful with the non-commutative functions. There the
    worst one should decide.
*/
    p = pat + 1;
    p2 = info;
    for ( i = 0; i < numfun; i++ ) {
        if ( i == nc ) {
            for ( k = 0; k <= nc; k++ ) {
```

```
                if ( AN.FunInfo[k].commute ) {
                    p1 = AN.FunInfo[k].location; j = p1[1];
                    NCOPY(p2,p1,j)
                }
            }
        }
        else if ( AN.FunInfo[i].commute == 0 ) {
            p1 = AN.FunInfo[i].location; j = p1[1];
            NCOPY(p2,p1,j)
        }
    }
    p = pat + 1; p1 = info;
    while ( p1 < p2 ) *p++ = *p1++;
/*
    And now we place the relevant information in the info part
*/
    p2 = info+1;
    for ( i = 0; i < numfun; i++ ) {
        if ( i == nc ) {
            for ( k = 0; k <= nc; k++ ) {
                if ( AN.FunInfo[k].commute ) {
                    finf = AN.FunInfo + k;
                    *p2++ = finf->numargs;
                    *p2++ = finf->numwildcards;
                    *p2++ = finf->numfunnies;
                    *p2++ = finf->symmet;
                }
            }
        }
        else if ( AN.FunInfo[i].commute == 0 ) {
            finf = AN.FunInfo + i;
            *p2++ = finf->numargs;
            *p2++ = finf->numwildcards;
            *p2++ = finf->numfunnies;
            *p2++ = finf->symmet;
        }
    }
    *info = p2-info;
    lhs[1] = p2-lhs;
    return(0);
}

/*
    #] StudyPattern :
    #[ MatchIsPossible :

    We come here when there are functions and there is nontrivial
    symmetry related wildcarding.
*/

int MatchIsPossible(WORD *pattern, WORD *term)
{
    GETIDENTITY
    WORD *info = pattern + *pattern;
    WORD *t, *tstop, *tt, *inf, *p;
    int numfun = 0, inpat, i, j, numargs;
    FUN_INFO *finf;
/*
    We need a list of functions and their number of arguments
*/
    GETSTOP(term,tstop);
    t = term + 1;
    while ( t < tstop ) {
        if ( *t >= FUNCTION ) numfun++;
        t += t[1];
    }
    if ( numfun == 0 ) goto NotPossible;
    if ( *info == SETSET ) info += info[1];
    inpat = (*info-1)/4;
    if ( inpat > numfun ) goto NotPossible;
/*
    We need now some room for the information about the functions
*/
    if ( numfun > AN.numfuninfo ) {
        if ( AN.FunInfo ) M_free(AN.FunInfo,"funinfo");
        AN.numfuninfo = numfun + 10;
        AN.FunInfo = (FUN_INFO *)Malloc1(AN.numfuninfo*sizeof(FUN_INFO),"funinfo");
    }
    t = term + 1; finf = AN.FunInfo;
    while ( t < tstop ) {
        if ( *t >= FUNCTION ) {
            finf->location = t;
            if ( functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
                numargs = t[1]-FUNHEAD;
                t += t[1];
            }
            else {
```

```
                numargs = 0;
                tt = t + t[1];
                t += FUNHEAD;
                while ( t < tt ) {
                    numargs++;
                    NEXTARG(t)
                }
            }
            finf->numargs = numargs;
            finf++;
        }
        else t += t[1];
    }
/*
    Now we first find a partner for each function in the pattern
    with a fixed number of arguments
*/
    for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
        if ( inf[2] ) continue;
        if ( *p >= (FUNCTION+WILDOFFSET) ) continue;
        for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
            if ( *p == *(finf->location) && *inf == finf->numargs ) {
                finf->numargs = -finf->numargs-1;
                break;
            }
        }
        if ( j >= numfun ) goto NotPossible;
    }
    for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
        if ( inf[2] ) continue;
        if ( *p < (FUNCTION+WILDOFFSET) ) continue;
        for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
            if ( *inf == finf->numargs ) {
                finf->numargs = -finf->numargs-1;
                break;
            }
        }
        if ( j >= numfun ) goto NotPossible;
    }
    for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
        if ( inf[2] == 0 ) continue;
        if ( *p >= (FUNCTION+WILDOFFSET) ) continue;
        for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
            if ( *p == *(finf->location) && (*inf-inf[2]) <= finf->numargs ) {
                finf->numargs = -finf->numargs-1;
                break;
            }
        }
        if ( j >= numfun ) goto NotPossible;
    }
    for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
        if ( inf[2] == 0 ) continue;
        if ( *p < (FUNCTION+WILDOFFSET) ) continue;
        for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
            if ( (*inf-inf[2]) <= finf->numargs ) {
                finf->numargs = -finf->numargs-1;
                break;
            }
        }
        if ( j >= numfun ) goto NotPossible;
    }
/*
    Thus far we have determined that for each function in the pattern
    there is a potential partner with enough arguments.
    For now the rest is up to the real pattern matcher.

    To undo the disabling of the number of arguments we need this code
    for ( i = 0, finf = AN.FunInfo; i < numfun; i++, finf++ ) {
        if ( finf->numargs < 0 ) finf->numargs = -finf->numargs-1;
    }
*/
    return(1);
NotPossible:
/*
    PrintTerm(pattern,"p");
    PrintTerm(term,"t");
*/
    return(0);
}

/*
    #] MatchIsPossible :
*/
```

4.100 sort.c File Reference

`#include "form3.h"`
Include dependency graph for sort.c:

Macros

- #define NEWSPLITMERGE
- #define INSLENGTH(x) w[1] = FUNHEAD+ARGHEAD+x; w[FUNHEAD] = ARGHEAD+x;

Functions

- VOID WriteStats (POSITION ∗plspace, WORD par)
- WORD NewSort (PHEAD0)
- LONG EndSort (PHEAD WORD ∗buffer, int par)
- LONG PutIn (FILEHANDLE ∗file, POSITION ∗position, WORD ∗buffer, WORD ∗∗take, int npat)
- WORD Sflush (FILEHANDLE ∗fi)
- WORD PutOut (PHEAD WORD ∗term, POSITION ∗position, FILEHANDLE ∗fi, WORD ncomp)
- WORD FlushOut (POSITION ∗position, FILEHANDLE ∗fi, int compr)
- WORD AddCoef (PHEAD WORD ∗∗ps1, WORD ∗∗ps2)
- WORD AddPoly (PHEAD WORD ∗∗ps1, WORD ∗∗ps2)
- VOID AddArgs (PHEAD WORD ∗s1, WORD ∗s2, WORD ∗m)
- WORD Compare1 (PHEAD WORD ∗term1, WORD ∗term2, WORD level)
- int CompareSymbols (PHEAD WORD ∗term1, WORD ∗term2, WORD par)
- int CompareHSymbols (PHEAD WORD ∗term1, WORD ∗term2, WORD par)
- LONG ComPress (WORD ∗∗ss, LONG ∗n)
- LONG SplitMerge (PHEAD WORD ∗∗Pointer, LONG number)
- VOID GarbHand ()
- WORD MergePatches (WORD par)
- WORD StoreTerm (PHEAD WORD ∗term)
- VOID StageSort (FILEHANDLE ∗fout)
- WORD SortWild (WORD ∗w, WORD nw)
- void CleanUpSort (int num)
- VOID LowerSortLevel ()
- WORD ∗ PolyRatFunSpecial (PHEAD WORD ∗t1, WORD ∗t2)

Variables

- char ∗ toterms [ ] = { " ", " >>", "-->" }

4.100.1 Detailed Description

Contains the sort routines. We distinguish levels of sorting. The ground level is the sorting of terms in an expression. When a term has functions, the arguments can contain terms that need sorting, which this then done by raising the level. This can happen recursively. NewSort and EndSort automatically raise and lower the level. Because the ground level does some special things, sometimes we have to raise immediately to the second level skipping the ground level.

Special routines for the parallel sorting are in the file threads.c Also the sorting of terms in polynomials is special but most of that is controled by changing the address of the compare routine. Other routines relevant for adding rational polynomials are in the file polynito.c

Definition in file sort.c.

4.100.2 Macro Definition Documentation

4.100.2.1 NEWSPLITMERGE

```
#define NEWSPLITMERGE
```

Definition at line 53 of file sort.c.

4.100.2.2 INSLENGTH

```
#define INSLENGTH(
              x ) w[1] = FUNHEAD+ARGHEAD+x; w[FUNHEAD] = ARGHEAD+x;
```

Definition at line 2215 of file sort.c.

4.100.3 Function Documentation

4.100.3.1 WriteStats()

```
VOID WriteStats (
            POSITION * plspace,
            WORD par )
```

Writes the statistics.

Parameters

plspace	The size in bytes currently occupied
par	par = 0 after a splitmerge. par = 1 after merge to sortfile. par = 2 after the sort

current expression is to be found in AR.CurExpr. terms are in S->TermsLeft. S->GenTerms.

Definition at line 91 of file sort.c.

References TimeCPU(), and TimeWallClock().

Referenced by EndSort(), PF_Processor(), and StoreTerm().

Here is the call graph for this function:

4.100.3.2 NewSort()

```
WORD NewSort (
            PHEAD0  )
```

Starts a new sort. At the lowest level this is a 'main sort' with the struct according to the parameters in S0. At higher levels this is a sort for functions, subroutines or dollars. We prepare the arrays and structs.

Returns

Regular convention (OK -> 0)

Definition at line 589 of file sort.c.

Referenced by InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Processor(), PolyFunMul(), Processor(), TakeArgContent(), TestMatch(), and TestSub().

4.100.3.3 EndSort()

```
LONG EndSort (
            PHEAD WORD * buffer,
            int par )
```

Finishes a sort. At AR.sLevel == 0 the output is to the regular output stream. When AR.sLevel > 0, the parameter par determines the actual output. The AR.sLevel will be popped. All ongoing stages are finished and if the sortfile is open it is closed. The statistics are printed when AR.sLevel == 0 par == 0 Output to the buffer. par == 1 Sort for function arguments. The output will be copied into the buffer. It is assumed that this is in the WorkSpace. par == 2 Sort for $-variable. We return the address of the buffer that contains the output in buffer (treated like WORD **). We first catch the output in a file (unless we can intercept things after the small buffer has been sorted) Then we read from the file into a buffer. Only when par == 0 data compression can be attempted at AT.SS==AT.S0.

Parameters

buffer	buffer for output when needed
par	See above

Returns

If negative: error. If positive: number of words in output.

Definition at line 675 of file sort.c.

References ComPress(), FlushOut(), MergePatches(), PF_EndSort(), PutOut(), SplitMerge(), StageSort(), and WriteStats().

Referenced by InFunction(), MakeDollarInteger(), MakeDollarMod(), PF_CollectModifiedDollars(), PF_Processor(), PolyFunMul(), Processor(), TakeArgContent(), TestMatch(), and TestSub().

Here is the call graph for this function:

4.100.3.4 PutIn()

```
LONG PutIn (
            FILEHANDLE * file,
            POSITION * position,
            WORD * buffer,
            WORD ** take,
            int npat )
```

Reads a new patch from position in file handle. It is put at buffer, anything after take is moved forward. This would be part of a term that hasn't been used yet. Because of this there should be some space before the start of the buffer

Parameters

file	The file system from which to read
position	The position from which to read
buffer	The buffer into which to read
take	The unused tail should be moved before the buffer
npat	The number of the patch. Is needed if the information was compressed with gzip, because each patch has its own independent gzip encoding.

Definition at line 1241 of file sort.c.

References FiLe::handle.

Referenced by MergePatches().

4.100.3.5 Sflush()

```
WORD Sflush (
            FILEHANDLE * fi )
```

Puts the contents of a buffer to output Only to be used when there is a single patch in the large buffer.

Parameters

fi	The filesystem (or its cache) to which the patch should be written

Definition at line 1301 of file sort.c.

References FiLe::handle.

Referenced by MergePatches().

4.100.3.6 PutOut()

```
WORD PutOut (
            PHEAD WORD * term,
            POSITION * position,
            FILEHANDLE * fi,
            WORD ncomp )
```

Routine writes one term to file handle at position. It returns the new value of the position.

NOTE: For 'final output' we may have to index the brackets. See the struct BRACKETINDEX. We should maintain: 1: a list with brackets array with the brackets 2: a list of objects of type BRACKETINDEX. It contains array with either pointers or offsets to the list of brackets. starting positions in the file. The index may be tied to a maximum size. In that case we may have to prune the list occasionally.

Parameters

term	The term to be written
position	The position in the file. Afterwards it is updated
fi	The file (or its cache) to which should be written
ncomp	Information about what type of compression should be used

Definition at line 1387 of file sort.c.

References FiLe::handle, and PF_ISendSbuf().

Referenced by EndSort(), MergePatches(), PF_EndSort(), PF_Processor(), and Processor().

Here is the call graph for this function:

4.100.3.7 FlushOut()

```
WORD FlushOut (
            POSITION * position,
            FILEHANDLE * fi,
            int compr )
```

Completes output to an output file and writes the trailing zero.

Parameters

position	The position in the file after writing
fi	The file (or its cache)
compr	Indicates whether there should be compression with gzip.

Returns

Regular conventions (OK -> 0).

Definition at line 1724 of file sort.c.

References FiLe::handle, BrAcKeTiNfO::indexbuffer, and PF_ISendSbuf().

Referenced by EndSort(), MergePatches(), PF_EndSort(), and Processor().

Here is the call graph for this function:

4.100.3.8 AddCoef()

```
WORD AddCoef (
            PHEAD WORD ** ps1,
            WORD ** ps2 )
```

Adds the coefficients of the terms ∗ps1 and ∗ps2. The problem comes when there is not enough space for a new longer coefficient. First a local solution is tried. If this is not succesfull we need to move terms around. The possibility of a garbage collection should not be ignored, as avoiding this costs very much extra space which is nearly wasted otherwise.

If the return value is zero the terms cancelled.

The resulting term is left in ∗ps1.

Definition at line 1935 of file sort.c.

References GarbHand(), and NormalModulus().

Referenced by SplitMerge().

Here is the call graph for this function:

4.100.3.9 AddPoly()

```
WORD AddPoly (
            PHEAD WORD ** ps1,
            WORD ** ps2 )
```

Routine should be called when S->PolyWise != 0. It points then to the position of AR.PolyFun in both terms.

We add the contents of the arguments of the two polynomials. Special attention has to be given to special arguments. We have to reserve a space equal to the size of one term + the size of the argument of the other. The addition has to be done in this routine because not all objects are reentrant.

Newer addition (12-nov-2007). The PolyFun can have two arguments. In that case S->PolyFlag is 2 and we have to call the routine for adding rational polynomials. We have to be rather careful what happens with: The location of the output The order of the terms in the arguments At first we allow only univariate polynomials in the PolyFun. This restriction will be lifted a.s.a.p.

Parameters

ps1	A pointer to the postion of the first term
ps2	A pointer to the postion of the second term

Returns

If zero the terms cancel. Otherwise the new term is in ∗ps1.

Definition at line 2062 of file sort.c.

References AddArgs(), and GarbHand().

Referenced by SplitMerge().

Here is the call graph for this function:

4.100.3.10 AddArgs()

```
VOID AddArgs (
            PHEAD WORD * s1,
            WORD * s2,
            WORD * m )
```

Adds the arguments of two occurrences of the PolyFun.

Parameters

s1	Pointer to the first occurrence.
s2	Pointer to the second occurrence.
m	Pointer to where the answer should be.

Definition at line 2224 of file sort.c.

Referenced by AddPoly(), and MergePatches().

4.100.3.11 Compare1()

```
WORD Compare1 (
            PHEAD WORD * term1,
            WORD * term2,
            WORD level )
```

Compares two terms. The answer is: 0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first. Some special precautions may be needed to keep the CompCoef routine from generating overflows, although this is very unlikely in subterms. This routine should not return an error condition.

Originally this routine was called Compare. With the treatment of special polynomials with terms that contain only symbols and the need for extreme speed for the polynomial routines we made a special compare routine and now we store the address of the current compare routine in AR.CompareRoutine and have a macro Compare which makes all existing code work properly and we can just replace the routine on a thread by thread basis (each thread has its own AR struct).

Parameters

term1	First input term
term2	Second input term
level	The sorting level (may influence on the result)

Returns

0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first.

Definition at line 2509 of file sort.c.

References CompCoef().

Referenced by InFunction(), and StartVariables().

Here is the call graph for this function:

4.100.3.12 CompareSymbols()

```
int CompareSymbols (
            PHEAD WORD * term1,
            WORD * term2,
            WORD par )
```

Compares the terms, based on the value of AN.polysortflag. If term1 < term2 the return value is -1 If term1 > term2 the return value is 1 If term1 = term2 the return value is 0 The coefficients may differ. The terms contain only a single subterm of type SYMBOL. If AN.polysortflag = 0 it is a 'regular' compare. If AN.polysortflag = 1 the sum of the powers is more important par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in sort.c

Definition at line 2945 of file sort.c.

Referenced by InFunction(), and TestSub().

4.100.3.13 CompareHSymbols()

```
int CompareHSymbols (
            PHEAD WORD * term1,
            WORD * term2,
            WORD par )
```

Compares terms that can have only SYMBOL and HAAKJE subterms. If term1 < term2 the return value is -1 If term1 > term2 the return value is 1 If term1 = term2 the return value is 0 par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in sort.c

Definition at line 2988 of file sort.c.

4.100.3.14 ComPress()

```
LONG ComPress (
            WORD ** ss,
            LONG * n )
```

Gets a list of pointers to terms and compresses the terms. In n it collects the number of terms and the return value of the function is the space that is occupied.

We have to pay some special attention to the compression of terms with a PolyFun. This PolyFun should occur only straight before the coefficient, so we can use the same trick as for the coefficient to sabotage compression of this object (Replace in the history the function pointer by zero. This is safe, because terms that would be identical otherwise would have been added).

Parameters

ss	Array of pointers to terms to be compressed.
n	Number of pointers in ss.

Returns

Total number of words needed for the compressed result.

Definition at line 3041 of file sort.c.

Referenced by EndSort(), and StoreTerm().

4.100.3.15 SplitMerge()

```
LONG SplitMerge (
            PHEAD WORD ** Pointer,
            LONG number )
```

Algorithm by J.A.M.Vermaseren (31-7-1988)

Note that AN.SplitScratch and AN.InScratch are used also in GarbHand

Merge sort in memory. The input is an array of pointers. Sorting is done recursively by dividing the array in two equal parts and calling SplitMerge for each. When the parts are small enough we can do the compare and take the appropriate action. An addition is that we look for 'runs'. Sequences that are already ordered. This happens a lot when there is very little action in a module. This made FORM faster by a few percent.

Parameters

Pointer	The array of pointers to the terms to be sorted.
number	The number of pointers in Pointer.

The terms are supposed to be sitting in the small buffer and there is supposed to be an extension to this buffer for when there are two terms that should be added and the result takes more space than each of the original terms. The notation guarantees that the result never needs more space than the sum of the spaces of the original terms.

Definition at line 3207 of file sort.c.

References AddCoef(), AddPoly(), and SplitMerge().

Referenced by EndSort(), SplitMerge(), and StoreTerm().

Here is the call graph for this function:

4.100.3.16 GarbHand()

```
VOID GarbHand ( )
```

Garbage collection that takes place when the small extension is full and we need to place more terms there. When this is the case there are many holes in the small buffer and the whole can be compactified. The major complication

is the buffer for SplitMerge. There are to options for temporary memory: 1: find some buffer that has enough space (maybe in the large buffer). 2: allocate a buffer. Give it back afterwards of course. If the small extension is properly dimensioned this routine should be called very rarely. Most of the time it will be called when the polyfun or polyratfun is active.

Definition at line 3383 of file sort.c.

Referenced by AddCoef(), and AddPoly().

4.100.3.17 MergePatches()

```
WORD MergePatches (
            WORD par )
```

The general merge routine. Can be used for the large buffer and the file merging. The array S->Patches tells where the patches start S->pStop tells where they end (has to be computed first). The end of a 'line to be merged' is indicated by a zero. If the end is reached without running into a zero or a term runs over the boundary of a patch it is a file merging operation and a new piece from the file is read in.

Parameters

par	If par == 0 the sort is for file -> outputfile. If par == 1 the sort is for large buffer -> sortfile. If par == 2 the sort is for large buffer -> outputfile.

Definition at line 3498 of file sort.c.

References AddArgs(), FlushOut(), FiLe::handle, NormalModulus(), PutIn(), PutOut(), Sflush(), and StageSort().

Referenced by EndSort(), and StoreTerm().

Here is the call graph for this function:

4.100.3.18 StoreTerm()

```
WORD StoreTerm (
            PHEAD WORD * term )
```

The central routine to accept terms, store them and keep things at least partially sorted. A call to EndSort will then complete storing and sorting.

Parameters

term	The term to be stored

Returns

Regular return conventions (OK -> 0)

Definition at line 4246 of file sort.c.

References ComPress(), MergePatches(), SplitMerge(), and WriteStats().

Referenced by Generator(), InFunction(), PF_Processor(), PolyFunMul(), Processor(), TakeArgContent(), and TestSub().

Here is the call graph for this function:

4.100.3.19 StageSort()

```
VOID StageSort (
            FILEHANDLE * fout )
```

Prepares a stage 4 or higher sort. Stage 4 sorts occur when the sort file contains more patches than can be merged in one pass.

Definition at line 4364 of file sort.c.

References FiLe::handle.

Referenced by EndSort(), and MergePatches().

4.100.3.20 SortWild()

```
WORD SortWild (
            WORD * w,
            WORD nw )
```

Sorts the wildcard entries in the parameter w. Double entries are removed. Full space taken is nw words. Routine serves for the reading of wildcards in the compiler. The entries come in the format: (type,4,number,0) in which the zero is reserved for the future replacement of 'number'.

Parameters

w	buffer with wildcard entries.
nw	number of wildcard entries.

Returns

Normal conventions (OK -> 0)

Definition at line 4444 of file sort.c.

4.100.3.21 CleanUpSort()

```
void CleanUpSort (
            int num )
```

Partially or completely frees function sort buffers.

Definition at line 4536 of file sort.c.

4.100.3.22 LowerSortLevel()

```
VOID LowerSortLevel ( )
```

Lowers the level in the sort system.

Definition at line 4610 of file sort.c.

Referenced by InFunction(), PF_CollectModifiedDollars(), PF_Processor(), PolyFunMul(), Processor(), TestMatch(), and TestSub().

4.100.3.23 PolyRatFunSpecial()

```
WORD * PolyRatFunSpecial (
            PHEAD WORD * t1,
            WORD * t2 )
```

Definition at line 4627 of file sort.c.

4.100.4 Variable Documentation

4.100.4.1 toterms

```
char* toterms[] = { " ", " >>", "-->" }
```

Definition at line 75 of file sort.c.

4.101 sort.c

Go to the documentation of this file.

```
00001
00017 /* #[ License : */
00018 /*
00019  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00020  *   When using this file you are requested to refer to the publication
00021  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00022  *   This is considered a matter of courtesy as the development was paid
00023  *   for by FOM the Dutch physics granting agency and we would like to
00024  *   be able to track its scientific use to convince FOM of its value
00025  *   for the community.
00026  *
00027  *   This file is part of FORM.
00028  *
00029  *   FORM is free software: you can redistribute it and/or modify it under the
00030  *   terms of the GNU General Public License as published by the Free Software
00031  *   Foundation, either version 3 of the License, or (at your option) any later
00032  *   version.
00033  *
00034  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00035  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00036  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00037  *   details.
00038  *
00039  *   You should have received a copy of the GNU General Public License along
00040  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00041  */
00042 /* #] License : */
00043 /*
00044     #[ Includes : sort.c
00045
00046     Sort routines according to new conventions (25-jun-1997).
00047     This is more object oriented.
00048     The active sort is indicated by AT.SS which should agree with
00049     AN.FunSorts[AR.sLevel];
00050
00051 #define GZIPDEBUG
00052 */
00053 #define NEWSPLITMERGE
00054
00055 #include "form3.h"
00056
00057 #ifdef WITHPTHREADS
00058 UBYTE THRbuf[100];
00059 #endif
00060
00061 #ifdef WITHSTATS
00062 extern LONG numwrites;
00063 extern LONG numreads;
00064 extern LONG numseeks;
00065 extern LONG nummallocs;
00066 extern LONG numfrees;
00067 #endif
00068
00069 /*
00070     #] Includes :
00071     #[ SortUtilities :
00072         #[ WriteStats :             VOID WriteStats(lspace,par)
00073 */
00074
00075 char *toterms[] = { "   ", " »", "-->" };
00076
00091 VOID WriteStats(POSITION *plspace, WORD par)
00092 {
00093     GETIDENTITY
00094     LONG millitime, y = 0x7FFFFFFFL » 1;
00095     WORD timepart;
```

```
    SORTING *S;
    POSITION pp;
    int use_wtime;
    if ( AT.SS == AT.S0 && AC.StatsFlag ) {
#ifdef WITHPTHREADS
        if ( AC.ThreadStats == 0 && identity > 0 ) return;
#elif defined(WITHMPI)
        if ( AC.OldParallelStats ) return;
        if ( ! AC.ProcessStats && PF.me != MASTER ) return;
#endif
        if ( Expressions == 0 ) return;

        if ( par == 0 ) {
            AR.ShortSortCount++;
            if ( AR.ShortSortCount < AC.ShortStatsMax ) return;
        }
        AR.ShortSortCount = 0;

        S = AT.SS;
        MLOCK(ErrorMessageLock);
        if ( AC.ShortStats ) {}
        else {
#ifdef WITHPTHREADS
            if ( identity > 0 ) {
                MesPrint("             Thread %d reporting",identity);
            }
            else {
                MesPrint("");
            }
#elif defined(WITHMPI)
            if ( PF.me != MASTER ) {
                MesPrint("             Process %d reporting",PF.me);
            }
            else {
                MesPrint("");
            }
#else
            MesPrint("");
#endif
        }
        /*
         * We define WTimeStatsFlag as a flag to print the wall-clock time on
         * the *master*, not in workers. This can be confusing in thread
         * statistics when short statistics is used. Technically,
         * TimeWallClock() is not thread-safe in TFORM.
         */
        use_wtime = AC.WTimeStatsFlag;
#if defined(WITHPTHREADS)
        if ( use_wtime && identity > 0 ) use_wtime = 0;
#elif defined(WITHMPI)
        if ( use_wtime && PF.me != MASTER ) use_wtime = 0;
#endif
        millitime = use_wtime ? TimeWallClock(1) * 10 : TimeCPU(1);
        timepart = (WORD)(millitime%1000);
        millitime /= 1000;
        timepart /= 10;
        if ( AC.ShortStats ) {
#if defined(WITHPTHREADS) || defined(WITHMPI)
#ifdef WITHPTHREADS
          if ( identity > 0 ) {
#else
          if ( PF.me != MASTER ) {
            const int identity = PF.me;
#endif
            if ( par == 0 || par == 2 ) {
                SETBASEPOSITION(pp,y);
                if ( ISLESSPOS(*plspace,pp) ) {
                    MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%10p %s %s",identity,
                    millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                    S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
/*
                    MesPrint("%d: %14s %17s %7l.%2is %8l>%10l%3s%10l:%10p",identity,
                    EXPRNAME(AR.CurExpr),AC.Commercial,millitime,timepart,
                    AN.ninterms,S->GenTerms,toterms[par],S->TermsLeft,plspace);
*/
                }
                else {
                    y = 1000000000L;
                    SETBASEPOSITION(pp,y);
                    MULPOS(pp,100);
                    if ( ISLESSPOS(*plspace,pp) ) {
                        MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%11p %s %s",identity,
                        millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                        S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                    }
                    else {
                        MULPOS(pp,10);
```

```
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%12p %s %s",identity,
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%13p %s %s",identity,
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%14p %s %s",identity,
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%15p %s %s",identity,
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%16p %s %s",identity,
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%17p %s %s",identity,
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        } } } } }
                    }
                }
            }
            else if ( par == 1 ) {
                SETBASEPOSITION(pp,y);
                if ( ISLESSPOS(*plspace,pp) ) {
                    MesPrint("%d: %7l.%2is %10l:%10p",identity,millitime,timepart,
                    S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                }
                else {
                    y = 1000000000L;
                    SETBASEPOSITION(pp,y);
                    MULPOS(pp,100);
                    if ( ISLESSPOS(*plspace,pp) ) {
                        MesPrint("%d: %7l.%2is %10l:%11p",identity,millitime,timepart,
                        S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                    }
                    else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %10l:%12p",identity,millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %10l:%13p",identity,millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %10l:%14p",identity,millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %10l:%15p",identity,millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %10l:%16p",identity,millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
```

```
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%d: %7l.%2is %10l:%17p",identity,millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        } } } } }
                    }
                }
            } } else
#endif
            {
            if ( par == 0 || par == 2 ) {
                SETBASEPOSITION(pp,y);
                if ( ISLESSPOS(*plspace,pp) ) {
                    MesPrint("%7l.%2is %8l>%10l%3s%10l:%10p %s %s",
                    millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                    S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
/*
                    MesPrint("%14s %17s %7l.%2is %8l>%10l%3s%10l:%10p",
                    EXPRNAME(AR.CurExpr),AC.Commercial,millitime,timepart,
                    AN.ninterms,S->GenTerms,toterms[par],S->TermsLeft,plspace);
*/
                }
                else {
                    y = 1000000000L;
                    SETBASEPOSITION(pp,y);
                    MULPOS(pp,100);
                    if ( ISLESSPOS(*plspace,pp) ) {
                        MesPrint("%7l.%2is %8l>%10l%3s%10l:%11p %s %s",
                        millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                        S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                    }
                    else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %8l>%10l%3s%10l:%12p %s %s",
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %8l>%10l%3s%10l:%13p %s %s",
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %8l>%10l%3s%10l:%14p %s %s",
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %8l>%10l%3s%10l:%15p %s %s",
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %8l>%10l%3s%10l:%16p %s %s",
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %8l>%10l%3s%10l:%17p %s %s",
                            millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        } } } } }
                    }
                }
            }
            else if ( par == 1 ) {
                SETBASEPOSITION(pp,y);
                if ( ISLESSPOS(*plspace,pp) ) {
                    MesPrint("%7l.%2is %10l:%10p",millitime,timepart,
                    S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                }
                else {
```

```
                    y = 1000000000L;
                    SETBASEPOSITION(pp,y);
                    MULPOS(pp,100);
                    if ( ISLESSPOS(*plspace,pp) ) {
                        MesPrint("%7l.%2is %10l:%11p",millitime,timepart,
                        S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                    }
                    else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %10l:%12p",millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %10l:%13p",millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %10l:%14p",millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %10l:%15p",millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %10l:%16p",millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        else {
                        MULPOS(pp,10);
                        if ( ISLESSPOS(*plspace,pp) ) {
                            MesPrint("%7l.%2is %10l:%17p",millitime,timepart,
                            S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
                        }
                        } } } } }
                    }
                }
            }
        } }
        else {
        if ( par == 1 ) {
            if ( use_wtime ) {
                MesPrint("WTime = %7l.%2i sec",millitime,timepart);
            }
            else {
                MesPrint("Time = %7l.%2i sec",millitime,timepart);
            }
        }
        else {
#if ( BITSINLONG > 32 )
            if ( S->GenTerms >= 10000000000L ) {
                if ( use_wtime ) {
                    MesPrint("WTime = %7l.%2i sec   Generated terms = %16l",
                        millitime,timepart,S->GenTerms);
                }
                else {
                    MesPrint("Time = %7l.%2i sec    Generated terms = %16l",
                        millitime,timepart,S->GenTerms);
                }
            }
            else {
                if ( use_wtime ) {
                    MesPrint("WTime = %7l.%2i sec   Generated terms = %10l",
                        millitime,timepart,S->GenTerms);
                }
                else {
                    MesPrint("Time = %7l.%2i sec    Generated terms = %10l",
                        millitime,timepart,S->GenTerms);
                }
            }
#else
            if ( use_wtime ) {
                MesPrint("WTime = %7l.%2i sec   Generated terms = %10l",
                    millitime,timepart,S->GenTerms);
            }
            else {
                MesPrint("Time = %7l.%2i sec    Generated terms = %10l",
                    millitime,timepart,S->GenTerms);
```

```
            }
#endif
        }
#if ( BITSINLONG > 32 )
        if ( par == 0 )
            if ( S->TermsLeft >= 10000000000L ) {
                MesPrint("%16s%8l Terms %s = %16l",EXPRNAME(AR.CurExpr),
                AN.ninterms,FG.swmes[par],S->TermsLeft);
            }
            else {
                MesPrint("%16s%8l Terms %s = %10l",EXPRNAME(AR.CurExpr),
                AN.ninterms,FG.swmes[par],S->TermsLeft);
            }
        else {
            if ( S->TermsLeft >= 10000000000L ) {
#ifdef WITHPTHREADS
                if ( identity > 0 && par == 2 ) {
                    MesPrint("%16s         Terms in thread = %16l",
                    EXPRNAME(AR.CurExpr),S->TermsLeft);
                }
                else
#elif defined(WITHMPI)
                if ( PF.me != MASTER && par == 2 ) {
                    MesPrint("%16s         Terms in process= %16l",
                    EXPRNAME(AR.CurExpr),S->TermsLeft);
                }
                else
#endif
                {
                    MesPrint("%16s         Terms %s = %16l",
                    EXPRNAME(AR.CurExpr),FG.swmes[par],S->TermsLeft);
                }
            }
            else {
#ifdef WITHPTHREADS
                if ( identity > 0 && par == 2 ) {
                    MesPrint("%16s         Terms in thread = %10l",
                    EXPRNAME(AR.CurExpr),S->TermsLeft);
                }
                else
#elif defined(WITHMPI)
                if ( PF.me != MASTER && par == 2 ) {
                    MesPrint("%16s         Terms in process= %10l",
                    EXPRNAME(AR.CurExpr),S->TermsLeft);
                }
                else
#endif
                {
                    MesPrint("%16s         Terms %s = %10l",
                    EXPRNAME(AR.CurExpr),FG.swmes[par],S->TermsLeft);
                }
            }
        }
#else
        if ( par == 0 )
            MesPrint("%16s%8l Terms %s = %10l",EXPRNAME(AR.CurExpr),
            AN.ninterms,FG.swmes[par],S->TermsLeft);
        else {
#ifdef WITHPTHREADS
            if ( identity > 0 && par == 2 ) {
                MesPrint("%16s         Terms in thread = %10l",
                EXPRNAME(AR.CurExpr),S->TermsLeft);
            }
            else
#elif defined(WITHMPI)
            if ( PF.me != MASTER && par == 2 ) {
                MesPrint("%16s         Terms in process= %10l",
                EXPRNAME(AR.CurExpr),S->TermsLeft);
            }
            else
#endif
            {
                MesPrint("%16s         Terms %s = %10l",
                EXPRNAME(AR.CurExpr),FG.swmes[par],S->TermsLeft);
            }
        }
#endif
        SETBASEPOSITION(pp,y);
        if ( ISLESSPOS(*plspace,pp) ) {
            MesPrint("%24s Bytes used      = %10p",AC.Commercial,plspace);
        }
        else {
            y = 1000000000L;
            SETBASEPOSITION(pp,y);
            MULPOS(pp,100);
            if ( ISLESSPOS(*plspace,pp) ) {
                MesPrint("%24s Bytes used      =%11p",AC.Commercial,plspace);
```

```
            }
            else {
                MULPOS(pp,10);
                if ( ISLESSPOS(*plspace,pp) ) {
                MesPrint("%24s Bytes used     =%12p",AC.Commercial,plspace);
                }
                else {
                MULPOS(pp,10);
                if ( ISLESSPOS(*plspace,pp) ) {
                MesPrint("%24s Bytes used    =%13p",AC.Commercial,plspace);
                }
                else {
                MULPOS(pp,10);
                if ( ISLESSPOS(*plspace,pp) ) {
                MesPrint("%24s Bytes used   =%14p",AC.Commercial,plspace);
                }
                else {
                MULPOS(pp,10);
                if ( ISLESSPOS(*plspace,pp) ) {
                MesPrint("%24s Bytes used  =%15p",AC.Commercial,plspace);
                }
                else {
                MULPOS(pp,10);
                if ( ISLESSPOS(*plspace,pp) ) {
                MesPrint("%24s Bytes used =%16p",AC.Commercial,plspace);
                }
                else {
                MULPOS(pp,10);
                if ( ISLESSPOS(*plspace,pp) ) {
                MesPrint("%24s Bytes used=%17p",AC.Commercial,plspace);
                }
                } } } } }
            }
        } }
#ifdef WITHSTATS
        MesPrint("Total number of writes: %l, reads: %l, seeks, %l"
            ,numwrites,numreads,numseeks);
        MesPrint("Total number of mallocs: %l, frees: %l"
            ,nummallocs,numfrees);
#endif
        MUNLOCK(ErrorMessageLock);
    }
}

/*
        #] WriteStats :
        #[ NewSort :                WORD NewSort()
*/
WORD NewSort(PHEAD0)
{
    GETBIDENTITY
    SORTING *S, **newFS;
    int i, newsize;
    if ( AN.SoScratC == 0 )
        AN.SoScratC = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"NewSort");
    AR.sLevel++;
    if ( AR.sLevel >= AN.NumFunSorts ) {
        if ( AN.NumFunSorts == 0 ) newsize = 100;
        else newsize = 2*AN.NumFunSorts;
        newFS = (SORTING **)Malloc1((newsize+1)*sizeof(SORTING *),"FunSort pointers");
        for ( i = 0; i < AN.NumFunSorts; i++ ) newFS[i] = AN.FunSorts[i];
        for ( ; i <= newsize; i++ ) newFS[i] = 0;
        if ( AN.FunSorts ) M_free(AN.FunSorts,"FunSort pointers");
        AN.FunSorts = newFS; AN.NumFunSorts = newsize;
    }
    if ( AR.sLevel == 0 ) {
        AN.FunSorts[0] = AT.S0;
        if ( AR.PolyFun == 0 ) { AT.S0->PolyFlag = 0; }
        else if ( AR.PolyFunType == 1 ) { AT.S0->PolyFlag = 1; }
        else if ( AR.PolyFunType == 2 ) {
            if ( AR.PolyFunExp == 2
              || AR.PolyFunExp == 3 ) AT.S0->PolyFlag = 1;
            else                      AT.S0->PolyFlag = 2;
        }
        AR.ShortSortCount = 0;
    }
    else {
        if ( AN.FunSorts[AR.sLevel] == 0 ) {
            AN.FunSorts[AR.sLevel] = AllocSort(
                AM.SLargeSize,AM.SSmallSize,AM.SSmallEsize,AM.STermsInSmall
                    ,AM.SMaxPatches,AM.SMaxFpatches,AM.SIOsize);
        }
        AN.FunSorts[AR.sLevel]->PolyFlag = 0;
    }
    AT.SS = S = AN.FunSorts[AR.sLevel];
    S->sFill = S->sBuffer;
    S->lFill = S->lBuffer;
```

```
    S->lPatch = 0;
    S->fPatchN = 0;
    S->GenTerms = S->TermsLeft = S->GenSpace = S->SpaceLeft = 0;
    S->PoinFill = S->sPointer;
    *S->PoinFill = S->sFill;
    PUTZERO(S->SizeInFile[0]); PUTZERO(S->SizeInFile[1]); PUTZERO(S->SizeInFile[2]);
    S->sTerms = 0;
    PUTZERO(S->file.POposition);
    S->stage4 = 0;
    if ( AR.sLevel > AN.MaxFunSorts ) AN.MaxFunSorts = AR.sLevel;
/*
    The next variable is for the staged sort only.
    It should be treated differently

    PUTZERO(AN.OldPosOut);
*/
    return(0);
}

/*
        #] NewSort :
        #[ EndSort :                WORD EndSort(PHEAD buffer,par)
*/
LONG EndSort(PHEAD WORD *buffer, int par)
{
  GETBIDENTITY
  SORTING *S = AT.SS;
  WORD j, **ss, *to, *t;
  LONG sSpace, over, tover, spare, retval = 0, jj;
  POSITION position, pp;
  off_t lSpace;
  FILEHANDLE *fout = 0, *oldoutfile = 0, *newout = 0;

  if ( AM.exitflag && AR.sLevel == 0 ) return(0);
#ifdef WITHMPI
  if( (retval = PF_EndSort()) > 0){
    oldoutfile = AR.outfile;
    retval = 0;
    goto RetRetval;
  }
  else if(retval < 0){
    retval = -1;
    goto RetRetval;
  }
    /* PF_EndSort returned 0: for S != AM.S0 and slaves still do the regular sort */
#endif /* WITHMPI */
    oldoutfile = AR.outfile;
/*      PolyFlag repair action
    if ( S == AT.S0 ) {
        if ( AR.PolyFun == 0 ) { S->PolyFlag = 0; }
        else if ( AR.PolyFunType == 1 ) { S->PolyFlag = 1; }
        else if ( AR.PolyFunType == 2 ) {
            if ( AR.PolyFunExp == 2
              || AR.PolyFunExp == 3 ) S->PolyFlag = 1;
            else                      S->PolyFlag = 2;
        }
        S->PolyWise = 0;
    }
    else {
        S->PolyFlag = S->PolyWise = 0;
    }
*/
    S->PolyWise = 0;
    *(S->PoinFill) = 0;

    SplitMerge(BHEAD S->sPointer,S->sTerms);

    sSpace = 0;
    tover = over = S->sTerms;
    ss = S->sPointer;
    if ( over >= 0 ) {
        if ( S->lPatch > 0 || S->file.handle >= 0 ) {
            ss[over] = 0;
            sSpace = ComPress(ss,&spare);
            S->TermsLeft -= over - spare;
            if ( par == 1 ) { AR.outfile = newout = AllocFileHandle(0,(char *)0); }
        }
        else if ( S != AT.S0 ) {
            ss[over] = 0;
            if ( par == 2 ) {
                sSpace = 3;
                while ( ( t = *ss++ ) != 0 ) { sSpace += *t; }
                if ( AN.tryterm > 0 && ( (sSpace+1)*sizeof(WORD) < (size_t)(AM.MaxTer) ) ) {
                    to = TermMalloc("$-sort space");
                }
                else {
                    LONG allocsp = sSpace+1;
```

```
                    if ( allocsp < 20 ) allocsp = 20;
                    allocsp = ((allocsp+7)/8)*8;
                    to = (WORD *)Malloc1(allocsp*sizeof(WORD),"$-sort space");
                    if ( AN.tryterm > 0 ) AN.tryterm = 0;
                }
                *((WORD **)buffer) = to;
                ss = S->sPointer;
                while ( ( t = *ss++ ) != 0 ) {
                    j = *t; while ( --j >= 0 ) *to++ = *t++;
                }
                *to = 0;
                retval = sSpace + 1;
            }
            else {
                to = buffer;
                sSpace = 0;
                while ( ( t = *ss++ ) != 0 ) {
                    j = *t;
                    if ( ( sSpace += j ) > AM.MaxTer/((LONG)sizeof(WORD)) ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Sorted function argument too long.");
                        MUNLOCK(ErrorMessageLock);
                        retval = -1; goto RetRetval;
                    }
                    while ( --j >= 0 ) *to++ = *t++;
                }
                *to = 0;
            }
            goto RetRetval;
        }
        else {
            POSITION oldpos;
            if ( S == AT.S0 ) {
                fout = AR.outfile;
                *AR.CompressPointer = 0;
                SeekScratch(AR.outfile,&position);
            }
            else {
                fout = &(S->file);
                PUTZERO(position);
            }
            oldpos = position;
            S->TermsLeft = 0;
/*
            Here we can go directly to the output.
*/
#ifdef WITHZLIB
            { int oldgzipCompress = AR.gzipCompress;
                AR.gzipCompress = 0;
                /* SetupOutputGZIP(fout); */
#endif
            if ( tover > 0 ) {
                ss = S->sPointer;
                while ( ( t = *ss++ ) != 0 ) {
                    if ( *t ) S->TermsLeft++;
#ifdef WITHPTHREADS
                    if ( AS.MasterSort && ( fout == AR.outfile ) ) { PutToMaster(BHEAD t); }
                    else
#endif
                    if ( PutOut(BHEAD t,&position,fout,1) < 0 ) {
                        retval = -1; goto RetRetval;
                    }
                }
            }
#ifdef WITHPTHREADS
            if ( AS.MasterSort && ( fout == AR.outfile ) ) { PutToMaster(BHEAD 0); }
            else
#endif
            if ( FlushOut(&position,fout,1) ) {
                retval = -1; goto RetRetval;
            }
#ifdef WITHZLIB
                AR.gzipCompress = oldgzipCompress;
            }
#endif
#ifdef WITHPTHREADS
            if ( AS.MasterSort && ( fout == AR.outfile ) ) goto RetRetval;
#endif
#ifdef WITHMPI
            if ( PF.me != MASTER && PF.exprtodo < 0 ) goto RetRetval;
#endif
            DIFPOS(oldpos,position,oldpos);
            S->SpaceLeft = BASEPOSITION(oldpos);
            WriteStats(&oldpos,(WORD)2);
            pp = oldpos;
            goto RetRetval;
        }
```

```
    }
    else if ( par == 1 && newout == 0 ) { AR.outfile = newout = AllocFileHandle(0,(char *)0); }
    sSpace++;
    lSpace = sSpace + (S->lFill - S->lBuffer) - (LONG)S->lPatch*(AM.MaxTer/sizeof(WORD));
/*          Note wrt MaxTer and lPatch: each patch starts with space for decompression */
/*          Not needed if only large buffer, but needed when using files (?) */
    SETBASEPOSITION(pp,lSpace);
    MULPOS(pp,sizeof(WORD));
    if ( S->file.handle >= 0 ) {
        ADD2POS(pp,S->fPatches[S->fPatchN]);
    }
    if ( S == AT.S0 ) {
        WORD oldLogHandle = AC.LogHandle;
        if ( AC.LogHandle >= 0 && AM.LogType && ( ( S->lPatch > 0 )
            || S->file.handle >= 0 ) ) AC.LogHandle = -1;
        if ( S->lPatch > 0 || S->file.handle >= 0 ) { WriteStats(&pp,0); }
        AC.LogHandle = oldLogHandle;
    }
    if ( par == 2 ) { AR.outfile = newout = AllocFileHandle(0,(char *)0); }
    if ( S->lPatch > 0 ) {
        if ( ( S->lPatch >= S->MaxPatches ) ||
            ( ( (WORD *)(((UBYTE *)(S->lFill + sSpace)) + 2*AM.MaxTer) ) >= S->lTop ) ) {
/*
            The large buffer is too full. Merge and write it
*/
#ifdef GZIPDEBUG
            MLOCK(ErrorMessageLock);
            MesPrint("%w EndSort: lPatch = %d, MaxPatches = %d,lFill = %x, sSpace = %ld, MaxTer = %d, lTop = %x"
                    ,S->lPatch,S->MaxPatches,S->lFill,sSpace,AM.MaxTer/sizeof(WORD),S->lTop);
            MUNLOCK(ErrorMessageLock);
#endif

            if ( MergePatches(1) ) {
                MLOCK(ErrorMessageLock);
                MesCall("EndSort");
                MUNLOCK(ErrorMessageLock);
                retval = -1; goto RetRetval;
            }
            S->lPatch = 0;
            pp = S->SizeInFile[1];
            MULPOS(pp,sizeof(WORD));
#ifndef WITHPTHREADS
            if ( S == AT.S0 )
#endif
            {
                WORD oldLogHandle = AC.LogHandle;
                POSITION pppp;
                SETBASEPOSITION(pppp,0);
                SeekFile(S->file.handle,&pppp,SEEK_CUR);
                SeekFile(S->file.handle,&pp,SEEK_END);
                SeekFile(S->file.handle,&pppp,SEEK_SET);
                if ( AC.LogHandle >= 0 && AM.LogType ) AC.LogHandle = -1;
                WriteStats(&pp,(WORD)1);
                AC.LogHandle = oldLogHandle;
                UpdateMaxSize();
            }
        }
        else {
            S->Patches[S->lPatch++] = S->lFill;
            to = (WORD *)(((UBYTE *)(S->lFill)) + AM.MaxTer);
            if ( tover > 0 ) {
                ss = S->sPointer;
                while ( ( t = *ss++ ) != 0 ) {
                    j = *t;
                    if ( j < 0 ) j = t[1] + 2;
                    while ( --j >= 0 ) *to++ = *t++;
                }
            }
            *to++ = 0;
            S->lFill = to;
            if ( S->file.handle < 0 ) {
                if ( MergePatches(2) ) {
                    MLOCK(ErrorMessageLock);
                    MesCall("EndSort");
                    MUNLOCK(ErrorMessageLock);
                    retval = -1; goto RetRetval;
                }
                if ( S == AT.S0 ) {
                    pp = S->SizeInFile[2];
                    MULPOS(pp,sizeof(WORD));
#ifdef WITHPTHREADS
                    if ( AS.MasterSort && ( fout == AR.outfile ) ) goto RetRetval;
#endif
                    WriteStats(&pp,2);
                    UpdateMaxSize();
                }
```

```
                else {
                    if ( par == 2 && newout->handle >= 0 ) {
                        POSITION zeropos;
                        PUTZERO(zeropos);
#ifdef ALLLOCK
                        LOCK(newout->pthreadslock);
#endif
                        SeekFile(newout->handle,&zeropos,SEEK_SET);
                        to = (WORD *)Malloc1(BASEPOSITION(newout->filesize)+sizeof(WORD)*2
                                ,"$-buffer reading");
                        if ( AN.tryterm > 0 ) AN.tryterm = 0;
                        if ( ( retval = ReadFile(newout->handle,(UBYTE
*)to,BASEPOSITION(newout->filesize)) ) !=
                                BASEPOSITION(newout->filesize) ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Error reading information for $ variable");
                            MUNLOCK(ErrorMessageLock);
                            M_free(to,"$-buffer reading");
                            retval = -1;
                        }
                        else {
                            *((WORD **)buffer) = to;
                            retval /= sizeof(WORD);
                        }
#ifdef ALLLOCK
                        UNLOCK(newout->pthreadslock);
#endif
                    }
                    else if ( newout->handle >= 0 ) {   /* output too large */
TooLarge:
                        MLOCK(ErrorMessageLock);
                        MesPrint("(1)Output should fit inside a single term. Increase MaxTermSize?");
                        MesCall("EndSort");
                        MUNLOCK(ErrorMessageLock);
                        retval = -1; goto RetRetval;
                    }
                    else {
                        t = newout->PObuffer;
                        if ( par == 2 ) {
                            jj = newout->POfill - t;
                            if ( AN.tryterm > 0 && ( (jj+2)*sizeof(WORD) < (size_t)(AM.MaxTer) ) ) {
                                to = TermMalloc("$-sort space");
                            }
                            else {
                                LONG allocsp = jj+2;
                                if ( allocsp < 20 ) allocsp = 20;
                                allocsp = ((allocsp+7)/8)*8;
                                to = (WORD *)Malloc1(allocsp*sizeof(WORD),"$-sort space");
                                if ( AN.tryterm > 0 ) AN.tryterm = 0;
                            }
                            *((WORD **)buffer) = to;
                            NCOPY(to,t,jj);
                        }
                        else {
                            j = newout->POfill - t;
                            to = buffer;
                            if ( to >= AT.WorkSpace && to < AT.WorkTop && to+j > AT.WorkTop )
                                goto WorkSpaceError;
                            if ( j > AM.MaxTer ) goto TooLarge;
                            NCOPY(to,t,j);
                        }
                    }
                }
                goto RetRetval;
            }
            if ( MergePatches(1) ) { /* --> SortFile */
                MLOCK(ErrorMessageLock);
                MesCall("EndSort");
                MUNLOCK(ErrorMessageLock);
                retval = -1; goto RetRetval;
            }
            UpdateMaxSize();
            pp = S->SizeInFile[1];
            MULPOS(pp,sizeof(WORD));
#ifndef WITHPTHREADS
            if ( S == AT.S0 )
#endif
            {
                WORD oldLogHandle = AC.LogHandle;
                POSITION pppp;
                SETBASEPOSITION(pppp,0);
                SeekFile(S->file.handle,&pppp,SEEK_CUR);
                SeekFile(S->file.handle,&pp,SEEK_END);
                SeekFile(S->file.handle,&pppp,SEEK_SET);
                if ( AC.LogHandle >= 0 && AM.LogType ) AC.LogHandle = -1;
                WriteStats(&pp,(WORD)1);
                AC.LogHandle = oldLogHandle;
```

```
            }
#ifdef WITHERRORXXX
            if ( S != AT.S0 ) {
/*
                This is wrong! We have sorted to the sort file.
                Things are not sitting in the output yet.
*/
                if ( newout->handle >= 0 ) goto TooLarge;
                t = newout->PObuffer;
                j = newout->POfill - t;
                to = buffer;
                if ( to >= AT.WorkSpace && to < AT.WorkTop && to+j > AT.WorkTop )
                    goto WorkSpaceError;
                if ( j > AM.MaxTer ) goto TooLarge;
                NCOPY(to,t,j);
                goto RetRetval;
            }
#endif
        }
    }
    if ( S->file.handle >= 0 ) {
#ifdef GZIPDEBUG
        MLOCK(ErrorMessageLock);
        MesPrint("%w EndSort: fPatchN = %d, lPatch = %d, position = %12p"
            ,S->fPatchN,S->lPatch,&(S->fPatches[S->fPatchN]));
        MUNLOCK(ErrorMessageLock);
#endif
        if ( S->lPatch <= 0 ) {
            StageSort(&(S->file));
            position = S->fPatches[S->fPatchN];
            ss = S->sPointer;
            if ( *ss ) {
#ifdef WITHZLIB
                *AR.CompressPointer = 0;
                if ( S == AT.S0 && AR.NoCompress == 0 && AR.gzipCompress > 0 )
                    S->fpcompressed[S->fPatchN] = 1;
                else
                    S->fpcompressed[S->fPatchN] = 0;
                SetupOutputGZIP(&(S->file));
#endif
                while ( ( t = *ss++ ) != 0 ) {
                    if ( PutOut(BHEAD t,&position,&(S->file),1) < 0 ) {
                        retval = -1; goto RetRetval;
                    }
                }
                if ( FlushOut(&position,&(S->file),1) ) {
                    retval = -1; goto RetRetval;
                }
                ++(S->fPatchN);
                S->fPatches[S->fPatchN] = position;
                UpdateMaxSize();
#ifdef GZIPDEBUG
                MLOCK(ErrorMessageLock);
                MesPrint("%w EndSort+: fPatchN = %d, lPatch = %d, position = %12p"
                    ,S->fPatchN,S->lPatch,&(S->fPatches[S->fPatchN]));
                MUNLOCK(ErrorMessageLock);
#endif
            }
        }
        AR.Stage4Name = 0;
#ifdef WITHPTHREADS
        if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
            if ( S->file.handle >= 0 ) {
                SynchFile(S->file.handle);
            }
        }
#endif
        UpdateMaxSize();
        if ( MergePatches(0) ) {
            MLOCK(ErrorMessageLock);
            MesCall("EndSort");
            MUNLOCK(ErrorMessageLock);
            retval = -1; goto RetRetval;
        }
        S->stage4 = 0;
#ifdef WITHPTHREADS
        if ( AS.MasterSort && ( fout == AR.outfile ) ) goto RetRetval;
#endif
        pp = S->SizeInFile[0];
        MULPOS(pp,sizeof(WORD));
        WriteStats(&pp,2);
        UpdateMaxSize();
    }
RetRetval:

#ifdef WITHMPI
    /* NOTE: PF_EndSort has been changed such that it sets S->TermsLeft. (TU 30 Jun 2011) */
```

```
    if ( AR.sLevel == 0 && (PF.me == MASTER || PF.exprtodo >= 0) ) {
        Expressions[AR.CurExpr].counter = S->TermsLeft;
        Expressions[AR.CurExpr].size = pp;
    }
#else
    if ( AR.sLevel == 0 ) {
        Expressions[AR.CurExpr].counter = S->TermsLeft;
        Expressions[AR.CurExpr].size = pp;
    }/*if ( AR.sLevel == 0 )*/
#endif
/*:[25nov2003 mt]*/
    if ( S->file.handle >= 0 && ( par != 1 ) && ( par != 2 ) ) {
                /* sortfile is still open */
        UpdateMaxSize();
        CloseFile(S->file.handle);
        S->file.handle = -1;
        remove(S->file.name);
#ifdef GZIPDEBUG
        MLOCK(ErrorMessageLock);
        MesPrint("%wEndSort: sortfile %s removed",S->file.name);
        MUNLOCK(ErrorMessageLock);
#endif
    }
    AR.outfile = oldoutfile;
    AR.sLevel--;
    if ( AR.sLevel >= 0 ) AT.SS = AN.FunSorts[AR.sLevel];
    if ( par == 1 ) {
        if ( retval < 0 ) {
            UpdateMaxSize();
            if ( newout ) {
                DeAllocFileHandle(newout);
                newout = 0;
            }
        }
        else if ( newout ) {
          if ( newout->handle >= 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("(2)Output should fit inside a single term. Increase MaxTermSize?");
            MesCall("EndSort");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
          }
          else if ( newout->POfill > newout->PObuffer ) {
/*
            Here we have to copy the contents of the 'file' into
            the buffer. We assume that this buffer lies in the WorkSpace.
            Hence
*/
            j = newout->POfill-newout->PObuffer;
            if ( buffer >= AT.WorkSpace && buffer < AT.WorkTop && buffer+j > AT.WorkTop )
                goto WorkSpaceError;
            else {
                to = buffer; t = newout->PObuffer;
                while ( j-- > 0 ) *to++ = *t++;
            }
            UpdateMaxSize();
          }
          DeAllocFileHandle(newout);
          newout = 0;
        }
    }
    else if ( par == 2 ) {
        if ( newout ) {
            if ( retval == 0 ) {
              if ( newout->handle >= 0 ) {
/*
                output resides at the moment in a file
                Find the size, make a buffer, copy into the buffer and clean up.
*/
                POSITION zeropos;
                PUTZERO(position);
#ifdef ALLLOCK
                LOCK(newout->pthreadslock);
#endif
                SeekFile(newout->handle,&position,SEEK_END);
                PUTZERO(zeropos);
                SeekFile(newout->handle,&zeropos,SEEK_SET);
                to = (WORD *)Malloc1(BASEPOSITION(position)+sizeof(WORD)*3
                        ,"$-buffer reading");
                if ( AN.tryterm > 0 ) AN.tryterm = 0;
                if ( ( retval = ReadFile(newout->handle,(UBYTE *)to,BASEPOSITION(position)) ) !=
                BASEPOSITION(position) ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Error reading information for $ variable");
                    MUNLOCK(ErrorMessageLock);
                    M_free(to,"$-buffer reading");
                    retval = -1;
```

```
                }
                else {
                    *((WORD **)buffer) = to;
                    retval /= sizeof(WORD);
                }
#ifdef ALLLOCK
                UNLOCK(newout->pthreadslock);
#endif
              }
              else {
/*
                output resides in the cache buffer and the file was never opened
*/
                LONG wsiz = newout->POfill - newout->PObuffer;
                if ( AN.tryterm > 0 && ( (wsiz+2)*sizeof(WORD) < (size_t)(AM.MaxTer) ) ) {
                    to = TermMalloc("$-sort space");
                }
                else {
                    LONG allocsp = wsiz+2;
                    if ( allocsp < 20 ) allocsp = 20;
                    allocsp = ((allocsp+7)/8)*8;
                    to = (WORD *)Malloc1(allocsp*sizeof(WORD),"$-buffer reading");
                    if ( AN.tryterm > 0 ) AN.tryterm = 0;
                }
                *((WORD **)buffer) = to; t = newout->PObuffer;
                retval = wsiz;
                NCOPY(to,t,wsiz);
              }
            }
            UpdateMaxSize();
            DeAllocFileHandle(newout);
            newout = 0;
        }
    }
    else {
        if ( newout ) {
            DeAllocFileHandle(newout);
            newout = 0;
        }
    }
    return(retval);
WorkSpaceError:
    MLOCK(ErrorMessageLock);
    MesWork();
    MesCall("EndSort");
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(-1);
}

/*
        #] EndSort :
        #[ PutIn :                  LONG PutIn(handle,position,buffer,take,npat)
*/
LONG PutIn(FILEHANDLE *file, POSITION *position, WORD *buffer, WORD **take, int npat)
{
    LONG i, RetCode;
    WORD *from, *to;
#ifndef WITHZLIB
    DUMMYUSE(npat);
#endif
    from = buffer + ( file->POsize * sizeof(UBYTE) )/sizeof(WORD);
    i = from - *take;
    if ( i*((LONG)(sizeof(WORD))) > AM.MaxTer ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Problems in PutIn");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    to = buffer;
    while ( --i >= 0 ) *--to = *--from;
    *take = to;
#ifdef WITHZLIB
    if ( ( RetCode = FillInputGZIP(file,position,(UBYTE *)buffer
                                   ,file->POsize,npat) ) < 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("PutIn: We have RetCode = %x while reading %x bytes",
            RetCode,file->POsize);
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
#else
#ifdef ALLLOCK
    LOCK(file->pthreadslock);
#endif
    SeekFile(file->handle,position,SEEK_SET);
    if ( ( RetCode = ReadFile(file->handle,(UBYTE *)buffer,file->POsize) ) < 0 ) {
```

```
#ifdef ALLLOCK
        UNLOCK(file->pthreadslock);
#endif
        MLOCK(ErrorMessageLock);
        MesPrint("PutIn: We have RetCode = %x while reading %x bytes",
            RetCode,file->POsize);
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
#ifdef ALLLOCK
    UNLOCK(file->pthreadslock);
#endif
#endif
    return(RetCode);
}

/*
        #] PutIn :
        #[ Sflush :                 WORD Sflush(file)
*/
WORD Sflush(FILEHANDLE *fi)
{
    LONG size, RetCode;
#ifdef WITHZLIB
    GETIDENTITY
    int dobracketindex = 0;
    if ( AR.sLevel <= 0 && Expressions[AR.CurExpr].newbracketinfo
        && ( fi == AR.outfile || fi == AR.hidefile ) ) dobracketindex = 1;
#endif
    if ( fi->handle < 0 ) {
        if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
#ifdef GZIPDEBUG
            MLOCK(ErrorMessageLock);
            MesPrint("%w Sflush created scratch file %s",fi->name);
            MUNLOCK(ErrorMessageLock);
#endif
            fi->handle = (WORD)RetCode;
            PUTZERO(fi->filesize);
            PUTZERO(fi->POposition);
        }
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("Cannot create scratch file %s",fi->name);
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
    }
#ifdef WITHZLIB
    if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
    && dobracketindex == 0 ) {
        if ( FlushOutputGZIP(fi) ) return(-1);
        fi->POfill = fi->PObuffer;
    }
    else
#endif
    {
#ifdef ALLLOCK
      LOCK(fi->pthreadslock);
#endif
      size = (fi->POfill-fi->PObuffer)*sizeof(WORD);
      SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
      if ( WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),size) != size ) {
#ifdef ALLLOCK
        UNLOCK(fi->pthreadslock);
#endif
        MLOCK(ErrorMessageLock);
        MesPrint("Write error while finishing sort. Disk full?");
        MUNLOCK(ErrorMessageLock);
        return(-1);
      }
      ADDPOS(fi->filesize,size);
      ADDPOS(fi->POposition,size);
      fi->POfill = fi->PObuffer;
#ifdef ALLLOCK
      UNLOCK(fi->pthreadslock);
#endif
    }
    return(0);
}

/*
        #] Sflush :
        #[ PutOut :                 WORD PutOut(term,position,file,ncomp)
*/
WORD PutOut(PHEAD WORD *term, POSITION *position, FILEHANDLE *fi, WORD ncomp)
{
    GETBIDENTITY
```

```
    WORD i, *p, ret, *r, *rr, j, k, first;
    int dobracketindex = 0;
    LONG RetCode;

    if ( AT.SS != AT.S0 ) {
/*
        For this case no compression should be used
*/
        if ( ( i = *term ) <= 0 ) return(0);
        ret = i;
        ADDPOS(*position,i*sizeof(WORD));
        p = fi->POfill;
        do {
            if ( p >= fi->POstop ) {
                if ( fi->handle < 0 ) {
                    if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
#ifdef GZIPDEBUG
                        MLOCK(ErrorMessageLock);
                        MesPrint("%w PutOut created sortfile %s",fi->name);
                        MUNLOCK(ErrorMessageLock);
#endif
                        fi->handle = (WORD)RetCode;
                        PUTZERO(fi->filesize);
                        PUTZERO(fi->POposition);
#ifdef WITHZLIB
                        fi->ziobuffer = 0;
#endif
                    }
                    else {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Cannot create scratch file %s",fi->name);
                        MUNLOCK(ErrorMessageLock);
                        return(-1);
                    }
                }
#ifdef ALLLOCK
                LOCK(fi->pthreadslock);
#endif
                if ( fi == AR.hidefile ) {
                    LOCK(AS.inputslock);
                }
                SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
                if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize) ) !=
    fi->POsize ) {
                    if ( fi == AR.hidefile ) {
                        UNLOCK(AS.inputslock);
                    }
#ifdef ALLLOCK
                    UNLOCK(fi->pthreadslock);
#endif
                    MLOCK(ErrorMessageLock);
                    MesPrint("Write error during sort. Disk full?");
                    MesPrint("Attempt to write %l bytes on file %d at position %15p",
                                fi->POsize,fi->handle,&(fi->POposition));
                    MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->PObuffer));
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                ADDPOS(fi->filesize,fi->POsize);
                p = fi->PObuffer;
                ADDPOS(fi->POposition,fi->POsize);
                if ( fi == AR.hidefile ) {
                    UNLOCK(AS.inputslock);
                }
#ifdef ALLLOCK
                UNLOCK(fi->pthreadslock);
#endif
#ifdef WITHPTHREADS
                if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
                    if ( fi->handle >= 0 ) SynchFile(fi->handle);
                }
#endif
            }
            *p++ = *term++;
        } while ( --i > 0 );
        fi->POfull = fi->POfill = p;
        return(ret);
    }
    if ( ( AP.PreDebug & DUMPOUTTERMS ) == DUMPOUTTERMS ) {
            MLOCK(ErrorMessageLock);
#ifdef WITHPTHREADS
            sprintf((char *)(THRbuf),"PutOut(%d)",AT.identity);
            PrintTerm(term,(char *)(THRbuf));
#else
            PrintTerm(term,"PutOut");
#endif
            MesPrint("ncomp = %d, AR.NoCompress = %d, AR.sLevel = %d",ncomp,AR.NoCompress,AR.sLevel);
```

```
            MesPrint("File %s, position %p",fi->name,position);
            MUNLOCK(ErrorMessageLock);
    }

    if ( AR.sLevel <= 0 && Expressions[AR.CurExpr].newbracketinfo
        && ( fi == AR.outfile || fi == AR.hidefile ) ) dobracketindex = 1;
    r = rr = AR.CompressPointer;
    first = j = k = ret = 0;
    if ( ( i = *term ) != 0 ) {
        if ( i < 0 ) {          /* Compressed term */
            i = term[1] + 2;
            if ( fi == AR.outfile || fi == AR.hidefile ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Ran into precompressed term");
                MUNLOCK(ErrorMessageLock);
                Crash();
                return(-1);
            }
        }
        else if ( !AR.NoCompress && ( ncomp > 0 ) && AR.sLevel <= 0 ) { /* Must compress */
            if ( dobracketindex ) {
                PutBracketInIndex(BHEAD term,position);
            }
            j = *r++ - 1;
            p = term + 1;
            i--;
            if ( AR.PolyFun ) {
                WORD *polystop, *sa;
                sa = p + i;
                sa -= ABS(sa[-1]);
                polystop = p;
                while ( polystop < sa && *polystop != AR.PolyFun ) {
                    polystop += polystop[1];
                }
                if ( polystop < sa ) {
                    if ( AR.PolyFunType == 2 ) polystop[2] &= ~MUSTCLEANPRF;
                    while ( i > 0 && j > 0 && *p == *r && p < polystop ) {
                        i--; j--; k--; p++; r++;
                    }
                }
                else {
                    while ( i > 0 && j > 0 && *p == *r && p < sa ) { i--; j--; k--; p++; r++; }
                }
            }
            else {
                WORD *sa;
                sa = p + i;
                sa -= ABS(sa[-1]);
                while ( i > 0 && j > 0 && *p == *r && p < sa ) { i--; j--; k--; p++; r++; }
            }
            if ( k > -2 ) {
nocompress:
                j = i = *term;
                k = 0;
                p = term;
                r = rr;
                NCOPY(r,p,j);
            }
            else {
                *rr = *term;
                term = p;
                j = i;
                NCOPY(r,p,j);
                j = i;
                i += 2;
                first = 2;
            }
/*                  Sabotage getting into the coefficient next time */
            r[-(ABS(r[-1]))] = 0;
            if ( r >= AR.ComprTop ) {
                MLOCK(ErrorMessageLock);
                MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
                MUNLOCK(ErrorMessageLock);
                Crash();
                return(-1);
            }
        }
        else if ( !AR.NoCompress && ( ncomp < 0 ) && AR.sLevel <= 0 ) {
                /* No compress but put in compress buffer anyway */
            if ( dobracketindex ) {
                PutBracketInIndex(BHEAD term,position);
            }
            j = *r++ - 1;
            p = term + 1;
            i--;
            if ( AR.PolyFun ) {
                WORD *polystop, *sa;
```

```
                sa = p + i;
                sa -= ABS(sa[-1]);
                polystop = p;
                while ( polystop < sa && *polystop != AR.PolyFun ) {
                    polystop += polystop[1];
                }
                if ( polystop < sa ) {
                    if ( AR.PolyFunType == 2 ) polystop[2] &= ~MUSTCLEANPRF;
                    while ( i > 0 && j > 0 && *p == *r && p < polystop ) {
                        i--; j--; k--; p++; r++;
                    }
                }
                else {
                    while ( i > 0 && j > 0 && *p == *r ) { i--; j--; k--; p++; r++; }
                }
            }
            else {
                while ( i > 0 && j > 0 && *p == *r ) { i--; j--; k--; p++; r++; }
            }
            goto nocompress;
        }
        else {
            if ( AR.PolyFunType == 2 ) {
                WORD *t, *tstop;
                tstop = term + *term;
                tstop -= ABS(tstop[-1]);
                t = term+1;
                while ( t < tstop ) {
                    if ( *t == AR.PolyFun ) {
                        t[2] &= ~MUSTCLEANPRF;
                    }
                    t += t[1];
                }
            }
            if ( dobracketindex ) {
                PutBracketInIndex(BHEAD term,position);
            }
        }
        ret = i;
        ADDPOS(*position,i*sizeof(WORD));
        p = fi->POfill;
        do {
            if ( p >= fi->POstop ) {
#ifdef WITHMPI /* [16mar1998 ar] */
              if ( PF.me != MASTER && AR.sLevel <= 0 && (fi == AR.outfile || fi == AR.hidefile) && PF.parallel && PF.exprtodo < 0 ) {
                PF_BUFFER *sbuf = PF.sbuf;
                sbuf->fill[sbuf->active] = fi->POstop;
                PF_ISendSbuf(MASTER,PF_BUFFER_MSGTAG);
                p = fi->PObuffer = fi->POfill = fi->POfull =
                  sbuf->buff[sbuf->active];
                fi->POstop = sbuf->stop[sbuf->active];
              }
              else
#endif /* WITHMPI [16mar1998 ar] */
              {
                if ( fi->handle < 0 ) {
                    if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
#ifdef GZIPDEBUG
                        MLOCK(ErrorMessageLock);
                        MesPrint("%w PutOut created sortfile %s",fi->name);
                        MUNLOCK(ErrorMessageLock);
#endif
                        fi->handle = (WORD)RetCode;
                        PUTZERO(fi->filesize);
                        PUTZERO(fi->POposition);
#ifdef WITHZLIB
                        fi->ziobuffer = 0;
#endif
                    }
                    else {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Cannot create scratch file %s",fi->name);
                        MUNLOCK(ErrorMessageLock);
                        return(-1);
                    }
                }
#ifdef WITHZLIB
                if ( !AR.NoCompress && ncomp > 0 && AR.gzipCompress > 0
                    && dobracketindex == 0 && fi->zsp != 0 ) {
                    fi->POfill = p;
                    if ( PutOutputGZIP(fi) ) return(-1);
                    p = fi->PObuffer;
                }
                else
#endif
                {
```

```
#ifdef ALLLOCK
                    LOCK(fi->pthreadslock);
#endif
                    if ( fi == AR.hidefile ) {
                      LOCK(AS.inputslock);
                    }
                    SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
                    if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize) ) !=
      fi->POsize ) {
                      if ( fi == AR.hidefile ) {
                          UNLOCK(AS.inputslock);
                      }
#ifdef ALLLOCK
                      UNLOCK(fi->pthreadslock);
#endif
                      MLOCK(ErrorMessageLock);
                      MesPrint("Write error during sort. Disk full?");
                      MesPrint("Attempt to write %l bytes on file %d at position %15p",
                                  fi->POsize,fi->handle,&(fi->POposition));
                      MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->PObuffer));
                      MUNLOCK(ErrorMessageLock);
                      return(-1);
                    }
                    ADDPOS(fi->filesize,fi->POsize);
                    p = fi->PObuffer;
                    ADDPOS(fi->POposition,fi->POsize);
                    if ( fi == AR.hidefile ) {
                      UNLOCK(AS.inputslock);
                    }
#ifdef ALLLOCK
                    UNLOCK(fi->pthreadslock);
#endif
#ifdef WITHPTHREADS
                    if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
                      if ( fi->handle >= 0 ) SynchFile(fi->handle);
                    }
#endif
                  }
                }
              }
              if ( first ) {
                  if ( first == 2 ) *p++ = k;
                  else *p++ = j;
                  first--;
              }
              else *p++ = *term++;
/*
              if ( AP.DebugFlag ) {
                  TalToLine((UWORD)(p[-1])); TokenToLine((UBYTE *)"  ");
              }
*/
          } while ( --i > 0 );
          fi->POfull = fi->POfill = p;
      }
/*
      if ( AP.DebugFlag ) {
          AO.OutSkip = 0;
          FiniLine();
      }
*/
      return(ret);
}

/*
          #] PutOut :
          #[ FlushOut :                WORD FlushOut(position,file,compr)
*/
WORD FlushOut(POSITION *position, FILEHANDLE *fi, int compr)
{
      GETIDENTITY
      LONG size, RetCode;
      int dobracketindex = 0;
#ifndef WITHZLIB
      DUMMYUSE(compr);
#endif
      if ( AR.sLevel <= 0 && Expressions[AR.CurExpr].newbracketinfo
          && ( fi == AR.outfile || fi == AR.hidefile ) ) dobracketindex = 1;
#ifdef WITHMPI /* [16mar1998 ar] */
      if ( PF.me != MASTER && AR.sLevel <= 0 && (fi == AR.outfile || fi == AR.hidefile) && PF.parallel
      && PF.exprtodo < 0 ) {
          PF_BUFFER *sbuf = PF.sbuf;
          if ( fi->POfill >= fi->POstop ){
            sbuf->fill[sbuf->active] = fi->POstop;
            PF_ISendSbuf(MASTER,PF_BUFFER_MSGTAG);
            fi->POfull = fi->POfill = fi->PObuffer = sbuf->buff[sbuf->active];
            fi->POstop = sbuf->stop[sbuf->active];
          }
```

```
        *(fi->POfill)++ = 0;
        sbuf->fill[sbuf->active] = fi->POfill;
        PF_ISendSbuf(MASTER,PF_ENDBUFFER_MSGTAG);
        fi->PObuffer = fi->POfill = fi->POfull = sbuf->buff[sbuf->active];
        fi->POstop = sbuf->stop[sbuf->active];
        return(0);
    }
#endif /* WITHMPI [16mar1998 ar] */
    if ( fi->POfill >= fi->POstop ) {
        if ( fi->handle < 0 ) {
            if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
#ifdef GZIPDEBUG
                MLOCK(ErrorMessageLock);
                MesPrint("%w FlushOut created scratch file %s",fi->name);
                MUNLOCK(ErrorMessageLock);
#endif
                PUTZERO(fi->filesize);
                PUTZERO(fi->POposition);
                fi->handle = (WORD)RetCode;
#ifdef WITHZLIB
                fi->ziobuffer = 0;
#endif
            }
            else {
                MLOCK(ErrorMessageLock);
                MesPrint("Cannot create scratch file %s",fi->name);
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
        }
#ifdef WITHZLIB
        if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
        && dobracketindex == 0 && ( compr > 0 ) && fi->zsp != 0 ) {
            if ( PutOutputGZIP(fi) ) return(-1);
            fi->POfill = fi->PObuffer;
        }
        else
#endif
        {
#ifdef ALLLOCK
          LOCK(fi->pthreadslock);
#endif
          if ( fi == AR.hidefile ) {
            LOCK(AS.inputslock);
          }
          SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
          if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize) ) != fi->POsize )
    {
#ifdef ALLLOCK
            UNLOCK(fi->pthreadslock);
#endif
            if ( fi == AR.hidefile ) {
                UNLOCK(AS.inputslock);
            }
            MLOCK(ErrorMessageLock);
            MesPrint("Write error while sorting. Disk full?");
            MesPrint("Attempt to write %l bytes on file %d at position %15p",
                        fi->POsize,fi->handle,&(fi->POposition));
            MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->PObuffer));
            MUNLOCK(ErrorMessageLock);
            return(-1);
          }
          ADDPOS(fi->filesize,fi->POsize);
          fi->POfill = fi->PObuffer;
          ADDPOS(fi->POposition,fi->POsize);
          if ( fi == AR.hidefile ) {
            UNLOCK(AS.inputslock);
          }
#ifdef ALLLOCK
          UNLOCK(fi->pthreadslock);
#endif
#ifdef WITHPTHREADS
          if ( AS.MasterSort && AC.ThreadSortFileSynch && fi != AR.hidefile ) {
            if ( fi->handle >= 0 ) SynchFile(fi->handle);
          }
#endif
        }
    }
    *(fi->POfill)++ = 0;
    fi->POfull = fi->POfill;
/*
    {
        UBYTE OutBuf[140];
        if ( AP.DebugFlag ) {
            AO.OutFill = AO.OutputLine = OutBuf;
            AO.OutSkip = 3;
            FiniLine();
```

```
            TokenToLine((UBYTE *)"End of expression written");
            FiniLine();
        }
    }
*/
    size = (fi->POfill-fi->PObuffer)*sizeof(WORD);
    if ( fi->handle >= 0 ) {
#ifdef WITHZLIB
        if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
        && dobracketindex == 0 && ( compr > 0 ) && fi->zsp != 0 ) {
            if ( FlushOutputGZIP(fi) ) return(-1);
            fi->POfill = fi->PObuffer;
        }
        else
#endif
        {
#ifdef ALLLOCK
          LOCK(fi->pthreadslock);
#endif
          if ( fi == AR.hidefile ) {
            LOCK(AS.inputslock);
          }
          SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
/*
          MesPrint("FlushOut: writing %l bytes to position %12p",size,&(fi->POposition));
*/
          if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),size) ) != size ) {
#ifdef ALLLOCK
            UNLOCK(fi->pthreadslock);
#endif
            if ( fi == AR.hidefile ) {
                UNLOCK(AS.inputslock);
            }
            MLOCK(ErrorMessageLock);
            MesPrint("Write error while finishing sorting. Disk full?");
            MesPrint("Attempt to write %l bytes on file %d at position %15p",
                        size,fi->handle,&(fi->POposition));
            MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->PObuffer));
            MUNLOCK(ErrorMessageLock);
            return(-1);
          }
          ADDPOS(fi->filesize,size);
          ADDPOS(fi->POposition,size);
          fi->POfill = fi->PObuffer;
          if ( fi == AR.hidefile ) {
            UNLOCK(AS.inputslock);
          }
#ifdef ALLLOCK
          UNLOCK(fi->pthreadslock);
#endif
#ifdef WITHPTHREADS
          if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
            if ( fi->handle >= 0 ) SynchFile(fi->handle);
          }
#endif
        }
    }
    if ( dobracketindex ) {
        BRACKETINFO *b = Expressions[AR.CurExpr].newbracketinfo;
        if ( b->indexfill > 0 ) {
            DIFPOS(b->indexbuffer[b->indexfill-1].next,*position,Expressions[AR.CurExpr].onfile);
        }
    }
#ifdef WITHZLIB
    if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
        && dobracketindex == 0 && ( compr > 0 ) && fi->zsp != 0 ) {
        PUTZERO(*position);
        if ( fi->handle >= 0 ) {
#ifdef ALLLOCK
            LOCK(fi->pthreadslock);
#endif
            SeekFile(fi->handle,position,SEEK_END);
#ifdef ALLLOCK
            UNLOCK(fi->pthreadslock);
#endif
        }
        else {
            ADDPOS(*position,((UBYTE *)fi->POfill-(UBYTE *)fi->PObuffer));
        }
    }
    else
#endif
    {
        ADDPOS(*position,sizeof(WORD));
    }
    return(0);
}
```

```
/*
        #] FlushOut :
        #[ AddCoef :                WORD AddCoef(pterm1,pterm2)
*/
WORD AddCoef(PHEAD WORD **ps1, WORD **ps2)
{
    GETBIDENTITY
    SORTING *S = AT.SS;
    WORD *s1, *s2;
    WORD l1, l2, i;
    WORD OutLen, *t, j;
    UWORD *OutCoef;
    OutCoef = AN.SoScratC;
    s1 = *ps1; s2 = *ps2;
    GETCOEF(s1,l1);
    GETCOEF(s2,l2);
    if ( AddRat(BHEAD (UWORD *)s1,l1,(UWORD *)s2,l2,OutCoef,&OutLen) ) {
        MLOCK(ErrorMessageLock);
        MesCall("AddCoef");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    if ( AN.ncmod != 0 ) {
        if ( ( AC.modmode & POSNEG ) != 0 ) {
            NormalModulus(OutCoef,&OutLen);
/*
            We had forgotten that this can also become smaller but the
            denominator isn't there. Correct in the other case
            17-may-2009 [JV]
*/
            j = ABS(OutLen); OutCoef[j] = 1;
            for ( i = 1; i < j; i++ ) OutCoef[j+i] = 0;
        }
        else if ( BigLong(OutCoef,OutLen,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
            SubPLon(OutCoef,OutLen,(UWORD *)AC.cmod,ABS(AN.ncmod),OutCoef,&OutLen);
            OutCoef[OutLen] = 1;
            for ( i = 1; i < OutLen; i++ ) OutCoef[OutLen+i] = 0;
        }
    }
    if ( !OutLen ) { *ps1 = *ps2 = 0; return(0); }
    OutLen «= 1;
    if ( OutLen < 0 ) i = - ( --OutLen );
    else              i = ++OutLen;
    if ( l1 < 0 ) l1 = -l1;
    l1 «= 1; l1++;
    if ( i <= l1 ) {    /* Fits in 1 */
        l1 -= i;
        **ps1 -= l1;
        s2 = (WORD *)OutCoef;
        while ( --i > 0 ) *s1++ = *s2++;
        *s1++ = OutLen;
        while ( --l1 >= 0 ) *s1++ = 0;
        goto RegEnd;
    }
    if ( l2 < 0 ) l2 = -l2;
    l2 «= 1; l2++;
    if ( i <= l2 ) {    /* Fits in 2 */
        l2 -= i;
        **ps2 -= l2;
        s1 = (WORD *)OutCoef;
        while ( --i > 0 ) *s2++ = *s1++;
        *s2++ = OutLen;
        while ( --l2 >= 0 ) *s2++ = 0;
        *ps1 = *ps2;
        goto RegEnd;
    }

    /* Doesn't fit. Make a new term. */

    t = s1;
    s1 = *ps1;
    j = *s1++ + i - l1;     /* Space needed */
    if ( (S->sFill + j) >= S->sTop2 ) {
        GarbHand();

        s1 = *ps1;
        t = s1 + *s1 - 1;
        j = *s1++ + i - l1;     /* Space needed */
        l1 = *t;
        if ( l1 < 0 ) l1  = - l1;
        t -= l1-1;
    }
    s2 = S->sFill;
    *s2++ = j;
    while ( s1 < t ) *s2++ = *s1++;
    s1 = (WORD *)OutCoef;
```

```
    while ( --i > 0 ) *s2++ = *s1++;
    *s2++ = OutLen;
    *ps1 = S->sFill;
    S->sFill = s2;
RegEnd:
    *ps2 = 0;
    if ( **ps1 > AM.MaxTer/((LONG)(sizeof(WORD))) ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Term to complex after polynomial addition. MaxTermSize = %10l",
        AM.MaxTer/sizeof(WORD));
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    return(1);
}

/*
        #] AddCoef :
        #[ AddPoly :                WORD AddPoly(pterm1,pterm2)
*/
WORD AddPoly(PHEAD WORD **ps1, WORD **ps2)
{
    GETBIDENTITY
    SORTING *S = AT.SS;
    WORD i;
    WORD *s1, *s2, *m, *w, *t, oldpw = S->PolyWise;
    s1 = *ps1 + S->PolyWise;
    s2 = *ps2 + S->PolyWise;
    w = AT.WorkPointer;
/*
    Add here the two arguments. Is a straight merge.
*/
    if ( S->PolyFlag == 2 && AR.PolyFunExp != 2 && AR.PolyFunExp != 3 ) {
        WORD **oldSplitScratch = AN.SplitScratch;
        LONG oldSplitScratchSize = AN.SplitScratchSize;
        LONG oldInScratch = AN.InScratch;
        WORD oldtype = AR.SortType;
        if ( (WORD *)((UBYTE *)w + AM.MaxTer) >= AT.WorkTop ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Program was adding polyratfun arguments");
            MesWork();
            MUNLOCK(ErrorMessageLock);
        }
        AR.SortType = SORTHIGHFIRST;
        S->PolyWise = 0;
        AN.SplitScratch = AN.SplitScratch1;
        AN.SplitScratchSize = AN.SplitScratchSize1;
        AN.InScratch = AN.InScratch1;
        poly_ratfun_add(BHEAD s1,s2);
        S->PolyWise = oldpw;
        AN.SplitScratch1 = AN.SplitScratch;
        AN.SplitScratchSize1 = AN.SplitScratchSize;
        AN.InScratch1 = AN.InScratch;
        AN.SplitScratch = oldSplitScratch;
        AN.SplitScratchSize = oldSplitScratchSize;
        AN.InScratch = oldInScratch;
        AT.WorkPointer = w;
        AR.SortType = oldtype;
        if ( w[1] <= FUNHEAD ||
            ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) ) {
            *ps1 = *ps2 = 0; return(0);
        }
    }
    else {
        if ( w + s1[1] + s2[1] + 12 + ARGHEAD >= AT.WorkTop ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Program was adding polyfun arguments");
            MesWork();
            MUNLOCK(ErrorMessageLock);
        }
        AddArgs(BHEAD s1,s2,w);
    }
/*
    Now we need to store the result in a convenient place.
*/
    if ( w[1] <= FUNHEAD ) { *ps1 = *ps2 = 0; return(0); }
    if ( w[1] <= s1[1] || w[1] <= s2[1] ) {   /* Fits in place. */
        if ( w[1] > s1[1] ) {
            *ps1 = *ps2;
            s1 = s2;
        }
        t = s1 + s1[1];
        m = *ps1 + **ps1;
        i = w[1];
        NCOPY(s1,w,i);
        if ( s1 != t ) {
            while ( t < m ) *s1++ = *t++;
```

```
            **ps1 = WORDDIF(s1,(*ps1));
        }
        *ps2 = 0;
    }
    else {      /* Make new term */
#ifdef TESTGARB
        s2 = *ps2;
#endif
        *ps2 = 0;
        if ( (S->sFill + (**ps1 + w[1] - s1[1])) >= S->sTop2 ) {
#ifdef TESTGARB
            MesPrint("------Garbage collection-------");
#endif
            AT.WorkPointer += w[1];
            GarbHand();
            AT.WorkPointer = w;
            s1 = *ps1;
            if ( (S->sFill + (**ps1 + w[1] - s1[1])) >= S->sTop2 ) {
#ifdef TESTGARB
                UBYTE OutBuf[140];
                MLOCK(ErrorMessageLock);
                AO.OutFill = AO.OutputLine = OutBuf;
                AO.OutSkip = 3;
                FiniLine();
                i = *s2;
                while ( --i >= 0 ) {
                    TalToLine((UWORD)(*s2++)); TokenToLine((UBYTE *)"  ");
                }
                FiniLine();
                AO.OutFill = AO.OutputLine = OutBuf;
                AO.OutSkip = 3;
                FiniLine();
                s2 = *ps1;
                i = *s2;
                while ( --i >= 0 ) {
                    TalToLine((UWORD)(*s2++)); TokenToLine((UBYTE *)"  ");
                }
                FiniLine();
                AO.OutFill = AO.OutputLine = OutBuf;
                AO.OutSkip = 3;
                FiniLine();
                s2 = w;
                i = w[1];
                while ( --i >= 0 ) {
                    TalToLine((UWORD)(*s2++)); TokenToLine((UBYTE *)"  ");
                }
                FiniLine();
                MesPrint("Please increase SmallExtension in %s",setupfilename);
                MUNLOCK(ErrorMessageLock);
#else
                MLOCK(ErrorMessageLock);
                MesPrint("Please increase SmallExtension in %s",setupfilename);
                MUNLOCK(ErrorMessageLock);
#endif
                Terminate(-1);
            }
        }
        t = *ps1;
        s2 = S->sFill;
        m = s2;
        i = S->PolyWise;
        NCOPY(s2,t,i);
        i = w[1];
        NCOPY(s2,w,i);
        t = t + t[1];
        w = *ps1 + **ps1;
        while ( t < w ) *s2++ = *t++;
        *m = WORDDIF(s2,m);
        *ps1 = m;
        S->sFill = s2;
        if ( *m > AM.MaxTer/((LONG)sizeof(WORD)) ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Term to complex after polynomial addition. MaxTermSize = %101",
            AM.MaxTer/sizeof(WORD));
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
    }
    return(1);
}

/*
        #] AddPoly :
        #[ AddArgs :                VOID AddArgs(arg1,arg2,to)
*/

#define INSLENGTH(x)  w[1] = FUNHEAD+ARGHEAD+x; w[FUNHEAD] = ARGHEAD+x;
```

```
VOID AddArgs(PHEAD WORD *s1, WORD *s2, WORD *m)
{
    GETBIDENTITY
    WORD i1, i2;
    WORD *w = m, *mm, *t, *t1, *t2, *tstop1, *tstop2;
    WORD tempterm[8+FUNHEAD];

    *m++ = AR.PolyFun; *m++ = 0; FILLFUN(m)
    *m++ = 0; *m++ = 0; FILLARG(m)
    if ( s1[FUNHEAD] < 0 || s2[FUNHEAD] < 0 ) {
        if ( s1[FUNHEAD] < 0 ) {
            if ( s2[FUNHEAD] < 0 ) {    /* Both are special */
                if ( s1[FUNHEAD] <= -FUNCTION ) {
                    if ( s2[FUNHEAD] == s1[FUNHEAD] ) {
                        *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
                        FILLFUN(m)
                        *m++ = 2; *m++ = 1; *m++ = 3;
                        INSLENGTH(4+FUNHEAD)
                    }
                    else if ( s2[FUNHEAD] <= -FUNCTION ) {
                        i1 = functions[-FUNCTION-s1[FUNHEAD]].commute != 0;
                        i2 = functions[-FUNCTION-s2[FUNHEAD]].commute != 0;
                        if ( ( !i1 && i2 ) || ( i1 == i2 && i1 > i2 ) ) {
                            i1 = s2[FUNHEAD];
                            s2[FUNHEAD] = s1[FUNHEAD];
                            s1[FUNHEAD] = i1;
                        }
                        *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
                        FILLFUN(m)
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        *m++ = 4+FUNHEAD; *m++ = -s2[FUNHEAD]; *m++ = FUNHEAD;
                        FILLFUN(m)
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        INSLENGTH(8+2*FUNHEAD)
                    }
                    else if ( s2[FUNHEAD] == -SYMBOL ) {
                        *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s2[FUNHEAD+1]; *m++ = 1;
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
                        FILLFUN(m)
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        INSLENGTH(12+FUNHEAD)
                    }
                    else {      /* number */
                        *m++ = 4;
                        *m++ = ABS(s2[FUNHEAD+1]); *m++ = 1; *m++ = s2[FUNHEAD+1] < 0 ? -3: 3;
                        *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
                        FILLFUN(m)
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        INSLENGTH(8+FUNHEAD)
                    }
                }
                else if ( s1[FUNHEAD] == -SYMBOL ) {
                    if ( s2[FUNHEAD] == s1[FUNHEAD] ) {
                        if ( s1[FUNHEAD+1] == s2[FUNHEAD+1] ) {
                            *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s1[FUNHEAD+1];
                            *m++ = 1; *m++ = 2; *m++ = 1; *m++ = 3;
                            INSLENGTH(8)
                        }
                        else {
                            if ( s1[FUNHEAD+1] > s2[FUNHEAD+1] )
                                { i1 = s2[FUNHEAD+1]; i2 = s1[FUNHEAD+1]; }
                            else { i1 = s1[FUNHEAD+1]; i2 = s2[FUNHEAD+1]; }
                            *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = i1;
                            *m++ = 1; *m++ = 1; *m++ = 1; *m++ = 3;
                            *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = i2;
                            *m++ = 1; *m++ = 1; *m++ = 1; *m++ = 3;
                            INSLENGTH(16)
                        }
                    }
                    else if ( s2[FUNHEAD] <= -FUNCTION ) {
                        *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s1[FUNHEAD+1]; *m++ = 1;
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        *m++ = 4+FUNHEAD; *m++ = -s2[FUNHEAD]; *m++ = FUNHEAD;
                        FILLFUN(m)
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        INSLENGTH(12+FUNHEAD)
                    }
                    else {
                        *m++ = 4;
                        *m++ = ABS(s2[FUNHEAD+1]); *m++ = 1; *m++ = s2[FUNHEAD+1] < 0 ? -3: 3;
                        *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s1[FUNHEAD+1]; *m++ = 1;
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        INSLENGTH(12)
                    }
                }
```

```
                else {  /* Must be -SNUMBER! */
                    if ( s2[FUNHEAD] <= -FUNCTION ) {
                        *m++ = 4;
                        *m++ = ABS(s1[FUNHEAD+1]); *m++ = 1; *m++ = s1[FUNHEAD+1] < 0 ? -3: 3;
                        *m++ = 4+FUNHEAD; *m++ = -s2[FUNHEAD]; *m++ = FUNHEAD;
                        FILLFUN(m)
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        INSLENGTH(8+FUNHEAD)
                    }
                    else if ( s2[FUNHEAD] == -SYMBOL ) {
                        *m++ = 4;
                        *m++ = ABS(s1[FUNHEAD+1]); *m++ = 1; *m++ = s1[FUNHEAD+1] < 0 ? -3: 3;
                        *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s2[FUNHEAD+1]; *m++ = 1;
                        *m++ = 1; *m++ = 1; *m++ = 3;
                        INSLENGTH(12)
                    }
                    else {      /* Both are numbers. add. */
                        LONG x1;
                        x1 = (LONG)s1[FUNHEAD+1] + (LONG)s2[FUNHEAD+1];
                        if ( x1 < 0 ) { i1 = (WORD)(-x1); i2 = -3; }
                        else { i1 = (WORD)x1; i2 = 3; }
                        if ( x1 && AN.ncmod != 0 ) {
                            m[0] = 4;
                            m[1] = i1;
                            m[2] = 1;
                            m[3] = i2;
                            if ( Modulus(m) ) Terminate(-1);
                            if ( *m == 0 ) w[1] = 0;
                            else {
                                if ( *m == 4 && ( m[1] & MAXPOSITIVE ) == m[1]
                                && m[3] == 3 ) {
                                    i1 = m[1];
                                    m -= ARGHEAD;
                                    *m++ = -SNUMBER;
                                    *m++ = i1;
                                    INSLENGTH(4)
                                }
                                else {
                                    INSLENGTH(*m)
                                    m += *m;
                                }
                            }
                        }
                        else {
                            if ( x1 == 0 ) {
                                w[1] = FUNHEAD;
                            }
                            else if ( ( i1 & MAXPOSITIVE ) == i1 ) {
                                m -= ARGHEAD;
                                *m++ = -SNUMBER;
                                *m++ = (WORD)x1;
                                w[1] = FUNHEAD+2;
                            }
                            else {
                                *m++ = 4; *m++ = i1; *m++ = 1; *m++ = i2;
                                INSLENGTH(4)
                            }
                        }
                    }
                }
            }
            else {  /* Only s1 is special */
s1only:
/*
            Compose a term in `tempterm'
*/
                t = tempterm;
                if ( s1[FUNHEAD] <= -FUNCTION ) {
                    *t++ = 4+FUNHEAD; *t++ = -s1[FUNHEAD]; *t++ = FUNHEAD;
                    FILLFUN(t)
                    *t++ = 1; *t++ = 1; *t++ = 3;
                }
                else if ( s1[FUNHEAD] == -SYMBOL ) {
                    *t++ = 8; *t++ = SYMBOL; *t++ = 4;
                    *t++ = s1[FUNHEAD+1]; *t++ = 1;
                    *t++ = 1; *t++ = 1; *t++ = 3;
                }
                else {
                    *t++ = 4;   *t++ = ABS(s1[FUNHEAD+1]);
                    *t++ = 1;   *t++ = s1[FUNHEAD+1] < 0 ? -3: 3;
                }
                tstop1 = t;
                s1 = tempterm;
                goto twogen;
            }
        }
        else {      /* Only s2 is special */
```

```
            t = s1;
            s1 = s2;
            s2 = t;
            goto s1only;
        }
    }
    else {
        int oldPolyFlag;
        tstop1 = s1 + s1[1];
        s1 += FUNHEAD+ARGHEAD;
twogen:
        tstop2 = s2 + s2[1];
        s2 += FUNHEAD+ARGHEAD;
/*
        Now we should merge the expressions in s1 and s2 into m.
*/
        oldPolyFlag = AT.SS->PolyFlag;
        AT.SS->PolyFlag = 0;
        while ( s1 < tstop1 && s2 < tstop2 ) {
            i1 = CompareTerms(BHEAD s1,s2,(WORD)(-1));
            if ( i1 > 0 ) {
                i2 = *s1;
                NCOPY(m,s1,i2);
            }
            else if ( i1 < 0 ) {
                i2 = *s2;
                NCOPY(m,s2,i2);
            }
            else {  /* Coefficients should be added. */
                WORD i;
                t = s1+*s1;
                i1 = t[-1];
                i2 = *s1 - ABS(i1);
                t2 = s2 + i2;
                s2 += *s2;
                mm = m;
                NCOPY(m,s1,i2);
                t1 = s1;
                s1 = t;
                i2 = s2[-1];
/*
                t1,i1 is the first coefficient
                t2,i2 is the second coefficient
                It should be placed at m,i1
*/
                i1 = REDLENG(i1);
                i2 = REDLENG(i2);
                if ( AddRat(BHEAD (UWORD *)t1,i1,(UWORD *)t2,i2,(UWORD *)m,&i) ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Addition of coefficients of PolyFun");
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                if ( i == 0 ) {
                    m = mm;
                }
                else {
                    i1 = INCLENG(i);
                    m += ABS(i1);
                    m[-1] = i1;
                    *mm = WORDDIF(m,mm);
                    if ( AN.ncmod != 0 ) {
                        if ( Modulus(mm) ) Terminate(-1);
                        if ( !*mm ) m = mm;
                        else m = mm + *mm;
                    }
                }
            }
        }
        while ( s1 < tstop1 ) *m++ = *s1++;
        while ( s2 < tstop2 ) *m++ = *s2++;
        w[1] = WORDDIF(m,w);
        w[FUNHEAD] = w[1] - FUNHEAD;
        if ( ToFast(w+FUNHEAD,w+FUNHEAD) ) {
            if ( w[FUNHEAD] <= -FUNCTION ) w[1] = FUNHEAD+1;
            else w[1] = FUNHEAD+2;
            if ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) w[1] = FUNHEAD;
        }
/*      AT.SS->PolyFlag = AR.PolyFunType;*/
        AT.SS->PolyFlag = oldPolyFlag;
    }
}

/*
        #] AddArgs :
        #[ Compare1 :               WORD Compare1(term1,term2,level)
*/
```

```
WORD Compare1(PHEAD WORD *term1, WORD *term2, WORD level)
{
    GETBIDENTITY
    SORTING *S = AT.SS;
    WORD *stopper1, *stopper2, *t2;
    WORD *s1, *s2, *t1;
    WORD *stopex1, *stopex2;
    WORD c1, c2;
    WORD prevorder;
    WORD count = -1, localPoly, polyhit = -1;

    if ( S->PolyFlag ) {
/*
        if ( S->PolyWise != 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("S->PolyWise is not zero!!!!!");
            MUNLOCK(ErrorMessageLock);
        }
*/
        count = 0; localPoly = 1; S->PolyWise = polyhit = 0;
        S->PolyFlag = AR.PolyFunType;
        if ( AR.PolyFunType == 2 &&
             ( AR.PolyFunExp == 2 || AR.PolyFunExp == 3 ) ) S->PolyFlag = 1;
    }
    else { localPoly = 0; }
    prevorder = 0;
    GETSTOP(term1,s1);
    stopper1 = s1;
    GETSTOP(term2,stopper2);
    t1 = term1 + 1;
    t2 = term2 + 1;
    while ( t1 < stopper1 && t2 < stopper2 ) {
        if ( *t1 != *t2 ) {
            if ( *t1 == HAAKJE ) return(PREV(-1));
            if ( *t2 == HAAKJE ) return(PREV(1));
            if ( *t1 >= (FUNCTION-1) ) {
                if ( *t2 < (FUNCTION-1) ) return(PREV(-1));
                if ( *t1 < FUNCTION && *t2 < FUNCTION ) return(PREV(*t2-*t1));
                if ( *t1 < FUNCTION ) return(PREV(1));
                if ( *t2 < FUNCTION ) return(PREV(-1));
                c1 = functions[*t1-FUNCTION].commute;
                c2 = functions[*t2-FUNCTION].commute;
                if ( !c1 ) {
                    if ( c2 ) return(PREV(1));
                    else return(PREV(*t2-*t1));
                }
                else {
                    if ( !c2 ) return(PREV(-1));
                    else return(PREV(*t2-*t1));
                }
            }
            else return(PREV(*t2-*t1));
        }
        s1 = t1 + 2;
        s2 = t2 + 2;
        c1 = *t1;
        t1 += t1[1];
        t2 += t2[1];
        if ( localPoly && c1 < FUNCTION ) {
            polyhit = 1;
        }
        if ( c1 <= (FUNCTION-1)
        || ( c1 >= FUNCTION && functions[c1-FUNCTION].spec ) ) {
            if ( c1 == SYMBOL ) {
                if ( *s1 == FACTORSYMBOL && *s2 == FACTORSYMBOL
                 && s1[-1] == 4 && s2[-1] == 4
                 && ( ( t1 < stopper1 && *t1 == HAAKJE )
                 || ( t1 == stopper1 && AT.fromindex ) ) ) {
/*
                    We have to be very careful with the criteria here, because
                    Compare1 is called both in the regular sorting and by the
                    routine that makes the bracket index. In the last case
                    there is no HAAKJE subterm.
*/
                    if ( s1[1] != s2[1] ) return(s2[1]-s1[1]);
                    s1 += 2; s2 += 2;
                }
                else if ( AR.SortType >= SORTPOWERFIRST ) {
                    WORD i1 = 0, *r1;
                    r1 = s1;
                    while ( s1 < t1 ) { i1 += s1[1]; s1 += 2; }
                    s1 = r1; r1 = s2;
                    while ( s2 < t2 ) { i1 -= s2[1]; s2 += 2; }
                    s2 = r1;
                    if ( i1 ) {
                        if ( AR.SortType >= SORTANTIPOWER ) i1 = -i1;
                        return(PREV(i1));
```

```
                    }
                }
                while ( s1 < t1 ) {
                    if ( s2 >= t2 ) {
/*                      return(PREV(1));  */
                        if ( AR.SortType==SORTLOWFIRST ) {
                            return(PREV((s1[1]>0?-1:1)));
                        }
                        else {
                            return(PREV((s1[1]<0?-1:1)));
                        }
                    }
                    if ( *s1 != *s2 ) {
/*                      return(PREV(*s2-*s1)); */
                        if ( AR.SortType==SORTLOWFIRST ) {
                            if ( *s1 < *s2 ) {
                                return(PREV((s1[1]<0?1:-1)));
                            }
                            else {
                                return(PREV((s2[1]<0?-1:1)));
                            }
                        }
                        else {
                            if ( *s1 < *s2 ) {
                                return(PREV((s1[1]<0?-1:1)));
                            }
                            else {
                                return(PREV((s2[1]<0?1:-1)));
                            }
                        }
                    }
                    s1++; s2++;
                    if ( *s1 != *s2 ) return(
                        PREV((AR.SortType==SORTLOWFIRST?*s2-*s1:*s1-*s2)));
                    s1++; s2++;
                }
                if ( s2 < t2 ) {
/*                  return(PREV(-1));  */
                    if ( AR.SortType==SORTLOWFIRST ) {
                        return(PREV((s2[1]<0?-1:1)));
                    }
                    else {
                        return(PREV((s2[1]<0?1:-1)));
                    }
                }
            }
            else if ( c1 == DOTPRODUCT ) {
                if ( AR.SortType >= SORTPOWERFIRST ) {
                    WORD i1 = 0, *r1;
                    r1 = s1;
                    while ( s1 < t1 ) { i1 += s1[2]; s1 += 3; }
                    s1 = r1; r1 = s2;
                    while ( s2 < t2 ) { i1 -= s2[2]; s2 += 3; }
                    s2 = r1;
                    if ( i1 ) {
                        if ( AR.SortType >= SORTANTIPOWER ) i1 = -i1;
                        return(PREV(i1));
                    }
                }
                while ( s1 < t1 ) {
                    if ( s2 >= t2 ) return(PREV(1));
                    if ( *s1 != *s2 ) return(PREV(*s2-*s1));
                    s1++; s2++;
                    if ( *s1 != *s2 ) return(PREV(*s2-*s1));
                    s1++; s2++;
                    if ( *s1 != *s2 ) return(
                        PREV((AR.SortType==SORTLOWFIRST?*s2-*s1:*s1-*s2)));
                    s1++; s2++;
                }
                if ( s2 < t2 ) return(PREV(-1));
            }
            else {
                while ( s1 < t1 ) {
                    if ( s2 >= t2 ) return(PREV(1));
                    if ( *s1 != *s2 ) return(PREV(*s2-*s1));
                    s1++; s2++;
                }
                if ( s2 < t2 ) return(PREV(-1));
            }
        }
        else {
#if FUNHEAD != 2
            s1 += FUNHEAD-2;
            s2 += FUNHEAD-2;
#endif
            if ( localPoly && c1 == AR.PolyFun ) {
                if ( count == 0 ) {
```

```
                    if ( S->PolyFlag == 1 ) {
                      WORD i1, i2;
                      if ( *s1 > 0 ) i1 = *s1;
                      else if ( *s1 <= -FUNCTION ) i1 = 1;
                      else i1 = 2;
                      if ( *s2 > 0 ) i2 = *s2;
                      else if ( *s2 <= -FUNCTION ) i2 = 1;
                      else i2 = 2;
                      if ( s1+i1 == t1 && s2+i2 == t2 ) { /* This is the stuff */
/*
                          Test for scalar nature
*/
                          if ( !polyhit ) {
                              WORD *u1, *u2, *ustop;
                              if ( *s1 < 0 ) {
                                  if ( *s1 != -SNUMBER && *s1 != -SYMBOL && *s1 > -FUNCTION )
                                      goto NoPoly;
                              }
                              else {
                                  u1 = s1 + ARGHEAD;
                                  while ( u1 < t1 ) {
                                      u2 = u1 + *u1;
                                      ustop = u2 - ABS(u2[-1]);
                                      u1++;
                                      while ( u1 < ustop ) {
                                          if ( *u1 == INDEX ) goto NoPoly;
                                          u1 += u1[1];
                                      }
                                      u1 = u2;
                                  }
                              }
                              if ( *s2 < 0 ) {
                                  if ( *s2 != -SNUMBER && *s2 != -SYMBOL && *s2 > -FUNCTION )
                                      goto NoPoly;
                              }
                              else {
                                  u1 = s2 + ARGHEAD;
                                  while ( u1 < t2 ) {
                                      u2 = u1 + *u1;
                                      ustop = u2 - ABS(u2[-1]);
                                      u1++;
                                      while ( u1 < ustop ) {
                                          if ( *u1 == INDEX ) goto NoPoly;
                                          u1 += u1[1];
                                      }
                                      u1 = u2;
                                  }
                              }
                          }
                          S->PolyWise = WORDDIF(s1,term1);
                          S->PolyWise -= FUNHEAD;
                          count = 1;
                          continue;
                      }
                      else {
NoPoly:
                          S->PolyWise = localPoly = 0;
                      }
                    }
                    else if ( AR.PolyFunType == 2 ) {
                      WORD i1, i2, i1a, i2a;
                      if ( *s1 > 0 ) i1 = *s1;
                      else if ( *s1 <= -FUNCTION ) i1 = 1;
                      else i1 = 2;
                      if ( *s2 > 0 ) i2 = *s2;
                      else if ( *s2 <= -FUNCTION ) i2 = 1;
                      else i2 = 2;
                      if ( s1[i1] > 0 ) i1a = s1[i1];
                      else if ( s1[i1] <= -FUNCTION ) i1a = 1;
                      else i1a = 2;
                      if ( s2[i2] > 0 ) i2a = s2[i2];
                      else if ( s2[i2] <= -FUNCTION ) i2a = 1;
                      else i2a = 2;
                      if ( s1+i1+i1a == t1 && s2+i2+i2a == t2 ) { /* This is the stuff */
/*
                          Test for scalar nature
*/
                          if ( !polyhit ) {
                              WORD *u1, *u2, *ustop;
                              if ( *s1 < 0 ) {
                                  if ( *s1 != -SNUMBER && *s1 != -SYMBOL && *s1 > -FUNCTION )
                                      goto NoPoly;
                              }
                              else {
                                  u1 = s1 + ARGHEAD;
                                  while ( u1 < s1+i1 ) {
                                      u2 = u1 + *u1;
```

```
                                    ustop = u2 - ABS(u2[-1]);
                                    u1++;
                                    while ( u1 < ustop ) {
                                        if ( *u1 == INDEX ) goto NoPoly;
                                        u1 += u1[1];
                                    }
                                    u1 = u2;
                                }
                            }
                            if ( s1[i1] < 0 ) {
                                if ( s1[i1] != -SNUMBER && s1[i1] != -SYMBOL && s1[i1] > -FUNCTION )
                                    goto NoPoly;
                            }
                            else {
                                u1 = s1 +i1 + ARGHEAD;
                                while ( u1 < t1 ) {
                                    u2 = u1 + *u1;
                                    ustop = u2 - ABS(u2[-1]);
                                    u1++;
                                    while ( u1 < ustop ) {
                                        if ( *u1 == INDEX ) goto NoPoly;
                                        u1 += u1[1];
                                    }
                                    u1 = u2;
                                }
                            }
                            if ( *s2 < 0 ) {
                                if ( *s2 != -SNUMBER && *s2 != -SYMBOL && *s2 > -FUNCTION )
                                    goto NoPoly;
                            }
                            else {
                                u1 = s2 + ARGHEAD;
                                while ( u1 < s2+i2 ) {
                                    u2 = u1 + *u1;
                                    ustop = u2 - ABS(u2[-1]);
                                    u1++;
                                    while ( u1 < ustop ) {
                                        if ( *u1 == INDEX ) goto NoPoly;
                                        u1 += u1[1];
                                    }
                                    u1 = u2;
                                }
                            }
                            if ( s2[i2] < 0 ) {
                                if ( s2[i2] != -SNUMBER && s2[i2] != -SYMBOL && s2[i2] > -FUNCTION )
                                    goto NoPoly;
                            }
                            else {
                                u1 = s2 + i2 + ARGHEAD;
                                while ( u1 < t2 ) {
                                    u2 = u1 + *u1;
                                    ustop = u2 - ABS(u2[-1]);
                                    u1++;
                                    while ( u1 < ustop ) {
                                        if ( *u1 == INDEX ) goto NoPoly;
                                        u1 += u1[1];
                                    }
                                    u1 = u2;
                                }
                            }
                        }
                        S->PolyWise = WORDDIF(s1,term1);
                        S->PolyWise -= FUNHEAD;
                        count = 1;
                        continue;
                    }
                    else {
                        S->PolyWise = localPoly = 0;
                    }
                  }
                  else {
                    S->PolyWise = localPoly = 0;
                  }
                }
                else {
                    t1 = term1 + S->PolyWise;
                    t2 = term2 + S->PolyWise;
                    S->PolyWise = 0;
                    localPoly = 0;
                    continue;
                }
            }
            while ( s1 < t1 ) {
/*
                The next statement was added 9-nov-2001. It made a bad error
*/
                if ( s2 >= t2 ) return(PREV(-1));
```

```
/*
                There is a little problem here with fast arguments
                We don't want to sacrifice speed, but we like to
                keep a rational ordering. This last one suffers in
                the solution that has been choosen here.
*/
                if ( AC.properorderflag ) {
                    WORD oldpolyflag;
                    oldpolyflag = S->PolyFlag;
                    S->PolyFlag = 0;
                    if ( ( c2 = -CompArg(s1,s2) ) != 0 ) {
                        S->PolyFlag = oldpolyflag; return(PREV(c2));
                    }
                    S->PolyFlag = oldpolyflag;
                    NEXTARG(s1)
                    NEXTARG(s2)
                }
                else {
                    if ( *s1 > 0 ) {
                        if ( *s2 > 0 ) {
                            WORD oldpolyflag;
                            stopex1 = s1 + *s1;
                            if ( s2 >= t2 ) return(PREV(-1));
                            stopex2 = s2 + *s2;
                            s1 += ARGHEAD; s2 += ARGHEAD;
                            oldpolyflag = S->PolyFlag;
                            S->PolyFlag = 0;
                            while ( s1 < stopex1 ) {
                                if ( s2 >= stopex2 ) {
                                    S->PolyFlag = oldpolyflag; return(PREV(-1));
                                }
                                if ( ( c2 = CompareTerms(BHEAD s1,s2,(WORD)1) ) != 0 ) {
                                    S->PolyFlag = oldpolyflag; return(PREV(c2));
                                }
                                s1 += *s1;
                                s2 += *s2;
                            }
                            S->PolyFlag = oldpolyflag;
                            if ( s2 < stopex2 ) return(PREV(1));
                        }
                        else return(PREV(1));
                    }
                    else {
                        if ( *s2 > 0 ) return(PREV(-1));
                        if ( *s1 != *s2 ) { return(PREV(*s1-*s2)); }
                        if ( *s1 > -FUNCTION ) {
                            if ( *++s1 != *++s2 ) { return(PREV(*s2-*s1)); }
                        }
                        s1++; s2++;
                    }
                }
            }
            if ( s2 < t2 ) return(PREV(1));
        }
    }
    {
        if ( AR.SortType != SORTLOWFIRST ) {
            if ( t1 < stopper1 ) return(PREV(1));
            if ( t2 < stopper2 ) return(PREV(-1));
        }
        else {
            if ( t1 < stopper1 ) return(PREV(-1));
            if ( t2 < stopper2 ) return(PREV(1));
        }
    }
    if ( level == 3 ) return(CompCoef(term1,term2));
    if ( level >= 1 )
        return(CompCoef(term2,term1));
    return(0);
}

/*
        #] Compare1 :
        #[ CompareSymbols :         int CompareSymbols(term1,term2,par)
*/
int CompareSymbols(PHEAD WORD *term1, WORD *term2, WORD par)
{
    int sum1, sum2;
    WORD *t1, *t2, *tt1, *tt2;
    int low, high;
    DUMMYUSE(par);
    if ( AR.SortType == SORTLOWFIRST ) { low = 1; high = -1; }
    else { low = -1; high = 1; }
    t1 = term1 + 1; tt1 = term1+*term1; tt1 -= ABS(tt1[-1]); t1 += 2;
    t2 = term2 + 1; tt2 = term2+*term2; tt2 -= ABS(tt2[-1]); t2 += 2;
    if ( AN.polysortflag > 0 ) {
        sum1 = 0; sum2 = 0;
```

```
        while ( t1 < tt1 ) { sum1 += t1[1]; t1 += 2; }
        while ( t2 < tt2 ) { sum2 += t2[1]; t2 += 2; }
        if ( sum1 < sum2 ) return(low);
        if ( sum1 > sum2 ) return(high);
        t1 = term1+3; t2 = term2 + 3;
    }
    while ( t1 < tt1 && t2 < tt2 ) {
        if ( *t1 > *t2 ) return(low);
        if ( *t1 < *t2 ) return(high);
        if ( t1[1] < t2[1] ) return(low);
        if ( t1[1] > t2[1] ) return(high);
        t1 += 2; t2 += 2;
    }
    if ( t1 < tt1 ) return(high);
    if ( t2 < tt2 ) return(low);
    return(0);
}

/*
        #] CompareSymbols :
        #[ CompareHSymbols :        int CompareHSymbols(term1,term2,par)
*/
int CompareHSymbols(PHEAD WORD *term1, WORD *term2, WORD par)
{
    WORD *t1, *t2, *tt1, *tt2, *ttt1, *ttt2;
    DUMMYUSE(par);
    DUMMYUSE(AT.WorkPointer);
    t1 = term1 + 1; tt1 = term1+*term1; tt1 -= ABS(tt1[-1]); t1 += 2;
    t2 = term2 + 1; tt2 = term2+*term2; tt2 -= ABS(tt2[-1]); t2 += 2;
    while ( t1 < tt1 && t2 < tt2 ) {
        if ( *t1 != *t2 ) {
            if ( t1[0] < t2[0] ) return(-1);
            return(1);
        }
        else if ( *t1 == HAAKJE ) {
            t1 += 3; t2 += 3; continue;
        }
        ttt1 = t1+t1[1]; ttt2 = t2+t2[1];
        while ( t1 < ttt1 && t2 < ttt2 ) {
            if ( *t1 > *t2 ) return(-1);
            if ( *t1 < *t2 ) return(1);
            if ( t1[1] < t2[1] ) return(-1);
            if ( t1[1] > t2[1] ) return(1);
            t1 += 2; t2 += 2;
        }
        if ( t1 < ttt1 ) return(1);
        if ( t2 < ttt2 ) return(-1);
    }
    if ( t1 < tt1 ) return(1);
    if ( t2 < tt2 ) return(-1);
    return(0);
}

/*
        #] CompareHSymbols :
        #[ ComPress :               LONG ComPress(ss,n)
*/
LONG ComPress(WORD **ss, LONG *n)
{
    GETIDENTITY
    WORD *t, *s, j, k;
    LONG size = 0;
    int newsize, i;
/*
            #[ debug :

    WORD **sss = ss;

    if ( AP.DebugFlag ) {
        UBYTE OutBuf[140];
        MLOCK(ErrorMessageLock);
        MesPrint("ComPress:");
        AO.OutFill = AO.OutputLine = OutBuf;
        AO.OutSkip = 3;
        FiniLine();
        ss = sss;
        while ( *ss ) {
            s = *ss++;
            j = *s;
            if ( j < 0 ) {
                j = s[1] + 2;
            }
            while ( --j >= 0 ) {
                TalToLine((UWORD)(*s++)); TokenToLine((UBYTE *)"  ");
            }
            FiniLine();
        }
```

```
        AO.OutSkip = 0;
        FiniLine();
        MUNLOCK(ErrorMessageLock);
        ss = sss;
    }

            #] debug :
*/
    *n = 0;
    if ( AT.SS == AT.S0 && !AR.NoCompress ) {
        if ( AN.compressSize == 0 ) {
            if ( *ss ) { AN.compressSize = **ss + 64; }
            else       { AN.compressSize = AM.MaxTer/sizeof(WORD) + 2; }
            AN.compressSpace = (WORD *)Malloc1(AN.compressSize*sizeof(WORD),"Compression");
        }
        AN.compressSpace[0] = 0;
        while ( *ss ) {
            k = 0;
            s = *ss;
            j = *s++;
            if ( j > AN.compressSize ) {
                newsize = j + 64;
                t = (WORD *)Malloc1(newsize*sizeof(WORD),"Compression");
                t[0] = 0;
                if ( AN.compressSpace ) {
                    for ( i = 0; i < *AN.compressSpace; i++ ) t[i] = AN.compressSpace[i];
                    M_free(AN.compressSpace,"Compression");
                }
                AN.compressSpace = t;
                AN.compressSize = newsize;
            }
            t = AN.compressSpace;
            i = *t - 1;
            *t++ = j; j--;
            if ( AR.PolyFun ) {
                WORD *polystop, *sa;
                sa = s + j;
                sa -= ABS(sa[-1]);
                polystop = s;
                while ( polystop < sa && *polystop != AR.PolyFun ) {
                    polystop += polystop[1];
                }
                while ( i > 0 && j > 0 && *s == *t && s < polystop ) {
                    i--; j--; s++; t++; k--;
                }
            }
            else {
                WORD *sa;
                sa = s + j;
                sa -= ABS(sa[-1]);
                while ( i > 0 && j > 0 && *s == *t && s < sa ) { i--; j--; s++; t++; k--; }
            }
            if ( k < -1 ) {
                s[-1] = j;
                s[-2] = k;
                *ss = s-2;
                size += j + 2;
            }
            else {
                size += *AN.compressSpace;
                if ( k == -1 ) { t--; s--; j++; }
            }
            while ( --j >= 0 ) *t++ = *s++;
/*                  Sabotage getting into the coefficient next time */
            t = AN.compressSpace + *AN.compressSpace;
            t[-(ABS(t[-1]))] = 0;
            ss++;
            (*n)++;
        }
    }
    else {
        while ( *ss ) {
            size += *(*ss++);
            (*n)++;
        }
    }
/*
            #[ debug :

    if ( AP.DebugFlag ) {
        UBYTE OutBuf[140];
        AO.OutFill = AO.OutputLine = OutBuf;
        AO.OutSkip = 3;
        FiniLine();
        ss = sss;
        while ( *ss ) {
            s = *ss++;
```

```
            j = *s;
            if ( j < 0 ) {
                j = s[1] + 2;
            }
            while ( --j >= 0 ) {
                TalToLine((UWORD)(*s++)); TokenToLine((UBYTE *)"  ");
            }
            FiniLine();
        }
        AO.OutSkip = 0;
        FiniLine();
    }

            #] debug :
*/
    return(size);
}

/*
        #] ComPress :
        #[ SplitMerge :             VOID SplitMerge(Point,number)
*/
#ifdef NEWSPLITMERGE

LONG SplitMerge(PHEAD WORD **Pointer, LONG number)
{
    GETBIDENTITY
    SORTING *S = AT.SS;
    WORD **pp3, **pp1, **pp2;
    LONG nleft, nright, i, newleft, newright;
    WORD **pptop;

    if ( number < 2 ) return(number);
    if ( number == 2 ) {
        pp1 = Pointer; pp2 = pp1 + 1;
        if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
            pp3 = (WORD **)(*pp1); *pp1 = *pp2; *pp2 = (WORD *)pp3;
        }
        else if ( i == 0 ) {
          number--;
          if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) == 0 ) { number = 0; } }
          else {               if ( AddCoef(BHEAD pp1,pp2) == 0 ) { number = 0; } }
        }
        return(number);
    }
    pptop = Pointer + number;
    nleft = number » 1; nright = number - nleft;
    newleft  = SplitMerge(BHEAD Pointer,nleft);
    newright = SplitMerge(BHEAD Pointer+nleft,nright);
/*
    We compare the last of the left with the first of the right
    If they are already in order, we will be done quickly.
    We may have to compactify the buffer because the recursion may
    have created holes. Also this compare may result in equal terms.
    Addition of 23-jul-1999. It makes things a bit faster.
*/
    if ( newleft > 0 && newright > 0 &&
    ( i = CompareTerms(BHEAD Pointer[newleft-1],Pointer[nleft],(WORD)0) ) >= 0 ) {
        pp2 = Pointer+nleft; pp1 = Pointer+newleft-1;
        if ( i == 0 ) {
          if ( S->PolyWise ) {
            if ( AddPoly(BHEAD pp1,pp2) > 0 ) pp1++;
            else newleft--;
          }
          else {
            if ( AddCoef(BHEAD pp1,pp2) > 0 ) pp1++;
            else newleft--;
          }
          *pp2++ = 0; newright--;
        }
        else pp1++;
        newleft += newright;
        if ( pp1 < pp2 ) {
            while ( --newright >= 0 ) *pp1++ = *pp2++;
            while ( pp1 < pptop ) *pp1++ = 0;
        }
        return(newleft);
    }
    if ( nleft > AN.SplitScratchSize ) {
        AN.SplitScratchSize = (nleft*3)/2+100;
        if ( AN.SplitScratchSize > S->Terms2InSmall/2 )
             AN.SplitScratchSize = S->Terms2InSmall/2;
        if ( AN.SplitScratch ) M_free(AN.SplitScratch,"AN.SplitScratch");
        AN.SplitScratch = (WORD **)Malloc1(AN.SplitScratchSize*sizeof(WORD *),"AN.SplitScratch");
    }
    pp3 = AN.SplitScratch; pp1 = Pointer; i = nleft;
    do { *pp3++ = *pp1; *pp1++ = 0; } while ( *pp1 && --i > 0 );
```

```
    if ( i > 0 ) { *pp3 = 0; i--; }
    AN.InScratch = nleft - i;
    pp1 = AN.SplitScratch; pp2 = Pointer + nleft; pp3 = Pointer;
    while ( nleft > 0 && nright > 0 && *pp1 && *pp2 ) {
        if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
            *pp3++ = *pp2;
            *pp2++ = 0;
            nright--;
        }
        else if ( i > 0 ) {
            *pp3++ = *pp1;
            *pp1++ = 0;
            nleft--;
        }
        else {
          if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
          else {               if ( AddCoef(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
          *pp1++ = 0; *pp2++ = 0; nleft--; nright--;
        }
    }
    while ( --nleft  >= 0 && *pp1 ) { *pp3++ = *pp1; *pp1++ = 0; }
    while ( --nright >= 0 && *pp2 ) { *pp3++ = *pp2++; }
    nleft = pp3 - Pointer;
    while ( pp3 < pptop ) *pp3++ = 0;
    AN.InScratch = 0;
    return(nleft);
}

#else

VOID SplitMerge(PHEAD WORD **Pointer, LONG number)
{
    GETBIDENTITY
    SORTING *S = AT.SS;
    WORD **pp3, **pp1, **pp2;
    LONG nleft, nright, i;
    WORD **pptop;

    if ( number < 2 ) return;
    if ( number == 2 ) {
        pp1 = Pointer; pp2 = pp1 + 1;
        if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
            pp3 = (WORD **)(*pp1); *pp1 = *pp2; *pp2 = (WORD *)pp3;
        }
        else if ( i == 0 ) {
          if ( S->PolyWise ) { if ( !AddPoly(BHEAD pp1,pp2) ) { *pp1 = 0; } }
          else {               if ( !AddCoef(BHEAD pp1,pp2) ) { *pp1 = 0; } }
          *pp2 = 0;
        }
        return;
    }
    pptop = Pointer + number;
    nleft = number » 1; nright = number - nleft;
    SplitMerge(BHEAD Pointer,nleft);
    SplitMerge(BHEAD Pointer+nleft,nright);
    if ( nleft > AN.SplitScratchSize ) {
        AN.SplitScratchSize = (nleft*3)/2+100;
        if ( AN.SplitScratchSize > S->Terms2InSmall/2 )
             AN.SplitScratchSize = S->Terms2InSmall/2;
        if ( AN.SplitScratch ) M_free(AN.SplitScratch,"AN.SplitScratch");
        AN.SplitScratch = (WORD **)Malloc1(AN.SplitScratchSize*sizeof(WORD *),"AN.SplitScratch");
    }
    pp3 = AN.SplitScratch; pp1 = Pointer; i = nleft;
    do { *pp3++ = *pp1; *pp1++ = 0; } while ( *pp1 && --i > 0 );
    if ( i > 0 ) { *pp3 = 0; i--; }
    AN.InScratch = nleft - i;
    pp1 = AN.SplitScratch; pp2 = Pointer + nleft; pp3 = Pointer;
    while ( *pp1 && *pp2 && nleft > 0 && nright > 0 ) {
        if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
            *pp3++ = *pp2;
            *pp2++ = 0;
            nright--;
        }
        else if ( i > 0 ) {
            *pp3++ = *pp1;
            *pp1++ = 0;
            nleft--;
        }
        else {
          if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
          else {               if ( AddCoef(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
          *pp1++ = 0; *pp2++ = 0; nleft--; nright--;
        }
    }
    while ( *pp1 && --nleft  >= 0 ) { *pp3++ = *pp1; *pp1++ = 0; }
    while ( *pp2 && --nright >= 0 ) { *pp3++ = *pp2++; }
    while ( pp3 < pptop ) *pp3++ = 0;
```

```
    AN.InScratch = 0;

    return;
}

#endif

/*
        #] SplitMerge :
        #[ GarbHand :               VOID GarbHand()
*/
VOID GarbHand()
{
    GETIDENTITY
    SORTING *S = AT.SS;
    WORD **Point, *s2, *t, *garbuf, i;
    LONG k, total = 0;
    int tobereturned = 0;
/*
    Compute the size needed. Put it in total.
*/
#ifdef TESTGARB
    MLOCK(ErrorMessageLock);
    MesPrint("in:  S->sFill = %x, S->sTop2 = %x",S->sFill,S->sTop2);
#endif
    Point = S->sPointer;
    k = S->sTerms;
    while ( --k >= 0 ) {
        if ( ( s2 = *Point++ ) != 0 ) { total += *s2; }
    }
    Point = AN.SplitScratch;
    k = AN.InScratch;
    while ( --k >= 0 ) {
        if ( ( s2 = *Point++ ) != 0 ) { total += *s2; }
    }
#ifdef TESTGARB
    MesPrint("total = %l, nterms = %l",2*total,AN.InScratch);
    MUNLOCK(ErrorMessageLock);
#endif
/*
    Test now whether it fits. If so deal with the problem inside
    the memory at the tail of the large buffer.
*/
    if ( S->lBuffer != 0 && S->lFill + total <= S->lTop ) {
        garbuf = S->lFill;
    }
    else {
        garbuf = (WORD *)Malloc1(total*sizeof(WORD),"Garbage buffer");
        tobereturned = 1;
    }
    t = garbuf;
    Point = S->sPointer;
    k = S->sTerms;
    while ( --k >= 0 ) {
        if ( *Point ) {
            s2 = *Point++;
            i = *s2;
            NCOPY(t,s2,i);
        }
        else { Point++; }
    }
    Point = AN.SplitScratch;
    k = AN.InScratch;
    while ( --k >= 0 ) {
        if ( *Point ) {
            s2 = *Point++;
            i = *s2;
            NCOPY(t,s2,i);
        }
        else Point++;
    }
    s2 = S->sBuffer;
    t = garbuf;
    Point = S->sPointer;
    k = S->sTerms;
    while ( --k >= 0 ) {
        if ( *Point ) {
            *Point++ = s2;
            i = *t;
            NCOPY(s2,t,i);
        }
        else { Point++; }
    }
    Point = AN.SplitScratch;
    k = AN.InScratch;
    while ( --k >= 0 ) {
        if ( *Point ) {
```

```
            *Point++ = s2;
            i = *t;
            NCOPY(s2,t,i);
        }
        else Point++;
    }
    S->sFill = s2;
#ifdef TESTGARB
    MLOCK(ErrorMessageLock);
    MesPrint("out: S->sFill = %x, S->sTop2 = %x",S->sFill,S->sTop2);
    if ( S->sFill >= S->sTop2 ) {
        MesPrint("We are in deep trouble");
    }
    MUNLOCK(ErrorMessageLock);
#endif
    if ( tobereturned ) M_free(garbuf,"Garbage buffer");
    return;
}

/*
        #] GarbHand :
        #[ MergePatches :           WORD MergePatches(par)
*/
WORD MergePatches(WORD par)
{
    GETIDENTITY
    SORTING *S = AT.SS;
    WORD **poin, **poin2, ul, k, i, im, *m1;
    WORD *p, lpat, mpat, level, l1, l2, r1, r2, r3, c;
    WORD *m2, *m3, r31, r33, ki, *rr;
    UWORD *coef;
    POSITION position;
    FILEHANDLE *fin, *fout;
    int fhandle;
/*
    UBYTE *s;
*/
#ifdef WITHZLIB
    POSITION position2;
    int oldgzipCompress = AR.gzipCompress;
    if ( par == 2 ) {
        AR.gzipCompress = 0;
    }
#endif
    fin = &S->file;
    fout = &(AR.FoStage4[0]);
NewMerge:
    coef = AN.SoScratC;
    poin = S->poina; poin2 = S->poin2a;
    rr = AR.CompressPointer;
    *rr = 0;
/*
        #[ Setup :
*/
    if ( par == 1 ) {
        fout = &(S->file);
        if ( fout->handle < 0 ) {
FileMake:
            PUTZERO(AN.OldPosOut);
            if ( ( fhandle = CreateFile(fout->name) ) < 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Cannot create file %s",fout->name);
                MUNLOCK(ErrorMessageLock);
                goto ReturnError;
            }
#ifdef GZIPDEBUG
            MLOCK(ErrorMessageLock);
            MesPrint("%w MergePatches created output file %s",fout->name);
            MUNLOCK(ErrorMessageLock);
#endif
            fout->handle = fhandle;
            PUTZERO(fout->filesize);
            PUTZERO(fout->POposition);
#ifdef WITHZLIB
            fout->ziobuffer = 0;
#endif
#ifdef ALLLOCK
            LOCK(fout->pthreadslock);
#endif
            SeekFile(fout->handle,&(fout->filesize),SEEK_SET);
#ifdef ALLLOCK
            UNLOCK(fout->pthreadslock);
#endif
            S->fPatchN = 0;
            PUTZERO(S->fPatches[0]);
            fout->POfill = fout->PObuffer;
            PUTZERO(fout->POposition);
```

```
        }
ConMer:
        StageSort(fout);
#ifdef WITHZLIB
        if ( S == AT.S0 && AR.NoCompress == 0 && AR.gzipCompress > 0 )
            S->fpcompressed[S->fPatchN] = 1;
        else
            S->fpcompressed[S->fPatchN] = 0;
        SetupOutputGZIP(fout);
#endif
    }
    else if ( par == 0 && S->stage4 > 0 ) {
/*
        We will have to do our job more than once.
        Input is from S->file and output will go to AR.FoStage4.
        The file corresponding to this last one must be made now.
*/
        AR.Stage4Name ^= 1;
/*
        s = (UBYTE *)(fout->name); while ( *s ) s++;
        if ( AR.Stage4Name ) s[-1] += 1;
        else                 s[-1] -= 1;
*/
        S->iPatches = S->fPatches;
        S->fPatches = S->inPatches;
        S->inPatches = S->iPatches;
        (S->inNum) = S->fPatchN;
        AN.OldPosIn = AN.OldPosOut;
#ifdef WITHZLIB
        m1 = S->fpincompressed;
        S->fpincompressed = S->fpcompressed;
        S->fpcompressed = m1;
        for ( i = 0; i < S->inNum; i++ ) {
            S->fPatchesStop[i] = S->iPatches[i+1];
#ifdef GZIPDEBUG
            MLOCK(ErrorMessageLock);
            MesPrint("%w fPatchesStop[%d] = %10p",i,&(S->fPatchesStop[i]));
            MUNLOCK(ErrorMessageLock);
#endif
        }
#endif
        S->stage4 = 0;
        goto FileMake;
    }
    else {
#ifdef WITHZLIB
/*
        The next statement is just for now
*/
        AR.gzipCompress = 0;
#endif
        if ( par == 0 ) {
            S->iPatches = S->fPatches;
            S->inNum = S->fPatchN;
#ifdef WITHZLIB
            m1 = S->fpincompressed;
            S->fpincompressed = S->fpcompressed;
            S->fpcompressed = m1;
            for ( i = 0; i < S->inNum; i++ ) {
                S->fPatchesStop[i] = S->fPatches[i+1];
#ifdef GZIPDEBUG
                MLOCK(ErrorMessageLock);
                MesPrint("%w fPatchesStop[%d] = %10p",i,&(S->fPatchesStop[i]));
                MUNLOCK(ErrorMessageLock);
#endif
            }
#endif
        }
        fout = AR.outfile;
    }
    if ( par ) {                /* Mark end of patches */
        S->Patches[S->lPatch] = S->lFill;
        for ( i = 0; i < S->lPatch; i++ ) {
            S->pStop[i] = S->Patches[i+1]-1;
            S->Patches[i] = (WORD *)(((UBYTE *)(S->Patches[i])) + AM.MaxTer);
        }
    }
    else {  /* Load the patches */
        S->lPatch = (S->inNum);
#ifdef WITHMPI
        if ( S->lPatch > 1 || ( (PF.exprtodo <0) && (fout == AR.outfile || fout == AR.hidefile ) ) ) {
#else
        if ( S->lPatch > 1 ) {
#endif
#ifdef WITHZLIB
            SetupAllInputGZIP(S);
#endif
```

```
            p = S->lBuffer;
            for ( i = 0; i < S->lPatch; i++ ) {
                p = (WORD *)(((UBYTE *)p)+2*AM.MaxTer+COMPINC*sizeof(WORD));
                S->Patches[i] = p;
                p = (WORD *)(((UBYTE *)p) + fin->POsize);
                S->pStop[i] = m2 = p;
#ifdef WITHZLIB
                PutIn(fin,&(S->iPatches[i]),S->Patches[i],&m2,i);
#else
                ADDPOS(S->iPatches[i],PutIn(fin,&(S->iPatches[i]),S->Patches[i],&m2,i));
#endif
            }
        }
    }
    if ( fout->handle >= 0 ) {
        PUTZERO(position);
#ifdef ALLLOCK
        LOCK(fout->pthreadslock);
#endif
        SeekFile(fout->handle,&position,SEEK_END);
        ADDPOS(position,((fout->POfill-fout->PObuffer)*sizeof(WORD)));
#ifdef ALLLOCK
        UNLOCK(fout->pthreadslock);
#endif
    }
    else {
        SETBASEPOSITION(position,(fout->POfill-fout->PObuffer)*sizeof(WORD));
    }
/*
        #] Setup :

    The old code had to be replaced because all output needs to go
    through PutOut. For this we have to go term by term and keep
    track of the compression.
*/
    if ( S->lPatch == 1 ) { /* Single patch --> direct copy. Very rare. */
        LONG length;

        if ( fout->handle < 0 ) if ( Sflush(fout) ) goto PatCall;
        if ( par ) {        /* Memory to file */
#ifdef WITHZLIB
/*
            We fix here the problem that the thing needs to go through PutOut
*/
            m2 = m1 = *S->Patches; /* The m2 is to keep the compiler from complaining */
            while ( *m1 ) {
                if ( *m1 < 0 ) { /* Need to uncompress */
                    i = -(*m1++); m2 += i; im = *m1+i+1;
                    while ( i > 0 ) { *m1-- = *m2--; i--; }
                    *m1 = im;
                }
#ifdef WITHPTHREADS
                if ( AS.MasterSort && ( fout == AR.outfile ) ) { im = PutToMaster(BHEAD m1); }
                else
#endif
                if ( ( im = PutOut(BHEAD m1,&position,fout,1) ) < 0 ) goto ReturnError;
                ADDPOS(S->SizeInFile[par],im);
                m2 = m1;
                m1 += *m1;
            }
#ifdef WITHPTHREADS
            if ( AS.MasterSort && ( fout == AR.outfile ) ) { PutToMaster(BHEAD 0); }
            else
#endif
            if ( FlushOut(&position,fout,1) ) goto ReturnError;
            ADDPOS(S->SizeInFile[par],1);
#else
/* old code */
            length = (LONG)(*S->pStop)-(LONG)(*S->Patches)+sizeof(WORD);
            if ( WriteFile(fout->handle,(UBYTE *)(*S->Patches),length) != length )
                goto PatwCall;
            ADDPOS(position,length);
            ADDPOS(fout->POposition,length);
            ADDPOS(fout->filesize,length);
            ADDPOS(S->SizeInFile[par],length/sizeof(WORD));
#endif
        }
        else {              /* File to file */
#ifdef WITHZLIB
/*
            Note: if we change FRONTSIZE we need to make the minimum value
            of SmallEsize in AllocSort correspondingly larger or smaller.
            Theoretically we could get close to 2*AM.MaxTer!
*/
            #define FRONTSIZE (2*AM.MaxTer)
            WORD *copybuf = (WORD *)(((UBYTE *)(S->sBuffer)) + FRONTSIZE);
            WORD *copytop;
```

```
            SetupOutputGZIP(fout);
            SetupAllInputGZIP(S);
            m1 = m2 = copybuf;
            position2 = S->iPatches[0];
            while ( ( length = FillInputGZIP(fin,&position2,
                    (UBYTE *)copybuf,
                    (S->SmallEsize*sizeof(WORD)-FRONTSIZE),0) ) > 0 ) {
                copytop = (WORD *)(((UBYTE *)copybuf)+length);
                while ( *m1 && ( ( *m1 > 0 && m1++*m1 < copytop ) ||
                ( *m1 < 0 && ( m1+1 < copytop ) && ( m1+m1[1]+1 < copytop ) ) ) )
/*
    22-jun-2013 JV  Extremely nasty bug that has been around for a while.
                    What if the end is in the remaining part? We will loose terms!
                while ( *m1 && ( (WORD *)(((UBYTE *)(m1)) + AM.MaxTer ) < S->sTop2 ) )
*/
                {
                    if ( *m1 < 0 ) { /* Need to uncompress */
                        i = -(*m1++); m2 += i; im = *m1+i+1;
                        while ( i > 0 ) { *m1-- = *m2--; i--; }
                        *m1 = im;
                    }
#ifdef WITHPTHREADS
                    if ( AS.MasterSort && ( fout == AR.outfile ) ) {
                        im = PutToMaster(BHEAD m1);
                    }
                    else
#endif
                    if ( ( im = PutOut(BHEAD m1,&position,fout,1) ) < 0 ) goto ReturnError;
                    ADDPOS(S->SizeInFile[par],im);
                    m2 = m1;
                    m1 += *m1;
                }
                if ( m1 < copytop && *m1 == 0 ) break;
/*
                Now move the remaining part 'back'
*/
                m3 = copybuf;
                m1 = copytop;
                while ( m1 > m2 ) *--m3 = *--m1;
                m2 = m3;
                m1 = m2 + *m2;
            }
            if ( length < 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Readerror");
                goto PatCall2;
            }
#ifdef WITHPTHREADS
            if ( AS.MasterSort && ( fout == AR.outfile ) ) { PutToMaster(BHEAD 0); }
            else
#endif
            if ( FlushOut(&position,fout,1) ) goto ReturnError;
            ADDPOS(S->SizeInFile[par],1);
#else
/* old code */
            SeekFile(fin->handle,&(S->iPatches[0]),SEEK_SET); /* needed for stage4 */
            while ( ( length = ReadFile(fin->handle,
                    (UBYTE *)(S->sBuffer),S->SmallEsize*sizeof(WORD)) ) > 0 ) {
                if ( WriteFile(fout->handle,(UBYTE *)(S->sBuffer),length) != length )
                    goto PatwCall;
                ADDPOS(position,length);
                ADDPOS(fout->POposition,length);
                ADDPOS(fout->filesize,length);
                ADDPOS(S->SizeInFile[par],length/sizeof(WORD));
            }
            if ( length < 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Readerror");
                goto PatCall2;
            }
#endif
        }
        goto EndOfAll;
    }
    else if ( S->lPatch > 0 ) {

        /* More than one patch. Construct the tree. */

        lpat = 1;
        do { lpat <<= 1; } while ( lpat < S->lPatch );
        mpat = ( lpat >> 1 ) - 1;
        k = lpat - S->lPatch;

        /* k is the number of empty places in the tree. they will
           be at the even positions from 2 to 2*k */

        for ( i = 1; i < lpat; i++ ) {
```

```
            S->tree[i] = -1;
        }
        for ( i = 1; i <= k; i++ ) {
            im = ( i « 1 ) - 1;
            poin[im] = S->Patches[i-1];
            poin2[im] = poin[im] + *(poin[im]);
            S->used[i] = im;
            S->ktoi[im] = i-1;
            S->tree[mpat+i] = 0;
            poin[im-1] = poin2[im-1] = 0;
        }
        for ( i = (k«1)+1; i <= lpat; i++ ) {
            S->used[i-k] = i;
            S->ktoi[i] = i-k-1;
            poin[i] = S->Patches[i-k-1];
            poin2[i] = poin[i] + *(poin[i]);
        }
/*
        the array poin tells the position of the i-th element of the S->tree
        'S->used' is a stack with the S->tree elements that need to be entered
        into the S->tree. at the beginning this is S->lPatch. during the
        sort there will be only very few elements.
        poin2 is the next value of poin. it has to be determined
        before the comparisons as the position or the size of the
        term indicated by poin may change.
        S->ktoi translates a S->tree element back to its stream number.

        start the sort
*/
        level = S->lPatch;

        /* introduce one term */
OneTerm:
        k = S->used[level];
        i = k + lpat - 1;
        if ( !*(poin[k]) ) {
            do { if ( !( i »= 1 ) ) goto EndOfMerge; } while ( !S->tree[i] );
            if ( S->tree[i] == -1 ) {
                S->tree[i] = 0;
                level--;
                goto OneTerm;
            }
            k = S->tree[i];
            S->used[level] = k;
            S->tree[i] = 0;
        }
/*
        move terms down the tree
*/
        while ( i »= 1 ) {
            if ( S->tree[i] > 0 ) {
                if ( ( c = CompareTerms(BHEAD poin[S->tree[i]],poin[k],(WORD)0) ) > 0 ) {
/*
                    S->tree[i] is the smaller. Exchange and go on.
*/
                    S->used[level] = S->tree[i];
                    S->tree[i] = k;
                    k = S->used[level];
                }
                else if ( !c ) {    /* Terms are equal */
                    S->TermsLeft--;
/*
                        Here the terms are equal and their coefficients
                        have to be added.
*/
                    l1 = *( m1 = poin[S->tree[i]] );
                    l2 = *( m2 = poin[k] );
                    if ( S->PolyWise ) {  /* Here we work with PolyFun */
                        WORD *tt1, *w;
                        tt1 = m1;
                        m1 += S->PolyWise;
                        m2 += S->PolyWise;
                        if ( S->PolyFlag == 2 ) {
                            w = poly_ratfun_add(BHEAD m1,m2);
                            if ( *tt1 + w[1] - m1[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
                                MLOCK(ErrorMessageLock);
                                MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l
    is too small",AM.MaxTer);
                                MUNLOCK(ErrorMessageLock);
                                Terminate(-1);
                            }
                            AT.WorkPointer = w;
                        }
                        else {
                            w = AT.WorkPointer;
                            if ( w + m1[1] + m2[1] > AT.WorkTop ) {
                                MLOCK(ErrorMessageLock);
```

```
                                MesPrint("A WorkSpace of %10l is too small",AM.WorkSize);
                                MUNLOCK(ErrorMessageLock);
                                Terminate(-1);
                            }
                            AddArgs(BHEAD m1,m2,w);
                        }
                        r1 = w[1];
                        if ( r1 <= FUNHEAD
                            || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) )
                                { goto cancelled; }
                        if ( r1 == m1[1] ) {
                            NCOPY(m1,w,r1);
                        }
                        else if ( r1 < m1[1] ) {
                            r2 = m1[1] - r1;
                            m2 = w + r1;
                            m1 += m1[1];
                            while ( --r1 >= 0 ) *--m1 = *--m2;
                            m2 = m1 - r2;
                            r1 = S->PolyWise;
                            while ( --r1 >= 0 ) *--m1 = *--m2;
                            *m1 -= r2;
                            poin[S->tree[i]] = m1;
                        }
                        else {
                            r2 = r1 - m1[1];
                            m2 = tt1 - r2;
                            r1 = S->PolyWise;
                            m1 = tt1;
                            *m1 += r2;
                            poin[S->tree[i]] = m2;
                            NCOPY(m2,m1,r1);
                            r1 = w[1];
                            NCOPY(m2,w,r1);
                        }
                    }
                    else {
                        r1 = *( m1 += l1 - 1 );
                        m1 -= ABS(r1) - 1;
                        r1 = ( ( r1 > 0 ) ? (r1-1) : (r1+1) ) » 1;
                        r2 = *( m2 += l2 - 1 );
                        m2 -= ABS(r2) - 1;
                        r2 = ( ( r2 > 0 ) ? (r2-1) : (r2+1) ) » 1;

                        if ( AddRat(BHEAD (UWORD *)m1,r1,(UWORD *)m2,r2,coef,&r3) ) {
                            MLOCK(ErrorMessageLock);
                            MesCall("MergePatches");
                            MUNLOCK(ErrorMessageLock);
                            SETERROR(-1)
                        }

                        if ( AN.ncmod != 0 ) {
                            if ( ( AC.modmode & POSNEG ) != 0 ) {
                                NormalModulus(coef,&r3);
                            }
                            else if ( BigLong(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
                                WORD ii;
                                SubPLon(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod),coef,&r3);
                                coef[r3] = 1;
                                for ( ii = 1; ii < r3; ii++ ) coef[r3+ii] = 0;
                            }
                        }
                        r3 «= 1;
                        r33 = ( r3 > 0 ) ? ( r3 + 1 ) : ( r3 - 1 );
                        if ( r3 < 0 ) r3 = -r3;
                        if ( r1 < 0 ) r1 = -r1;
                        r1 «= 1;
                        r31 = r3 - r1;
                        if ( !r3 ) {      /* Terms cancel */
cancelled:
                            ul = S->used[level] = S->tree[i];
                            S->tree[i] = -1;
/*
                            We skip to the next term in stream ul
*/
                            im = *poin2[ul];
                            if ( im < 0 ) {
                                r1 = poin2[ul][1] - im + 1;
                                m1 = poin2[ul] + 2;
                                m2 = poin[ul] - im + 1;
                                while ( ++im <= 0 ) *--m1 = *--m2;
                                *--m1 = r1;
                                poin2[ul] = m1;
                                im = r1;
                            }
                            poin[ul] = poin2[ul];
                            ki = S->ktoi[ul];
```

```
                        if ( !par && (poin[ul] + im + COMPINC) >= S->pStop[ki]
                        && im > 0 ) {
#ifdef WITHZLIB
                            PutIn(fin,&(S->iPatches[ki]),S->Patches[ki],&(poin[ul]),ki);
#else
                            ADDPOS(S->iPatches[ki],PutIn(fin,&(S->iPatches[ki]),
                            S->Patches[ki],&(poin[ul]),ki));
#endif
                            poin2[ul] = poin[ul] + im;
                        }
                        else {
                            poin2[ul] += im;
                        }
                        S->used[++level] = k;
                        S->TermsLeft--;
                      }
                      else if ( !r31 ) {        /* copy coef into term1 */
                        goto CopCof2;
                      }
                      else if ( r31 < 0 ) {     /* copy coef into term1
                                                   and adjust the length of term1 */
                        goto CopCoef;
                      }
                      else {
/*
                            this is the dreaded calamity.
                            is there enough space?
*/
                        if( (poin[S->tree[i]]+l1+r31) >= poin2[S->tree[i]] ) {
/*
                                no space! now the special trick for which
                                we left 2*maxlng spaces open at the beginning
                                of each patch.
*/
                            if ( (l1 + r31) > AM.MaxTer/((LONG)sizeof(WORD)) ) {
                                MLOCK(ErrorMessageLock);
                                MesPrint("Coefficient overflow during sort");
                                MUNLOCK(ErrorMessageLock);
                                goto ReturnError;
                            }
                            m2 = poin[S->tree[i]];
                            m3 = ( poin[S->tree[i]] -= r31 );
                            do { *m3++ = *m2++; } while ( m2 < m1 );
                            m1 = m3;
                        }
CopCoef:
                        *(poin[S->tree[i]]) += r31;
CopCof2:
                        m2 = (WORD *)coef; im = r3;
                        NCOPY(m1,m2,im);
                        *m1 = r33;
                      }
                    }
/*
                    Now skip to the next term in stream k.
*/
NextTerm:
                    im = poin2[k][0];
                    if ( im < 0 ) {
                        r1 = poin2[k][1] - im + 1;
                        m1 = poin2[k] + 2;
                        m2 = poin[k] - im + 1;
                        while ( ++im <= 0 ) *--m1 = *--m2;
                        *--m1 = r1;
                        poin2[k] = m1;
                        im = r1;
                    }
                    poin[k] = poin2[k];
                    ki = S->ktoi[k];
                    if ( !par && ( (poin[k] + im + COMPINC) >= S->pStop[ki] )
                    && im > 0 ) {
#ifdef WITHZLIB
                        PutIn(fin,&(S->iPatches[ki]),S->Patches[ki],&(poin[k]),ki);
#else
                        ADDPOS(S->iPatches[ki],PutIn(fin,&(S->iPatches[ki]),
                        S->Patches[ki],&(poin[k]),ki));
#endif
                        poin2[k] = poin[k] + im;
                    }
                    else {
                        poin2[k] += im;
                    }
                    goto OneTerm;
                }
            }
            else if ( S->tree[i] < 0 ) {
                S->tree[i] = k;
```

```
                level--;
                goto OneTerm;
            }
        }
/*
            found the smallest in the set. indicated by k.
            write to its destination.
*/
#ifdef WITHPTHREADS
        if ( AS.MasterSort && ( fout == AR.outfile ) ) { im = PutToMaster(BHEAD poin[k]); }
        else
#endif
        if ( ( im = PutOut(BHEAD poin[k],&position,fout,1) ) < 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Called from MergePatches with k = %d (stream %d)",k,S->ktoi[k]);
            MUNLOCK(ErrorMessageLock);
            goto ReturnError;
        }
        ADDPOS(S->SizeInFile[par],im);
        goto NextTerm;
    }
    else {
        goto NormalReturn;
    }
EndOfMerge:
#ifdef WITHPTHREADS
        if ( AS.MasterSort && ( fout == AR.outfile ) ) { PutToMaster(BHEAD 0); }
        else
#endif
    if ( FlushOut(&position,fout,1) ) goto ReturnError;
    ADDPOS(S->SizeInFile[par],1);
EndOfAll:
    if ( par == 1 ) {   /* Set the fpatch pointers */
#ifdef WITHZLIB
        SeekFile(fout->handle,&position,SEEK_CUR);
#endif
        (S->fPatchN)++;
        S->fPatches[S->fPatchN] = position;
    }
    if ( par == 0 && fout != AR.outfile ) {
/*
            Output went to sortfile. We have two possibilities:
            1:  We are not finished with the current in-out cycle
                In that case we should pop to the next set of patches
            2:  We finished a cycle and should clean up the in file
                Then we restart the sort.
*/
        (S->fPatchN)++;
        S->fPatches[S->fPatchN] = position;
        if ( ISNOTZEROPOS(AN.OldPosIn) ) {      /* We are not done */

            SeekFile(fin->handle,&(AN.OldPosIn),SEEK_SET);
/*
            We don't need extra provisions for the zlib compression here.
            If part of an expression has been sorted, the whole has been so.
            This means that S->fpincompressed[] will remain the same
*/
            if ( (ULONG)ReadFile(fin->handle,(UBYTE *)(&(S->inNum)),(LONG)sizeof(WORD)) !=
                sizeof(WORD)
              || (ULONG)ReadFile(fin->handle,(UBYTE *)(&AN.OldPosIn),(LONG)sizeof(POSITION)) !=
                sizeof(POSITION)
              || (ULONG)ReadFile(fin->handle,(UBYTE *)S->iPatches,(LONG)((S->inNum)+1)
                    *sizeof(POSITION)) != ((S->inNum)+1)*sizeof(POSITION) ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Read error fourth stage sorting");
                MUNLOCK(ErrorMessageLock);
                goto ReturnError;
            }
            *rr = 0;
#ifdef WITHZLIB
            for ( i = 0; i < S->inNum; i++ ) {
                S->fPatchesStop[i] = S->iPatches[i+1];
#ifdef GZIPDEBUG
                MLOCK(ErrorMessageLock);
                MesPrint("%w fPatchesStop[%d] = %10p",i,&(S->fPatchesStop[i]));
                MUNLOCK(ErrorMessageLock);
#endif
            }
#endif
            goto ConMer;
        }
        else {
/*
            if ( fin == &(AR.FoStage4[0]) ) {
                s = (UBYTE *)(fin->name); while ( *s ) s++;
                if ( AR.Stage4Name == 1 ) s[-1] -= 1;
                else                      s[-1] += 1;
```

```
            }
*/
/*          TruncateFile(fin->handle); */
            UpdateMaxSize();
            CloseFile(fin->handle);
            remove(fin->name);      /* Gives diskspace free again. */
#ifdef GZIPDEBUG
            MLOCK(ErrorMessageLock);
            MesPrint("%w MergePatches removed in file %s",fin->name);
            MUNLOCK(ErrorMessageLock);
#endif
/*
            if ( fin == &(AR.FoStage4[0]) ) {
                s = (UBYTE *)(fin->name); while ( *s ) s++;
                if ( AR.Stage4Name == 1 ) s[-1] += 1;
                else                      s[-1] -= 1;
            }
*/
            fin->handle = -1;
            { FILEHANDLE *ff = fin; fin = fout; fout = ff; }
            PUTZERO(S->SizeInFile[0]);
            goto NewMerge;
        }
    }
    if ( par == 0 ) {
/*      TruncateFile(fin->handle); */
        UpdateMaxSize();
        CloseFile(fin->handle);
        remove(fin->name);
        fin->handle = -1;
#ifdef GZIPDEBUG
        MLOCK(ErrorMessageLock);
        MesPrint("%w MergePatches removed in file %s",fin->name);
        MUNLOCK(ErrorMessageLock);
#endif
    }
NormalReturn:
#ifdef WITHZLIB
    AR.gzipCompress = oldgzipCompress;
#endif
    return(0);
ReturnError:
#ifdef WITHZLIB
    AR.gzipCompress = oldgzipCompress;
#endif
    return(-1);
#ifndef WITHZLIB
PatwCall:
    MLOCK(ErrorMessageLock);
    MesPrint("Error while writing to file.");
    goto PatCall2;
#endif
PatCall:;
    MLOCK(ErrorMessageLock);
PatCall2:;
    MesCall("MergePatches");
    MUNLOCK(ErrorMessageLock);
#ifdef WITHZLIB
    AR.gzipCompress = oldgzipCompress;
#endif
    SETERROR(-1)
}

/*
        #] MergePatches :
        #[ StoreTerm :              WORD StoreTerm(term)
*/
WORD StoreTerm(PHEAD WORD *term)
{
    GETBIDENTITY
    SORTING *S = AT.SS;
    WORD **ss, *lfill, j, *t;
    POSITION pp;
    LONG lSpace, sSpace, RetCode, over, tover;

    if ( ( ( AP.PreDebug & DUMPTOSORT ) == DUMPTOSORT ) && AR.sLevel == 0 ) {
#ifdef WITHPTHREADS
        sprintf((char *)(THRbuf),"StoreTerm(%d)",AT.identity);
        PrintTerm(term,(char *)(THRbuf));
#else
        PrintTerm(term,"StoreTerm");
#endif
    }
    if ( AM.exitflag && AR.sLevel == 0 ) return(0);
    S->sFill = *(S->PoinFill);
    if ( S->sTerms >= S->TermsInSmall || ( S->sFill + *term ) >= S->sTop ) {
/*
```

```
    The small buffer is full. It has to be sorted and written.
*/
        tover = over = S->sTerms;
        ss = S->sPointer;
        ss[over] = 0;
/*
        PrintTime();
*/
        SplitMerge(BHEAD ss,over);
        sSpace = 0;
        if ( over > 0 ) {
            ss[over] = 0;
            sSpace = ComPress(ss,&RetCode);
            S->TermsLeft -= over - RetCode;
        }
        sSpace++;

        lSpace = sSpace + (S->lFill - S->lBuffer)
                 - (AM.MaxTer/sizeof(WORD))*((LONG)S->lPatch);
        SETBASEPOSITION(pp,lSpace);
        MULPOS(pp,sizeof(WORD));
        if ( S->file.handle >= 0 ) {
            ADD2POS(pp,S->fPatches[S->fPatchN]);
        }
        if ( S == AT.S0 ) { /* Only statistics at ground level */
            WORD oldLogHandle = AC.LogHandle;
            if ( AC.LogHandle >= 0 && AM.LogType ) AC.LogHandle = -1;
            WriteStats(&pp,(WORD)0);
            AC.LogHandle = oldLogHandle;
        }
        if ( ( S->lPatch >= S->MaxPatches ) ||
            ( ( (WORD *)(((UBYTE *)(S->lFill + sSpace)) + 2*AM.MaxTer ) ) >= S->lTop ) ) {
/*
            The large buffer is too full. Merge and write it
*/
            if ( MergePatches(1) ) goto StoreCall;
/*
            pp = S->SizeInFile[1];
            ADDPOS(pp,sSpace);
            MULPOS(pp,sizeof(WORD));
*/
            SETBASEPOSITION(pp,sSpace);
            MULPOS(pp,sizeof(WORD));
            ADD2POS(pp,S->fPatches[S->fPatchN]);

            if ( S == AT.S0 ) { /* Only statistics at ground level */
                WORD oldLogHandle = AC.LogHandle;
                if ( AC.LogHandle >= 0 && AM.LogType ) AC.LogHandle = -1;
                WriteStats(&pp,(WORD)1);
                AC.LogHandle = oldLogHandle;
            }
            S->lPatch = 0;
            S->lFill = S->lBuffer;
        }
        S->Patches[S->lPatch++] = S->lFill;
        lfill = (WORD *)(((UBYTE *)(S->lFill)) + AM.MaxTer);
        if ( tover > 0 ) {
            ss = S->sPointer;
            while ( ( t = *ss++ ) != 0 ) {
                j = *t;
                if ( j < 0 ) j = t[1] + 2;
                while ( --j >= 0 ){
                  *lfill++ = *t++;
                }
            }
        }
        *lfill++ = 0;
        S->lFill = lfill;
        S->sTerms = 0;
        S->PoinFill = S->sPointer;
        *(S->PoinFill) = S->sFill = S->sBuffer;
    }
    j = *term;
    while ( --j >= 0 ) *S->sFill++ = *term++;
    S->sTerms++;
    S->GenTerms++;
    S->TermsLeft++;
    *++S->PoinFill = S->sFill;

    return(0);

StoreCall:
    MLOCK(ErrorMessageLock);
    MesCall("StoreTerm");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}
```

```
/*
        #] StoreTerm :
        #[ StageSort :              VOID StageSort(FILEHANDLE *fout)
*/
VOID StageSort(FILEHANDLE *fout)
{
    GETIDENTITY
    SORTING *S = AT.SS;
    if ( S->fPatchN >= S->MaxFpatches ) {
        POSITION position;
        PUTZERO(position);
        MLOCK(ErrorMessageLock);
#ifdef WITHPTHREADS
        MesPrint("StageSort in thread %d",identity);
#elif defined(WITHMPI)
        MesPrint("StageSort in process %d",PF.me);
#else
        MesPrint("StageSort");
#endif
        MUNLOCK(ErrorMessageLock);
        SeekFile(fout->handle,&position,SEEK_END);
/*
        No extra compression data has to be written.
        S->fpincompressed should remain valid.
*/
        if ( (ULONG)WriteFile(fout->handle,(UBYTE *)(&(S->fPatchN)),(LONG)sizeof(WORD)) !=
            sizeof(WORD)
          || (ULONG)WriteFile(fout->handle,(UBYTE *)(&(AN.OldPosOut)),(LONG)sizeof(POSITION)) !=
            sizeof(POSITION)
          || (ULONG)WriteFile(fout->handle,(UBYTE *)(S->fPatches),(LONG)(S->fPatchN+1)
                    *sizeof(POSITION)) != (S->fPatchN+1)*sizeof(POSITION) ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Write error while staging sort. Disk full?");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        AN.OldPosOut = position;
        fout->filesize = position;
        ADDPOS(fout->filesize,(S->fPatchN+2)*sizeof(POSITION) + sizeof(WORD));
        fout->POposition = fout->filesize;
        S->fPatches[0] = fout->filesize;
        S->fPatchN = 0;

        if ( AR.FoStage4[0].PObuffer == 0 ) {
            AR.FoStage4[0].PObuffer = (WORD *)Malloc1(AR.FoStage4[0].POsize*sizeof(WORD)
                                        ,"Stage 4 buffer");
            AR.FoStage4[0].POfill   = AR.FoStage4[0].PObuffer;
            AR.FoStage4[0].POstop   = AR.FoStage4[0].PObuffer
                        + AR.FoStage4[0].POsize/sizeof(WORD);
#ifdef WITHPTHREADS
            AR.FoStage4[0].pthreadslock = dummylock;
#endif
        }
        if ( AR.FoStage4[1].PObuffer == 0 ) {
            AR.FoStage4[1].PObuffer = (WORD *)Malloc1(AR.FoStage4[1].POsize*sizeof(WORD)
                                        ,"Stage 4 buffer");
            AR.FoStage4[1].POfill   = AR.FoStage4[1].PObuffer;
            AR.FoStage4[1].POstop   = AR.FoStage4[1].PObuffer
                        + AR.FoStage4[1].POsize/sizeof(WORD);
#ifdef WITHPTHREADS
            AR.FoStage4[1].pthreadslock = dummylock;
#endif
        }
        S->stage4 = 1;
    }
}

/*
        #] StageSort :
        #[ SortWild :               WORD SortWild(w,nw)
*/
WORD SortWild(WORD *w, WORD nw)
{
    GETIDENTITY
    WORD *v, *s, *m, k, i;
    WORD *pScrat, *stop, *sv, error = 0;
    pScrat = AT.WorkPointer;
    if ( ( AT.WorkPointer + 8 * AM.MaxWildcards ) >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    stop = w + nw;
    i = 0;
    while ( i < nw ) {
```

```
        m = w + i;
        v = m + m[1];
        while ( v < stop && (
             *v == FROMSET || *v == SETTONUM || *v == LOADDOLLAR ) ) v += v[1];
        while ( v < stop ) {
            if ( *v >= 0 ) {
                if ( AM.Ordering[*v] < AM.Ordering[*m] ) {
                    m = v;
                }
                else if ( *v == *m ) {
                    if ( v[2] < m[2] ) {
                        m = v;
                    }
                    else if ( v[2] == m[2] ) {
                        s = m + m[1];
                        sv = v + v[1];
                        if ( s < stop && ( *s == FROMSET
                         || *s == SETTONUM || *s == LOADDOLLAR ) ) {
                            if ( sv < stop && ( *sv == FROMSET
                            || *sv == SETTONUM || *sv == LOADDOLLAR ) ) {
                                if ( s[2] != sv[2] ) {
                                    error = -1;
                                    MLOCK(ErrorMessageLock);
                                    MesPrint("&Wildcard set conflict");
                                    MUNLOCK(ErrorMessageLock);
                                }
                            }
                            *v = -1;
                        }
                        else {
                            if ( sv < stop && ( *sv == FROMSET
                            || *sv == SETTONUM || *sv == LOADDOLLAR ) ) {
                                *m = -1;
                                m = v;
                            }
                            else {
                                *v = -1;
                            }
                        }
                    }
                }
            }
            v += v[1];
            while ( v < stop && ( *v == FROMSET
             || *v == SETTONUM || *v == LOADDOLLAR ) ) v += v[1];
        }
        s = pScrat;
        v = m;
        k = m[1];
        NCOPY(s,m,k);
        while ( m < stop && ( *m == FROMSET
         || *m == SETTONUM || *m == LOADDOLLAR ) ) {
            k = m[1];
            NCOPY(s,m,k);
        }
        *v = -1;
        pScrat = s;
        i = 0;
        while ( i < nw && ( w[i] < 0 || w[i] == FROMSET
        || w[i] == SETTONUM || w[i] == LOADDOLLAR ) ) i += w[i+1];
    }
    AC.NwildC = k = WORDDIF(pScrat,AT.WorkPointer);
    s = AT.WorkPointer;
    m = w;
    NCOPY(m,s,k);
    AC.WildC = m;
    return(error);
}

/*
        #] SortWild :
        #[ CleanUpSort :            VOID CleanUpSort(num)
*/
void CleanUpSort(int num)
{
    GETIDENTITY
    SORTING *S;
    int minnum = num, i;
    if ( AN.FunSorts ) {
        if ( num == -1 ) {
            if ( AN.MaxFunSorts > 3 ) {
                minnum = (AN.MaxFunSorts+4)/2;
            }
            else minnum = 4;
        }
        else if ( minnum == 0 ) minnum = 1;
        for ( i = minnum; i < AN.NumFunSorts; i++ ) {
```

```
            S = AN.FunSorts[i];
            if ( S ) {
                if ( S->file.handle >= 0 ) {
/*                  TruncateFile(S->file.handle); */
                    UpdateMaxSize();
                    CloseFile(S->file.handle);
                    S->file.handle = -1;
                    remove(S->file.name);
#ifdef GZIPDEBUG
                    MLOCK(ErrorMessageLock);
                    MesPrint("%w CleanUpSort removed file %s",S->file.name);
                    MUNLOCK(ErrorMessageLock);
#endif
                }
                M_free(S,"sorting struct");
            }
            AN.FunSorts[i] = 0;
        }
        AN.MaxFunSorts = minnum;
        if ( num == 0 ) {
            S = AN.FunSorts[0];
            if ( S ) {
                if ( S->file.handle >= 0 ) {
/*                  TruncateFile(S->file.handle); */
                    UpdateMaxSize();
                    CloseFile(S->file.handle);
                    S->file.handle = -1;
                    remove(S->file.name);
#ifdef GZIPDEBUG
                    MLOCK(ErrorMessageLock);
                    MesPrint("%w CleanUpSort removed file %s",S->file.name);
                    MUNLOCK(ErrorMessageLock);
#endif
                }
            }
        }
    }
    for ( i = 0; i < 2; i++ ) {
        if ( AR.FoStage4[i].handle >= 0 ) {
            UpdateMaxSize();
            CloseFile(AR.FoStage4[i].handle);
            remove(AR.FoStage4[i].name);
            AR.FoStage4[i].handle = -1;
#ifdef GZIPDEBUG
            MLOCK(ErrorMessageLock);
            MesPrint("%w CleanUpSort removed stage4 file %s",AR.FoStage4[i].name);
            MUNLOCK(ErrorMessageLock);
#endif
        }
    }
}

/*
        #] CleanUpSort :
        #[ LowerSortLevel :         VOID LowerSortLevel()
*/
VOID LowerSortLevel()
{
    GETIDENTITY
    if ( AR.sLevel >= 0 ) {
        AR.sLevel--;
        if ( AR.sLevel >= 0 ) AT.SS = AN.FunSorts[AR.sLevel];
    }
}

/*
        #] LowerSortLevel :
        #[ PolyRatFunSpecial :

        Keeps only the most divergent term in AR.PolyFunVar
        We assume that the terms are already in that notation.
*/

WORD *PolyRatFunSpecial(PHEAD WORD *t1, WORD *t2)
{
    WORD *oldworkpointer = AT.WorkPointer, *t, *r;
    WORD exp1, exp2;
    int i;
    t = t1+FUNHEAD;
    if ( *t == -SYMBOL ) {
        if ( t[1] != AR.PolyFunVar ) goto Illegal;
        exp1 = 1;
        if ( t[2] != -SNUMBER ) goto Illegal;
        t[3] = 1;
    }
    else if ( *t == -SNUMBER ) {
        t[1] = 1;
```

```
        t += 2;
        if ( *t == -SYMBOL ) {
            if ( t[1] != AR.PolyFunVar ) goto Illegal;
            exp1 = -1;
        }
        else if ( *t == -SNUMBER ) {
            t[1] = 1;
            exp1 = 0;
        }
        else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
            && t[ARGHEAD+3] == AR.PolyFunVar ) {
            t[ARGHEAD+5] = 1;
            t[ARGHEAD+6] = 1;
            t[ARGHEAD+7] = 3;
            exp1 = -t[ARGHEAD+4];
        }
        else goto Illegal;
    }
    else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
        && t[ARGHEAD+3] == AR.PolyFunVar ) {
        t[ARGHEAD+5] = 1;
        t[ARGHEAD+6] = 1;
        t[ARGHEAD+7] = 3;
        exp1 = t[ARGHEAD+4];
        t += *t;
        if ( *t != -SNUMBER ) goto Illegal;
        t[1] = 1;
    }
    else goto Illegal;

    t = t2+FUNHEAD;
    if ( *t == -SYMBOL ) {
        if ( t[1] != AR.PolyFunVar ) goto Illegal;
        exp2 = 1;
        if ( t[2] != -SNUMBER ) goto Illegal;
        t[3] = 1;
    }
    else if ( *t == -SNUMBER ) {
        t[1] = 1;
        t += 2;
        if ( *t == -SYMBOL ) {
            if ( t[1] != AR.PolyFunVar ) goto Illegal;
            exp2 = -1;
        }
        else if ( *t == -SNUMBER ) {
            t[1] = 1;
            exp2 = 0;
        }
        else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
            && t[ARGHEAD+3] == AR.PolyFunVar ) {
            t[ARGHEAD+5] = 1;
            t[ARGHEAD+6] = 1;
            t[ARGHEAD+7] = 3;
            exp2 = -t[ARGHEAD+4];
        }
        else goto Illegal;
    }
    else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
        && t[ARGHEAD+3] == AR.PolyFunVar ) {
        t[ARGHEAD+5] = 1;
        t[ARGHEAD+6] = 1;
        t[ARGHEAD+7] = 3;
        exp2 = t[ARGHEAD+4];
        t += *t;
        if ( *t != -SNUMBER ) goto Illegal;
        t[1] = 1;
    }
    else goto Illegal;

    if ( exp1 <= exp2 ) { i = t1[1]; r = t1; }
    else                { i = t2[1]; r = t2; }
    t = oldworkpointer;
    NCOPY(t,r,i)

    return(oldworkpointer);
Illegal:
    MesPrint("Illegal occurrence of PolyRatFun with divergent option");
    Terminate(-1);
    return(0);
}

/*
        #] PolyRatFunSpecial :
    #] SortUtilities :
*/
```

4.102 spectator.c File Reference

`#include "form3.h"`
Include dependency graph for spectator.c:

Functions

- int CoCreateSpectator (UBYTE *inp)
- int CoToSpectator (UBYTE *inp)
- int CoRemoveSpectator (UBYTE *inp)
- int CoEmptySpectator (UBYTE *inp)
- int PutInSpectator (WORD *term, WORD specnum)
- void FlushSpectators (VOID)
- int CoCopySpectator (UBYTE *inp)
- WORD GetFromSpectator (WORD *term, WORD specnum)
- void ClearSpectators (WORD par)

4.102.1 Detailed Description

File contains the code for the spectator files and their control.

Definition in file spectator.c.

4.102.2 Function Documentation

4.102.2.1 CoCreateSpectator()

```
int CoCreateSpectator (
            UBYTE * inp )
```

Definition at line 89 of file spectator.c.

4.102.2.2 CoToSpectator()

```
int CoToSpectator (
            UBYTE * inp )
```

Definition at line 181 of file spectator.c.

4.102.2.3 CoRemoveSpectator()

```
int CoRemoveSpectator (
            UBYTE * inp )
```

Definition at line 233 of file spectator.c.

4.102.2.4 CoEmptySpectator()

```
int CoEmptySpectator (
            UBYTE * inp )
```

Definition at line 289 of file spectator.c.

4.102.2.5 PutInSpectator()

```
int PutInSpectator (
            WORD * term,
            WORD specnum )
```

Definition at line 341 of file spectator.c.

4.102.2.6 FlushSpectators()

```
void FlushSpectators (
            VOID  )
```

Definition at line 398 of file spectator.c.

4.102.2.7 CoCopySpectator()

```
int CoCopySpectator (
            UBYTE * inp )
```

Definition at line 455 of file spectator.c.

4.102.2.8 GetFromSpectator()

```
WORD GetFromSpectator (
            WORD * term,
            WORD specnum )
```

Definition at line 593 of file spectator.c.

4.102.2.9 ClearSpectators()

```
void ClearSpectators (
            WORD par )
```

Definition at line 667 of file spectator.c.

4.103 spectator.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ includes :
*/

#include "form3.h"

/*
    #] includes :
    #[ Commentary :

    We use an array of SPECTATOR structs in AM.SpectatorFiles.
    When a spectator is removed this leaves a hole. This means that
    we cannot use AM.NumSpectatorFiles but always have to scan up to
    AM.SizeForSpectatorFiles which is the size of the array.
    An element is in use when it has a name. This is the name of the
    expression that is associated with it. There is also the number of
    the expression, but because the expressions are frequently renumbered
    at the end of a module, we always search for the spectators by name.
    The expression number is only valid in the current module.
    During execution we use the number of the spectator.

    The FILEHANDLE struct is borrowed from the structs for the scratch
    files, but we do not keep copies for all workers as with the scratch
    files. This brings some limitations (but saves much space). Basically
    the reading can only be done by one master or worker. And because
    we use the buffer both for writing and for reading we cannot read and
    write in the same module.

    Processor can see that an expression is to be filled with a spectator
    because we replace the compiler buffer number in the prototype by
    -specnum-1. Of course, after this filling has taken place we should
    make sure that in the next module there is a nonnegative number there.
    The input is then obtained from GetFromSpectator instead from GetTerm.
    This needed an extra argument in ThreadsProcessor. InParallelProcessor
    can figure it out by itself. ParFORM still needs to be treated for this.

    The writing is to a single buffer. Hence it needs a lock. It is possible
    to give all workers their own buffers (at great memory cost) and merge
    the results when needed. That would be friendlier on ParFORM. We ALWAYS
    assume that the order of the terms in the spectator file is random.

    In the first version there is no compression in the file. This could
    change in later versions because both the writing and the reading are
    purely sequential. Brackets are not possible.

    Currently, ParFORM allows use of spectators only in the sequential
    mode. The parallelization is switched off in modules containing
    ToSpectator or CopySpectator. Workers never create or access to
    spectator files. Their file handles are always -1. We leave the
    parallelization of modules with spectators for future work.

    #] Commentary :
    #[ CoCreateSpectator :

    Syntax: CreateSpectator name_of_expr "filename";
```

```
*/

int CoCreateSpectator(UBYTE *inp)
{
    UBYTE *p, *q, *filename, c, cc;
    WORD c1, c2, numexpr = 0, specnum, HadOne = 0;
    FILEHANDLE *fh;
    while ( *inp == ',' ) inp++;
    if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
        MesPrint("&Illegal name for expression");
        return(1);
    }
    c = *q; *q = 0;
    if ( GetVar(inp,&c1,&c2,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
        if ( c2 == CEXPRESSION &&
                Expressions[c1].status == DROPSPECTATOREXPRESSION ) {
            numexpr = c1;
            Expressions[numexpr].status = SPECTATOREXPRESSION;
            HadOne = 1;
        }
        else {
            MesPrint("&The name %s has been used already.",inp);
            *q = c;
            return(1);
        }
    }
    p = q+1;
    while ( *p == ',' ) p++;
    if ( *p != '"' ) goto Syntax;
    p++; filename = p;
    while ( *p && *p != '"' ) {
        if ( *p == '\\' ) p++;
        p++;
    }
    if ( *p != '"' ) goto Syntax;
    q = p+1;
    while ( *q && ( *q == ',' || *q == ' ' || *q == '\t' ) ) q++;
    if ( *q ) goto Syntax;
    cc = *p; *p = 0;
/*
    Now we need to: create a struct for the spectator file.
*/
    if ( HadOne == 0 )
        numexpr = EntVar(CEXPRESSION,inp,SPECTATOREXPRESSION,0,0,0);
    fh = AllocFileHandle(1,(char *)filename);
/*
    Make sure there is space in the AM.spectatorfiles array
*/
    if ( AM.NumSpectatorFiles >= AM.SizeForSpectatorFiles || AM.SpectatorFiles == 0 ) {
        int newsize, i;
        SPECTATOR *newspectators;
        if ( AM.SizeForSpectatorFiles == 0 ) {
            newsize = 10;
            AM.NumSpectatorFiles = AM.SizeForSpectatorFiles = 0;
        }
        else newsize = AM.SizeForSpectatorFiles*2;
        newspectators = (SPECTATOR *)Malloc1(newsize*sizeof(SPECTATOR),"Spectators");
        for ( i = 0; i < AM.NumSpectatorFiles; i++ )
            newspectators[i] = AM.SpectatorFiles[i];
        for ( ; i < newsize; i++ ) {
            newspectators[i].fh = 0;
            newspectators[i].name = 0;
            newspectators[i].exprnumber = -1;
            newspectators[i].flags = 0;
            PUTZERO(newspectators[i].position);
            PUTZERO(newspectators[i].readpos);
        }
        AM.SizeForSpectatorFiles = newsize;
        if ( AM.SpectatorFiles != 0 ) M_free(AM.SpectatorFiles,"Spectators");
        AM.SpectatorFiles = newspectators;
        specnum = AM.NumSpectatorFiles++;
    }
    else {
        for ( specnum = 0; specnum < AM.SizeForSpectatorFiles; specnum++ ) {
            if ( AM.SpectatorFiles[specnum].name == 0 ) break;
        }
        AM.NumSpectatorFiles++;
    }
    PUTZERO(AM.SpectatorFiles[specnum].position);
    AM.SpectatorFiles[specnum].name = (char *)(strDup1(inp,"Spectator expression name"));
    AM.SpectatorFiles[specnum].fh = fh;
    AM.SpectatorFiles[specnum].exprnumber = numexpr;
    *p = cc;
    return(0);
Syntax:
    MesPrint("&Proper syntax is: CreateSpectator,exprname,\"filename\";");
    return(-1);
```

```
}

/*
    #] CoCreateSpectator :
    #[ CoToSpectator :
*/

int CoToSpectator(UBYTE *inp)
{
    UBYTE *q;
    WORD c1, numexpr;
    int i;
    while ( *inp == ',' ) inp++;
    if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
        MesPrint("&Illegal name for expression");
        return(1);
    }
    if ( *q != 0 ) goto Syntax;
    if ( GetVar(inp,&c1,&numexpr,ALLVARIABLES,NOAUTO) == NAMENOTFOUND ||
            c1 != CEXPRESSION ) {
        MesPrint("&%s is not a valid expression.",inp);
        return(1);
    }
    if ( Expressions[numexpr].status != SPECTATOREXPRESSION ) {
        MesPrint("&%s is not an active spectator.",inp);
        return(1);
    }
    for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
      if ( AM.SpectatorFiles[i].name != 0 ) {
        if ( StrCmp((UBYTE *)(AM.SpectatorFiles[i].name),(UBYTE *)(inp)) == 0 ) break;
      }
    }
    if ( i >= AM.SizeForSpectatorFiles ) {
        MesPrint("&Spectator %s not found.",inp);
        return(1);
    }
    if ( ( AM.SpectatorFiles[i].flags & READSPECTATORFLAG ) != 0 ) {
        MesPrint("&Spectator %s: It is not permitted to read from and write to the same spectator in one module.",inp);
        return(1);
    }
    AM.SpectatorFiles[i].exprnumber = numexpr;
    Add3Com(TYPETOSPECTATOR,i);
#ifdef WITHMPI
    /*
     * In ParFORM, ToSpectator has to be executed on the master.
     */
    AC.mparallelflag |= NOPARALLEL_SPECTATOR;
#endif
    return(0);
Syntax:
    MesPrint("&Proper syntax is: ToSpectator,exprname;");
    return(-1);
}

/*
    #] CoToSpectator :
    #[ CoRemoveSpectator :
*/

int CoRemoveSpectator(UBYTE *inp)
{
    UBYTE *q;
    WORD c1, numexpr;
    int i;
    SPECTATOR *sp;
    while ( *inp == ',' ) inp++;
    if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
        MesPrint("&Illegal name for expression");
        return(1);
    }
    if ( *q != 0 ) goto Syntax;
    if ( GetVar(inp,&c1,&numexpr,ALLVARIABLES,NOAUTO) == NAMENOTFOUND ||
            c1 != CEXPRESSION ) {
        MesPrint("&%s is not a valid expression.",inp);
        return(1);
    }
    if ( Expressions[numexpr].status != SPECTATOREXPRESSION ) {
        MesPrint("&%s is not a spectator.",inp);
        return(1);
    }
    for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
        if ( StrCmp((UBYTE *)(AM.SpectatorFiles[i].name),(UBYTE *)(inp)) == 0 ) break;
    }
    if ( i >= AM.SizeForSpectatorFiles ) {
        MesPrint("&Spectator %s not found.",inp);
        return(1);
```

```
    }
    sp = AM.SpectatorFiles+i;
    Expressions[numexpr].status = DROPSPECTATOREXPRESSION;
    if ( sp->fh->handle != -1 ) {
        CloseFile(sp->fh->handle);
        sp->fh->handle = -1;
        remove(sp->fh->name);
    }
    M_free(sp->fh,"Temporary FileHandle");
    M_free(sp->name,"Spectator expression name");

    PUTZERO(sp->position);
    PUTZERO(sp->readpos);
    sp->fh = 0;
    sp->name = 0;
    sp->exprnumber = -1;
    sp->flags = 0;
    AM.NumSpectatorFiles--;
    return(0);
Syntax:
    MesPrint("&Proper syntax is: RemoveSpectator,exprname;");
    return(-1);
}

/*
    #] CoRemoveSpectator :
    #[ CoEmptySpectator :
*/

int CoEmptySpectator(UBYTE *inp)
{
    UBYTE *q;
    WORD c1, numexpr;
    int i;
    SPECTATOR *sp;
    while ( *inp == ',' ) inp++;
    if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
        MesPrint("&Illegal name for expression");
        return(1);
    }
    if ( *q != 0 ) goto Syntax;
    if ( GetVar(inp,&c1,&numexpr,ALLVARIABLES,NOAUTO) == NAMENOTFOUND ||
            c1 != CEXPRESSION ) {
        MesPrint("&%s is not a valid expression.",inp);
        return(1);
    }
    if ( Expressions[numexpr].status != SPECTATOREXPRESSION ) {
        MesPrint("&%s is not a spectator.",inp);
        return(1);
    }
    for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
        if ( StrCmp((UBYTE *)(AM.SpectatorFiles[i].name),(UBYTE *)(inp)) == 0 ) break;
    }
    if ( i >= AM.SizeForSpectatorFiles ) {
        MesPrint("&Spectator %s not found.",inp);
        return(1);
    }
    sp = AM.SpectatorFiles+i;
    if ( sp->fh->handle != -1 ) {
        CloseFile(sp->fh->handle);
        sp->fh->handle = -1;
        remove(sp->fh->name);
    }
    sp->fh->POfill = sp->fh->POfull = sp->fh->PObuffer;
    PUTZERO(sp->position);
    PUTZERO(sp->readpos);
    return(0);
Syntax:
    MesPrint("&Proper syntax is: EmptySpectator,exprname;");
    return(-1);
}

/*
    #] CoEmptySpectator :
    #[ PutInSpectator :

    We need to use locks! There is only one file.
    The code was copied (and modified) from PutOut.
    Here we use no compression.
*/

int PutInSpectator(WORD *term,WORD specnum)
{
    GETBIDENTITY
    WORD i, *p, ret;
    LONG RetCode;
    SPECTATOR *sp = &(AM.SpectatorFiles[specnum]);
```

```
    FILEHANDLE *fi = sp->fh;

    if ( ( i = *term ) <= 0 ) return(0);
    LOCK(fi->pthreadslock);
    ret = i;
    p = fi->POfill;
    do {
        if ( p >= fi->POstop ) {
            if ( fi->handle < 0 ) {
                if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
                    fi->handle = (WORD)RetCode;
                    PUTZERO(fi->filesize);
                    PUTZERO(fi->POposition);
                }
                else {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Cannot create spectator file %s",fi->name);
                    MUNLOCK(ErrorMessageLock);
                    UNLOCK(fi->pthreadslock);
                    return(-1);
                }
            }
            SeekFile(fi->handle,&(sp->position),SEEK_SET);
            if ( ( RetCode = WriteFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize) ) != fi->POsize ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Error during spectator write. Disk full?");
                MesPrint("Attempt to write %l bytes on file %d at position %15p",
                            fi->POsize,fi->handle,&(fi->POposition));
                MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG)(fi->PObuffer));
                MUNLOCK(ErrorMessageLock);
                UNLOCK(fi->pthreadslock);
                return(-1);
            }
            ADDPOS(fi->filesize,fi->POsize);
            p = fi->PObuffer;
            ADDPOS(sp->position,fi->POsize);
            fi->POposition = sp->position;
        }
        *p++ = *term++;
    } while ( --i > 0 );
    fi->POfull = fi->POfill = p;
    Expressions[AM.SpectatorFiles[specnum].exprnumber].counter++;
    UNLOCK(fi->pthreadslock);
    return(ret);
}

/*
    #] PutInSpectator :
    #[ FlushSpectators :
*/

void FlushSpectators(VOID)
{
    SPECTATOR *sp = AM.SpectatorFiles;
    FILEHANDLE *fh;
    LONG RetCode;
    int i;
    LONG size;
    if ( AM.NumSpectatorFiles <= 0 ) return;
    for ( i = 0; i < AM.SizeForSpectatorFiles; i++, sp++ ) {
        if ( sp->name == 0 ) continue;
        fh = sp->fh;
        if ( ( sp->flags & READSPECTATORFLAG ) != 0 ) { /* reset for writing */
            sp->flags &= ~READSPECTATORFLAG;
            fh->POfill = fh->PObuffer;
            if ( fh->handle >= 0 ) {
                SeekFile(fh->handle,&(sp->position),SEEK_SET);
                fh->POposition = sp->position;
            }
            continue;
        }
        if ( fh->POfill <= fh->PObuffer ) continue; /* is clean */
        if ( fh->handle < 0 ) {   /* File needs to be created */
            if ( ( RetCode = CreateFile(fh->name) ) >= 0 ) {
                PUTZERO(fh->filesize);
                PUTZERO(fh->POposition);
                fh->handle = (WORD)RetCode;
            }
            else {
                MLOCK(ErrorMessageLock);
                MesPrint("Cannot create spectator file %s",fh->name);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            PUTZERO(sp->position);
        }
```

```
        SeekFile(fh->handle,&(sp->position),SEEK_SET);
        size = (fh->POfill - fh->PObuffer)*sizeof(WORD);
        if ( ( RetCode = WriteFile(fh->handle,(UBYTE *)(fh->PObuffer),size) ) != size ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Write error synching spectator file. Disk full?");
            MesPrint("Attempt to write %l bytes on file %s at position %15p",
                    size,fh->name,&(sp->position));
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        fh->POfill = fh->PObuffer;
        SeekFile(fh->handle,&(sp->position),SEEK_END);
        fh->POposition = sp->position;
    }
    return;
}

/*
    #] FlushSpectators :
    #[ CoCopySpectator :
*/

int CoCopySpectator(UBYTE *inp)
{
    GETIDENTITY
    UBYTE *q, c, *exprname, *p;
    WORD c1, c2, numexpr;
    int specnum, error = 0;
    SPECTATOR *sp;
    while ( *inp == ',' ) inp++;
    if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
        MesPrint("&Illegal name for expression");
        return(1);
    }
    if ( *q == 0 ) goto Syntax;
    c = *q; *q = 0;
    if ( GetVar(inp,&c1,&c2,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
        MesPrint("&%s is the name of an existing variable.",inp);
        return(1);
    }
    numexpr = EntVar(CEXPRESSION,inp,LOCALEXPRESSION,0,0,0);
    p = q;
    exprname = inp;
    *q = c;
    while ( *q == ' ' || *q == ',' || *q == '\t' ) q++;
    if ( *q != '=' ) goto Syntax;
    q++;
    while ( *q == ' ' || *q == ',' || *q == '\t' ) q++;
    inp = q;
    if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
        MesPrint("&Illegal name for spectator expression");
        return(1);
    }
    if ( *q != 0 ) goto Syntax;
    if ( AM.NumSpectatorFiles <= 0 ) {
        MesPrint("&CopySpectator: There are no spectator expressions!");
        return(1);
    }
    sp = AM.SpectatorFiles;
    for ( specnum = 0; specnum < AM.SizeForSpectatorFiles; specnum++, sp++ ) {
      if ( sp->name != 0 ) {
        if ( StrCmp((UBYTE *)(sp->name),(UBYTE *)(inp)) == 0 ) break;
      }
    }
    if ( specnum >= AM.SizeForSpectatorFiles ) {
        MesPrint("&Spectator %s not found.",inp);
        return(1);
    }
    sp->flags |= READSPECTATORFLAG;
    PUTZERO(sp->fh->POposition);
    PUTZERO(sp->readpos);
    sp->fh->POfill = sp->fh->PObuffer;
    if ( sp->fh->handle >= 0 ) {
        SeekFile(sp->fh->handle,&(sp->fh->POposition),SEEK_SET);
    }
/*
    Now we have:
    1: The name of the target expression: numexpr
    2: The spectator: sp (or specnum).
    Time for some action. We need:
    a: Write a prototype to create the expression
    b: Signal to Processor that this is a spectator.
       We do this by giving a negative compiler buffer number.
*/
    {
        WORD *OldWork, *w;
        POSITION pos;
```

```
        OldWork = w = AT.WorkPointer;
        *w++ = TYPEEXPRESSION;
        *w++ = 3+SUBEXPSIZE;
        *w++ = numexpr;
        AC.ProtoType = w;
        AR.CurExpr = numexpr;               /* Block expression numexpr */
        *w++ = SUBEXPRESSION;
        *w++ = SUBEXPSIZE;
        *w++ = numexpr;
        *w++ = 1;
        *w++ = -specnum-1;      /* Indicates "spectator" to Processor */
        FILLSUB(w)
        *w++ = 1;
        *w++ = 1;
        *w++ = 3;
        *w++ = 0;
        SeekScratch(AR.outfile,&pos);
        Expressions[numexpr].counter = 1;
        Expressions[numexpr].onfile = pos;
        Expressions[numexpr].whichbuffer = 0;
#ifdef PARALLELCODE
        Expressions[numexpr].partodo = AC.inparallelflag;
#endif
        OldWork[2] = w - OldWork - 3;
        AT.WorkPointer = w;

        if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0 ) {
            c = *p; *p = 0;
            MesPrint("&Cannot create expression %s",exprname);
            *p = c;
            error = -1;
        }
        else {
            OldWork[2] = 4+SUBEXPSIZE;
            OldWork[4] = SUBEXPSIZE;
            OldWork[5] = numexpr;
            OldWork[SUBEXPSIZE+3] = 1;
            OldWork[SUBEXPSIZE+4] = 1;
            OldWork[SUBEXPSIZE+5] = 3;
            OldWork[SUBEXPSIZE+6] = 0;
            if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0
            || FlushOut(&pos,AR.outfile,0) ) {
                c = *p; *p = 0;
                MesPrint("&Cannot create expression %s",exprname);
                *p = c;
                error = -1;
            }
            AR.outfile->POfull = AR.outfile->POfill;
        }
        OldWork[2] = numexpr;
        AddNtoL(OldWork[1],OldWork);
        AT.WorkPointer = OldWork;
        if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
    }
#ifdef WITHMPI
    /*
     * In ParFORM, substitutions of spectators has to be done on the master.
     */
    AC.mparallelflag |= NOPARALLEL_SPECTATOR;
#endif
    return(error);
Syntax:
    MesPrint("&Proper syntax is: CopySpectator,exprname=spectatorname;");
    return(-1);
}

/*
    #] CoCopySpectator :
    #[ GetFromSpectator :

    Note that if we did things right, we do not need a lock for the reading.
*/

WORD GetFromSpectator(WORD *term,WORD specnum)
{
    SPECTATOR *sp = &(AM.SpectatorFiles[specnum]);
    FILEHANDLE *fh = sp->fh;
    WORD i, size, *t = term;
    LONG InIn;
    if ( fh-> handle < 0 ) {
        *term = 0;
        return(0);
    }
/*
    sp->position marks the 'end' of the file: the point where writing should
    take place. sp->readpos marks from where to read.
    fh->POposition marks where the file is currently positioned.
```

```
    Note that when we read, we need to
*/
    if ( ISZEROPOS(sp->readpos) ) { /* we start reading. Fill buffer. */
FillBuffer:
        SeekFile(fh->handle,&(sp->readpos),SEEK_SET);
        InIn = ReadFile(fh->handle,(UBYTE *)(fh->PObuffer),fh->POsize);
        if ( InIn < 0 || ( InIn & 1 ) ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Error reading information for %s spectator",sp->name);
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
        InIn /= sizeof(WORD);
        if ( InIn == 0 ) { *term = 0; return(0); }
        SeekFile(fh->handle,&(sp->readpos),SEEK_CUR);
        fh->POposition = sp->readpos;
        fh->POfull = fh->PObuffer+InIn;
        fh->POfill = fh->PObuffer;
    }
    if ( fh->POfill == fh->POfull ) { /* not even the size of the term! */
        if ( ISLESSPOS(sp->readpos,sp->position) ) goto FillBuffer;
        *term = 0;
        return(0);
    }
    size = *fh->POfill++; *t++ = size;
    for ( i = 1; i < size; i++ ) {
        if ( fh->POfill >= fh->POfull ) {
            SeekFile(fh->handle,&(sp->readpos),SEEK_SET);
            InIn = ReadFile(fh->handle,(UBYTE *)(fh->PObuffer),fh->POsize);
            if ( InIn < 0 || ( InIn & 1 ) ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Error reading information for %s spectator",sp->name);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            InIn /= sizeof(WORD);
            if ( InIn == 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Reading incomplete information for %s spectator",sp->name);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            SeekFile(fh->handle,&(sp->readpos),SEEK_CUR);
            fh->POposition = sp->readpos;
            fh->POfull = fh->PObuffer+InIn;
            fh->POfill = fh->PObuffer;
        }
        *t++ = *fh->POfill++;
    }
    return(size);
}

/*
    #] GetFromSpectator :
    #[ ClearSpectators :

    Removes all spectators.
    In case of .store, the ones that are protected by .global stay.
*/

void ClearSpectators(WORD par)
{
    SPECTATOR *sp = AM.SpectatorFiles;
    WORD numexpr, c1;
    int i;
    if ( AM.NumSpectatorFiles > 0 ) {
        for ( i = 0; i < AM.SizeForSpectatorFiles; i++, sp++ ) {
            if ( sp->name == 0 ) continue;
            if ( ( sp->flags & GLOBALSPECTATORFLAG ) == 1 && par == STOREMODULE ) continue;

            if ( GetVar((UBYTE *)(sp->name),&c1,&numexpr,ALLVARIABLES,NOAUTO) == NAMENOTFOUND ||
                c1 != CEXPRESSION ) {
                MesPrint("&%s is not a valid expression.",sp->name);
                continue;
            }
            Expressions[numexpr].status = DROPPEDEXPRESSION;
            if ( sp->fh->handle != -1 ) {
                CloseFile(sp->fh->handle);
                sp->fh->handle = -1;
                remove(sp->fh->name);
            }
            M_free(sp->fh,"Temporary FileHandle");
            M_free(sp->name,"Spectator expression name");
            PUTZERO(sp->position);
            sp->fh = 0;
            sp->name = 0;
            sp->exprnumber = -1;
```

```
            sp->flags = 0;
            AM.NumSpectatorFiles--;
        }
    }
}

/*
    #] ClearSpectators :
*/
```

4.104 startup.c File Reference

```
#include "form3.h"
#include "inivar.h"
#include <signal.h>
```

Include dependency graph for startup.c:

Macros

- #define STRINGIFY(x) STRINGIFY__(x)
- #define STRINGIFY__(x) #x
- #define FORMNAME "FORM"
- #define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION)
- #define VERSIONSTR FORMNAME " " VERSIONSTR__ " (" PRODUCTIONDATE ")"
- #define TAKEPATH(x) if(s[1]== '='){x=s+2;} else{x=*argv++;argc--;}

Functions

- int DoTail (int argc, UBYTE **argv)
- int OpenInput ()
- VOID ReserveTempFiles (int par)
- VOID StartVariables ()
- VOID StartMore ()
- WORD IniVars ()
- int main (int argc, char **argv)
- VOID CleanUp (WORD par)
- VOID Terminate (int errorcode)
- VOID PrintRunningTime ()
- LONG GetRunningTime ()

Variables

- UBYTE * emptystring = (UBYTE *)""
- UBYTE * defaulttempfilename = (UBYTE *)"xformxxx.str"

4.104.1 Detailed Description

This file contains the main program. It also deals with the very early stages of the startup of FORM and the final stages when the program attemps some cleanup. Here is the routine that analyses the command tail.

Definition in file startup.c.

4.104.2 Macro Definition Documentation

4.104.2.1 STRINGIFY

```
#define STRINGIFY(
              x ) STRINGIFY__(x)
```

Definition at line 50 of file startup.c.

4.104.2.2 STRINGIFY__

```
#define STRINGIFY__(
              x ) #x
```

Definition at line 51 of file startup.c.

4.104.2.3 FORMNAME

```
#define FORMNAME "FORM"
```

Definition at line 61 of file startup.c.

4.104.2.4 VERSIONSTR__

```
#define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION)
```

Definition at line 92 of file startup.c.

4.104.2.5 VERSIONSTR

```
#define VERSIONSTR FORMNAME " " VERSIONSTR__ " (" PRODUCTIONDATE ")"
```

Definition at line 94 of file startup.c.

4.104.2.6 TAKEPATH

```
#define TAKEPATH(
              x ) if(s[1]== '=' ){x=s+2;} else{x=*argv++;argc--;}
```

Definition at line 215 of file startup.c.

4.104.3 Function Documentation

4.104.3.1 DoTail()

```
int DoTail (
            int argc,
            UBYTE ** argv )
```

Definition at line 217 of file startup.c.

4.104.3.2 OpenInput()

```
int OpenInput ( )
```

Definition at line 480 of file startup.c.

4.104.3.3 ReserveTempFiles()

```
VOID ReserveTempFiles (
            int par )
```

Definition at line 593 of file startup.c.

4.104.3.4 StartVariables()

```
VOID StartVariables ( )
```

All functions (well, nearly all) are declared here.

Definition at line 828 of file startup.c.

References AddNtoC(), AddRHS(), Compare1(), inicbufs(), FuNcTiOn::name, PF_BroadcastNumber(), PutPreVar(), and FuNcTiOn::symmetric.

Here is the call graph for this function:

4.104.3.5 StartMore()

```
VOID StartMore ( )
```

Definition at line 1200 of file startup.c.

4.104.3.6 IniVars()

```
WORD IniVars ( )
```

Definition at line 1238 of file startup.c.

4.104.3.7 main()

```
int main (
            int argc,
            char ** argv )
```

Definition at line 1526 of file startup.c.

4.104.3.8 CleanUp()

```
VOID CleanUp (
            WORD par )
```

Definition at line 1619 of file startup.c.

4.104.3.9 Terminate()

```
VOID Terminate (
            int errorcode )
```

Definition at line 1696 of file startup.c.

4.104.3.10 PrintRunningTime()

```
VOID PrintRunningTime ( )
```

Definition at line 1777 of file startup.c.

4.104.3.11 GetRunningTime()

```
LONG GetRunningTime ( )
```

Definition at line 1818 of file startup.c.

4.104.4 Variable Documentation

4.104.4.1 emptystring

```
UBYTE* emptystring = (UBYTE *)"."
```

Definition at line 590 of file startup.c.

4.104.4.2 defaulttempfilename

```
UBYTE* defaulttempfilename = (UBYTE *)"xformxxx.str"
```

Definition at line 591 of file startup.c.

4.105 startup.c

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034 /*
00035         #[ includes :
00036 */
00037
00038 #include "form3.h"
00039 #include "inivar.h"
00040
00041 #ifdef TRAPSIGNALS
00042 #include "portsignals.h"
00043 #else
00044 #include <signal.h>
00045 #endif
00046
00047 /*
00048  * A macro for translating the contents of `x' into a string after expanding.
00049  */
00050 #define STRINGIFY(x)  STRINGIFY__(x)
00051 #define STRINGIFY__(x) #x
00052
00053 /*
00054  * FORMNAME = "FORM" or "TFORM" or "ParFORM".
00055  */
00056 #if defined(WITHPTHREADS)
00057     #define FORMNAME "TFORM"
00058 #elif defined(WITHMPI)
00059     #define FORMNAME "ParFORM"
00060 #else
00061     #define FORMNAME "FORM"
00062 #endif
00063
00064 /*
00065  * VERSIONSTR is the version information printed in the header line.
00066  */
00067 #ifdef HAVE_CONFIG_H
00068     /* We have also version.h. */
00069     #include "version.h"
00070     #ifndef REPO_VERSION
00071         #define REPO_VERSION STRINGIFY(REPO_MAJOR_VERSION) "." STRINGIFY(REPO_MINOR_VERSION)
00072     #endif
00073     #ifndef REPO_DATE
00074         /* The build date, instead of the repo date. */
00075         #define REPO_DATE __DATE__
00076     #endif
00077     #ifdef REPO_REVISION
00078         #define VERSIONSTR FORMNAME " " REPO_VERSION " (" REPO_DATE ", " REPO_REVISION ")"
00079     #else
00080         #define VERSIONSTR FORMNAME " " REPO_VERSION " (" REPO_DATE ")"
00081     #endif
00082     #define MAJORVERSION REPO_MAJOR_VERSION
00083     #define MINORVERSION REPO_MINOR_VERSION
00084 #else
00085     /*
00086      * Otherwise, form3.h defines MAJORVERSION, MINORVERSION and PRODUCTIONDATE,
00087      * possibly BETAVERSION.
00088      */
```

```
    #ifdef BETAVERSION
        #define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION) "Beta"
    #else
        #define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION)
    #endif
    #define VERSIONSTR FORMNAME " " VERSIONSTR__ " (" PRODUCTIONDATE ")"
#endif

/*
        #] includes :
        #[ PrintHeader :
*/

static void PrintHeader(int with_full_info)
{
#ifdef WITHMPI
    if ( PF.me == MASTER && !AM.silent ) {
#else
    if ( !AM.silent ) {
#endif
        char buffer1[250], buffer2[80], *s = buffer1, *t = buffer2;
        WORD length, n;
        for ( n = 0; n < 250; n++ ) buffer1[n] = ' ';
        /*
         * NOTE: we expect that the compiler optimizes strlen("string literal")
         * to just a number.
         */
        if ( strlen(VERSIONSTR) <= 100 ) {
            strcpy(s,VERSIONSTR);
            s += strlen(VERSIONSTR);
            *s = 0;
        }
        else {
            /*
             * Truncate when it is too long.
             */
            strncpy(s,VERSIONSTR,97);
            s[97] = '.';
            s[98] = '.';
            s[99] = ')';
            s[100] = '\0';
            s += 100;
        }

        s += sprintf(s," %d-bits",(WORD)(sizeof(WORD)*16));
/*      while ( *s ) s++; */
        *s = 0;

        if ( with_full_info ) {
#if defined(WITHPTHREADS) || defined(WITHMPI)
#if defined(WITHPTHREADS)
            int nworkers = AM.totalnumberofthreads-1;
#elif defined(WITHMPI)
            int nworkers = PF.numtasks-1;
#endif
            s += sprintf(s," %d worker",nworkers);
            *s = 0;
/*          while ( *s ) s++; */
            if ( nworkers != 1 ) {
                *s++ = 's';
                *s = '\0';
            }
#endif

            sprintf(t,"Run: %s",MakeDate());
            while ( *t ) t++;

            /*
             * Align the date to the right, if it fits in a line.
             */
            length = (s-buffer1) + (t-buffer2);
            if ( length+2 <= AC.LineLength ) {
                for ( n = AC.LineLength-length; n > 0; n-- ) *s++ = ' ';
                *s = 0;
                strcat(s,buffer2);
                while ( *s ) s++;
            }
            else {
                *s = 0;
                strcat(s,"  ");
                while ( *s ) s++;
                *s = 0;
                strcat(s,buffer2);
                while ( *s ) s++;
            }
        }

```

```
        /*
         * If the header information doesn't fit in a line, we need to extend
         * the line length temporarily.
         */
        length = s-buffer1;
        if ( length <= AC.LineLength ) {
            MesPrint("%s",buffer1);
        }
        else {
            WORD oldLineLength = AC.LineLength;
            AC.LineLength = length;
            MesPrint("%s",buffer1);
            AC.LineLength = oldLineLength;
        }
    }
}

/*
        #] PrintHeader :
        #[ DoTail :

        Routine reads the command tail and handles the commandline options.
        It sets the flags for later actions and stored pathnames for
        the setup file, include/prc/sub directories etc.
        Finally the name of the program is passed on.
        Note that we do not support interactive use yet. This will come
        to pass in the distant future when we can couple STedi to FORM.
        Routine made 23-feb-1993 by J.Vermaseren
*/

#ifdef WITHINTERACTION
static UBYTE deflogname[] = "formsession.log";
#endif

#define TAKEPATH(x) if(s[1]== '=' ){x=s+2;} else{x=*argv++;argc--;}

int DoTail(int argc, UBYTE **argv)
{
    int errorflag = 0, onlyversion = 1;
    UBYTE *s, *t, *copy;
    int threadnum = 0;
    argc--; argv++;
    AM.LogType = -1;
    AM.HoldFlag = AM.qError = AM.Interact = AM.FileOnlyFlag = 0;
    AM.InputFileName = AM.LogFileName = AM.IncDir = AM.TempDir = AM.TempSortDir =
    AM.SetupDir = AM.SetupFile = AM.Path = 0;
    if ( argc < 1 ) {
        onlyversion = 0;
        goto printversion;
    }
    while ( argc >= 1 ) {
        s = *argv++; argc--;
        if ( *s == '-' || ( *s == '/' && ( argc > 0 || AM.Interact ) ) ) {
            s++;
            switch (*s) {
                case 'c': /* Error checking only */
                            AM.qError = 1;   break;
                case 'D':
                case 'd': /* Next arg is define preprocessor var. */
                            t = copy = strDup1(*argv,"Dotail");
                            while ( *t && *t != '=' ) t++;
                            if ( *t == 0 ) {
                                if ( PutPreVar(copy,(UBYTE *)"1",0,0) < 0 ) return(-1);
                            }
                            else {
                                *t++ = 0;
                                if ( PutPreVar(copy,t,0,0) < 0 ) return(-1);
                                t[-1] = '=';
                            }
                            M_free(copy,"-d prevar");
                            argv++; argc--; break;
                case 'f': /* Output only to regular log file */
                            AM.FileOnlyFlag = 1; AM.LogType = 0; break;
                case 'F': /* Output only to log file. Further like L. */
                            AM.FileOnlyFlag = 1; AM.LogType = 1; break;
                case 'h': /* For old systems: wait for key before exit */
                            AM.HoldFlag = 1; break;
#ifdef WITHINTERACTION
                case 'i': /* Interactive session (not used yet) */
                            AM.Interact = 1; break;
#endif
                case 'I': /* Next arg is dir for inc/prc/sub files */
                            TAKEPATH(AM.IncDir)  break;
                case 'l': /* Make regular log file */
                            if ( s[1] == 'l' ) AM.LogType = 1; /*compatibility! */
                            else               AM.LogType = 0;
                            break;
```

```
                case 'L': /* Make log file with only final statistics */
                            AM.LogType = 1;  break;
                case 'M': /* Multirun. Name of tempfiles will contain PID */
                            AM.MultiRun = 1;
                            break;
                case 'm': /* Read number of threads */
                case 'w': /* Read number of workers */
                            t = s++;
                            threadnum = 0;
                            while ( *s >= '0' && *s <= '9' )
                                threadnum = 10*threadnum + *s++ - '0';
                            if ( *s ) {
#ifdef WITHMPI
                                if ( PF.me == MASTER )
#endif
                                printf("Illegal value for option m or w: %s\n",t);
                                errorflag++;
                            }
/*                          if ( threadnum == 1 ) threadnum = 0; */
                            threadnum++;
                            break;
                case 'W': /* Print the wall-clock time on the master. */
                            AM.ggWTimeStatsFlag = 1;
                            break;
/*
                case 'n':
                            Reserved for number of slaves without MPI
*/
                case 'p':
#ifdef WITHEXTERNALCHANNEL
                    /*There are two possibilities: -p|-pipe*/
                    if(s[1]=='i'){
                        if( (s[2]=='p')&&(s[3]=='e')&&(s[4]=='\0') ){
                            argc--;
                            /*Initialize pre-set external channels, see
                                the file extcmd.c:*/
                            if(initPresetExternalChannels(*argv++,AX.timeout)<1){
#ifdef WITHMPI
                                if ( PF.me == MASTER )
#endif
                                printf("Error initializing preset external channels\n");
                                errorflag++;
                            }
                            AX.timeout=-1;/*This indicates that preset channels
                                                    are initialized from cmdline*/
                        }else{
#ifdef WITHMPI
                            if ( PF.me == MASTER )
#endif
                            printf("Illegal option in call of FORM: %s\n",s);
                            errorflag++;
                        }
                    }else
#else
                    if ( s[1] ) {
                        if ( ( s[1]=='i' ) && ( s[2] == 'p' ) && (s[3] == 'e' )
                        && ( s[4] == '\0' ) ){
#ifdef WITHMPI
                            if ( PF.me == MASTER )
#endif
                            printf("Illegal option: Pipes not supported on this system.\n");
                        }
                        else {
#ifdef WITHMPI
                            if ( PF.me == MASTER )
#endif
                            printf("Illegal option: %s\n",s);
                        }
                        errorflag++;
                    }
                    else
#endif
                    {
                             /* Next arg is a path variable like in environment */
                        TAKEPATH(AM.Path)
                    }
                    break;
                case 'q': /* Quiet option. Only output. Same as -si */
                            AM.silent = 1; break;
                case 'R': /* recover from saved snapshot */
                            AC.CheckpointFlag = -1;
                            break;
                case 's': /* Next arg is dir with form.set to be used */
                            if ( ( s[1] == 'o' ) && ( s[2] == 'r' ) && ( s[3] == 't' ) ) {
                                if(s[4]== '=' ) {
                                    AM.TempSortDir = s+5;
                                }
```

```
                                else {
                                    AM.TempSortDir = *argv++;
                                    argc--;
                                }
                            }
                            else if ( s[1] == 'i' ) { /* compatibility: silent/quiet */
                                AM.silent = 1;
                            }
                            else {
                                TAKEPATH(AM.SetupDir)
                            }
                            break;
                    case 'S': /* Next arg is setup file */
                            TAKEPATH(AM.SetupFile) break;
                    case 't': /* Next arg is directory for temp files */
                            if ( s[1] == 's' ) {
                                s++;
                                AM.havesortdir = 1;
                                TAKEPATH(AM.TempSortDir)
                            }
                            else {
                                TAKEPATH(AM.TempDir)
                            }
                            break;
                    case 'T': /* Print the total size used at end of job */
                            AM.PrintTotalSize = 1; break;
                    case 'v':
printversion:;
#ifdef WITHMPI
                            if ( PF.me == MASTER )
#endif
                                PrintHeader(0);
                            if ( onlyversion ) return(1);
                            goto NoFile;
                    case 'y': /* Preprocessor dumps output. No compilation. */
                            AP.PreDebug = PREPROONLY;   break;
                    default:
                            if ( FG.cTable[*s] == 1 ) {
                                AM.SkipClears = 0; t = s;
                                while ( FG.cTable[*t] == 1 )
                                    AM.SkipClears = 10*AM.SkipClears + *t++ - '0';
                                if ( *t != 0 ) {
#ifdef WITHMPI
                                    if ( PF.me == MASTER )
#endif
                                    printf("Illegal numerical option in call of FORM: %s\n",s);
                                    errorflag++;
                                }
                            }
                            else {
#ifdef WITHMPI
                                if ( PF.me == MASTER )
#endif
                                printf("Illegal option in call of FORM: %s\n",s);
                                errorflag++;
                            }
                            break;
                }
            }
            else if ( argc == 0 && !AM.Interact ) AM.InputFileName = argv[-1];
            else {
#ifdef WITHMPI
                if ( PF.me == MASTER )
#endif
                printf("Illegal option in call of FORM: %s\n",s);
                errorflag++;
            }
        }
        AM.totalnumberofthreads = threadnum;
        if ( AM.InputFileName ) {
            s = AM.InputFileName;
            while ( *s ) s++;
            if ( s < AM.InputFileName+4 ||
            s[-4] != '.' || s[-3] != 'f' || s[-2] != 'r' || s[-1] != 'm' ) {
                t = (UBYTE *)Malloc1((s-AM.InputFileName)+5,"adding .frm");
                s = AM.InputFileName;
                AM.InputFileName = t;
                while ( *s ) *t++ = *s++;
                *t++ = '.'; *t++ = 'f'; *t++ = 'r'; *t++ = 'm'; *t = 0;
            }
            if ( AM.LogType >= 0 ) {
                AM.LogFileName = strDup1(AM.InputFileName,"name of logfile");
                s = AM.LogFileName;
                while ( *s ) s++;
                s[-3] = 'l'; s[-2] = 'o'; s[-1] = 'g';
            }
        }
```

```c
#ifdef WITHINTERACTION
    else if ( AM.Interact ) {
        if ( AM.LogType >= 0 ) {
/*
            We may have to do better than just taking a name.
            It is not unique! This will be left for later.
*/
            AM.LogFileName = deflogname;
        }
    }
#endif
    else {
NoFile:
#ifdef WITHMPI
        if ( PF.me == MASTER )
#endif
        printf("No filename specified in call of FORM\n");
        errorflag++;
    }
    if ( AM.Path == 0 ) AM.Path = (UBYTE *)getenv("FORMPATH");
    if ( AM.Path ) {
        /*
         * AM.Path is taken from argv or getenv. Reallocate it to avoid invalid
         * frees when AM.Path has to be changed.
         */
        AM.Path = strDup1(AM.Path,"DoTail Path");
    }
    return(errorflag);
}

/*
        #] DoTail :
        #[ OpenInput :

        Major task here after opening is to skip the proper number of
        .clear instructions if so desired without using interpretation
*/

int OpenInput()
{
    int oldNoShowInput = AC.NoShowInput;
    UBYTE c;
    if ( !AM.Interact ) {
        if ( OpenStream(AM.InputFileName,FILESTREAM,0,PRENOACTION) == 0 ) {
            Error1("Cannot open file",AM.InputFileName);
            return(-1);
        }
        if ( AC.CurrentStream->inbuffer <= 0 ) {
            Error1("No input in file",AM.InputFileName);
            return(-1);
        }
        AC.NoShowInput = 1;
        while ( AM.SkipClears > 0 ) {
            c = GetInput();
            if ( c == ENDOFINPUT ) {
                Error0("Not enough .clear instructions in input file");
            }
            if ( c == '\\' ) {
                c = GetInput();
                if ( c == ENDOFINPUT )
                    Error0("Not enough .clear instructions in input file");
                continue;
            }
            if ( c == ' ' || c == '\t' ) continue;
            if ( c == '.' ) {
                c = GetInput();
                if ( tolower(c) == 'c' ) {
                    c = GetInput();
                    if ( tolower(c) == 'l' ) {
                        c = GetInput();
                        if ( tolower(c) == 'e' ) {
                            c = GetInput();
                            if ( tolower(c) == 'a' ) {
                                c = GetInput();
                                if ( tolower(c) == 'r' ) {
                                    c = GetInput();
                                    if ( FG.cTable[c] > 2 ) {
                                        AM.SkipClears--;
                                    }
                                }
                            }
                        }
                    }
                }
                while ( c != '\n' && c != '\r' && c != ENDOFINPUT ) {
                    c = GetInput();
                    if ( c == '\\' ) c = GetInput();
```

```
                }
            }
            else if ( c == '\n' || c == '\r' ) continue;
            else {
                while ( ( c = GetInput() ) != '\n' && c != '\r' ) {
                    if ( c == ENDOFINPUT ) {
                        Error0("Not enough .clear instructions in input file");
                    }
                }
            }
        }
        AC.NoShowInput = oldNoShowInput;
    }
    if ( AM.LogFileName ) {
#ifdef WITHMPI
        if ( PF.me != MASTER ) {
            /*
             * Only the master writes to the log file. On slaves, we need
             * a dummy handle, without opening the file.
             */
            extern FILES **filelist;  /* in tools.c */
            int i = CreateHandle();
            RWLOCKW(AM.handlelock);
            filelist[i] = (FILES *)123;  /* Must be nonzero to prevent a reuse in CreateHandle. */
            UNRWLOCK(AM.handlelock);
            AC.LogHandle = i;
        }
        else
#endif
        if ( AC.CheckpointFlag != -1 ) {
            if ( ( AC.LogHandle = CreateLogFile((char *)(AM.LogFileName)) ) < 0 ) {
                Error1("Cannot create logfile",AM.LogFileName);
                return(-1);
            }
        }
        else {
            if ( ( AC.LogHandle = OpenAddFile((char *)(AM.LogFileName)) ) < 0 ) {
                Error1("Cannot re-open logfile",AM.LogFileName);
                return(-1);
            }
        }
    }
    return(0);
}

/*
        #] OpenInput :
        #[ ReserveTempFiles :

        Order of preference:
        a: if there is a path in the commandtail, take that.
        b: if none, try in the form.set file.
        c: if still none, try in the environment for the variable FORMTMP
        d: if still none, try the current directory.

        The parameter indicates action in the case of multithreaded running.
        par = 0 : We just run on a single processor. Keep everything normal.
        par = 1 : Multithreaded running startup phase 1.
        par = 2 : Multithreaded running startup phase 2.
*/

UBYTE *emptystring = (UBYTE *)".";
UBYTE *defaulttempfilename = (UBYTE *)"xformxxx.str";

VOID ReserveTempFiles(int par)
{
    GETIDENTITY
    SETUPPARAMETERS *sp;
    UBYTE *s, *t, *tenddir, *tenddir2, c;
    int i = 0;
    WORD j;
    if ( par == 0 || par == 1 ) {
    if ( AM.TempDir == 0 ) {
        sp = GetSetupPar((UBYTE *)"tempdir");
        if ( ( sp->flags & USEDFLAG ) != USEDFLAG ) {
            AM.TempDir = (UBYTE *)getenv("FORMTMP");
            if ( AM.TempDir == 0 ) AM.TempDir = emptystring;
        }
        else AM.TempDir = (UBYTE *)(sp->value);
    }
    if ( AM.TempSortDir == 0 ) {
        if ( AM.havesortdir ) {
            sp = GetSetupPar((UBYTE *)"tempsortdir");
            AM.TempSortDir = (UBYTE *)(sp->value);
        }
        else {
            AM.TempSortDir = (UBYTE *)getenv("FORMTMPSORT");
```

```
            if ( AM.TempSortDir == 0 ) AM.TempSortDir = AM.TempDir;
        }
    }
/*
    We have now in principle a path but we will use its first element only.
    Later that should become more complicated. Then we will use a path and
    when one device is full we can continue on the next one.
*/
    s = AM.TempDir; i = 200;   /* Some extra for VMS */
    while ( *s && *s != ':' ) { if ( *s == '\\' ) s++; s++; i++; }

    FG.fname = (char *)Malloc1(sizeof(UBYTE)*(i+14),"name for temporary files");
    s = AM.TempDir; t = (UBYTE *)FG.fname;
    while ( *s && *s != ':' ) { if ( *s == '\\' ) s++; *t++ = *s++; }
    if ( (char *)t > FG.fname && t[-1] != SEPARATOR && t[-1] != ALTSEPARATOR )
        *t++ = SEPARATOR;
    *t = 0;
    tenddir = t;
    FG.fnamebase = t-(UBYTE *)(FG.fname);

    s = AM.TempSortDir; i = 200;   /* Some extra for VMS */
    while ( *s && *s != ':' ) { if ( *s == '\\' ) s++; s++; i++; }

    FG.fname2 = (char *)Malloc1(sizeof(UBYTE)*(i+14),"name for sort files");
    s = AM.TempSortDir; t = (UBYTE *)FG.fname2;
    while ( *s && *s != ':' ) { if ( *s == '\\' ) s++; *t++ = *s++; }
    if ( (char *)t > FG.fname2 && t[-1] != SEPARATOR && t[-1] != ALTSEPARATOR )
        *t++ = SEPARATOR;
    *t = 0;
    tenddir2 = t;
    FG.fname2base = t-(UBYTE *)(FG.fname2);

    t = tenddir;
    s = defaulttempfilename;
#ifdef WITHMPI
    {
      int iii;
#ifdef SMP
      /* Very dirty quick-hack for the qcm smp machine at TTP */
      M_free(FG.fname,"name for temporary files");
      if(PF.me == 0){
      /*[04nov2003 mt] To avoid segfault with -fast optimization option*/
        /*[04nov2003 mt]:*/ /*NOTE, this is only a temporary stub!*/
        /*FG.fname = "/formswap/xxxxxxxxxxxxxxxxxxxx";*/
        FG.fname = calloc(128,1);
        strcpy(FG.fname,"/formswap/xxxxxxxxxxxxxxxxxxxx");
        /*:[04nov2003 mt]*/
        t = (UBYTE *)FG.fname + 10;
        FG.fnamebase = t-FG.fname;
      }
      else{
        /*[04nov2003 mt]:*/
        /*FG.fname = "/formswapx/xxxxxxxxxxxxxxxxxxxx";*/
        FG.fname = calloc(128,1);
        strcpy(FG.fname,"/formswapx/xxxxxxxxxxxxxxxxxxxx");
        /*:[04nov2003 mt]*/
        FG.fname[9] = '0' + PF.me;
        t = (UBYTE *)FG.fname + 11;
        FG.fnamebase = t-FG.fname;
      }
#else
      iii = sprintf((char*)t,"%d",PF.me);
      t+= iii;
      s+= iii; /* in case defaulttmpfilename is too short */
#endif
    }
#endif
    while ( *s ) *t++ = *s++;
    *t = 0;
/*
        There are problems when running many FORM jobs at the same time
        from make or minos. If they start up simultaneously, occasionally
        they can make the same .str file. We prevent this with first trying
        a file that contains the digits of the pid. If this file
        has already been taken we fall back on the old scheme.
        The whole is controled with the -M (MultiRun) parameter in the
        command tail.
*/
    if ( AM.MultiRun ) {
        int num = ((int)GetPID())%100000;
        t += 2;
        *t = 0;
        t[-1] = 'r';
        t[-2] = 't';
        t[-3] = 's';
        t[-4] = '.';
        t[-5] = (UBYTE)('0' + num%10);
```

```
        t[-6] = (UBYTE)('0' + (num/10)%10);
        t[-7] = (UBYTE)('0' + (num/100)%10);
        t[-8] = (UBYTE)('0' + (num/1000)%10);
        t[-9] = (UBYTE)('0' + num/10000);
        if ( ( AC.StoreHandle = CreateFile((char *)FG.fname) ) < 0 ) {
            t[-5] = 'x'; t[-6] = 'x'; t[-7] = 'x'; t[-8] = 'x'; t[-9] = 'x';
            goto classic;
        }
    }
    else
    {
classic:;
      for(;;) {
        if ( ( AC.StoreHandle = OpenFile((char *)FG.fname) ) < 0 ) {
            if ( ( AC.StoreHandle = CreateFile((char *)FG.fname) ) >= 0 ) break;
        }
        else CloseFile(AC.StoreHandle);
        c = t[-5];
        if ( c == 'x' ) t[-5] = '0';
        else if ( c == '9' ) {
            t[-5] = '0';
            c = t[-6];
            if ( c == 'x' ) t[-6] = '0';
            else if ( c == '9' ) {
                t[-6] = '0';
                c = t[-7];
                if ( c == 'x' ) t[-7] = '0';
                else if ( c == '9' ) {
/*
                    Note that we tried 1111 names!
*/
                    MesPrint("Name space for temp files exhausted");
                    t[-7] = 0;
                    MesPrint("Please remove files of the type %s or try a different directory"
                        ,FG.fname);
                    Terminate(-1);
                }
                else t[-7] = (UBYTE)(c+1);
            }
            else t[-6] = (UBYTE)(c+1);
        }
        else t[-5] = (UBYTE)(c+1);
      }
    }
/*
    Now we should make sure that the tempsortdir cq tempsortfilename makes it
    into a similar construction.
*/
    s = tenddir; t = tenddir2; while ( *s ) *t++ = *s++;
    *t = 0;

/*
    Now we should asign a name to the main sort file and the two stage 4 files.
*/
    AM.S0->file.name = (char *)Malloc1(sizeof(char)*(i+14),"name for temporary files");
    s = (UBYTE *)AM.S0->file.name;
    t = (UBYTE *)FG.fname2;
    i = 1;
    while ( *t ) { *s++ = *t++; i++; }
    s[-2] = 'o'; *s = 0;
    }
/*
    With the stage4 and scratch file names we have to be a bit more careful.
    They are to be allocated after the threads are initialized when there
    are threads of course.
*/
    if ( par == 0 ) {
        s = (UBYTE *)((void *)(FG.fname2)); i = 0;
        while ( *s ) { s++; i++; }
        s = (UBYTE *)Malloc1(sizeof(char)*(i+1),"name for stage4 file a");
        AR.FoStage4[1].name = (char *)s;
        t = (UBYTE *)FG.fname2;
        while ( *t ) *s++ = *t++;
        s[-2] = '4'; s[-1] = 'a'; *s = 0;
        s = (UBYTE *)((void *)(FG.fname)); i = 0;
        while ( *s ) { s++; i++; }
        s = (UBYTE *)Malloc1(sizeof(char)*(i+1),"name for stage4 file b");
        AR.FoStage4[0].name = (char *)s;
        t = (UBYTE *)FG.fname;
        while ( *t ) *s++ = *t++;
        s[-2] = '4'; s[-1] = 'b'; *s = 0;
        for ( j = 0; j < 3; j++ ) {
            s = (UBYTE *)Malloc1(sizeof(char)*(i+1),"name for scratch file");
            AR.Fscr[j].name = (char *)s;
            t = (UBYTE *)FG.fname;
            while ( *t ) *s++ = *t++;
            s[-2] = 'c'; s[-1] = (UBYTE)('0'+j); *s = 0;
```

```
        }
    }
#ifdef WITHPTHREADS
    else if ( par == 2 ) {
        s = (UBYTE *)((void *)(FG.fname2)); i = 0;
        while ( *s ) { s++; i++; }
        s = (UBYTE *)Malloc1(sizeof(char)*(i+12),"name for stage4 file a");
        sprintf((char *)s,"%s.%d",FG.fname2,AT.identity);
        s[i-2] = '4'; s[i-1] = 'a';
        AR.FoStage4[1].name = (char *)s;
        s = (UBYTE *)((void *)(FG.fname)); i = 0;
        while ( *s ) { s++; i++; }
        s = (UBYTE *)Malloc1(sizeof(char)*(i+12),"name for stage4 file b");
        sprintf((char *)s,"%s.%d",FG.fname,AT.identity);
        s[i-2] = '4'; s[i-1] = 'b';
        AR.FoStage4[0].name = (char *)s;
        if ( AT.identity == 0 ) {
            for ( j = 0; j < 3; j++ ) {
                s = (UBYTE *)Malloc1(sizeof(char)*(i+1),"name for scratch file");
                AR.Fscr[j].name = (char *)s;
                t = (UBYTE *)FG.fname;
                while ( *t ) *s++ = *t++;
                s[-2] = 'c'; s[-1] = (UBYTE)('0'+j); *s = 0;
            }
        }
    }
#endif
}

/*
        #] ReserveTempFiles :
        #[ StartVariables :
*/

#ifdef WITHPTHREADS
ALLPRIVATES *DummyPointer = 0;
#endif

VOID StartVariables()
{
    int i, ii;
    PUTZERO(AM.zeropos);
    StartPrepro();
/*
    The module counter:
*/
    AC.CModule=0;
#ifdef WITHPTHREADS
/*
    We need a value in AB because in the startup some routines may call AB[0].
*/
    AB = (ALLPRIVATES **)&DummyPointer;
#endif
/*
    separators used to delimit arguments in #call and #do, by default ',' and '|':
    Be sure, it is en empty set:
*/
    set_sub(AC.separators,AC.separators,AC.separators);
    set_set(',',AC.separators);
    set_set('|',AC.separators);

    AM.BracketFactors[0] = 8;
    AM.BracketFactors[1] = SYMBOL;
    AM.BracketFactors[2] = 4;
    AM.BracketFactors[3] = FACTORSYMBOL;
    AM.BracketFactors[4] = 1;
    AM.BracketFactors[5] = 1;
    AM.BracketFactors[6] = 1;
    AM.BracketFactors[7] = 3;

    AM.SkipClears = 0;
    AC.Cnumpows = 0;
    AC.OutputMode = 72;
    AC.OutputSpaces = NORMALFORMAT;
    AC.LineLength = 79;
    AM.gIsFortran90 = AC.IsFortran90 = ISNOTFORTRAN90;
    AM.gFortran90Kind = AC.Fortran90Kind = 0;
    AM.gCnumpows = 0;
    AC.exprfillwarning = 0;
    AM.gLineLength = 79;
    AM.OutBufSize = 80;
    AM.MaxStreamSize = MAXFILESTREAMSIZE;
    AP.MaxPreAssignLevel = 4;
    AC.iBufferSize = 512;
    AP.pSize = 128;
    AP.MaxPreIfLevel = 10;
    AP.cComChar = AP.ComChar = '*';
```

```
    AM.OffsetVector = -2*WILDOFFSET+MINSPEC;
    AC.cbufList.num = 0;
    AM.hparallelflag = AM.gparallelflag =
    AC.parallelflag = AC.mparallelflag = PARALLELFLAG;
#ifdef WITHMPI
    if ( PF.numtasks < 2 ) AM.hparallelflag |= NOPARALLEL_NPROC;
#endif
    AC.tablefilling = 0;
    AM.resetTimeOnClear = 1;
    AM.gnumextrasym = AM.ggnumextrasym = 0;
    AM.havesortdir = 0;
    AM.SpectatorFiles = 0;
    AM.NumSpectatorFiles = 0;
    AM.SizeForSpectatorFiles = 0;
/*
    Information for the lists of variables. Part of error message and size:
*/
    AP.ProcList.message = "procedure";
    AP.ProcList.size = sizeof(PROCEDURE);
    AP.LoopList.message = "doloop";
    AP.LoopList.size = sizeof(DOLOOP);
    AP.PreVarList.message = "PreVariable";
    AP.PreVarList.size = sizeof(PREVAR);
    AC.SymbolList.message = "symbol";
    AC.SymbolList.size = sizeof(struct SyMbOl);
    AC.IndexList.message = "index";
    AC.IndexList.size = sizeof(struct InDeX);
    AC.VectorList.message = "vector";
    AC.VectorList.size = sizeof(struct VeCtOr);
    AC.FunctionList.message = "function";
    AC.FunctionList.size = sizeof(struct FuNcTiOn);
    AC.SetList.message = "set";
    AC.SetList.size = sizeof(struct SeTs);
    AC.SetElementList.message = "set element";
    AC.SetElementList.size = sizeof(WORD);
    AC.ExpressionList.message = "expression";
    AC.ExpressionList.size = sizeof(struct ExPrEsSiOn);
    AC.cbufList.message = "compiler buffer";
    AC.cbufList.size = sizeof(CBUF);
    AC.ChannelList.message = "channel buffer";
    AC.ChannelList.size = sizeof(CHANNEL);
    AP.DollarList.message = "$-variable";
    AP.DollarList.size = sizeof(struct DoLlArS);
    AC.DubiousList.message = "ambiguous variable";
    AC.DubiousList.size = sizeof(struct DuBiOuS);
    AC.TableBaseList.message = "list of tablebases";
    AC.TableBaseList.size = sizeof(DBASE);
    AC.TestValue = 0;
    AC.InnerTest = 0;

    AC.AutoSymbolList.message = "autosymbol";
    AC.AutoSymbolList.size = sizeof(struct SyMbOl);
    AC.AutoIndexList.message = "autoindex";
    AC.AutoIndexList.size = sizeof(struct InDeX);
    AC.AutoVectorList.message = "autovector";
    AC.AutoVectorList.size = sizeof(struct VeCtOr);
    AC.AutoFunctionList.message = "autofunction";
    AC.AutoFunctionList.size = sizeof(struct FuNcTiOn);
    AC.PotModDolList.message = "potentially modified dollar";
    AC.PotModDolList.size = sizeof(WORD);
    AC.ModOptDolList.message = "moduleoptiondollar";
    AC.ModOptDolList.size = sizeof(MODOPTDOLLAR);

    AO.FortDotChar = '_';
    AO.ErrorBlock = 0;
    AC.firstconstindex = 1;
    AO.Optimize.mctsconstant.fval = 1.0;
    AO.Optimize.horner = O_MCTS;
    AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
    AO.Optimize.method = O_GREEDY;
    AO.Optimize.mctstimelimit = 0;
    AO.Optimize.mctsnumexpand = 1000;
    AO.Optimize.mctsnumkeep = 10;
    AO.Optimize.mctsnumrepeat = 1;
    AO.Optimize.greedytimelimit = 0;
    AO.Optimize.greedyminnum = 10;
    AO.Optimize.greedymaxperc = 5;
    AO.Optimize.printstats = 0;
    AO.Optimize.debugflags = 0;
    AO.OptimizeResult.code = NULL;
    AO.inscheme = 0;
    AO.schemenum = 0;
    AO.wpos = 0;
    AO.wpoin = 0;
    AO.wlen = 0;
    AM.dollarzero = 0;
    AC.doloopstack = 0;
```

```
    AC.doloopstacksize = 0;
    AC.doloopplevel = 0;
/*
    Set up the main name trees:
*/
    AC.varnames  = MakeNameTree();
    AC.exprnames = MakeNameTree();
    AC.dollarnames = MakeNameTree();
    AC.autonames = MakeNameTree();
    AC.activenames = &(AC.varnames);
    AP.preError = 0;
/*
    Initialize the compiler:
*/
    inictable();
    AM.rbufnum = inicbufs();        /* Regular compiler buffer */
#ifndef WITHPTHREADS
    AT.ebufnum = inicbufs();        /* Buffer for extras during execution */
    AT.fbufnum = inicbufs();        /* Buffer for caching in factorization */
    AT.allbufnum = inicbufs();      /* Buffer for id,all */
    AT.aebufnum = inicbufs();       /* Buffer for id,all */
    AN.tryterm = 0;
#else
    AS.MasterSort = 0;
#endif
    AM.dbufnum = inicbufs();        /* Buffer for dollar variables */
    AM.sbufnum = inicbufs();        /* Subterm buffer for polynomials and optimization */
    AC.ffbufnum = inicbufs();       /* Buffer number for user defined factorizations */
    AM.zbufnum = inicbufs();        /* For very special values */
    {
        CBUF *C = cbuf+AM.zbufnum;
        WORD one[5] = {4,1,1,3,0};
        WORD zero = 0;
        AddRHS(AM.zbufnum,1);
        AM.zerorhs = C->numrhs;
        AddNtoC(AM.zbufnum,1,&zero,17);
        AddRHS(AM.zbufnum,1);
        AM.onerhs = C->numrhs;
        AddNtoC(AM.zbufnum,5,one,17);
    }
    AP.inside.inscbuf = inicbufs(); /* For the #inside instruction */
/*
    Enter the built in objects
*/
    AC.Symbols = &(AC.SymbolList);
    AC.Indices = &(AC.IndexList);
    AC.Vectors = &(AC.VectorList);
    AC.Functions = &(AC.FunctionList);
    AC.vetofilling = 0;

    AddDollar((UBYTE *)"$",DOLUNDEFINED,0,0);

    cbuf[AM.dbufnum].mnumlhs = cbuf[AM.dbufnum].numlhs;
    cbuf[AM.dbufnum].mnumrhs = cbuf[AM.dbufnum].numrhs;

    AddSymbol((UBYTE *)"i_",-MAXPOWER,MAXPOWER,VARTYPEIMAGINARY,0);
    AddSymbol((UBYTE *)"pi_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
    AddSymbol((UBYTE *)"coeff_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
    AddSymbol((UBYTE *)"num_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
    AddSymbol((UBYTE *)"den_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
    AddSymbol((UBYTE *)"xarg_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
    AddSymbol((UBYTE *)"dimension_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
    AddSymbol((UBYTE *)"factor_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
    AddSymbol((UBYTE *)"sep_",-MAXPOWER,MAXPOWER,VARTYPENONE,0);
    i = BUILTINSYMBOLS;  /* update this in ftypes.h when we add new symbols */
/*
    Next we add a number of dummy symbols for ensuring that the user defined
    symbols start at a fixed given number FIRSTUSERSYMBOL
    We do want to give them unique names though that the user cannot access.
*/
    {
        char dumstr[10];
        for ( ; i < FIRSTUSERSYMBOL; i++ ) {
            sprintf(dumstr,":%d:",i);
            AddSymbol((UBYTE *)dumstr,-MAXPOWER,MAXPOWER,VARTYPENONE,0);
        }
    }

    AddIndex((UBYTE *)"iarg_",4,0);
    AddVector((UBYTE *)"parg_",VARTYPENONE,0);

    AM.NumFixedFunctions = sizeof(fixedfunctions)/sizeof(struct fixedfun);
    for ( i = 0; i < AM.NumFixedFunctions; i++ ) {
        ii = AddFunction((UBYTE *)fixedfunctions[i].name
                                ,fixedfunctions[i].commu
                                ,fixedfunctions[i].tensor
                                ,fixedfunctions[i].complx
```

```
                                ,fixedfunctions[i].symmetric
                                ,0,-1,-1);
        if ( fixedfunctions[i].tensor == GAMMAFUNCTION )
                        functions[ii].flags |= COULDCOMMUTE;
    }
/*
    Next we add a number of dummy functions for ensuring that the user defined
    functions start at a fixed given number FIRSTUSERFUNCTION.
    We do want to give them unique names though that the user cannot access.
*/
    {
        char dumstr[10];
        for ( ; i < FIRSTUSERFUNCTION-FUNCTION; i++ ) {
            sprintf(dumstr,"::%d::",i);
            AddFunction((UBYTE *)dumstr,0,0,0,0,0,-1,-1);
        }
    }
    AM.NumFixedSets = sizeof(fixedsets)/sizeof(struct fixedset);
    for ( i = 0; i < AM.NumFixedSets; i++ ) {
        ii = AddSet((UBYTE *)fixedsets[i].name,fixedsets[i].dimension);
        Sets[ii].type = fixedsets[i].type;
    }
    AM.RepMax = MAXREPEAT;
#ifndef WITHPTHREADS
    AT.RepCount = (int *)Malloc1((LONG)((AM.RepMax+3)*sizeof(int)),"repeat buffers");
    AN.RepPoint = AT.RepCount;
    AT.RepTop = AT.RepCount + AM.RepMax;
    AN.polysortflag = 0;
    AN.subsubveto = 0;
#endif
    AC.NumWildcardNames = 0;
    AC.WildcardBufferSize = 50;
    AC.WildcardNames = (UBYTE *)Malloc1((LONG)AC.WildcardBufferSize,"argument list names");
#ifndef WITHPTHREADS
    AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
                ,"argument list names");
    AT.WildcardBufferSize = AC.WildcardBufferSize;
    AR.CompareRoutine = &Compare1;
    AT.nfac = AT.nBer = 0;
    AT.factorials = 0;
    AT.bernoullis = 0;
    AR.wranfia = 0;
    AR.wranfcall = 0;
    AR.wranfnpair1 = NPAIR1;
    AR.wranfnpair2 = NPAIR2;
    AR.wranfseed = 0;
#endif
    AM.atstartup = 1;
    AM.oldnumextrasymbols = strDup1((UBYTE *)"OLDNUMEXTRASYMBOLS_","oldnumextrasymbols");
    PutPreVar((UBYTE *)"VERSION_",(UBYTE *)STRINGIFY(MAJORVERSION),0,0);
    PutPreVar((UBYTE *)"SUBVERSION_",(UBYTE *)STRINGIFY(MINORVERSION),0,0);
    PutPreVar((UBYTE *)"DATE_",(UBYTE *)MakeDate(),0,0);
    PutPreVar((UBYTE *)"random_",(UBYTE *)"________",(UBYTE *)"?a",0);
    PutPreVar((UBYTE *)"optimminvar_",(UBYTE *)("0"),0,0);
    PutPreVar((UBYTE *)"optimmaxvar_",(UBYTE *)("0"),0,0);
    PutPreVar(AM.oldnumextrasymbols,(UBYTE *)("0"),0,0);
    PutPreVar((UBYTE *)"optimvalue_",(UBYTE *)("0"),0,0);
    PutPreVar((UBYTE *)"optimscheme_",(UBYTE *)("0"),0,0);
    PutPreVar((UBYTE *)"tolower_",(UBYTE *)("0"),(UBYTE *)("?a"),0);
    PutPreVar((UBYTE *)"toupper_",(UBYTE *)("0"),(UBYTE *)("?a"),0);
    {
        char buf[41];  /* up to 128-bit */
        LONG pid;
#ifndef WITHMPI
        pid = GetPID();
#else
        pid = ( PF.me == MASTER ) ? GetPID() : (LONG)0;
        pid = PF_BroadcastNumber(pid);
#endif
        LongCopy(pid,buf);
        PutPreVar((UBYTE *)"PID_",(UBYTE *)buf,0,0);
    }
    AM.atstartup = 0;
    AP.MaxPreTypes = 10;
    AP.NumPreTypes = 0;
    AP.PreTypes = (int *)Malloc1(sizeof(int)*(AP.MaxPreTypes+1),"preprocessor types");
    AP.inside.buffer = 0;
    AP.inside.size = 0;

    AC.SortType = AC.lSortType = AM.gSortType = SORTLOWFIRST;
#ifdef WITHPTHREADS
#else
    AR.SortType = AC.SortType;
#endif
    AC.LogHandle = -1;
    AC.SetList.numtemp        = AC.SetList.num;
    AC.SetElementList.numtemp = AC.SetElementList.num;
```

```c

    GetName(AC.varnames,(UBYTE *)"exp_",&AM.expnum,NOAUTO);
    GetName(AC.varnames,(UBYTE *)"denom_",&AM.denomnum,NOAUTO);
    GetName(AC.varnames,(UBYTE *)"fac_",&AM.facnum,NOAUTO);
    GetName(AC.varnames,(UBYTE *)"invfac_",&AM.invfacnum,NOAUTO);
    GetName(AC.varnames,(UBYTE *)"sum_",&AM.sumnum,NOAUTO);
    GetName(AC.varnames,(UBYTE *)"sump_",&AM.sumpnum,NOAUTO);
    GetName(AC.varnames,(UBYTE *)"term_",&AM.termfunnum,NOAUTO);
    GetName(AC.varnames,(UBYTE *)"match_",&AM.matchfunnum,NOAUTO);
    GetName(AC.varnames,(UBYTE *)"count_",&AM.countfunnum,NOAUTO);
    AM.termfunnum += FUNCTION;
    AM.matchfunnum += FUNCTION;
    AM.countfunnum += FUNCTION;

    AC.ThreadStats = AM.gThreadStats = AM.ggThreadStats = 1;
    AC.FinalStats = AM.gFinalStats = AM.ggFinalStats = 1;
    AC.StatsFlag = AM.gStatsFlag = AM.ggStatsFlag = 1;
    AC.ThreadsFlag = AM.gThreadsFlag = AM.ggThreadsFlag = 1;
    AC.ThreadBalancing = AM.gThreadBalancing = AM.ggThreadBalancing = 1;
    AC.ThreadSortFileSynch = AM.gThreadSortFileSynch = AM.ggThreadSortFileSynch = 0;
    AC.ProcessStats = AM.gProcessStats = AM.ggProcessStats = 1;
    AC.OldParallelStats = AM.gOldParallelStats = AM.ggOldParallelStats = 0;
    AC.OldFactArgFlag = AM.gOldFactArgFlag = AM.ggOldFactArgFlag = NEWFACTARG;
    AC.OldGCDflag = AM.gOldGCDflag = AM.ggOldGCDflag = 1;
    AC.WTimeStatsFlag = AM.gWTimeStatsFlag = AM.ggWTimeStatsFlag = 0;
    AM.gcNumDollars = AP.DollarList.num;
    AC.SizeCommuteInSet = AM.gSizeCommuteInSet = 0;
    AC.CommuteInSet = 0;

    AM.PrintTotalSize = 0;

    AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers = AM.ggNoSpacesInNumbers = 0;
    AO.IndentSpace = AM.gIndentSpace = AM.ggIndentSpace = INDENTSPACE;
    AO.BlockSpaces = 0;
    AO.OptimizationLevel = 0;
    PUTZERO(AS.MaxExprSize);
    PUTZERO(AC.StoreFileSize);

#ifdef WITHPTHREADS
    AC.inputnumbers = 0;
    AC.pfirstnum = 0;
    AC.numpfirstnum = AC.sizepfirstnum = 0;
#endif
    AC.MemDebugFlag = 1;

#ifdef WITHEXTERNALCHANNEL
    AX.currentExternalChannel=0;
    AX.killSignal=SIGKILL;
    AX.killWholeGroup=1;
    AX.daemonize=1;
    AX.currentPrompt=0;
    AX.timeout=1000;/*One second to initialize preset channels*/
    AX.shellname=strDup1((UBYTE *)"/bin/sh -c","external channel shellname");
    AX.stderrname=strDup1((UBYTE *)"/dev/null","external channel stderrname");
#endif
}

/*
        #] StartVariables :
        #[ StartMore :
*/

VOID StartMore()
{
#ifdef WITHEXTERNALCHANNEL
    /*If env.variable "FORM_PIPES" is defined, we have to initialize
        corresponding pre-set external channels, see file extcmd.c.*/
    /*This line must be after all setup settings: in future, timeout
        could be changed at setup.*/
    if(AX.timeout>=0)/*if AX.timeout<0, this was done by cmdline option -pipe*/
        initPresetExternalChannels((UBYTE*)getenv("FORM_PIPES"),AX.timeout);
#endif

#ifdef WITHMPI
/*
    Define preprocessor variable PARALLELTASK_ as a process number, 0 is the master
    Define preprocessor variable NPARALLELTASKS_ as a total number of processes
*/
    {
        UBYTE buf[32];
        sprintf((char*)buf,"%d",PF.me);
        PutPreVar((UBYTE *)"PARALLELTASK_",buf,0,0);
        sprintf((char*)buf,"%d",PF.numtasks);
        PutPreVar((UBYTE *)"NPARALLELTASKS_",buf,0,0);
    }
#else
    PutPreVar((UBYTE *)"PARALLELTASK_",(UBYTE *)"0",0,0);
```

```
    PutPreVar((UBYTE *)"NPARALLELTASKS_",(UBYTE *)"1",0,0);
#endif

    PutPreVar((UBYTE *)"NAME_",AM.InputFileName,0,0);
}

/*
        #] StartMore :
        #[ IniVars :

        This routine initializes the parameters that may change during the run.
*/

WORD IniVars()
{
#ifdef WITHPTHREADS
    GETIDENTITY
#else
    WORD *t;
#endif
    WORD *fi, i, one = 1;
    CBUF *C = cbuf+AC.cbufnum;

#ifdef WITHPTHREADS
    UBYTE buf[32];
    sprintf((char*)buf,"%d",AM.totalnumberofthreads);
    PutPreVar((UBYTE *)"NTHREADS_",buf,0,1);
#else
    PutPreVar((UBYTE *)"NTHREADS_",(UBYTE *)"1",0,1);
#endif

    AC.ShortStats = 0;
    AC.WarnFlag = 1;
    AR.SortType = AC.SortType = AC.lSortType = AM.gSortType;
    AC.OutputMode = 72;
    AC.OutputSpaces = NORMALFORMAT;
    AR.Eside = 0;
    AC.DumNum = 0;
    AC.ncmod = AM.gncmod = 0;
    AC.modmode = AM.gmodmode = 0;
    AC.npowmod = AM.gnpowmod = 0;
    AC.halfmod = 0; AC.nhalfmod = 0;
    AC.modinverses = 0;
    AC.lPolyFun = AM.gPolyFun = 0;
    AC.lPolyFunInv = AM.gPolyFunInv = 0;
    AC.lPolyFunType = AM.gPolyFunType = 0;
    AC.lPolyFunExp = AM.gPolyFunExp = 0;
    AC.lPolyFunVar = AM.gPolyFunVar = 0;
    AC.lPolyFunPow = AM.gPolyFunPow = 0;
    AC.DirtPow = 0;
    AC.lDefDim = AM.gDefDim = 4;
    AC.lDefDim4 = AM.gDefDim4 = 0;
    AC.lUnitTrace = AM.gUnitTrace = 4;
    AC.NamesFlag = AM.gNamesFlag = 0;
    AC.CodesFlag = AM.gCodesFlag = 0;
    AC.extrasymbols = AM.gextrasymbols = AM.ggextrasymbols = 0;
    AC.extrasym = (UBYTE *)Malloc1(2*sizeof(UBYTE),"extrasym");
    AM.gextrasym = (UBYTE *)Malloc1(2*sizeof(UBYTE),"extrasym");
    AM.ggextrasym = (UBYTE *)Malloc1(2*sizeof(UBYTE),"extrasym");
    AC.extrasym[0] = AM.gextrasym[0] = AM.ggextrasym[0] = 'Z';
    AC.extrasym[1] = AM.gextrasym[1] = AM.ggextrasym[1] = 0;
    AC.TokensWriteFlag = AM.gTokensWriteFlag = 0;
    AC.SetupFlag = 0;
    AC.LineLength = AM.gLineLength = 79;
    AC.NwildC = 0;
    AC.OutputMode = 0;
    AM.gOutputMode = 0;
    AC.OutputSpaces = NORMALFORMAT;
    AM.gOutputSpaces = NORMALFORMAT;
    AC.OutNumberType = RATIONALMODE;
    AM.gOutNumberType = RATIONALMODE;
#ifdef WITHZLIB
    AR.gzipCompress = GZIPDEFAULT;
#endif
    AR.BracketOn = 0;
    AC.bracketindexflag = 0;
    AT.bracketindexflag = 0;
    AT.bracketinfo = 0;
    AO.IsBracket = 0;
    AM.gfunpowers = AC.funpowers = COMFUNPOWERS;
    AC.parallelflag = AM.gparallelflag;
    AC.properorderflag = AM.gproperorderflag = PROPERORDERFLAG;
    AC.ProcessBucketSize = AC.mProcessBucketSize = AM.gProcessBucketSize;
    AC.ThreadBucketSize = AM.gThreadBucketSize;
    AC.ShortStatsMax = 0;
    AM.gShortStatsMax = 0;
    AM.ggShortStatsMax = 0;
```

```c

    GlobalSymbols     = NumSymbols;
    GlobalIndices     = NumIndices;
    GlobalVectors     = NumVectors;
    GlobalFunctions   = NumFunctions;
    GlobalSets        = NumSets;
    GlobalSetElements = NumSetElements;
    AC.modpowers = (UWORD *)0;

    i = AM.OffsetIndex;
    fi = AC.FixIndices;
    if ( i > 0 ) do { *fi++ = one; } while ( --i >= 0 );
    AR.sLevel = -1;
    AM.Ordering[0] = 5;
    AM.Ordering[1] = 6;
    AM.Ordering[2] = 7;
    AM.Ordering[3] = 0;
    AM.Ordering[4] = 1;
    AM.Ordering[5] = 2;
    AM.Ordering[6] = 3;
    AM.Ordering[7] = 4;
    for ( i = 8; i < 15; i++ ) AM.Ordering[i] = i;
    AM.gUniTrace[0] =
    AC.lUniTrace[0] = SNUMBER;
    AM.gUniTrace[1] =
    AC.lUniTrace[1] =
    AM.gUniTrace[2] =
    AC.lUniTrace[2] = 4;
    AM.gUniTrace[3] =
    AC.lUniTrace[3] = 1;
#ifdef WITHPTHREADS
    AS.Balancing = 0;
#else
    AT.MinVecArg[0] = 7+ARGHEAD;
    AT.MinVecArg[ARGHEAD] = 7;
    AT.MinVecArg[1+ARGHEAD] = INDEX;
    AT.MinVecArg[2+ARGHEAD] = 3;
    AT.MinVecArg[3+ARGHEAD] = 0;
    AT.MinVecArg[4+ARGHEAD] = 1;
    AT.MinVecArg[5+ARGHEAD] = 1;
    AT.MinVecArg[6+ARGHEAD] = -3;
    t = AT.FunArg;
    *t++ = 4+ARGHEAD+FUNHEAD;
    for ( i = 1; i < ARGHEAD; i++ ) *t++ = 0;
    *t++ = 4+FUNHEAD;
    *t++ = 0;
    *t++ = FUNHEAD;
    for ( i = 2; i < FUNHEAD; i++ ) *t++ = 0;
    *t++ = 1; *t++ = 1; *t++ = 3;

#ifdef WITHMPI
    AS.printflag = 0;
#endif

    AT.comsym[0] = 8;
    AT.comsym[1] = SYMBOL;
    AT.comsym[2] = 4;
    AT.comsym[3] = 0;
    AT.comsym[4] = 1;
    AT.comsym[5] = 1;
    AT.comsym[6] = 1;
    AT.comsym[7] = 3;
    AT.comnum[0] = 4;
    AT.comnum[1] = 1;
    AT.comnum[2] = 1;
    AT.comnum[3] = 3;
    AT.comfun[0] = FUNHEAD+4;
    AT.comfun[1] = FUNCTION;
    AT.comfun[2] = FUNHEAD;
    AT.comfun[3] = 0;
#if FUNHEAD == 4
    AT.comfun[4] = 0;
#endif
    AT.comfun[FUNHEAD+1] = 1;
    AT.comfun[FUNHEAD+2] = 1;
    AT.comfun[FUNHEAD+3] = 3;
    AT.comind[0] = 7;
    AT.comind[1] = INDEX;
    AT.comind[2] = 3;
    AT.comind[3] = 0;
    AT.comind[4] = 1;
    AT.comind[5] = 1;
    AT.comind[6] = 3;
    AT.locwildvalue[0] = SUBEXPRESSION;
    AT.locwildvalue[1] = SUBEXPSIZE;
    for ( i = 2; i < SUBEXPSIZE; i++ ) AT.locwildvalue[i] = 0;
    AT.mulpat[0] = TYPEMULT;
```

```
    AT.mulpat[1] = SUBEXPSIZE+3;
    AT.mulpat[2] = 0;
    AT.mulpat[3] = SUBEXPRESSION;
    AT.mulpat[4] = SUBEXPSIZE;
    AT.mulpat[5] = 0;
    AT.mulpat[6] = 1;
    for ( i = 7; i < SUBEXPSIZE+5; i++ ) AT.mulpat[i] = 0;
    AT.proexp[0] = SUBEXPSIZE+4;
    AT.proexp[1] = EXPRESSION;
    AT.proexp[2] = SUBEXPSIZE;
    AT.proexp[3] = -1;
    AT.proexp[4] = 1;
    for ( i = 5; i < SUBEXPSIZE+1; i++ ) AT.proexp[i] = 0;
    AT.proexp[SUBEXPSIZE+1] = 1;
    AT.proexp[SUBEXPSIZE+2] = 1;
    AT.proexp[SUBEXPSIZE+3] = 3;
    AT.proexp[SUBEXPSIZE+4] = 0;
    AT.dummysubexp[0] = SUBEXPRESSION;
    AT.dummysubexp[1] = SUBEXPSIZE+4;
    for ( i = 2; i < SUBEXPSIZE; i++ ) AT.dummysubexp[i] = 0;
    AT.dummysubexp[SUBEXPSIZE] = WILDDUMMY;
    AT.dummysubexp[SUBEXPSIZE+1] = 4;
    AT.dummysubexp[SUBEXPSIZE+2] = 0;
    AT.dummysubexp[SUBEXPSIZE+3] = 0;

    AT.inprimelist = -1;
    AT.sizeprimelist = 0;
    AT.primelist = 0;
    AT.LeaveNegative = 0;
    AT.TrimPower = 0;
    AN.SplitScratch = 0;
    AN.SplitScratchSize = AN.InScratch = 0;
    AN.SplitScratch1 = 0;
    AN.SplitScratchSize1 = AN.InScratch1 = 0;
    AN.idfunctionflag = 0;
#endif
    AO.OutputLine = AO.OutFill = BufferForOutput;
    AO.FactorMode = 0;
    C->Pointer = C->Buffer;

    AP.PreOut = 0;
    AP.ComChar = AP.cComChar;
    AC.cbufnum = AM.rbufnum;        /* Select the default compiler buffer */
    AC.HideLevel = 0;
    AP.PreAssignFlag = 0;
    return(0);
}

/*
        #] IniVars :
        #[ Signal handlers :
*/
/*[28apr2004 mt]:*/
#ifdef TRAPSIGNALS

static int exitInProgress = 0;
static int trappedTerminate = 0;

/*INTSIGHANDLER : some systems require a signal handler to return an integer,
  so define the macro INTSIGHANDLER if compiler fails:*/
#ifdef INTSIGHANDLER
static int onErrSig(int i)
#else
static VOID onErrSig(int i)
#endif
{
    if (exitInProgress){
        signal(i,SIG_DFL);/* Use default behaviour*/
        raise (i);/*reproduce trapped signal*/
#ifdef INTSIGHANDLER
        return(i);
#else
        return;
#endif
    }
    trappedTerminate = 1;
    /*[13jul2005 mt]*//*Terminate(-1) on signal is here:*/
    Terminate(-1);
}

#ifdef INTSIGHANDLER
static VOID setNewSig(int i, int (*handler)(int))
#else
static VOID setNewSig(int i, void (*handler)(int))
#endif
{
    if(! (i<NSIG) )/* Invalid signal -- see comments in the file */
```

```
        return;
    if ( signal(i,SIG_IGN) !=SIG_IGN)
    /* if compiler fails here, try to define INTSIGHANDLER):*/
        signal(i,handler);
}

VOID setSignalHandlers()
{
    /* Reset various unrecoverable error signals:*/
    setNewSig(SIGSEGV,onErrSig);
    setNewSig(SIGFPE,onErrSig);
    setNewSig(SIGILL,onErrSig);
    setNewSig(SIGEMT,onErrSig);
    setNewSig(SIGSYS,onErrSig);
    setNewSig(SIGPIPE,onErrSig);
    setNewSig(SIGLOST,onErrSig);
    setNewSig(SIGXCPU,onErrSig);
    setNewSig(SIGXFSZ,onErrSig);

    /* Reset interrupt signals:*/
    setNewSig(SIGTERM,onErrSig);
    setNewSig(SIGINT,onErrSig);
    setNewSig(SIGQUIT,onErrSig);
    setNewSig(SIGHUP,onErrSig);
    setNewSig(SIGALRM,onErrSig);
    setNewSig(SIGVTALRM,onErrSig);
/*  setNewSig(SIGPROF,onErrSig); */  /* Why did Tentukov forbid profilers?? */
}

#endif
/*:[28apr2004 mt]*/
/*
        #] Signal handlers :
        #[ main :
*/

#ifdef WITHPTHREADS
ALLPRIVATES *ABdummy[10];
#endif

int main(int argc, char **argv)
{
    int retval;
    bzero((VOID *)(&A),sizeof(A)); /* make sure A is initialized at zero */
    iniTools();
#ifdef TRAPSIGNALS
    setSignalHandlers();
#endif

#ifdef WITHPTHREADS
    AB = ABdummy;
    StartHandleLock();
    BeginIdentities();
#else
    AM.SumTime = TimeCPU(0);
    TimeWallClock(0);
#endif

#ifdef WITHMPI
    if ( PF_Init(&argc,&argv) ) exit(-1);
#endif

    StartFiles();
    StartVariables();
#ifdef WITHMPI
    /*
     * Here MesPrint() is ready. We turn on AS.printflag to print possible
     * errors occurring on slaves in the initialization. With AS.printflag = -1
     * MesPrint() does not use the synchronized output. This may lead broken
     * texts in the output somewhat, but it is safer to implement in this way
     * for the situation in which some of MesPrint() calls use MLOCK()-MUNLOCK()
     * and some do not. In future if we set AS.printflag = 1 and modify the
     * source code such that all MesPrint() calls are sandwiched by MLOCK()-
     * MUNLOCK(), we need also to modify the code for the master to catch
     * messages corresponding to MUNLOCK() calls at some point.
     *
     * AS.printflag will be set to 0 in IniVars() to prevent slaves from
     * printing redundant errors in the preprocessor and compiler (e.g., syntax
     * errors).
     */
    AS.printflag = -1;
#endif

    if ( ( retval = DoTail(argc,(UBYTE **)argv) ) != 0 ) {
        if ( retval > 0 ) Terminate(0);
        else              Terminate(-1);
    }
```

```
    if ( DoSetups() ) Terminate(-2);
#ifdef WITHMPI
    /* It is messy if all errors in OpenInput() on slaves are printed. */
    AS.printflag = 0;
#endif
    if ( OpenInput() ) Terminate(-3);
#ifdef WITHMPI
    AS.printflag = -1;
#endif
    if ( TryEnvironment() ) Terminate(-2);
    if ( TryFileSetups() ) Terminate(-2);
    if ( MakeSetupAllocs() ) Terminate(-2);
    StartMore();
    InitRecovery();
    CheckRecoveryFile();
    if ( AM.totalnumberofthreads == 0 ) AM.totalnumberofthreads = 1;
    AS.MultiThreaded = 0;
#ifdef WITHPTHREADS
    if ( AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
    ReserveTempFiles(1);
    StartAllThreads(AM.totalnumberofthreads);
    IniFbufs();
#else
    ReserveTempFiles(0);
    IniFbuffer(AT.fbufnum);
#endif
    PrintHeader(1);
    IniVars();
    Globalize(1);
    TimeCPU(0);
    TimeChildren(0);
    TimeWallClock(0);
    PreProcessor();
    Terminate(0);
    return(0);
}
/*
        #] main :
        #[ CleanUp :

        if par < 0 we have to keep the storage file.
        when par > 0 we ran into a .clear statement.
        In that case we keep the zero level input and the log file.

*/

VOID CleanUp(WORD par)
{
    GETIDENTITY
    int i;

    if ( FG.fname ) {
    CleanUpSort(0);
    for ( i = 0; i < 3; i++ ) {
        if ( AR.Fscr[i].handle >= 0 ) {
            if ( AR.Fscr[i].name ) {
/*
                If there are more threads referring to the same file
                only the one with the name is the owner of the file.
*/
                CloseFile(AR.Fscr[i].handle);
                remove(AR.Fscr[i].name);
            }
            AR.Fscr[i].handle = - 1;
            AR.Fscr[i].POfill = 0;
        }
    }
    if ( par > 0 ) {
/*
    Close all input levels above the lowest?
*/
    }
    if ( AC.StoreHandle >= 0 && par <= 0 ) {
#ifdef TRAPSIGNALS
        if ( trappedTerminate ) { /* We don't throw .str if it has contents */
            POSITION pos;
            PUTZERO(pos);
            SeekFile(AC.StoreHandle,&pos,SEEK_END);
            if ( ISNOTZEROPOS(pos) ) {
                CloseFile(AC.StoreHandle);
                goto dontremove;
            }
        }
        CloseFile(AC.StoreHandle);
        if ( par >= 0 || AR.StoreData.Handle < 0 ) {
            remove(FG.fname);
        }
```

```
dontremove:;
#else
        CloseFile(AC.StoreHandle);
        if ( par >= 0 || AR.StoreData.Handle < 0 ) {
            remove(FG.fname);
        }
#endif
    }
    }
    ClearSpectators(CLEARMODULE);
/*
    Remove recovery file on exit if everything went well
*/
    if ( par == 0 ) {
        DeleteRecoveryFile();
    }
/*
    Now the final message concerning the total time
*/
    if ( AC.LogHandle >= 0 && par <= 0 ) {
        WORD lh = AC.LogHandle;
        AC.LogHandle = -1;
#ifdef WITHMPI
        if ( PF.me == MASTER )  /* Only the master opened the real file. */
#endif
        CloseFile(lh);
    }
}

/*
        #] CleanUp :
        #[ Terminate :
*/

static int firstterminate = 1;

VOID Terminate(int errorcode)
{
    if ( errorcode && firstterminate ) {
        firstterminate = 0;
#ifdef WITHPTHREADS
        MesPrint("Program terminating in thread %w at &");
#elif defined(WITHMPI)
        MesPrint("Program terminating in process %w at &");
#else
        MesPrint("Program terminating at &");
#endif
        Crash();
    }
#ifdef TRAPSIGNALS
    exitInProgress=1;
#endif
#ifdef WITHEXTERNALCHANNEL
/*
    This function can be called from the error handler, so it is better to
    clean up all started processes before any activity:
*/
    closeAllExternalChannels();
    AX.currentExternalChannel=0;
    /*[08may2006 mt]:*/
    AX.killSignal=SIGKILL;
    AX.killWholeGroup=1;
    AX.daemonize=1;
    /*:[08may2006 mt]*/
    if(AX.currentPrompt){
        M_free(AX.currentPrompt,"external channel prompt");
        AX.currentPrompt=0;
    }
    /*[08may2006 mt]:*/
    if(AX.shellname){
        M_free(AX.shellname,"external channel shellname");
        AX.shellname=0;
    }
    if(AX.stderrname){
        M_free(AX.stderrname,"external channel stderrname");
        AX.stderrname=0;
    }
    /*:[08may2006 mt]*/
#endif
#ifdef WITHPTHREADS
    TerminateAllThreads();
#endif
    if ( AC.FinalStats ) {
        if ( AM.PrintTotalSize ) {
            MesPrint("Max. space for expressions: %19p bytes",&(AS.MaxExprSize));
        }
        PrintRunningTime();
```

```
    }
#ifdef WITHMPI
    if ( AM.HoldFlag && PF.me == MASTER ) {
        WriteFile(AM.StdOut,(UBYTE *)("Hit any key "),12);
        PF_FlushStdOutBuffer();
        getchar();
    }
#else
    if ( AM.HoldFlag ) {
        WriteFile(AM.StdOut,(UBYTE *)("Hit any key "),12);
        getchar();
    }
#endif
#ifdef WITHMPI
    PF_Terminate(errorcode);
#endif
    CleanUp(errorcode);
    M_print();
#ifdef VMS
    P_term(errorcode? 0: 1);
#else
    P_term(errorcode);
#endif
}

/*
        #] Terminate :
        #[ PrintRunningTime :
*/

VOID PrintRunningTime()
{
#if (defined(WITHPTHREADS) && (defined(WITHPOSIXCLOCK) || defined(WINDOWS))) || defined(WITHMPI)
    LONG mastertime;
    LONG workertime;
    LONG wallclocktime;
    LONG totaltime;
#if defined(WITHPTHREADS)
    if ( AB[0] != 0 ) {
        workertime = GetWorkerTimes();
#else
    workertime = PF_GetSlaveTimes();  /* must be called on all processors */
    if ( PF.me == MASTER ) {
#endif
        mastertime = AM.SumTime + TimeCPU(1);
        wallclocktime = TimeWallClock(1);
        totaltime = mastertime+workertime;
        if ( !AM.silent ) {
        MesPrint("  %l.%2i sec + %l.%2i sec: %l.%2i sec out of %l.%2i sec",
            mastertime/1000,(WORD)((mastertime%1000)/10),
            workertime/1000,(WORD)((workertime%1000)/10),
            totaltime/1000,(WORD)((totaltime%1000)/10),
            wallclocktime/100,(WORD)(wallclocktime%100));
        }
    }
#else
    LONG mastertime = AM.SumTime + TimeCPU(1);
    LONG wallclocktime = TimeWallClock(1);
    if ( !AM.silent ) {
    MesPrint("  %l.%2i sec out of %l.%2i sec",
        mastertime/1000,(WORD)((mastertime%1000)/10),
        wallclocktime/100,(WORD)(wallclocktime%100));
    }
#endif
}

/*
        #] PrintRunningTime :
        #[ GetRunningTime :
*/

LONG GetRunningTime()
{
#if defined(WITHPTHREADS) && (defined(WITHPOSIXCLOCK) || defined(WINDOWS))
    LONG mastertime;
    if ( AB[0] != 0 ) {
/*
#if ( defined(APPLE64) || defined(APPLE32) )
        mastertime = AM.SumTime + TimeCPU(1);
        return(mastertime);
#else
*/
        LONG workertime = GetWorkerTimes();
        mastertime = AM.SumTime + TimeCPU(1);
        return(mastertime+workertime);
/*
#endif
```

```
*/
    }
    else {
        return(AM.SumTime + TimeCPU(1));
    }
#elif defined(WITHMPI)
    LONG mastertime, t = 0;
    LONG workertime = PF_GetSlaveTimes();  /* must be called on all processors */
    if ( PF.me == MASTER ) {
        mastertime = AM.SumTime + TimeCPU(1);
        t = mastertime + workertime;
    }
    return PF_BroadcastNumber(t);  /* must be called on all processors */
#else
    return(AM.SumTime + TimeCPU(1));
#endif
}

/*
        #] GetRunningTime :
*/
```

4.106 store.c File Reference

`#include "form3.h"`

Include dependency graph for store.c:

Macros

- #define SAVEREVISION 0x02

Functions

- WORD OpenTemp ()
- VOID SeekScratch (FILEHANDLE *fi, POSITION *pos)
- VOID SetEndScratch (FILEHANDLE *f, POSITION *position)
- VOID SetEndHScratch (FILEHANDLE *f, POSITION *position)
- VOID SetScratch (FILEHANDLE *f, POSITION *position)
- WORD RevertScratch ()
- WORD ResetScratch ()
- int ReadFromScratch (FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length)
- int AddToScratch (FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length, int withflush)
- int CoSave (UBYTE *inp)
- int CoLoad (UBYTE *inp)
- WORD DeleteStore (WORD par)
- WORD PutInStore (INDEXENTRY *ind, WORD num)
- WORD GetTerm (PHEAD WORD *term)

- WORD GetOneTerm (PHEAD WORD *term, FILEHANDLE *fi, POSITION *pos, int par)
- WORD GetMoreTerms (WORD *term)
- WORD GetMoreFromMem (WORD *term, WORD **tpoin)
- WORD GetFromStore (WORD *to, POSITION *position, RENUMBER renumber, WORD *InCompState, WORD nexpr)
- VOID DetVars (WORD *term, WORD par)
- WORD ToStorage (EXPRESSIONS e, POSITION *length)
- INDEXENTRY * NextFileIndex (POSITION *indexpos)
- WORD SetFileIndex ()
- WORD VarStore (UBYTE *s, WORD n, WORD name, WORD namesize)
- WORD TermRenumber (WORD *term, RENUMBER renumber, WORD nexpr)
- WORD FindrNumber (WORD n, VARRENUM *v)
- INDEXENTRY * FindInIndex (WORD expr, FILEDATA *f, WORD par, WORD mode)
- RENUMBER GetTable (WORD expr, POSITION *position, WORD mode)
- int CopyExpression (FILEHANDLE *from, FILEHANDLE *to)
- WORD WriteStoreHeader (WORD handle)
- WORD ReadSaveHeader ()
- WORD ReadSaveIndex (FILEINDEX *fileind)
- WORD ReadSaveVariables (UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize, INDEXENTRY *ind, LONG *stage)
- UBYTE * ReadSaveTerm32 (UBYTE *bin, UBYTE *binend, UBYTE **bout, UBYTE *boutend, UBYTE *top, int terminbuf)
- WORD ReadSaveExpression (UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize)

4.106.1 Detailed Description

Contains all functions that deal with store-files and the system independent save-files.

Definition in file store.c.

4.106.2 Macro Definition Documentation

4.106.2.1 SAVEREVISION

```
#define SAVEREVISION 0x02
```

Definition at line 3915 of file store.c.

4.106.3 Function Documentation

4.106.3.1 OpenTemp()

```
WORD OpenTemp ( )
```

Definition at line 48 of file store.c.

4.106.3.2 SeekScratch()

```
VOID SeekScratch (
            FILEHANDLE * fi,
            POSITION * pos )
```

Definition at line 64 of file store.c.

4.106.3.3 SetEndScratch()

```
VOID SetEndScratch (
            FILEHANDLE * f,
            POSITION * position )
```

Definition at line 75 of file store.c.

4.106.3.4 SetEndHScratch()

```
VOID SetEndHScratch (
            FILEHANDLE * f,
            POSITION * position )
```

Definition at line 89 of file store.c.

4.106.3.5 SetScratch()

```
VOID SetScratch (
            FILEHANDLE * f,
            POSITION * position )
```

Definition at line 115 of file store.c.

4.106.3.6 RevertScratch()

```
WORD RevertScratch ( )
```

Definition at line 190 of file store.c.

4.106.3.7 ResetScratch()

```
WORD ResetScratch ( )
```

Definition at line 235 of file store.c.

4.106.3.8 ReadFromScratch()

```
int ReadFromScratch (
            FILEHANDLE * fi,
            POSITION * pos,
            UBYTE * buffer,
            POSITION * length )
```

Definition at line 273 of file store.c.

4.106.3.9 AddToScratch()

```
int AddToScratch (
            FILEHANDLE * fi,
            POSITION * pos,
            UBYTE * buffer,
            POSITION * length,
            int withflush )
```

Definition at line 300 of file store.c.

4.106.3.10 CoSave()

```
int CoSave (
            UBYTE * inp )
```

Definition at line 363 of file store.c.

4.106.3.11 CoLoad()

```
int CoLoad (
            UBYTE * inp )
```

Definition at line 561 of file store.c.

4.106.3.12 DeleteStore()

```
WORD DeleteStore (
            WORD par )
```

Definition at line 761 of file store.c.

4.106.3.13 PutInStore()

```
WORD PutInStore (
            INDEXENTRY * ind,
            WORD num )
```

Definition at line 861 of file store.c.

4.106.3.14 GetTerm()

```
WORD GetTerm (
            PHEAD WORD * term )
```

Definition at line 980 of file store.c.

4.106.3.15 GetOneTerm()

```
WORD GetOneTerm (
            PHEAD WORD * term,
            FILEHANDLE * fi,
            POSITION * pos,
            int par )
```

Definition at line 1255 of file store.c.

4.106.3.16 GetMoreTerms()

```
WORD GetMoreTerms (
            WORD * term )
```

Definition at line 1412 of file store.c.

4.106.3.17 GetMoreFromMem()

```
WORD GetMoreFromMem (
            WORD * term,
            WORD ** tpoin )
```

Definition at line 1508 of file store.c.

4.106.3.18 GetFromStore()

```
WORD GetFromStore (
            WORD * to,
            POSITION * position,
            RENUMBER renumber,
            WORD * InCompState,
            WORD nexpr )
```

Definition at line 1615 of file store.c.

4.106.3.19 DetVars()

```
VOID DetVars (
            WORD * term,
            WORD par )
```

Definition at line 1814 of file store.c.

4.106.3.20 ToStorage()

```
WORD ToStorage (
            EXPRESSIONS e,
            POSITION * length )
```

Definition at line 2001 of file store.c.

4.106.3.21 NextFileIndex()

```
INDEXENTRY * NextFileIndex (
            POSITION * indexpos )
```

Definition at line 2237 of file store.c.

4.106.3.22 SetFileIndex()

```
WORD SetFileIndex ( )
```

Reads the next file index and puts it into AR.StoreData.Index. TODO

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 2300 of file store.c.

References WriteStoreHeader().

Here is the call graph for this function:

4.106.3.23 VarStore()

```
WORD VarStore (
            UBYTE * s,
            WORD n,
            WORD name,
            WORD namesize )
```

Definition at line 2349 of file store.c.

4.106.3.24 TermRenumber()

```
WORD TermRenumber (
            WORD * term,
            RENUMBER renumber,
            WORD nexpr )
```

!! WORD ∗memterm=term; static LONG ctrap=0; !!!

!! ctrap++; !!!

Definition at line 2407 of file store.c.

References ReNuMbEr::func, ReNuMbEr::funnum, ReNuMbEr::indi, ReNuMbEr::indnum, ReNuMbEr::symb, ReNuMbEr::symnum, TermRenumber(), ReNuMbEr::vecnum, and ReNuMbEr::vect.

Referenced by TermRenumber().

Here is the call graph for this function:

4.106.3.25 FindrNumber()

```
WORD FindrNumber (
            WORD n,
            VARRENUM * v )
```

Definition at line 2567 of file store.c.

4.106.3.26 FindInIndex()

```
INDEXENTRY * FindInIndex (
            WORD expr,
            FILEDATA * f,
            WORD par,
            WORD mode )
```

Definition at line 2644 of file store.c.

4.106.3.27 GetTable()

```
RENUMBER GetTable (
            WORD expr,
            POSITION * position,
            WORD mode )
```

Definition at line 2805 of file store.c.

4.106.3.28 CopyExpression()

```
int CopyExpression (
            FILEHANDLE * from,
            FILEHANDLE * to )
```

Definition at line 3262 of file store.c.

4.106.3.29 WriteStoreHeader()

```
WORD WriteStoreHeader (
            WORD handle )
```

Writes header with information about system architecture and FORM revision to an open store file.

Called by SetFileIndex().

Parameters

handle	specifies open file to which header will be written

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 3926 of file store.c.

References STOREHEADER::endianness, STOREHEADER::lenLONG, STOREHEADER::lenPOINTER, STOREHEADER::lenPOS, STOREHEADER::lenWORD, STOREHEADER::maxpower, STOREHEADER::sFun, STOREHEADER::sInd, STOREHEADER::sSym, STOREHEADER::sVec, and STOREHEADER::wildoffset.

Referenced by SetFileIndex().

4.106.3.30 ReadSaveHeader()

```
WORD ReadSaveHeader ( )
```

Reads the header in the save file and sets function pointers and flags according to the information found there. Must be called before any other ReadSave... function.

Currently works only for the exchange between 32bit and 64bit machines (WORD size must be 2 or 4 bytes)!

It is called by CoLoad().

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4019 of file store.c.

4.106.3.31 ReadSaveIndex()

```
WORD ReadSaveIndex (
            FILEINDEX * fileind )
```

Reads a FILEINDEX from the open save file specified by AO.SaveData.Handle. Translations for adjusting endianness and data sizes are done if necessary.

Depends on the assumption that sizeof(FILEINDEX) is the same everywhere. If FILEINDEX or INDEXENTRY change, then this functions has to be adjusted.

Called by CoLoad() and FindInIndex().

Parameters

fileind	contains the read FILEINDEX after succesful return. must point to allocated, big enough memory.

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4137 of file store.c.

References INFILEINDEX, FiLeInDeX::next, FiLeInDeX::number, and FuNcTiOn::number.

4.106.3.32 ReadSaveVariables()

```
WORD ReadSaveVariables (
            UBYTE * buffer,
            UBYTE * top,
            LONG * size,
            LONG * outsize,
            INDEXENTRY * ind,
            LONG * stage )
```

Reads the variables from the open file specified by AO.SaveData.Handle. It reads the ∗size bytes and writes them to the ∗buffer. It is called by PutInStore().

If translation is necessary, the data might shrink or grow in size, then ∗*size* is adjusted so that the reading and writing fits into the memory from the buffer to the top. The actual number of read bytes is returned in ∗*size*, the number of written bytes is returned in ∗*outsize*.

If the ∗size is smaller than the actual size of the variables, this function will be called several times and needs to remember the current position in the variable structure. The parameter *stage* does this job. When ReadSaveVariables() is called for the first time, this parameter should have the value -1.

The parameter *ind* is used to get the number of variables.

Parameters

buffer	read variables are written into this allocated memory
top	upper end of allocated memory
size	number of bytes to read. might return a smaller number of read bytes if translation was necessary
outsize	if translation has be done, outsize contains the number of written bytes
ind	pointer of INDEXENTRY for the current expression. read-only
stage	should be -1 for the first call, will be increased by ReadSaveVariables to memorize the position in the variable structure

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4323 of file store.c.

References InDeXeNtRy::nfunctions, InDeXeNtRy::nindices, InDeXeNtRy::nsymbols, InDeXeNtRy::nvectors, and FuNcTiOn::spec.

4.106.3.33 ReadSaveTerm32()

```
UBYTE * ReadSaveTerm32 (
            UBYTE * bin,
            UBYTE * binend,
            UBYTE ** bout,
            UBYTE * boutend,
            UBYTE * top,
            int terminbuf )
```

Reads a single term from the given buffer at *bin* and write the translated term back to this buffer at *bout*.

ReadSaveTerm32() is currently the only instantiation of a ReadSaveTerm-function. It only deals with data that already has the correct endianness and that is resized to 32bit words but without being renumbered or translated in any other way. It uses the compress buffer AR.CompressBuffer.

The function is reentrant in order to cope with nested function arguments. It is called by ReadSaveExpression() and itself.

The *return value* indicates the position in the input buffer up to which the data has already been successfully processed. The parameter *bout* returns the corresponding position in the output buffer.

Parameters

bin	start of the input buffer
binend	end of the input buffer
bout	as input points to the beginning of the output buffer, as output points behind the already translated data in the output buffer
boutend	end of already decompressed data in output buffer
top	end of output buffer
terminbuf	flag whether decompressed data is already in the output buffer. used in recursive calls

Returns

pointer to the next unprocessed data in the input buffer

Definition at line 4683 of file store.c.

References ReadSaveTerm32().

Referenced by ReadSaveExpression(), and ReadSaveTerm32().

Here is the call graph for this function:

4.106.3.34 ReadSaveExpression()

```
WORD ReadSaveExpression (
            UBYTE * buffer,
            UBYTE * top,
            LONG * size,
            LONG * outsize )
```

Reads an expression from the open file specified by AO.SaveData.Handle. The endianness flip and a resizing without renumbering is done in this function. Thereafter the buffer consists of chunks with a uniform maximal word size (32bit at the moment). The actual renumbering is then done by calling the function ReadSaveTerm32(). The result is returned in *buffer*.

If the translation at some point doesn't fit into the buffer anymore, the function returns and must be called again. In any case *size* returns the number of successfully read bytes, *outsize* returns the number of successfully written bytes, and the file will be positioned at the next byte after the successfully read data.

It is called by PutInStore().

Parameters

buffer	output buffer, holds the (translated) expression
top	end of buffer
size	number of read bytes
outsize	number of written bytes

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 5092 of file store.c.

References ReadSaveTerm32().

Here is the call graph for this function:

4.107 store.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033 #define HIDEDEBUG
00034     #[ Includes : store.c
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040     #] Includes :
00041     #[ StoreExpressions :
00042         #[ OpenTemp :
00043
00044         Opens the scratch files for the input -> output operations.
00045
00046 */
00047
00048 WORD OpenTemp()
00049 {
00050     GETIDENTITY
00051     if ( AR.outfile->handle >= 0 ) {
00052         SeekFile(AR.outfile->handle,&(AR.outfile->filesize),SEEK_SET);
00053         AR.outfile->POposition = AR.outfile->filesize;
00054         AR.outfile->POfill = AR.outfile->PObuffer;
00055     }
00056     return(0);
00057 }
```

```c

/*
        #] OpenTemp :
        #[ SeekScratch :
*/

VOID SeekScratch(FILEHANDLE *fi, POSITION *pos)
{
    *pos = fi->POposition;
    ADDPOS(*pos,(TOLONG(fi->POfill)-TOLONG(fi->PObuffer)));
}

/*
        #] SeekScratch :
        #[ SetEndScratch :
*/

VOID SetEndScratch(FILEHANDLE *f, POSITION *position)
{
    if ( f->handle < 0 ) {
        SETBASEPOSITION(*position,(f->POfull-f->PObuffer)*sizeof(WORD));
    }
    else *position = f->filesize;
    SetScratch(f,position);
}

/*
        #] SetEndScratch :
        #[ SetEndHScratch :
*/

VOID SetEndHScratch(FILEHANDLE *f, POSITION *position)
{
    if ( f->handle < 0 ) {
        SETBASEPOSITION(*position,(f->POfull-f->PObuffer)*sizeof(WORD));
        f->POfill = f->POfull;
    }
    else {
#ifdef HIDEDEBUG
        POSITION possize;
        PUTZERO(possize);
        SeekFile(f->handle,&possize,SEEK_END);
        MesPrint("SetEndHScratch: filesize(th) = %12p, filesize(ex) = %12p",&(f->filesize),
                &(possize));
#endif
        *position = f->filesize;
        f->POposition = f->filesize;
        f->POfill = f->POfull = f->PObuffer;
    }
/*  SetScratch(f,position); */
}

/*
        #] SetEndHScratch :
        #[ SetScratch :
*/

VOID SetScratch(FILEHANDLE *f, POSITION *position)
{
    GETIDENTITY
    POSITION possize;
    LONG size, *whichInInBuf;
    if ( f == AR.hidefile ) whichInInBuf = &(AR.InHiBuf);
    else                    whichInInBuf = &(AR.InInBuf);
#ifdef HIDEDEBUG
    if ( f == AR.hidefile ) MesPrint("In the hide file");
    else MesPrint("In the input file");
    MesPrint("SetScratch to position %15p",position);
    MesPrint("POposition = %15p, full = %l, fill = %l"
        ,&(f->POposition),(f->POfull-f->PObuffer)*sizeof(WORD)
        ,(f->POfill-f->PObuffer)*sizeof(WORD));
#endif
    if ( ISLESSPOS(*position,f->POposition) ||
    ISGEPOSINC(*position,f->POposition,(f->POfull-f->PObuffer)*sizeof(WORD)) ) {
        if ( f->handle < 0 ) {
            if ( ISEQUALPOSINC(*position,f->POposition,
                (f->POfull-f->PObuffer)*sizeof(WORD)) ) goto endpos;
            MesPrint("Illegal position in SetScratch");
            Terminate(-1);
        }
        possize = *position;
        LOCK(AS.inputslock);
        SeekFile(f->handle,&possize,SEEK_SET);
        if ( ISNOTEQUALPOS(possize,*position) ) {
            UNLOCK(AS.inputslock);
            MesPrint("Cannot position file in SetScratch");
            Terminate(-1);
```

```
        }
#ifdef HIDEDEBUG
            MesPrint("SetScratch1(%w): position = %12p, size = %l, address = %x",position,f->POsize,f->PObuffer);
#endif
        if ( ( size = ReadFile(f->handle,(UBYTE *)(f->PObuffer),f->POsize) ) < 0
        || ( size & 1 ) != 0 ) {
            UNLOCK(AS.inputslock);
            MesPrint("Read error in SetScratch");
            Terminate(-1);
        }
        UNLOCK(AS.inputslock);
        if ( size == 0 ) {
            f->PObuffer[0] = 0;
        }
        f->POfill = f->PObuffer;
        f->POposition = *position;
#ifdef WORD2
        *whichInInBuf = size » 1;
#else
        *whichInInBuf = size / TABLESIZE(WORD,UBYTE);
#endif
        f->POfull = f->PObuffer + *whichInInBuf;
#ifdef HIDEDEBUG
            MesPrint("SetScratch2: size = %l, InInBuf = %l, fill = %l, full = %l"
            ,size,*whichInInBuf,(f->POfill-f->PObuffer)*sizeof(WORD)
            ,(f->POfull-f->PObuffer)*sizeof(WORD));
#endif
    }
    else {
endpos:
        DIFPOS(possize,*position,f->POposition);
        f->POfill = (WORD *)(BASEPOSITION(possize)+(UBYTE *)(f->PObuffer));
        *whichInInBuf = f->POfull-f->POfill;
    }
}

/*
        #] SetScratch :
        #[ RevertScratch :

        Reverts the input/output directions. This way input comes
        always from AR.infile

*/

WORD RevertScratch()
{
    GETIDENTITY
    FILEHANDLE *f;
    if ( AR.infile->handle >= 0 && AR.infile->handle != AR.outfile->handle ) {
        CloseFile(AR.infile->handle);
        AR.infile->handle = -1;
        remove(AR.infile->name);
    }
    f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
    AR.infile->POfull = AR.infile->POfill;
    AR.infile->POfill = AR.infile->PObuffer;
    if ( AR.infile->handle >= 0 ) {
        POSITION scrpos;
        PUTZERO(scrpos);
        SeekFile(AR.infile->handle,&scrpos,SEEK_SET);
        if ( ISNOTZEROPOS(scrpos) ) {
            return(MesPrint("Error with scratch output."));
        }
        if ( ( AR.InInBuf = ReadFile(AR.infile->handle,(UBYTE *)(AR.infile->PObuffer)
            ,AR.infile->POsize) ) < 0 || AR.InInBuf & 1 ) {
            return(MesPrint("Error while reading from scratch file"));
        }
        else {
            AR.InInBuf /= TABLESIZE(WORD,UBYTE);
        }
        AR.infile->POfull = AR.infile->PObuffer + AR.InInBuf;
    }
    PUTZERO(AR.infile->POposition);
    AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
    PUTZERO(AR.outfile->POposition);
    PUTZERO(AR.outfile->filesize);
    return(0);
}

/*
        #] RevertScratch :
        #[ ResetScratch :

        Resets the output scratch file to its beginning in such a way
        that the write routines can read it. The output buffers are
```

```
        left untouched as they may still be needed for extra declarations.

*/

WORD ResetScratch()
{
    GETIDENTITY
    FILEHANDLE *f;
    if ( AR.infile->handle >= 0 ) {
        CloseFile(AR.infile->handle); AR.infile->handle = -1;
        remove(AR.infile->name);
        PUTZERO(AR.infile->POposition);
        AR.infile->POfill = AR.infile->POfull = AR.infile->PObuffer;
    }
    if ( AR.outfile->handle >= 0 ) {
        POSITION scrpos;
        PUTZERO(scrpos);
        SeekFile(AR.outfile->handle,&scrpos,SEEK_SET);
        if ( ISNOTZEROPOS(scrpos) ) {
            return(MesPrint("Error with scratch output."));
        }
        if ( ( AR.InInBuf = ReadFile(AR.outfile->handle,(UBYTE *)(AR.outfile->PObuffer)
        ,AR.outfile->POsize) ) < 0 || AR.InInBuf & 1 ) {
            return(MesPrint("Error while reading from scratch file"));
        }
        else AR.InInBuf /= TABLESIZE(WORD,UBYTE);
        AR.outfile->POfull = AR.outfile->PObuffer + AR.InInBuf;
    }
    else AR.outfile->POfull = AR.outfile->POfill;
    AR.outfile->POfill = AR.outfile->PObuffer;
    PUTZERO(AR.outfile->POposition);
    f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
    return(0);
}

/*
        #] ResetScratch :
        #[ ReadFromScratch :

    Routine is used to copy files from scratch to hide.
*/

int ReadFromScratch(FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length)
{
    GETIDENTITY
    LONG l = BASEPOSITION(*length);
    if ( fi->handle < 0 ) {
        memcpy(buffer,fi->POfill,l);
    }
    else {
        SeekFile(fi->handle,pos,SEEK_SET);
        if ( ReadFile(fi->handle,buffer,l) != l ) {
            if ( fi == AR.hidefile )
                MesPrint("Error reading from hide file.");
            else
                MesPrint("Error reading from scratch file.");
            return(-1);
        }
    }
    return(0);
}

/*
        #] ReadFromScratch :
        #[ AddToScratch :

    Routine is used to copy files from scratch to hide.
*/

int AddToScratch(FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length,
             int withflush)
{
    GETIDENTITY
    LONG l = BASEPOSITION(*length), avail;
    DUMMYUSE(pos)
    fi->POfill = fi->POfull;
    while ( fi->POfill+l/sizeof(WORD) > fi->POstop ) {
        avail = (fi->POstop-fi->POfill)*sizeof(WORD);
        if ( avail > 0 ) {
            memcpy(fi->POfill,buffer,avail);
            l -= avail; buffer += avail;
        }
        if ( fi->handle < 0 ) {
            if ( ( fi->handle = (WORD)CreateFile(fi->name) ) < 0 ) {
                if ( fi == AR.hidefile )
                    MesPrint("Cannot create hide file %s",fi->name);
                else
```

```
                    MesPrint("Cannot create scratch file %s",fi->name);
                return(-1);
            }
            PUTZERO(fi->POposition);
        }
        SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
        if ( WriteFile(fi->handle,(UBYTE *)fi->PObuffer,fi->POsize) != fi->POsize )
            goto writeerror;
        ADDPOS(fi->POposition,fi->POsize);
        fi->POfill = fi->POfull = fi->PObuffer;
    }
    if ( l > 0 ) {
        memcpy(fi->POfill,buffer,l);
        fi->POfill += l/sizeof(WORD);
        fi->POfull = fi->POfill;
    }
    if ( withflush && fi->handle >= 0 && fi->POfill > fi->PObuffer ) {    /* flush */
        l = (LONG)fi->POfill - (LONG)fi->PObuffer;
        SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
        if ( WriteFile(fi->handle,(UBYTE *)fi->PObuffer,l) != l ) goto writeerror;
        ADDPOS(fi->POposition,fi->POsize);
        fi->POfill = fi->POfull = fi->PObuffer;
    }
    if ( withflush && fi->handle >= 0 )
        SETBASEPOSITION(fi->filesize,TellFile(fi->handle));
    return(0);
writeerror:
    if ( fi == AR.hidefile )
        MesPrint("Error writing to hide file. Disk full?");
    else
        MesPrint("Error writing to scratch file. Disk full?");
    return(-1);
}

/*
        #] AddToScratch :
        #[ CoSave :

        The syntax of the save statement is:

        save filename
        save filename expr1 expr2

*/

int CoSave(UBYTE *inp)
{
    GETIDENTITY
    UBYTE *p, c;
    WORD n = 0, i;
    WORD error = 0, type, number;
    LONG RetCode = 0, wSize;
    EXPRESSIONS e;
    INDEXENTRY *ind;
    INDEXENTRY *indold;
    WORD TMproto[SUBEXPSIZE];
    POSITION scrpos, scrpos1, filesize;
    int ii, j = sizeof(FILEINDEX)/(sizeof(LONG));
    LONG *lo;
    while ( *inp == ',' ) inp++;
    p = inp;

#ifdef WITHMPI
    if( PF.me != MASTER) return(0);
#endif

    if ( !*p ) return(MesPrint("No filename in save statement"));
    if ( FG.cTable[*p] > 1 && ( *p != '.' ) && ( *p != SEPARATOR ) && ( *p != ALTSEPARATOR ) )
                return(MesPrint("Illegal filename"));
    while ( *++p && *p != ',' ) {}
    c = *p;
    *p = 0;
    if ( !AP.preError ) {
        if ( ( RetCode = CreateFile((char *)inp) ) < 0 ) {
            return(MesPrint("Cannot open file %s",inp));
        }
    }
    AO.SaveData.Handle = (WORD)RetCode;
    PUTZERO(filesize);

    e = Expressions;
    n = NumExpressions;
    if ( c ) {          /* There follows a list of expressions */
        *p++ = c;
        inp = p;
        i = (WORD)(INFILEINDEX);
        if ( WriteStoreHeader(AO.SaveData.Handle) ) return(MesPrint("Error writing storage file
```

```
header"));
/*      PUTZERO(AO.SaveData.Index.number); */
/*      PUTZERO(AO.SaveData.Index.next); */
        lo = (LONG *)(&AO.SaveData.Index);
        for ( ii = 0; ii < j; ii++ ) *lo++ = 0;
        SETBASEPOSITION(AO.SaveData.Position,(LONG)sizeof(STOREHEADER));
        ind = AO.SaveData.Index.expression;
        if ( !AP.preError && WriteFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index))
        ,(LONG)sizeof(struct FiLeInDeX))!= (LONG)sizeof(struct FiLeInDeX) ) goto SavWrt;
        SeekFile(AO.SaveData.Handle,&(filesize),SEEK_END);
/*      ADDPOS(filesize,sizeof(struct FiLeInDeX)); */

        do {            /* Scan the list */
            if ( !FG.cTable[*p] || *p == '[' ) {
                p = SkipAName(p);
                if ( p == 0 ) return(-1);
            }
            c = *p; *p = 0;
            if ( GetVar(inp,&type,&number,CEXPRESSION,NOAUTO) != NAMENOTFOUND ) {
                if ( e[number].status == STOREDEXPRESSION ) {
/*
        Here we have to locate the stored expression, copy its index entry
        possibly after making a new fileindex and then copy the whole
        expression.
*/
                    if ( AP.preError ) goto NextExpr;
                    TMproto[0] = EXPRESSION;
                    TMproto[1] = SUBEXPSIZE;
                    TMproto[2] = number;
                    TMproto[3] = 1;
                    { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
                    AT.TMaddr = TMproto;
                    if ( ( indold = FindInIndex(number,&AR.StoreData,0,0) ) != 0 ) {
                        if ( i <= 0 ) {
/*
                            AO.SaveData.Index.next = filesize;

*/
                            SeekFile(AO.SaveData.Handle,&(AO.SaveData.Index.next),SEEK_END);
                            scrpos = AO.SaveData.Position;
                            SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
                            if ( ISNOTEQUALPOS(scrpos,AO.SaveData.Position) ) goto SavWrt;
                            if ( WriteFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index))
                                    ,(LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX)
)
                                goto SavWrt;
                            i = (WORD)(INFILEINDEX);
                            AO.SaveData.Position = AO.SaveData.Index.next;
                            lo = (LONG *)(&AO.SaveData.Index);
                            for ( ii = 0; ii < j; ii++ ) *lo++ = 0;
                            ind = AO.SaveData.Index.expression;
                            scrpos = AO.SaveData.Position;
                            SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
                            if ( ISNOTEQUALPOS(scrpos,AO.SaveData.Position) ) goto SavWrt;
                            if ( WriteFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index))
                            ,(LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) )
                                goto SavWrt;
                            ADDPOS(filesize,sizeof(struct FiLeInDeX));
                        }
                        *ind = *indold;
/*
                        ind->variables = SeekFile(AO.SaveData.Handle,&(AM.zeropos),SEEK_END);
*/
                        ind->variables = filesize;
                        ind->position = ind->variables;
                        ADDPOS(ind->position,DIFBASE(indold->position,indold->variables));
                        SeekFile(AR.StoreData.Handle,&(indold->variables),SEEK_SET);
                        wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
                        scrpos = ind->length;
                        ADDPOS(scrpos,DIFBASE(ind->position,ind->variables));
                        ADD2POS(filesize,scrpos);
                        SETBASEPOSITION(scrpos1,wSize);
                        do {
                            if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
                            if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,wSize)
                            != wSize ) {
                                MesPrint("ReadError");
                                error = -1;
                                goto EndSave;
                            }
                            if ( WriteFile(AO.SaveData.Handle,(UBYTE *)AT.WorkPointer,wSize)
                            != wSize ) goto SavWrt;
                            ADDPOS(scrpos,-wSize);
                        } while ( ISPOSPOS(scrpos) );
                        ADDPOS(AO.SaveData.Index.number,1);
                        ind++;
                    }
                    else error = -1;
```

```
                    i--;
                }
                else {
                    MesPrint("%s is not a stored expression",inp);
                    error = -1;
                }
NextExpr:;
            }
            else {
                MesPrint("%s is not an expression",inp);
                error = -1;
            }
            *p = c;
            if ( c != ',' && c ) {
                MesComp("Illegal character",inp,p);
                error = -1;
                goto EndSave;
            }
            if ( c ) c = *++p;
            inp = p;
        } while ( c );
        if ( !AP.preError ) {
            scrpos = AO.SaveData.Position;
            SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
            if ( ISNOTEQUALPOS(scrpos,AO.SaveData.Position) ) goto SavWrt;
        }
        if ( !AP.preError &&
        WriteFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index))
            ,(LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) ) goto SavWrt;
    }
    else if ( !AP.preError ) {              /* All stored expressions should be saved. Easy */
        if ( n > 0 ) { do {
            if ( e->status == STOREDEXPRESSION ) break;
            e++;
        } while ( --n > 0 ); }
        if ( n ) {
            wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
            PUTZERO(scrpos);
            SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);     /* Start at the beginning */
            scrpos = AR.StoreData.Fill;         /* Number of bytes to be copied */
            SETBASEPOSITION(scrpos1,wSize);
            do {
                if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
                if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,wSize) != wSize ) {
                    MesPrint("ReadError");
                    error = -1;
                    goto EndSave;
                }
                if ( WriteFile(AO.SaveData.Handle,(UBYTE *)AT.WorkPointer,wSize) != wSize )
                    goto SavWrt;
                ADDPOS(scrpos,-wSize);
            } while ( ISPOSPOS(scrpos) );
        }
    }
EndSave:
    if ( !AP.preError ) {
        CloseFile(AO.SaveData.Handle);
        AO.SaveData.Handle = -1;
    }
    return(error);
SavWrt:
    MesPrint("WriteError");
    error = -1;
    goto EndSave;
}

/*
        #] CoSave :
        #[ CoLoad :
*/

int CoLoad(UBYTE *inp)
{
    GETIDENTITY
    INDEXENTRY *ind;
    LONG RetCode;
    UBYTE *p, c;
    WORD num, i, error = 0;
    WORD type, number, silentload = 0;
    WORD TMproto[SUBEXPSIZE];
    POSITION scrpos,firstposition;
    while ( *inp == ',' ) inp++;
    p = inp;
    if ( ( *p == ',' && p[1] == '-' ) || *p == '-' ) {
        if ( *p == ',' ) p++;
        p++;
        if ( *p == 's' || *p == 'S' ) {
```

```
            silentload = 1;
            while ( *p && ( *p != ',' && *p != '-' && *p != '+'
            && *p != SEPARATOR && *p != ALTSEPARATOR && *p != '.' ) ) p++;
        }
        else if ( *p != ',' ) {
            return(MesPrint("Illegal option in Load statement"));
        }
        while ( *p == ',' ) p++;
    }
    inp = p;
    if ( !*p ) return(MesPrint("No filename in load statement"));
    if ( FG.cTable[*p] > 1 && ( *p != '.' ) && ( *p != SEPARATOR ) && ( *p != ALTSEPARATOR ) )
                return(MesPrint("Illegal filename"));
    while ( *++p && *p != ',' ) {}
    c = *p;
    *p = 0;
    if ( ( RetCode = OpenFile((char *)inp) ) < 0 ) {
        return(MesPrint("Cannot open file %s",inp));
    }

    if ( SetFileIndex() ) {
        MesCall("CoLoad");
        SETERROR(-1)
    }

    AO.SaveData.Handle = (WORD)(RetCode);

#ifdef SYSDEPENDENTSAVE
    if ( ReadFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index)),
         (LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) ) goto LoadRead;
#else
    if ( ReadSaveHeader() ) goto LoadRead;
    TELLFILE(AO.SaveData.Handle,&firstposition);
    if ( ReadSaveIndex(&AO.SaveData.Index) ) goto LoadRead;
#endif
    if ( c ) {          /* There follows a list of expressions */
        *p++ = c;
        inp = p;

        do {            /* Scan the list */
            if ( !FG.cTable[*p] || *p == '[' ) {
                p = SkipAName(p);
                if ( p == 0 ) return(-1);
            }
            c = *p; *p = 0;
            if ( GetVar(inp,&type,&number,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
                MesPrint("Conflicting name: %s",inp);
                error = -1;
            }
            else {
                if ( ( num = EntVar(CEXPRESSION,inp,STOREDEXPRESSION,0,0,0) ) >= 0 ) {
                    TMproto[0] = EXPRESSION;
                    TMproto[1] = SUBEXPSIZE;
                    TMproto[2] = num;
                    TMproto[3] = 1;
                    { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
                    AT.TMaddr = TMproto;
                    SeekFile(AO.SaveData.Handle,&firstposition,SEEK_SET);
                    AO.SaveData.Position = firstposition;
                    if ( ReadSaveIndex(&AO.SaveData.Index) ) goto LoadRead;
                    if ( ( ind = FindInIndex(num,&AO.SaveData,1,0) ) != 0 ) {
                        if ( !error ) {
                            if ( PutInStore(ind,num) ) error = -1;
                            else if ( !AM.silent && silentload == 0 )
                                MesPrint(" %s loaded",ind->name);
                        }
/*
!!! Added 1-feb-1998
*/
                        Expressions[num].counter = -1;
                    }
                    else {
                        MesPrint(" %s not found",inp);
                        error = -1;
                    }
                }
                else error = -1;
            }
            *p = c;
            if ( c != ',' && c ) {
                MesComp("Illegal character",inp,p);
                error = -1;
                goto EndLoad;
            }
            if ( c ) c = *++p;
            inp = p;
        } while ( c );
```

```
        scrpos = AR.StoreData.Position;
        SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
        if ( ISNOTEQUALPOS(scrpos,AR.StoreData.Position) ) goto LoadWrt;
        if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(AR.StoreData.Index))
            ,(LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) ) goto LoadWrt;
    }
    else {              /* All saved expressions should be stored. Easy */
        i = (WORD)BASEPOSITION(AO.SaveData.Index.number);
        ind = AO.SaveData.Index.expression;
#ifdef SYSDEPENDENTSAVE
        if ( i > 0 ) { do {
            if ( GetVar((UBYTE *)(ind->name),&type,&number,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
                MesPrint("Conflicting name: %s",ind->name);
                error = -1;
            }
            else {
                if ( ( num = EntVar(CEXPRESSION,(UBYTE *)(ind->name),STOREDEXPRESSION,0,0,0) ) >= 0 ) {
                    if ( !error ) {
                        if ( PutInStore(ind,num) ) error = -1;
                        else if ( !AM.silent && silentload == 0 )
                            MesPrint(" %s loaded",ind->name);
                    }
                }
                else error = -1;
            }
            i--;
            if ( i == 0 && ISNOTZEROPOS(AO.SaveData.Index.next) ) {
                SeekFile(AO.SaveData.Handle,&(AO.SaveData.Index.next),SEEK_SET);
                if ( ReadFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index)),
                    (LONG)sizeof(struct FiLeInDeX)) != (LONG)sizeof(struct FiLeInDeX) ) goto LoadRead;
                i = (WORD)BASEPOSITION(AO.SaveData.Index.number);
                ind = AO.SaveData.Index.expression;
            }
            else ind++;
        } while ( i > 0 ); }
#else
        if ( i > 0 ) {
            do {
                if ( GetVar((UBYTE *)(ind->name),&type,&number,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
                    MesPrint("Conflicting name: %s",ind->name);
                    error = -1;
                }
                else {
                    if ( ( num = EntVar(CEXPRESSION,(UBYTE *)(ind->name),STOREDEXPRESSION,0,0,0) ) >= 0 ) {
                        if ( !error ) {
                            if ( PutInStore(ind,num) ) error = -1;
                            else if ( !AM.silent && silentload == 0 )
                                MesPrint(" %s loaded",ind->name);
                        }
                    }
                    else error = -1;
                }
                i--;
                if ( i == 0 && (ISNOTZEROPOS(AO.SaveData.Index.next) || AO.bufferedInd) ) {
                    SeekFile(AO.SaveData.Handle,&(AO.SaveData.Index.next),SEEK_SET);
                    if ( ReadSaveIndex(&AO.SaveData.Index) ) goto LoadRead;
                    i = (WORD)BASEPOSITION(AO.SaveData.Index.number);
                    ind = AO.SaveData.Index.expression;
                }
                else ind++;
            } while ( i > 0 );
        }
#endif
    }
EndLoad:
#ifndef SYSDEPENDENTSAVE
    if ( AO.powerFlag ) {
        MesPrint("WARNING: min-/maxpower had to be adjusted!");
    }
    if ( AO.resizeFlag ) {
        MesPrint("ERROR: could not downsize data!");
        return ( -2 );
    }
#endif
    CloseFile(AO.SaveData.Handle);
    AO.SaveData.Handle = -1;
    SeekFile(AR.StoreData.Handle,&(AC.StoreFileSize),SEEK_END);
    return(error);
LoadWrt:
    MesPrint("WriteError");
    error = -1;
    goto EndLoad;
LoadRead:
    MesPrint("ReadError");
```

```
    error = -1;
    goto EndLoad;
}

/*
        #] CoLoad :
        #[ DeleteStore :

        Routine deletes the contents of the entire storage file.
        We close the file and recreate it.
        If par > 0 we have to remove the expressions from the namelists.
*/

WORD DeleteStore(WORD par)
{
    GETIDENTITY
    char *s;
    WORD j, n = 0;
    EXPRESSIONS e_in, e_out;
    WORD DidClean = 0;
    if ( AR.StoreData.Handle >= 0 ) {
        if ( par > 0 ) {
            n = NumExpressions;
            j = 0;
            e_in = e_out = Expressions;
            if ( n > 0 ) { do {
                if ( e_in->status == STOREDEXPRESSION ) {
                    NAMENODE *node = GetNode(AC.exprnames,
                            AC.exprnames->namebuffer+e_in->name);
                    node->type = CDELETE;
                    DidClean = 1;
                }
                else {
                    if ( e_out != e_in ) {
                        NAMENODE *node;
                        node = GetNode(AC.exprnames,
                            AC.exprnames->namebuffer+e_in->name);
                        node->number = (WORD)(e_out - Expressions);
                        e_out->onfile = e_in->onfile;
                        e_out->prototype = e_in->prototype;
                        e_out->printflag = 0;
                        e_out->status = e_in->status;
                        e_out->name = e_in->name;
                        e_out->inmem = e_in->inmem;
                        e_out->counter = e_in->counter;
                        e_out->numfactors = e_in->numfactors;
                        e_out->numdummies = e_in->numdummies;
                        e_out->compression = e_in->compression;
                        e_out->namesize = e_in->namesize;
                        e_out->whichbuffer = e_in->whichbuffer;
                        e_out->hidelevel = e_in->hidelevel;
                        e_out->node = e_in->node;
                        e_out->replace = e_in->replace;
                        e_out->vflags = e_in->vflags;
#ifdef PARALLELCODE
                        e_out->partodo = e_in->partodo;
#endif
                    }
                    e_out++;
                    j++;
                }
                e_in++;
            } while ( --n > 0 ); }
            NumExpressions = j;
            if ( DidClean ) CompactifyTree(AC.exprnames,EXPRNAMES);
        }
        AR.StoreData.Handle = -1;
        CloseFile(AC.StoreHandle);
        AC.StoreHandle = -1;
        {
/*
            Knock out the storage caches (25-apr-1990!)
*/
            STORECACHE st;
            st = (STORECACHE)(AT.StoreCache);
            while ( st ) {
                SETBASEPOSITION(st->position,-1);
                SETBASEPOSITION(st->toppos,-1);
                st = st->next;
            }
#ifdef WITHPTHREADS
            for ( j = 1; j < AM.totalnumberofthreads; j++ ) {
                st = (STORECACHE)(AB[j]->T.StoreCache);
                while ( st ) {
                    SETBASEPOSITION(st->position,-1);
                    SETBASEPOSITION(st->toppos,-1);
                    st = st->next;
```

```
                }
            }
#endif
        }
        PUTZERO(AC.StoreFileSize);
        s = FG.fname; while ( *s ) s++;
#ifdef VMS
        *s = ';'; s[1] = '*'; s[2] = 0;
        remove(FG.fname);
        *s = 0;
#endif
        return(AC.StoreHandle = CreateFile(FG.fname));
    }
    else return(0);
}

/*
        #] DeleteStore :
        #[ PutInStore :

        Copies the expression indicated by ind from a load file to the
        internal storage file. A return value of zero indicates that
        everything is OK.

*/

WORD PutInStore(INDEXENTRY *ind, WORD num)
{
    GETIDENTITY
    INDEXENTRY *newind;
    LONG wSize;
#ifndef SYSDEPENDENTSAVE
    LONG wSizeOut;
    LONG stage;
#endif
    POSITION scrpos,scrpos1;
    newind = NextFileIndex(&(Expressions[num].onfile));
    *newind = *ind;
#ifndef SYSDEPENDENTSAVE
    SETBASEPOSITION(newind->length, 0);
#endif
    newind->variables = AR.StoreData.Fill;
    SeekFile(AR.StoreData.Handle,&(newind->variables),SEEK_SET);
    if ( ISNOTEQUALPOS(newind->variables,AR.StoreData.Fill) ) goto PutErrS;
    newind->position = newind->variables;
#ifdef SYSDEPENDENTSAVE
    ADDPOS(newind->position,DIFBASE(ind->position,ind->variables));
#endif
    /* set read position to ind->variables */
    scrpos = ind->variables;
    SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
    if ( ISNOTEQUALPOS(scrpos,ind->variables) ) goto PutErrS;
    /* set max size for read-in */
    wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
#ifdef SYSDEPENDENTSAVE
    scrpos = ind->length;
    ADDPOS(scrpos,DIFBASE(ind->position,ind->variables));
    ADD2POS(AR.StoreData.Fill,scrpos);
#endif
    SETBASEPOSITION(scrpos1,wSize);
#ifndef SYSDEPENDENTSAVE
    /* prepare look-up table for tensor functions */
    if ( ind->nfunctions ) {
        AO.tensorList = (UBYTE *)Malloc1(MAXSAVEFUNCTION,"PutInStore");
    }
    SETBASEPOSITION(scrpos, DIFBASE(ind->position,ind->variables));
    /* copy variables first */
    stage = -1;
    do {
        wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
        if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
        wSizeOut = wSize;
        if ( ReadSaveVariables(
            (UBYTE *)AT.WorkPointer, (UBYTE *)AT.WorkTop, &wSize, &wSizeOut, ind, &stage) ) {
            goto PutErrS;
        }
        if ( WriteFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, wSizeOut)
        != wSizeOut ) goto PutErrS;
        ADDPOS(scrpos,-wSize);
        ADDPOS(newind->position, wSizeOut);
        ADDPOS(AR.StoreData.Fill, wSizeOut);
    } while ( ISPOSPOS(scrpos) );
    /* then copy the expression itself */
    wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
    scrpos = ind->length;
#endif
    do {
```

```
        wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
        if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
#ifdef SYSDEPENDENTSAVE
        if ( ReadFile(AO.SaveData.Handle,(UBYTE *)AT.WorkPointer,wSize)
        != wSize ) goto PutErrS;
        if ( WriteFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,wSize)
        != wSize ) goto PutErrS;
        ADDPOS(scrpos,-wSize);
#else
        wSizeOut = wSize;

        if ( ReadSaveExpression((UBYTE *)AT.WorkPointer, (UBYTE *)AT.WorkTop, &wSize, &wSizeOut) ) {
            goto PutErrS;
        }

        if ( WriteFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, wSizeOut)
        != wSizeOut ) goto PutErrS;
        ADDPOS(scrpos,-wSize);
        ADDPOS(AR.StoreData.Fill, wSizeOut);
        ADDPOS(newind->length, wSizeOut);
#endif
    } while ( ISPOSPOS(scrpos) );
    /* free look-up table for tensor functions */
    if ( ind->nfunctions ) {
        M_free(AO.tensorList,"PutInStore");
    }
    scrpos = AR.StoreData.Position;
    SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
    if ( ISNOTEQUALPOS(scrpos,AR.StoreData.Position) ) goto PutErrS;
    if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&AR.StoreData.Index),(LONG)sizeof(FILEINDEX))
    == (LONG)sizeof(FILEINDEX) ) return(0);
PutErrS:
    return(MesPrint("File error"));
}

/*
        #] PutInStore :
        #[ GetTerm :

        Gets one term from input scratch stream.
        Puts it in 'term'.
        Returns the length of the term.

        Used by Processor        (proces.c)
                WriteAll         (sch.c)
                WriteOne         (sch.c)
                GetMoreTerms     (store.c)
                ToStorage        (store.c)
                CoFillExpression (comexpr.c)
                FactorInExpr     (factor.c)
                LoadOpti         (optim.c)
                PF_Processor     (parallel.c)
                ThreadsProcessor (threads.c)
        In multi thread/processor mode all calls are done by the master.
        Note however that other routines, used by the threads, can use
        the same file. Hence we need to be careful about SeekFile and locks.
*/

WORD GetTerm(PHEAD WORD *term)
{
    GETBIDENTITY
    WORD *inp, i, j = 0, len;
    LONG InIn, *whichInInBuf;
    WORD *r, *m, *mstop = 0, minsiz = 0, *bra = 0, *from;
    WORD first, *start = 0, testing = 0;
    FILEHANDLE *fi;
    AN.deferskipped = 0;
    if ( AR.GetFile == 2 ) {
        fi = AR.hidefile;
        whichInInBuf = &(AR.InHiBuf);
    }
    else {
        fi = AR.infile;
        whichInInBuf = &(AR.InInBuf);
    }
    InIn = *whichInInBuf;
    from = term;
    if ( AR.KeptInHold ) {
        r = AR.CompressBuffer;
        i = *r;
        AR.KeptInHold = 0;
        if ( i <= 0 ) { *term = 0; goto RegRet; }
        m = term;
        NCOPY(m,r,i);
        goto RegRet;
    }
    if ( AR.DeferFlag ) {
```

```
        m = AR.CompressBuffer;
        if ( *m > 0 ) {
            mstop = m + *m;
            mstop -= ABS(mstop[-1]);
            m++;
            while ( m < mstop ) {
                if ( *m == HAAKJE ) {
                    testing = 1;
                    mstop = m + m[1];
                    bra = (WORD *)(((UBYTE *)(term)) + 2*AM.MaxTer);
                    m = AR.CompressBuffer+1;
                    r = bra;
                    while ( m < mstop ) *r++ = *m++;
                    mstop = r;
                    minsiz = WORDDIF(mstop,bra);
                    goto ReStart;
/*
                    We have the bracket to be tested in bra till mstop
*/
                }
                m += m[1];
            }
        }
        bra = (WORD *)(((UBYTE *)(term)) + 2*AM.MaxTer);
        mstop = bra+1;
        *bra = 0;
        minsiz = 1;
        testing = 1;
    }
ReStart:
    first = 0;
    r = AR.CompressBuffer;
    if ( fi->handle >= 0 ) {
        if ( InIn <= 0 ) {
            ADDPOS(fi->POposition,(fi->POfull-fi->PObuffer)*sizeof(WORD));
            LOCK(AS.inputslock);
            SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
            InIn = ReadFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize);
            UNLOCK(AS.inputslock);
            if ( ( InIn < 0 ) || ( InIn & 1 ) ) {
                goto GTerr;
            }
#ifdef WORD2
            InIn >>= 1;
#else
            InIn /= TABLESIZE(WORD,UBYTE);
#endif
            *whichInInBuf = InIn;
            if ( !InIn ) { *r = 0; *from = 0; goto RegRet; }
            fi->POfill = fi->PObuffer;
            fi->POfull = fi->PObuffer + InIn;
        }
        inp = fi->POfill;
        if ( ( len = i = *inp ) == 0 ) {
            (*whichInInBuf)--;
            (fi->POfill)++;
            *r = 0;
            *from = 0;
            goto RegRet;
        }
        if ( i < 0 ) {
            InIn--;
            inp++;
            r++;
            start = term;
            *term++ = -i + 1;
            while ( ++i <= 0 ) *term++ = *r++;
            if ( InIn > 0 ) {
                i = *inp++;
                InIn--;
                *start += i;
                *(AR.CompressBuffer) = len = *start;
            }
            else {
                first = 1;
                goto NewIn;
            }
        }
        InIn -= i;
        if ( InIn < 0 ) {
            j = (WORD)(- InIn);
            i -= j;
        }
        else j = 0;
        while ( --i >= 0 ) {
                *r++ = *term++ = *inp++;
        }
```

```
        if ( j ) {
NewIn:
            ADDPOS(fi->POposition,(fi->POfull-fi->PObuffer)*sizeof(WORD));
            LOCK(AS.inputslock);
            SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
            InIn = ReadFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize);
            UNLOCK(AS.inputslock);
            if ( ( InIn <= 0 ) || ( InIn & 1 ) ) {
                goto GTerr;
            }
#ifdef WORD2
            InIn »= 1;
#else
            InIn /= TABLESIZE(WORD,UBYTE);
#endif
            inp = fi->PObuffer;
            fi->POfull = inp + InIn;

            if ( first ) {
                j = *inp++;
                InIn--;
                *start += j;
                *(AR.CompressBuffer) = len = *start;
            }
            InIn -= j;
            while ( --j >= 0 ) { *r++ = *term++ = *inp++; }
        }
        fi->POfill = inp;
        *whichInInBuf = InIn;
        AR.DefPosition = fi->POposition;
        ADDPOS(AR.DefPosition,((UBYTE *)(fi->POfill)-(UBYTE *)(fi->PObuffer)));
    }
    else {
        inp = fi->POfill;
        if ( inp >= fi->POfull ) { *from = 0; goto RegRet; }
        len = j = *inp;
        if ( j < 0 ) {
            inp++;
            *term++ = *r++ = len = - j + 1 + *inp;
            while ( ++j <= 0 ) *term++ = *r++;
            j = *inp++;
        }
        else if ( !j ) j = 1;
        while ( --j >= 0 ) { *r++ = *term++ = *inp++; }
        fi->POfill = inp;
/*%%%%%ADDED 7-apr-2006 for Keep Brackets in bucket */
        SETBASEPOSITION(AR.DefPosition,((UBYTE *)(fi->POfill)-(UBYTE *)(fi->PObuffer)));
        if ( inp > fi->POfull ) {
            goto GTerr;
        }
    }
    if ( r >= AR.ComprTop ) {
        MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
        Terminate(-1);
    }
    AR.CompressPointer = r; *r = 0;
/*
        The next *from is a bug fix that made the program read in forbidden
        territory.
*/
    if ( testing && *from != 0 ) {
        WORD jj;
        r = from;
        jj = *r - 1 - ABS(*(r+*r-1));
        if ( jj < minsiz ) goto strip;
        r++;
        m = bra;
        while ( m < mstop ) {
            if ( *m != *r ) {
strip:          r = from;
                m = r + *r;
                mstop = m - ABS(m[-1]);
                r++;
                while ( r < mstop ) {
                    if ( *r == HAAKJE ) {
                        *r++ = 1;
                        *r++ = 1;
                        *r++ = 3;
                        len = WORDDIF(r,from);
                        *from = len;
                        goto RegRet;
                    }
                    r += r[1];
                }
                goto RegRet;
            }
            m++;
```

```
            r++;
        }
        term = from;
        AN.deferskipped++;
        goto ReStart;
    }
RegRet:;
/*
            #[ debug :
*/
    {
        UBYTE OutBuf[140];
/*      if ( AP.DebugFlag ) { */
        if ( ( AP.PreDebug & DUMPINTERMS ) == DUMPINTERMS ) {
            MLOCK(ErrorMessageLock);
            AO.OutFill = AO.OutputLine = OutBuf;
            AO.OutSkip = 3;
            FiniLine();
            r = from;
            i = *r;
            TokenToLine((UBYTE *)("Input: "));
            if ( i == 0 ) {
                TokenToLine((UBYTE *)"zero");
            }
            else if ( i < 0 ) {
                TokenToLine((UBYTE *)"negative!!");
            }
            else {
                while ( --i >= 0 ) {
                    TalToLine((UWORD)(*r++)); TokenToLine((UBYTE *)"  ");
                }
            }
            FiniLine();
            MUNLOCK(ErrorMessageLock);
        }
    }
/*
            #] debug :
*/
    return(*from);
GTerr:
    MesPrint("Error while reading scratch file in GetTerm");
    Terminate(-1);
    return(-1);
}

/*
        #] GetTerm :
        #[ GetOneTerm :

        Gets one term from stream AR.infile->handle.
        Puts it in 'term'.
        Returns the length of the term.
        Input is unbuffered.
        Compression via AR.CompressPointer
        par is actually in all calls a file handle

        Routine is called from
            DoOnePow           Get one power of an expression
            Deferred           Get the contents of a bracket
            GetFirstBracket
            FindBracket
        We should do something about the lack of buffering.
        Maybe a buffer of a few times AM.MaxTer (MaxTermSize*sizeof(WORD)).
        Each thread will need its own buffer!

        If par == 0 we use ReadPosFile which can fill the whole buffer.
        If par == 1 we use ReadFile and do actual read operations.

        Note: we cannot use ReadPosFile when running in the master thread.
*/

WORD GetOneTerm(PHEAD WORD *term, FILEHANDLE *fi, POSITION *pos, int par)
{
    GETBIDENTITY
    WORD i, *p;
    LONG j, siz;
    WORD *r, *rr = AR.CompressPointer;
    int error = 0;
    r = rr;
    if ( fi->handle >= 0 ) {
#ifdef READONEBYONE
#ifdef WITHPTHREADS
/*
        This code needs some investigation.
        It may be that we should do this always.
        It may be that even for workers it is no good.
```

```
        We may have to make a variable like AM.ReadDirect with
        if ( AM.ReadDirect ) par = 1;
        and a user command like
        On ReadDirect;
*/
        if ( AT.identity > 0 ) par = 1;
#endif
#endif
/*
        To be changed:
        1: check first whether the term lies completely inside the buffer
        2: if not a: use old strategy for AT.identity == 0 (master)
                 b: for workers, position file and read buffer
*/
        if ( par == 0 ) {
            siz = ReadPosFile(BHEAD fi,(UBYTE *)term,1L,pos);
        }
        else {
            LOCK(AS.inputslock);
            SeekFile(fi->handle,pos,SEEK_SET);
            siz = ReadFile(fi->handle,(UBYTE *)term,sizeof(WORD));
            UNLOCK(AS.inputslock);
            ADDPOS(*pos,siz);
        }
        if ( siz == sizeof(WORD) ) {
            p = term;
            j = i = *term++;
            if ( ( i > AM.MaxTer/((WORD)sizeof(WORD)) ) || ( -i >= AM.MaxTer/((WORD)sizeof(WORD)) ) )
            {
                error = 1;
                goto ErrGet;
            }
            r++;
            if ( i < 0 ) {
                *p = -i + 1;
                while ( ++i <= 0 ) *term++ = *r++;
                if ( par == 0 ) {
                    siz = ReadPosFile(BHEAD fi,(UBYTE *)term,1L,pos);
                }
                else {
                    LOCK(AS.inputslock);
                    SeekFile(fi->handle,pos,SEEK_SET);
                    siz = ReadFile(fi->handle,(UBYTE *)term,sizeof(WORD));
                    UNLOCK(AS.inputslock);
                    ADDPOS(*pos,sizeof(WORD));
                }
                if ( siz != sizeof(WORD) ) {
                    error = 2;
                    goto ErrGet;
                }
                *p += *term;
                j = *term;
                if ( ( j > AM.MaxTer/((WORD)sizeof(WORD)) ) || ( j <= 0 ) )
                {
                    error = 3;
                    goto ErrGet;
                }
                *rr = *p;
            }
            else {
                if ( !j ) return(0);
                j--;
            }
            i = (WORD)j;
            if ( par == 0 ) {
                siz = ReadPosFile(BHEAD fi,(UBYTE *)term,j,pos);
                j *= TABLESIZE(WORD,UBYTE);
            }
            else {
                j *= TABLESIZE(WORD,UBYTE);
                LOCK(AS.inputslock);
                SeekFile(fi->handle,pos,SEEK_SET);
                siz = ReadFile(fi->handle,(UBYTE *)term,j);
                UNLOCK(AS.inputslock);
                ADDPOS(*pos,j);
            }
            if ( siz != j ) {
                error = 4;
                goto ErrGet;
            }
            while ( --i >= 0 ) *r++ = *term++;
            if ( r >= AR.ComprTop ) {
                MLOCK(ErrorMessageLock);
                MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
```

```
            AR.CompressPointer = r; *r = 0;
            return(*p);
        }
        error = 5;
    }
    else {
/*
        Here the whole expression is in the buffer.
*/
        fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + BASEPOSITION(*pos));
        p = fi->POfill;
        if ( p >= fi->POfull ) { *term = 0; return(0); }
        j = i = *p;
        if ( i < 0 ) {
            p++;
            j = *r++ = *term++ = -i + 1 + *p;
            while ( ++i <= 0 ) *term++ = *r++;
            i = *p++;
        }
        if ( i == 0 ) { i = 1; *r++ = 0; *term++ = 0; }
        else { while ( --i >= 0 ) { *r++ = *term++ = *p++; } }
        fi->POfill = p;
        SETBASEPOSITION(*pos,(UBYTE *)(fi->POfill)-(UBYTE *)(fi->PObuffer));
        if ( p <= fi->POfull ) {
            if ( r >= AR.ComprTop ) {
                MLOCK(ErrorMessageLock);
                MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
            AR.CompressPointer = r; *r = 0;
            return((WORD)j);
        }
        error = 6;
    }
ErrGet:
    MLOCK(ErrorMessageLock);
    MesPrint("Error while reading scratch file in GetOneTerm (%d)",error);
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(-1);
}

/*
        #] GetOneTerm :
        #[ GetMoreTerms :
    Routine collects more contents of brackets inside a function,
    indicated by the number in AC.CollectFun.
    The first term is in term already.
    We can keep calling GetTerm either till a bracket is finished
    or till it would make the term too long (> AM.MaxTer/2)
    In all cases this function makes that the routine GetTerm
    has a term in 'hold', so the AR.KeptInHold flag must be turned on.
*/

WORD GetMoreTerms(WORD *term)
{
    GETIDENTITY
    WORD *t, *r, *m, *h, *tstop, i, inc, same;
    WORD extra;
    WORD retval = 0;
/*
    We use 23% as a quasi-random default value.
*/
    extra = ((AM.MaxTer/sizeof(WORD))*((LONG)100-AC.CollectPercentage))/100;
    if ( extra < 23 ) extra = 23;
/*
    First find the bracket pointer
*/
    t = term + *term;
    tstop = t - ABS(t[-1]);
    h = term+1;
    while ( *h != HAAKJE && h < tstop ) h += h[1];
    if ( h >= tstop ) return(retval);
    inc = FUNHEAD+ARGHEAD+1-h[1];
    same = WORDDIF(h,term) + h[1] - 1;
    r = m = t + inc;
    tstop = h + h[1];
    while ( t > tstop ) *--r = *--t;
    r--;
    *r = WORDDIF(m,r);
    while ( GetTerm(BHEAD m) > 0 ) {
        r = m + 1;
        t = m + *m - 1;
        if ( same > ( i = ( *m - ABS(*t) -1 ) ) ) { /* Must fail */
            if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
            break;
```

```
        }
        t = term+1;
        i = same;
        while ( --i >= 0 ) {
            if ( *r != *t ) {
                if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
                goto FullTerm;
            }
            r++; t++;
        }
        if ( ( WORDDIF(m,term) + i + extra ) > (WORD)(AM.MaxTer/sizeof(WORD)) ) {
/* 23 = 3 +20. The 20 is to have some extra for substitutions or whatever */
            if ( AS.CollectOverFlag == 0 && AC.AltCollectFun == 0 ) {
                Warning("Bracket contents too long in Collect statement");
                Warning("Contents spread over more than one term");
                Warning("If possible: increase MaxTermSize in setfile");
                AS.CollectOverFlag = 1;
            }
            else if ( AC.AltCollectFun ) {
                AS.CollectOverFlag = 2;
            }
            break;
        }
        tstop = m + *m;
        *m -= same;
        m++;
        while ( r < tstop ) *m++ = *r++;
        retval++;
        if ( extra == 23 ) extra = ((AM.MaxTer/sizeof(WORD))/6);
    }
FullTerm:
    h[1] = WORDDIF(m,h);
    if ( AS.CollectOverFlag > 1 ) {
        *h = AC.AltCollectFun;
        if ( AS.CollectOverFlag == 3 ) AS.CollectOverFlag = 1;
    }
    else *h = AC.CollectFun;
    h[2] |= DIRTYFLAG;
    h[FUNHEAD] = h[1] - FUNHEAD;
    h[FUNHEAD+1] = 0;
    if ( ToFast(h+FUNHEAD,h+FUNHEAD) ) {
        if ( h[FUNHEAD] <= -FUNCTION ) {
            h[1] = FUNHEAD+1;
            m = h + FUNHEAD+1;
        }
        else {
            h[1] = FUNHEAD+2;
            m = h + FUNHEAD+2;
        }
    }
    *m++ = 1;
    *m++ = 1;
    *m++ = 3;
    *term = WORDDIF(m,term);
    AR.KeptInHold = 1;
    return(retval);
}

/*
        #] GetMoreTerms :
        #[ GetMoreFromMem :

*/

WORD GetMoreFromMem(WORD *term, WORD **tpoin)
{
    GETIDENTITY
    WORD *t, *r, *m, *h, *tstop, i, j, inc, same;
    LONG extra = 23;
/*
    First find the bracket pointer
*/
    t = term + *term;
    tstop = t - ABS(t[-1]);
    h = term+1;
    while ( *h != HAAKJE && h < tstop ) h += h[1];
    if ( h >= tstop ) return(0);
    inc = FUNHEAD+ARGHEAD+1-h[1];
    same = WORDDIF(h,term) + h[1] - 1;
    r = m = t + inc;
    tstop = h + h[1];
    while ( t > tstop ) *--r = *--t;
    r--;
    *r = WORDDIF(m,r);
    while ( **tpoin ) {
        r = *tpoin; j = *r;
        for ( i = 0; i < j; i++ ) m[i] = *r++;
```

```
        *tpoin = r;
        r = m + 1;
        t = m + *m - 1;
        if ( same > ( i = ( *m - ABS(*t) -1 ) ) ) { /* Must fail */
            if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
            break;
        }
        t = term+1;
        i = same;
        while ( --i >= 0 ) {
            if ( *r != *t ) {
                if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
                goto FullTerm;
            }
            r++; t++;
        }
        if ( ( WORDDIF(m,term) + i + extra ) > (LONG)(AM.MaxTer/(2*sizeof(WORD))) ) {
/* 23 = 3 +20. The 20 is to have some extra for substitutions or whatever */
            if ( AS.CollectOverFlag == 0 && AC.AltCollectFun == 0 ) {
                Warning("Bracket contents too long in Collect statement");
                Warning("Contents spread over more than one term");
                Warning("If possible: increase MaxTermSize in setfile");
                AS.CollectOverFlag = 1;
            }
            else if ( AC.AltCollectFun ) {
                AS.CollectOverFlag = 2;
            }
            break;
        }
        tstop = m + *m;
        *m -= same;
        m++;
        while ( r < tstop ) *m++ = *r++;
        if ( extra == 23 ) extra = ((AM.MaxTer/sizeof(WORD))/6);
    }
FullTerm:
    h[1] = WORDDIF(m,h);
    if ( AS.CollectOverFlag > 1 ) {
        *h = AC.AltCollectFun;
        if ( AS.CollectOverFlag == 3 ) AS.CollectOverFlag = 1;
    }
    else *h = AC.CollectFun;
    h[2] |= DIRTYFLAG;
    h[FUNHEAD] = h[1] - FUNHEAD;
    h[FUNHEAD+1] = 0;
    if ( ToFast(h+FUNHEAD,h+FUNHEAD) ) {
        if ( h[FUNHEAD] <= -FUNCTION ) {
            h[1] = FUNHEAD+1;
            m = h + FUNHEAD+1;
        }
        else {
            h[1] = FUNHEAD+2;
            m = h + FUNHEAD+2;
        }
    }
    *m++ = 1;
    *m++ = 1;
    *m++ = 3;
    *term = WORDDIF(m,term);
    AR.KeptInHold = 1;
    return(0);
}

/*
        #] GetMoreFromMem :
        #[ GetFromStore :

        Gets a single term from the storage file at position and puts
        it at 'to'.
        The value to be returned is the number of words read.
        Renumbering is done also.
        This is controled by the renumber table, given in 'renumber'

        This routine should work with a number of cache buffers. The
        exact number should be definable in form.set.
        The parameters are:
        AM.SizeStoreCache   (4096)
        The numbers are the proposed default values.

        The cache is a pure read cache.
*/

static int gfs = 0;

WORD GetFromStore(WORD *to, POSITION *position, RENUMBER renumber, WORD *InCompState, WORD nexpr)
{
    GETIDENTITY
```

```
    LONG RetCode, num, first = 0;
    WORD *from, *m;
    struct StOrEcAcHe sc;
    STORECACHE s;
    STORECACHE snext, sold;
    WORD *r, *rr = AR.CompressPointer;
    r = rr;
    gfs++;
    sc.next = AT.StoreCache;
    sold = s = &sc;
    snext = s->next;
    while ( snext ) {
        sold = s;
        s = snext;
        snext = s->next;
        if ( BASEPOSITION(s->position) == -1 ) break;
        if ( ISLESSPOS(*position,s->toppos) &&
        ISGEPOS(*position,s->position) ) {  /* Hit */
            if ( AT.StoreCache != s ) {
                sold->next = s->next;
                s->next = AT.StoreCache->next;
                AT.StoreCache = s;
            }
            from = (WORD *)(((UBYTE *)(s->buffer)) + DIFBASE(*position,s->position));
            num = *from;
            if ( !num ) { return(*to = 0); }
            *InCompState = (WORD)num;
            m = to;
            if ( num < 0 ) {
                from++;
                ADDPOS(*position,sizeof(WORD));
                *m++ = (WORD)(-num+1);
                r++;
                while ( ++num <= 0 ) *m++ = *r++;
                if ( ISLESSPOS(*position,s->toppos) ) {
                    num = *from++;
                    *to += (WORD)num;
                    ADDPOS(*position,sizeof(WORD));
                    *InCompState = (WORD)(num + 2);
                }
                else {
                    first = 1;
                    goto InNew;
                }
            }
PastCon:;
            while ( num > 0 && ISLESSPOS(*position,s->toppos) ) {
                *r++ = *m++ = *from++; ADDPOS(*position,sizeof(WORD)); num--;
            }
            if ( num > 0 ) {
InNew:
                SETBASEPOSITION(s->position,-1);
                SETBASEPOSITION(s->toppos,-1);
                LOCK(AM.storefilelock);
                SeekFile(AR.StoreData.Handle,position,SEEK_SET);
                RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)(s->buffer),AM.SizeStoreCache);
                UNLOCK(AM.storefilelock);
                if ( RetCode < 0 ) goto PastErr;
                if ( !RetCode ) return( *to = 0 );
                s->position = *position;
                s->toppos = *position;
                ADDPOS(s->toppos,RetCode);
                from = s->buffer;
                if ( first ) {
                    num = *from++;
                    ADDPOS(*position,sizeof(WORD));
                    *to += (WORD)num;
/*                  first = 0; */
                    *InCompState = (WORD)(num + 2);
                }
                goto PastCon;
            }
            goto PastEnd;
        }
    }
    if ( AT.StoreCache ) {      /* Fill the last buffer */
        s->position = *position;
        LOCK(AM.storefilelock);
        SeekFile(AR.StoreData.Handle,position,SEEK_SET);
        RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)(s->buffer),AM.SizeStoreCache);
        UNLOCK(AM.storefilelock);
        if ( RetCode < 0 ) goto PastErr;
        if ( !RetCode ) return( *to = 0 );
        s->toppos = *position;
        ADDPOS(s->toppos,RetCode);
        if ( AT.StoreCache != s ) {
            sold->next = s->next;
```

```
            s->next = AT.StoreCache->next;
            AT.StoreCache = s;
        }
        m = to;
        from = s->buffer;
        num = *from;
        if ( !num ) { return( *to = 0 ); }
        *InCompState = (WORD)num;
        if ( num < 0 ) {
            *m++ = (WORD)(-num+1);
            r++;
            from++;
            ADDPOS(*position,sizeof(WORD));
            while ( ++num <= 0 ) *m++ = *r++;
            num = *from++;
            *to += (WORD)num;
            ADDPOS(*position,sizeof(WORD));
            *InCompState = (WORD)(num+2);
        }
        goto PastCon;
    }
/*      No caching available */
    LOCK(AM.storefilelock);
    SeekFile(AR.StoreData.Handle,position,SEEK_SET);
    RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)to,(LONG)sizeof(WORD));
    SeekFile(AR.StoreData.Handle,position,SEEK_CUR);
    UNLOCK(AM.storefilelock);
    if ( RetCode != sizeof(WORD) ) {
        *to = 0;
        return((WORD)RetCode);
    }
    if ( !*to ) return(0);
    m = to;
    if ( *to < 0 ) {
        num = *m++;
        *to = *r++ = (WORD)(-num + 1);
        while ( ++num <= 0 ) *m++ = *r++;
        LOCK(AM.storefilelock);
        SeekFile(AR.StoreData.Handle,position,SEEK_SET);
        RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)m,(LONG)sizeof(WORD));
        SeekFile(AR.StoreData.Handle,position,SEEK_CUR);
        UNLOCK(AM.storefilelock);
        if ( RetCode != sizeof(WORD) ) {
            MLOCK(ErrorMessageLock);
            MesPrint("@Error in compression of store file");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        num = *m;
        *to += (WORD)num;
        *InCompState = (WORD)(num + 2);
    }
    else {
        *InCompState = *to;
        num = *to - 1; m = to + 1; r = rr + 1;
    }
    first = num;
    num *= wsizeof(WORD);
    if ( num < 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("@Error in stored expressions file at position %9p",position);
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    LOCK(AM.storefilelock);
    SeekFile(AR.StoreData.Handle,position,SEEK_SET);
    RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)m,num);
    SeekFile(AR.StoreData.Handle,position,SEEK_CUR);
    UNLOCK(AM.storefilelock);
    if ( RetCode != num ) {
        MLOCK(ErrorMessageLock);
        MesPrint("@Error in stored expressions file at position %9p",position);
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    NCOPY(r,m,first);
PastEnd:
    *rr = *to;
    if ( r >= AR.ComprTop ) {
        MLOCK(ErrorMessageLock);
        MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    AR.CompressPointer = r; *r = 0;
    if ( !TermRenumber(to,renumber,nexpr) ) {
        MarkDirty(to,DIRTYSYMFLAG);
```

```
        if ( AR.CurDum > AM.IndDum && Expressions[nexpr].numdummies > 0 )
            MoveDummies(BHEAD to,AR.CurDum - AM.IndDum);
        return((WORD)*to);
    }
PastErr:
    MLOCK(ErrorMessageLock);
    MesCall("GetFromStore");
    MUNLOCK(ErrorMessageLock);
    SETERROR(-1)
}

/*
        #] GetFromStore :
        #[ DetVars :            VOID DetVars(term)

    Determines which variables are used in term.

    When par = 1 we are scanning a prototype expression which involves
    completely different rules.

*/

VOID DetVars(WORD *term, WORD par)
{
    GETIDENTITY
    WORD *stopper;
    WORD *t, sym;
    WORD *sarg;
    stopper = term + *term - 1;
    stopper = stopper - ABS(*stopper) + 1;
    term++;
    if ( par ) {        /* Prototype expression */
        WORD n;
        if ( ( n = NumSymbols ) > 0 ) {
            SYMBOLS tt;
            tt = symbols;
            do {
                (tt++)->flags &= ~INUSE;
            } while ( --n > 0 );
        }
        if ( ( n = NumIndices ) > 0 ) {
            INDICES tt;
            tt = indices;
            do {
                (tt++)->flags &= ~INUSE;
            } while ( --n > 0 );
        }
        if ( ( n = NumVectors ) > 0 ) {
            VECTORS tt;
            tt = vectors;
            do {
                (tt++)->flags &= ~INUSE;
            } while ( --n > 0 );
        }
        if ( ( n = NumFunctions ) > 0 ) {
            FUNCTIONS tt;
            tt = functions;
            do {
                (tt++)->flags &= ~INUSE;
            } while ( --n > 0 );
        }
        term += SUBEXPSIZE;
        while ( term < stopper ) {
            if ( *term == SYMTOSYM || *term == SYMTONUM ) {
                term += 2;
                AN.UsedSymbol[*term] = 1;
                symbols[*term].flags |= INUSE;
            }
            else if ( *term == VECTOVEC ) {
                term += 2;
                AN.UsedVector[*term-AM.OffsetVector] = 1;
                vectors[*term-AM.OffsetVector].flags |= INUSE;
            }
            else if ( *term == INDTOIND ) {
                term += 2;
                sym = indices[*term - AM.OffsetIndex].dimension;
                if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
                AN.UsedIndex[(*term) - AM.OffsetIndex] = 1;
                sym = indices[*term-AM.OffsetIndex].nmin4;
                if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
                indices[*term-AM.OffsetIndex].flags |= INUSE;
            }
            else if ( *term == FUNTOFUN ) {
                term += 2;
                AN.UsedFunction[*term-FUNCTION] = 1;
                functions[*term-FUNCTION].flags |= INUSE;
            }
```

```
            term += 2;
        }
    }
    else {
        while ( term < stopper ) {
            t = term + term[1];
            if ( *term == SYMBOL ) {
                term += 2;
                do {
                    AN.UsedSymbol[*term] = 1;
                    term += 2;
                } while ( term < t );
            }
            else if ( *term == DOTPRODUCT ) {
                term += 2;
                do {
                    AN.UsedVector[(*term++) - AM.OffsetVector] = 1;
                    AN.UsedVector[(*term) - AM.OffsetVector] = 1;
                    term += 2;
                } while ( term < t );
            }
            else if ( *term == VECTOR ) {
                term += 2;
                do {
                    AN.UsedVector[(*term++) - AM.OffsetVector] = 1;
                    if ( *term >= AM.OffsetIndex && *term < AM.DumInd ) {
                        sym = indices[*term - AM.OffsetIndex].dimension;
                        if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
                        AN.UsedIndex[*term - AM.OffsetIndex] = 1;
                        sym = indices[(*term++)-AM.OffsetIndex].nmin4;
                        if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
                    }
                    else term++;
                } while ( term < t );
            }
            else if ( *term == INDEX || *term == LEVICIVITA || *term == GAMMA
            || *term == DELTA ) {
/*
Tensors:
                term += 2;
*/
                if ( *term == INDEX || *term == DELTA ) term += 2;
                else {
Tensors:
                    term += FUNHEAD;
                }
                while ( term < t ) {
                    if ( *term  >= AM.OffsetIndex && *term < AM.DumInd ) {
                        sym = indices[*term - AM.OffsetIndex].dimension;
                        if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
                        AN.UsedIndex[(*term) - AM.OffsetIndex] = 1;
                        sym = indices[*term-AM.OffsetIndex].nmin4;
                        if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
                    }
                    else if ( *term < (WILDOFFSET+AM.OffsetVector) )
                        AN.UsedVector[(*term) - AM.OffsetVector] = 1;
                    term++;
                }
            }
            else if ( *term == HAAKJE ) term = t;
            else {
                if ( *term > MAXBUILTINFUNCTION )
                    AN.UsedFunction[(*term)-FUNCTION] = 1;
                if ( *term >= FUNCTION && functions[*term-FUNCTION].spec
                >= TENSORFUNCTION && term[1] > FUNHEAD ) goto Tensors;
                term += FUNHEAD;            /* First argument */
                while ( term < t ) {
                    sarg = term;
                    NEXTARG(sarg)
                    if ( *term > 0 ) {
                        sarg = term + *term;    /* End of argument */
                        term += ARGHEAD;        /* First term in argument */
                        if ( term < sarg ) { do {
                            DetVars(term,par);
                            term += *term;
                        } while ( term < sarg ); }
                    }
                    else {
                        if ( *term < -MAXBUILTINFUNCTION ) {
                            AN.UsedFunction[-*term-FUNCTION] = 1;
                        }
                        else if ( *term == -SYMBOL ) {
                            AN.UsedSymbol[term[1]] = 1;
                        }
                        else if ( *term == -INDEX ) {
                            if ( term[1] < (WILDOFFSET+AM.OffsetVector) ) {
                                AN.UsedVector[term[1]-AM.OffsetVector] = 1;
```

```
                            }
                            else if ( term[1] >= AM.OffsetIndex && term[1] < AM.DumInd ) {
                                sym = indices[term[1] - AM.OffsetIndex].dimension;
                                if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
                                AN.UsedIndex[term[1] - AM.OffsetIndex] = 1;
                                sym = indices[term[1]-AM.OffsetIndex].nmin4;
                                if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
                            }
                        }
                        else if ( *term == -VECTOR || *term == -MINVECTOR ) {
                            AN.UsedVector[term[1]-AM.OffsetVector] = 1;
                        }
                    }
                    term = sarg;            /* Next argument */
                }
                term = t;
            }
        }
    }
}

/*
        #] DetVars :
        #[ ToStorage :

    This routine takes an expression in the scratch buffer (indicated by e)
    and puts it in the storage file. The necessary actions are:

    1:  determine the list of the used variables.
    2:  make an index entry.
    3:  write the namelists.
    4:  copy the 'length' bytes of the expression.

*/

WORD ToStorage(EXPRESSIONS e, POSITION *length)
{
    GETIDENTITY
    WORD *w, i, j;
    WORD *term;
    INDEXENTRY *indexent;
    LONG size;
    POSITION indexpos, scrpos;
    FILEHANDLE *f;
    if ( ( indexent = NextFileIndex(&indexpos) ) == 0 ) {
        MesCall("ToStorage");
        SETERROR(-1)
    }
    indexent->CompressSize = 0;     /* thus far no compression */
    f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
    if ( e->status == HIDDENGEXPRESSION ) {
        AR.InHiBuf = 0; f = AR.hidefile; AR.GetFile = 2;
    }
    else {
        AR.InInBuf = 0; f = AR.infile;   AR.GetFile = 0;
    }
    if ( f->handle >= 0 ) {
        scrpos = e->onfile;
        SeekFile(f->handle,&scrpos,SEEK_SET);
        if ( ISNOTEQUALPOS(scrpos,e->onfile) ) {
            MesPrint(":::Error in Scratch file");
            goto ErrReturn;
        }
        f->POposition = e->onfile;
        f->POfull = f->PObuffer;
        if ( e->status == HIDDENGEXPRESSION ) AR.InHiBuf = 0;
        else                                  AR.InInBuf = 0;
    }
    else {
        f->POfill = (WORD *)((UBYTE *)(f->PObuffer)+BASEPOSITION(e->onfile));
    }
    w = AT.WorkPointer;
    AN.UsedSymbol   = w;    w += NumSymbols;
    AN.UsedVector   = w;    w += NumVectors;
    AN.UsedIndex    = w;    w += NumIndices;
    AN.UsedFunction = w;    w += NumFunctions;
    term = w;
    w = (WORD *)(((UBYTE *)(w)) + AM.MaxTer);
    if ( w > AT.WorkTop ) {
        MesWork();
        goto ErrReturn;
    }
    w = AN.UsedSymbol;
    i = NumSymbols + NumVectors + NumIndices + NumFunctions;
    do { *w++ = 0; } while ( --i > 0 );
    if ( GetTerm(BHEAD term) > 0 ) {
        DetVars(term,1);
```

```
        if ( GetTerm(BHEAD term) ) {
            do { DetVars(term,0); } while ( GetTerm(BHEAD term) > 0 );
        }
    }
    j = 0;
    w = AN.UsedSymbol;
    i = NumSymbols;
    while ( --i >= 0 ) { if ( *w++ ) j++; }
    indexent->nsymbols = j;
/*  size = j * sizeof(struct SyMbOl);  */
    j = 0;
    w = AN.UsedIndex;
    i = NumIndices;
    while ( --i >= 0 ) { if ( *w++ ) j++; }
    indexent->nindices = j;
/*  size += j * sizeof(struct InDeX);  */
    j = 0;
    w = AN.UsedVector;
    i = NumVectors;
    while ( --i >= 0 ) { if ( *w++ ) j++; }
    indexent->nvectors = j;
/*  size += j * sizeof(struct VeCtOr);  */
    j = 0;
    w = AN.UsedFunction;
    i = NumFunctions;
    while ( --i >= 0 ) { if ( *w++ ) j++; }
    indexent->nfunctions = j;
/*  size += j * sizeof(struct FuNcTiOn); */
    indexent->length = *length;
    indexent->variables = AR.StoreData.Fill;
/*  indexent->position = AR.StoreData.Fill + size; */
    StrCopy(AC.exprnames->namebuffer+e->name,(UBYTE *)(indexent->name));
    SeekFile(AR.StoreData.Handle,&(AR.StoreData.Fill),SEEK_SET);
    AO.wlen = 100000;
    AO.wpos = (UBYTE *)Malloc1(AO.wlen,"AO.wpos buffer");
    AO.wpoin = AO.wpos;
    {
         SYMBOLS a;
        w = AN.UsedSymbol;
        a = symbols;
        j = 0;
        i = indexent->nsymbols;
        while ( --i >= 0 ) {
            while ( !*w ) { w++; a++; j++; }
            a->number = j;
            if ( VarStore((UBYTE *)a,(WORD)(sizeof(struct SyMbOl)),a->name,
                a->namesize) ) goto ErrToSto;
            w++; j++; a++;
        }
    }
    {
         INDICES a;
        w = AN.UsedIndex;
        a = indices;
        j = 0;
        i = indexent->nindices;
        while ( --i >= 0 ) {
            while ( !*w ) { w++; a++; j++; }
            a->number = j;
            if ( VarStore((UBYTE *)a,(WORD)(sizeof(struct InDeX)),a->name,
                a->namesize) ) goto ErrToSto;
            w++; j++; a++;
        }
    }
    {
         VECTORS a;
        w = AN.UsedVector;
        a = vectors;
        j = 0;
        i = indexent->nvectors;
        while ( --i >= 0 ) {
            while ( !*w ) { w++; a++; j++; }
            a->number = j;
            if ( VarStore((UBYTE *)a,(WORD)(sizeof(struct VeCtOr)),a->name,
                a->namesize) ) goto ErrToSto;
            w++; j++; a++;
        }
    }
    {
         FUNCTIONS a;
        w = AN.UsedFunction;
        a = functions;
        j = 0;
        i = indexent->nfunctions;
        while ( --i >= 0 ) {
            while ( !*w ) { w++; a++; j++; }
            a->number = j;
```

```
            if ( VarStore((UBYTE *)a,(WORD)(sizeof(struct FuNcTiOn)),a->name,
                a->namesize) ) goto ErrToSto;
            w++; a++; j++;
        }
    }
    if ( VarStore((UBYTE *)0L,(WORD)0,(WORD)0,(WORD)0) ) goto ErrToSto; /* Flush buffer */
    TELLFILE(AR.StoreData.Handle,&(indexent->position));
    indexent->size = (WORD)DIFBASE(indexent->position,indexent->variables);
/*
        The following code was added when it became apparent (30-jan-2007)
        that we need provisions for extra space without upsetting existing
        .sav files. Here we can put as much as we want.
        Look in GetTable on how to recover numdummies.
        Forgetting numdummies has been in there from the beginning.
*/
    if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(e->numdummies)),(LONG)sizeof(WORD)) !=
        sizeof(WORD) ) {
            MesPrint("Error while writing storage file");
            goto ErrReturn;
    }
    if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(e->numfactors)),(LONG)sizeof(WORD)) !=
        sizeof(WORD) ) {
            MesPrint("Error while writing storage file");
            goto ErrReturn;
    }
    if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(e->vflags)),(LONG)sizeof(WORD)) !=
        sizeof(WORD) ) {
            MesPrint("Error while writing storage file");
            goto ErrReturn;
    }
    TELLFILE(AR.StoreData.Handle,&(indexent->position));
    if ( f->handle >= 0 ) {
        POSITION llength;
        llength = *length;
        SeekFile(f->handle,&(e->onfile),SEEK_SET);
        while ( ISPOSPOS(llength) ) {
            SETBASEPOSITION(scrpos,AO.wlen);
            if ( ISLESSPOS(llength,scrpos) ) size = BASEPOSITION(llength);
            else                             size = AO.wlen;
            if ( ReadFile(f->handle,AO.wpos,size) != size ) {
                MesPrint("Error while reading scratch file");
                goto ErrReturn;
            }
            if ( WriteFile(AR.StoreData.Handle,AO.wpos,size) != size ) {
                MesPrint("Error while writing storage file");
                goto ErrReturn;
            }
            ADDPOS(llength,-size);
        }
    }
    else {
        WORD *ppp;
        ppp = (WORD *)((UBYTE *)(f->PObuffer) + BASEPOSITION(e->onfile));
        if ( WriteFile(AR.StoreData.Handle,(UBYTE *)ppp,BASEPOSITION(*length)) !=
        BASEPOSITION(*length) ) {
            MesPrint("Error while writing storage file");
            goto ErrReturn;
        }
    }
    ADD2POS(*length,indexent->position);
    e->onfile = indexpos;
/*
    AR.StoreData.Fill = SeekFile(AR.StoreData.Handle,&(AM.zeropos),SEEK_END);
*/
    AR.StoreData.Fill = *length;
    SeekFile(AR.StoreData.Handle,&(AR.StoreData.Fill),SEEK_SET);
    scrpos = AR.StoreData.Position;
    ADDPOS(scrpos,sizeof(POSITION));
    SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
    if ( WriteFile(AR.StoreData.Handle,((UBYTE *)&(AR.StoreData.Index.number))
        ,(LONG)(sizeof(POSITION))) != sizeof(POSITION) ) goto ErrInSto;
    SeekFile(AR.StoreData.Handle,&indexpos,SEEK_SET);
    if ( WriteFile(AR.StoreData.Handle,(UBYTE *)indexent,(LONG)(sizeof(INDEXENTRY))) !=
        sizeof(INDEXENTRY) ) goto ErrInSto;
    FlushFile(AR.StoreData.Handle);
    SeekFile(AR.StoreData.Handle,&(AC.StoreFileSize),SEEK_END);
    f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
    if ( AO.wpos ) M_free(AO.wpos,"AO.wpos buffer");
    AO.wpos = AO.wpoin = 0;
    return(0);
ErrToSto:
    MesPrint("---Error while storing namelists");
    goto ErrReturn;
ErrInSto:
    MesPrint("Error in storage");
ErrReturn:
    if ( AO.wpos ) M_free(AO.wpos,"AO.wpos buffer");
```

```
    AO.wpos = AO.wpoin = 0;
    f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
    return(-1);
}

/*
        #] ToStorage :
        #[ NextFileIndex :
*/

INDEXENTRY *NextFileIndex(POSITION *indexpos)
{
    GETIDENTITY
    INDEXENTRY *ind;
    int i, j = sizeof(FILEINDEX)/(sizeof(LONG));
    LONG *lo;
    if ( AR.StoreData.Handle <= 0 ) {
        if ( SetFileIndex() ) {
            MesCall("NextFileIndex");
            return(0);
        }
        SETBASEPOSITION(AR.StoreData.Index.number,1);
#ifdef SYSDEPENDENTSAVE
        SETBASEPOSITION(*indexpos,(2*sizeof(POSITION)));
#else
        SETBASEPOSITION(*indexpos,(2*sizeof(POSITION)+sizeof(STOREHEADER)));
#endif
        return(AR.StoreData.Index.expression);
    }
    while ( BASEPOSITION(AR.StoreData.Index.number) >= (LONG)(INFILEINDEX) ) {
        if ( ISNOTZEROPOS(AR.StoreData.Index.next) ) {
            SeekFile(AR.StoreData.Handle,&(AR.StoreData.Index.next),SEEK_SET);
            AR.StoreData.Position = AR.StoreData.Index.next;
            if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(&AR.StoreData.Index),(LONG)(sizeof(FILEINDEX))) !=
            (LONG)(sizeof(FILEINDEX)) ) goto ErrNextS;
        }
        else {
            PUTZERO(AR.StoreData.Index.number);
            SeekFile(AR.StoreData.Handle,&(AR.StoreData.Position),SEEK_SET);
            if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&(AR.StoreData.Fill)),(LONG)(sizeof(POSITION)))
            != (LONG)(sizeof(POSITION)) ) goto ErrNextS;
            PUTZERO(AR.StoreData.Index.next);
            SeekFile(AR.StoreData.Handle,&(AR.StoreData.Fill),SEEK_SET);
            AR.StoreData.Position = AR.StoreData.Fill;
            lo = (LONG *)(&AR.StoreData.Index);
            for ( i = 0; i < j; i++ ) *lo++ = 0;
            if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&AR.StoreData.Index),(LONG)(sizeof(FILEINDEX))) !=
            (LONG)(sizeof(FILEINDEX)) ) goto ErrNextS;
            ADDPOS(AR.StoreData.Fill,sizeof(FILEINDEX));
        }
    }
    *indexpos = AR.StoreData.Position;
    ADDPOS(*indexpos,(2*sizeof(POSITION)) +
        BASEPOSITION(AR.StoreData.Index.number) * sizeof(INDEXENTRY));
    ind = &AR.StoreData.Index.expression[BASEPOSITION(AR.StoreData.Index.number)];
    ADDPOS(AR.StoreData.Index.number,1);
    return(ind);
ErrNextS:
    MesPrint("Error in storage file");
    return(0);
}

/*
        #] NextFileIndex :
        #[ SetFileIndex :
*/

WORD SetFileIndex()
{
    GETIDENTITY
    int i, j = sizeof(FILEINDEX)/(sizeof(LONG));
    LONG *lo;
    if ( AR.StoreData.Handle < 0 ) {
        AR.StoreData.Handle = AC.StoreHandle;
        PUTZERO(AR.StoreData.Index.next);
        PUTZERO(AR.StoreData.Index.number);
#ifdef SYSDEPENDENTSAVE
        SETBASEPOSITION(AR.StoreData.Fill,sizeof(FILEINDEX));
#else
        if ( WriteStoreHeader(AR.StoreData.Handle) ) return(MesPrint("Error writing storage file header"));
        SETBASEPOSITION(AR.StoreData.Fill, (LONG)sizeof(FILEINDEX)+(LONG)sizeof(STOREHEADER));
#endif
        lo = (LONG *)(&AR.StoreData.Index);
```

```
        for ( i = 0; i < j; i++ ) *lo++ = 0;
        if ( WriteFile(AR.StoreData.Handle,(UBYTE *)(&AR.StoreData.Index),(LONG)(sizeof(FILEINDEX))) !=
        (LONG)(sizeof(FILEINDEX)) ) return(MesPrint("Error writing storage file"));
    }
    else {
        POSITION scrpos;
#ifdef SYSDEPENDENTSAVE
        PUTZERO(scrpos);
#else
        SETBASEPOSITION(scrpos, (LONG)(sizeof(STOREHEADER)));
#endif
        SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
        if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(&AR.StoreData.Index),(LONG)(sizeof(FILEINDEX))) !=
        (LONG)(sizeof(FILEINDEX)) ) return(MesPrint("Error reading storage file"));
    }
#ifdef SYSDEPENDENTSAVE
    PUTZERO(AR.StoreData.Position);
#else
    SETBASEPOSITION(AR.StoreData.Position, (LONG)(sizeof(STOREHEADER)));
#endif
    return(0);
}

/*
        #] SetFileIndex :
        #[ VarStore :

        The n -= sizeof(WORD); makes that the real length comes in the
        padding space, provided there is padding space (it seems so).
        The reading of the information assumes this is the case and hence
        things work....
*/

WORD VarStore(UBYTE *s, WORD n, WORD name, WORD namesize)
{
    GETIDENTITY
    UBYTE *t, *u;
    if ( s ) {
        n -= sizeof(WORD);
        t = (UBYTE *)AO.wpoin;
/*
        u = (UBYTE *)AT.WorkTop;
*/
        u = AO.wpos+AO.wlen;
        while ( n > 0 && t < u ) { *t++ = *s++; n--; }
        while ( t >= u ) {
            if ( WriteFile(AR.StoreData.Handle,AO.wpos,AO.wlen) != AO.wlen ) return(-1);
            t = AO.wpos;
            while ( n > 0 && t < u ) { *t++ = *s++; n--; }
        }
        s = AC.varnames->namebuffer + name;
        n = namesize;
        n += sizeof(void *)-1; n &= -(sizeof(void *));
        *((WORD *)t) = n;
        t += sizeof(WORD);
        while ( n > 0 && t < u ) {
            if ( namesize > 0 ) { *t++ = *s++; namesize--; }
            else { *t++ = 0; }
            n--;
        }
        while ( t >= u ) {
            if ( WriteFile(AR.StoreData.Handle,AO.wpos,AO.wlen) != AO.wlen ) return(-1);
            t = AO.wpos;
            while ( n > 0 && t < u ) {
                if ( namesize > 0 ) { *t++ = *s++; namesize--; }
                else { *t++ = 0; }
                n--;
            }
        }
        AO.wpoin = t;
    }
    else {
        LONG size;
        size = AO.wpoin - AO.wpos;
        if ( WriteFile(AR.StoreData.Handle,AO.wpos,size) != size ) return(-1);
        AO.wpoin = AO.wpos;
    }
    return(0);
}

/*
        #] VarStore :
        #[ TermRenumber :

        renumbers the variables inside term according to the information
        in struct renumber.
```

```
        The search is binary. This avoided having to read/write the
        expression twice when it was stored.

*/

WORD TermRenumber(WORD *term, RENUMBER renumber, WORD nexpr)
{
    WORD *stopper;
    WORD *t, *sarg, n;
    stopper = term + *term - 1;
    stopper = stopper - ABS(*stopper) + 1;
    term++;
    while ( term < stopper ) {
        if ( *term == SYMBOL ) {
            t = term + term[1];
            term += 2;
            do {
                if ( ( n = FindrNumber(*term,&(renumber->symb)) ) < 0 ) goto ErrR;
                *term = renumber->symnum[n];
                term += 2;
            } while ( term < t );
        }
        else if ( *term == DOTPRODUCT ) {
            t = term + term[1];
            term += 2;
            do {
                if ( ( n = FindrNumber(*term,&(renumber->vect)) )
                     < 0 ) goto ErrR;
                *term++ = renumber->vecnum[n];
                if ( ( n = FindrNumber(*term,&(renumber->vect)) )
                     < 0 ) goto ErrR;
                *term = renumber->vecnum[n];
                term += 2;
            } while ( term < t );
        }
        else if ( *term == VECTOR ) {
            t = term + term[1];
            term += 2;
            do {
                if ( ( n = FindrNumber(*term,&(renumber->vect)) )
                     < 0 ) goto ErrR;
                *term++ = renumber->vecnum[n];
                if ( ( *term >= AM.OffsetIndex ) && ( *term < AM.IndDum ) ) {
                    if ( ( n = FindrNumber(*term,&(renumber->indi)) )
                         < 0 ) goto ErrR;
                    *term++ = renumber->indnum[n];
                }
                else term++;
            } while ( term < t );
        }
        else if ( *term == INDEX || *term == LEVICIVITA || *term == GAMMA
        || *term == DELTA ) {
Tensors:
            t = term + term[1];
            if ( *term == INDEX || * term == DELTA ) term += 2;
            else term += FUNHEAD;
/*
            term += 2;
*/
            while ( term < t ) {
                if ( *term >= AM.OffsetIndex + WILDOFFSET ) {
/*
        Still TOBEDONE
*/
                }
                else if ( ( *term  >= AM.OffsetIndex ) && ( *term < AM.IndDum ) ) {
                    if ( ( n = FindrNumber(*term,&(renumber->indi)) )
                         < 0 ) goto ErrR;
                    *term = renumber->indnum[n];
                }
                else if ( *term < (WILDOFFSET+AM.OffsetVector) ) {
                    if ( ( n = FindrNumber(*term,&(renumber->vect)) )
                         < 0 ) goto ErrR;
                    *term = renumber->vecnum[n];
                }
                term++;
            }
        }
        else if ( *term == HAAKJE ) term += term[1];
        else {
            if ( *term > MAXBUILTINFUNCTION ) {
                if ( ( n = FindrNumber(*term,&(renumber->func)) )
                     < 0 ) goto ErrR;
                *term = renumber->funnum[n];
            }
            if ( *term >= FUNCTION && functions[*term-FUNCTION].spec
                    >= TENSORFUNCTION && term[1] > FUNHEAD ) goto Tensors;
```

```
            t = term + term[1];         /* General stopper */
            term += FUNHEAD;            /* First argument */
            while ( term < t ) {
                sarg = term;
                NEXTARG(sarg)
                if ( *term > 0 ) {
/*
                    Problem here:
                    Marking the argument as dirty attacks the heap
                    very heavily and costs much computer time.
*/
                    *++term = 1;
                    term += ARGHEAD-1;
                    while ( term < sarg ) {
                        if ( TermRenumber(term,renumber,nexpr) ) goto ErrR;
                        term += *term;
                    }
                }
                else {
                    if ( *term <= -MAXBUILTINFUNCTION ) {
                        if ( ( n = FindrNumber(-*term,&(renumber->func)) )
                         < 0 ) goto ErrR;
                        *term = -renumber->funnum[n];
                    }
                    else if ( *term == -SYMBOL ) {
                        term++;
                        if ( ( n = FindrNumber(*term,
                            &(renumber->symb)) ) < 0 ) goto ErrR;
                        *term = renumber->symnum[n];
                    }
                    else if ( *term == -INDEX ) {
                        term++;
                        if ( *term >= AM.OffsetIndex + WILDOFFSET ) {
/*
            Still TOBEDONE
*/
                        }
                        else if ( ( *term  >= AM.OffsetIndex ) && ( *term < AM.IndDum ) ) {
                            if ( ( n = FindrNumber(*term,&(renumber->indi)) )
                                 < 0 ) goto ErrR;
                            *term = renumber->indnum[n];
                        }
                        else if ( *term < (WILDOFFSET+AM.OffsetVector) ) {
                            if ( ( n = FindrNumber(*term,&(renumber->vect)) )
                                 < 0 ) goto ErrR;
                            *term = renumber->vecnum[n];
                        }
                    }
                    else if ( *term == -VECTOR || *term == -MINVECTOR ) {
                        term++;
                        if ( ( n = FindrNumber(*term,&(renumber->vect)) )
                         < 0 ) goto ErrR;
                        *term = renumber->vecnum[n];
                    }
                }
                term = sarg;            /* Next argument */
            }
            term = t;
        }
    }
    return(0);
ErrR:
    MesCall("TermRenumber");
    SETERROR(-1)
}

/*
        #] TermRenumber :
        #[ FindrNumber :
*/

WORD FindrNumber(WORD n, VARRENUM *v)
{
    WORD *hi,*med,*lo;
    hi = v->hi;
    lo = v->lo;
    med = v->start;
    if ( *hi == 0 ) {
        if ( n != *hi ) {
            MesPrint("Serious problems coming up in FindrNumber");
            return(-1);
        }
        return(*hi);
    }
    while ( *med != n ) {
        if ( *med < n ) {
            if ( med == hi ) goto ErrFindr;
```

```
            lo = med;
            med = hi - ((WORDDIF(hi,med))/2);
        }
        else {
            if ( med == lo ) goto ErrFindr;
            hi = med;
            med = lo + ((WORDDIF(med,lo))/2);
        }
    }
    return(WORDDIF(med,v->lo));
ErrFindr:
/*
    Reconstruction:
*/
    {
        int i;
        i = WORDDIF(v->hi,v->lo);
        MesPrint("FindrNumber: n = %d, list has %d members",n,i);
        while ( i >= 0 ) {
            MesPrint("v->lo[%d] = %d",i,v->lo[i]); i--;
        }
        hi = v->hi;
        lo = v->lo;
        med = v->start;
        MesPrint("Start with %d,%d,%d",0,WORDDIF(med,v->lo),WORDDIF(hi,v->lo));
        while ( *med != n ) {
            if ( *med < n ) {
                if ( med == hi ) goto ErrFindr2;
                lo = med;
                med = hi - ((WORDDIF(hi,med))/2);
            }
            else {
                if ( med == lo ) goto ErrFindr2;
                hi = med;
                med = ((WORDDIF(med,lo))/2) + lo;
            }
            MesPrint("New: %d,%d,%d, *med =
%d",WORDDIF(lo,v->lo),WORDDIF(med,v->lo),WORDDIF(hi,v->lo),*med);
        }
    }
    return(WORDDIF(med,v->lo));
ErrFindr2:
    return(MesPrint("Renumbering problems"));
}

/*
        #] FindrNumber :
        #[ FindInIndex :

        Finds an expression in the storage index if it exists.
        If found it returns a pointer to the index entry, otherwise zero.
        par = 0     Search by address (--> f == &AR.StoreData, called by GetTable, CoSave )
        par = 1     Search by name    (--> f == &AO.SaveData,  called by CoLoad )

        When comparing parameter fields the parameters of the expression
        to be searched are in AT.TMaddr. This includes the primary expression
        and a possible FROMBRAC information. The FROMBRAC is always last.

        The parameter mode tells whether we should worry about arguments of
        a stored expression.
*/

INDEXENTRY *FindInIndex(WORD expr, FILEDATA *f, WORD par, WORD mode)
{
    GETIDENTITY
    INDEXENTRY *ind;
    WORD i, hand, *m;
    WORD *start, *stop, *stop2, *m2, nomatch = 0;
    POSITION stindex, indexpos, scrpos;
    LONG number, num;
    stindex = f->Position;
    m = AT.TMaddr;
    stop = m + m[1];
    m += SUBEXPSIZE;
    start = m;
    while ( m < stop ) {
        if ( *m == FROMBRAC || *m == WILDCARDS ) break;
        m += m[1];
    }
    stop = m;
    if ( !par ) hand = AR.StoreData.Handle;
    else        hand = AO.SaveData.Handle;
    for(;;) {
        if ( ( i = (WORD)BASEPOSITION(f->Index.number) ) != 0 ) {
            indexpos = f->Position;
            ADDPOS(indexpos,(2*sizeof(POSITION)));
            ind = f->Index.expression;
```

```
            do {
                if ( ( !par && ISEQUALPOS(indexpos,Expressions[expr].onfile) )
                || ( par && !StrCmp(EXPRNAME(expr),(UBYTE *)(ind->name)) ) ) {
                    nomatch = 1;
/*
MesPrint("index: position: %8p",&(ind->position));
MesPrint("index: length: %8p",&(ind->length));
MesPrint("index: variables: %8p",&(ind->variables));
MesPrint("index: nsymbols: %d",ind->nsymbols);
MesPrint("index: nindices: %d",ind->nindices);
MesPrint("index: nvectors: %d",ind->nvectors);
MesPrint("index: nfunctions: %d",ind->nfunctions);
MesPrint("index: size: %d",ind->size);
*/
                    if ( par ) return(ind);
                    scrpos = ind->position;
                    SeekFile(hand,&scrpos,SEEK_SET);
                    if ( ISNOTEQUALPOS(scrpos,ind->position) ) goto ErrGt2;
                    if ( ReadFile(hand,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
                    sizeof(WORD) || !*AT.WorkPointer ) goto ErrGt2;
                    num = *AT.WorkPointer - 1;
                    num *= wsizeof(WORD);
                    if ( *AT.WorkPointer < 0 ||
                    ReadFile(hand,(UBYTE *)(AT.WorkPointer+1),num) != num ) goto ErrGt2;
                    m = start;  /* start of parameter field to be searched */
                    m2 = AT.WorkPointer + 1;
                    stop2 = m2 + m2[1];
                    m2 += SUBEXPSIZE;
                    while ( m < stop && m2 < stop2 ) {
                        if ( *m == SYMBOL ) {
                            if ( *m2 != SYMTOSYM ) break;
                            m2[3] = m[2];
                        }
                        else if ( *m == INDEX ) {
                            if ( m[2] >= 0 ) {
                                if ( *m2 != INDTOIND ) break;
                            }
                            else {
                                if ( *m2 != VECTOVEC ) break;
                            }
                            m2[3] = m[2];
                        }
                        else if ( *m >= FUNCTION ) {
                            if ( *m2 != FUNTOFUN ) break;
                            m2[3] = *m;
                        }
                        else {}
                        m += m[1];
                        m2 += m2[1];
                    }
                    if ( ( m >= stop && m2 >= stop2 ) || mode == 0 ) {
                        AT.WorkPointer = stop2;

                        return(ind);
                    }
                }
                ind++;
                ADDPOS(indexpos,sizeof(INDEXENTRY));
            } while ( --i > 0 );
        }
        f->Position = f->Index.next;
#ifndef SYSDEPENDENTSAVE
        if ( !ISNOTZEROPOS(f->Position) ) ADDPOS(f->Position,sizeof(STOREHEADER));
        number = sizeof(struct FiLeInDeX);
#endif
        if ( ISEQUALPOS(f->Position,stindex) && !AO.bufferedInd ) goto ErrGetTab;
        if ( !par ) {
            SeekFile(AR.StoreData.Handle,&(f->Position),SEEK_SET);
            if ( ISNOTEQUALPOS(f->Position,AR.StoreData.Position) ) goto ErrGt2;
#ifndef SYSDEPENDENTSAVE
            if ( ReadFile(f->Handle, (UBYTE *)(&(f->Index)), number) != number ) goto ErrGt2;
#endif
        }
        else {
            SeekFile(AO.SaveData.Handle,&(f->Position),SEEK_SET);
            if ( ISNOTEQUALPOS(f->Position,AO.SaveData.Position) ) goto ErrGt2;
#ifndef SYSDEPENDENTSAVE
            if ( ReadSaveIndex(&f->Index) ) goto ErrGt2;
#endif
        }
#ifdef SYSDEPENDENTSAVE
        number = sizeof(struct FiLeInDeX);
        if ( ReadFile(f->Handle,(UBYTE *)(&(f->Index)),number) !=
        number ) goto ErrGt2;
#endif
    }
ErrGetTab:
```

```
    if ( nomatch ) {
        MesPrint("Parameters of expression %s don't match."
        ,EXPRNAME(expr));
    }
    else {
        MesPrint("Cannot find expression %s",EXPRNAME(expr));
    }
    return(0);
ErrGt2:
    MesPrint("Readerror in IndexSearch");
    return(0);
}

/*
        #] FindInIndex :
        #[ GetTable :

        Locates stored files and constructs the renumbering tables.
        They are allocated in the WorkSpace.
        First the expression data are located. The Index is treated
        as a circularly linked buffer which is paged forwardly.
        If the indexentry is located (in ind) the two renumber tables
        have to be constructed.
        Finally the prototype has to be put in the proper buffer, so
        that wildcards can be passed. There should be a test with
        an already existing prototype that is constructed by the
        pattern matcher. This has not been put in yet.

        There is a problem with the parallel processing.
        Feeding in the variables that were erased by a .store could in
        principle happen in different orders (ParFORM) or simultaneously
        (TFORM). The proper resolution is to have the compiler call GetTable
        when a stored expression is encountered.

        This has been mended in development of TFORM by reading the
        symbol tables during compilation. See the call to GetTable
        in the CodeGenerator.

        Next is the problem of FindInIndex which writes in AR.StoreData
        Copying this is expensive!

        This Doesn't work well for TFORM yet.!!!!!!!!
        e[x1,x2] versus e[x2,x1] messes up.
        For the rest is the reloading during execution not thread safe.

        The parameter mode tells whether we should worry about arguments
        of a stored expression.
*/

RENUMBER GetTable(WORD expr, POSITION *position, WORD mode)
{
    GETIDENTITY
    WORD i, j;
    WORD *w;
    RENUMBER r;
    LONG num, nsize, xx;
    WORD jsym, jind, jvec, jfun;
    WORD k, type, error = 0, *oldw, *neww, *oldwork = AT.WorkPointer;
    struct SyMbOl SyM;
    struct InDeX InD;
    struct VeCtOr VeC;
    struct FuNcTiOn FuN;
    INDEXENTRY *ind;
/*
    Prepare for FindInIndex to put the prototype in the WorkSpace.
    oldw will point at the "wildcards"
*/
/*
        Bug fix. Look also in Generator.
#ifndef WITHPTHREADS

    if ( ( r = Expressions[expr].renum ) != 0 ) { }
    else {
        Expressions[expr].renum =
        r = (RENUMBER)Malloc1(sizeof(struct ReNuMbEr),"Renumber");
    }
#else
        r = (RENUMBER)Malloc1(sizeof(struct ReNuMbEr),"Renumber");
#endif
*/
        r = (RENUMBER)Malloc1(sizeof(struct ReNuMbEr),"Renumber");

    oldw = AT.WorkPointer + 1 + SUBEXPSIZE;
/*
    The protoype is loaded in the WorkSpace by the Index routine.
    After all it has to find an occurrence with the proper arguments.
    This sets the WorkPointer. Hence be careful now.
```

```
*/
    LOCK(AM.storefilelock);
    if ( ( ind = FindInIndex(expr,&AR.StoreData,0,mode) ) == 0 ) {
        UNLOCK(AM.storefilelock);
        return(0);
    }

    xx = ind->nsymbols+ind->nindices+ind->nvectors+ind->nfunctions;
    if ( xx == 0 ) {
        Expressions[expr].renumlists =
        w = AN.dummyrenumlist;
    }
    else {
/*
#ifndef WITHPTHREADS
        Expressions[expr].renumlists =
#endif
*/
        w = (WORD *)Malloc1(sizeof(WORD)*(xx*2),"VarSpace");
    }
    r->symb.lo = w;
    r->symb.start = w + ind->nsymbols/2;
    w += ind->nsymbols;
    r->symb.hi = w - 1;
    r->symnum = w;
    w += ind->nsymbols;

    r->indi.lo = w;
    r->indi.start = w + ind->nindices/2;
    w += ind->nindices;
    r->indi.hi = w - 1;
    r->indnum = w;
    w += ind->nindices;

    r->vect.lo = w;
    r->vect.start = w + ind->nvectors/2;
    w += ind->nvectors;
    r->vect.hi = w - 1;
    r->vecnum = w;
    w += ind->nvectors;

    r->func.lo = w;
    r->func.start = w + ind->nfunctions/2;
    w += ind->nfunctions;
    r->func.hi = w - 1;
    r->funnum = w;
/*  w += ind->nfunctions; */

    SeekFile(AR.StoreData.Handle,&(ind->variables),SEEK_SET);
    *position = ind->position;
    jsym = ind->nsymbols;
    jvec = ind->nvectors;
    jind = ind->nindices;
    jfun = ind->nfunctions;
/*
            #[ Symbols :
*/
    {
    SYMBOLS s = &SyM;
    w = r->symb.lo; j = jsym;
    for ( i = 0; i < j; i++ ) {
        if ( ReadFile(AR.StoreData.Handle,(UBYTE *)s,(LONG)(sizeof(struct SyMbOl)))
        != sizeof(struct SyMbOl) ) goto ErrGt2;
        nsize = s->namesize; nsize += sizeof(void *)-1;
        nsize &= -sizeof(void *);
        if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer),nsize)
        != nsize ) goto ErrGt2;
        *w = s->number;
        if ( ( s->flags & INUSE ) != 0 ) {
            /* Find the replacement. It must exist! */
            neww = oldw;
            while ( *neww != SYMTOSYM || neww[2] != *w ) neww += neww[1];
            k = neww[3];
        }
        else if ( GetVar((UBYTE *)AT.WorkPointer,&type,&k,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
            if ( type != CSYMBOL ) {
                MesPrint("Error: Conflicting types for %s",(AT.WorkPointer));
                error = -1;
            }
            else {
                if ( ( s->complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) !=
                ( symbols[k].complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) ) {
                    MesPrint("Warning: Conflicting complexity for %s",AT.WorkPointer);
                    error = -1;
                }
                if ( ( s->complex & (VARTYPEROOTOFUNITY) ) !=
                ( symbols[k].complex & (VARTYPEROOTOFUNITY) ) ) {
```

```
                    MesPrint("Warning: Conflicting root of unity properties for %s",AT.WorkPointer);
                    error = -1;
                }
                if ( ( s->complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
                    if ( s->maxpower != symbols[k].maxpower ) {
                        MesPrint("Warning: Conflicting n in n-th root of unity properties for %s",AT.WorkPointer);
                        error = -1;
                    }
                }
                else if ( ( s->minpower !=
                symbols[k].minpower || s->maxpower !=
                symbols[k].maxpower ) && AC.WarnFlag ) {
                    MesPrint("Warning: Conflicting power restrictions for %s",AT.WorkPointer);
                }
            }
        }
        else {
            if ( ( k = EntVar(CSYMBOL,(UBYTE *)(AT.WorkPointer),s->complex,s->minpower,
            s->maxpower,s->dimension) ) < 0 ) goto GetTcall;
        }
        *(w+j) = k;
        w++;
    }
    }
/*
            #] Symbols :
            #[ Indices :
*/
    {
    INDICES s = &InD;
    w = r->indi.lo; j = jind;
    for ( i = 0; i < j; i++ ) {
        if ( ReadFile(AR.StoreData.Handle,(UBYTE *)s,(LONG)(sizeof(struct InDeX)))
        != sizeof(struct InDeX) ) goto ErrGt2;
        nsize = s->namesize; nsize += sizeof(void *)-1;
        nsize &= -sizeof(void *);
        if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer),nsize)
        != nsize ) goto ErrGt2;
        *w = s->number + AM.OffsetIndex;
        if ( s->dimension < 0 ) {   /* Relabel the dimension */
            s->dimension = -r->symnum[FindrNumber(-s->dimension,&(r->symb))];
            if ( s->nmin4 < -NMIN4SHIFT ) { /* Relabel n-4 */
                s->nmin4 = -r->symnum[FindrNumber(-s->nmin4-NMIN4SHIFT
                ,&(r->symb))]-NMIN4SHIFT;
            }
        }
        if ( ( s->flags & INUSE ) != 0 ) {
            /* Find the replacement. It must exist! */
            neww = oldw;
            while ( *neww != INDTOIND || neww[2] != *w ) neww += neww[1];
            k = neww[3] - AM.OffsetIndex;
        }
        else if ( s->type == DUMMY ) {
/*
-------->           Here we may have to execute some renumbering
*/
        }
        else if ( GetVar((UBYTE *)(AT.WorkPointer),&type,&k,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
            if ( type != CINDEX ) {
                MesPrint("Error: Conflicting types for %s",(AT.WorkPointer));
                error = -1;
            }
            else {
                if ( s->type !=
                indices[k].type ) {
                    MesPrint("Warning: %s is also a dummy index",(AT.WorkPointer));
                    error = -1;
                    goto GetTb3;
                }
                if ( s->dimension != indices[k].dimension ) {
                    MesPrint("Warning: Conflicting dimensions for %s",(AT.WorkPointer));
                    error = -1;
                }
            }
        }
        else {
GetTb3:
            if ( ( k = EntVar(CINDEX,(UBYTE *)(AT.WorkPointer),
            s->dimension,0,s->nmin4,0) ) < 0 ) goto GetTcall;

        }
        *(w+j) = k + AM.OffsetIndex;
        w++;
    }
    }
/*
```

```
            #] Indices :
            #[ Vectors :
*/
    {
    VECTORS s = &VeC;
    w = r->vect.lo; j = jvec;
    for ( i = 0; i < j; i++ ) {
        if ( ReadFile(AR.StoreData.Handle,(UBYTE *)s,(LONG)(sizeof(struct VeCtOr)))
        != sizeof(struct VeCtOr) ) goto ErrGt2;
        nsize = s->namesize; nsize += sizeof(void *)-1;
        nsize &= -sizeof(void *);
        if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer),nsize)
        != nsize ) goto ErrGt2;
        *w = s->number + AM.OffsetVector;
        if ( ( s->flags & INUSE ) != 0 ) {
            /* Find the replacement. It must exist! */
            neww = oldw;
            while ( *neww != VECTOVEC || neww[2] != *w ) neww += neww[1];
            k = neww[3] - AM.OffsetVector;
        }
        else if ( GetVar((UBYTE *)(AT.WorkPointer),&type,&k,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
            if ( type != CVECTOR ) {
                MesPrint("Error: Conflicting types for %s",(AT.WorkPointer));
                error = -1;
            }
            else {
                if ( ( ( s->complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) !=
                ( vectors[k].complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) ) {
                    MesPrint("Warning: Conflicting complexity for %s",(AT.WorkPointer));
                    error = -1;
                }
            }
        }
        else {
            if ( ( k = EntVar(CVECTOR,(UBYTE *)(AT.WorkPointer),
            s->complex,0,0,s->dimension) ) < 0 ) goto GetTcall;
        }
        *(w+j) = k + AM.OffsetVector;
        w++;
    }
    }
/*
            #] Vectors :
            #[ Functions :
*/
    {
    FUNCTIONS s = &FuN;
    w = r->func.lo; j = jfun;
    for ( i = 0; i < j; i++ ) {
        if ( ReadFile(AR.StoreData.Handle,(UBYTE *)s,(LONG)(sizeof(struct FuNcTiOn)))
        != sizeof(struct FuNcTiOn) ) goto ErrGt2;
        nsize = s->namesize; nsize += sizeof(void *)-1;
        nsize &= -sizeof(void *);
        if ( ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer),nsize)
        != nsize ) goto ErrGt2;
        *w = s->number + FUNCTION;
        if ( ( s->flags & INUSE ) != 0 ) {
            /* Find the replacement. It must exist! */
            neww = oldw;
            while ( *neww != FUNTOFUN || neww[2] != *w ) neww += neww[1];
            k = neww[3] - FUNCTION;
        }
        else if ( GetVar((UBYTE *)(AT.WorkPointer),&type,&k,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
            if ( type != CFUNCTION ) {
                MesPrint("Error: Conflicting types for %s",(AT.WorkPointer));
                error = -1;
            }
            else {
                if ( s->complex != functions[k].complex ) {
                    MesPrint("Warning: Conflicting complexity for %s",(AT.WorkPointer));
                    error = -1;
                }
                else if ( s->symmetric != functions[k].symmetric ) {
                    MesPrint("Warning: Conflicting symmetry properties for %s",(AT.WorkPointer));
                    error = -1;
                }
                else if ( ( s->maxnumargs != functions[k].maxnumargs )
                || ( s->minnumargs != functions[k].minnumargs ) ) {
                    MesPrint("Warning: Conflicting argument restriction properties for
%s",(AT.WorkPointer));
                    error = -1;
                }
            }
        }
        else {
            if ( ( k = EntVar(CFUNCTION,(UBYTE *)(AT.WorkPointer),
            s->complex,s->commute,s->spec,s->dimension) ) < 0 ) goto GetTcall;
```

```
            functions[k].symmetric = s->symmetric;
            functions[k].maxnumargs = s->maxnumargs;
            functions[k].minnumargs = s->minnumargs;
        }
        *(w+j) = k + FUNCTION;
        w++;
    }
    }
/*
            #] Functions :

    Now we skip the prototype. This sets the start position at the first term
*/
    if ( error ) {
        UNLOCK(AM.storefilelock);
        AT.WorkPointer = oldwork;
        return(0);
    }

    {
/*
        For clarity we look where we are.
        We want to know: is this position already known?
        Could we have inserted extra information here?

        nummystery indicates extra words. We have currently in order
        (if they exist)
            numdummies
            numfactors
            vflags
*/
        POSITION pos;
        int nummystery;
        TELLFILE(AR.StoreData.Handle,&pos);
        nummystery = DIFBASE(ind->position,pos);
/*
        MesPrint("--> We are at position       %8p",&pos);
        MesPrint("--> The index says        at %8p",&(ind->position));
        MesPrint("--> There are %d mystery bytes",nummystery);
*/
        if ( nummystery > 0 ) {
            if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
            sizeof(WORD) ) {
                UNLOCK(AM.storefilelock);
                AT.WorkPointer = oldwork;
                return(0);
            }
            Expressions[expr].numdummies = *AT.WorkPointer;
/*
            MesPrint("--> numdummies = %d",Expressions[expr].numdummies);
*/
            nummystery -= sizeof(WORD);
        }
        else {
            Expressions[expr].numdummies = 0;
        }
        if ( nummystery > 0 ) {
            if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
            sizeof(WORD) ) {
                UNLOCK(AM.storefilelock);
                AT.WorkPointer = oldwork;
                return(0);
            }
            if ( ( AS.OldNumFactors == 0 ) || ( AS.NumOldNumFactors < NumExpressions ) ) {
                WORD *buffer;
                int capacity = 20;
                if (capacity < NumExpressions) capacity = NumExpressions * 2;

                buffer = (WORD *)Malloc1(capacity * sizeof(WORD), "numfactors pointers");
                if (AS.OldNumFactors) {
                    WCOPY(buffer, AS.OldNumFactors, AS.NumOldNumFactors);
                    M_free(AS.OldNumFactors, "numfactors pointers");
                }
                AS.OldNumFactors = buffer;

                buffer = (WORD *)Malloc1(capacity * sizeof(WORD), "vflags pointers");
                if (AS.Oldvflags) {
                    WCOPY(buffer, AS.Oldvflags, AS.NumOldNumFactors);
                    M_free(AS.Oldvflags, "vflags pointers");
                }
                AS.Oldvflags = buffer;

                AS.NumOldNumFactors = capacity;
            }

            AS.OldNumFactors[expr] =
            Expressions[expr].numfactors = *AT.WorkPointer;
```

```
/*
            MesPrint("--> numfactors = %d",Expressions[expr].numfactors);
*/
            nummystery -= sizeof(WORD);
        }
        else {
            Expressions[expr].numfactors = 0;
        }
        if ( nummystery > 0 ) {
            if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
            sizeof(WORD) ) {
                UNLOCK(AM.storefilelock);
                AT.WorkPointer = oldwork;
                return(0);
            }
            AS.Oldvflags[expr] =
            Expressions[expr].vflags = *AT.WorkPointer;
/*
            MesPrint("--> vflags = %d",Expressions[expr].vflags);
*/
            nummystery -= sizeof(WORD);
        }
        else {
            Expressions[expr].vflags = 0;
        }
    }

    SeekFile(AR.StoreData.Handle,&(ind->position),SEEK_SET);
    if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,(LONG)sizeof(WORD)) !=
    sizeof(WORD) || !*AT.WorkPointer ) {
        UNLOCK(AM.storefilelock);
        AT.WorkPointer = oldwork;
        return(0);
    }
    num = *AT.WorkPointer - 1;
    num *= sizeof(WORD);
    if ( *AT.WorkPointer < 0 ||
    ReadFile(AR.StoreData.Handle,(UBYTE *)(AT.WorkPointer+1),num) != num ) {
        MesPrint("@Error in stored expressions file at position %10p",*position);
        UNLOCK(AM.storefilelock);
        AT.WorkPointer = oldwork;
        return(0);
    }
    UNLOCK(AM.storefilelock);
    ADDPOS(*position,num+sizeof(WORD));
    r->startposition = *position;
    AT.WorkPointer = oldwork;
    return(r);
GetTcall:
    UNLOCK(AM.storefilelock);
    AT.WorkPointer = oldwork;
    MesCall("GetTable");
    return(0);
ErrGt2:
    UNLOCK(AM.storefilelock);
    AT.WorkPointer = oldwork;
    MesPrint("Readerror in GetTable");
    return(0);
}

/*
        #] GetTable :
        #[ CopyExpression :

        Copies from one scratch buffer to another.
        We assume here that the complete 'from' scratch buffer is taken.
        We also assume that the 'from' buffer is positioned at the end of
        the expression.

        The locks should be placed in the calling routine. We need basically
        AS.outputslock.
*/

int CopyExpression(FILEHANDLE *from, FILEHANDLE *to)
{
    POSITION posfrom, poscopy;
    LONG fullsize,i;
    WORD *t1, *t2;
    int RetCode;
    SeekScratch(from,&posfrom);
    if ( from->handle < 0 ) { /* input is in memory */
        fullsize = (BASEPOSITION(posfrom))/sizeof(WORD);
        if ( ( to->POstop - to->POfull ) >= fullsize ) {
/*
            Fits inside the buffer of the output. This will be fast.
*/
            t1 = from->PObuffer;
```

```
            t2 = to->POfull;
            NCOPY(t2,t1,fullsize)
            to->POfull = to->POfill = t2;
            goto WriteTrailer;
        }
        if ( to->handle < 0 ) {       /* First open the file */
            if ( ( RetCode = CreateFile(to->name) ) >= 0 ) {
                to->handle = (WORD)RetCode;
                PUTZERO(to->filesize);
                PUTZERO(to->POposition);
            }
            else {
                MLOCK(ErrorMessageLock);
                MesPrint("Cannot create scratch file %s",to->name);
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
        }
        t1 = from->PObuffer;
        while ( fullsize > 0 ) {
            i = to->POstop - to->POfull;
            if ( i > fullsize ) i = fullsize;
            fullsize -= i;
            t2 = to->POfull;
            NCOPY(t2,t1,i)
            if ( fullsize > 0 ) {
                SeekFile(to->handle,&(to->POposition),SEEK_SET);
                if ( WriteFile(to->handle,((UBYTE *)(to->PObuffer)),to->POsize) != to->POsize ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Error while writing to disk. Disk full?");
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                ADDPOS(to->POposition,to->POsize);
/*              SeekFile(to->handle,&(to->POposition),SEEK_CUR); */
                to->filesize = to->POposition;
                to->POfill = to->POfull = to->PObuffer;
            }
            else {
                to->POfill = to->POfull = t2;
            }
        }
        goto WriteTrailer;
    }
/*
    Now the input involves a file. This needs the use of the PObuffer of from.
    First make sure the tail of the buffer has been written
*/
    if ( ((UBYTE *)(from->POfill)-(UBYTE *)(from->PObuffer)) > 0 ) {
        if ( WriteFile(from->handle,((UBYTE *)(from->PObuffer)),((UBYTE *)(from->POfill)-(UBYTE
    *)(from->PObuffer)))
        != ((UBYTE *)(from->POfill)-(UBYTE *)(from->PObuffer)) ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Error while writing to disk. Disk full?");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        SeekFile(from->handle,&(from->POposition),SEEK_CUR);
        posfrom = from->filesize = from->POposition;
        from->POfill = from->POfull = from->PObuffer;
    }
/*
    Now copy the complete contents
*/
    PUTZERO(poscopy);
    SeekFile(from->handle,&poscopy,SEEK_SET);
    while ( ISLESSPOS(poscopy,posfrom) ) {
        fullsize = ReadFile(from->handle,((UBYTE *)(from->PObuffer)),from->POsize);
        if ( fullsize < 0 || ( fullsize % sizeof(WORD) ) != 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Error while reading from disk while copying expression.");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        fullsize /= sizeof(WORD);
        from->POfull = from->PObuffer + fullsize;
        t1 = from->PObuffer;

        if ( ( to->POstop - to->POfull ) >= fullsize ) {
/*
            Fits inside the buffer of the output. This will be fast.
*/
            t2 = to->POfull;
            NCOPY(t2,t1,fullsize)
            to->POfill = to->POfull = t2;
        }
        else {
```

```
          if ( to->handle < 0 ) {     /* First open the file */
            if ( ( RetCode = CreateFile(to->name) ) >= 0 ) {
                to->handle = (WORD)RetCode;
                PUTZERO(to->POposition);
                PUTZERO(to->filesize);
            }
            else {
                MLOCK(ErrorMessageLock);
                MesPrint("Cannot create scratch file %s",to->name);
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
          }
          while ( fullsize > 0 ) {
            i = to->POstop - to->POfull;
            if ( i > fullsize ) i = fullsize;
            fullsize -= i;
            t2 = to->POfull;
            NCOPY(t2,t1,i)
            if ( fullsize > 0 ) {
                SeekFile(to->handle,&(to->POposition),SEEK_SET);
                if ( WriteFile(to->handle,((UBYTE *)(to->PObuffer)),to->POsize) != to->POsize ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Error while writing to disk. Disk full?");
                    MUNLOCK(ErrorMessageLock);
                    return(-1);
                }
                ADDPOS(to->POposition,to->POsize);
/*              SeekFile(to->handle,&(to->POposition),SEEK_CUR); */
                to->filesize = to->POposition;
                to->POfill = to->POfull = to->PObuffer;
            }
            else {
                to->POfill = to->POfull = t2;
            }
          }
        }
        SeekFile(from->handle,&poscopy,SEEK_CUR);
    }
WriteTrailer:
    if ( ( to->handle >= 0 ) && ( to->POfill > to->PObuffer ) ) {
        fullsize = (UBYTE *)(to->POfill) - (UBYTE *)(to->PObuffer);
/*
        PUTZERO(to->POposition);
        SeekFile(to->handle,&(to->POposition),SEEK_END);
*/
        SeekFile(to->handle,&(to->filesize),SEEK_SET);
        if ( WriteFile(to->handle,((UBYTE *)(to->PObuffer)),fullsize) != fullsize ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Error while writing to disk. Disk full?");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        ADDPOS(to->filesize,fullsize);
        to->POposition = to->filesize;
        to->POfill = to->POfull = to->PObuffer;
    }

    return(0);
}

/*
        #] CopyExpression :
        #[ ExprStatus :
*/

#ifdef HIDEDEBUG

static UBYTE *statusexpr[] = {
      (UBYTE *)"LOCALEXPRESSION"
     ,(UBYTE *)"SKIPLEXPRESSION"
     ,(UBYTE *)"DROPLEXPRESSION"
     ,(UBYTE *)"DROPPEDEXPRESSION"
     ,(UBYTE *)"GLOBALEXPRESSION"
     ,(UBYTE *)"SKIPGEXPRESSION"
     ,(UBYTE *)"DROPGEXPRESSION"
     ,(UBYTE *)"UNKNOWN"
     ,(UBYTE *)"STOREDEXPRESSION"
     ,(UBYTE *)"HIDDENLEXPRESSION"
     ,(UBYTE *)"HIDELEXPRESSION"
     ,(UBYTE *)"DROPHLEXPRESSION"
     ,(UBYTE *)"UNHIDELEXPRESSION"
     ,(UBYTE *)"HIDDENGEXPRESSION"
     ,(UBYTE *)"HIDEGEXPRESSION"
     ,(UBYTE *)"DROPHGEXPRESSION"
     ,(UBYTE *)"UNHIDEGEXPRESSION"
     ,(UBYTE *)"INTOHIDELEXPRESSION"
```

```
    ,(UBYTE *)"INTOHIDEGEXPRESSION"
};

void ExprStatus(EXPRESSIONS e)
{
    MesPrint("Expression %s(%d) has status %s(%d,%d). Buffer: %d, Position: %15p",
        AC.exprnames->namebuffer+e->name,(WORD)(e-Expressions),
        statusexpr[e->status],e->status,e->hidelevel,
        e->whichbuffer,&(e->onfile));
}

#endif

/*
        #] ExprStatus :
    #] StoreExpressions :
    #[ System Independent Saved Expressions :

    All functions concerned with the system independent reading of save-files
    are here. They are called by the functions CoLoad, PutInStore,
    SetFileIndex, FindInIndex. In case no translation (endianness flip,
    resizing of words, renumbering) has to be done, they just do simple file
    reading. The function SaveFileHeader() for writing a header with
    information about the system architecture, FORM version, etc. is also
    located here.

        #[ Flip :
*/

#ifndef INT16
#error "INT16 not defined!"
#endif
#ifndef INT32
#error "INT32 not defined!"
#endif

static void FlipN(UBYTE *p, int length)
{
    UBYTE *q, buf;
    q = p + length;
    do {
        --q;
        buf = *p; *p = *q; *q = buf;
    } while ( ++p != q );
}

static void Flip16(UBYTE *p)
{
    INT16 in = *((INT16 *)p);
    INT16 out = (INT16)( (((in) » 8) & 0x00FF) | (((in) « 8) & 0xFF00) );
    *((INT16 *)p) = out;
}

static void Flip32(UBYTE *p)
{
    INT32 in = *((INT32 *)p);
    INT32 out =
        ( (((in) » 24) & 0x000000FF) | (((in) »  8) & 0x0000FF00) | \
          (((in) «  8) & 0x00FF0000) | (((in) « 24) & 0xFF000000) );
    *((INT32 *)p) = out;
}

#ifdef INT64
static void Flip64(UBYTE *p)
{
    INT64 in = *((INT64 *)p);
    INT64 out =
        ( (((in) » 56) & (INT64)0x00000000000000FFLL) | (((in) » 40) & (INT64)0x000000000000FF00LL) | \
          (((in) » 24) & (INT64)0x0000000000FF0000LL) | (((in) »  8) & (INT64)0x00000000FF000000LL) | \
          (((in) «  8) & (INT64)0x000000FF00000000LL) | (((in) « 24) & (INT64)0x0000FF0000000000LL) | \
          (((in) « 40) & (INT64)0x00FF000000000000LL) | (((in) « 56) & (INT64)0xFF00000000000000LL) );
    *((INT64 *)p) = out;
}
#else
static void Flip64(UBYTE *p) { FlipN(p, 8); }
#endif /* INT64 */

static void Flip128(UBYTE *p) { FlipN(p, 16); }

/*
        #] Flip :
        #[ Resize :
*/

```

```
static void ResizeDataBE(UBYTE *src, int slen, UBYTE *dst, int dlen)
{
    if ( slen > dlen ) {
        src += slen - dlen;
        while ( dlen-- ) { *dst++ = *src++; }
    }
    else {
        int i = dlen - slen;
        while ( i-- ) { *dst++ = 0; }
        while ( slen-- ) { *dst++ = *src++; }
    }
}

static void ResizeDataLE(UBYTE *src, int slen, UBYTE *dst, int dlen)
{
    if ( slen > dlen ) {
        while ( dlen-- ) { *dst++ = *src++; }
    }
    else {
        int i = dlen - slen;
        while ( slen-- ) { *dst++ = *src++; }
        while ( i-- ) { *dst++ = 0; }
    }
}

static void Resize16t16(UBYTE *src, UBYTE *dst)
{
    *((INT16 *)dst) = *((INT16 *)src);
}

static void Resize16t32(UBYTE *src, UBYTE *dst)
{
    INT16 in = *((INT16 *)src);
    INT32 out = (INT32)in;
    *((INT32 *)dst) = out;
}

#ifdef INT64
static void Resize16t64(UBYTE *src, UBYTE *dst)
{
    INT16 in = *((INT16 *)src);
    INT64 out = (INT64)in;
    *((INT64 *)dst) = out;
}
#else
static void Resize16t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 2, dst, 8); }
#endif /* INT64 */

static void Resize16t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 2, dst, 16); }

static void Resize32t32(UBYTE *src, UBYTE *dst)
{
    *((INT32 *)dst) = *((INT32 *)src);
}

#ifdef INT64
static void Resize32t64(UBYTE *src, UBYTE *dst)
{
    INT32 in = *((INT32 *)src);
    INT64 out = (INT64)in;
    *((INT64 *)dst) = out;
}
#else
static void Resize32t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 4, dst, 8); }
#endif /* INT64 */

static void Resize32t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 4, dst, 16); }

#ifdef INT64
static void Resize64t64(UBYTE *src, UBYTE *dst)
{
    *((INT64 *)dst) = *((INT64 *)src);
}
#else
static void Resize64t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 8); }
#endif /* INT64 */

static void Resize64t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 16); }

static void Resize128t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 16); }

static void Resize32t16(UBYTE *src, UBYTE *dst)
{
    INT32 in = *((INT32 *)src);
    INT16 out = (INT16)in;
    if ( in > (1«15)-1 || in < -(1«15)+1 ) AO.resizeFlag |= 1;
    *((INT16 *)dst) = out;
```

```
}

static void Resize32t16NC(UBYTE *src, UBYTE *dst)
{
    INT32 in = *((INT32 *)src);
    INT16 out = (INT16)in;
    *((INT16 *)dst) = out;
}

#ifdef INT64
static void Resize64t16(UBYTE *src, UBYTE *dst)
{
    INT64 in = *((INT64 *)src);
    INT16 out = (INT16)in;
    if ( in > (1«15)-1 || in < -(1«15)+1 ) AO.resizeFlag |= 1;
    *((INT16 *)dst) = out;
}
static void Resize64t16NC(UBYTE *src, UBYTE *dst)
{
    INT64 in = *((INT64 *)src);
    INT16 out = (INT16)in;
    *((INT16 *)dst) = out;
}
#else
static void Resize64t16(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 2); }
static void Resize64t16NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 2); }
#endif /* INT64 */

#ifdef INT64
static void Resize64t32(UBYTE *src, UBYTE *dst)
{
    INT64 in = *((INT64 *)src);
    INT32 out = (INT32)in;
    if ( in > ((INT64)1«31)-1 || in < -((INT64)1«31)+1 ) AO.resizeFlag |= 1;
    *((INT32 *)dst) = out;
}
static void Resize64t32NC(UBYTE *src, UBYTE *dst)
{
    INT64 in = *((INT64 *)src);
    INT32 out = (INT32)in;
    *((INT32 *)dst) = out;
}
#else
static void Resize64t32(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 4); }
static void Resize64t32NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 4); }
#endif /* INT64 */

static void Resize128t16(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 2); }

static void Resize128t16NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 2); }

static void Resize128t32(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 4); }

static void Resize128t32NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 4); }

static void Resize128t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 8); }

static void Resize128t64NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 8); }

/*
        #] Resize :
        #[ CheckPower and RenumberVec :
*/

static void CheckPower32(UBYTE *p)
{
    if ( *((INT32 *)p) < -MAXPOWER ) {
        AO.powerFlag |= 0x01;
        *((INT32 *)p) = -MAXPOWER;
    }
    p += sizeof(INT32);
    if ( *((INT32 *)p) > MAXPOWER ) {
        AO.powerFlag |= 0x02;
        *((INT32 *)p) = MAXPOWER;
    }
}

static void RenumberVec32(UBYTE *p)
{
/*  INT32 wildoffset = *((INT32 *)AO.SaveHeader.wildoffset); */
    void *dummy = (void *)AO.SaveHeader.wildoffset;  /* to remove a warning about strict-aliasing rules in gcc */
    INT32 wildoffset = *(INT32 *)dummy;
    INT32 in = *((INT32 *)p);
    in = in + 2*wildoffset;
    in = in - 2*WILDOFFSET;
    *((INT32 *)p) = in;
```

```
}

/*
        #] CheckPower and RenumberVec :
        #[ ResizeCoeff :
*/

static void ResizeCoeff32(UBYTE **bout, UBYTE *bend, UBYTE *top)
{
    int i;
    INT32 sign;
    INT32 *in, *p;
    INT32 *out = (INT32 *)*bout;
    INT32 *end = (INT32 *)bend;

    if ( sizeof(WORD) == 2 ) {
        /* 4 -> 2 */
        INT32 len = (end - 1 - out) / 2;
        int zeros = 2;
        p = out + len - 1;

        if ( *p & 0xFFFF0000 ) --zeros;
        p += len;
        if ( *p & 0xFFFF0000 ) --zeros;

        in = end - 1;
        sign = ( *in-- > 0 ) ? 1 : -1;
        p = out + 4*len;
        if ( zeros == 2 ) p -= 2;
        out = p--;

        if ( zeros < 2 ) *p-- = *in » 16;
        *p-- = *in-- & 0x0000FFFF;
        for ( i = 1; i < len; ++i ) {
            *p-- = *in » 16;
            *p-- = *in-- & 0x0000FFFF;
        }
        if ( zeros < 2 ) *p-- = *in » 16;
        *p-- = *in-- & 0x0000FFFF;
        for ( i = 1; i < len; ++i ) {
            *p-- = *in » 16;
            *p-- = *in-- & 0x0000FFFF;
        }

        *out = (out - p) * sign;
        *bout = (UBYTE *)(out+1);

    }
    else {
        /* 2 -> 4 */
        INT32 len = (end - 1 - out) / 2;
        if ( len == 1 ) {
            *out = *(unsigned INT16 *)out;
            ++out;
            *out = *(unsigned INT16 *)out;
            ++out;
            ++out;
        }
        else {
            p = out;
            *out = *(unsigned INT16 *)out;
            in = out + 1;
            for ( i = 1; i < len; ++i ) {
                /* shift */
                *out = (unsigned INT32)(*(unsigned INT16 *)out)
                    + ((unsigned INT32)(*(unsigned INT16 *)in) « 16);
                ++in;
                if ( ++i == len ) break;
                /* copy */
                ++out;
                *out = *(unsigned INT16 *)in;
                ++in;
            }
            ++out;
            *out = *(unsigned INT16 *)in;
            ++in;
            for ( i = 1; i < len; ++i ) {
                /* shift */
                *out = (unsigned INT32)(*(unsigned INT16 *)out)
                    + ((unsigned INT32)(*(unsigned INT16 *)in) « 16);
                ++in;
                if ( ++i == len ) break;
                /* copy */
                ++out;
                *out = *(unsigned INT16 *)in;
                ++in;
            }
```

```
            ++out;
            if ( *in < 0 ) *out = -(out - p + 1);
            else *out = out - p + 1;
            ++out;
        }

        if ( out > (INT32 *)top ) {
            MesPrint("Error in resizing coefficient!");
        }

        *bout = (UBYTE *)out;
    }
}

/*
        #] ResizeCoeff :
        #[ WriteStoreHeader :
*/

#define SAVEREVISION 0x02

WORD WriteStoreHeader(WORD handle)
{
    /* template of the STOREHEADER */
    static STOREHEADER sh = {
        { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF }, /* store header mark */
        0, 0, 0, 0,                                         /* sizeof of WORD,LONG,POSITION,void* */
        { 0 },                                              /* endianness check number */
        0, 0, 0, 0,                                         /* sizeof variable structs */
        { 0 },                                              /* maxpower */
        { 0 },                                              /* wildoffset */
        SAVEREVISION,                                       /* revision */
        { 0 } };                                            /* reserved */
    int endian, i;

    /* if called for the first time ... */
    if ( sh.lenWORD == 0 ) {
        sh.lenWORD = sizeof(WORD);
        sh.lenLONG = sizeof(LONG);
        sh.lenPOS = sizeof(POSITION);
        sh.lenPOINTER = sizeof(void *);

        endian = 1;
        for ( i = 1; i < (int)sizeof(int); ++i ) {
            endian <<= 8;
            endian += i+1;
        }
        for ( i = 0; i < (int)sizeof(int); ++i ) sh.endianness[i] = ((char *)&endian)[i];

        sh.sSym = sizeof(struct SyMbOl);
        sh.sInd = sizeof(struct InDeX);
        sh.sVec = sizeof(struct VeCtOr);
        sh.sFun = sizeof(struct FuNcTiOn);

/*      *((WORD *)sh.maxpower) = MAXPOWER;
        *((WORD *)sh.wildoffset) = WILDOFFSET; */
        {
            WORD dumw[8];
            UBYTE *dummy;
            for ( i = 0; i < 8; i++ ) dumw[i] = 0;
            dummy = (UBYTE *)dumw;
            dumw[0] = (WORD)MAXPOWER;
            for ( i = 0; i < 16; i++ ) sh.maxpower[i] = dummy[i];
            dumw[0] = (WORD)WILDOFFSET;
            for ( i = 0; i < 16; i++ ) sh.wildoffset[i] = dummy[i];
        }
    }

    return ( WriteFile(handle,(UBYTE *)(&sh),(LONG)(sizeof(STOREHEADER)))
              != (LONG)(sizeof(STOREHEADER)) );
}

/*
        #] WriteStoreHeader :
        #[ CompactifySizeof :
*/

static unsigned int CompactifySizeof(unsigned int size)
{
    switch ( size ) {
        case  2: return 0;
        case  4: return 1;
        case  8: return 2;
        case 16: return 3;
        default: MesPrint("Error compactifying size.");
                 return 3;
    }
```

```c
}

/*
        #] CompactifySizeof :
        #[ ReadSaveHeader :
*/

WORD ReadSaveHeader()
{
    /* Read-only tables of function pointers for conversions. */
    static VOID (*flipJumpTable[4])(UBYTE *) =
        { Flip16, Flip32, Flip64, Flip128 };
    static VOID (*resizeJumpTable[4][4])(UBYTE *, UBYTE *) = /* "own x saved"-sizes  */
        { { Resize16t16,  Resize32t16,  Resize64t16,  Resize128t16  },
          { Resize16t32,  Resize32t32,  Resize64t32,  Resize128t32  },
          { Resize16t64,  Resize32t64,  Resize64t64,  Resize128t64  },
          { Resize16t128, Resize32t128, Resize64t128, Resize128t128 } };
    static VOID (*resizeNCJumpTable[4][4])(UBYTE *, UBYTE *) = /* "own x saved"-sizes  */
        { { Resize16t16,  Resize32t16NC,  Resize64t16NC,  Resize128t16NC  },
          { Resize16t32,  Resize32t32,  Resize64t32NC,  Resize128t32NC  },
          { Resize16t64,  Resize32t64,  Resize64t64,  Resize128t64NC  },
          { Resize16t128, Resize32t128, Resize64t128, Resize128t128 } };

    int endian, i;
    WORD idxW = CompactifySizeof(sizeof(WORD));
    WORD idxL = CompactifySizeof(sizeof(LONG));
    WORD idxP = CompactifySizeof(sizeof(POSITION));
    WORD idxVP = CompactifySizeof(sizeof(void *));

    AO.transFlag = 0;
    AO.powerFlag = 0;
    AO.resizeFlag = 0;
    AO.bufferedInd = 0;

    if ( ReadFile(AO.SaveData.Handle,(UBYTE *)(&AO.SaveHeader),
        (LONG)sizeof(STOREHEADER)) != (LONG)sizeof(STOREHEADER) )
        return(MesPrint("Error reading save file header"));

    /* check whether save-file has no header. if yes then it is an old version
       of FORM -> go back to position 0 in file which then contains the first
       index and skip the rest. */
    for ( i = 0; i < 8; ++i ) {
        if ( AO.SaveHeader.headermark[i] != 0xFF ) {
            POSITION p;
            PUTZERO(p);
            SeekFile(AO.SaveData.Handle, &p, SEEK_SET);
            return ( 0 );
        }
    }

    if ( AO.SaveHeader.revision != SAVEREVISION ) {
        return(MesPrint("Save file header from an old version. Cannot read this file."));
    }

    endian = 1;
    for ( i = 1; i < (int)sizeof(int); ++i ) {
        endian «= 8;
        endian += i+1;
    }
    if ( ((char *)&endian)[0] < ((char *)&endian)[1] ) {
        /* this machine is big-endian */
        AO.ResizeData = ResizeDataBE;
    }
    else {
        /* this machine is little-endian */
        AO.ResizeData = ResizeDataLE;
    }

    /* set AO.transFlag if ANY conversion has to be done later */
    if ( AO.SaveHeader.endianness[0] > AO.SaveHeader.endianness[1] ) {
        AO.transFlag = ( ((char *)&endian)[0] < ((char *)&endian)[1] );
    }
    else {
        AO.transFlag = ( ((char *)&endian)[0] > ((char *)&endian)[1] );
    }
    if ( (WORD)AO.SaveHeader.lenWORD != sizeof(WORD) ) AO.transFlag |= 0x02;
    if ( (WORD)AO.SaveHeader.lenLONG != sizeof(LONG) ) AO.transFlag |= 0x04;
    if ( (WORD)AO.SaveHeader.lenPOS != sizeof(POSITION) ) AO.transFlag |= 0x08;
    if ( (WORD)AO.SaveHeader.lenPOINTER != sizeof(void *) ) AO.transFlag |= 0x10;

    AO.FlipWORD = flipJumpTable[idxW];
    AO.FlipLONG = flipJumpTable[idxL];
    AO.FlipPOS = flipJumpTable[idxP];
    AO.FlipPOINTER = flipJumpTable[idxVP];

    /* Works only for machines where WORD is not greater than 32bit ! */
    AO.CheckPower = CheckPower32;
```

```
    AO.RenumberVec = RenumberVec32;

    AO.ResizeWORD = resizeJumpTable[idxW][CompactifySizeof(AO.SaveHeader.lenWORD)];
    AO.ResizeNCWORD = resizeNCJumpTable[idxW][CompactifySizeof(AO.SaveHeader.lenWORD)];
    AO.ResizeLONG = resizeJumpTable[idxL][CompactifySizeof(AO.SaveHeader.lenLONG)];
    AO.ResizePOS = resizeJumpTable[idxP][CompactifySizeof(AO.SaveHeader.lenPOS)];
    AO.ResizePOINTER = resizeJumpTable[idxVP][CompactifySizeof(AO.SaveHeader.lenPOINTER)];

    {
        WORD dumw[8];
        UBYTE *dummy;
        for ( i = 0; i < 8; i++ ) dumw[i] = 0;
        dummy = (UBYTE *)dumw;
        for ( i = 0; i < 16; i++ ) dummy[i] = AO.SaveHeader.maxpower[i];
        AO.mpower = dumw[0];
    }

    return ( 0 );
}

/*
        #] ReadSaveHeader :
        #[ ReadSaveIndex :
*/

WORD ReadSaveIndex(FILEINDEX *fileind)
{
    /* do we need some translation for the FILEINDEX? */
    if ( AO.transFlag ) {
        /* if a translated FILEINDEX can hold less entries than the original
           FILEINDEX, then we need to buffer the extra entires in this static
           variable (can happen going from 32bit to 64bit */
        static FILEINDEX sbuffer;

        FILEINDEX buffer;
        UBYTE *p, *q;
        int i;

        /* shortcuts */
        int lenW = AO.SaveHeader.lenWORD;
        int lenL = AO.SaveHeader.lenLONG;
        int lenP = AO.SaveHeader.lenPOS;

        /* if we have a buffered FILEINDEX then just return it */
        if ( AO.bufferedInd ) {
            *fileind = sbuffer;
            AO.bufferedInd = 0;
            return ( 0 );
        }

        if ( ReadFile(AO.SaveData.Handle, (UBYTE *)fileind, sizeof(FILEINDEX))
                != sizeof(FILEINDEX) ) {
            return ( MesPrint("Error(1) reading stored expression.") );
        }

        /* do we need to flip the endianness? */
        if ( AO.transFlag & 1 ) {
            LONG number;
            /* padding bytes */
            int padp = lenL - ((lenW*5+(MAXENAME + 1)) & (lenL-1));
            p = (UBYTE *)fileind;
            AO.FlipPOS(p); p += lenP;           /* next */
            AO.FlipPOS(p);                      /* number */
            AO.ResizePOS(p, (UBYTE *)&number);
            p += lenP;
            for ( i = 0; i < number; ++i ) {
                AO.FlipPOS(p); p += lenP;       /* position */
                AO.FlipPOS(p); p += lenP;       /* length */
                AO.FlipPOS(p); p += lenP;       /* variables */
                AO.FlipLONG(p); p += lenL;      /* CompressSize */
                AO.FlipWORD(p); p += lenW;      /* nsymbols */
                AO.FlipWORD(p); p += lenW;      /* nindices */
                AO.FlipWORD(p); p += lenW;      /* nvectors */
                AO.FlipWORD(p); p += lenW;      /* nfunctions */
                AO.FlipWORD(p); p += lenW;      /* size */
                p += padp;
            }
        }

        /* do we need to resize data? */
        if ( AO.transFlag > 1 ) {
            LONG number, maxnumber;
            int n;
            /* padding bytes */
            int padp = lenL - ((lenW*5+(MAXENAME + 1)) & (lenL-1));
            int padq = sizeof(LONG) - ((sizeof(WORD)*5+(MAXENAME + 1)) & (sizeof(LONG)-1));

```

```
            p = (UBYTE *)fileind; q = (UBYTE *)&buffer;
            AO.ResizePOS(p, q);                     /* next */
            p += lenP; q += sizeof(POSITION);
            AO.ResizePOS(p, q);                 /* number */
            p += lenP;
            number = BASEPOSITION(*((POSITION *)q));
            /* if FILEINDEX in file contains more entries than the FILEINDEX in
               memory can contain, then adjust the numbers and prepare for
               buffering */
            if ( number > (LONG)INFILEINDEX ) {
                AO.bufferedInd = number-INFILEINDEX;
                if ( AO.bufferedInd > (WORD)INFILEINDEX ) {
                    /* can happen when reading 32bit and writing >=128bit.
                       Fix: more than one static buffer for FILEINDEX */
                    return ( MesPrint("Too many index entries.") );
                }
                maxnumber = INFILEINDEX;
                SETBASEPOSITION(*((POSITION *)q),INFILEINDEX);
            }
            else {
                maxnumber = number;
            }
            q += sizeof(POSITION);
            /* read all INDEXENTRY that fit into the output buffer */
            for ( i = 0; i < maxnumber; ++i ) {
                AO.ResizePOS(p, q);                 /* position */
                p += lenP; q += sizeof(POSITION);
                AO.ResizePOS(p, q);                 /* length */
                p += lenP; q += sizeof(POSITION);
                AO.ResizePOS(p, q);                 /* variables */
                p += lenP; q += sizeof(POSITION);
                AO.ResizeLONG(p, q);                /* CompressSize */
                p += lenL; q += sizeof(LONG);
                AO.ResizeWORD(p, q);                /* nsymbols */
                p += lenW; q += sizeof(WORD);
                AO.ResizeWORD(p, q);                /* nindices */
                p += lenW; q += sizeof(WORD);
                AO.ResizeWORD(p, q);                /* nvectors */
                p += lenW; q += sizeof(WORD);
                AO.ResizeWORD(p, q);                /* nfunctions */
                p += lenW; q += sizeof(WORD);
                AO.ResizeWORD(p, q);                /* size (unchanged!) */
                p += lenW; q += sizeof(WORD);
                n = MAXENAME + 1;
                NCOPYB(q, p, n)
                p += padp;
                q += padq;
            }
            /* read all the remaining INDEXENTRY and put them into the static buffer */
            if ( AO.bufferedInd ) {
                sbuffer.next = buffer.next;
                SETBASEPOSITION(sbuffer.number,AO.bufferedInd);
                q = (UBYTE *)&sbuffer + sizeof(POSITION) + sizeof(LONG);
                for ( i = maxnumber; i < number; ++i ) {
                    AO.ResizePOS(p, q);             /* position */
                    p += lenP; q += sizeof(POSITION);
                    AO.ResizePOS(p, q);             /* length */
                    p += lenP; q += sizeof(POSITION);
                    AO.ResizePOS(p, q);             /* variables */
                    p += lenP; q += sizeof(POSITION);
                    AO.ResizeLONG(p, q);            /* CompressSize */
                    p += lenL; q += sizeof(LONG);
                    AO.ResizeWORD(p, q);            /* nsymbols */
                    p += lenW; q += sizeof(WORD);
                    AO.ResizeWORD(p, q);            /* nindices */
                    p += lenW; q += sizeof(WORD);
                    AO.ResizeWORD(p, q);            /* nvectors */
                    p += lenW; q += sizeof(WORD);
                    AO.ResizeWORD(p, q);            /* nfunctions */
                    p += lenW; q += sizeof(WORD);
                    AO.ResizeWORD(p, q);            /* size (unchanged!) */
                    p += lenW; q += sizeof(WORD);
                    n = MAXENAME + 1;
                    NCOPYB(q, p, n)
                    p += padp;
                    q += padq;
                }
            }
            /* copy to output */
            p = (UBYTE *)fileind; q = (UBYTE *)&buffer; n = sizeof(FILEINDEX);
            NCOPYB(p, q, n)
        }
        return ( 0 );
    } else {
        return ( ReadFile(AO.SaveData.Handle, (UBYTE *)fileind, sizeof(FILEINDEX))
                 != sizeof(FILEINDEX) );
    }
```

```
}

/*
        #] ReadSaveIndex :
        #[ ReadSaveVariables :
*/

WORD ReadSaveVariables(UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize,\
                       INDEXENTRY *ind, LONG *stage)
{
    /* do we need some translation for the variables? */
    if ( AO.transFlag ) {
        /* counters for the number of already read symbols, indices, ... that
           need to remain valid between different calls to ReadSaveVariables().
           are initialized if stage == -1 */
        static WORD numReadSym;
        static WORD numReadInd;
        static WORD numReadVec;
        static WORD numReadFun;

        POSITION pos;
        UBYTE *in, *out, *pp = 0, *end, *outbuf;
        LONG numread;
        WORD namelen, realnamelen;
        /* shortcuts */
        WORD lenW = AO.SaveHeader.lenWORD;
        WORD lenL = AO.SaveHeader.lenLONG;
        WORD lenP = AO.SaveHeader.lenPOINTER;
        WORD flip = AO.transFlag & 1;

        /* remember file position in case we have to rewind */
        TELLFILE(AO.SaveData.Handle,&pos);

        /* decide on the position of the in and out buffers.
           if the input is "bigger" than the output, we resize in-place, i.e.
           we immediately overwrite the source data by the translated data. in
           and out buffers start at the same place.
           if not, we read from the end of the given buffer and write at the
           beginning. */
        if ( (lenW > (WORD)sizeof(WORD))
        || ( (lenW == (WORD)sizeof(WORD))
             && ( (lenL > (WORD)sizeof(LONG))
                  || ( (lenL == (WORD)sizeof(LONG)) && lenP > (WORD)sizeof(void *))
                )
           ) ) {
            in = out = buffer;
            end = buffer + *size;
        }
        else {
            /* data will grow roughly by sizeof(WORD)/lenW. the exact value is
               not important. if reading and writing areas start to overlap, the
               reading will already be near the end of the data and overwriting
               doesn't matter. */
            LONG newsize = (top - buffer) / (1 + sizeof(WORD)/lenW);
            end = top;
            out = buffer;
            in = end - newsize;
            if ( *size > newsize ) *size = newsize;
        }

        if ( ( numread = ReadFile(AO.SaveData.Handle, in, *size) ) != *size ) {
            return ( MesPrint("Error(2) reading stored expression.") );
        }

        *size = 0;
        *outsize = 0;

        /* first time in ReadSaveVariables(). initialize counters. */
        if ( *stage == -1 ) {
            numReadSym = 0;
            numReadInd = 0;
            numReadVec = 0;
            numReadFun = 0;
            +++stage;
        }

        while ( in < end ) {
            /* Symbols */
            if ( *stage == 0 ) {
                if ( ind->nsymbols <= numReadSym ) {
                    +++stage;
                    continue;
                }
                if ( end - in < AO.SaveHeader.sSym ) {
                    goto RSVEnd;
                }
                if ( flip ) {
```

```
                    pp = in;
                    AO.FlipLONG(pp); pp += lenL;
                    while ( pp < in + AO.SaveHeader.sSym ) {
                        AO.FlipWORD(pp); pp += lenW;
                    }
                }
                pp = in + AO.SaveHeader.sSym;
                AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name     */
                AO.CheckPower(in);
                AO.ResizeWORD(in, out); in += lenW;
                if ( *((WORD *)out) == -AO.mpower ) *((WORD *)out) = -MAXPOWER;
                                                    out += sizeof(WORD); /* minpower */
                AO.ResizeWORD(in, out); in += lenW;
                if ( *((WORD *)out) == AO.mpower ) *((WORD *)out) = MAXPOWER;
                                                    out += sizeof(WORD); /* maxpower */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* complex  */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number   */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags    */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node     */
                AO.ResizeWORD(in, out); in += lenW;                      /* namesize */
                realnamelen = *((WORD *)out);
                realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
                out += sizeof(WORD);
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
                while ( in < pp ) {
                    AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
                }
                namelen = *((WORD *)out-1); /* cares for padding "bug" */
                if ( end - in < namelen ) {
                    goto RSVEnd;
                }
                *((WORD *)out-1) = realnamelen;
                *size += AO.SaveHeader.sSym + namelen;
                *outsize += sizeof(struct SyMbOl) + realnamelen;
                if ( realnamelen > namelen ) {
                    int j = namelen;
                    NCOPYB(out, in, j);
                    out += realnamelen - namelen;
                }
                else {
                    int j = realnamelen;
                    NCOPYB(out, in, j);
                    in += namelen - realnamelen;
                }
                ++numReadSym;
                continue;
            }
            /* Indices */
            if ( *stage == 1 ) {
                if ( ind->nindices <= numReadInd ) {
                    +++stage;
                    continue;
                }
                if ( end - in < AO.SaveHeader.sInd ) {
                    goto RSVEnd;
                }
                if ( flip ) {
                    pp = in;
                    AO.FlipLONG(pp); pp += lenL;
                    while ( pp < in + AO.SaveHeader.sInd ) {
                        AO.FlipWORD(pp); pp += lenW;
                    }
                }
                pp = in + AO.SaveHeader.sInd;
                AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name      */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* type      */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number    */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags     */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* nmin4     */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node      */
                AO.ResizeWORD(in, out); in += lenW;                      /* namesize  */
                realnamelen = *((WORD *)out);
                realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
                out += sizeof(WORD);
                while ( in < pp ) {
                    AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
                }
                namelen = *((WORD *)out-1); /* cares for padding "bug" */
                if ( end - in < namelen ) {
                    goto RSVEnd;
                }
                *((WORD *)out-1) = realnamelen;
                *size += AO.SaveHeader.sInd + namelen;
                *outsize += sizeof(struct InDeX) + realnamelen;
                if ( realnamelen > namelen ) {
                    int j = namelen;
```

```
                    NCOPYB(out, in, j);
                    out += realnamelen - namelen;
                }
                else {
                    int j = realnamelen;
                    NCOPYB(out, in, j);
                    in += namelen - realnamelen;
                }
                ++numReadInd;
                continue;
            }
            /* Vectors */
            if ( *stage == 2 ) {
                if ( ind->nvectors <= numReadVec ) {
                    ++*stage;
                    continue;
                }
                if ( end - in < AO.SaveHeader.sVec ) {
                    goto RSVEnd;
                }
                if ( flip ) {
                    pp = in;
                    AO.FlipLONG(pp); pp += lenL;
                    while ( pp < in + AO.SaveHeader.sVec ) {
                        AO.FlipWORD(pp); pp += lenW;
                    }
                }
                pp = in + AO.SaveHeader.sVec;
                AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name     */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* complex  */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number   */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags    */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node     */
                AO.ResizeWORD(in, out); in += lenW;                      /* namesize */
                realnamelen = *((WORD *)out);
                realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
                out += sizeof(WORD);
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
                while ( in < pp ) {
                    AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
                }
                namelen = *((WORD *)out-1); /* cares for padding "bug" */
                if ( end - in < namelen ) {
                    goto RSVEnd;
                }
                *((WORD *)out-1) = realnamelen;
                *size += AO.SaveHeader.sVec + namelen;
                *outsize += sizeof(struct VeCtOr) + realnamelen;
                if ( realnamelen > namelen ) {
                    int j = namelen;
                    NCOPYB(out, in, j)
                    out += realnamelen - namelen;
                }
                else {
                    int j = realnamelen;
                    NCOPYB(out, in, j)
                    in += namelen - realnamelen;
                }
                ++numReadVec;
                continue;
            }
            /* Functions */
            if ( *stage == 3 ) {
                if ( ind->nfunctions <= numReadFun ) {
                    ++*stage;
                    continue;
                }
                if ( end - in < AO.SaveHeader.sFun ) {
                    goto RSVEnd;
                }
                if ( flip ) {
                    pp = in;
                    AO.FlipPOINTER(pp); pp += lenP;
                    AO.FlipLONG(pp); pp += lenL;
                    AO.FlipLONG(pp); pp += lenL;
                    while ( pp < in + AO.SaveHeader.sFun ) {
                        AO.FlipWORD(pp); pp += lenW;
                    }
                }
                pp = in + AO.SaveHeader.sFun;
                outbuf = out;
                AO.ResizePOINTER(in, out); in += lenP; out += sizeof(void *); /* tabl */
                AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* symminfo  */
                AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name      */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* commute   */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* complex   */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number    */
```

```
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags     */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* spec      */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* symmetric */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* numargs */
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node      */
                AO.ResizeWORD(in, out); in += lenW;                      /* namesize  */
                realnamelen = *((WORD *)out);
                realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
                out += sizeof(WORD);
                AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
                while ( in < pp ) {
                    AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
                }
                namelen = *((WORD *)out-1); /* cares for padding "bug" */
                if ( end - in < namelen ) {
                    goto RSVEnd;
                }
                *((WORD *)out-1) = realnamelen;
                *size += AO.SaveHeader.sFun + namelen;
                *outsize += sizeof(struct FuNcTiOn) + realnamelen;
                if ( realnamelen > namelen ) {
                    int j = namelen;
                    NCOPYB(out, in, j);
                    out += realnamelen - namelen;
                }
                else {
                    int j = realnamelen;
                    NCOPYB(out, in, j);
                    in += namelen - realnamelen;
                }
                ++numReadFun;
                /* we use the information whether a function is tensorial later in ReadSaveTerm */
                AO.tensorList[((FUNCTIONS)outbuf)->number+FUNCTION] =
                    (UBYTE)(((FUNCTIONS)outbuf)->spec == TENSORFUNCTION);
                continue;
            }
            /* handle numdummies */
            if ( end - in >= lenW ) {
                if ( flip ) AO.FlipWORD(in);
                AO.ResizeWORD(in, out);
                *size += lenW;
                *outsize += sizeof(WORD);
            }
            /* handle numfactors */
            if ( end - in >= lenW ) {
                if ( flip ) AO.FlipWORD(in);
                AO.ResizeWORD(in, out);
                *size += lenW;
                *outsize += sizeof(WORD);
            }
            /* handle vflags */
            if ( end - in >= lenW ) {
                if ( flip ) AO.FlipWORD(in);
                AO.ResizeWORD(in, out);
                *size += lenW;
                *outsize += sizeof(WORD);
            }
            return ( 0 );
        }

RSVEnd:
        /* we are here because the remaining buffer cannot hold the next
           struct. we position the file behind the last sucessfully translated
           struct and return. */
        ADDPOS(pos, *size);
        SeekFile(AO.SaveData.Handle, &pos, SEEK_SET);
        return ( 0 );
    } else {
        return ( ReadFile(AO.SaveData.Handle, buffer, *size) != *size );
    }
}

/*
        #] ReadSaveVariables :
        #[ ReadSaveTerm :
*/

UBYTE *
ReadSaveTerm32(UBYTE *bin, UBYTE *binend, UBYTE **bout, UBYTE *boutend, UBYTE *top, int terminbuf)
{
    GETIDENTITY

    UBYTE *boutbuf;
    INT32 len, j, id;
    INT32 *r, *t, *coeff, *end, *newtermsize, *rend;
    INT32 *newsubtermp;
    INT32 *in = (INT32 *)bin;
```

```
    INT32 *out = (INT32 *)*bout;

    /* if called recursively the term is already decompressed in buffer.
       is this the case? */
    if ( terminbuf ) {
        /* don't do any decompression, just adjust the pointers */
        len = *out;
        end = out + len;
        r = in + 1;
        rend = (INT32 *)boutend;
        coeff = end - ABS(*(end-1));
        newtermsize = (INT32 *)*bout;
        out = newtermsize + 1;
    }
    else {
        /* do deprompression of necessary. always return if the space in the
           buffer is not sufficient */
        INT32 rbuf;
        r = (INT32 *)AR.CompressBuffer;
        rbuf = *r;
        len = j = *in;
        /* first copy from AR.CompressBuffer if necessary */
        if ( j < 0 ) {
            ++in;
            if ( (UBYTE *)in >= binend ) {
                return ( bin );
            }
            *out = len = -j + 1 + *in;
            end = out + *out;
            if ( (UBYTE *)end >= top ) {
                return ( bin );
            }
            ++out;
            *r++ = len;
            while ( ++j <= 0 ) {
                INT32 bb = *r++;
                *out++ = bb;
            }
            j = *in++;
        }
        else if ( j == 0 ) {
            /* care for padding words */
            while ( (UBYTE *)in < binend ) {
                *out++ = 0;
                if ( (UBYTE *)out > top ) {
                    return ( (UBYTE *)bin );
                }

                *r++ = 0;
                ++in;
            }
            *bout = (UBYTE *)out;
            return ( (UBYTE *)in );
        }
        else {
            end = out + len;
            if ( (UBYTE *)end >= top ) {
                return ( bin );
            }
        }
        if ( (UBYTE *)(in + j) >= binend ) {
            *(AR.CompressBuffer) = rbuf;
            return ( bin );
        }
        if ( (UBYTE *)out + j >= top ) {
            return ( bin );
        }
        /* second copy from input buffer */
        while ( --j >= 0 ) {
            INT32 bb = *in++;
            *r++ = *out++ = bb;
        }

        rend = r;
        r = (INT32 *)AR.CompressBuffer + 1;
        coeff = end - ABS(*(end-1));
        newtermsize = (INT32 *)*bout;
        out = newtermsize + 1;
    }

    /* iterate over subterms */
    while ( out < coeff ) {

        id = *out++;
        ++r;
        t = out + *out - 1;
        newsubtermp = out;
```

```
        ++out; ++r;

        if ( id == SYMBOL ) {
            while ( out < t ) {
                ++out; ++r; /* symbol number */
                /* if exponent is too big, rewrite as exponent function */
                if ( ABS(*out) >= MAXPOWER ) {
                    INT32 *a, *b;
                    INT32 n;
                    INT32 num = *(out-1);
                    INT32 exp = *out;
                    coeff += 9;
                    end += 9;
                    t += 9;
                    if ( (UBYTE *)end > top ) return ( bin );
                    out -= 3;
                    *out++ = EXPONENT;      /* id */
                    *out++ = 13;            /* size */
                    *out++ = 1;             /* dirtyflag */
                    *out++ = -SYMBOL;       /* first short arg */
                    *out++ = num;
                    *out++ = 8;             /* second arg, size */
                    *out++ = 0;             /* dirtyflag */
                    *out++ = 6;             /* term size */
                    *out++ = ABS(exp) & 0x0000FFFF;
                    *out++ = ABS(exp) » 16;
                    *out++ = 1;
                    *out++ = 0;
                    *out++ = ( exp < 0 ) ? -5 : 5;
                    a = ++r;
                    b = out;
                    n = rend - r;
                    NCOPYI32(b, a, n)
                }
                else {
                    ++out; ++r;
                }
            }
        }
        else if ( id == DOTPRODUCT ) {
            while ( out < t ) {
                AO.RenumberVec((UBYTE *)out); /* vector 1 */
                ++out; ++r;
                AO.RenumberVec((UBYTE *)out); /* vector 2 */
                ++out; ++r;
                /* if exponent is too big, rewrite as exponent function */
                if ( ABS(*out) >= MAXPOWER ) {
                    INT32 *a, *b;
                    INT32 n;
                    INT32 num1 = *(out-2);
                    INT32 num2 = *(out-1);
                    INT32 exp = *out;
                    coeff += 17;
                    end += 17;
                    t += 17;
                    if ( (UBYTE *)end > top ) return ( bin );
                    out -= 4;
                    *out++ = EXPONENT;      /* id */
                    *out++ = 22;            /* size */
                    *out++ = 1;             /* dirtyflag */
                    *out++ = 11;            /* first arg, size */
                    *out++ = 0;             /* dirtyflag */
                    *out++ = 9;             /* term size */
                    *out++ = DOTPRODUCT;    /* p1.p2 */
                    *out++ = 5;             /* subterm size */
                    *out++ = num1;          /* p1 */
                    *out++ = num2;          /* p2 */
                    *out++ = 1;             /* exponent */
                    *out++ = 1;             /* coeff */
                    *out++ = 1;
                    *out++ = 3;
                    *out++ = 8;             /* second arg, size */
                    *out++ = 0;             /* dirtyflag */
                    *out++ = 6;             /* term size */
                    *out++ = ABS(exp) & 0x0000FFFF;
                    *out++ = ABS(exp) » 16;
                    *out++ = 1;
                    *out++ = 0;
                    *out++ = ( exp < 0 ) ? -5 : 5;
                    a = ++r;
                    b = out;
                    n = rend - r;
                    NCOPYI32(b, a, n)
                }
                else {
                    ++out; ++r;
                }
```

```
            }
        }
        else if ( id == VECTOR ) {
            while ( out < t ) {
                AO.RenumberVec((UBYTE *)out); /* vector number */
                ++out; ++r;
                ++out; ++r; /* index, do nothing */
            }
        }
        else if ( id == INDEX ) {
/*          INT32 vectoroffset = -2 * *((INT32 *)AO.SaveHeader.wildoffset); */
            void *dummy = (void *)AO.SaveHeader.wildoffset;  /* to remove a warning about strict-aliasing rules in gcc */
            INT32 vectoroffset = -2 * *((INT32 *)dummy);
            while ( out < t ) {
                /* if there is a vector, renumber it */
                if ( *out < vectoroffset ) {
                    AO.RenumberVec((UBYTE *)out);
                }
                ++out; ++r;
            }
        }
        else if ( id == SUBEXPRESSION ) {
            /* nothing to translate */
            while ( out < t ) {
                ++out; ++r;
            }
        }
        else if ( id == DELTA ) {
            /* nothing to translate */
            r += t - out;
            out = t;
        }
        else if ( id == HAAKJE ) {
            /* nothing to translate */
            r += t - out;
            out = t;
        }
        else if ( id == GAMMA || id == LEVICIVITA || (id >= FUNCTION && AO.tensorList[id]) ) {
/*          INT32 vectoroffset = -2 * *((INT32 *)AO.SaveHeader.wildoffset); */
            void *dummy = (void *)AO.SaveHeader.wildoffset;  /* to remove a warning about strict-aliasing rules in gcc */
            INT32 vectoroffset = -2 * *((INT32 *)dummy);
            while ( out < t ) {
                /* if there is a vector as an argument, renumber it */
                if ( *out < vectoroffset ) {
                    AO.RenumberVec((UBYTE *)out);
                }
                ++out; ++r;
            }
        }
        else if ( id >= FUNCTION ) {
            INT32 *argEnd;
            UBYTE *newbin;

            ++out; ++r; /* dirty flags */

            /* loop over arguments */
            while ( out < t ) {
                if ( *out < 0 ) {
                    /* short notation arguments */
                    switch ( -*out ) {
                        case SYMBOL:
                            ++out; ++r;
                            ++out; ++r;
                            break;
                        case SNUMBER:
                            ++out; ++r;
                            if ( sizeof(WORD) == 2 ) {
                                /* resize if needed */
                                if ( *out > (1«15)-1 || *out < -(1«15)+1 ) {
                                    INT32 *a, *b;
                                    INT32 n;
                                    INT32 num = *out;
                                    coeff += 6;
                                    end += 6;
                                    argEnd += 6;
                                    t += 6;
                                    if ( (UBYTE *)end > top ) return ( bin );
                                    --out;
                                    *out++ = 8;         /* argument size */
                                    *out++ = 0;         /* dirtyflag */
                                    *out++ = 6;         /* term size */
                                    *out++ = ABS(num) & 0x0000FFFF;
                                    *out++ = ABS(num) » 16;
                                    *out++ = 1;
                                    *out++ = 0;
```

```
                                    *out++ = ( num < 0 ) ? -5 : 5;
                                    a = ++r;
                                    b = out;
                                    n = rend - r;
                                    NCOPYI32(b, a, n)
                                }
                                else {
                                    ++out; ++r;
                                }
                            }
                            else {
                                ++out; ++r;
                            }
                            break;
                        case VECTOR:
                            ++out; ++r;
                            AO.RenumberVec((UBYTE *)out);
                            ++out; ++r;
                            break;
                        case INDEX:
                            ++out; ++r;
                            ++out; ++r;
                            break;
                        case MINVECTOR:
                            ++out; ++r;
                            AO.RenumberVec((UBYTE *)out);
                            ++out; ++r;
                            break;
                        default:
                            if ( -*out >= FUNCTION ) {
                                ++out; ++r;
                                break;
                            } else {
                                MesPrint("short function code %d not implemented.", *out);
                                return ( (UBYTE *)in );
                            }
                    }
                }
                else {
                    /* long arguments */
                    INT32 *newargsize = out;
                    argEnd = out + *out;
                    ++out; ++r;
                    ++out; ++r; /* dirty flags */
                    while ( out < argEnd ) {
                        INT32 *keepsizep = out + *out;
                        INT32 lenbuf = *out;
                        INT32 **ppp = &out; /* to avoid a compiler warning */
                        /* recursion */
                        newbin = ReadSaveTerm32((UBYTE *)r, binend, (UBYTE **)ppp, (UBYTE *)rend, top, 1);
                        r += lenbuf;
                        if ( newbin == (UBYTE *)r ) {
                            return ( (UBYTE *)in );
                        }
                        /* if the term done by recursion has changed in size,
                           we need to move the rest of the data accordingly */
                        if ( out > keepsizep ) {
                            INT32 *a, *b;
                            INT32 n;
                            INT32 extention = out - keepsizep;
                            a = r;
                            b = out;
                            n = rend - r;
                            NCOPYI32(b, a, n)
                            coeff += extention;
                            end += extention;
                            argEnd += extention;
                            t += extention;
                        }
                        else if ( out < keepsizep ) {
                            INT32 *a, *b;
                            INT32 n;
                            INT32 extention = keepsizep - out;
                            a = keepsizep;
                            b = out;
                            n = rend - r;
                            NCOPYI32(b, a, n)
                            coeff -= extention;
                            end -= extention;
                            argEnd -= extention;
                            t -= extention;
                        }
                    }
                    *newargsize = out - newargsize;
                }
            }
```

```
        }
        else {
            MesPrint("ID %d not recognized.", id);
            return ( (UBYTE *)in );
        }

        *newsubtermp = out - newsubtermp + 1;
    }

    if ( (UBYTE *)end >= top ) {
        return ( bin );
    }

    /* do coefficient and adjust term size */
    boutbuf = *bout;
    *bout = (UBYTE *)out;

    ResizeCoeff32(bout, (UBYTE *)end, top);

    if ( *bout >= top ) {
        *bout = boutbuf;
        return ( bin );
    }

    *newtermsize = (INT32 *)*bout - newtermsize;

    return ( (UBYTE *)in );
}

/*
        #] ReadSaveTerm :
        #[ ReadSaveExpression :
*/

WORD ReadSaveExpression(UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize)
{
    if ( AO.transFlag ) {
        UBYTE *in, *end, *out, *outend, *p;
        POSITION pos;
        LONG half;
        WORD lenW = AO.SaveHeader.lenWORD;

        /* remember the last file position in case an expression cannot be
           fully processed */
        TELLFILE(AO.SaveData.Handle,&pos);

        /* adjust 'size' depending on whether the translated data is bigger or
           smaller */
        half = (top-buffer)/2;
        if ( *size > half ) *size = half;
        if ( lenW < (WORD)sizeof(WORD) ) {
            if ( *size * (LONG)sizeof(WORD)/lenW > half ) *size = half*lenW/(LONG)sizeof(WORD);
        }
        else {
            if ( *size > half ) *size = half;
        }

        /* depending on the necessary resizing we position the input pointer
           either at the start of the buffer or in the middle. if the data will
           roughly remain the same size, we need only one processing step, so
           we put the 'in' at the middle and 'out' and the beginning. in the
           other cases we need two processing steps, so first we put 'in' at
           the beginning and write at the middle. the second step can then read
           from the middle and put its results at the beginning. */
        in = out = buffer;
        if ( lenW == sizeof(WORD) ) in += half;
        else out += half;
        end = in + *size;
        outend = out + *size;

        if ( ReadFile(AO.SaveData.Handle, in, *size) != *size ) {
            return ( MesPrint("Error(3) reading stored expression.") );
        }

        if ( AO.transFlag & 1 ) {
            p = in;
            end -= lenW;
            while ( p <= end ) {
                AO.FlipWORD(p); p += lenW;
            }
            end += lenW;
        }

        if ( lenW > (WORD)sizeof(WORD) ) {
            /* renumber first */
            do {
                outend = out++*size;
```

```
                if ( outend > top ) outend = top;
                p = ReadSaveTerm32(in, end, &out, outend, top, 0);
                if ( p == in ) break;
                in = p;
            } while ( in <= end - lenW );
            /* then resize */
            *size = in - buffer;
            in = buffer + half;
            end = out;
            out = buffer;

            while ( in < end ) {
                /* resize without checking */
                AO.ResizeNCWORD(in, out);
                in += lenW; out += sizeof(WORD);
            }
        }
        else {
            if ( lenW < (WORD)sizeof(WORD) ) {
                /* resize first */
                while ( in < end ) {
                    AO.ResizeWORD(in, out);
                    in += lenW; out += sizeof(WORD);
                }
                in = buffer + half;
                end = out;
                out = buffer;
            }
            /* then renumber */
            do {
                p = ReadSaveTerm32(in, end, &out,  buffer+half, buffer+half, 0);
                if ( p == in ) break;
                in = p;
            } while ( in <= end - sizeof(WORD) );
            *size = (in - buffer - half) * lenW / (ULONG)sizeof(WORD);
        }
        *outsize = out - buffer;
        ADDPOS(pos, *size);
        SeekFile(AO.SaveData.Handle, &pos, SEEK_SET);

        return ( 0 );
    }
    else {
        return ( ReadFile(AO.SaveData.Handle, buffer, *size) != *size );
    }
}

/*
        #] ReadSaveExpression :
    #] System Independent Saved Expressions :
*/
```

4.108 structs.h File Reference

This graph shows which files directly or indirectly include this file:

Data Structures

- struct PoSiTiOn
- struct STOREHEADER
- struct InDeXeNtRy
- struct FiLeInDeX
- struct FiLeDaTa
- struct VaRrEnUm
- struct ReNuMbEr
- struct LIST

- struct KEYWORD
- struct NaMeNode
- struct NaMeTree
- struct tree
- struct MiNmAx
- struct BrAcKeTiNdEx
- struct BrAcKeTiNfO
- struct TaBlEs
- struct ExPrEsSiOn
- struct SyMbOl
- struct InDeX
- struct VeCtOr
- struct FuNcTiOn
- struct SeTs
- struct DuBiOuS
- struct FaCdOlLaR
- struct DoLlArS
- struct MoDoPtDoLlArS
- struct fixedset
- struct TaBlEbAsEsUbInDeX
- struct TaBlEbAsE
- struct FUN_INFO
- struct FiLe
- struct StreaM
- struct SpecTatoR
- struct TrAcEs
- struct TrAcEn
- struct pReVaR
- struct INSIDEINFO
- struct PRELOAD
- struct PROCEDURE
- struct DoLoOp
- struct bit_field
- struct HANDLERS
- struct CbUf
- struct ChAnNeL
- struct SETUPPARAMETERS
- struct NeStInG
- struct StOrEcAcHe
- struct PeRmUtE
- struct PeRmUtEp
- struct DiStRiBuTe
- struct PaRtI
- struct sOrT
- struct POLYMOD
- struct SHvariables
- struct COST
- struct MODNUM
- struct OPTIMIZE
- struct OPTIMIZERESULT
- struct DICTIONARY_ELEMENT
- struct DICTIONARY
- struct M_const
- struct P_const
- struct C_const

- struct S_const
- struct R_const
- struct T_const
- struct N_const
- struct O_const
- struct X_const
- struct AllGlobals
- struct FixedGlobals

Macros

- #define INFILEINDEX ((512-2∗sizeof(POSITION))/sizeof(INDEXENTRY))
- #define EMPTYININDEX (512-2∗sizeof(POSITION)-INFILEINDEX∗sizeof(INDEXENTRY))

Typedefs

- typedef struct PoSiTiOn **POSITION**
- typedef struct InDeXeNtRy INDEXENTRY
- typedef struct FiLeInDeX FILEINDEX
- typedef struct FiLeDaTa **FILEDATA**
- typedef struct VaRrEnUm VARRENUM
- typedef struct ReNuMbEr ∗ RENUMBER
- typedef struct NaMeNode NAMENODE
- typedef struct NaMeTree NAMETREE
- typedef struct tree COMPTREE
- typedef struct MiNmAx **MINMAX**
- typedef struct BrAcKeTiNdEx **BRACKETINDEX**
- typedef struct BrAcKeTiNfO **BRACKETINFO**
- typedef struct TaBlEs ∗ TABLES
- typedef struct ExPrEsSiOn ∗ **EXPRESSIONS**
- typedef struct SyMbOl ∗ **SYMBOLS**
- typedef struct InDeX ∗ **INDICES**
- typedef struct VeCtOr ∗ **VECTORS**
- typedef struct FuNcTiOn ∗ FUNCTIONS
- typedef struct SeTs ∗ **SETS**
- typedef struct DuBiOuS ∗ **DUBIOUSV**
- typedef struct FaCdOlLaR **FACDOLLAR**
- typedef struct DoLlArS ∗ **DOLLARS**
- typedef struct MoDoPtDoLlArS **MODOPTDOLLAR**
- typedef struct fixedset **FIXEDSET**
- typedef struct TaBlEbAsEsUbInDeX **TABLEBASESUBINDEX**
- typedef struct TaBlEbAsE **TABLEBASE**
- typedef struct FiLe FILEHANDLE
- typedef struct StreaM STREAM
- typedef struct SpecTatoR **SPECTATOR**
- typedef struct TrAcEs TRACES
- typedef struct TrAcEn ∗ TRACEN
- typedef struct pReVaR PREVAR
- typedef struct DoLoOp DOLOOP
- typedef struct bit_field set_of_char[32]
- typedef struct bit_field ∗ one_byte
- typedef struct CbUf CBUF
- typedef struct ChAnNeL CHANNEL

- typedef struct NeStInG * NESTING
- typedef struct StOrEcAcHe * STORECACHE
- typedef struct PeRmUtE PERM
- typedef struct PeRmUtEp PERMP
- typedef struct DiStRiBuTe DISTRIBUTE
- typedef struct PaRtI PARTI
- typedef struct sOrT SORTING
- typedef struct AllGlobals ALLGLOBALS
- typedef WORD(* WCN) ()
- typedef WORD(* WCN2) ()
- typedef struct FixedGlobals FIXEDGLOBALS

Functions

- **STATIC_ASSERT** (sizeof(STOREHEADER)==512)
- **STATIC_ASSERT** (sizeof(FILEINDEX)==512)

4.108.1 Detailed Description

Contains definitions for global structs.

!!!CAUTION!!! Changes in this file will most likely have consequences for the recovery mechanism (see checkpoint.c). You need to care for the code in checkpoint.c as well and modify the code there accordingly!

The marker [D] is used in comments in this file to mark pointers to which dynamically allocated memory is assigned by a call to malloc() during runtime (in contrast to pointers that point into already allocated memory). This information is especially helpful if one needs to know which pointers need to be freed (cf. checkpoint.c).

Definition in file structs.h.

4.108.2 Macro Definition Documentation

4.108.2.1 INFILEINDEX

```
#define INFILEINDEX ((512-2*sizeof(POSITION))/sizeof(INDEXENTRY))
```

Maximum number of entries (struct InDeXeNtRy) in a file index (struct FiLeInDeX). Number is calculated such that the size of a file index is no more than 512 bytes.

Definition at line 118 of file structs.h.

4.108.2.2 EMPTYININDEX

```
#define EMPTYININDEX (512-2*sizeof(POSITION)-INFILEINDEX*sizeof(INDEXENTRY))
```

Number of empty filling bytes for a file index (struct FiLeInDeX). It is calculated such that the size of a file index is always 512 bytes.

Definition at line 123 of file structs.h.

4.108.3 Typedef Documentation

4.108.3.1 INDEXENTRY

```
typedef struct InDeXeNtRy INDEXENTRY
```

Defines the structure of an entry in a file index (see struct FiLeInDeX).

It represents one expression in the file.

4.108.3.2 FILEINDEX

```
typedef struct FiLeInDeX FILEINDEX
```

Defines the structure of a file index in store-files and save-files.

It contains several entries (see struct InDeXeNtRy) up to a maximum of INFILEINDEX.

The variable number has been made of type POSITION to avoid padding problems with some types of computers/↩ OS and keep system independence of the .sav files.

This struct is always 512 bytes long.

4.108.3.3 VARRENUM

```
typedef struct VaRrEnUm VARRENUM
```

Contains the pointers to an array in which a binary search will be performed.

4.108.3.4 RENUMBER

```
typedef struct ReNuMbEr * RENUMBER
```

Only symb.lo gets dynamically allocated. All other pointers points into this memory.

4.108.3.5 NAMENODE

```
typedef struct NaMeNode NAMENODE
```

The names of variables are kept in an array. Elements of type NAMENODE define a tree (that is kept balanced) that make it easy and fast to look for variables. See also NAMETREE.

4.108.3.6 NAMETREE

```
typedef struct NaMeTree NAMETREE
```

A struct of type NAMETREE controls a complete (balanced) tree of names for the compiler. The compiler maintains several of such trees and the system has been set up in such a way that one could define more of them if we ever want to work with local name spaces.

4.108.3.7 COMPTREE

```
typedef struct tree COMPTREE
```

The subexpressions in the compiler are kept track of in a (balanced) tree to reduce the need for subexpressions and hence save much space in large rhs expressions (like when we have xxxxxxx occurrences of objects like f(x+1,x+1) in which each x+1 becomes a subexpression. The struct that controls this tree is COMPTREE.

4.108.3.8 TABLES

```
typedef struct TaBlEs * TABLES
```

buffers, mm, flags, and prototype are always dynamically allocated, tablepointers only if needed (=0 if unallocated), boomlijst and argtail only for sparse tables.

Allocation is done for both the normal and the stub instance (spare), except for prototype and argtail which share memory.

4.108.3.9 FUNCTIONS

```
typedef struct FuNcTiOn * FUNCTIONS
```

Contains all information about a function. Also used for tables. It is used in the LIST elements of AC.

4.108.3.10 FILEHANDLE

```
typedef struct FiLe FILEHANDLE
```

The type FILEHANDLE is the struct that controls all relevant information of a file, whether it is open or not. The file may even not yet exist. There is a system of caches (PObuffer) and as long as the information to be written still fits inside the cache the file may never be created. There are variables that can store information about different types of files, like scratch files or sort files. Depending on what is available in the system we may also have information about gzip compression (currently sort file only) or locks (TFORM).

4.108.3.11 STREAM

```
typedef struct StreaM STREAM
```

Input is read from 'streams' which are represented by objects of type STREAM. A stream can be a file, a do-loop, a procedure, the string value of a preprocessor variable ..... When a new stream is opened we have to keep information about where to fall back in the parent stream to allow this to happen even in the middle of reading names etc as would be the case with a`i'b

4.108.3.12 TRACES

```
typedef struct TrAcEs TRACES
```

The struct TRACES keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue. The 4-dimensional traces are more complicated than the n-dimensional traces (see TRACEN) because of the extra tricks that can be used. They are responsible for the shorter final expressions.

4.108.3.13 TRACEN

```
typedef struct TrAcEn * TRACEN
```

The struct TRACEN keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue.

4.108.3.14 PREVAR

```
typedef struct pReVaR PREVAR
```

An element of the type PREVAR is needed for each preprocessor variable.

4.108.3.15 DOLOOP

```
typedef struct DoLoOp DOLOOP
```

Each preprocessor do loop has a struct of type DOLOOP to keep track of all relevant parameters like where the beginning of the loop is, what the boundaries, increment and value of the loop parameter are, etc. Also we keep the whole loop inside a buffer of type PRELOAD

4.108.3.16 set_of_char

```
typedef struct bit_field set_of_char[32]
```

Used in set_in, set_set, set_del and set_sub.

Definition at line 890 of file structs.h.

4.108.3.17 one_byte

```
typedef struct bit_field* one_byte
```

Used in set_in, set_set, set_del and set_sub.

Definition at line 896 of file structs.h.

4.108.3.18 CBUF

```
typedef struct CbUf CBUF
```

The CBUF struct is used by the compiler. It is a compiler buffer of which since version 3.0 there can be many.

4.108.3.19 CHANNEL

```
typedef struct ChAnNeL CHANNEL
```

When we read input from text files we have to remember not only their handle but also their name. This is needed for error messages. Hence we call such a file a channel and reserve a struct of type CHANNEL to allow to lay this link.

4.108.3.20 NESTING

```
typedef struct NeStInG * NESTING
```

The NESTING struct is used when we enter the argument of functions and there is the possibility that we have to change something there. Because functions can be nested we have to keep track of all levels of functions in case we have to move the outer layers to make room for a larger function argument.

4.108.3.21 STORECACHE

```
typedef struct StOrEcAcHe * STORECACHE
```

The struct of type STORECACHE is used by a caching system for reading terms from stored expressions. Each thread should have its own system of caches.

4.108.3.22 PERM

```
typedef struct PeRmUtE PERM
```

The struct PERM is used to generate all permutations when the pattern matcher has to try to match (anti)symmetric functions.

4.108.3.23 PERMP

```
typedef struct PeRmUtEp PERMP
```

Like struct PERM but works with pointers.

4.108.3.24 DISTRIBUTE

```
typedef struct DiStRiBuTe DISTRIBUTE
```

The struct DISTRIBUTE is used to help the pattern matcher when matching antisymmetric tensors.

4.108.3.25 PARTI

```
typedef struct PaRtI PARTI
```

The struct PARTI is used to help determining whether a partition_ function can be replaced.

4.108.3.26 SORTING

```
typedef struct sOrT SORTING
```

The struct SORTING is used to control a sort operation. It includes a small and a large buffer and arrays for keeping track of various stages of the (merge) sorts. Each sort level has its own struct and different levels can have different sizes for its arrays. Also different threads have their own set of SORTING structs.

4.108.3.27 ALLGLOBALS

```
typedef struct AllGlobals ALLGLOBALS
```

Without pthreads (FORM) the ALLGLOBALS struct has all the global variables

4.108.3.28 WCN

```
typedef WORD(* WCN) ()
```

Definition at line 2423 of file structs.h.

4.108.3.29 WCN2

```
typedef WORD(* WCN2) ()
```

Definition at line 2424 of file structs.h.

4.108.3.30 FIXEDGLOBALS

```
typedef struct FixedGlobals FIXEDGLOBALS
```

The FIXEDGLOBALS struct is an anachronism. It started as the struct with global variables that needed initialization. It contains the elements Operation and OperaFind which define a very early way of automatically jumping to the proper operation. We find the results of it in parts of the file opera.c Later operations were treated differently in a more transparent way. We never changed the existing code. The most important part is currently the cTable which is used intensively in the compiler.

4.109 structs.h

Go to the documentation of this file.

```
00001
00017 /* #[ License : */
00018 /*
00019  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00020  *   When using this file you are requested to refer to the publication
00021  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00022  *   This is considered a matter of courtesy as the development was paid
00023  *   for by FOM the Dutch physics granting agency and we would like to
00024  *   be able to track its scientific use to convince FOM of its value
00025  *   for the community.
00026  *
00027  *   This file is part of FORM.
00028  *
00029  *   FORM is free software: you can redistribute it and/or modify it under the
00030  *   terms of the GNU General Public License as published by the Free Software
00031  *   Foundation, either version 3 of the License, or (at your option) any later
00032  *   version.
00033  *
00034  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00035  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00036  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00037  *   details.
00038  *
00039  *   You should have received a copy of the GNU General Public License along
00040  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00041  */
00042 /* #] License : */
00043
00044 #ifndef __STRUCTS__
00045
00046 #define __STRUCTS__
00047 #ifdef _MSC_VER
00048 #include <wchar.h>  /* off_t */
00049 #endif
00050 /*
00051     #[ sav&store :
00052 */
00053
00058 typedef struct PoSiTiOn {
00059     off_t p1;
00060 } POSITION;
00061
00062 /*  Next are the index structs for stored and saved expressions */
00063
00074 typedef struct {
00075     UBYTE headermark[8];
00077     UBYTE lenWORD;
00078     UBYTE lenLONG;
00079     UBYTE lenPOS;
00080     UBYTE lenPOINTER;
00081     UBYTE endianness[16];
00082     UBYTE sSym;
00083     UBYTE sInd;
00084     UBYTE sVec;
00085     UBYTE sFun;
00086     UBYTE maxpower[16];
00087     UBYTE wildoffset[16];
00088     UBYTE revision;
00089     UBYTE reserved[512-8-4-16-4-16-16-1];
00090 } STOREHEADER;
00091
00092 STATIC_ASSERT(sizeof(STOREHEADER) == 512);
00093
00099 typedef struct InDeXeNtRy {
00100     POSITION    position;
00101     POSITION    length;
00102     POSITION    variables;
00103     LONG    CompressSize;
00104     WORD    nsymbols;
00105     WORD    nindices;
00106     WORD    nvectors;
00107     WORD    nfunctions;
00108     WORD    size;
00109     SBYTE   name[MAXENAME+1];
00110     PADPOSITION(0,1,0,5,MAXENAME+1);
00111 } INDEXENTRY;
00112
00118 #define INFILEINDEX ((512-2*sizeof(POSITION))/sizeof(INDEXENTRY))
00123 #define EMPTYININDEX (512-2*sizeof(POSITION)-INFILEINDEX*sizeof(INDEXENTRY))
00124
00137 typedef struct FiLeInDeX {
00138     POSITION    next;
```

```
    POSITION    number;
    INDEXENTRY expression[INFILEINDEX];
    SBYTE   empty[EMPTYININDEX];
} FILEINDEX;

STATIC_ASSERT(sizeof(FILEINDEX) == 512);

typedef struct FiLeDaTa {
    FILEINDEX Index;
    POSITION Fill;
    POSITION Position;
    WORD Handle;
    WORD dirtyflag;
    PADPOSITION(0,0,0,2,0);
} FILEDATA;

typedef struct VaRrEnUm {
    WORD  *start;
    WORD  *lo;
    WORD  *hi;
} VARRENUM;

typedef struct ReNuMbEr {
    POSITION   startposition;
    /* First stage renumbering */
    VARRENUM   symb;
    VARRENUM   indi;
    VARRENUM   vect;
    VARRENUM   func;
    /* Second stage renumbering */
    WORD       *symnum;
    WORD       *indnum;
    WORD       *vecnum;
    WORD       *funnum;
    PADPOSITION(4,0,0,0,sizeof(VARRENUM)*4);
} *RENUMBER;

/*
    #] sav&store :
    #[ Variables :
*/

typedef struct {
    void *lijst;
    char *message;
    int num;
    int maxnum;
    int size;
    int numglobal;
    int numtemp;
    int numclear;
    PADPOINTER(0,6,0,0);
} LIST;

typedef struct {
    char *name;
    TFUN func;
    int type;
    int flags;
} KEYWORD;

typedef struct NaMeNode {
    LONG name;
    WORD parent;
    WORD left;
    WORD right;
    WORD balance;
    WORD type;
    WORD number;
    PADLONG(0,6,0);
} NAMENODE;

typedef struct NaMeTree {
    NAMENODE *namenode;
    UBYTE    *namebuffer;
    LONG     nodesize;
    LONG     nodefill;
    LONG     namesize;
    LONG     namefill;
    LONG     oldnamefill;
    LONG     oldnodefill;
    LONG     globalnamefill;
    LONG     globalnodefill;
    LONG     clearnamefill;
    LONG     clearnodefill;
    WORD     headnode;
    PADPOINTER(10,0,1,0);
```

```
} NAMETREE;

typedef struct tree {
    int parent;
    int left;
    int right;
    int value;
    int blnce;
    int usage;
} COMPTREE;

typedef struct MiNmAx {
    WORD mini;
    WORD maxi;
    WORD size;
} MINMAX;

typedef struct BrAcKeTiNdEx {   /* For indexing brackets in local expressions */
    POSITION start;             /* Place where bracket starts - start of expr */
    POSITION next;              /* Place of next indexed bracket in expr */
    LONG bracket;               /* Offset of position in bracketbuffer */
    LONG termsinbracket;
    PADPOSITION(0,2,0,0,0);
} BRACKETINDEX;

typedef struct BrAcKeTiNfO {
    BRACKETINDEX *indexbuffer;
    WORD  *bracketbuffer;
    LONG  bracketbuffersize;
    LONG  indexbuffersize;
    LONG  bracketfill;
    LONG  indexfill;
    WORD  SortType;
    PADPOINTER(4,0,1,0);
} BRACKETINFO;

typedef struct TaBlEs {
    WORD    *tablepointers;
#ifdef WITHPTHREADS
    WORD    **prototype;
    WORD    **pattern;
#else
    WORD    *prototype;
    WORD    *pattern;
#endif
    MINMAX  *mm;
    WORD    *flags;
    COMPTREE *boomlijst;
    UBYTE   *argtail;
    struct TaBlEs *spare;
    WORD    *buffers;
    LONG    totind;
    LONG    reserved;
    LONG    defined;
    LONG    mdefined;
    int     prototypeSize;
    int     numind;
    int     bounds;
    int     strict;
    int     sparse;
    int     numtree;
    int     rootnum;
    int     MaxTreeSize;
    WORD    bufnum;
    WORD    bufferssize;
    WORD    buffersfill;
    WORD    tablenum;
    WORD    mode;
    WORD    numdummies;
    PADPOINTER(4,8,6,0);
} *TABLES;

typedef struct ExPrEsSiOn {
    POSITION    onfile;
    POSITION    prototype;
    POSITION    size;
    RENUMBER renum;         /* For Renumbering of global stored expressions */
    BRACKETINFO *bracketinfo;
    BRACKETINFO *newbracketinfo;
    WORD    *renumlists;
    WORD    *inmem;         /* If in memory like e.g. a polynomial */
    LONG    counter;
    LONG    name;
    WORD    hidelevel;
    WORD    vflags;         /* Various flags */
    WORD    printflag;
    WORD    status;
```

```
    WORD    replace;
    WORD    node;
    WORD    whichbuffer;
    WORD    namesize;
    WORD    compression;
    WORD    numdummies;
    WORD    numfactors;
    WORD    sizeprototype;
#ifdef PARALLELCODE
    WORD    partodo;        /* Whether to be done in parallel mode */
    PADPOSITION(5,2,0,13,0);
#else
    PADPOSITION(5,2,0,12,0);
#endif
} *EXPRESSIONS;

typedef struct SyMbOl {           /* Don't change unless altering .sav too */
    LONG    name;               /* Location in names buffer */
    WORD    minpower;           /* Minimum power admissible */
    WORD    maxpower;           /* Maximum power admissible */
    WORD    complex;            /* Properties wrt complex conjugation */
    WORD    number;             /* Number when stored in file */
    WORD    flags;              /* Used to indicate usage when storing */
    WORD    node;
    WORD    namesize;
    WORD    dimension;          /* For dimensionality checks */
    PADLONG(0,8,0);
} *SYMBOLS;

typedef struct InDeX {         /* Don't change unless altering .sav too */
    LONG    name;               /* Location in names buffer */
    WORD    type;               /* Regular or dummy */
    WORD    dimension;          /* Value of d_(n,n) or -number of symbol */
    WORD    number;             /* Number when stored in file */
    WORD    flags;              /* Used to indicate usage when storing */
    WORD    nmin4;              /* Used for n-4 if dimension < 0 */
    WORD    node;
    WORD    namesize;
    PADLONG(0,7,0);
} *INDICES;

typedef struct VeCtOr {           /* Don't change unless altering .sav too */
    LONG    name;               /* Location in names buffer */
    WORD    complex;            /* Properties under complex conjugation */
    WORD    number;             /* Number when stored in file */
    WORD    flags;              /* Used to indicate usage when storing */
    WORD    node;
    WORD    namesize;
    WORD    dimension;          /* For dimensionality checks */
    PADLONG(0,6,0);
} *VECTORS;

typedef struct FuNcTiOn {  /* Don't change unless altering .sav too */
    TABLES  tabl;
    LONG    symminfo;
    LONG    name;
    WORD    commute;
    WORD    complex;
    WORD    number;
    WORD    flags;
    WORD    spec;
    WORD    symmetric;
    WORD    node;
    WORD    namesize;
    WORD    dimension;     /* For dimensionality checks */
    WORD    maxnumargs;
    WORD    minnumargs;
    PADPOINTER(2,0,11,0);
} *FUNCTIONS;

typedef struct SeTs {
    LONG    name;               /* Location in names buffer */
    WORD    type;               /* Symbol, vector, index or function */
    WORD    first;              /* First element in setstore */
    WORD    last;               /* Last element in setstore (excluding) */
    WORD    node;
    WORD    namesize;
    WORD    dimension;          /* For dimensionality checks */
    PADLONG(0,6,0);
} *SETS;

typedef struct DuBiOuS {     /* Undeclared objects. Just for compiler. */
    LONG    name;               /* Location in names buffer */
    WORD    node;
    WORD    dummy;
    PADLONG(0,2,0);
} *DUBIOUSV;
```

```
typedef struct FaCdOlLaR {
    WORD    *where;             /* A pointer(!) to the content */
    LONG    size;
    WORD    type;               /* Type can be DOLNUMBER or DOLTERMS */
    WORD    value;              /* in case it is a (short) number */
    PADPOINTER(1,0,2,0);
} FACDOLLAR;

typedef struct DoLlArS {
    WORD    *where;             /* A pointer(!) to the object */
    FACDOLLAR *factors;            /* an array of factors. nfactors elements */
#ifdef WITHPTHREADS
    pthread_mutex_t pthreadslockread;
    pthread_mutex_t pthreadslockwrite;
#endif
    LONG    size;               /* The number of words */
    LONG    name;
    WORD    type;
    WORD    node;
    WORD    index;
    WORD    zero;
    WORD    numdummies;
    WORD    nfactors;
#ifdef WITHPTHREADS
    PADPOINTER(2,0,6,sizeof(pthread_mutex_t)*2);
#else
    PADPOINTER(2,0,6,0);
#endif
} *DOLLARS;

typedef struct MoDoPtDoLlArS {
#ifdef WITHPTHREADS
    DOLLARS  dstruct;    /* If local dollar: list of DOLLARS for each thread */
#endif
    WORD    number;
    WORD    type;
#ifdef WITHPTHREADS
    PADPOINTER(0,0,2,0);
#endif
} MODOPTDOLLAR;

typedef struct fixedset {
    char *name;
    char *description;
    int type;
    int dimension;
} FIXEDSET;

typedef struct TaBlEbAsEsUbInDeX {
    POSITION where;
    LONG size;
    PADPOSITION(0,1,0,0,0);
} TABLEBASESUBINDEX;

typedef struct TaBlEbAsE {
    POSITION fillpoint;
    POSITION current;
    UBYTE *name;
    int *tablenumbers;      /* Number of each table */
    TABLEBASESUBINDEX *subindex;       /* For each table */
    int numtables;
    PADPOSITION(3,0,1,0,0);
} TABLEBASE;

typedef struct {
    WORD *location;
    int numargs;
    int numfunnies;
    int numwildcards;
    int symmet;
    int tensor;
    int commute;
    PADPOINTER(0,6,0,0);
} FUN_INFO;

/*
    #] Variables :
    #[ Files :
*/

typedef struct FiLe {
    POSITION POposition;        /* File position */
    POSITION filesize;          /* Because SEEK_END is unsafe on IBM */
    WORD *PObuffer;             /* Address of the intermediate buffer */
    WORD *POstop;               /* End of the buffer */
    WORD *POfill;               /* Fill position of the buffer */
```

```
    WORD *POfull;                /* Full buffer when only cached */
#ifdef WITHPTHREADS
    WORD *wPObuffer;              /* Address of the intermediate worker buffer */
    WORD *wPOstop;                /* End of the worker buffer */
    WORD *wPOfill;                /* Fill position of the worker buffer */
    WORD *wPOfull;                /* Full buffer when only worker cached */
#endif
    char *name;                  /* name of the file */
#ifdef WITHZLIB
    z_streamp zsp;               /* The pointer to the stream struct for gzip */
    Bytef *ziobuffer;            /* The output buffer for compression */
#endif
    ULONG numblocks;             /* Number of blocks in file */
    ULONG inbuffer;              /* Block in the buffer */
    LONG POsize;                 /* size of the buffer */
#ifdef WITHZLIB
    LONG ziosize;                /* size of the zoutbuffer */
#endif
#ifdef WITHPTHREADS
    LONG wPOsize;                 /* size of the worker buffer */
    pthread_mutex_t pthreadslock;
#endif
    int handle;
    int active;                  /* File is open or closed. Not used. */
#ifdef WITHPTHREADS
#ifdef WITHZLIB
    PADPOSITION(11,5,2,0,sizeof(pthread_mutex_t));
#else
    PADPOSITION(9,4,2,0,sizeof(pthread_mutex_t));
#endif
#else
#ifdef WITHZLIB
    PADPOSITION(7,4,2,0,0);
#else
    PADPOSITION(5,3,2,0,0);
#endif
#endif
} FILEHANDLE;

typedef struct StreaM {
    off_t fileposition;
    off_t linenumber;
    off_t prevline;
    UBYTE *buffer;
    UBYTE *pointer;
    UBYTE *top;
    UBYTE *FoldName;
    UBYTE *name;
    UBYTE *pname;
    LONG buffersize;
    LONG bufferposition;
    LONG inbuffer;
    int previous;
    int handle;
    int type;
    int prevars;
    int previousNoShowInput;
    int eqnum;
    int afterwards;
    int olddelay;
    int oldnoshowinput;
    UBYTE isnextchar;
    UBYTE nextchar[2];
    UBYTE reserved;
    PADPOSITION(6,3,9,0,4);
} STREAM;

typedef struct SpecTatoR {
    POSITION position;   /* The place where we will be writing */
    POSITION readpos;    /* The place from which we read */
    FILEHANDLE *fh;
    char *name;          /* We identify the spectator by the name of the expression */
    WORD exprnumber;     /* During running we use the number. */
    WORD flags;          /* local, global? */
    PADPOSITION(2,0,0,2,0);
} SPECTATOR;

/*
    #] Files :
    #[ Traces :
*/

typedef struct TrAcEs {           /* For computing 4 dimensional traces */
    WORD        *accu;      /* NUMBER * 2 */
    WORD        *accup;
    WORD        *termp;
    WORD        *perm;      /* number */
```

```
    WORD        *inlist;    /* number */
    WORD        *nt3;       /* number/2 */
    WORD        *nt4;       /* number/2 */
    WORD        *j3;        /* number*2 */
    WORD        *j4;        /* number*2 */
    WORD        *e3;        /* number*2 */
    WORD        *e4;        /* number */
    WORD        *eers;      /* number/2 */
    WORD        *mepf;      /* number/2 */
    WORD        *mdel;      /* number/2 */
    WORD        *pepf;      /* number*2 */
    WORD        *pdel;      /* number*3/2 */
    WORD        sgn;
    WORD        stap;
    WORD        step1,kstep,mdum;
    WORD        gamm,ad,a3,a4,lc3,lc4;
    WORD        sign1,sign2,gamma5,num,level,factor,allsign;
    WORD        finalstep;
    PADPOINTER(0,0,19,0);
} TRACES;

typedef struct TrAcEn {           /* For computing n dimensional traces */
    WORD        *accu;      /* NUMBER */
    WORD        *accup;
    WORD        *termp;
    WORD        *perm;      /* number */
    WORD        *inlist;    /* number */
    WORD        sgn,num,level,factor,allsign;
    PADPOINTER(0,0,5,0);
} *TRACEN;

/*
    #] Traces :
    #[ Preprocessor :
*/

typedef struct pReVaR {
    UBYTE *name;
    UBYTE *value;
    UBYTE *argnames;
    int nargs;
    int wildarg;
    PADPOINTER(0,2,0,0);
} PREVAR;

typedef struct {
    WORD *buffer;
    int oldcompiletype;
    int oldparallelflag;
    int oldnumpotmoddollars;
    WORD size;
    WORD numdollars;
    WORD oldcbuf;
    WORD oldrbuf;
    WORD inscbuf;
    WORD oldcnumlhs;
    PADPOINTER(0,3,6,0);
} INSIDEINFO;

typedef struct {
    UBYTE   *buffer;
    LONG    size;
    PADPOINTER(1,0,0,0);
} PRELOAD;

typedef struct {
    PRELOAD p;
    UBYTE   *name;
    int     loadmode;
    PADPOINTER(0,1,0,0);
} PROCEDURE;

typedef struct DoLoOp {
    PRELOAD p;
    UBYTE *name;
    UBYTE *vars;        /* for {} or name of expression */
    UBYTE *contents;
    UBYTE *dollarname;
    LONG startlinenumber;
    LONG firstnum;
    LONG lastnum;
    LONG incnum;
    int type;
    int NoShowInput;
    int errorsinloop;
    int firstloopcall;
    WORD firstdollar;   /* When >= 0 we have to get the value from a dollar */
```

```
    WORD lastdollar;    /* When >= 0 we have to get the value from a dollar */
    WORD incdollar;     /* When >= 0 we have to get the value from a dollar */
    WORD NumPreTypes;
    WORD PreIfLevel;
    WORD PreSwitchLevel;
    PADPOINTER(4,4,6,0);
} DOLOOP;

struct  bit_field {    /* Assume 8 bits per byte */
    UBYTE bit_0        : 1;
    UBYTE bit_1        : 1;
    UBYTE bit_2        : 1;
    UBYTE bit_3        : 1;
    UBYTE bit_4        : 1;
    UBYTE bit_5        : 1;
    UBYTE bit_6        : 1;
    UBYTE bit_7        : 1;
/*
    UINT bit_0        : 1;
    UINT bit_1        : 1;
    UINT bit_2        : 1;
    UINT bit_3        : 1;
    UINT bit_4        : 1;
    UINT bit_5        : 1;
    UINT bit_6        : 1;
    UINT bit_7        : 1;
*/
};

typedef struct bit_field set_of_char[32];

typedef struct bit_field *one_byte;

typedef struct {
    WORD newlogonly;
    WORD newhandle;
    WORD oldhandle;
    WORD oldlogonly;
    WORD oldprinttype;
    WORD oldsilent;
} HANDLERS;

/*
    #] Preprocessor :
    #[ Varia :
*/

typedef struct CbUf {
    WORD *Buffer;
    WORD *Top;
    WORD *Pointer;
    WORD **lhs;
    WORD **rhs;
    LONG *CanCommu;
    LONG *NumTerms;
    WORD *numdum;
    WORD *dimension;
    COMPTREE *boomlijst;
    LONG BufferSize;
    int numlhs;
    int numrhs;
    int maxlhs;
    int maxrhs;
    int mnumlhs;
    int mnumrhs;
    int numtree;
    int rootnum;
    int MaxTreeSize;
    PADPOINTER(1,9,0,0);
} CBUF;

typedef struct ChAnNeL {
    char *name;
    int handle;
    PADPOINTER(0,1,0,0);
} CHANNEL;

typedef struct {
    UBYTE *parameter;
    int type;
    int flags;
    LONG value;
} SETUPPARAMETERS;

typedef struct NeStInG {
    WORD *termsize;
    WORD *funsize;
```

```
    WORD *argsize;
} *NESTING;

typedef struct StOrEcAcHe {
    POSITION position;
    POSITION toppos;
    struct StOrEcAcHe *next;
    WORD buffer[2];
    PADPOSITION(1,0,0,2,0);
} *STORECACHE;

typedef struct PeRmUtE {
    WORD *objects;
    WORD sign;
    WORD n;
    WORD cycle[MAXMATCH];
    PADPOINTER(0,0,MAXMATCH+2,0);
} PERM;

typedef struct PeRmUtEp {
    WORD **objects;
    WORD sign;
    WORD n;
    WORD cycle[MAXMATCH];
    PADPOINTER(0,0,MAXMATCH+2,0);
} PERMP;

typedef struct DiStRiBuTe {
    WORD *obj1;
    WORD *obj2;
    WORD *out;
    WORD sign;
    WORD n1;
    WORD n2;
    WORD n;
    WORD cycle[MAXMATCH];
    PADPOINTER(0,0,(MAXMATCH+4),0);
} DISTRIBUTE;

typedef struct PaRtI {
    WORD *psize;  /* the sizes of the partitions */
    WORD *args;   /* the offsets of the arguments to be partitioned */
    WORD *nargs;  /* argument numbers (different number = different argument) */
    WORD *nfun;   /* the functions into which the partitions go */
    WORD numargs; /* the number of arguments to be partitioned */
    WORD numpart; /* the number of partitions */
    WORD where;   /* offset of the function in the term */
    PADPOINTER(0,0,3,0);
} PARTI;

typedef struct sOrT {
    FILEHANDLE file;            /* The own sort file */
    POSITION SizeInFile[3];     /* Sizes in the various files */
    WORD *lBuffer;              /* The large buffer */
    WORD *lTop;                 /* End of the large buffer */
    WORD *lFill;                /* The filling point of the large buffer */
    WORD *used;                 /*  auxiliary during actual sort */
    WORD *sBuffer;              /* The small buffer */
    WORD *sTop;                 /* End of the small buffer */
    WORD *sTop2;                /* End of the extension of the small buffer */
    WORD *sHalf;                /* Halfway point in the extension */
    WORD *sFill;                /* Filling point in the small buffer */
    WORD **sPointer;            /* Pointers to terms in the small buffer */
    WORD **PoinFill;            /* Filling point for pointers to the sm.buf */
    WORD **SplitScratch;        /* Excess pointer space for the merge sort */
    WORD *cBuffer;              /* Compress buffer (if it exists) */
    WORD **Patches;             /* Positions of patches in large buffer */
    WORD **pStop;               /* Ends of patches in the large buffer */
    WORD **poina;               /*  auxiliary during actual sort */
    WORD **poin2a;              /*  auxiliary during actual sort */
    WORD *ktoi;                 /*  auxiliary during actual sort */
    WORD *tree;                 /*  auxiliary during actual sort */
#ifdef WITHZLIB
    WORD *fpcompressed;         /*  is output filepatch compressed? */
    WORD *fpincompressed;       /*  is input filepatch compressed? */
    z_streamp zsparray;         /*  the array of zstreams for decompression */
#endif
    POSITION *fPatches;         /* Positions of output file patches */
    POSITION *inPatches;        /* Positions of input file patches */
    POSITION *fPatchesStop;     /* Positions of output file patches */
    POSITION *iPatches;         /* Input file patches, Points to fPatches or inPatches */
    FILEHANDLE *f;              /* The actual output file */
    FILEHANDLE **ff;            /* Handles for a staged sort */
    LONG sTerms;                /* Terms in small buffer */
    LONG LargeSize;             /* Size of large buffer (in words) */
    LONG SmallSize;             /* Size of small buffer (in words) */
    LONG SmallEsize;            /* Size of small + extension (in words) */
```

```
    LONG TermsInSmall;          /* Maximum number of terms in small buffer */
    LONG Terms2InSmall;         /* with extension for polyfuns etc. */
    LONG GenTerms;              /* Number of generated terms */
    LONG TermsLeft;             /* Number of terms still in existence */
    LONG GenSpace;              /* Amount of space of generated terms */
    LONG SpaceLeft;             /* Space needed for still existing terms */
    LONG putinsize;             /* Size of buffer in putin */
    LONG ninterms;              /* Which input term ? */
    int MaxPatches;             /* Maximum number of patches in large buffer */
    int MaxFpatches;            /* Maximum number of patches in one filesort */
    int type;                   /* Main, function or sub(routine) */
    int lPatch;                 /* Number of patches in the large buffer */
    int fPatchN1;               /* Number of patches in input file */
    int PolyWise;               /* Is there a polyfun and if so, where? */
    int PolyFlag;               /*  */
    int cBufferSize;            /* Size of the compress buffer */
    int maxtermsize;            /* Keeps track for buffer allocations */
    int newmaxtermsize;         /* Auxiliary for maxtermsize */
    int outputmode;             /* Tells where the output is going */
    int stagelevel;             /* In case we have a 'staged' sort */
    WORD fPatchN;               /* Number of patches on file (output) */
    WORD inNum;                 /* Number of patches on file (input) */
    WORD stage4;                /* Are we using stage4? */
#ifdef WITHZLIB
    PADPOSITION(28,12,12,3,0);
#else
    PADPOSITION(25,12,12,3,0);
#endif
} SORTING;

#ifdef WITHPTHREADS

typedef struct SoRtBlOcK {
    pthread_mutex_t *MasterBlockLock;
    WORD    **MasterStart;
    WORD    **MasterFill;
    WORD    **MasterStop;
    int     MasterNumBlocks;
    int     MasterBlock;
    int     FillBlock;
    PADPOINTER(0,3,0,0);
} SORTBLOCK;
#endif

#ifdef DEBUGGER
typedef struct DeBuGgInG {
    int eflag;
    int printflag;
    int logfileflag;
    int stdoutflag;
} DEBUGSTR;
#endif

#ifdef WITHPTHREADS

typedef struct ThReAdBuCkEt {
    POSITION *deferbuffer;      /* For Keep Brackets: remember position */
    WORD *threadbuffer;         /* Here are the (primary) terms */
    WORD *compressbuffer;       /* For keep brackets we need the compressbuffer */
    LONG threadbuffersize;      /* Number of words in threadbuffer */
    LONG ddterms;               /* Number of primary+secondary terms represented */
    LONG firstterm;             /* The number of the first term in the bucket */
    LONG firstbracket;          /* When doing complete brackets */
    LONG lastbracket;           /* When doing complete brackets */
    pthread_mutex_t lock;       /* For the load balancing phase */
    int  free;                  /* Status of the bucket */
    int  totnum;                /* Total number of primary terms */
    int  usenum;                /* Which is the term being used at the moment */
    int  busy;                  /*  */
    int  type;                  /* Doing brackets? */
    PADPOINTER(5,5,0,sizeof(pthread_mutex_t));
} THREADBUCKET;

#endif

typedef struct {
    WORD    *coefs;             /* The array of coefficients */
    WORD    numsym;             /* The number of the symbol in the polynomial */
    WORD    arraysize;          /* The size of the allocation of coefs */
    WORD    polysize;           /* The maximum power in the polynomial */
    WORD    modnum;             /* The prime number of the modulus */
} POLYMOD;

typedef struct {
    WORD    *outterm;            /* Used in DoShuffle/Merge/FinishShuffle system */
    WORD    *outfun;
    WORD    *incoef;
```

```
    WORD    *stop1;
    WORD    *stop2;
    WORD    *ststop1;
    WORD    *ststop2;
    void    *finishuf;
    void    *do_uffle;
    LONG    combilast;
    WORD    nincoef;
    WORD    level;
    WORD    thefunction;
    WORD    option;
    PADPOINTER(1,0,4,0);
} SHvariables;

typedef struct {                /* Used for computing calculational cost in optim.c */
    LONG    add;
    LONG    mul;
    LONG    div;
    LONG    pow;
} COST;

typedef struct {
    UWORD   *a;     /* The number array */
    UWORD   *m;     /* The modulus array */
    WORD    na;     /* Size of the number */
    WORD    nm;     /* size of the number in the modulus array */
} MODNUM;

/*
    Struct for optimizing outputs. If changed, do not forget to change
    the padding information in the AO struct.
*/
typedef struct {
    union { /* we do this to allow padding */
        float fval;
        int ival[2]; /* This should be enough */
    } mctsconstant;
    int   horner;
    int   hornerdirection;
    int   method;
    int   mctstimelimit;
    int   mctsnumexpand;
    int   mctsnumkeep;
    int   mctsnumrepeat;
    int   greedytimelimit;
    int   greedyminnum;
    int   greedymaxperc;
    int   printstats;
    int   debugflags;
    int   schemeflags;
    int   mctsdecaymode;
    int   saIter; /* Simulated annealing updates */
    union {
        float fval;
        int ival[2];
    } saMaxT; /* Maximum temperature of SA */
    union {
        float fval;
        int ival[2];
    } saMinT; /* Minimum temperature of SA */
} OPTIMIZE;

typedef struct {
    WORD  *code;
    UBYTE *nameofexpr;  /* It is easier to remember an expression by name */
    LONG  codesize;     /* We need this for the checkpoints */
    WORD  exprnr;       /* Problem here is: we renumber them in execute.c */
    WORD  minvar;
    WORD  maxvar;

    PADPOSITION(2,1,0,3,0);
} OPTIMIZERESULT;

typedef struct {
    WORD *lhs;      /* Object to be replaced */
    WORD *rhs;      /* Depending on the type it will be UBYTE* or WORD* */
    int type;
    int size;       /* Size of the lhs */
} DICTIONARY_ELEMENT;

typedef struct {
    DICTIONARY_ELEMENT **elements;
    UBYTE *name;
    int sizeelements;
    int numelements;
    int numbers;        /* deal with numbers */
    int variables;      /* deal with single variables */
```

```
    int characters;     /* deal with special characters */
    int funwith;        /* deal with functions with arguments */
    int gnumelements;   /* if .global shrinks the dictionary */
    int ranges;
} DICTIONARY;

/*
    #] Varia :
    #[ A :
        #[ M : The M struct is for global settings at startup or .clear
*/

struct M_const {
    POSITION zeropos;              /* (M) is zero */
    SORTING *S0;
    UWORD   *gcmod;
    UWORD   *gpowmod;
    UBYTE   *TempDir;              /* (M) Path with where the temporary files go */
    UBYTE   *TempSortDir;          /* (M) Path with where the sort files go */
    UBYTE   *IncDir;               /* (M) Directory path for include files */
    UBYTE   *InputFileName;        /* (M) */
    UBYTE   *LogFileName;          /* (M) */
    UBYTE   *OutBuffer;            /* (M) Output buffer in pre.c */
    UBYTE   *Path;                 /* (M) */
    UBYTE   *SetupDir;             /* (M) Directory with setup file */
    UBYTE   *SetupFile;            /* (M) Name of setup file */
    UBYTE   *gFortran90Kind;
    UBYTE   *gextrasym;
    UBYTE   *ggextrasym;
    UBYTE   *oldnumextrasymbols;
    SPECTATOR *SpectatorFiles;
#ifdef WITHPTHREADS
    pthread_rwlock_t handlelock;   /* (M) */
    pthread_mutex_t storefilelock; /* (M) */
    pthread_mutex_t sbuflock;      /* (M) Lock for writing in the AM.sbuffer */
    LONG    ThreadScratSize;       /* (M) Size of Fscr[0/2] buffers of the workers */
    LONG    ThreadScratOutSize;    /* (M) Size of Fscr[1] buffers of the workers */
#endif
    LONG    MaxTer;                /* (M) Maximum term size. Fixed at setup. In Bytes!!!*/
    LONG    CompressSize;          /* (M) Size of Compress buffer */
    LONG    ScratSize;             /* (M) Size of Fscr[] buffers */
    LONG    HideSize;              /* (M) Size of Fscr[2] buffer */
    LONG    SizeStoreCache;        /* (M) Size of the chaches for reading global expr. */
    LONG    MaxStreamSize;         /* (M) Maximum buffer size in reading streams */
    LONG    SIOsize;               /* (M) Sort InputOutput buffer size */
    LONG    SLargeSize;            /* (M) */
    LONG    SSmallEsize;           /* (M) */
    LONG    SSmallSize;            /* (M) */
    LONG    STermsInSmall;         /* (M) */
    LONG    MaxBracketBufferSize;  /* (M) Max Size for B+ or AB+ per expression */
    LONG    hProcessBucketSize;    /* (M) */
    LONG    gProcessBucketSize;    /* (M) */
    LONG    shmWinSize;            /* (M) size for shared memory window used in communications */
    LONG    OldChildTime;          /* (M) Zero time. Needed in timer. */
    LONG    OldSecTime;            /* (M) Zero time for measuring wall clock time */
    LONG    OldMilliTime;          /* (M) Same, but milli seconds */
    LONG    WorkSize;              /* (M) Size of WorkSpace */
    LONG    gThreadBucketSize;     /* (C) */
    LONG    ggThreadBucketSize;    /* (C) */
    LONG    SumTime;               /*     Used in .clear */
    LONG    SpectatorSize;         /*     Size of the buffer in bytes */
    int     FileOnlyFlag;          /* (M) Writing only to file */
    int     Interact;              /* (M) Interactive mode flag */
    int     MaxParLevel;           /* (M) Maximum nesting of parantheses */
    int     OutBufSize;            /* (M) size of OutBuffer */
    int     SMaxFpatches;          /* (M) */
    int     SMaxPatches;           /* (M) */
    int     StdOut;                /* (M) Regular output channel */
    int     ginsidefirst;          /* (M) Not used yet */
    int     gDefDim;               /* (M) */
    int     gDefDim4;              /* (M) */
    int     NumFixedSets;          /* (M) Number of the predefined sets */
    int     NumFixedFunctions;     /* (M) Number of built in functions */
    int     rbufnum;               /* (M) startup compiler buffer */
    int     dbufnum;               /* (M) dollar variables */
    int     sbufnum;               /* (M) subterm variables */
    int     zbufnum;               /* (M) special values */
    int     SkipClears;            /* (M) Number of .clear to skip at start */
    int     gTokensWriteFlag;      /* (M) */
    int     gfunpowers;            /* (M) */
    int     gStatsFlag;            /* (M) */
    int     gNamesFlag;            /* (M) */
    int     gCodesFlag;            /* (M) */
    int     gSortType;             /* (M) */
    int     gproperorderflag;      /* (M) */
    int     hparallelflag;         /* (M) */
    int     gparallelflag;         /* (M) */
```

```
int     totalnumberofthreads;  /* (M) */
int     gSizeCommuteInSet;
int     gThreadStats;
int     ggThreadStats;
int     gFinalStats;
int     ggFinalStats;
int     gThreadsFlag;
int     ggThreadsFlag;
int     gThreadBalancing;
int     ggThreadBalancing;
int     gThreadSortFileSynch;
int     ggThreadSortFileSynch;
int     gProcessStats;
int     ggProcessStats;
int     gOldParallelStats;
int     ggOldParallelStats;
int     maxFlevels;            /* () maximum function levels */
int     resetTimeOnClear;      /* (M) */
int     gcNumDollars;          /* () number of dollars for .clear */
int     MultiRun;
int     gNoSpacesInNumbers;    /* For very long numbers */
int     ggNoSpacesInNumbers;   /* For very long numbers */
int     gIsFortran90;
int     PrintTotalSize;
int     fbuffersize;           /* Size for the AT.fbufnum factorization caches */
int     gOldFactArgFlag;
int     ggOldFactArgFlag;
int     gnumextrasym;
int     ggnumextrasym;
int     NumSpectatorFiles;     /* Elements used in AM.spectatorfiles; */
int     SizeForSpectatorFiles; /* Size in AM.spectatorfiles; */
int     gOldGCDflag;
int     ggOldGCDflag;
int     gWTimeStatsFlag;
int     ggWTimeStatsFlag;
WORD    MaxTal;                /* (M) Maximum number of words in a number */
WORD    IndDum;                /* (M) Basis value for dummy indices */
WORD    DumInd;                /* (M) */
WORD    WilInd;                /* (M) Offset for wildcard indices */
WORD    gncmod;                /* (M) Global setting of modulus. size of gcmod */
WORD    gnpowmod;              /* (M) Global printing as powers. size gpowmod */
WORD    gmodmode;              /* (M) Global mode for modulus */
WORD    gUnitTrace;            /* (M) Global value of Tr[1] */
WORD    gOutputMode;           /* (M) */
WORD    gOutputSpaces;         /* (M) */
WORD    gOutNumberType;        /* (M) */
WORD    gCnumpows;             /* (M) */
WORD    gUniTrace[4];          /* (M) */
WORD    MaxWildcards;          /* (M) Maximum number of wildcards */
WORD    mTraceDum;             /* (M) Position/Offset for generated dummies */
WORD    OffsetIndex;           /* (M) */
WORD    OffsetVector;          /* (M) */
WORD    RepMax;                /* (M) Max repeat levels */
WORD    LogType;               /* (M) Type of writing to log file */
WORD    ggStatsFlag;           /* (M) */
WORD    gLineLength;           /* (M) */
WORD    qError;                /* (M) Only error checking {-c option} */
WORD    FortranCont;           /* (M) Fortran Continuation character */
WORD    HoldFlag;              /* (M) Exit on termination? */
WORD    Ordering[15];          /* (M) Auxiliary for ordering wildcards */
WORD    silent;                /* (M) Silent flag. Only results in output. */
WORD    tracebackflag;         /* (M) For tracing errors */
WORD    expnum;                /* (M) internal number of ^ function */
WORD    denomnum;              /* (M) internal number of / function */
WORD    facnum;                /* (M) internal number of fac_ function */
WORD    invfacnum;             /* (M) internal number of invfac_ function */
WORD    sumnum;                /* (M) internal number of sum_ function */
WORD    sumpnum;               /* (M) internal number of sump_ function */
WORD    OldOrderFlag;          /* (M) Flag for allowing old statement order */
WORD    termfunnum;            /* (M) internal number of term_ function */
WORD    matchfunnum;           /* (M) internal number of match_ function */
WORD    countfunnum;           /* (M) internal number of count_ function */
WORD    gPolyFun;              /* (M) global value of PolyFun */
WORD    gPolyFunInv;           /* (M) global value of Inverse of PolyFun */
WORD    gPolyFunType;          /* (M) global value of PolyFun */
WORD    gPolyFunExp;
WORD    gPolyFunVar;
WORD    gPolyFunPow;
WORD    dollarzero;            /* (M) for dollars with zero value */
WORD    atstartup;             /* To protect against DATE_ ending in \n */
WORD    exitflag;              /* (R) For the exit statement */
WORD    NumStoreCaches;        /* () Number of storage caches per processor */
WORD    gIndentSpace;          /* For indentation in output */
WORD    ggIndentSpace;
WORD    gShortStatsMax;
WORD    ggShortStatsMax;
WORD    gextrasymbols;
```

```
    WORD    ggextrasymbols;
    WORD    zerorhs;
    WORD    onerhs;
    WORD    havesortdir;
    WORD    BracketFactors[8];
#ifdef WITHPTHREADS
    PADPOSITION(17,25,61,81,sizeof(pthread_rwlock_t)+sizeof(pthread_mutex_t)*2);
#else
    PADPOSITION(17,23,61,81,0);
#endif
};
/*
        #] M :
        #[ P : The P struct defines objects set by the preprocessor
*/
struct P_const {
    LIST DollarList;                /* (R) Dollar variables. Contains pointers
                                        to contents of the variables.*/
    LIST PreVarList;                /* (R) List of preprocessor variables
                                        Points to contents. Can be changed */
    LIST LoopList;                  /* (P) List of do loops */
    LIST ProcList;                  /* (P) List of procedures */
    INSIDEINFO inside;              /*     Information during #inside/#endinside */
    UBYTE **PreSwitchStrings;       /* (P) The string in a switch */
    UBYTE *preStart;                /* (P) Preprocessor instruction buffer */
    UBYTE *preStop;                 /* (P) end of preStart */
    UBYTE *preFill;                 /* (P) Filling point in preStart */
    UBYTE *procedureExtension;      /* (P) Extension for procedure files (prc) */
    UBYTE *cprocedureExtension;     /* (P) Extension after .clear */
    LONG  *PreAssignStack;          /* For nesting #$name assignments */
    int *PreIfStack;                /* (P) Tracks nesting of #if */
    int *PreSwitchModes;            /* (P) Stack of switch status */
    int *PreTypes;                  /* (P) stack of #call, #do etc nesting */
#ifdef WITHPTHREADS
    pthread_mutex_t PreVarLock;     /* (P) */
#endif
    LONG    StopWatchZero;          /* For `timer_' and #reset timer */
    LONG    InOutBuf;               /* (P) Characters in the output buf in pre.c */
    LONG    pSize;                  /* (P) size of preStart */
    int     PreAssignFlag;          /* (C) Indicates #assign -> catch dollar */
    int     PreContinuation;        /* (C) Indicates whether the statement is new */
    int     PreproFlag;             /* (P) Internal use to mark work on prepro instr. */
    int     iBufError;              /* (P) Flag for errors with input buffer */
    int     PreOut;                 /* (P) Flag for #+ #- */
    int     PreSwitchLevel;         /* (P) Nesting of #switch */
    int     NumPreSwitchStrings;    /* (P) Size of PreSwitchStrings */
    int     MaxPreTypes;            /* (P) Size of PreTypes */
    int     NumPreTypes;            /* (P) Number of nesting objects in PreTypes */
    int     MaxPreIfLevel;          /* (C) Maximum number of nested #if. Dynamic */
    int     PreIfLevel;             /* (C) Current position if PreIfStack */
    int     PreInsideLevel;         /* (C) #inside active? */
    int     DelayPrevar;            /* (P) Delaying prevar substitution */
    int     AllowDelay;             /* (P) Allow delayed prevar substitution */
    int     lhdollarerror;          /* (R) */
    int     eat;                    /* () */
    int     gNumPre;                /* (P) Number of preprocessor variables for .clear */
    int     PreDebug;               /* (C) */
    int     OpenDictionary;
    int     PreAssignLevel;         /* For nesting #$name = ...; assignments */
    int     MaxPreAssignLevel;      /* For nesting #$name = ...; assignments */
    WORD    DebugFlag;              /* (P) For debugging purposes */
    WORD    preError;               /* (P) Blocks certain types of execution */
    UBYTE   ComChar;                /* (P) Commentary character */
    UBYTE   cComChar;               /* (P) Old commentary character for .clear */
    PADPOINTER(3,21,2,2);
};

/*
        #] P :
        #[ C : The C struct defines objects changed by the compiler
*/

struct C_const {
    set_of_char separators;
    POSITION StoreFileSize;         /* () Size of store file */
    NAMETREE *dollarnames;
    NAMETREE *exprnames;
    NAMETREE *varnames;
    LIST    ChannelList;
                                    /*     Later also for write? */
    LIST    DubiousList;
    LIST    FunctionList;
    LIST    ExpressionList;
    LIST    IndexList;
    LIST    SetElementList;
    LIST    SetList;
    LIST    SymbolList;
```

```
    LIST    VectorList;
    LIST    PotModDolList;
    LIST    ModOptDolList;
    LIST    TableBaseList;
/*
    Compile buffer variables
*/
    LIST    cbufList;
/*
    Objects for auto declarations
*/
    LIST    AutoSymbolList;         /* (C) */
    LIST    AutoIndexList;          /* (C) */
    LIST    AutoVectorList;         /* (C) */
    LIST    AutoFunctionList;       /* (C) */
    NAMETREE *autonames;
    LIST    *Symbols;               /* (C) Pointer for autodeclare. Which list is
                                       it searching. Later also for subroutines */
    LIST    *Indices;               /* (C) id. */
    LIST    *Vectors;               /* (C) id. */
    LIST    *Functions;             /* (C) id. */
    NAMETREE **activenames;
    STREAM  *Streams;
    STREAM  *CurrentStream;
    LONG    *termstack;
    LONG    *termsortstack;
    UWORD   *cmod;
    UWORD   *powmod;
    UWORD   *modpowers;
    UWORD   *halfmod;               /* (C) half the modulus when not zero */
    WORD    *ProtoType;             /* (C) The subexpression prototype {wildcards} */
    WORD    *WildC;                 /* (C) Filling point for wildcards. */
    LONG    *IfHeap;
    LONG    *IfCount;
    LONG    *IfStack;
    UBYTE   *iBuffer;
    UBYTE   *iPointer;
    UBYTE   *iStop;
    UBYTE   **LabelNames;
    WORD    *FixIndices;
    WORD    *termsumcheck;
    UBYTE   *WildcardNames;
    int     *Labels;
    SBYTE   *tokens;
    SBYTE   *toptokens;
    SBYTE   *endoftokens;
    WORD    *tokenarglevel;
    UWORD   *modinverses;           /* Table for inverses of primes */
    UBYTE   *Fortran90Kind;         /* The kind of number in Fortran 90 as in _ki */
    WORD    **MultiBracketBuf;      /* Array of buffers for multi-level brackets */
    UBYTE   *extrasym;              /* Array with the name for extra symbols in ToPolynomial */
    WORD    *doloopstack;           /* To keep track of begin and end of doloops */
    WORD    *doloopnest;            /* To keep track of nesting of doloops etc */
    char    *CheckpointRunAfter;
    char    *CheckpointRunBefore;
    WORD    *IfSumCheck;
    WORD    *CommuteInSet;          /* groups of noncommuting functions that can commute */
    UBYTE   *TestValue;             /* For debugging */
#ifdef PARALLELCODE
    LONG    *inputnumbers;
    WORD    *pfirstnum;
#endif
#ifdef WITHPTHREADS
    pthread_mutex_t halfmodlock;    /* () Lock for adding buffer for halfmod */
#endif
    LONG    argstack[MAXNEST];      /* (C) {contents} Stack for nesting of Argument */
    LONG    insidestack[MAXNEST];   /* (C) {contents} Stack for Argument or Inside. */
    LONG    inexprstack[MAXNEST];   /* (C) {contents} Stack for Argument or Inside. */
    LONG    iBufferSize;            /* (C) Size of the input buffer */
    LONG    TransEname;             /* (C) Used when a new definition overwrites
                                        an old expression. */
    LONG    ProcessBucketSize;      /* (C) */
    LONG    mProcessBucketSize;     /* (C) */
    LONG    CModule;                /* (C) Counter of current module */
    LONG    ThreadBucketSize;       /* (C) Roughly the maximum number of input terms */
    LONG    CheckpointStamp;
    LONG    CheckpointInterval;
    int     cbufnum;
    int     AutoDeclareFlag;
    int     NoShowInput;            /* (C) No listing of input as in .prc, #do */
    int     ShortStats;             /* (C) */
    int     compiletype;            /* (C) type of statement {DECLARATION etc} */
    int     firstconstindex;        /* (C) flag for giving first error message */
    int     insidefirst;            /* (C) Not used yet */
    int     minsidefirst;           /* (?) Not used yet */
    int     wildflag;               /* (C) Flag for marking use of wildcards */
    int     NumLabels;              /* (C) Number of labels {in Labels} */
```

```
    int     MaxLabels;              /* (C) Size of Labels array */
    int     lDefDim;                /* (C) */
    int     lDefDim4;               /* (C) */
    int     NumWildcardNames;       /* (C) Number of ?a variables */
    int     WildcardBufferSize;     /* (C) size of WildcardNames buffer */
    int     MaxIf;                  /* (C) size of IfHeap, IfSumCheck, IfCount */
    int     NumStreams;             /* (C) */
    int     MaxNumStreams;          /* (C) */
    int     firstctypemessage;      /* (C) Flag for giving first st order error */
    int     tablecheck;             /* (C) For table checking */
    int     idoption;               /* (C) */
    int     BottomLevel;            /* (C) For propercount. Not used!!! */
    int     CompileLevel;           /* (C) Subexpression level */
    int     TokensWriteFlag;        /* (C) */
    int     UnsureDollarMode;       /* (C)?Controls error messages undefined $'s */
    int     outsidefun;             /* (C) Used for writing Tables to file */
    int     funpowers;              /* (C) */
    int     WarnFlag;               /* (C) */
    int     StatsFlag;              /* (C) */
    int     NamesFlag;              /* (C) */
    int     CodesFlag;              /* (C) */
    int     SetupFlag;              /* (C) */
    int     SortType;               /* (C) */
    int     lSortType;              /* (C) */
    int     ThreadStats;            /* (C) */
    int     FinalStats;             /* (C) */
    int     OldParallelStats;       /* (C) */
    int     ThreadsFlag;
    int     ThreadBalancing;
    int     ThreadSortFileSynch;
    int     ProcessStats;           /* (C) */
    int     BracketNormalize;       /* (C) Indicates whether the bracket st is normalized */
    int     maxtermlevel;           /* (C) Size of termstack */
    int     dumnumflag;             /* (C) Where there dummy indices in tokenizer? */
    int     bracketindexflag;       /* (C) Are brackets going to be indexed? */
    int     parallelflag;           /* (C) parallel allowed? */
    int     mparallelflag;          /* (C) parallel allowed in this module? */
    int     inparallelflag;         /* (C) inparallel allowed? */
    int     partodoflag;            /* (C) parallel allowed? */
    int     properorderflag;        /* (C) clean normalizing. */
    int     vetofilling;            /* (C) vetoes overwriting in tablebase stubs */
    int     tablefilling;           /* (C) to notify AddRHS we are filling a table */
    int     vetotablebasefill;      /* (C) For the load in tablebase */
    int     exprfillwarning;        /* (C) Warning has been printed for expressions in fill statements
*/
    int     lhdollarflag;           /* (R) left hand dollar present */
    int     NoCompress;             /* (R) Controls native compression */
    int     IsFortran90;            /* Tells whether the Fortran is Fortran90 */
    int     MultiBracketLevels;     /* Number of elements in MultiBracketBuf */
    int     topolynomialflag;       /* To avoid ToPolynomial and FactArg together */
    int     ffbufnum;               /* Buffer number for user defined factorizations */
    int     OldFactArgFlag;
    int     MemDebugFlag;           /* Only used when MALLOCDEBUG in tools.c */
    int     OldGCDflag;
    int     WTimeStatsFlag;
    int     doloopstacksize;
    int     doloopplevel;
    int     CheckpointFlag;
    int     SizeCommuteInSet;       /* Size of the CommuteInSet buffer */
#ifdef PARALLELCODE
    int     numpfirstnum;           /* For redefine */
    int     sizepfirstnum;          /* For redefine */
#endif
    int     origin;                 /* Determines whether .sort or ModuleOption */
    int     vectorlikeLHS;
    WORD    argsumcheck[MAXNEST];   /* (C) Checking of nesting */
    WORD    insidesumcheck[MAXNEST];/* (C) Checking of nesting */
    WORD    inexprsumcheck[MAXNEST];/* (C) Checking of nesting */
    WORD    RepSumCheck[MAXREPEAT];/* (C) Checks nesting of repeat, if, argument */
    WORD    lUniTrace[4];           /* (C) */
    WORD    RepLevel;               /* (C) Tracks nesting of repeat. */
    WORD    arglevel;               /* (C) level of nested argument statements */
    WORD    insidelevel;            /* (C) level of nested inside statements */
    WORD    inexprlevel;            /* (C) level of nested inexpr statements */
    WORD    termlevel;              /* (C) level of nested term statements */
    WORD    MustTestTable;          /* (C) Indicates whether tables have been changed */
    WORD    DumNum;                 /* (C) */
    WORD    ncmod;                  /* (C) Local setting of modulus. size of cmod */
    WORD    npowmod;                /* (C) Local printing as powers. size powmod */
    WORD    modmode;                /* (C) Mode for modulus calculus */
    WORD    nhalfmod;               /* relevant word size of halfmod when defined */
    WORD    DirtPow;                /* (C) Flag for changes in printing mod powers */
    WORD    lUnitTrace;             /* (C) Local value of Tr[1] */
    WORD    NwildC;                 /* (C) Wildcard counter */
    WORD    ComDefer;               /* (C) defer brackets */
    WORD    CollectFun;             /* (C) Collect function number */
    WORD    AltCollectFun;          /* (C) Alternate Collect function number */
```

```
    WORD    OutputMode;            /* (C) */
    WORD    Cnumpows;
    WORD    OutputSpaces;          /* (C) */
    WORD    OutNumberType;         /* (C) Controls RATIONAL/FLOAT output */
    WORD    DidClean;              /* (C) Test whether nametree needs cleaning */
    WORD    IfLevel;               /* (C) */
    WORD    WhileLevel;            /* (C) */
    WORD    LogHandle;             /* (C) The Log File */
    WORD    LineLength;            /* (C) */
    WORD    StoreHandle;           /* (C) Handle of .str file */
    WORD    HideLevel;             /* (C) Hiding indicator */
    WORD    lPolyFun;              /* (C) local value of PolyFun */
    WORD    lPolyFunInv;           /* (C) local value of Inverse of PolyFun */
    WORD    lPolyFunType;          /* (C) local value of PolyFunType */
    WORD    lPolyFunExp;
    WORD    lPolyFunVar;
    WORD    lPolyFunPow;
    WORD    SymChangeFlag;         /* (C) */
    WORD    CollectPercentage;     /* (C) Collect function percentage */
    WORD    ShortStatsMax;         /* For  On FewerStatistics 10; */
    WORD    extrasymbols;          /* Flag for the extra symbsols output mode */
    WORD    PolyRatFunChanged;     /* Keeps track whether we changed in the compiler */
    WORD    ToBeInFactors;
    WORD    InnerTest;            /* For debugging */
#ifdef WITHMPI
    WORD    RhsExprInModuleFlag;   /* (C) Set by the compiler if RHS expressions exists. */
#endif
    UBYTE   Commercial[COMMERCIALSIZE+2]; /* (C) Message to be printed in statistics */
    UBYTE   debugFlags[MAXFLAGS+2];    /* On/Off Flag number(s) */
#if defined(WITHPTHREADS)

    PADPOSITION(47,8+3*MAXNEST,72,45+3*MAXNEST+MAXREPEAT,COMMERCIALSIZE+MAXFLAGS+4+sizeof(LIST)*17+sizeof(pthread_mutex_t)),
#elif defined(WITHMPI)
    PADPOSITION(47,8+3*MAXNEST,72,46+3*MAXNEST+MAXREPEAT,COMMERCIALSIZE+MAXFLAGS+4+sizeof(LIST)*17);
#else
    PADPOSITION(45,8+3*MAXNEST,70,45+3*MAXNEST+MAXREPEAT,COMMERCIALSIZE+MAXFLAGS+4+sizeof(LIST)*17);
#endif
};
/*
        #] C :
        #[ S : The S struct defines objects changed at the start of the run (Processor)
               Basically only set by the master.
*/
struct S_const {
    POSITION MaxExprSize;          /* ( ) Maximum size of in/out/sort */
#ifdef WITHPTHREADS
    pthread_mutex_t inputslock;
    pthread_mutex_t outputslock;
    pthread_mutex_t MaxExprSizeLock;
#endif
    POSITION *OldOnFile;           /* (S) File positions of expressions */
    WORD    *OldNumFactors;        /* ( ) NumFactors in (old) expression */
    WORD    *Oldvflags;            /* ( ) vflags in (old) expression */
    int     NumOldOnFile;          /* (S) Number of expressions in OldOnFile */
    int     NumOldNumFactors;      /* (S) Number of expressions in OldNumFactors */
    int     MultiThreaded;         /* (S) Are we running multi-threaded? */
#ifdef WITHPTHREADS
    int     MasterSort;            /* Final stage of sorting to the master */
#endif
#ifdef WITHMPI
    int     printflag;             /* controls MesPrint() on each slave */
#endif
    int     Balancing;             /* For second stage loadbalancing */
    WORD    ExecMode;              /* (S) */

    WORD    CollectOverFlag;       /* (R) Indicates overflow at Collect */
#ifdef WITHPTHREADS
    WORD    sLevel;                /* Copy of AR0.sLevel because it can get messy */
#endif
#if defined(WITHPTHREADS)
    PADPOSITION(3,0,5,3,sizeof(pthread_mutex_t)*3);
#elif defined(WITHMPI)
    PADPOSITION(3,0,5,2,0);
#else
    PADPOSITION(3,0,4,2,0);
#endif
};
/*
        #] S :
        #[ R : The R struct defines objects changed at run time.
               They determine the environment that has to be transfered
               together with a term during multithreaded execution.
*/
struct R_const {
    FILEDATA    StoreData;         /* (O) */
    FILEHANDLE  Fscr[3];           /* (R) Dollars etc play with it too */
    FILEHANDLE  FoStage4[2];       /* (R) In Sort. Stage 4. */
```

```
    POSITION DefPosition;          /* (R) Deferred position of keep brackets. */
    FILEHANDLE *infile;            /* (R) Points alternatingly to Fscr[0] or Fscr[1] */
    FILEHANDLE *outfile;           /* (R) Points alternatingly to Fscr[1] or Fscr[0] */
    FILEHANDLE *hidefile;          /* (R) Points to Fscr[2] */

    WORD    *CompressBuffer;       /* (M) */
    WORD    *ComprTop;             /* (M) */
    WORD    *CompressPointer;      /* (R) */
    VOID    *CompareRoutine;
    ULONG   *wranfia;

    LONG    OldTime;               /* (R) Zero time. Needed in timer. */
    LONG    InInBuf;               /* (R) Characters in input buffer. Scratch files. */
    LONG    InHiBuf;               /* (R) Characters in hide buffer. Scratch file. */
    LONG    pWorkSize;             /* (R) Size of pWorkSpace */
    LONG    lWorkSize;             /* (R) Size of lWorkSpace */
    LONG    posWorkSize;           /* (R) Size of posWorkSpace */
    ULONG   wranfseed;
    int     NoCompress;            /* (R) Controls native compression */
    int     gzipCompress;          /* (R) Controls gzip compression */
    int     Cnumlhs;               /* Local copy of cbuf[rbufnum].numlhs */
    int     outtohide;             /* Indicates that output is directly to hide */
#ifdef WITHPTHREADS
    int     exprtodo;              /* The expression to do in parallel mode */
#endif
    int     wranfcall;
    int     wranfnpair1;
    int     wranfnpair2;
#if ( BITSINWORD == 32 )
    WORD    PrimeList[5000];
    WORD    numinprimelist;
    WORD    notfirstprime;
#endif
    WORD    GetFile;               /* (R) Where to get the terms {like Hide} */
    WORD    KeptInHold;            /* (R) */
    WORD    BracketOn;             /* (R) Intensly used in poly_ */
    WORD    MaxBracket;            /* (R) Size of BrackBuf. Changed by poly_ */
    WORD    CurDum;                /* (R) Current maximum dummy number */
    WORD    DeferFlag;             /* (R) For defered brackets */
    WORD    TePos;                 /* (R) */
    WORD    sLevel;                /* (R) Sorting level */
    WORD    Stage4Name;            /* (R) Sorting only */
    WORD    GetOneFile;            /* (R) Getting from hide or regular */
    WORD    PolyFun;               /* (C) Number of the PolyFun function */
    WORD    PolyFunInv;            /* (C) Number of the Inverse of the PolyFun function */
    WORD    PolyFunType;           /* () value of PolyFunType */
    WORD    PolyFunExp;
    WORD    PolyFunVar;
    WORD    PolyFunPow;
    WORD    Eside;                 /* () Tells which side of = sign */
    WORD    MaxDum;                /* Maximum dummy value in an expression */
    WORD    level;                 /* Running level in Generator */
    WORD    expchanged;            /* (R) Info about expression */
    WORD    expflags;              /* (R) Info about expression */
    WORD    CurExpr;               /* (S) Number of current expression */
    WORD    SortType;              /* A copy of AC.SortType to play with */
    WORD    ShortSortCount;        /* For On FewerStatistics 10; */
#if ( BITSINWORD == 32 )
#ifdef WITHPTHREADS
    PADPOSITION(8,7,8,5026,0);
#else
    PADPOSITION(8,7,7,5026,0);
#endif
#else
#ifdef WITHPTHREADS
    PADPOSITION(8,7,8,24,0);
#else
    PADPOSITION(8,7,7,24,0);
#endif
#endif
};

/*
        #] R :
        #[ T : These are variables that stay in each thread during multi threaded execution.
*/
struct T_const {
#ifdef WITHPTHREADS
    SORTBLOCK SB;
#endif
    SORTING *S0;                   /* (-) The thread specific sort buffer */
    SORTING *SS;                   /* (R) Current sort buffer */
    NESTING     Nest;              /* (R) Nesting of function levels etc. */
    NESTING     NestStop;          /* (R) */
    NESTING     NestPoin;          /* (R) */
    WORD    *BrackBuf;             /* (R) Bracket buffer. Used by poly_ at runtime. */
    STORECACHE  StoreCache;        /* (R) Cache for picking up stored expr. */
```

```
    STORECACHE  StoreCacheAlloc;   /* (R) Permanent address of StoreCache to keep valgrind happy */
    WORD    **pWorkSpace;          /* (R) Workspace for pointers. Dynamic. */
    LONG    *lWorkSpace;           /* (R) WorkSpace for LONG. Dynamic. */
    POSITION *posWorkSpace;        /* (R) WorkSpace for file positions */
    WORD    *WorkSpace;            /* (M) */
    WORD    *WorkTop;              /* (M) */
    WORD    *WorkPointer;          /* (R) Pointer in the WorkSpace heap. */
    int     *RepCount;             /* (M) Buffer for repeat nesting */
    int     *RepTop;               /* (M) Top of RepCount buffer */
    WORD    *WildArgTaken;         /* (N) Stack for wildcard pattern matching */
    UWORD   *factorials;           /* (T) buffer of factorials. Dynamic. */
    WORD    *small_power_n;        /*     length of the number */
      UWORD  **small_power;          /*     the number*/
    UWORD   *bernoullis;           /* (T) The buffer with bernoulli numbers. Dynamic. */
    WORD    *primelist;
    LONG    *pfac;                 /* (T) array of positions of factorials. Dynamic. */
    LONG    *pBer;                 /* (T) array of positions of Bernoulli's. Dynamic. */
    WORD    *TMaddr;               /* (R) buffer for TestSub */
    WORD    *WildMask;             /* (N) Wildcard info during pattern matching */
    WORD    *previousEfactor;      /* () Cache for factors in expressions */
    WORD    **TermMemHeap;        /* For TermMalloc. Set zero in Checkpoint */
    UWORD    **NumberMemHeap;      /* For NumberMalloc. Set zero in Checkpoint */
    UWORD    **CacheNumberMemHeap; /* For CacheNumberMalloc. Set zero in Checkpoint */
    BRACKETINFO *bracketinfo;
    WORD    **ListPoly;
    WORD    *ListSymbols;
    UWORD   *NumMem;
    PARTI   partitions;
    LONG    sBer;                  /* (T) Size of the bernoullis buffer */
    LONG    pWorkPointer;          /* (R) Offset-pointer in pWorkSpace */
    LONG    lWorkPointer;          /* (R) Offset-pointer in lWorkSpace */
    LONG    posWorkPointer;        /* (R) Offset-pointer in posWorkSpace */
    LONG    InNumMem;
    int     sfact;                 /* (T) size of the factorials buffer */
    int     mfac;                  /* (T) size of the pfac array. */
    int     ebufnum;               /* (R) extra compiler buffer */
    int     fbufnum;               /* extra compiler buffer for factorization cache */
    int     allbufnum;             /* extra compiler buffer for id,all */
    int     aebufnum;              /* extra compiler buffer for id,all */
    int     idallflag;             /* indicates use of id,all buffers */
    int     idallnum;
    int     idallmaxnum;
    int     WildcardBufferSize;    /* () local copy for updates */
#ifdef WITHPTHREADS
    int     identity;              /* () When we work with B->T */
    int     LoadBalancing;         /* Needed for synchronization */
#ifdef WITHSORTBOTS
    int     SortBotIn1;            /* Input stream 1 for a SortBot */
    int     SortBotIn2;            /* Input stream 2 for a SortBot */
#endif
#endif
    int     TermMemMax;            /* For TermMalloc. Set zero in Checkpoint */
    int     TermMemTop;            /* For TermMalloc. Set zero in Checkpoint */
    int     NumberMemMax;          /* For NumberMalloc. Set zero in Checkpoint */
    int     NumberMemTop;          /* For NumberMalloc. Set zero in Checkpoint */
    int     CacheNumberMemMax;     /* For CacheNumberMalloc. Set zero in Checkpoint */
    int     CacheNumberMemTop;     /* For CacheNumberMalloc. Set zero in Checkpoint */
    int     bracketindexflag;      /* Are brackets going to be indexed? */
    int     optimtimes;            /* Number of the evaluation of the MCTS tree */
    int     ListSymbolsSize;
    int     NumListSymbols;
    int     numpoly;
    int     LeaveNegative;
    int     TrimPower;             /* Indicates trimming in polyratfun expansion */
    WORD    small_power_maxx;      /*     size of the cache for small powers  */
    WORD    small_power_maxn;      /*     size of the cache for small powers */
    WORD    dummysubexp[SUBEXPSIZE+4]; /* () used in normal.c */
    WORD    comsym[8];             /* () Used in tools.c = {8,SYMBOL,4,0,1,1,1,3} */
    WORD    comnum[4];             /* () Used in tools.c = { 4,1,1,3 } */
    WORD    comfun[FUNHEAD+4];     /* () Used in tools.c = {7,FUNCTION,3,0,1,1,3} */
                                   /*            or { 8,FUNCTION,4,0,0,1,1,3 } */
    WORD    comind[7];             /* () Used in tools.c = {7,INDEX,3,0,1,1,3} */
    WORD    MinVecArg[7+ARGHEAD];  /* (N) but should be more local */
    WORD    FunArg[4+ARGHEAD+FUNHEAD]; /* (N) but can be more local */
    WORD    locwildvalue[SUBEXPSIZE]; /* () Used in argument.c = {SUBEXPRESSION,SUBEXPSIZE,0,0,0} */
    WORD    mulpat[SUBEXPSIZE+5];  /* () Used in argument.c = {TYPEMULT, SUBEXPSIZE+3, 0, */
                                   /* SUBEXPRESSION, SUBEXPSIZE, 0, 1, 0, 0, 0 } */
    WORD    proexp[SUBEXPSIZE+5];  /* () Used in poly.c */
    WORD    TMout[40];             /* (R) Passing info */
    WORD    TMbuff;                /* (R) Communication between TestSub and Genera */
    WORD    TMdolfac;              /* factor number for dollar */
    WORD    nfac;                  /* (T) Number of highest stored factorial */
    WORD    nBer;                  /* (T) Number of highest bernoulli number. */
    WORD    mBer;                  /* (T) Size of buffer pBer. */
    WORD    PolyAct;               /* (R) Used for putting the PolyFun at end. ini at 0 */
    WORD    RecFlag;               /* (R) Used in TestSub. ini at zero. */
    WORD    inprimelist;
```

```
    WORD    sizeprimelist;
    WORD    fromindex;              /* Tells the compare routine whether call from index */
#ifdef WITHPTHREADS
#ifdef WITHSORTBOTS
    PADPOINTER(5,27,100+SUBEXPSIZE*4+FUNHEAD*2+ARGHEAD*2,0);
#else
    PADPOINTER(5,25,100+SUBEXPSIZE*4+FUNHEAD*2+ARGHEAD*2,0);
#endif
#else
    PADPOINTER(5,23,100+SUBEXPSIZE*4+FUNHEAD*2+ARGHEAD*2,0);
#endif
};
/*
        #] T :
        #[ N : The N struct contains variables used in running information
               that is inside blocks that should not be split, like pattern
               matching, traces etc. They are local for each thread.
               They don't need initializations.
*/
struct N_const {
    POSITION OldPosIn;              /* (R) Used in sort. */
    POSITION OldPosOut;             /* (R) Used in sort */
    POSITION theposition;           /* () Used in index.c */
    WORD    *EndNest;               /* (R) Nesting of function levels etc. */
    WORD    *Frozen;                /* (R) Bracket info */
    WORD    *FullProto;             /* (R) Prototype of a subexpression or table */
    WORD    *cTerm;                 /* (R) Current term for coef_ and term_ */
    int     *RepPoint;              /* (R) Pointer in RepCount buffer. Tracks repeat */
    WORD    *WildValue;             /* (N) Wildcard info during pattern matching */
    WORD    *WildStop;              /* (N) Wildcard info during pattern matching */
    WORD    *argaddress;            /* (N) Used in pattern matching of arguments */
    WORD    *RepFunList;            /* (N) For pattern matching */
    WORD    *patstop;               /* (N) Used in pattern matching */
    WORD    *terstop;               /* (N) Used in pattern matching */
    WORD    *terstart;              /* (N) Used in pattern matching */
    WORD    *terfirstcomm;          /* (N) Used in pattern matching */
    WORD    *DumFound;              /* (N) For renumbering indices {make local?} */
    WORD    **DumPlace;             /* (N) For renumbering indices {make local?} */
    WORD    **DumFunPlace;          /* (N) For renumbering indices {make local?} */
    WORD    *UsedSymbol;            /* (N) When storing terms of a global expr. */
    WORD    *UsedVector;            /* (N) When storing terms of a global expr. */
    WORD    *UsedIndex;             /* (N) When storing terms of a global expr. */
    WORD    *UsedFunction;          /* (N) When storing terms of a global expr. */
    WORD    *MaskPointer;           /* (N) For wildcard pattern matching */
    WORD    *ForFindOnly;           /* (N) For wildcard pattern matching */
    WORD    *findTerm;              /* (N) For wildcard pattern matching */
    WORD    *findPattern;           /* (N) For wildcard pattern matching */
#ifdef WITHZLIB
    Bytef   **ziobufnum;            /* () Used in compress.c */
    Bytef   *ziobuffers;            /* () Used in compress.c */
#endif
    WORD    *dummyrenumlist;        /* () Used in execute.c and store.c */
    int     *funargs;               /* () Used in lus.c */
    WORD    **funlocs;              /* () Used in lus.c */
    int     *funinds;               /* () Used in lus.c */
    UWORD   *NoScrat2;              /* () Used in normal.c */
    WORD    *ReplaceScrat;          /* () Used in normal.c */
    TRACES    *tracestack;            /* () used in opera.c */
    WORD    *selecttermundo;        /* () Used in pattern.c */
    WORD    *patternbuffer;         /* () Used in pattern.c */
    WORD    *termbuffer;            /* () Used in pattern.c */
    WORD    **PoinScratch;          /* () used in reshuf.c */
    WORD    **FunScratch;           /* () used in reshuf.c */
    WORD    *RenumScratch;          /* () used in reshuf.c */
    FUN_INFO *FunInfo;              /* () Used in smart.c */
    WORD    **SplitScratch;         /* () Used in sort.c */
    WORD    **SplitScratch1;        /* () Used in sort.c */
    SORTING **FunSorts;             /* () Used in sort.c */
    UWORD   *SoScratC;              /* () Used in sort.c */
    WORD    *listinprint;           /* () Used in proces.c and message.c */
    WORD    *currentTerm;           /* () Used in proces.c and message.c */
    WORD    **arglist;              /* () Used in function.c */
    int     *tlistbuf;              /* () used in lus.c */
#ifdef WHICHSUBEXPRESSION
    UWORD   *BinoScrat;             /* () Used in proces.c */
#endif
    WORD    *compressSpace;         /* () Used in sort.c */
#ifdef WITHPTHREADS
    THREADBUCKET *threadbuck;
    EXPRESSIONS expr;
#endif
    UWORD   *SHcombi;
    WORD    *poly_vars;
    UWORD   *cmod;                  /* Local setting of modulus. Pointer to value. */
    SHvariables SHvar;
    LONG    deferskipped;           /* () Used in proces.c store.c and parallel.c */
    LONG    InScratch;              /* () Used in sort.c */
```

```
    LONG    SplitScratchSize;       /* () Used in sort.c */
    LONG    InScratch1;             /* () Used in sort.c */
    LONG    SplitScratchSize1;      /* () Used in sort.c */
    LONG    ninterms;               /* () Used in proces.c and sort.c */
#ifdef WITHPTHREADS
    LONG    inputnumber;            /* () For use in redefine */
    LONG    lastinindex;
#endif
#ifdef WHICHSUBEXPRESSION
    LONG    last2;                  /* () Used in proces.c */
    LONG    last3;                  /* () Used in proces.c */
#endif
    LONG    SHcombisize;
    int     NumTotWildArgs;         /* (N) Used in pattern matching */
    int     UseFindOnly;            /* (N) Controls pattern routines */
    int     UsedOtherFind;          /* (N) Controls pattern routines */
    int     ErrorInDollar;          /* (R) */
    int     numfargs;               /* () Used in lus.c */
    int     numflocs;               /* () Used in lus.c */
    int     nargs;                  /* () Used in lus.c */
    int     tohunt;                 /* () Used in lus.c */
    int     numoffuns;              /* () Used in lus.c */
    int     funisize;               /* () Used in lus.c */
    int     RSsize;                 /* () Used in normal.c */
    int     numtracesctack;         /* () used in opera.c */
    int     intracestack;           /* () used in opera.c */
    int     numfuninfo;             /* () Used in smart.c */
    int     NumFunSorts;            /* () Used in sort.c */
    int     MaxFunSorts;            /* () Used in sort.c */
    int     arglistsize;            /* () Used in function.c */
    int     tlistsize;              /* () used in lus.c */
    int     filenum;                /* () used in setfile.c */
    int     compressSize;           /* () Used in sort.c */
    int     polysortflag;
    int     nogroundlevel;          /* () Used to see whether pattern matching at groundlevel */
    int     subsubveto;             /* () Sabotage combining subexpressions in TestSub */
    WORD    MaxRenumScratch;        /* () used in reshuf.c */
    WORD    oldtype;                /* (N) WildCard info at pattern matching */
    WORD    oldvalue;               /* (N) WildCard info at pattern matching */
    WORD    NumWild;                /* (N) Used in Wildcard */
    WORD    RepFunNum;              /* (N) Used in pattern matching */
    WORD    DisOrderFlag;           /* (N) Disorder option? Used in pattern matching */
    WORD    WildDirt;               /* (N) dirty in wldcard substitution. */
    WORD    NumFound;               /* (N) in reshuf only. Local? */
    WORD    WildReserve;            /* (N) Used in the wildcards */
    WORD    TeInFun;                /* (R) Passing type of action */
    WORD    TeSuOut;                /* (R) Passing info. Local? */
    WORD    WildArgs;               /* (R) */
    WORD    WildEat;                /* (R) */
    WORD    PolyNormFlag;           /* (R) For polynomial arithmetic */
    WORD    PolyFunTodo;            /* deals with expansions and multiplications */
    WORD    sizeselecttermundo;     /* () Used in pattern.c */
    WORD    patternbuffersize;      /* () Used in pattern.c */
    WORD    numlistinprint;         /* () Used in process.c */
    WORD    ncmod;                  /* () used as some type of flag to disable */
    WORD    ExpectedSign;
    WORD    SignCheck;
    WORD    IndDum;                 /* Active dummy indices */
    WORD    poly_num_vars;
    WORD    idfunctionflag;
    WORD    poly_vars_type;         /* type of allocation. For free. */
    WORD    tryterm;                /* For EndSort(...,2) */
#ifdef WHICHSUBEXPRESSION
    WORD    nbino;                  /* () Used in proces.c */
    WORD    last1;                  /* () Used in proces.c */
#endif
#ifdef WITHPTHREADS
#ifdef WHICHSUBEXPRESSION
#ifdef WITHZLIB
    PADPOSITION(55,11,23,28,sizeof(SHvariables));
#else
    PADPOSITION(53,11,23,28,sizeof(SHvariables));
#endif
#else
#ifdef WITHZLIB
    PADPOSITION(54,9,23,26,sizeof(SHvariables));
#else
    PADPOSITION(52,9,23,26,sizeof(SHvariables));
#endif
#endif
#else
#ifdef WHICHSUBEXPRESSION
#ifdef WITHZLIB
    PADPOSITION(53,9,23,28,sizeof(SHvariables));
#else
    PADPOSITION(51,9,23,28,sizeof(SHvariables));
#endif
```

```
#else
#ifdef WITHZLIB
    PADPOSITION(52,7,23,26,sizeof(SHvariables));
#else
    PADPOSITION(50,7,23,26,sizeof(SHvariables));
#endif
#endif
#endif
};

/*
        #] N :
        #[ O : The O struct concerns output variables
*/
struct O_const {
    FILEDATA    SaveData;           /* (O) */
    STOREHEADER SaveHeader;         /* ()  System Independent save-Files */
    OPTIMIZERESULT OptimizeResult;
    UBYTE   *OutputLine;            /* (O) Sits also in debug statements */
    UBYTE   *OutStop;               /* (O) Top of OutputLine buffer */
    UBYTE   *OutFill;               /* (O) Filling point in OutputLine buffer */
    WORD    *bracket;               /* (O) For writing brackets */
    WORD    *termbuf;               /* (O) For writing terms */
    WORD    *tabstring;
    UBYTE   *wpos;                  /* (O) Only when storing file {local?} */
    UBYTE   *wpoin;                 /* (O) Only when storing file {local?} */
    UBYTE   *DollarOutBuffer;       /* (O) Outputbuffer for Dollars */
    UBYTE   *CurBufWrt;             /* (O) Name of currently written expr. */
    VOID    (*FlipWORD)(UBYTE *);   /* ()  Function pointers for translations. Initialized by ReadSaveHeader() */
    VOID    (*FlipLONG)(UBYTE *);
    VOID    (*FlipPOS)(UBYTE *);
    VOID    (*FlipPOINTER)(UBYTE *);
    VOID    (*ResizeData)(UBYTE *,int,UBYTE *,int);
    VOID    (*ResizeWORD)(UBYTE *,UBYTE *);
    VOID    (*ResizeNCWORD)(UBYTE *,UBYTE *);
    VOID    (*ResizeLONG)(UBYTE *,UBYTE *);
    VOID    (*ResizePOS)(UBYTE *,UBYTE *);
    VOID    (*ResizePOINTER)(UBYTE *,UBYTE *);
    VOID    (*CheckPower)(UBYTE *);
    VOID    (*RenumberVec)(UBYTE *);
    DICTIONARY **Dictionaries;
    UBYTE   *tensorList;            /* Dynamically allocated list with functions that are tensorial. */
    WORD    *inscheme;              /* for feeding a Horner scheme to Optimize */
/*----Leave NumInBrack as first non-pointer. This is used by the checkpoints--*/
    LONG    NumInBrack;             /* (O) For typing [] option in print */
    LONG    wlen;                   /* (O) Used to store files. */
    LONG    DollarOutSizeBuffer;    /* (O) Size of DollarOutBuffer */
    LONG    DollarInOutBuffer;      /* (O) Characters in DollarOutBuffer */
#if defined(mBSD) && defined(MICROTIME)
    LONG    wrap;                   /* (O) For statistics time. wrap around */
    LONG    wrapnum;                /* (O) For statistics time. wrap around */
#endif
    OPTIMIZE Optimize;
    int     OutInBuffer;            /* (O) Which routine does the writing */
    int     NoSpacesInNumbers;      /*     For very long numbers */
    int     BlockSpaces;            /*     For very long numbers */
    int     CurrentDictionary;
    int     SizeDictionaries;
    int     NumDictionaries;
    int     CurDictNumbers;
    int     CurDictVariables;
    int     CurDictSpecials;
    int     CurDictFunWithArgs;
    int     CurDictNumberWarning;
    int     CurDictNotInFunctions;
    int     CurDictInDollars;
    int     gNumDictionaries;
    WORD    schemenum;              /* for feeding a Horner scheme to Optimize */
    WORD    transFlag;              /* ()  >0 indicades that translations have to be done */
    WORD    powerFlag;              /* ()  >0 indicades that some exponents/powers had to be adjusted */
    WORD    mpower;                 /*     For maxpower adjustment to larger value */
    WORD    resizeFlag;             /* ()  >0 indicades that something went wrong when resizing words */
    WORD    bufferedInd;            /* ()  Contains extra INDEXENTRIES, see ReadSaveIndex() for an explanation */
    WORD    OutSkip;                /* (O) How many chars to skip in output line */
    WORD    IsBracket;              /* (O) Controls brackets */
    WORD    InFbrack;               /* (O) For writing only */
    WORD    PrintType;              /* (O) */
    WORD    FortFirst;              /* (O) Only in sch.c */
    WORD    DoubleFlag;             /* (O) Output in double precision */
    WORD    IndentSpace;            /* For indentation in output */
    WORD    FactorMode;             /* When the output should be written as factors */
    WORD    FactorNum;              /* Number of factor currently treated */
    WORD    ErrorBlock;
```

```
    WORD    OptimizationLevel;     /* Level of optimization in the output */
    UBYTE   FortDotChar;           /* (O) */
/*
    For the padding, please count also the number of int's in the OPTIMIZE struct.
*/
#if defined(mBSD) && defined(MICROTIME)
    PADPOSITION(25,6,35,17,1);
#else
    PADPOSITION(25,4,35,17,1);
#endif
};
/*
        #] O :
        #[ X : The X struct contains variables that deal with the external channel
*/
struct X_const {
    UBYTE *currentPrompt;
    UBYTE *shellname;          /* if !=NULL (default is "/bin/sh -c"), start in
                                  the specified subshell*/
    UBYTE *stderrname;         /* If !=NULL (default if "/dev/null"), stderr is
                                  redirected to the specified file*/
    int timeout;               /* timeout to initialize preset channels.
                                  If timeout<0, the preset channels are
                                  already initialized*/
    int killSignal;            /* signal number, SIGKILL by default*/
    int killWholeGroup;        /* if 0, the signal is sent only to a process,
                                  if !=0 (default) is sent to a whole process group*/
    int daemonize;             /* if !=0 (default), start in a daemon mode */
    int currentExternalChannel;
    PADPOINTER(0,5,0,0);
};
/*
        #] X :
        #[ Definitions :
*/

#ifdef WITHPTHREADS

typedef struct AllGlobals {
    struct M_const M;
    struct C_const C;
    struct S_const S;
    struct O_const O;
    struct P_const P;
    struct X_const X;
    PADPOSITION(0,0,0,0,sizeof(struct P_const)+sizeof(struct X_const));
} ALLGLOBALS;

typedef struct AllPrivates {
    struct R_const R;
    struct N_const N;
    struct T_const T;
    PADPOSITION(0,0,0,0,sizeof(struct T_const));
} ALLPRIVATES;

#else

typedef struct AllGlobals {
    struct M_const M;
    struct C_const C;
    struct S_const S;
    struct R_const R;
    struct N_const N;
    struct O_const O;
    struct P_const P;
    struct T_const T;
    struct X_const X;
    PADPOSITION(0,0,0,0,sizeof(struct P_const)+sizeof(struct T_const)+sizeof(struct X_const));
} ALLGLOBALS;

#endif

/*
        #] Definitions :
    #] A :
    #[ FG :
*/

#ifdef ANSI
typedef WORD (*WCN)(PHEAD WORD *,WORD *,WORD,WORD);
typedef WORD (*WCN2)(PHEAD WORD *,WORD *);
#else
typedef WORD (*WCN)();
typedef WORD (*WCN2)();
#endif

typedef struct FixedGlobals {
```

```
    WCN     Operation[8];
    WCN2    OperaFind[6];
    char    *VarType[10];
    char    *ExprStat[21];
    char    *FunNam[2];
    char    *swmes[3];
    char    *fname;
    char    *fname2;
    UBYTE   *s_one;
    WORD    fnamebase;
    WORD    fname2base;
    UINT    cTable[256];
} FIXEDGLOBALS;

/*
    #] FG :
*/

#endif
```

4.110 symmetr.c File Reference

`#include "form3.h"`

Include dependency graph for symmetr.c:

Functions

- WORD MatchE (PHEAD WORD ∗pattern, WORD ∗fun, WORD ∗inter, WORD par)
- WORD Permute (PERM ∗perm, WORD first)
- WORD PermuteP (PERMP ∗perm, WORD first)
- WORD Distribute (DISTRIBUTE ∗d, WORD first)
- int MatchCy (PHEAD WORD ∗pattern, WORD ∗fun, WORD ∗inter, WORD par)
- int FunMatchCy (PHEAD WORD ∗pattern, WORD ∗fun, WORD ∗inter, WORD par)
- int FunMatchSy (PHEAD WORD ∗pattern, WORD ∗fun, WORD ∗inter, WORD par)
- int MatchArgument (PHEAD WORD ∗arg, WORD ∗pat)

4.110.1 Detailed Description

The routines that deal with the pattern matching of functions with symmetric properties.

Definition in file symmetr.c.

4.110.2 Function Documentation

4.110.2.1 MatchE()

```
WORD MatchE (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )
```

Definition at line 56 of file symmetr.c.

4.110.2.2 Permute()

```
WORD Permute (
            PERM * perm,
            WORD first )
```

Definition at line 443 of file symmetr.c.

4.110.2.3 PermuteP()

```
WORD PermuteP (
            PERMP * perm,
            WORD first )
```

Definition at line 477 of file symmetr.c.

4.110.2.4 Distribute()

```
WORD Distribute (
            DISTRIBUTE * d,
            WORD first )
```

Definition at line 509 of file symmetr.c.

4.110.2.5 MatchCy()

```
int MatchCy (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )
```

Definition at line 570 of file symmetr.c.

4.110.2.6 FunMatchCy()

```
int FunMatchCy (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )
```

Definition at line 1066 of file symmetr.c.

4.110.2.7 FunMatchSy()

```
int FunMatchSy (
            PHEAD WORD * pattern,
            WORD * fun,
            WORD * inter,
            WORD par )
```

Definition at line 1524 of file symmetr.c.

4.110.2.8 MatchArgument()

```
int MatchArgument (
            PHEAD WORD * arg,
            WORD * pat )
```

Definition at line 2051 of file symmetr.c.

4.111 symmetr.c

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033     #[ Includes : function.c
00034 */
00035
```

```
#include "form3.h"

/*
    #] Includes :
    #[ MatchE :                 WORD MatchE(pattern,fun,inter,par)

        Matches symmetric and antisymmetric tensors.
        Pattern and fun point at a tensor.
        Problem is the wildcarding and all its possible permutations.
        This routine loops over all of them and calls for each
        possible wildcarding the recursion in ScanFunctions.
        Note that this can be very costly.

        Originally this routine did only Levi Civita tensors and hence
        it dealt only with commuting objects.
        Because of the backtracking we cannot fall back to the calling
        ScanFunctions routine and check the sequence of functions when
        non-commuting objects are involved.
*/

WORD MatchE(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
{
    GETBIDENTITY
    WORD *m, *t, *r, i, retval;
    WORD *mstop, *tstop, j, newvalue, newfun;
    WORD fixvec[MAXMATCH],wcvec[MAXMATCH],fixind[MAXMATCH],wcind[MAXMATCH];
    WORD tfixvec[MAXMATCH],tfixind[MAXMATCH];
    WORD vwc,vfix,ifix,iwc,tvfix,tifix,nv,ni;
    WORD sign = 0, *rstop, first1, first2, first3, funwild;
    WORD *OldWork, nwstore, oRepFunNum;
    PERM perm1,perm2;
    DISTRIBUTE distr;
    WORD *newpat, /* *newter, *instart, */ offset;
/*  instart = fun; */
    offset = WORDDIF(fun,AN.terstart);
    if ( pattern[1] != fun[1] ) return(0);
    if ( *pattern >= FUNCTION+WILDOFFSET ) {
        if ( CheckWild(BHEAD *pattern-WILDOFFSET,FUNTOFUN,*fun,&newfun) ) return(0);
        funwild = 1;
    }
    else funwild = 0;
    mstop = pattern + pattern[1];
    tstop = fun + fun[1];
    m = pattern + FUNHEAD;
    t = fun + FUNHEAD;
    while ( m < mstop ) {
        if ( *m != *t ) break;
        m++; t++;
    }
    if ( m >= mstop ) {
        AN.RepFunList[AN.RepFunNum++] = offset;
        AN.RepFunList[AN.RepFunNum++] = 0;
        newpat = pattern + pattern[1];
        if ( funwild ) {
            m = AN.WildValue;
            t = OldWork = AT.WorkPointer;
            nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
            r = AT.WildMask;
            if ( i > 0 ) {
                do {
                    *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
                } while ( --i > 0 );
            }
            if ( t >= AT.WorkTop ) {
                MLOCK(ErrorMessageLock);
                MesWork();
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
            AT.WorkPointer = t;
            AddWild(BHEAD *pattern-WILDOFFSET,FUNTOFUN,newfun);
            if ( newpat >= AN.patstop ) {
                if ( AN.UseFindOnly == 0 ) {
                    if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                        AN.UsedOtherFind = 1;
                        return(1);
                    }
                    retval = 0;
                }
                else return(1);
            }
            else {
/*              newter = instart; */
                retval = ScanFunctions(BHEAD newpat,inter,par);
            }
            if ( retval == 0 ) {
                m = AN.WildValue;
```

```
                t = OldWork; r = AT.WildMask; i = nwstore;
                if ( i > 0 ) {
                    do {
                        *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                    } while ( --i > 0 );
                }
            }
            AT.WorkPointer = OldWork;
            return(retval);
        }
        else {
            if ( newpat >= AN.patstop ) {
                if ( AN.UseFindOnly == 0 ) {
                    if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                        AN.UsedOtherFind = 1;
                        return(1);
                    }
                    else return(0);
                }
                else return(1);
            }
/*          newter = instart; */
            i = ScanFunctions(BHEAD newpat,inter,par);
            return(i);
        }
/*
        Now the recursion
*/
    }
/*
    Strategy:
    1: match the fixed arguments
    2: match, permuting the wildcards if needed.
    3: keep track of sign.
*/
    vwc = 0;
    vfix = 0;
    ifix = 0;
    iwc = 0;
    r = pattern+FUNHEAD;
    while ( r < mstop ) {
        if ( *r < (AM.OffsetVector+WILDOFFSET) ) {
            fixvec[vfix++] = *r;            /* Fixed vectors */
            sign += vwc + ifix + iwc;
        }
        else if ( *r < MINSPEC ) {
            wcvec[vwc++] = *r;              /* Wildcard vectors */
            sign += ifix + iwc;
        }
        else if ( *r < (AM.OffsetIndex+WILDOFFSET) ) {
            fixind[ifix++] = *r;            /* Fixed indices */
            sign += iwc;
        }
        else if ( *r < (AM.OffsetIndex+(WILDOFFSET«1)) ) {
            wcind[iwc++] = *r;              /* Wildcard indices */
        }
        else {
            fixind[ifix++] = *r;            /* Generated indices ~ fixed */
            sign += iwc;
        }
        r++;
    }
    if ( iwc == 0 && vwc == 0 ) return(0);
    tvfix = tifix = 0;
    t = fun + FUNHEAD;
    m = fixvec;
    mstop = m + vfix;
    r = fixind;
    rstop = r + ifix;
    nv = 0; ni = 0;
    while ( t < tstop ) {
        if ( *t < 0 ) {
            nv++;
            if ( m < mstop && *t == *m ) {
                m++;
            }
            else {
                sign += WORDDIF(mstop,m);
                tfixvec[tvfix++] = *t;
            }
        }
        else {
            ni++;
            if ( r < rstop && *r == *t ) {
                r++;
            }
            else {
```

```
                sign += WORDDIF(rstop,r);
                tfixind[tifix++] = *t;
            }
        }
        t++;
    }
    if ( m < mstop || r < rstop ) return(0);
    if ( tvfix < vwc || (tvfix+tifix) < (vwc+iwc) ) return(0);
    sign += ( nv - vfix - vwc ) & ni;
/*
    Take now the wildcards that have an assignment already.
    See whether they match.
*/
    {
        WORD *wv, *wm, n;
        wm = AT.WildMask;
        wv = AN.WildValue;
        n = AN.NumWild;
        do {
            if ( *wm ) {
                if ( *wv == VECTOVEC ) {
                    for ( ni = 0; ni < vwc; ni++ ) {
                        if ( wcvec[ni]-WILDOFFSET == wv[2] ) {  /* Has been assigned */
                            sign += ni;
                            vwc--;
                            while ( ni < vwc ) {
                                wcvec[ni] = wcvec[ni+1];
                                ni++;
                            }
/* TryVect: */
                            for ( ni = 0; ni < tvfix; ni++ ) {
                                if ( tfixvec[ni] == wv[3] ) {
                                    sign += ni;
                                    tvfix--;
                                    while ( ni < tvfix ) {
                                        tfixvec[ni] = tfixvec[ni+1];
                                        ni++;
                                    }
                                    goto NextWV;
                                }
                            }
                            return(0);
                        }
                    }
                }
                else if ( *wv == INDTOIND ) {
                    for ( ni = 0; ni < iwc; ni++ ) {
                        if ( wcind[ni]-WILDOFFSET == wv[2] ) {  /* Has been assigned */
                            sign += ni;
                            iwc--;
                            while ( ni < iwc ) {
                                wcind[ni] = wcind[ni+1];
                                ni++;
                            }
                            for ( ni = 0; ni < tifix; ni++ ) {
                                if ( tfixind[ni] == wv[3] ) {
                                    sign += ni;
                                    tifix--;
                                    while ( ni < tifix ) {
                                        tfixind[ni] = tfixind[ni+1];
                                        ni++;
                                    }
                                    goto NextWV;
                                }
                            }
/*                          goto TryVect; */
                            return(0);

                        }
                    }
                }
                else if ( *wv == VECTOSUB ) {
                    for ( ni = 0; ni < vwc; ni++ ) {
                        if ( wcvec[ni]-WILDOFFSET == wv[2] ) return(0);
                    }
                }
                else if ( *wv == INDTOSUB ) {
                    for ( ni = 0; ni < iwc; ni++ ) {
                        if ( wcind[ni]-WILDOFFSET == wv[2] ) return(0);
                    }
                }
            }
NextWV:
            wm++;
            wv += wv[1];
            n--;
            if ( n > 0 ) {
```

```
                while ( n > 0 && ( *wv == FROMSET || *wv == SETTONUM
                || *wv == LOADDOLLAR ) ) { wv += wv[1]; wm++; n--; }
/*
                Freak problem: doesn't test for n and ran into a reamining
                code equal to SETTONUM followed by a big number and then
                ran out of the memory.

                while ( *wv == FROMSET || *wv == SETTONUM
                || ( *wv == LOADDOLLAR && n > 0 ) ) { wv += wv[1]; wm++; n--; }
*/
            }
        } while ( n > 0 );
    }
/*
    Now there are only free wildcards left.
    Possibly the assigned values ate too many vectors.
    The rest has to be done the 'hard way' via permutations.
    This is too bad when there are 10 indices.
    This could cause 10! tries.
    We try to avoid the worst case by using a very special
    (somewhat slow) permutation routine that has as its worst
    cases some rather unlikely configurations, rather than some
    common ones (as would have been the case with the conventional
    permuation routine).
        assume:
        vvvvvvvvvvvvvvvvvvvv     (tvfix in tfixvec and tifix in tfixind)
        VVVVVVVVVVVVVVVVVVVV     (vwc in wcvec and iwc in wcind)
    Note: all further assignments are possible at this point!
    Strategy:
        permute v
        permute i
        loop over the ordered distribution of the leftover v's
        through the i's.
*/
    if ( tvfix < vwc ) { return(0); }
    perm1.n = tvfix;
    perm1.sign = 0;
    perm1.objects = tfixvec;
    perm2.n = tifix;
    perm2.sign = 0;
    perm2.objects = tfixind;
    distr.n1 = tvfix - vwc;
    distr.n2 = tifix;
    distr.obj1 = tfixvec + vwc;
    distr.obj2 = tfixind;
    distr.out = fixvec;             /* For scratch */
    first1 = 1;
/*
            Store the current Wildcard assignments
*/
    m = AN.WildValue;
    t = OldWork = AT.WorkPointer;
    nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
    r = AT.WildMask;
    if ( i > 0 ) {
        do {
            *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
        } while ( --i > 0 );
    }
    if ( t >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    AT.WorkPointer = t;
    while ( (first1 = Permute(&perm1,first1) ) == 0 ) {
        first2 = 1;
        while ( (first2 = Permute(&perm2,first2) ) == 0 ) {
            first3 = 1;
            while ( (first3 = Distribute(&distr,first3) ) == 0 ) {
/*
                Make now the wildcard assignments
*/
                for ( i = 0; i < vwc; i++ ) {
                    j = wcvec[i] - WILDOFFSET;
                    if ( CheckWild(BHEAD j,VECTOVEC,tfixvec[i],&newvalue) )
                        goto NoCaseB;
                    AddWild(BHEAD j,VECTOVEC,newvalue);
                }
                for ( i = 0; i < iwc; i++ ) {
                    j = wcind[i] - WILDOFFSET;
                    if ( CheckWild(BHEAD j,INDTOIND,fixvec[i],&newvalue) )
                        goto NoCaseB;
                    AddWild(BHEAD j,INDTOIND,newvalue);
                }
/*
```

```
                Go into the recursion
*/
                oRepFunNum = AN.RepFunNum;
                AN.RepFunList[AN.RepFunNum++] = offset;
                AN.RepFunList[AN.RepFunNum++] =
                    ( perm1.sign + perm2.sign + distr.sign + sign ) & 1;
                newpat = pattern + pattern[1];
                if ( funwild ) AddWild(BHEAD *pattern-WILDOFFSET,FUNTOFUN,newfun);
                if ( newpat >= AN.patstop ) {
                    if ( AN.UseFindOnly == 0 ) {
                        if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                            AN.UsedOtherFind = 1;
                            return(1);
                        }
                    }
                    else return(1);
                }
                else {
/*                  newter = instart; */
                    if ( ScanFunctions(BHEAD newpat,inter,par) ) { return(1); }
                }
/*
                Restore the old Wildcard assignments
*/
                AN.RepFunNum = oRepFunNum;
NoCaseB:        m = AN.WildValue;
                t = OldWork; r = AT.WildMask; i = nwstore;
                if ( i > 0 ) {
                    do {
                        *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                    } while ( --i > 0 );
                }
                AT.WorkPointer = t;
            }
        }
    }
    AT.WorkPointer = OldWork;
    return(0);
}

/*
    #] MatchE :
    #[ Permute :                WORD Permute(perm,first)

        Special permutation function.
        Works recursively.
        The aim is to cycle through in as fast a way as possible,
        to take care that each object hits the various positions
        already early in the game.

        Start at two: -> cycle of two
        then three -> cycle of three
        etc;
        The innermost cycle is the longest. This is the opposite
        of the usual way of generating permutations and it is
        certainly not the fastest one. It allows for the fastest
        hit in the assignment of wildcards though.
*/

WORD Permute(PERM *perm, WORD first)
{
    WORD *s, c, i, j;
    if ( first ) {
        perm->sign = ( perm->sign <= 1 ) ? 0: 1;
        for ( i = 0; i < perm->n; i++ ) perm->cycle[i] = 0;
        return(0);
    }
    i = perm->n;
    while ( --i > 0 ) {
        s = perm->objects;
        c = s[0];
        j = i;
        while ( --j >= 0 ) { *s = s[1]; s++; }
        *s = c;
        if ( ( i & 1 ) != 0 ) perm->sign ^= 1;
        if ( perm->cycle[i] < i ) {
            (perm->cycle[i])++;
            return(0);
        }
        else {
            perm->cycle[i] = 0;
        }
    }
    return(1);
}

/*
```

```
    #] Permute :
    #[ PermuteP :                WORD PermuteP(perm,first)

    Like Permute, but works on an array of pointers
*/

WORD PermuteP(PERMP *perm, WORD first)
{
    WORD **s, *c, i, j;
    if ( first ) {
        perm->sign = ( perm->sign <= 1 ) ? 0: 1;
        for ( i = 0; i < perm->n; i++ ) perm->cycle[i] = 0;
        return(0);
    }
    i = perm->n;
    while ( --i > 0 ) {
        s = perm->objects;
        c = s[0];
        j = i;
        while ( --j >= 0 ) { *s = s[1]; s++; }
        *s = c;
        if ( ( i & 1 ) != 0 ) perm->sign ^= 1;
        if ( perm->cycle[i] < i ) {
            (perm->cycle[i])++;
            return(0);
        }
        else {
            perm->cycle[i] = 0;
        }
    }
    return(1);
}

/*
    #] PermuteP :
    #[ Distribute :
*/

WORD Distribute(DISTRIBUTE *d, WORD first)
{
    WORD *to, *from, *inc, *from2, i, j;
    if ( first ) {
        d->n = d->n1 + d->n2;
        to = d->out;
        from = d->obj2;
        for ( i = 0; i < d->n2; i++ ) {
            d->cycle[i] = 0;
            *to++ = *from++;
        }
        from = d->obj1;
        while ( i < d->n ) {
            d->cycle[i++] = 1;
            *to++ = *from++;
        }
        d->sign = 0;
        return(0);
    }
    if ( d->n1 == 0 || d->n2 == 0 ) return(1);
    j = 0;
    i = 0;
    inc = d->cycle;
    from = inc + d->n;
    while ( *inc ) { j++; inc++; }
    while ( !*inc && inc < from ) { i++; inc++; }
    if ( inc >= from ) return(1);
    d->sign ^= ((i&j)-j+1) & 1;
    *inc = 0;
    *--inc = 1;
    while ( --j >= 0 ) *--inc = 1;
    while ( --i > 0 ) *--inc = 0;
    to = d->out;
    from = d->obj1;
    from2 = d->obj2;
    for ( i = 0; i < d->n; i++ ) {
        if ( *inc++ ) {
            *to++ = *from++;
        }
        else {
            *to++ = *from2++;
        }
    }
    return(0);
}

/*
    #] Distribute :
    #[ MatchCy :
```

```
        Matching of (r)cyclic tensors.
        Parameters like in MatchE.
        The structure of the routine is much simpler, because the number
        of possibilities is much more limited.
        The major complication is the ?a-type wildcards
        We need a strategy for T(i1?,?a,i1?,?b). Which is the shorter
        match: ?a or ?b ? (if possible of course)
        This is also relevant in the case of the shortest match if there
        is more than one choice for i1.
*/

int MatchCy(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
{
    GETBIDENTITY
    WORD *t, *tstop, *p, *pstop, *m, *r, *oldworkpointer = AT.WorkPointer;
    WORD *thewildcards, *multiplicity, *renum, wc, newvalue, oldwilval = 0;
    WORD *params, *lowlevel = 0;
    int argcount = 0, funnycount = 0, tcount = fun[1] - FUNHEAD;
    int type = 0, pnum, i, j, k, nwstore, iraise, itop, sumeat;
    CBUF *C = cbuf+AT.ebufnum;
    int ntwa = 3*AN.NumTotWildArgs+1;
    LONG oldcpointer = C->Pointer - C->Buffer;
    WORD offset = fun-AN.terstart, *newpat;

    if ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
    pnum = pattern[0];
    nwstore = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
    if ( pnum > FUNCTION + WILDOFFSET ) {
        pnum -= WILDOFFSET;
        if ( CheckWild(BHEAD pnum,FUNTOFUN,fun[0],&newvalue) ) return(0);
        oldwilval = 1;
        t = lowlevel = AT.WorkPointer;
        m = AN.WildValue;
        i = nwstore;
        r = AT.WildMask;
        if ( i > 0 ) {
            do {
                *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
            } while ( --i > 0 );
        }
        *t++ = C->numrhs;
        if ( t >= AT.WorkTop ) {
            MLOCK(ErrorMessageLock);
            MesWork();
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        AT.WorkPointer = t;
        AddWild(BHEAD pnum,FUNTOFUN,newvalue);
    }
    if ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;

    /* First we have to make an inventory. Are there FUNNYWILD pointers? */

    p = pattern + FUNHEAD;
    pstop = pattern + pattern[1];
    while ( p < pstop ) {
        if ( *p == FUNNYWILD ) { p += 2; funnycount++; }
        else { p++; argcount++; }
    }
    if ( argcount > tcount ) goto NoSuccess;
    if ( argcount < tcount && funnycount == 0 ) goto NoSuccess;
    if ( argcount == 0 && tcount == 0 && funnycount == 0 ) {
        AN.RepFunList[AN.RepFunNum++] = offset;
        AN.RepFunList[AN.RepFunNum++] = 0;
        newpat = pattern + pattern[1];
        if ( newpat >= AN.patstop ) {
            if ( AN.UseFindOnly == 0 ) {
                if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                    AT.WorkPointer = oldworkpointer;
                    AN.UsedOtherFind = 1;
                    return(1);
                }
                j = 0;
            }
            else {
                AT.WorkPointer = oldworkpointer;
                return(1);
            }
        }
        else j = ScanFunctions(BHEAD newpat,inter,par);
        if ( j ) return(j);
        goto NoSuccess;
    }
    tstop = fun + fun[1];

```

```
    /* Store the wildcard assignments */

    params = AT.WorkPointer;
    thewildcards = t = params + tcount;
    t += ntwa;
    if ( oldwilval ) lowlevel = oldworkpointer;
    else lowlevel = t;
    m = AN.WildValue;
    i = nwstore;
    if ( i > 0 ) {
        r = AT.WildMask;
        do {
            *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
        } while ( --i > 0 );
        *t++ = C->numrhs;
    }
    if ( t >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    AT.WorkPointer = t;
/*
    #[ Case 1: no funnies or all funnies must be empty. We just cycle through.
*/
    if ( argcount == tcount ) {
        if ( funnycount > 0 ) { /* Test all funnies first */
            p = pattern + FUNHEAD;
            t = fun + FUNHEAD;
            while ( p < pstop ) {
                if ( *p != FUNNYWILD ) { p++; continue; }
                AN.argaddress = t;
                if ( CheckWild(BHEAD p[1],ARGTOARG,0,t) ) goto nomatch;
                AddWild(BHEAD p[1],ARGTOARG,0);
                p += 2;
            }
            oldwilval = 1;
        }
        for ( k = 0; k <= type; k++ ) {
            if ( k == 0 ) {
                p = params; t = fun + FUNHEAD;
                while ( t < tstop ) *p++ = *t++;
            }
            else {
                p = params+tcount; t = fun + FUNHEAD;
                while ( t < tstop ) *--p = *t++;
            }
            for ( i = 0; i < tcount; i++ ) { /* The various cycles */
                p = pattern + FUNHEAD;
                wc = 0;
                for ( j = 0; j < tcount; j++, p++ ) { /* The arguments */
                    while ( *p == FUNNYWILD ) p += 2;
                    t = params + (i+j)%tcount;
                    if ( *t == *p ) continue;
                    if ( *p >= AM.OffsetIndex + WILDOFFSET
                    && *p < AM.OffsetIndex + 2*WILDOFFSET ) {

                        /* Test wildcard index */

                        wc = *p - WILDOFFSET;
                        if ( CheckWild(BHEAD wc,INDTOIND,*t,&newvalue) ) break;
                        AddWild(BHEAD wc,INDTOIND,newvalue);
                    }
                    else if ( *t < MINSPEC && p[j] < MINSPEC
                        && *p >= AM.OffsetVector + WILDOFFSET ) {

                        /* Test wildcard vector */

                        wc = *p - WILDOFFSET;
                        if ( CheckWild(BHEAD wc,VECTOVEC,*t,&newvalue) ) break;
                        AddWild(BHEAD wc,VECTOVEC,newvalue);
                    }
                    else break;
                }
                if ( j >= tcount ) { /* Match! */

                    /* Continue with other functions. Make sure of the funnies */

                    AN.RepFunList[AN.RepFunNum++] = offset;
                    AN.RepFunList[AN.RepFunNum++] = 0;

                    if ( funnycount > 0 ) {
                        p = pattern + FUNHEAD;
                        t = fun + FUNHEAD;
                        while ( p < pstop ) {
                            if ( *p != FUNNYWILD ) { p++; continue; }
```

```
                            AN.argaddress = t;
                            AddWild(BHEAD p[1],ARGTOARG,0);
                            p += 2;
                        }
                    }
                    newpat = pattern + pattern[1];
                    if ( newpat >= AN.patstop ) {
                        if ( AN.UseFindOnly == 0 ) {
                            if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                                AT.WorkPointer = oldworkpointer;
                                AN.UsedOtherFind = 1;
                                return(1);
                            }
                            j = 0;
                        }
                        else {
                            AT.WorkPointer = oldworkpointer;
                            return(1);
                        }
                    }
                    else j = ScanFunctions(BHEAD newpat,inter,par);
                    if ( j ) {
                        AT.WorkPointer = oldworkpointer;
                        return(j); /* Full match. Return our success */
                    }
                    AN.RepFunNum -= 2;
                }

                /* No (deeper) match. -> reset wildcards and continue */

                if ( wc && nwstore > 0 ) {
                    j = nwstore;
                    m = AN.WildValue;
                    t = thewildcards + ntwa; r = AT.WildMask;
                    if ( j > 0 ) {
                        do {
                            *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                        } while ( --j > 0 );
                    }
                    C->numrhs = *t++;
                    C->Pointer = C->Buffer + oldcpointer;
                }
            }
        }
        goto NoSuccess;
    }
/*
    #] Case 1:
    #[ Case 2: One FUNNYWILD. Fix its length.
*/
    if ( funnycount == 1 ) {
        funnycount = tcount - argcount; /* Number or arguments to be eaten */
        for ( k = 0; k <= type; k++ ) {
            if ( k == 0 ) {
                p = params; t = fun + FUNHEAD;
                while ( t < tstop ) *p++ = *t++;
            }
            else {
                p = params+tcount; t = fun + FUNHEAD;
                while ( t < tstop ) *--p = *t++;
            }
            for ( i = 0; i < tcount; i++ ) { /* The various cycles */
                p = pattern + FUNHEAD;
                t = params;
                wc = 0;
                for ( j = 0; j < tcount; j++, p++, t++ ) { /* The arguments */
                    if ( *t == *p ) continue;
                    if ( *p == FUNNYWILD ) {
                        p++; wc = 1;
                        AN.argaddress = t;
                        if ( CheckWild(BHEAD *p,ARGTOARG,funnycount|EATTENSOR,t) ) break;
                        AddWild(BHEAD *p,ARGTOARG,funnycount|EATTENSOR);
                        j += funnycount-1; t += funnycount-1;
                    }
                    else if ( *p >= AM.OffsetIndex + WILDOFFSET
                    && *p < AM.OffsetIndex + 2*WILDOFFSET ) {

                        /* Test wildcard index */

                        wc = *p - WILDOFFSET;
                        if ( CheckWild(BHEAD wc,INDTOIND,*t,&newvalue) ) break;
                        AddWild(BHEAD wc,INDTOIND,newvalue);
                    }
                    else if ( *t < MINSPEC && *p < MINSPEC
                        && *p >= AM.OffsetVector + WILDOFFSET ) {

                        /* Test wildcard vector */
```

```
                        wc = *p - WILDOFFSET;
                        if ( CheckWild(BHEAD wc,VECTOVEC,*t,&newvalue) ) break;
                        AddWild(BHEAD wc,VECTOVEC,newvalue);
                    }
                    else break;
                }
                if ( j >= tcount ) { /* Match! */

                    /* Continue with other functions. Make sure of the funnies */

                    AN.RepFunList[AN.RepFunNum++] = offset;
                    AN.RepFunList[AN.RepFunNum++] = 0;
                    newpat = pattern + pattern[1];
                    if ( newpat >= AN.patstop ) {
                        if ( AN.UseFindOnly == 0 ) {
                            if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                                AT.WorkPointer = oldworkpointer;
                                AN.UsedOtherFind = 1;
                                return(1);
                            }
                            j = 0;
                        }
                        else {
                            AT.WorkPointer = oldworkpointer;
                            return(1);
                        }
                    }
                    else j = ScanFunctions(BHEAD newpat,inter,par);
                    if ( j ) {
                        AT.WorkPointer = oldworkpointer;
                        return(j); /* Full match. Return our success */
                    }
                    AN.RepFunNum -= 2;
                }

                /* No (deeper) match. -> reset wildcards and continue */

                if ( wc ) {
                    j = nwstore;
                    m = AN.WildValue;
                    t = thewildcards + ntwa; r = AT.WildMask;
                    if ( j > 0 ) {
                        do {
                            *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                        } while ( --j > 0 );
                    }
                    C->numrhs = *t++;
                    C->Pointer = C->Buffer + oldcpointer;
                }
                t = params;
                wc = *t;
                for ( j = 1; j < tcount; j++ ) { *t = t[1]; t++; }
                *t = wc;
            }
        }
        goto NoSuccess;
    }
/*
    #] Case 2:
    #[ Case 3: More than one FUNNYWILD. Complicated.
*/

    sumeat = tcount - argcount; /* Total number to be eaten by Funnies */
/*
    In the first funnycount elements of 'thewildcards' we arrange
    for the summing over the various possibilities.
    The renumbering table is in thewildcards[2*funnycount]
    The multiplicity table is in thewildcards[funnycount]
    The number of arguments for each is in thewildcards[]
*/
    p = pattern+FUNHEAD;
    for ( i = funnycount; i < ntwa; i++ ) thewildcards[i] = -1;
    multiplicity = thewildcards + funnycount;
    renum = multiplicity + funnycount;
    j = 0;
    while ( p < pstop ) {
        if ( *p != FUNNYWILD ) { p++; continue; }
        p++;
        if ( renum[*p] < 0 ) {
            renum[*p] = j;
            multiplicity[j] = 1;
            j++;
        }
        else multiplicity[renum[*p]]++;
        p++;
    }
```

```
/*
    Strategy: First 'declared' has a tendency to be smaller
*/
    for ( i = 1; i < AN.NumTotWildArgs; i++ ) {
        if ( renum[i] < 0 ) continue;
        for ( j = i+1; j <= AN.NumTotWildArgs; j++ ) {
            if ( renum[j] < 0 ) continue;
            if ( renum[i] < renum[j] ) continue;
            k = multiplicity[renum[i]];
            multiplicity[renum[i]] = multiplicity[renum[j]];
            multiplicity[renum[j]] = k;
            k = renum[i]; renum[i] = renum[j]; renum[j] = k;
        }
    }
    for ( i = 0; i < funnycount; i++ ) thewildcards[i] = 0;
    iraise = funnycount-1;
    for ( ;; ) {
        for ( i = 0, j = sumeat; i < iraise; i++ )
                j -= thewildcards[i]*multiplicity[i];
        if ( j < 0 || j % multiplicity[iraise] != 0 ) {
            if ( j > 0 ) {
                thewildcards[iraise-1]++;
                continue;
            }
            itop = iraise-1;
            while ( itop > 0 && j < 0 ) {
                j += thewildcards[itop]*multiplicity[itop];
                thewildcards[itop] = 0;
                itop--;
            }
            if ( itop <= 0 && j <= 0 ) break;
            thewildcards[itop]++;
            continue;
        }
        thewildcards[iraise] = j / multiplicity[iraise];

        for ( k = 0; k <= type; k++ ) {
            if ( k == 0 ) {
                p = params; t = fun + FUNHEAD;
                while ( t < tstop ) *p++ = *t++;
            }
            else {
                p = params+tcount; t = fun + FUNHEAD;
                while ( t < tstop ) *--p = *t++;
            }
            for ( i = 0; i < tcount; i++ ) { /* The various cycles */
                p = pattern + FUNHEAD;
                t = params;
                wc = 0;
                for ( j = 0; j < tcount; j++, p++, t++ ) { /* The arguments */
                    if ( *t == *p ) continue;
                    if ( *p == FUNNYWILD ) {
                        p++; wc = thewildcards[renum[*p]];
                        AN.argaddress = t;
                        if ( CheckWild(BHEAD *p,ARGTOARG,wc|EATTENSOR,t) ) break;
                        AddWild(BHEAD *p,ARGTOARG,wc|EATTENSOR);
                        j += wc-1; t += wc-1; wc = 1;
                    }
                    else if ( *p >= AM.OffsetIndex + WILDOFFSET
                    && *p < AM.OffsetIndex + 2*WILDOFFSET ) {

                        /* Test wildcard index */

                        wc = *p - WILDOFFSET;
                        if ( CheckWild(BHEAD wc,INDTOIND,*t,&newvalue) ) break;
                        AddWild(BHEAD wc,INDTOIND,newvalue);
                    }
                    else if ( *t < MINSPEC && *p < MINSPEC
                        && *p >= AM.OffsetVector + WILDOFFSET ) {

                        /* Test wildcard vector */

                        wc = *p - WILDOFFSET;
                        if ( CheckWild(BHEAD wc,VECTOVEC,*t,&newvalue) ) break;
                        AddWild(BHEAD wc,VECTOVEC,newvalue);
                    }
                    else break;
                }
                if ( j >= tcount ) { /* Match! */

                    /* Continue with other functions. Make sure of the funnies */

                    AN.RepFunList[AN.RepFunNum++] = offset;
                    AN.RepFunList[AN.RepFunNum++] = 0;
                    newpat = pattern + pattern[1];
                    if ( newpat >= AN.patstop ) {
                        if ( AN.UseFindOnly == 0 ) {
```

```
                            if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                                AT.WorkPointer = oldworkpointer;
                                AN.UsedOtherFind = 1;
                                return(1);
                            }
                            j = 0;
                        }
                        else {
                            AT.WorkPointer = oldworkpointer;
                            return(1);
                        }
                    }
                    else j = ScanFunctions(BHEAD newpat,inter,par);
                    if ( j ) {
                        AT.WorkPointer = oldworkpointer;
                        return(j); /* Full match. Return our success */
                    }
                    AN.RepFunNum -= 2;
                }

                /* No (deeper) match. -> reset wildcards and continue */

                if ( wc ) {
                    j = nwstore;
                    m = AN.WildValue;
                    t = thewildcards + ntwa; r = AT.WildMask;
                    if ( j > 0 ) {
                        do {
                            *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                        } while ( --j > 0 );
                    }
                    C->numrhs = *t++;
                    C->Pointer = C->Buffer + oldcpointer;
                }
                t = params;
                wc = *t;
                for ( j = 1; j < tcount; j++ ) { *t = t[1]; t++; }
                *t = wc;
            }
        }
        (thewildcards[iraise-1])++;
    }
/*
    #] Case 3:
*/
NoSuccess:
    if ( oldwilval > 0 ) {
nomatch:;
        j = nwstore;
        if ( j > 0 ) {
            m = AN.WildValue;
            t = lowlevel; r = AT.WildMask;
            if ( j > 0 ) {
                do {
                    *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                } while ( --j > 0 );
            }
            C->numrhs = *t++;
            C->Pointer = C->Buffer + oldcpointer;
        }
    }
    AT.WorkPointer = oldworkpointer;
    return(0);
}

/*
    #] MatchCy :
    #[ FunMatchCy :

        Matching of (r)cyclic functions.
        Like MatchCy, but now for general functions.
*/

int FunMatchCy(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
{
    GETBIDENTITY
    WORD *t, *tstop, *p, *pstop, *m, *r, *oldworkpointer = AT.WorkPointer;
    WORD **a, *thewildcards, *multiplicity, *renum, wc, wcc, oldwilval = 0;
    LONG oww = AT.pWorkPointer;
    WORD newvalue, *lowlevel = 0;
    int argcount = 0, funnycount = 0, tcount = 0;
    int type = 0, pnum, i, j, k, nwstore, iraise, itop, sumeat;
    CBUF *C = cbuf+AT.ebufnum;
    int ntwa = 3*AN.NumTotWildArgs+1;
    LONG oldcpointer = C->Pointer - C->Buffer;
    WORD offset = fun-AN.terstart, *newpat;

```

```
    if ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
    pnum = pattern[0];
    nwstore = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
    if ( pnum > FUNCTION + WILDOFFSET ) {
        pnum -= WILDOFFSET;
        if ( CheckWild(BHEAD pnum,FUNTOFUN,fun[0],&newvalue) ) return(0);
        oldwilval = 1;
        t = lowlevel = oldworkpointer;
        m = AN.WildValue;
        i = nwstore;
        r = AT.WildMask;
        if ( i > 0 ) {
            do {
                *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
            } while ( --i > 0 );
        }
        *t++ = C->numrhs;
        if ( t >= AT.WorkTop ) {
            MLOCK(ErrorMessageLock);
            MesWork();
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        AT.WorkPointer = t;
        AddWild(BHEAD pnum,FUNTOFUN,newvalue);
    }
    if ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;

    /* First we have to make an inventory. Are there -ARGWILD pointers? */

    p = pattern + FUNHEAD;
    pstop = pattern + pattern[1];
    while ( p < pstop ) {
        if ( *p == -ARGWILD ) { p += 2; funnycount++; }
        else { NEXTARG(p); argcount++; }
    }
    t = fun + FUNHEAD;
    tstop = fun + fun[1];
    while ( t < tstop ) { NEXTARG(t); tcount++; }

    if ( argcount > tcount ) return(0);
    if ( argcount < tcount && funnycount == 0 ) return(0);
    if ( argcount == 0 && tcount == 0 && funnycount == 0 ) {
        AN.RepFunList[AN.RepFunNum++] = offset;
        AN.RepFunList[AN.RepFunNum++] = 0;
        newpat = pattern + pattern[1];
        if ( newpat >= AN.patstop ) {
            if ( AN.UseFindOnly == 0 ) {
                if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                    AT.WorkPointer = oldworkpointer;
                    AN.UsedOtherFind = 1;
                    return(1);
                }
                j = 0;
            }
            else {
                AT.WorkPointer = oldworkpointer;
                return(1);
            }
        }
        else j = ScanFunctions(BHEAD newpat,inter,par);
        if ( j ) return(j);
        goto NoSuccess;
    }

    /* Store the wildcard assignments */

    WantAddPointers(tcount);
    AT.pWorkPointer += tcount;
    thewildcards = t = AT.WorkPointer;
    t += ntwa;
    if ( oldwilval ) lowlevel = oldworkpointer;
    else lowlevel = t;
    m = AN.WildValue;
    i = nwstore;
    if ( i > 0 ) {
        r = AT.WildMask;
        do {
            *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
        } while ( --i > 0 );
        *t++ = C->numrhs;
    }
    if ( t >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
```

```
    }
    AT.WorkPointer = t;
/*
    #[ Case 1: no funnies or all funnies must be empty. We just cycle through.
*/
    if ( argcount == tcount ) {
        if ( funnycount > 0 ) { /* Test all funnies first */
            p = pattern + FUNHEAD;
            t = fun + FUNHEAD;
            while ( p < pstop ) {
                if ( *p != -ARGWILD ) { p++; continue; }
                AN.argaddress = t;
                if ( CheckWild(BHEAD p[1],ARGTOARG,0,t) ) goto nomatch;
                AddWild(BHEAD p[1],ARGTOARG,0);
                p += 2;
            }
            oldwilval = 1;
        }
        for ( k = 0; k <= type; k++ ) {
            if ( k == 0 ) {
                a = AT.pWorkSpace+oww; t = fun + FUNHEAD;
                while ( t < tstop ) { *a++ = t; NEXTARG(t); }
            }
            else {
                a = AT.pWorkSpace+oww+tcount; t = fun + FUNHEAD;
                while ( t < tstop ) { *--a = t; NEXTARG(t); }
            }
            for ( i = 0; i < tcount; i++ ) { /* The various cycles */
                p = pattern + FUNHEAD;
                wc = 0;
                for ( j = 0; j < tcount; j++ ) { /* The arguments */
                    while ( *p == -ARGWILD ) p += 2;
                    t = AT.pWorkSpace[oww+((i+j)%tcount)];
                    if ( ( wcc =  MatchArgument(BHEAD t,p) ) == 0 ) break;
                    if ( wcc > 1 ) wc = 1;
                    NEXTARG(p);
                }
                if ( j >= tcount ) { /* Match! */

                    /* Continue with other functions. Make sure of the funnies */

                    AN.RepFunList[AN.RepFunNum++] = offset;
                    AN.RepFunList[AN.RepFunNum++] = 0;

                    if ( funnycount > 0 ) {
                        p = pattern + FUNHEAD;
                        t = fun + FUNHEAD;
                        while ( p < pstop ) {
                            if ( *p != -ARGWILD ) { p++; continue; }
                            AN.argaddress = t;
                            AddWild(BHEAD p[1],ARGTOARG,0);
                            p += 2;
                        }
                    }
                    newpat = pattern + pattern[1];
                    if ( newpat >= AN.patstop ) {
                        if ( AN.UseFindOnly == 0 ) {
                            if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                                AT.WorkPointer = oldworkpointer;
                                AT.pWorkPointer = oww;
                                AN.UsedOtherFind = 1;
                                return(1);
                            }
                            j = 0;
                        }
                        else {
                            AT.WorkPointer = oldworkpointer;
                            AT.pWorkPointer = oww;
                            return(1);
                        }
                    }
                    else j = ScanFunctions(BHEAD newpat,inter,par);
                    if ( j ) {
                        AT.WorkPointer = oldworkpointer;
                        AT.pWorkPointer = oww;
                        return(j); /* Full match. Return our success */
                    }
                    AN.RepFunNum -= 2;
                }

                /* No (deeper) match. -> reset wildcards and continue */

                if ( wc && nwstore > 0 ) {
                    j = nwstore;
                    m = AN.WildValue;
                    t = thewildcards + ntwa; r = AT.WildMask;
                    if ( j > 0 ) {
```

```
                        do {
                            *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                        } while ( --j > 0 );
                    }
                    C->numrhs = *t++;
                    C->Pointer = C->Buffer + oldcpointer;
                }
            }
        }
        goto NoSuccess;
    }
/*
    #] Case 1:
    #[ Case 2: One -ARGWILD. Fix its length.
*/
    if ( funnycount == 1 ) {
        funnycount = tcount - argcount; /* Number or arguments to be eaten */
        for ( k = 0; k <= type; k++ ) {
            if ( k == 0 ) {
                a =  AT.pWorkSpace+oww; t = fun + FUNHEAD;
                while ( t < tstop ) { *a++ = t; NEXTARG(t); }
            }
            else {
                a =  AT.pWorkSpace+oww+tcount; t = fun + FUNHEAD;
                while ( t < tstop ) { *--a = t; NEXTARG(t); }
            }
            for ( i = 0; i < tcount; i++ ) { /* The various cycles */
                p = pattern + FUNHEAD;
                a = AT.pWorkSpace+oww;
                wc = 0;
                for ( j = 0; j < tcount; j++, a++ ) { /* The arguments */
                    t = *a;
                    if ( *p == -ARGWILD ) {
                        wc = 1;
                        AN.argaddress = (WORD *)a;
                        if ( CheckWild(BHEAD p[1],ARLTOARL,funnycount,(WORD *)a) ) break;
                        AddWild(BHEAD p[1],ARLTOARL,funnycount);
                        j += funnycount-1; a += funnycount-1;
                    }
                    else if ( MatchArgument(BHEAD t,p) == 0 ) break;
                    NEXTARG(p);
                }
                if ( j >= tcount ) { /* Match! */

                    /* Continue with other functions. Make sure of the funnies */

                    AN.RepFunList[AN.RepFunNum++] = offset;
                    AN.RepFunList[AN.RepFunNum++] = 0;
                    newpat = pattern + pattern[1];
                    if ( newpat >= AN.patstop ) {
                        if ( AN.UseFindOnly == 0 ) {
                            if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                                AT.WorkPointer = oldworkpointer;
                                AT.pWorkPointer = oww;
                                AN.UsedOtherFind = 1;
                                return(1);
                            }
                            j = 0;
                        }
                        else {
                            AT.WorkPointer = oldworkpointer;
                            AT.pWorkPointer = oww;
                            return(1);
                        }
                    }
                    else j = ScanFunctions(BHEAD newpat,inter,par);
                    if ( j ) {
                        AT.WorkPointer = oldworkpointer;
                        AT.pWorkPointer = oww;
                        return(j); /* Full match. Return our success */
                    }
                    AN.RepFunNum -= 2;
                }

                /* No (deeper) match. -> reset wildcards and continue */

                if ( wc ) {
                    j = nwstore;
                    m = AN.WildValue;
                    t = thewildcards + ntwa; r = AT.WildMask;
                    if ( j > 0 ) {
                        do {
                            *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                        } while ( --j > 0 );
                    }
                    C->numrhs = *t++;
                    C->Pointer = C->Buffer + oldcpointer;
```

```
                }
                a = AT.pWorkSpace+oww;
                t = *a;
                for ( j = 1; j < tcount; j++ ) { *a = a[1]; a++; }
                *a = t;
            }
        }
        goto NoSuccess;
    }
/*
    #] Case 2:
    #[ Case 3: More than one -ARGWILD. Complicated.
*/

    sumeat = tcount - argcount; /* Total number to be eaten by Funnies */
/*
    In the first funnycount elements of 'thewildcards' we arrange
    for the summing over the various possibilities.
    The renumbering table is in thewildcards[2*funnycount]
    The multiplicity table is in thewildcards[funnycount]
    The number of arguments for each is in thewildcards[]
*/
    p = pattern+FUNHEAD;
    for ( i = funnycount; i < ntwa; i++ ) thewildcards[i] = -1;
    multiplicity = thewildcards + funnycount;
    renum = multiplicity + funnycount;
    j = 0;
    while ( p < pstop ) {
        if ( *p != -ARGWILD ) { p++; continue; }
        p++;
        if ( renum[*p] < 0 ) {
            renum[*p] = j;
            multiplicity[j] = 1;
            j++;
        }
        else multiplicity[renum[*p]]++;
        p++;
    }
/*
    Strategy: First 'declared' has a tendency to be smaller
*/
    for ( i = 1; i < AN.NumTotWildArgs; i++ ) {
        if ( renum[i] < 0 ) continue;
        for ( j = i+1; j <= AN.NumTotWildArgs; j++ ) {
            if ( renum[j] < 0 ) continue;
            if ( renum[i] < renum[j] ) continue;
            k = multiplicity[renum[i]];
            multiplicity[renum[i]] = multiplicity[renum[j]];
            multiplicity[renum[j]] = k;
            k = renum[i]; renum[i] = renum[j]; renum[j] = k;
        }
    }
    for ( i = 0; i < funnycount; i++ ) thewildcards[i] = 0;
    iraise = funnycount-1;
    for ( ;; ) {
        for ( i = 0, j = sumeat; i < iraise; i++ )
                j -= thewildcards[i]*multiplicity[i];
        if ( j < 0 || j % multiplicity[iraise] != 0 ) {
            if ( j > 0 ) {
                thewildcards[iraise-1]++;
                continue;
            }
            itop = iraise-1;
            while ( itop > 0 && j < 0 ) {
                j += thewildcards[itop]*multiplicity[itop];
                thewildcards[itop] = 0;
                itop--;
            }
            if ( itop <= 0 && j <= 0 ) break;
            thewildcards[itop]++;
            continue;
        }
        thewildcards[iraise] = j / multiplicity[iraise];

        for ( k = 0; k <= type; k++ ) {
            if ( k == 0 ) {
                a = AT.pWorkSpace+oww; t = fun + FUNHEAD;
                while ( t < tstop ) { *a++ = t; NEXTARG(t); }
            }
            else {
                a = AT.pWorkSpace+oww+tcount; t = fun + FUNHEAD;
                while ( t < tstop ) { *--a = t; NEXTARG(t); }
            }
            for ( i = 0; i < tcount; i++ ) { /* The various cycles */
                p = pattern + FUNHEAD;
                a = AT.pWorkSpace+oww;
                wc = 0;
```

```
                for ( j = 0; j < tcount; j++, a++ ) { /* The arguments */
                    t = *a;
                    if ( *p == -ARGWILD ) {
                        wc = thewildcards[renum[p[1]]];
                        AN.argaddress = (WORD *)a;
                        if ( CheckWild(BHEAD p[1],ARLTOARL,wc,(WORD *)a) ) break;
                        AddWild(BHEAD p[1],ARLTOARL,wc);
                        j += wc-1; a += wc-1; wc = 1;
                    }
                    else if ( MatchArgument(BHEAD t,p) == 0 ) break;
                    NEXTARG(p);
                }
                if ( j >= tcount ) { /* Match! */

                    /* Continue with other functions. Make sure of the funnies */

                    AN.RepFunList[AN.RepFunNum++] = offset;
                    AN.RepFunList[AN.RepFunNum++] = 0;
                    newpat = pattern + pattern[1];
                    if ( newpat >= AN.patstop ) {
                        if ( AN.UseFindOnly == 0 ) {
                            if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                                AT.WorkPointer = oldworkpointer;
                                AT.pWorkPointer = oww;
                                AN.UsedOtherFind = 1;
                                return(1);
                            }
                            j = 0;
                        }
                        else {
                            AT.WorkPointer = oldworkpointer;
                            AT.pWorkPointer = oww;
                            return(1);
                        }
                    }
                    else j = ScanFunctions(BHEAD newpat,inter,par);
                    if ( j ) {
                        AT.WorkPointer = oldworkpointer;
                        AT.pWorkPointer = oww;
                        return(j); /* Full match. Return our success */
                    }
                    AN.RepFunNum -= 2;
                }

                /* No (deeper) match. -> reset wildcards and continue */

                if ( wc ) {
                    j = nwstore;
                    m = AN.WildValue;
                    t = thewildcards + ntwa; r = AT.WildMask;
                    if ( j > 0 ) {
                        do {
                            *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                        } while ( --j > 0 );
                    }
                    C->numrhs = *t++;
                    C->Pointer = C->Buffer + oldcpointer;
                }
                a = AT.pWorkSpace+oww;
                t = *a;
                for ( j = 1; j < tcount; j++ ) { *a = a[1]; a++; }
                *a = t;
            }
        }
        (thewildcards[iraise-1])++;
    }
/*
    #] Case 3:
*/
NoSuccess:
    if ( oldwilval > 0 ) {
nomatch:;
        j = nwstore;
        m = AN.WildValue;
        t = lowlevel; r = AT.WildMask;
        if ( j > 0 ) {
            do {
                *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
            } while ( --j > 0 );
        }
        C->numrhs = *t++;
        C->Pointer = C->Buffer + oldcpointer;
    }
    AT.WorkPointer = oldworkpointer;
    AT.pWorkPointer = oww;
    return(0);
}
```

```

/*
    #] FunMatchCy :
    #[ FunMatchSy :

        Matching of (anti)symmetric functions.
        Like MatchE, but now for general functions.
*/

int FunMatchSy(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
{
    GETBIDENTITY
    WORD *t, *tstop, *p, *pstop, *m, *r, *oldworkpointer = AT.WorkPointer;
    WORD **a, *thewildcards, oldwilval = 0;
    WORD newvalue, *lowlevel = 0, num, assig;
    WORD *cycles;
    LONG oww = AT.pWorkPointer, lhpars, lhfunnies;
    int argcount = 0, funnycount = 0, tcount = 0, signs = 0, signfun = 0, signo;
    int type = 0, pnum, i, j, k, nwstore, iraise, cou2;
    CBUF *C = cbuf+AT.ebufnum;
    int ntwa = 3*AN.NumTotWildArgs+1;
    LONG oldcpointer = C->Pointer - C->Buffer;
    WORD offset = fun-AN.terstart, *newpat;

    if ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
    pnum = pattern[0];
    nwstore = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
    if ( pnum > FUNCTION + WILDOFFSET ) {
        pnum -= WILDOFFSET;
        if ( CheckWild(BHEAD pnum,FUNTOFUN,fun[0],&newvalue) ) return(0);
        oldwilval = 1;
        t = lowlevel = oldworkpointer;
        m = AN.WildValue;
        i = nwstore;
        r = AT.WildMask;
        if ( i > 0 ) {
            do {
                *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
            } while ( --i > 0 );
        }
        *t++ = C->numrhs;
        if ( t >= AT.WorkTop ) {
            MLOCK(ErrorMessageLock);
            MesWork();
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        AT.WorkPointer = t;
        AddWild(BHEAD pnum,FUNTOFUN,newvalue);
    }
    if ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;

    /* Try for a straight match. After all, both have been normalized */

    if ( fun[1] == pattern[1] ) {
        i = fun[1]-FUNHEAD; p = pattern+FUNHEAD; t = fun + FUNHEAD;
        while ( --i >= 0 ) { if ( *p++ != *t++ ) break; }
        if ( i < 0 ) goto quicky;
    }

    /* First we have to make an inventory. Are there -ARGWILD pointers? */

    p = pattern + FUNHEAD;
    pstop = pattern + pattern[1];
    while ( p < pstop ) {
        if ( *p == -ARGWILD ) { p += 2; funnycount++; }
        else { NEXTARG(p); argcount++; }
    }
    t = fun + FUNHEAD;
    tstop = fun + fun[1];
    while ( t < tstop ) { NEXTARG(t); tcount++; }

    if ( argcount > tcount ) return(0);
    if ( argcount < tcount && funnycount == 0 ) return(0);
    if ( argcount == 0 && tcount == 0 && funnycount == 0 ) {
quicky:
        if ( AN.SignCheck && signs != AN.ExpectedSign ) goto NoSuccess;
        AN.RepFunList[AN.RepFunNum++] = offset;
        AN.RepFunList[AN.RepFunNum++] = signs;
        newpat = pattern + pattern[1];
        if ( newpat >= AN.patstop ) {
            if ( AN.UseFindOnly == 0 ) {
                if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                    AT.WorkPointer = oldworkpointer;
                    AN.UsedOtherFind = 1;
                    return(1);
                }
```

```
                j = 0;
            }
            else {
                AT.WorkPointer = oldworkpointer;
                return(1);
            }
        }
        else j = ScanFunctions(BHEAD newpat,inter,par);
        if ( j ) {
            AT.WorkPointer = oldworkpointer;
            return(j);
        }
        goto NoSuccess;
    }

    /* Store the wildcard assignments */

    WantAddPointers(tcount+argcount+funnycount);
    AT.pWorkPointer += tcount+argcount+funnycount;
    thewildcards = t = AT.WorkPointer;
    t += ntwa;
    if ( oldwilval ) lowlevel = oldworkpointer;
    else lowlevel = t;
    m = AN.WildValue;
    i = nwstore; assig = 0;
    if ( i > 0 ) {
        r = AT.WildMask;
        do {
            assig += *r;
            *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
        } while ( --i > 0 );
        *t++ = C->numrhs;
    }
    if ( t >= AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    AT.WorkPointer = t;

    /* Store pointers to the arguments */

    t = fun + FUNHEAD; a = AT.pWorkSpace+oww;
    while ( t < tstop ) { *a++ = t; NEXTARG(t) }
    lhpars = a-AT.pWorkSpace;
    t = pattern + FUNHEAD;
    while ( t < pstop ) {
        if ( *t != -ARGWILD ) *a++ = t;
        NEXTARG(t)
    }
    lhfunnies = a-AT.pWorkSpace;
    t = pattern + FUNHEAD; cou2 = 0;
    while ( t < pstop ) {
        cou2++;
        if ( *t == -ARGWILD ) {
            *a++ = t;
/*
    signfun: last ?a: tcount-argcount: number of arguments in ?a (assume one ?a)
        argcount+funnycount-cou2: arguments after ?a.
    Together tells whether moving ?a to end of list is even or odd
*/
            signfun = ((argcount+funnycount-cou2)*(tcount-argcount)) & 1;
        }
        NEXTARG(t)
    }
    signs += signfun;
    if ( funnycount > 0 ) {
        if ( ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == SYMMETRIC )
            || ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
            || ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == SYMMETRIC )
            || ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC ) ) {
                AT.WorkPointer = oldworkpointer;
                AT.pWorkPointer = oww;
                MLOCK(ErrorMessageLock);
                MesPrint("Sorry: no argument field wildcards yet in (anti)symmetric functions");
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
        }
    }
/*
    Sort the regular arguments by
    1: no wildcards, fast.
    2: wildcards that have been assigned.
    3: general arguments.
    4: wildcards without an assignment.
*/
```

```
    iraise = argcount;
    for ( i = 0; i < iraise; i++ ) {
        t = AT.pWorkSpace[i+lhpars];
        if ( *t > 0 ) { /* Category 3: general argument */
            continue;
        }
        else if ( *t <= -FUNCTION ) {
            if ( *t > -FUNCTION - WILDOFFSET ) goto cat1;
            type = FUNTOFUN; num = -*t - WILDOFFSET;
        }
        else if ( *t == -SYMBOL ) {
            if ( t[1] < 2*MAXPOWER ) goto cat1;
            type = SYMTOSYM; num = t[1] - 2*MAXPOWER;
        }
        else if ( *t == -INDEX ) {
            if ( t[1] < AM.OffsetIndex + WILDOFFSET ) goto cat1;
            type = INDTOIND; num = t[1] - WILDOFFSET;
        }
        else if ( *t == -VECTOR || *t == -MINVECTOR ) {
            if ( t[1] < AM.OffsetVector + WILDOFFSET ) goto cat1;
            type = VECTOVEC; num = t[1] - WILDOFFSET;
        }
        else goto cat1; /* Things like -SNUMBER etc. */
/*
        Now we have a wildcard and have to see whether it was assigned
*/
        m = AN.WildValue;
        j = nwstore;
        r = AT.WildMask;
        while ( --j >= 0 ) {
            if ( m[2] == num && *r ) {
                if ( type == *m ) break;
                if ( type == SYMTOSYM ) {
                    if ( *m == SYMTONUM || *m == SYMTOSUB ) break;
                }
                else if ( type == INDTOIND ) {
                    if ( *m == INDTOSUB ) break;
                }
                else if ( type == VECTOVEC ) {
                    if ( *m == VECTOMIN || *m == VECTOSUB ) break;
                }
            }
            m += 4; r++;
        }
        if ( j < 0 ) {  /* Category 4: Wildcard that was not assigned */
            a = AT.pWorkSpace+lhpars;
            iraise--;
            if ( iraise != i ) signs++;
            m = a[iraise];
            a[iraise] = a[i];
            a[i] = m; i--;
        }
        else {  /* Category 2: Wildcard that was assigned */
            for ( j = 0; j < tcount; j++ ) {
                if ( MatchArgument(BHEAD AT.pWorkSpace[oww+j],t) ) {
                    k = nwstore;
                    r = AT.WildMask;
                    num = 0;
                    while ( --k >= 0 ) num += *r++;
                    if ( num == assig ) { /* no wildcards were changed */
                        goto oneless;
                    }
                    break;
                }
            }
            if ( j >= tcount ) goto NoSuccess;
            j = nwstore;
            m = AN.WildValue;
            t = thewildcards + ntwa; r = AT.WildMask;
            if ( j > 0 ) {
                do {    /* undo assignment */
                    *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                } while ( --j > 0 );
            }
            C->numrhs = *t++;
        }
        continue;
cat1:
        for ( j = 0; j < tcount; j++ ) {
            m = AT.pWorkSpace[j+oww];
            if ( *t != *m ) continue;
            if ( *t < 0 ) {
                if ( *t <= -FUNCTION ) break;
                if ( t[1] == m[1] ) break;
            }
            else {
                k = *t; r = t;
```

```
                while ( --k >= 0 && *m++ == *r++ ) {}
                if ( k < 0 ) break;
            }
        }
        if ( j >= tcount ) goto NoSuccess;  /* Even the fixed ones don't match */
oneless:
        signs += j - i;
/*
        The next statements replace the one that is commented out
*/
        tcount--;
        while ( j < tcount ) {
            AT.pWorkSpace[oww+j] = AT.pWorkSpace[oww+j+1]; j++;
        }
/*
        AT.pWorkSpace[oww+j] = AT.pWorkSpace[oww+(--tcount)];
*/
        argcount--; j = i;
        while ( j < argcount ) {
            AT.pWorkSpace[lhpars+j] = AT.pWorkSpace[lhpars+j+1]; j++;
        }
        iraise--; i--;
    }
/*
    Now we see whether there are any ARGWILD objects that have been
    assigned already. In that case the work simplifies considerably.
    Currently (12-nov-2001) only in (R)CYCLIC functions; hence we do not
    test the sign!
*/
    for ( i = 0; i < funnycount; i++ ) {
        k = AT.pWorkSpace[lhfunnies+i][1];
        m = AN.WildValue;
        j = nwstore;
        r = AT.WildMask;
        while ( --j >= 0 ) {
            if ( *m == ARGTOARG && m[2] == k ) break;
            m += 4; r++;
        }
        if ( *r == 0 ) continue; /* not assigned yet */
        m = cbuf[AT.ebufnum].rhs[m[3]];
        if ( *m > 0 ) {     /* Tensor arguments */
            j = *m;
            if ( j > tcount - argcount ) goto NoSuccess;
            while ( --j >= 0 ) {
                m++;
                if ( *m < 0 ) type = -VECTOR;
                else if ( *m < AM.OffsetIndex ) type = -SNUMBER;
                else type = -INDEX;
                a = AT.pWorkSpace+oww;
                for ( k = 0; k < tcount; k++ ) {
                    if ( a[k][0] != type || a[k][1] != *m ) continue;
                    a[k] = a[--tcount];
                    goto nextjarg;
                }
                goto NoSuccess;
nextjarg:;
            }
        }
        else {
            m++;
            while ( *m ) {
                for ( k = 0; k < tcount; k++ ) {
                    t = AT.pWorkSpace[oww+k];
                    if ( *t != *m ) continue;
                    r = m;
                    if ( *r < 0 ) {
                        if ( *r < -FUNCTION ) goto nextargw;
                        else if ( r[1] == t[1] ) goto nextargw;
                    }
                    else {
                        j = *r;
                        while ( --j >= 0 && *r++ == *t++ ) {}
                        if ( j < 0 ) goto nextargw;
                    }
                }
                goto NoSuccess;
nextargw:;
                AT.pWorkSpace[oww+k] = AT.pWorkSpace[oww+(--tcount)];
                NEXTARG(m)
            }
        }
        AT.pWorkSpace[lhfunnies+i] = AT.pWorkSpace[lhfunnies+(--funnycount)];
    }
    if ( tcount == 0 ) {
        if ( argcount > 0 ) goto NoSuccess;
        for ( i = 0; i < funnycount; i++ ) {
            AddWild(BHEAD AT.pWorkSpace[lhfunnies+i][1],ARGTOARG,0);
```

```
        }
        goto quicky;
    }
/*
    We have now in lhpars first iraise elements with a dubious nature.
    Then argcount-iraise wildcards that have not been assigned.
    In lhfunnies we have funnycount ARGTOARG objects. ( (R)CyCLIC only )

    First work our way through the 'dubious' objects
    We check whether assig changes.
*/
    for ( i = 0; i < iraise; i++ ) {
        for ( j = 0; j < tcount; j++ ) {
            if ( MatchArgument(BHEAD AT.pWorkSpace[oww+j],AT.pWorkSpace[lhpars+i]) ) {
                k = nwstore;
                r = AT.WildMask;
                num = 0;
                while ( --k >= 0 ) num += *r++;
                if ( num == assig ) { /* no wildcards were changed */
                    signs += j-i;
                    AT.pWorkSpace[oww+j] = AT.pWorkSpace[oww+(--tcount)];
                    if ( tcount > j ) signs += tcount-j-1;
                    argcount--;
                    a = AT.pWorkSpace + lhpars;
                    for ( j = i; j < argcount; j++ ) a[j] = a[j+1];
                    iraise--;
                    goto nextiraise;
                }
                else {  /* We cannot use this yet */
                    j = nwstore;
                    m = AN.WildValue;
                    t = thewildcards + ntwa; r = AT.WildMask;
                    if ( j > 0 ) {
                        do {    /* undo assignment */
                            *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
                        } while ( --j > 0 );
                    }
                    C->numrhs = *t++;
                    C->Pointer = C->Buffer + oldcpointer;
                    goto nextiraise;
                }
            }
        }
        goto NoSuccess;
nextiraise:;
    }
/*
    Now all leftover patterns have unassigned wildcards in them.
    From now on we are in potential factorial territory.

    Strategy:
        1: cycle through the regular objects.
        2: save wildcard settings
        3: divide the ARGWILDs
        4: make permutations of leftover arguments
        5: try them all
*/
    cycles = AT.WorkPointer;
    for ( i = 0; i < tcount; i++ ) cycles[i] = tcount-i;
    AT.WorkPointer += tcount;
    signo = 0;
/*MesPrint("<1> signs = %d",signs);*/
    for (;;) {
        WORD oRepFunNum = AN.RepFunNum;
        for ( j = 0; j < argcount; j++ ) {
            if ( MatchArgument(BHEAD AT.pWorkSpace[oww+j],AT.pWorkSpace[lhpars+j]) == 0 ) {
                break;
            }
        }
        if ( j >= argcount ) {
/*
            Thus far we have a match. Now the funnies
*/
            if ( funnycount ) {
                AT.WorkPointer = oldworkpointer;
                AT.pWorkPointer = oww;
                MLOCK(ErrorMessageLock);
                MesPrint("Sorry: no argument field wildcards yet in (anti)symmetric functions");
                MUNLOCK(ErrorMessageLock);
/*
                Bugfix 31-oct-2001, reported by Kasper Peeters
                We returned here with value -1 but that is not caught.
                Extra note (12-nov-2001): the sign becomes a bit problematic
                if we have funnies. No more than one allowed in antisymmetric
                functions, or we have serious problems.
*/
                Terminate(-1);
```

```
            }

            AN.RepFunList[AN.RepFunNum++] = offset;
            if ( ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
                || ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC ) ) {
                AN.RepFunList[AN.RepFunNum++] = ( signs + signo ) & 1;
            }
            else {
                AN.RepFunList[AN.RepFunNum++] = 0;
            }
            newpat = pattern + pattern[1];
            if ( newpat >= AN.patstop ) {
                WORD countsgn, sgn = 0;
                for ( countsgn = oRepFunNum+1; countsgn < AN.RepFunNum; countsgn += 2 ) {
                    if ( AN.RepFunList[countsgn] ) sgn ^= 1;
                }
                if ( AN.SignCheck == 0 || sgn == AN.ExpectedSign ) {
                    AT.WorkPointer = oldworkpointer;
                    AT.pWorkPointer = oww;
                    return(1);
                }
                if ( AN.UseFindOnly == 0 ) {
                    if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
                        AT.WorkPointer = oldworkpointer;
                        AT.pWorkPointer = oww;
                        AN.UsedOtherFind = 1;
                        return(1);
                    }
                }
                j = 0;
            }
            else j = ScanFunctions(BHEAD newpat,inter,par);
            if ( j ) {
                WORD countsgn, sgn = 0;
                for ( countsgn = oRepFunNum+1; countsgn < AN.RepFunNum; countsgn += 2 ) {
                    if ( AN.RepFunList[countsgn] ) sgn ^= 1;
                }
                if ( AN.SignCheck == 0 || sgn == AN.ExpectedSign ) {
                    AT.WorkPointer = oldworkpointer;
                    AT.pWorkPointer = oww;
                    return(j);
                }
            }
            AN.RepFunNum = oRepFunNum;
            i = argcount - 1;
        }
        else i = j;
        j = nwstore;
        m = AN.WildValue;
        t = thewildcards + ntwa; r = AT.WildMask;
        if ( j > 0 ) {
            do {
                *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
            } while ( --j > 0 );
        }
        C->numrhs = *t++;
        C->Pointer = C->Buffer + oldcpointer;
/*
        On to the next cycle
*/
        a = AT.pWorkSpace + oww;
        for ( j = i+1, t = a[i]; j < tcount; j++ ) a[j-1] = a[j];
        a[tcount-1] = t; cycles[i]--;
        signo += tcount - i - 1;
        while ( cycles[i] <= 0 ) {
            cycles[i] = tcount - i;
            i--;
            if ( i < 0 ) goto NoSuccess;
/*
            MLOCK(ErrorMessageLock);
            MesPrint("Cycle i = %d",i);
            MUNLOCK(ErrorMessageLock);
*/
            for ( j = i+1, t = a[i]; j < tcount; j++ ) a[j-1] = a[j];
            a[tcount-1] = t; cycles[i]--;
            signo += tcount - i - 1;
        }
    }
NoSuccess:
    if ( oldwilval > 0 ) {
        j = nwstore;
        m = AN.WildValue;
        t = lowlevel; r = AT.WildMask;
        if ( j > 0 ) {
            do {
                *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
            } while ( --j > 0 );
```

```
        }
        C->numrhs = *t++;
        C->Pointer = C->Buffer + oldcpointer;
    }
    AT.WorkPointer = oldworkpointer;
    AT.pWorkPointer = oww;
    return(0);
}

/*
    #] FunMatchSy :
    #[ MatchArgument :
*/

int MatchArgument(PHEAD WORD *arg, WORD *pat)
{
    GETBIDENTITY
    WORD *m = pat, *t = arg, i, j, newvalue;
    WORD *argmstop = pat, *argtstop = arg;
    WORD *cto, *cfrom, *csav, ci;
    WORD oRepFunNum, *oRepFunList;
    WORD *oterstart,*oterstop,*opatstop;
    WORD wildargs, wildeat;
    WORD *mtrmstop, *ttrmstop, *msubstop, msizcoef;
    WORD *wildargtaken;
    int wc = 1;

    NEXTARG(argmstop);
    NEXTARG(argtstop);
/*
    #[ Both fast :
*/
    if ( *m < 0 && *t < 0 ) {
        if ( *t <= -FUNCTION ) {
            if ( *t == *m ) {}
            else if ( *m <= -FUNCTION-WILDOFFSET
            && functions[-*t-FUNCTION].spec
            == functions[-*m-FUNCTION-WILDOFFSET].spec ) {
                i = -*m - WILDOFFSET; wc = 2;
                if ( CheckWild(BHEAD i,FUNTOFUN,-*t,&newvalue) ) {
                    return(0);
                }
                AddWild(BHEAD i,FUNTOFUN,newvalue);
            }
            else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER ) {
                i = m[1] - 2*MAXPOWER;
                AN.argaddress = AT.FunArg;
                AT.FunArg[ARGHEAD+1] = -*t;
                if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) return(0);
                AddWild(BHEAD i,SYMTOSUB,0);
            }
            else return(0);
        }
        else if ( *t == *m ) {
            if ( t[1] == m[1] ) {}
            else if ( *t == -SYMBOL ) {
                j = SYMTOSYM;
SymAll:         if ( ( i = m[1] - 2*MAXPOWER ) < 0 ) return(0);
                wc = 2;
                if ( CheckWild(BHEAD i,j,t[1],&newvalue) ) return(0);
                AddWild(BHEAD i,j,newvalue);
            }
            else if ( *t == -INDEX ) {
IndAll:         i = m[1] - WILDOFFSET;
                if ( i < AM.OffsetIndex || i >= WILDOFFSET+AM.OffsetIndex )
                                                        return(0);
                                /* We kill the summed over indices here */
                wc = 2;
                if ( CheckWild(BHEAD i,INDTOIND,t[1],&newvalue) ) return(0);
                AddWild(BHEAD i,INDTOIND,newvalue);
            }
            else if ( *t == -VECTOR || *t == -MINVECTOR ) {
                i = m[1] - WILDOFFSET;
                if ( i < AM.OffsetVector ) return(0);
                wc = 2;
                if ( CheckWild(BHEAD i,VECTOVEC,t[1],&newvalue) ) return(0);
                AddWild(BHEAD i,VECTOVEC,newvalue);
            }
            else return(0);
        }
        else if ( *m == -INDEX && m[1] >= AM.OffsetIndex+WILDOFFSET
        && m[1] < AM.OffsetIndex+(WILDOFFSET«1) ) {
            if ( *t == -VECTOR ) goto IndAll;
            if ( *t == -SNUMBER && t[1] >= 0 && t[1] < AM.OffsetIndex ) goto IndAll;
            if ( *t == -MINVECTOR ) {
                i = m[1] - WILDOFFSET;
                AN.argaddress = AT.MinVecArg;
```

```
                AT.MinVecArg[ARGHEAD+3] = t[1];
                wc = 2;
                if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) return(0);
                AddWild(BHEAD i,INDTOSUB,(WORD)0);
            }
            else return(0);
        }
        else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER && *t == -SNUMBER ) {
            j = SYMTONUM;
            goto SymAll;
        }
        else if ( *m == -VECTOR && *t == -MINVECTOR &&
        ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
            wc = 2;
/*
            AN.argaddress = AT.MinVecArg;
            AT.MinVecArg[ARGHEAD+3] = t[1];
            if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) return(0);
            AddWild(BHEAD i,VECTOSUB,(WORD)0);
*/
            if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) return(0);
            AddWild(BHEAD i,VECTOMIN,newvalue);

        }
        else if ( *m == -MINVECTOR && *t == -VECTOR &&
        ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
            wc = 2;
/*
            AN.argaddress = AT.MinVecArg;
            AT.MinVecArg[ARGHEAD+3] = t[1];
            if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) return(0);
            AddWild(BHEAD i,VECTOSUB,(WORD)0);
*/
            if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) return(0);
            AddWild(BHEAD i,VECTOMIN,newvalue);
        }
        else return(0);
    }
/*
    #] Both fast :
    #[ Fast arg :
*/
    else if ( *m > 0 && *t <= -FUNCTION ) {
        if ( ( m[ARGHEAD]+ARGHEAD == *m ) && m[*m-1] == 3
        && m[*m-2] == 1 && m[*m-3] == 1 && m[ARGHEAD+1] >= FUNCTION
        && m[ARGHEAD+2] == *m-ARGHEAD-4 ) { /* Check for f(?a) etc */
            WORD *mmmst, *mmm, mmmi;
            if ( m[ARGHEAD+1] >= FUNCTION+WILDOFFSET ) {
                mmmi = *m - WILDOFFSET;
                wc = 2;
                if ( CheckWild(BHEAD mmmi,FUNTOFUN,-*t,&newvalue) ) return(0);
                AddWild(BHEAD mmmi,FUNTOFUN,newvalue);
            }
            else if ( m[ARGHEAD+1] != -*t ) return(0);
/*
                Only arguments allowed are ?a etc.
*/
            mmmst = m+*m-3;
            mmm = m + ARGHEAD + FUNHEAD + 1;
            while ( mmm < mmmst ) {
                if ( *mmm != -ARGWILD ) return(0);
                mmmi = 0;
                AN.argaddress = t; wc = 2;
                if ( CheckWild(BHEAD mmm[1],ARGTOARG,mmmi,t) ) return(0);
                AddWild(BHEAD mmm[1],ARGTOARG,mmmi);
                mmm += 2;
            }
        }
        else return(0);
    }
/*
    #] Fast arg :
    #[ Fast pat :
*/
    else if ( *m < 0 && *t > 0 ) {
        if ( *m == -SYMBOL ) {          /* SYMTOSUB */
            if ( m[1] < 2*MAXPOWER ) return(0);
            i = m[1] - 2*MAXPOWER;
            AN.argaddress = t; wc = 2;
            if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) return(0);
            AddWild(BHEAD i,SYMTOSUB,0);
        }
        else if ( *m == -VECTOR ) {
            if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetVector ) return(0);
            AN.argaddress = t; wc = 2;
            if ( CheckWild(BHEAD i,VECTOSUB,1,t) ) return(0);
            AddWild(BHEAD i,VECTOSUB,(WORD)0);
```

```
        }
        else if ( *m == -INDEX ) {
            if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetIndex ) return(0);
            if ( i >= AM.OffsetIndex + WILDOFFSET ) return(0);
            AN.argaddress = t; wc = 2;
            if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) return(0);
            AddWild(BHEAD i,INDTOSUB,(WORD)0);
        }
        else return(0);
    }
/*
    #] Fast pat :
    #[ Both general :
*/
    else if ( *m > 0 && *t > 0 ) {
        i = *m;
        do { if ( *m++ != *t++ ) break; } while ( --i > 0 );
        if ( i > 0 ) {
/*
            Not an exact match here.
            We have to hope that the pattern contains a composite wildcard.
*/
            m = pat; t = arg;
            m += ARGHEAD; t += ARGHEAD;         /* Point at (first?) term */
            mtrmstop = m + *m;
            ttrmstop = t + *t;
            if ( mtrmstop < argmstop ) return(0);/* More than one term */
            msizcoef = mtrmstop[-1];
            if ( msizcoef < 0 ) msizcoef = -msizcoef;
            msubstop = mtrmstop - msizcoef;
            m++;
            if ( m >= msubstop ) return(0); /* Only coefficient */
/*
            Here we have a composite term. It can match provided it
            matches the entire argument. This argument must be a
            single term also and the coefficients should match
            (more or less).
            The matching takes:
            1:  Match the functions etc. Nothing can be left.
            2:  Match dotproducts and symbols. ONLY must match
                and nothing may be left.
            For safety it is best to take the term out and put it
            in workspace.
*/
            if ( argtstop > ttrmstop ) return(0);
            m--;

            oterstart = AN.terstart;
            oterstop = AN.terstop;
            opatstop = AN.patstop;
            oRepFunList = AN.RepFunList;
            oRepFunNum = AN.RepFunNum;
            AN.RepFunNum = 0;
            wildargtaken = AT.WorkPointer;
            AN.RepFunList = wildargtaken + AN.NumTotWildArgs;
            AT.WorkPointer = (WORD *)(((UBYTE *)(AN.RepFunList)) + AM.MaxTer/2);
            csav = cto = AT.WorkPointer;
            cfrom = t;
            ci = *t;
            while ( --ci >= 0 ) *cto++ = *cfrom++;
            AT.WorkPointer = cto;
            ci = msizcoef;
            cfrom = mtrmstop;
            while ( --ci >= 0 ) {
                if ( *--cfrom != *--cto ) {
                    AT.WorkPointer = wildargtaken;
                    AN.RepFunList = oRepFunList;
                    AN.RepFunNum = oRepFunNum;
                    AN.terstart = oterstart;
                    AN.terstop = oterstop;
                    AN.patstop = opatstop;
                    return(0);
                }
            }
            *m -= msizcoef;
            wildargs = AN.WildArgs;
            wildeat = AN.WildEat;
            for ( i = 0; i < wildargs; i++ ) wildargtaken[i] = AT.WildArgTaken[i];
            AN.ForFindOnly = 0; AN.UseFindOnly = 1;
            AN.nogroundlevel++;
            if ( FindRest(BHEAD csav,m) && ( AN.UsedOtherFind || FindOnly(BHEAD csav,m) ) ) { }
            else {
                *m += msizcoef;
                AT.WorkPointer = wildargtaken;
                AN.RepFunList = oRepFunList;
                AN.RepFunNum = oRepFunNum;
                AN.terstart = oterstart;
```

```
                AN.terstop = oterstop;
                AN.patstop = opatstop;
                AN.WildArgs = wildargs;
                AN.WildEat = wildeat;
                AN.nogroundlevel--;
                for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
                return(0);
            }
            AN.nogroundlevel--;
            AN.WildArgs = wildargs;
            AN.WildEat = wildeat;
            for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
            Substitute(BHEAD csav,m,1);
            cto = csav;
            cfrom = cto + *cto - msizcoef;
            cto++;
            *m += msizcoef;
            AT.WorkPointer = wildargtaken;
            AN.RepFunList = oRepFunList;
            AN.RepFunNum = oRepFunNum;
            AN.terstart = oterstart;
            AN.terstop = oterstop;
            AN.patstop = opatstop;
            if ( *cto != SUBEXPRESSION ) return(0);
            cto += cto[1];
            if ( cto < cfrom ) return(0);
        }
    }
/*
    #] Both general :
*/
    else return(0);
/*
    And now the success: (wc = 2 means that there was a wildcard involved)
*/
    return(wc);
}

/*
    #] MatchArgument :
*/
```

4.112 tables.c File Reference

```
#include "form3.h"
#include "minos.h"
```

Include dependency graph for tables.c:

Functions

- void ClearTableTree (TABLES T)
- int InsTableTree (TABLES T, WORD *tp)
- void RedoTableTree (TABLES T, int newsize)
- int FindTableTree (TABLES T, WORD *tp, int inc)
- WORD DoTableExpansion (WORD *term, WORD level)

- int CoTableBase (UBYTE *s)
- int FlipTable (FUNCTIONS f, int type)
- int SpareTable (TABLES TT)
- DBASE * FindTB (UBYTE *name)
- int CoTBcreate (UBYTE *s)
- int CoTBopen (UBYTE *s)
- int CoTBaddto (UBYTE *s)
- int CoTBenter (UBYTE *s)
- int CoTestUse (UBYTE *s)
- int CheckTableDeclarations (DBASE *d)
- int CoTBload (UBYTE *ss)
- WORD TestUse (WORD *term, WORD level)
- int CoTBaudit (UBYTE *s)
- int CoTBon (UBYTE *s)
- int CoTBoff (UBYTE *s)
- int CoTBcleanup (UBYTE *s)
- int CoTBreplace (UBYTE *s)
- int CoTBuse (UBYTE *s)
- int CoApply (UBYTE *s)
- int CoTBhelp (UBYTE *s)
- VOID ReWorkT (WORD *term, WORD *funs, WORD numfuns)
- WORD Apply (WORD *term, WORD level)
- int ApplyExec (WORD *term, int maxtogo, WORD level)
- WORD ApplyReset (WORD level)
- WORD TableReset ()
- int ReleaseTB ()

Variables

- char * helptb [ ]

4.112.1 Detailed Description

Contains all functions that deal with the table bases on the 'FORM level' The low level databse routines are in minos.c

Definition in file tables.c.

4.112.2 Function Documentation

4.112.2.1 ClearTableTree()

```
void ClearTableTree (
            TABLES T )
```

Definition at line 71 of file tables.c.

4.112.2.2 InsTableTree()

```
int InsTableTree (
            TABLES T,
            WORD * tp )
```

Definition at line 102 of file tables.c.

4.112.2.3 RedoTableTree()

```
void RedoTableTree (
            TABLES T,
            int newsize )
```

Definition at line 263 of file tables.c.

4.112.2.4 FindTableTree()

```
int FindTableTree (
            TABLES T,
            WORD * tp,
            int inc )
```

Definition at line 288 of file tables.c.

4.112.2.5 DoTableExpansion()

```
WORD DoTableExpansion (
            WORD * term,
            WORD level )
```

Definition at line 326 of file tables.c.

4.112.2.6 CoTableBase()

```
int CoTableBase (
            UBYTE * s )
```

Definition at line 575 of file tables.c.

4.112.2.7 FlipTable()

```
int FlipTable (
            FUNCTIONS f,
            int type )
```

Definition at line 619 of file tables.c.

4.112.2.8 SpareTable()

```
int SpareTable (
            TABLES TT )
```

Definition at line 642 of file tables.c.

4.112.2.9 FindTB()

```
DBASE * FindTB (
            UBYTE * name )
```

Definition at line 682 of file tables.c.

4.112.2.10 CoTBcreate()

```
int CoTBcreate (
            UBYTE * s )
```

Definition at line 704 of file tables.c.

4.112.2.11 CoTBopen()

```
int CoTBopen (
            UBYTE * s )
```

Definition at line 720 of file tables.c.

4.112.2.12 CoTBaddto()

```
int CoTBaddto (
            UBYTE * s )
```

Definition at line 738 of file tables.c.

4.112.2.13 CoTBenter()

```
int CoTBenter (
            UBYTE * s )
```

Definition at line 906 of file tables.c.

4.112.2.14 CoTestUse()

```
int CoTestUse (
            UBYTE * s )
```

Definition at line 1064 of file tables.c.

4.112.2.15 CheckTableDeclarations()

```
int CheckTableDeclarations (
            DBASE * d )
```

Definition at line 1109 of file tables.c.

4.112.2.16 CoTBload()

```
int CoTBload (
            UBYTE * ss )
```

Definition at line 1178 of file tables.c.

4.112.2.17 TestUse()

```
WORD TestUse (
            WORD * term,
            WORD level )
```

Definition at line 1334 of file tables.c.

4.112.2.18 CoTBaudit()

```
int CoTBaudit (
            UBYTE * s )
```

Definition at line 1393 of file tables.c.

4.112.2.19 CoTBon()

```
int CoTBon (
            UBYTE * s )
```

Definition at line 1433 of file tables.c.

4.112.2.20 CoTBoff()

```
int CoTBoff (
            UBYTE * s )
```

Definition at line 1464 of file tables.c.

4.112.2.21 CoTBcleanup()

```
int CoTBcleanup (
            UBYTE * s )
```

Definition at line 1495 of file tables.c.

4.112.2.22 CoTBreplace()

```
int CoTBreplace (
            UBYTE * s )
```

Definition at line 1507 of file tables.c.

4.112.2.23 CoTBuse()

```
int CoTBuse (
            UBYTE * s )
```

Definition at line 1522 of file tables.c.

4.112.2.24 CoApply()

```
int CoApply (
            UBYTE * s )
```

Definition at line 1713 of file tables.c.

4.112.2.25 CoTBhelp()

```
int CoTBhelp (
            UBYTE * s )
```

Definition at line 1800 of file tables.c.

4.112.2.26 ReWorkT()

```
VOID ReWorkT (
            WORD * term,
            WORD * funs,
            WORD numfuns )
```

Definition at line 1816 of file tables.c.

4.112.2.27 Apply()

```
WORD Apply (
            WORD * term,
            WORD level )
```

Definition at line 1897 of file tables.c.

4.112.2.28 ApplyExec()

```
int ApplyExec (
            WORD * term,
            int maxtogo,
            WORD level )
```

Definition at line 1956 of file tables.c.

4.112.2.29 ApplyReset()

```
WORD ApplyReset (
            WORD level )
```

Definition at line 2173 of file tables.c.

4.112.2.30 TableReset()

```
WORD TableReset ( )
```

Definition at line 2209 of file tables.c.

4.112.2.31 ReleaseTB()

```
int ReleaseTB ( )
```

Definition at line 2236 of file tables.c.

4.112.3 Variable Documentation

4.112.3.1 helptb

```
char* helptb[]
```

Definition at line 1768 of file tables.c.

4.113 tables.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes :

    File contains the routines for the tree structure of sparse tables
    We insert elements by
    InsTableTree(T,tp) with T the TABLES element and tp the pointer
        to the indices.
    We look for elements with
    FindTableTree(T,tp,inc) with T the TABLES element, tp the pointer to the
        indices or the function arguments and inc tells which of these options.
    The tree is cleared with ClearTableTree(T) and we rebuild the tree
    after a .store in which we lost a part of the table with
    RedoTableTree(T,newsize)

    In T->tablepointers we have the lists of indices for each element.
    Additionally for each element there is an extension. There are
    TABLEEXTENSION WORDs reserved for that. The old system had two words
    One for the element in the rhs of the compile buffer and one for
    an additional rhs in case the original would be overwritten by a new
    definition, but the old was fixed by .global and hence it should be possible
    to restore it.
    New use (new = 24-sep-2001)
        rhs1,numCompBuffer1,rhs2,numCompBuffer2,usage
    Hence TABLEEXTENSION will be 5. Note that for 64 bits the use of the
    compiler buffer is overdoing it a bit, but it would be too complicated
    to try to give it special code.
*/

#include "form3.h"
#include "minos.h"

/* static UBYTE *sparse = (UBYTE *)"sparse"; */
static UBYTE *tablebase = (UBYTE *)"tablebase";

/*
    #] Includes :
    #[ ClearTableTree :
*/

void ClearTableTree(TABLES T)
{
    COMPTREE *root;
    if ( T->boomlijst == 0 ) {
        T->MaxTreeSize = 125;
        T->boomlijst = (COMPTREE *)Malloc1(T->MaxTreeSize*sizeof(COMPTREE),
                "ClearTableTree");
    }
    root = T->boomlijst;
    T->numtree = 0;
    T->rootnum = 0;
    root->left = -1;
    root->right = -1;
    root->parent = -1;
    root->blnce = 0;
    root->value = -1;
```

```
    root->usage = 0;
}

/*
    #] ClearTableTree :
    #[ InsTableTree :

    int InsTableTree(TABLES T,WORD *,arglist)
        Searches for the element specified by the list of arguments.
        If found, it returns -(the offset in T->tablepointers)
        If not found, it will allocate a new element, balance the tree if
        necessary and return the number of the element in the boomlijst
        This number is always > 0, because we start from 1.
*/

int InsTableTree(TABLES T, WORD *tp)
{
    COMPTREE *boomlijst, *q, *p, *s;
    WORD *v1, *v2, *v3;
    int ip, iq, is;
    if ( T->numtree + 1 >= T->MaxTreeSize ) {
        if ( T->MaxTreeSize == 0 ) ClearTableTree(T);
        else {
            is = T->MaxTreeSize * 2;
            s  = (COMPTREE *)Malloc1(is*sizeof(COMPTREE),"InsTableTree");
            for ( ip = 0; ip < T->MaxTreeSize; ip++ ) { s[ip] = T->boomlijst[ip]; }
            if ( T->boomlijst ) M_free(T->boomlijst,"InsTableTree");
            T->boomlijst = s;
            T->MaxTreeSize = is;
        }
    }
    boomlijst = T->boomlijst;
    q = boomlijst + T->rootnum;
    if ( q->right == -1 ) { /* First element */
        T->numtree++;
        s = boomlijst+T->numtree;
        q->right = T->numtree;
        s->parent = T->rootnum;
        s->left = s->right = -1;
        s->blnce = 0;
        s->value = tp - T->tablepointers;
        s->usage = 0;
        return(T->numtree);
    }
    ip = q->right;
    while ( ip >= 0 ) {
        p = boomlijst + ip;
        v1 = T->tablepointers + p->value;
        v2 = tp; v3 = tp + T->numind;
        while ( *v1 == *v2 && v2 < v3 ) { v1++; v2++; }
        if ( v2 >= v3 ) return(-p->value);
        if ( *v1 > *v2 ) {
            iq = p->right;
            if ( iq >= 0 ) { ip = iq; }
            else {
                T->numtree++;
                is = T->numtree;
                p->right = is;
                s = boomlijst + is;
                s->parent = ip; s->left = s->right = -1;
                s->blnce = 0;   s->value = tp - T->tablepointers;
                s->usage = 0;
                p->blnce++;
                if ( p->blnce == 0 ) return(T->numtree);
                goto balance;
            }
        }
        else if ( *v1 < *v2 ) {
            iq = p->left;
            if ( iq >= 0 ) { ip = iq; }
            else {
                T->numtree++;
                is = T->numtree;
                s = boomlijst+is;
                p->left = is;
                s->parent = ip; s->left = s->right = -1;
                s->blnce = 0;   s->value = tp - T->tablepointers;
                s->usage = 0;
                p->blnce--;
                if ( p->blnce == 0 ) return(T->numtree);
                goto balance;
            }
        }
    }
    MesPrint("Serious problems in InsTableTree!\n");
    Terminate(-1);
    return(0);
```

```
balance:;
    for (;;) {
        p = boomlijst + ip;
        iq = p->parent;
        if ( iq == T->rootnum ) break;
        q = boomlijst + iq;
        if ( ip == q->left ) q->blnce--;
        else                 q->blnce++;
        if ( q->blnce == 0 ) break;
        if ( q->blnce == -2 ) {
            if ( p->blnce == -1 ) { /* single rotation */
                q->left = p->right;
                p->right = iq;
                p->parent = q->parent;
                q->parent = ip;
                if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
                else                                   boomlijst[p->parent].right = ip;
                if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
                q->blnce = p->blnce = 0;
            }
            else {  /* double rotation */
                s = boomlijst + is;
                q->left = s->right;
                p->right = s->left;
                s->right = iq;
                s->left = ip;
                if ( p->right >= 0 ) boomlijst[p->right].parent = ip;
                if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
                s->parent = q->parent;
                q->parent = is;
                p->parent = is;
                if ( boomlijst[s->parent].left == iq )
                       boomlijst[s->parent].left = is;
                else   boomlijst[s->parent].right = is;
                if ( s->blnce > 0 ) { q->blnce = s->blnce = 0; p->blnce = -1; }
                else if ( s->blnce < 0 ) { p->blnce = s->blnce = 0; q->blnce = 1; }
                else { p->blnce = s->blnce = q->blnce = 0; }
            }
            break;
        }
        else if ( q->blnce == 2 ) {
            if ( p->blnce == 1 ) { /* single rotation */
                q->right = p->left;
                p->left = iq;
                p->parent = q->parent;
                q->parent = ip;
                if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
                else                                   boomlijst[p->parent].right = ip;
                if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
                q->blnce = p->blnce = 0;
            }
            else {  /* double rotation */
                s = boomlijst + is;
                q->right = s->left;
                p->left = s->right;
                s->left = iq;
                s->right = ip;
                if ( p->left >= 0 ) boomlijst[p->left].parent = ip;
                if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
                s->parent = q->parent;
                q->parent = is;
                p->parent = is;
                if ( boomlijst[s->parent].left == iq ) boomlijst[s->parent].left = is;
                else                                   boomlijst[s->parent].right = is;
                if ( s->blnce < 0 ) { q->blnce = s->blnce = 0; p->blnce = 1; }
                else if ( s->blnce > 0 ) { p->blnce = s->blnce = 0; q->blnce = -1; }
                else { p->blnce = s->blnce = q->blnce = 0; }
            }
            break;
        }
        is = ip; ip = iq;
    }
    return(T->numtree);
}

/*
    #] InsTableTree :
    #[ RedoTableTree :

    To be used when a sparse table is trimmed due to a .store
    We rebuild the tree. In the future one could try to become faster
    at the cost of quite some complexity.
    We need to keep the first 'size' elements in the boomlijst.
    Kill all others and reconstruct the tree with the original ordering.
    This is very complicated! Because .store will either keep the whole
    table or remove the whole table we should not come here often.
    Hence we choose the slow solution for now.
```

```
*/

void RedoTableTree(TABLES T, int newsize)
{
    WORD *tp;
    int i;
    ClearTableTree(T);
    for ( i = 0, tp = T->tablepointers; i < newsize; i++ ) {
        InsTableTree(T,tp);
        tp += T->numind+TABLEEXTENSION;
    }
}

/*
    #] RedoTableTree :
    #[ FindTableTree :

    int FindTableTree(TABLES T,WORD *,arglist,int,inc)
        Searches for the element specified by the list of arguments.
        If found, it returns the offset in T->tablepointers
        If not found, it will return -1
        The list here is from the list of function arguments. Hence it
        has pairs of numbers -SNUMBER,index
        Actually inc says how many numbers there are and the above case is
        for inc = 2. For inc = 1 we have just a list of indices.
*/

int FindTableTree(TABLES T, WORD *tp, int inc)
{
    COMPTREE *boomlijst = T->boomlijst, *q = boomlijst + T->rootnum, *p;
    WORD *v1, *v2, *v3;
    int ip, iq;
    if ( q->right == -1 ) return(-1);
    ip = q->right;
    if ( inc > 1 ) tp += inc-1;
    while ( ip >= 0 ) {
        p = boomlijst + ip;
        v1 = T->tablepointers + p->value;
        v2 = tp; v3 = v1 + T->numind;
        while ( *v1 == *v2 && v1 < v3 ) { v1++; v2 += inc; }
        if ( v1 == v3 ) {
            p->usage++;
            return(p->value);
        }
        if ( *v1 > *v2 ) {
            iq = p->right;
            if ( iq >= 0 ) { ip = iq; }
            else return(-1);
        }
        else if ( *v1 < *v2 ) {
            iq = p->left;
            if ( iq >= 0 ) { ip = iq; }
            else return(-1);
        }
    }
    MesPrint("Serious problems in FindTableTree\n");
    Terminate(-1);
    return(-1);
}

/*
    #] FindTableTree :
    #[ DoTableExpansion :
*/

WORD DoTableExpansion(WORD *term, WORD level)
{
    GETIDENTITY
    WORD *t, *tstop, *stopper, *termout, *m, *mm, *tp, *r;
    TABLES T = 0;
    int i, j, num;
    AN.TeInFun = AR.TePos = 0;
    tstop = term + *term;
    stopper = tstop - ABS(tstop[-1]);
    t = term+1;
    while ( t < stopper ) {
        if ( *t != TABLEFUNCTION ) { t += t[1]; continue; }
        if ( t[FUNHEAD] > -FUNCTION ) { t += t[1]; continue; }
        T = functions[-t[FUNHEAD]-FUNCTION].tabl;
        if ( T == 0 ) { t += t[1]; continue; }
        if ( T->spare ) T = T->spare;
        if ( t[1] == FUNHEAD+2 && t[FUNHEAD+1] <= -FUNCTION ) break;
        if ( t[1] < FUNHEAD+1+2*T->numind ) { t += t[1]; continue; }
        for ( i = 0; i < T->numind; i++ ) {
            if ( t[FUNHEAD+1+2*i] != -SYMBOL ) break;
        }
        if ( i >= T->numind ) break;
```

```
        t += t[1];
    }
    if ( t >= stopper ) {
        MesPrint("Internal error: Missing table_ function");
        Terminate(-1);
    }
/*
    Table in T. Now collect the numbers of the symbols;
*/
    termout = AT.WorkPointer;
    if ( T->sparse ) {
        for ( i = 0; i < T->totind; i++ ) {
/*
            Loop over all table elements
*/
            m = termout + 1; mm = term + 1;
            while ( mm < t ) *m++ = *mm++;
            r = m;
            if ( t[1] == FUNHEAD+2 && t[FUNHEAD+1] <= -FUNCTION ) {
                *m++ = -t[FUNHEAD+1];
                *m++ = FUNHEAD+T->numind*2;
                for ( j = 2; j < FUNHEAD; j++ ) *m++ = 0;
                tp = T->tablepointers + (T->numind+TABLEEXTENSION)*i;
                for ( j = 0; j < T->numind; j++ ) {
                    *m++ = -SNUMBER; *m++ = *tp++;
                }
            }
            else {
                *m++ = SYMBOL; *m++ = 2+T->numind*2; mm = t + FUNHEAD+1;
                tp = T->tablepointers + (T->numind+TABLEEXTENSION)*i;
                for ( j = 0; j < T->numind; j++, mm += 2, tp++ ) {
                    if ( *tp != 0 ) { *m++ = mm[1]; *m++ = *tp; }
                }
                r[1] = m-r;
                if ( r[1] == 2 ) m = r;
            }
/*
            The next code replaces this old code

            *m++ = SUBEXPRESSION;
            *m++ = SUBEXPSIZE;
            *m++ = *tp;
            *m++ = 1;
            *m++ = T->bufnum;
            FILLSUB(m);
            mm = t + t[1];

            We had forgotten to take the parameters into account.
            Hence the subexpression prototype for wildcards was missed
            Now we slow things down a little bit, but we do not run
            any risks. There is still one problem. We have not checked
            that the prototype matches.
*/
            r = m;
            *m++ = -t[FUNHEAD];
            *m++ = t[1] - 1;
            for ( j = 2; j < FUNHEAD; j++ ) *m++ = t[j];
            tp = T->tablepointers + (T->numind+TABLEEXTENSION)*i;
            for ( j = 0; j < T->numind; j++ ) {
                *m++ = -SNUMBER; *m++ = *tp++;
            }
            tp = t + FUNHEAD + 1 + 2*T->numind;
            mm = t + t[1];
            while ( tp < mm ) *m++ = *tp++;
            r[1] = m-r;
/*
            From now on is old code
*/
            while ( mm < tstop ) *m++ = *mm++;
            *termout = m - termout;
            AT.WorkPointer = m;
            if ( Generator(BHEAD termout,level) ) {
                MesCall("DoTableExpand");
                return(-1);
            }
        }
    }
    else {
        for ( i = 0; i < T->totind; i++ ) {
#if TABLEEXTENSION == 2
            if ( T->tablepointers[i] < 0 ) continue;
#else
            if ( T->tablepointers[TABLEEXTENSION*i] < 0 ) continue;
#endif
            m = termout + 1; mm = term + 1;
            while ( mm < t ) *m++ = *mm++;
            r = m;
```

```
            if ( t[1] == FUNHEAD+2 && t[FUNHEAD+1] <= -FUNCTION ) {
                *m++ = -t[FUNHEAD+1];
                *m++ = FUNHEAD+T->numind*2;
                for ( j = 2; j < FUNHEAD; j++ ) *m++ = 0;
                tp = T->tablepointers + (T->numind+TABLEEXTENSION)*i;
                for ( j = 0; j < T->numind; j++ ) {
                    if ( j > 0 ) {
                        num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
                    }
                    else {
                        num = T->mm[j].mini + i / T->mm[j].size;
                    }
                    *m++ = -SNUMBER; *m++ = num;
                }
            }
            else {
                *m++ = SYMBOL; *m++ = 2+T->numind*2; mm = t + FUNHEAD+1;
                for ( j = 0; j < T->numind; j++, mm += 2 ) {
                    if ( j > 0 ) {
                        num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
                    }
                    else {
                        num = T->mm[j].mini + i / T->mm[j].size;
                    }
                    if ( num != 0 ) { *m++ = mm[1]; *m++ = num; }
                }
                r[1] = m-r;
                if ( r[1] == 2 ) m = r;
            }
/*
            The next code replaces this old code

            *m++ = SUBEXPRESSION;
            *m++ = SUBEXPSIZE;
            *m++ = *tp;
            *m++ = 1;
            *m++ = T->bufnum;
            FILLSUB(m);
            mm = t + t[1];

            We had forgotten to take the parameters into account.
            Hence the subexpression prototype for wildcards was missed
            Now we slow things down a little bit, but we do not run
            any risks. There is still one problem. We have not checked
            that the prototype matches.
*/
            r = m;
            *m++ = -t[FUNHEAD];
            *m++ = t[1] - 1;
            for ( j = 2; j < FUNHEAD; j++ ) *m++ = t[j];
            for ( j = 0; j < T->numind; j++ ) {
                if ( j > 0 ) {
                    num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
                }
                else {
                    num = T->mm[j].mini + i / T->mm[j].size;
                }
                *m++ = -SNUMBER; *m++ = num;
            }
            tp = t + FUNHEAD + 1 + 2*T->numind;
            mm = t + t[1];
            while ( tp < mm ) *m++ = *tp++;
            r[1] = m - r;
/*
            From now on is old code
*/
            while ( mm < tstop ) *m++ = *mm++;
            *termout = m - termout;
            AT.WorkPointer = m;
            if ( Generator(BHEAD termout,level) ) {
                MesCall("DoTableExpand");
                return(-1);
            }
        }
    }
    return(0);
}

/*
    #] DoTableExpansion :
    #[ TableBase :

    File with all the database related things.
    We have the routines for the generic database command
    TableBase,options;
    TB,options;
    Options are:
```

```
        Open "File.tbl";                  Open for R/W
        Create "File.tbl";                Create for write
        Load "File.tbl", tablename;       Loads stubs of table
        Load "File.tbl";                  Loads stubs of all tables
        Enter "File.tbl", tablename;      Loads whole table
        Enter "File.tbl";                 Loads all tables
        Audit "File.tbl", options;        Print list of contents
        Replace "File.tbl", tablename;    Saves a table (with overwrite)
        Replace "File.tbl", table element; Saves a table element   „
        Cleanup "File.tbl";               Makes tables contingent
        AddTo "File.tbl" tablename;       Add if not yet there.
        AddTo "File.tbl" table element;   Add if not yet there.
        Delete "File.tbl" tablename;
        Delete "File.tbl" table element;

        On/Off substitute;
        On/Off compress "File.tbl";
    id tbl_(f?,?a) = f(?a);
    When a tbl_ is used, automatically the corresponding element is compiled
    at the start of the next module.
    if TB,On,substitue [tablename], use of table RHS (if loaded)
    if TB,Off,substitue [tablename], use of tbl_(table,...);


    Still needed: Something like OverLoad to allow loading parts of a table
    from more than one file. Date stamps needed? In that case we need a touch
    command as well.

    If we put all our diagrams inside, we have to go outside the concept
    of tables.

    #] TableBase :
    #[ CoTableBase :

    To be followed by ,subkey
*/
static KEYWORD tboptions[] = {
     {"addto",           (TFUN)CoTBaddto,      0,    PARTEST}
    ,{"audit",           (TFUN)CoTBaudit,      0,    PARTEST}
    ,{"cleanup",         (TFUN)CoTBcleanup,    0,    PARTEST}
    ,{"create",          (TFUN)CoTBcreate,     0,    PARTEST}
    ,{"enter",           (TFUN)CoTBenter,      0,    PARTEST}
    ,{"help",            (TFUN)CoTBhelp,       0,    PARTEST}
    ,{"load",            (TFUN)CoTBload,       0,    PARTEST}
    ,{"off",             (TFUN)CoTBoff,        0,    PARTEST}
    ,{"on",              (TFUN)CoTBon,         0,    PARTEST}
    ,{"open",            (TFUN)CoTBopen,       0,    PARTEST}
    ,{"replace",         (TFUN)CoTBreplace,    0,    PARTEST}
    ,{"use",             (TFUN)CoTBuse,        0,    PARTEST}
};

static UBYTE *tablebasename = 0;

int CoTableBase(UBYTE *s)
{
    UBYTE *option, c, *t;
    int i,optlistsize = sizeof(tboptions)/sizeof(KEYWORD), error = 0;
    while ( *s == ' ' ) s++;
    if ( *s != '"' ) {
        if ( ( tolower(*s) == 'h' ) && ( tolower(s[1]) == 'e' )
         && ( tolower(s[2]) == 'l' ) && ( tolower(s[3]) == 'p' )
         && ( FG.cTable[s[4]] > 1 ) ) {
            CoTBhelp(s);
            return(0);
        }
proper:;
        MesPrint("&Proper syntax: TableBase \"filename\" options");
        return(1);
    }
    s++; tablebasename = s;
    while ( *s && *s != '"' ) s++;
    if ( *s != '"' ) goto proper;
    t = s; s++; *t = 0;
    while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
    option = s;
    while ( FG.cTable[*s] == 0 ) s++;
    c = *s; *s = 0;
    for ( i = 0; i < optlistsize; i++ ) {
        if ( StrICmp(option,(UBYTE *)(tboptions[i].name)) == 0 ) {
            *s = c;
            while ( *s == ',' ) s++;
            error = (tboptions[i].func)(s);
            *t = '"';
            return(error);
        }
    }
    MesPrint("&Unrecognized option %s in TableBase statement",option);
```

```
    return(1);
}

/*
    #] CoTableBase :
    #[ FlipTable :

    Flips the table between use as 'stub' and regular use
*/

int FlipTable(FUNCTIONS f, int type)
{
    TABLES T, TT;
    T = f->tabl;
    if ( ( TT = T->spare ) == 0 ) {
        MesPrint("Error: trying to change mode on a table that has no tablebase");
        return(-1);
    }
    if ( TT->mode == type ) f->tabl = TT;
    return(0);
}

/*
    #] FlipTable :
    #[ SpareTable :

    Creates a spare element for a table. This is used in the table bases.
    It is a (thus far) empty copy of the TT table.
    By using FlipTable we can switch between them and alter which version of
    a table we will be using. Note that this also causes some extra work in the
    ResetVariables and the Globalize routines.
*/

int SpareTable(TABLES TT)
{
    TABLES T;
    T = (TABLES)Malloc1(sizeof(struct TaBlEs),"table");
    T->defined = T->mdefined = 0; T->sparse = TT->sparse; T->mm = 0; T->flags = 0;
    T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
    T->boomlijst = 0;
    T->strict = TT->strict;
    T->bounds = TT->bounds;
    T->bufnum = inicbufs();
    T->argtail = TT->argtail;
    T->spare = TT;
    T->bufferssize = 8;
    T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"SpareTable buffers");
    T->buffersfill = 0;
    T->buffers[T->buffersfill++] = T->bufnum;
    T->mode = 0;
    T->numind = TT->numind;
    T->totind = 0;
    T->prototype = TT->prototype;
    T->pattern = TT->pattern;
    T->tablepointers = 0;
    T->reserved = 0;
    T->tablenum = 0;
    T->numdummies = 0;
    T->mm = (MINMAX *)Malloc1(T->numind*sizeof(MINMAX),"table dimensions");
    T->flags = (WORD *)Malloc1(T->numind*sizeof(WORD),"table flags");
    ClearTableTree(T);
    TT->spare = T;
    TT->mode = 1;
    return(0);
}

/*
    #] SpareTable :
    #[ FindTB :

    Looks for a tablebase with the given name in the active tablebases.
*/

DBASE *FindTB(UBYTE *name)
{
    DBASE *d;
    int i;
    for ( i = 0; i < NumTableBases; i++ ) {
        d = tablebases+i;
        if ( d->name && ( StrCmp(name,(UBYTE *)(d->name)) == 0 ) ) { return(d); }
    }
    return(0);
}

/*
    #] FindTB :
    #[ CoTBcreate :
```

```c

    Creates a new tablebase.
    Error is when there is already an active tablebase by this name.
    If a file with the given name exists already, but it does not correspond
    to an active table base, its contents will be lost.
    Note that tablebasename is a static variable, defined in CoTableBase
*/

int CoTBcreate(UBYTE *s)
{
    DUMMYUSE(s);
    if ( FindTB(tablebasename) != 0 ) {
        MesPrint("&There is already an open TableBase with the name %s",tablebasename);
        return(-1);
    }
    NewDbase((char *)tablebasename,0);
    return(0);
}

/*
    #] CoTBcreate :
    #[ CoTBopen :
*/

int CoTBopen(UBYTE *s)
{
    DBASE *d;
    DUMMYUSE(s);
    if ( ( d = FindTB(tablebasename) ) != 0 ) {
        MesPrint("&There is already an open TableBase with the name %s",tablebasename);
        return(-1);
    }
    d = GetDbase((char *)tablebasename);
    if ( CheckTableDeclarations(d) ) return(-1);
    return(0);
}

/*
    #] CoTBopen :
    #[ CoTBaddto :
*/

int CoTBaddto(UBYTE *s)
{
    GETIDENTITY
    DBASE *d;
    UBYTE *tablename, c, *t, elementstring[ELEMENTSIZE+20], *ss, *es;
    WORD type, funnum, lbrac, first, num, *expr, *w;
    TABLES T = 0;
    MLONG basenumber;
    LONG x;
    int i, j, error = 0, sum;
    if ( ( d = FindTB(tablebasename) ) == 0 ) {
        MesPrint("&No open tablebase with the name %s",tablebasename);
        return(-1);
    }
    AO.DollarOutSizeBuffer = 32;
    AO.DollarOutBuffer = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,
                            "TableOutBuffer");
/*
    Now loop through the names and start adding
*/
    while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    while ( *s ) {
        tablename = s;
        if ( ( s = SkipAName(s) ) == 0 ) goto tableabort;
        c = *s; *s = 0;
        if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
            || ( T = functions[funnum].tabl ) == 0 ) {
            MesPrint("&%s should be a previously declared table",tablename);
            *s = c; goto tableabort;
        }
        if ( T->sparse == 0 ) {
            MesPrint("&%s should be a sparse table",tablename);
            *s = c; goto tableabort;
        }
        basenumber = AddTableName(d,(char *)tablename,T);
        if ( T->spare && ( T->mode ==  1 ) ) T = T->spare;
        if ( basenumber < 0 ) basenumber = -basenumber;
        else if ( basenumber == 0 ) { *s = c; goto tableabort; }
        *s = c;
        if ( *s == '(' ) { /* Addition of single element */
            s++; es = s;
            for ( i = 0, w = AT.WorkPointer; i < T->numind; i++ ) {
              ParseSignedNumber(x,s);
              if ( FG.cTable[s[-1]] != 1 || ( *s != ',' && *s != ')' ) ) {
                MesPrint("&Table arguments in TableBase addto statement should be numbers");
```

```
                return(1);
              }
              *w++ = x;
              if ( *s == ')' ) break;
              s++;
            }
            if ( *s != ')' || i < ( T->numind - 1 ) ) {
              MesPrint("&Incorrect number of table arguments in TableBase addto statement. Should be %d"
                ,T->numind);
              error = 1;
            }
            c = *s; *s = 0;
            i = FindTableTree(T,AT.WorkPointer,1);
            if ( i < 0 ) {
                MesPrint("&Element %s has not been defined",es);
                error = 1;
                *s++ = c;
            }
            else if ( ExistsObject(d,basenumber,(char *)es) ) {}
            else {
              int dict = AO.CurrentDictionary;
              AO.CurrentDictionary = 0;
              sum = i + T->numind;
/*
              See also commentary below
*/
              AO.DollarInOutBuffer = 1;
              AO.PrintType = 1;
              ss = AO.DollarOutBuffer;
              *ss = 0;
              AO.OutInBuffer = 1;
#if ( TABLEEXTENSION == 2 )
              expr = cbuf[T->bufnum].rhs[T->tablepointers[sum]];
#else
              expr = cbuf[T->tablepointers[sum+1]].rhs[T->tablepointers[sum]];
#endif
              lbrac = 0; first = 0;
              while ( *expr ) {
                if ( WriteTerm(expr,&lbrac,first,PRINTON,0) ) {
                        error = 1; break;
                }
                expr += *expr;
              }
              AO.OutInBuffer = 0;
              AddObject(d,basenumber,(char *)es,(char *)(AO.DollarOutBuffer));
              *s++ = c;
              AO.CurrentDictionary = dict;
            }
        }
        else {
/*
        Now we have to start looping through all defined elements of this table.
        We have to construct the arguments in text format.
*/
          for ( i = 0; i < T->totind; i++ ) {
#if ( TABLEEXTENSION == 2 )
            if ( !T->sparse && T->tablepointers[i] < 0 ) continue;
#else
            if ( !T->sparse && T->tablepointers[TABLEEXTENSION*i] < 0 ) continue;
#endif
            sum = i * ( T->numind + TABLEEXTENSION );
            t = elementstring;
            for ( j = 0; j < T->numind; j++, sum++ ) {
                if ( j > 0 ) *t++ = ',';
                num = T->tablepointers[sum];
                t = NumCopy(num,t);
                if ( ( t - elementstring ) >= ELEMENTSIZE ) {
                    MesPrint("&Table element specification takes more than %ld characters and cannot be handled",
                        (MLONG)ELEMENTSIZE);
                    goto tableabort;
                }
            }
            if ( ExistsObject(d,basenumber,(char *)elementstring) ) { continue; }
/*
            We have the number in basenumber and the element in elementstring.
            Now we need the rhs. We can use the code from WriteDollarToBuffer.
            Main complication: in the table compiler buffer there can be
            brackets. The dollars do not have those......
*/
            AO.DollarInOutBuffer = 1;
            AO.PrintType = 1;
            ss = AO.DollarOutBuffer;
            *ss = 0;
            AO.OutInBuffer = 1;
#if ( TABLEEXTENSION == 2 )
```

```
            expr = cbuf[T->bufnum].rhs[T->tablepointers[sum]];
#else
            expr = cbuf[T->tablepointers[sum+1]].rhs[T->tablepointers[sum]];
#endif
            lbrac = 0; first = 0;
            while ( *expr ) {
                if ( WriteTerm(expr,&lbrac,first,PRINTON,0) ) {
                    error = 1; break;
                }
                expr += *expr;
            }
            AO.OutInBuffer = 0;
            AddObject(d,basenumber,(char *)elementstring,(char *)(AO.DollarOutBuffer));
          }
        }
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    }
    if ( WriteIniInfo(d) ) goto tableabort;
    M_free(AO.DollarOutBuffer,"DollarOutBuffer");
    AO.DollarOutBuffer = 0;
    AO.DollarOutSizeBuffer = 0;
    return(error);
tableabort:;
    M_free(AO.DollarOutBuffer,"DollarOutBuffer");
    AO.DollarOutBuffer = 0;
    AO.DollarOutSizeBuffer = 0;
    AO.OutInBuffer = 0;
    return(1);
}

/*
    #] CoTBaddto :
    #[ CoTBenter :

    Loads the elements of the tables specified into memory and sends them
    one by one to the compiler as Fill statements.
*/

int CoTBenter(UBYTE *s)
{
    DBASE *d;
    MLONG basenumber;
    UBYTE *arguments, *rhs, *buffer, *t, *u, c, *tablename;
    LONG size;
    int i, j, error = 0, error1 = 0, printall = 0;
    TABLES T = 0;
    WORD type, funnum;
    int dict = AO.CurrentDictionary;
    AO.CurrentDictionary = 0;
    if ( ( d = FindTB(tablebasename) ) == 0 ) {
        MesPrint("&No open tablebase with the name %s",tablebasename);
        error = -1;
        goto Endofall;
    }
    while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    if ( *s == '!' ) { printall = 1; s++; }
    while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    if ( *s ) {
      while ( *s ) {
        tablename = s;
        if ( ( s = SkipAName(s) ) == 0 ) { error = 1; goto Endofall; }
        c = *s; *s = 0;
        if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
            || ( T = functions[funnum].tabl ) == 0 ) {
            MesPrint("&%s should be a previously declared table",tablename);
            basenumber = 0;
        }
        else if ( T->sparse == 0 ) {
            MesPrint("&%s should be a sparse table",tablename);
            basenumber = 0;
        }
        else { basenumber = GetTableName(d,(char *)tablename); }
        if ( T->spare == 0 ) { SpareTable(T); }
        if ( basenumber > 0 ) {
            for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
                for ( j = 0; j < NUMOBJECTS; j++ ) {
                    if ( basenumber != d->iblocks[i]->objects[j].tablenumber )
                        continue;
                    arguments = (UBYTE *)(d->iblocks[i]->objects[j].element);
                    rhs = (UBYTE *)ReadObject(d,basenumber,(char *)arguments);
                    if ( printall ) {
                        if ( rhs ) {
                            MesPrint("%s(%s) = %s",tablename,arguments,rhs);
                        }
                        else {
                            MesPrint("%s(%s) = 0",tablename,arguments);
                        }
```

```
                    }
                    if ( rhs ) {
                        u = rhs; while ( *u ) u++;
                        size = u-rhs;
                        u = arguments; while ( *u ) u++;
                        size += u-arguments;
                        u = tablename; while ( *u ) u++;
                        size += u-tablename;
                        size += 6;
                        buffer  = (UBYTE *)Malloc1(size,"TableBase copy");
                        t = tablename; u = buffer;
                        while ( *t ) *u++ = *t++;
                        *u++ = '(';
                        t = arguments;
                        while ( *t ) *u++ = *t++;
                        *u++ = ')'; *u++ = '=';
                        t = rhs;
                        while ( *t ) *u++ = *t++;
                        if ( t == rhs ) *u++ = '0';
                        *u++ = 0; *u = 0;
                        M_free(rhs,"rhs in TBenter");

                        error1 = CoFill(buffer);

                        if ( error1 < 0 ) goto Endofall;
                        if ( error1 != 0 ) error = error1;
                        M_free(buffer,"TableBase copy");
                    }
                }
            }
        }
        *s = c;
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
      }
    }
    else {
      s = (UBYTE *)(d->tablenames); basenumber = 0;
      while ( *s ) {
        basenumber++;
        tablename = s; while ( *s ) s++; s++;
        while ( *s ) s++; s++;
        if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
            || ( T = functions[funnum].tabl ) == 0 ) {
            MesPrint("&%s should be a previously declared table",tablename);
        }
        else if ( T->sparse == 0 ) {
            MesPrint("&%s should be a sparse table",tablename);
        }
        if ( T->spare == 0 ) { SpareTable(T); }
        for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
            for ( j = 0; j < NUMOBJECTS; j++ ) {
                if ( d->iblocks[i]->objects[j].tablenumber == basenumber ) {
                    arguments = (UBYTE *)(d->iblocks[i]->objects[j].element);
                    rhs = (UBYTE *)ReadObject(d,basenumber,(char *)arguments);
                    if ( printall ) {
                        if ( rhs ) {
                            MesPrint("%s%s = %s",tablename,arguments,rhs);
                        }
                        else {
                            MesPrint("%s%s = 0",tablename,arguments);
                        }
                    }
                    if ( rhs ) {
                        u = rhs; while ( *u ) u++;
                        size = u-rhs;
                        u = arguments; while ( *u ) u++;
                        size += u-arguments;
                        u = tablename; while ( *u ) u++;
                        size += u-tablename;
                        size += 6;
                        buffer  = (UBYTE *)Malloc1(size,"TableBase copy");
                        t = tablename; u = buffer;
                        while ( *t ) *u++ = *t++;
                        *u++ = '(';
                        t = arguments;
                        while ( *t ) *u++ = *t++;
                        *u++ = ')'; *u++ = '=';
                        t = rhs;
                        while ( *t ) *u++ = *t++;
                        if ( t == rhs ) *u++ = '0';
                        *u++ = 0; *u = 0;
                        M_free(rhs,"rhs in TBenter");

                        error1 = CoFill(buffer);

                        if ( error1 < 0 ) goto Endofall;
                        if ( error1 != 0 ) error = error1;
```

```
                        M_free(buffer,"TableBase copy");
                    }
                }
            }
        }
      }
    }
Endofall:;
    AO.CurrentDictionary = dict;
    return(error);
}

/*
    #] CoTBenter :
    #[ CoTestUse :

    Possibly to be followed by names of tables.
    We make an array of TABLES structs to be tested in AC.usedtables.
    Note: only sparse tables are allowed.
    No arguments means all tables.
*/

int CoTestUse(UBYTE *s)
{
    GETIDENTITY
    UBYTE *tablename, c;
    WORD type, funnum, *w;
    TABLES T;
    int error = 0;
    w = AT.WorkPointer;
    *w++ = TYPETESTUSE; *w++ = 2;
    while ( *s ) {
        tablename = s;
        if ( ( s = SkipAName(s) ) == 0 ) return(1);
        c = *s; *s = 0;
        if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
            || ( T = functions[funnum].tabl ) == 0 ) {
            MesPrint("&%s should be a previously declared table",tablename);
            error = 1;
        }
        else if ( T->sparse == 0 ) {
            MesPrint("&%s should be a sparse table",tablename);
            error = 1;
        }
        *w++ = funnum + FUNCTION;
        *s = c;
        while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
    }
    AT.WorkPointer[1] = w - AT.WorkPointer;
/*
    if ( AT.WorkPointer[1] > 2 ) {
        AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
    }
*/
    AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
    return(error);
}

/*
    #] CoTestUse :
    #[ CheckTableDeclarations :

        Checks that all tables in a tablebase  have identical properties to
        possible previous declarations. If they have not been declared
        before, they are declared here.
*/

int CheckTableDeclarations(DBASE *d)
{
    WORD type, funnum;
    UBYTE *s, *ss, *t, *command = 0;
    int k, error = 0, error1, i;
    TABLES T;
    LONG commandsize = 0;

    s = (UBYTE *)(d->tablenames);
    for ( k = 0; k < d->topnumber; k++ ) {
        if ( GetVar(s,&type,&funnum,ANYTYPE,NOAUTO) == NAMENOTFOUND ) {
/*
            We have to declare the table
*/
            ss = s; i = 0; while ( *ss ) { ss++; i++; } /* name */
            ss++; while ( *ss ) { ss++; i++; } /* tail */
            if ( commandsize == 0 ) {
                commandsize = i + 15;
                if ( commandsize < 100 ) commandsize = 100;
            }
```

```
            if ( (i+11) > commandsize ) {
                if ( command ) { M_free(command,"table command"); command = 0; }
                commandsize = i+10;
            }
            if ( command == 0 ) {
                command = (UBYTE *)Malloc1(commandsize,"table command");
            }
            t = command; ss = tablebase; while ( *ss ) *t++ = *ss++;
            *t++ = ','; while ( *s ) *t++ = *s++;
            s++; while ( *s ) *t++ = *s++;
            *t++ = ')'; *t = 0; s++;
            error1 = DoTable(command,1);
            if ( error1 ) error = error1;
        }
        else if ( ( type != CFUNCTION )
             || ( ( T = functions[funnum].tabl ) == 0 )
             || ( T->sparse == 0 ) ) {
            MesPrint("&%s has been declared previously, but not as a sparse table.",s);
            error = 1;
            while ( *s ) s++; s++; while ( *s ) s++; s++;
        }
        else {
/*
            Test dimension and argtail. There should be an exact match.
            We are not going to rename arguments when reading the elements.
*/
            ss = s;
            while ( *s ) s++; s++;
            if ( StrCmp(s,T->argtail) ) {
                MesPrint("&Declaration of table %s in %s different from previous declaration",ss,d->name);
                error = 1;
            }
            while ( *s ) s++; s++;
        }
    }
    if ( command ) { M_free(command,"table command"); }
    return(error);
}

/*
    #] CheckTableDeclarations :
    #[ CoTBload :

        Loads the table stubbs of the specified tables in the indicated
        tablebase. Syntax:
        TableBase "tablebasename.tbl" load [tablename(s)];
        If no tables are specified all tables are taken.
*/

int CoTBload(UBYTE *ss)
{
    DBASE *d;
    UBYTE *s, *name, *t, *r, *command, *arguments, *tail;
    LONG commandsize;
    int num, cs, es, ns, ts, i, j, error = 0, error1;
    if ( ( d = FindTB(tablebasename) ) == 0 ) {
        MesPrint("&No open tablebase with the name %s",tablebasename);
        return(-1);
    }
    commandsize = 120;
    command = (UBYTE *)Malloc1(commandsize,"Fill command");
    AC.vetofilling = 1;
    if ( *ss ) {
      while ( *ss == ',' || *ss == ' ' || *ss == '\t' ) ss++;
      while ( *ss ) {
        name = ss; ss = SkipAName(ss); *ss = 0;
        s = (UBYTE *)(d->tablenames);
        num = 0; ns = 0;
        while ( *s ) {
            num++;
            if ( StrCmp(s,name) ) {
                while ( *s ) s++; s++; while ( *s ) s++; s++; num++;
                continue;
            }
            name = s; while ( *s ) s++; ns = s-name; s++;
            tail = s; while ( *s ) s++; ts = s-tail; s++;
            tail++; while ( FG.cTable[*tail] == 1 ) tail++;
/*
            Go through all elements
*/
            for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
                for ( j = 0; j < NUMOBJECTS; j++ ) {
                    if ( d->iblocks[i]->objects[j].tablenumber == num ) {
                        t = arguments = (UBYTE *)(d->iblocks[i]->objects[j].element);
                        while ( *t ) t++; es = t - arguments;
                        cs = 2*es + 2*ns + ts + 10;
```

```c
                        if ( cs > commandsize ) {
                            commandsize = 2*cs;
                            if ( command ) M_free(command,"Fill command");
                            command = (UBYTE *)Malloc1(commandsize,"Fill command");
                        }
                        r = command; t = name; while ( *t ) *r++ = *t++;
                        *r++ = '('; t = arguments; while ( *t ) *r++ = *t++;
                        *r++ = ')'; *r++ = '='; *r++ = 't'; *r++ = 'b'; *r++ = 'l';
                        *r++ = '_'; *r++ = '('; t = name; while ( *t ) *r++ = *t++;
                        *r++ = ','; t = arguments; while ( *t ) *r++ = *t++;
                        t = tail; while ( *t ) {
                            if ( *t == '?' && r[-1] != ',' ) {
                                t++;
                                if ( FG.cTable[*t] == 0 || *t == '$' || *t == '[' ) {
                                    t = SkipAName(t);
                                    if ( *t == '[' ) {
                                        SKIPBRA1(t);
                                    }
                                }
                                else if ( *t == '{' ) {
                                    SKIPBRA2(t);
                                }
                                else if ( *t ) { *r++ = *t++; continue; }
                            }
                            else *r++ = *t++;
                        }
                        *r++ = ')'; *r = 0;
/*
                        Still to do: replacemode or no replacemode?
*/
                        AC.vetotablebasefill = 1;
                        error1 = CoFill(command);
                        AC.vetotablebasefill = 0;
                        if ( error1 < 0 ) goto finishup;
                        if ( error1 != 0 ) error = error1;
                    }
                }
            }
            break;
        }
        while ( *ss == ',' || *ss == ' ' || *ss == '\t' ) ss++;
      }
    }
    else {  /* do all of them */
        s = (UBYTE *)(d->tablenames);
        num = 0; ns = 0;
        while ( *s ) {
            num++;
            name = s; while ( *s ) s++; ns = s-name; s++;
            tail = s; while ( *s ) s++; ts = s-tail; s++;
            tail++; while ( FG.cTable[*tail] == 1 ) tail++;
/*
            Go through all elements
*/
            for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
                for ( j = 0; j < NUMOBJECTS; j++ ) {
                    if ( d->iblocks[i]->objects[j].tablenumber == num ) {
                        t = arguments = (UBYTE *)(d->iblocks[i]->objects[j].element);
                        while ( *t ) t++; es = t - arguments;
                        cs = 2*es + 2*ns + ts + 10;
                        if ( cs > commandsize ) {
                            commandsize = 2*cs;
                            if ( command ) M_free(command,"Fill command");
                            command = (UBYTE *)Malloc1(commandsize,"Fill command");
                        }
                        r = command; t = name; while ( *t ) *r++ = *t++;
                        *r++ = '('; t = arguments; while ( *t ) *r++ = *t++;
                        *r++ = ')'; *r++ = '='; *r++ = 't'; *r++ = 'b'; *r++ = 'l';
                        *r++ = '_'; *r++ = '('; t = name; while ( *t ) *r++ = *t++;
                        *r++ = ','; t = arguments; while ( *t ) *r++ = *t++;
                        t = tail; while ( *t ) {
                            if ( *t == '?' && r[-1] != ',' ) {
                                t++;
                                if ( FG.cTable[*t] == 0 || *t == '$' || *t == '[' ) {
                                    t = SkipAName(t);
                                    if ( *t == '[' ) {
                                        SKIPBRA1(t);
                                    }
                                }
                                else if ( *t == '{' ) {
                                    SKIPBRA2(t);
                                }
                                else if ( *t ) { *r++ = *t++; continue; }
                            }
                            else *r++ = *t++;
                        }
                        *r++ = ')'; *r = 0;
```

```
/*
                        Still to do: replacemode or no replacemode?
*/
                        AC.vetotablebasefill = 1;
                        error1 = CoFill(command);
                        AC.vetotablebasefill = 0;
                        if ( error1 < 0 ) goto finishup;
                        if ( error1 != 0 ) error = error1;
                    }
                }
            }
        }
    }
finishup:;
    AC.vetofilling = 0;
    if ( command ) M_free(command,"Fill command");
    return(error);
}

/*
    #] CoTBload :
    #[ TestUse :

    Look for tbl_(tablename,arguments)
    if tablename is encountered, check first whether the element is in
    use already. If not, check in the tables in AC.usedtables.
    If the element is not there, add it to AC.usedtables.


    We need the arguments of TestUse to see for which tables it is to be done
*/

WORD TestUse(WORD *term, WORD level)
{
    WORD *tstop, *t, *m, *tstart, tabnum;
    WORD *funs, numfuns, error = 0;
    TABLES T;
    LONG i;
    CBUF *C = cbuf+AM.rbufnum;
    int isp;

    numfuns = C->lhs[level][1] - 2;
    funs = C->lhs[level] + 2;
    GETSTOP(term,tstop);
    t = term+1;
    while ( t < tstop ) {
        if ( *t != TABLESTUB ) { t += t[1]; continue; }
        tstart = t;
        m = t + FUNHEAD;
        t += t[1];
        if ( *m >= -FUNCTION ) continue;
        tabnum = -*m;
        if ( ( T = functions[tabnum-FUNCTION].tabl ) == 0 ) continue;
        if ( T->sparse == 0 ) continue;
/*
        Check whether we have to test this one
*/
        if ( numfuns > 0 ) {
            for ( i = 0; i < numfuns; i++ ) {
                if ( tabnum == funs[i] ) break;
            }
            if ( i >= numfuns && numfuns > 0 ) continue;
        }
/*
        Test whether the element has been defined already.
            If not, mark it as used.
        Note: we only allow sparse tables (for now)
*/
        m++;
        for ( i = 0; i < T->numind; i++, m += 2 ) {
            if ( m >= t || *m != -SNUMBER ) break;
        }
        if ( ( i == T->numind ) &&
         ( ( isp = FindTableTree(T,tstart+FUNHEAD+1,2) ) >= 0 ) ) {
            if ( ( T->tablepointers[isp+T->numind+4] & ELEMENTLOADED ) == 0 ) {
                    T->tablepointers[isp+T->numind+4] |= ELEMENTUSED;
            }
        }
        else {
            MesPrint("TestUse: Encountered a table element inside tbl_ that does not correspond to a
tablebase element");
            error = -1;
        }
    }
    return(error);
}

```

```c
/*
    #] TestUse :
    #[ CoTBaudit :
*/

int CoTBaudit(UBYTE *s)
{
    DBASE *d;
    UBYTE *name, *tail;
    int i, j, error = 0, num;

    if ( ( d = FindTB(tablebasename) ) == 0 ) {
        MesPrint("&No open tablebase with the name %s",tablebasename);
        return(-1);
    }
    while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    while ( *s ) {
/*
        Get the options here
        They will mainly involve the sorting of the output.
*/
        s++;
    }
    s = (UBYTE *)(d->tablenames); num = 0;
    while ( *s ) {
        num++;
        name = s; while ( *s ) s++; s++;
        tail = s; while ( *s ) s++; s++;
        MesPrint("Table,sparse,%s%s)",name,tail);
        for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
            for ( j = 0; j < NUMOBJECTS; j++ ) {
                if ( d->iblocks[i]->objects[j].tablenumber == num ) {
                    MesPrint("    %s(%s)",name,d->iblocks[i]->objects[j].element);
                }
            }
        }
    }
    return(error);
}

/*
    #] CoTBaudit :
    #[ CoTBon :
*/

int CoTBon(UBYTE *s)
{
    DBASE *d;
    UBYTE *ss, c;
    int error = 0;
    if ( ( d = FindTB(tablebasename) ) == 0 ) {
        MesPrint("&No open tablebase with the name %s",tablebasename);
        return(-1);
    }
    while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    while ( *s ) {
        ss = SkipAName(s);
        c = *ss; *ss = 0;
        if ( StrICmp(s,(UBYTE *)("compress")) == 0 ) {
            d->mode &= ~NOCOMPRESS;
        }
        else {
            MesPrint("&subkey %s not defined in TableBase On statement");
            error = 1;
        }
        *ss = c; s = ss;
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    }
    return(error);
}

/*
    #] CoTBon :
    #[ CoTBoff :
*/

int CoTBoff(UBYTE *s)
{
    DBASE *d;
    UBYTE *ss, c;
    int error = 0;
    if ( ( d = FindTB(tablebasename) ) == 0 ) {
        MesPrint("&No open tablebase with the name %s",tablebasename);
        return(-1);
    }
    while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    while ( *s ) {
```

```
        ss = SkipAName(s);
        c = *ss; *ss = 0;
        if ( StrICmp(s,(UBYTE *)("compress")) == 0 ) {
            d->mode |= NOCOMPRESS;
        }
        else {
            MesPrint("&subkey %s not defined in TableBase Off statement");
            error = 1;
        }
        *ss = c; s = ss;
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    }
    return(error);
}

/*
    #] CoTBoff :
    #[ CoTBcleanup :
*/

int CoTBcleanup(UBYTE *s)
{
    DUMMYUSE(s);
    MesPrint("&TableBase Cleanup statement not yet implemented");
    return(1);
}

/*
    #] CoTBcleanup :
    #[ CoTBreplace :
*/

int CoTBreplace(UBYTE *s)
{
    DUMMYUSE(s);
    MesPrint("&TableBase Replace statement not yet implemented");
    return(1);
}

/*
    #] CoTBreplace :
    #[ CoTBuse :

    Here the actual table use as determined in TestUse causes the needed
    table elements to be loaded
*/

int CoTBuse(UBYTE *s)
{
    GETIDENTITY
    DBASE *d;
    MLONG basenumber;
    UBYTE *arguments, *rhs, *buffer, *t, *u, c, *tablename, *p;
    LONG size, sum, x;
    int i, j, error = 0, error1 = 0, k;
    TABLES T = 0;
    WORD type, funnum, mode, *w;
    if ( ( d = FindTB(tablebasename) ) == 0 ) {
        MesPrint("&No open tablebase with the name %s",tablebasename);
        return(-1);
    }
    while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
    if ( *s ) {
      while ( *s ) {
        tablename = s;
        if ( ( s = SkipAName(s) ) == 0 ) return(1);
        c = *s; *s = 0;
        if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
            || ( T = functions[funnum].tabl ) == 0 ) {
            MesPrint("&%s should be a previously declared table",tablename);
            basenumber = 0;
        }
        else if ( T->sparse == 0 ) {
            MesPrint("&%s should be a sparse table",tablename);
            basenumber = 0;
        }
        else { basenumber = GetTableName(d,(char *)tablename); }
/*      if ( T->spare == 0 ) { SpareTable(T); } */
        if ( basenumber > 0 ) {
            for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
                for ( j = 0; j < NUMOBJECTS; j++ ) {
                    if ( d->iblocks[i]->objects[j].tablenumber != basenumber ) continue;
                    arguments = p = (UBYTE *)(d->iblocks[i]->objects[j].element);
/*
                    Now translate the arguments and see whether we need
                    this one....
*/
```

```
                    for ( k = 0, w = AT.WorkPointer; k < T->numind; k++ ) {
                        ParseSignedNumber(x,p);
                        *w++ = x; p++;
                    }
                    sum = FindTableTree(T,AT.WorkPointer,1);
                    if ( sum < 0 ) {
                        MesPrint("Table %s in tablebase %s has not been loaded properly"
                                ,tablename,tablebasename);
                        error = 1;
                        continue;
                    }
                    sum += T->numind + 4;
                    mode = T->tablepointers[sum];
                    if ( ( mode & ELEMENTLOADED ) == ELEMENTLOADED ) {
                        T->tablepointers[sum] &= ~ELEMENTUSED;
                        continue;
                    }
                    if ( ( mode & ELEMENTUSED ) == 0 ) continue;
/*
                    We need this one!
*/
                    rhs = (UBYTE *)ReadijObject(d,i,j,(char *)arguments);
                    if ( rhs ) {
                        u = rhs; while ( *u ) u++;
                        size = u-rhs;
                        u = arguments; while ( *u ) u++;
                        size += u-arguments;
                        u = tablename; while ( *u ) u++;
                        size += u-tablename;
                        size += 6;
                        buffer  = (UBYTE *)Malloc1(size,"TableBase copy");
                        t = tablename; u = buffer;
                        while ( *t ) *u++ = *t++;
                        *u++ = '(';
                        t = arguments;
                        while ( *t ) *u++ = *t++;
                        *u++ = ')'; *u++ = '=';
                        t = rhs;
                        while ( *t ) *u++ = *t++;
                        if ( t == rhs ) { *u++ = '0'; }
                        *u++ = 0; *u = 0;
                        M_free(rhs,"rhs in TBuse xxx");

                        error1 = CoFill(buffer);

                        if ( error1 < 0 ) { return(error); }
                        if ( error1 != 0 ) error = error1;
                        M_free(buffer,"TableBase copy");
                    }
                    T->tablepointers[sum] &= ~ELEMENTUSED;
                    T->tablepointers[sum] |= ELEMENTLOADED;
                }
            }
        }
        *s = c;
        while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
      }
    }
    else {
      s = (UBYTE *)(d->tablenames); basenumber = 0;
      while ( *s ) {
        basenumber++;
        tablename = s; while ( *s ) s++; s++;
        while ( *s ) s++; s++;
        if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
            || ( T = functions[funnum].tabl ) == 0 ) {
            MesPrint("&%s should be a previously declared table",tablename);
        }
        else if ( T->sparse == 0 ) {
            MesPrint("&%s should be a sparse table",tablename);
        }
    if ( T->spare && T->mode == 0 ) {
        MesPrint("In table %s we have a problem with stubb orders in CoTBuse",tablename);
        error = -1;
    }
/*      if ( T->spare == 0 ) { SpareTable(T); } */
        for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
            for ( j = 0; j < NUMOBJECTS; j++ ) {
                if ( d->iblocks[i]->objects[j].tablenumber == basenumber ) {
                    arguments = p = (UBYTE *)(d->iblocks[i]->objects[j].element);
/*
                    Now translate the arguments and see whether we need
                    this one....
*/
                    for ( k = 0, w = AT.WorkPointer; k < T->numind; k++ ) {
                        ParseSignedNumber(x,p);
                        *w++ = x; p++;
```

```
                    }
                    sum = FindTableTree(T,AT.WorkPointer,1);
                    if ( sum < 0 ) {
                        MesPrint("Table %s in tablebase %s has not been loaded properly"
                                ,tablename,tablebasename);
                        error = 1;
                        continue;
                    }
                    sum += T->numind + 4;
                    mode = T->tablepointers[sum];
                    if ( ( mode & ELEMENTLOADED ) == ELEMENTLOADED ) {
                        T->tablepointers[sum] &= ~ELEMENTUSED;
                        continue;
                    }
                    if ( ( mode & ELEMENTUSED ) == 0 ) continue;
/*
                    We need this one!
*/
                    rhs = (UBYTE *)ReadijObject(d,i,j,(char *)arguments);
                    if ( rhs ) {
                        u = rhs; while ( *u ) u++;
                        size = u-rhs;
                        u = arguments; while ( *u ) u++;
                        size += u-arguments;
                        u = tablename; while ( *u ) u++;
                        size += u-tablename;
                        size += 6;
                        buffer  = (UBYTE *)Malloc1(size,"TableBase copy");
                        t = tablename; u = buffer;
                        while ( *t ) *u++ = *t++;
                        *u++ = '(';
                        t = arguments;
                        while ( *t ) *u++ = *t++;
                        *u++ = ')'; *u++ = '=';

                        t = rhs;
                        while ( *t ) *u++ = *t++;
                        if ( t == rhs ) { *u++ = '0'; }
                        *u++ = 0; *u = 0;
                        M_free(rhs,"rhs in TBuse");

                        error1 = CoFill(buffer);

                        if ( error1 < 0 ) { return(error); }
                        if ( error1 != 0 ) error = error1;
                        M_free(buffer,"TableBase copy");
                    }
                    T->tablepointers[sum] &= ~ELEMENTUSED;
                    T->tablepointers[sum] |= ELEMENTLOADED;
                }
            }
        }
      }
    }
    return(error);
}

/*
    #] CoTBuse :
    #[ CoApply :

    Possibly to be followed by names of tables.
*/

int CoApply(UBYTE *s)
{
    GETIDENTITY
    UBYTE *tablename, c;
    WORD type, funnum, *w;
    TABLES T;
    LONG maxtogo = MAXPOSITIVE;
    int error = 0;
    w = AT.WorkPointer;
    if ( FG.cTable[*s] == 1 ) {
        maxtogo = 0;
        while ( FG.cTable[*s] == 1 ) {
            maxtogo = maxtogo*10 + (*s-'0');
            s++;
        }
        while ( *s == ',' ) s++;
        if ( maxtogo > MAXPOSITIVE || maxtogo < 0 ) maxtogo = MAXPOSITIVE;
    }
    *w++ = TYPEAPPLY; *w++ = 3; *w++ = maxtogo;
    while ( *s ) {
        tablename = s;
        if ( ( s = SkipAName(s) ) == 0 ) return(1);
        c = *s; *s = 0;
```

```
        if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
            || ( T = functions[funnum].tabl ) == 0 ) {
            MesPrint("&%s should be a previously declared table",tablename);
            error = 1;
        }
        else if ( T->sparse == 0 ) {
            MesPrint("&%s should be a sparse table",tablename);
            error = 1;
        }
        *w++ = funnum + FUNCTION;
        *s = c;
        while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
    }
    AT.WorkPointer[1] = w - AT.WorkPointer;
/*
    if ( AT.WorkPointer[1] > 2 ) {
        AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
    }
*/
    AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
/*
    AT.WorkPointer[0] = TYPEAPPLYRESET;
    AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
*/
    return(error);
}

/*
    #] CoApply :
    #[ CoTBhelp :
*/

char *helptb[] = {
     "The TableBase statement is used as follows:"
    ,"TableBase \"file.tbl\" keyword subkey(s)"
    ,"    in which we have"
    ,"Keyword   Subkey(s)   Action"
    ,"open                  Opens file.tbl for R/W"
    ,"create                Creates file.tbl for R/W. Old contents are lost"
    ,"load                  Loads all stubs of all tables"
    ,"load    tablename(s) Loads all stubs the tables mentioned"
    ,"enter                 Loads all stubs and rhs of all tables"
    ,"enter   tablename(s) Loads all stubs and rhs of the tables mentioned"
    ,"audit                 Prints list of contents"
/*  ,"replace tablename    saves a table (with overwrite)" */
/*  ,"replace tableelement saves a table element (with overwrite)" */
/*  ,"cleanup              makes tables contingent" */
    ,"addto   tablename    adds all elements if not yet there"
    ,"addto   tableelement adds element if not yet there"
/*  ,"delete  tablename    removes table from tablebase" */
/*  ,"delete  tableelement removes element from tablebase" */
    ,"on      compress     elements are stored in gzip format (default)"
    ,"off     compress     elements are stored in uncompressed format"
    ,"use                  compiles all needed elements"
    ,"use     tablename(s) compiles all needed elements of these tables"
    ,""
    ,"Related commands are:"
    ,"testuse              marks which tbl_ elements occur for all tables"
    ,"testuse tablename(s) marks which tbl_ elements occur for given tables"
    ,"apply                replaces tbl_ if rhs available"
    ,"apply   tablename(s) replaces tbl_ for given tables if rhs available"
    ,""
        };

int CoTBhelp(UBYTE *s)
{
    int i, ii = sizeof(helptb)/sizeof(char *);
    DUMMYUSE(s);
    for ( i = 0; i < ii; i++ ) MesPrint("%s",helptb[i]);
    return(0);
}

/*
    #] CoTBhelp :
    #[ ReWorkT :

    Replaces the STUBBS of the functions in the list.
    This gains one space. Hence we have to be very careful
*/

VOID ReWorkT(WORD *term, WORD *funs, WORD numfuns)
{
    WORD *tstop, *tend, *m, *t, *tt, *mm, *mmm, *r, *rr;
    int i, j;
    tend = term + *term; tstop = tend - ABS(tend[-1]);
    m = t = term+1;
    while ( t < tstop ) {
```

```
        if ( *t == TABLESTUB ) {
            for ( i = 0; i < numfuns; i++ ) {
                if ( -t[FUNHEAD] == funs[i] ) break;
            }
            if ( numfuns == 0 || i < numfuns ) { /* Hit */
                i = t[1] - 1;
                *m++ = -t[FUNHEAD]; *m++ = i; t += 2; i -= FUNHEAD;
                if ( m < t ) { for ( j = 0; j < FUNHEAD-2; j++ ) *m++ = *t++; }
                else { m += FUNHEAD-2; t += FUNHEAD-2; }
                t++;
                while ( i-- > 0 ) { *m++ = *t++; }
                tt = t; mm = m;
                if ( mm < tt ) {
                    while ( tt < tend ) *mm++ = *tt++;
                    *term = mm - term;
                    tend = term + *term; tstop = tend - ABS(tend[-1]);
                    t = m;
                }
            }
            else { goto inc; }
        }
        else if ( *t >= FUNCTION ) {
            tt = t + t[1];
            mm = m;
            for ( j = 0; j < FUNHEAD; j++ ) {
                if ( m == t ) { m++; t++; }
                else *m++ = *t++;
            }
            while ( t < tt ) {
                if ( *t <= -FUNCTION ) {
                    if ( m == t ) { m++; t++; }
                    else *m++ = *t++;
                }
                else if ( *t < 0 ) {
                    if ( m == t ) { m += 2; t += 2; }
                    else { *m++ = *t++; *m++ = *t++; }
                }
                else {
                    rr = t + *t; mmm = m;
                    for ( j = 0; j < ARGHEAD; j++ ) {
                        if ( m == t ) { m++; t++; }
                        else *m++ = *t++;
                    }
                    while ( t < rr ) {
                        r = t + *t;
                        ReWorkT(t,funs,numfuns);
                        j = *t;
                        if ( m == t ) { m += j; t += j; }
                        else { while ( j-- >= 0 ) *m++ = *t++; }
                        t = r;
                    }
                    *mmm = m-mmm;
                }
            }
            mm[1] = m - mm;
            t = tt;
        }
        else {
inc:        j = t[1];
            if ( m < t ) { while ( j-- >= 0 ) *m++ = *t++; }
            else { m += j; t += j; }
        }
    }
    if ( m < t ) {
        while ( t < tend ) *m++ = *t++;
        *term = m - term;
    }
}

/*
    #] ReWorkT :
    #[ Apply :
*/

WORD Apply(WORD *term, WORD level)
{
    WORD *funs, numfuns;
    TABLES T;
    int i, j;
    CBUF *C = cbuf+AM.rbufnum;
/*
    Point the tables in the proper direction
*/
    numfuns = C->lhs[level][1] - 2;
    funs = C->lhs[level] + 2;
    if ( numfuns > 0 ) {
        for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
```

```
            if ( ( T = functions[i].tabl ) != 0 ) {
                for ( j = 0; j < numfuns; j++ ) {
                    if ( i == (funs[j]-FUNCTION) && T->spare ) {
                        FlipTable(&(functions[i]),0);
                        break;
                    }
                }
            }
        }
    }
    else {
        for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
            if ( ( T = functions[i].tabl ) != 0 ) {
                if ( T->spare ) FlipTable(&(functions[i]),0);
            }
        }
    }
/*
    Now the replacements everywhere of
        id tbl_(table,?a) = table(?a);
    Actually, this has to be done recursively.
    Note that we actually gain one space.
*/
    ReWorkT(term,funs,numfuns);
    return(0);
}

/*
    #] Apply :
    #[ ApplyExec :

    Replaces occurrences of tbl_(table,indices,pattern) by the proper
    rhs of table(indices,pattern). It does this up to maxtogo times
    in the given term. It starts with the occurrences inside the
    arguments of functions. If necessary it finishes at groundlevel.
    An infite number of tries is indicates by maxtogo = 2^15-1 or 2^31-1.
    The occurrences are replaced by subexpressions. This allows TestSub
    to finish the job properly.

    The main trick here is T = T->spare which turns to the proper rhs.

    The return value is the number of substitutions that can still be made
    based on maxtogo. Hence, if the returnvalue is different from maxtogo
    there was a substitution.
*/

int ApplyExec(WORD *term, int maxtogo, WORD level)
{
    GETIDENTITY
    WORD rhsnumber, *Tpattern, *funs, numfuns, funnum;
    WORD ii, *t, *t1, *w, *p, *m, *m1, *u, *r, tbufnum, csize, wilds;
    NESTING NN;
    int i, j, isp, stilltogo;
    CBUF *C;
    TABLES T;
/*
    Startup. We need NestPoin for when we have to replace something deep down.
*/
    t = term;
    m = t + *t;
    csize = ABS(m[-1]);
    m -= csize;
    AT.NestPoin->termsize = t;
    if ( AT.NestPoin == AT.Nest ) AN.EndNest = t + *t;
    t++;
/*
    First we look inside function arguments. Also when clean!
*/
    while ( t < m ) {
        if ( *t < FUNCTION ) { t += t[1]; continue; }
        if ( functions[*t-FUNCTION].spec > 0 ) { t += t[1]; continue; }
        AT.NestPoin->funsize = t;
        r = t + t[1];
        t += FUNHEAD;
        while ( t < r ) {
            if ( *t < 0 ) { NEXTARG(t); continue; }
            AT.NestPoin->argsize = t1 = t;
            u = t + *t;
            t += ARGHEAD;
            AT.NestPoin++;
            while ( t < u ) {
/*
                Now we loop over the terms inside a function argument
                This defines a recursion and we have to call ApplyExec again.
                The real problem is when we catch something and we have
                to insert a subexpression pointer. This may use more or
                less space and the whole term has to be readjusted.
```

```
                This is why we have the NestPoin variables. They tell us
                where the sizes of the term, the function and the arguments
                are sitting, and also where the dirty flags are.
                This readjusting is of course done in the groundlevel code.
                Here we worry abound the maxtogo count.
*/
                stilltogo = ApplyExec(t,maxtogo,level);
                if ( stilltogo != maxtogo ) {
                    if ( stilltogo <= 0 ) {
                        AT.NestPoin--;
                        return(stilltogo);
                    }
                    maxtogo = stilltogo;
                    u = t1 + *t1;
                    m = term + *term - csize;
                }
                t += *t;
            }
            AT.NestPoin--;
        }
    }
/*
    Now we look at the ground level
*/
    C = cbuf+AM.rbufnum;
    t = term + 1;
    while ( t < m ) {
        if ( *t != TABLESTUB ) { t += t[1]; continue; }
        funnum = -t[FUNHEAD];
        if ( ( funnum < FUNCTION )
          || ( funnum >= FUNCTION+WILDOFFSET )
          || ( ( T = functions[funnum-FUNCTION].tabl ) == 0 )
          || ( T->sparse == 0 )
          || ( T->spare == 0 ) ) { t += t[1]; continue; }
        numfuns = C->lhs[level][1] - 3;
        funs = C->lhs[level] + 3;
        if ( numfuns > 0 ) {
            for ( i = 0; i < numfuns; i++ ) {
                if ( funs[i] == funnum ) break;
            }
            if ( i >= numfuns ) { t += t[1]; continue; }
        }
        r = t + t[1];
        AT.NestPoin->funsize = t + 1;
        t1 = t;
        t += FUNHEAD + 1;
/*
        Test whether the table catches
        Test 1: index arguments and range. isp will be the number
            of the element in the table.
*/
        T = T->spare;
#ifdef WITHPTHREADS
        Tpattern = T->pattern[identity];
#else
        Tpattern = T->pattern;
#endif
        p = Tpattern+FUNHEAD+1;
        for ( i = 0; i < T->numind; i++, t += 2 ) {
            if ( *t != -SNUMBER ) break;
        }
        if ( i < T->numind ) { t = r; continue; }
        isp = FindTableTree(T,t1+FUNHEAD+1,2);
        if ( isp < 0 ) { t = r; continue; }
        rhsnumber = T->tablepointers[isp+T->numind];
#if ( TABLEEXTENSION == 2 )
        tbufnum = T->bufnum;
#else
        tbufnum = T->tablepointers[isp+T->numind+1];
#endif
        t = t1+FUNHEAD+2;
        ii = T->numind;
        while ( --ii >= 0 ) {
            *p = *t; t += 2; p += 2;
        }
/*
        If there are more arguments we have to do some
        pattern matching. This should be easy. We addapted the
        pattern, so that the array indices match already.
*/
#ifdef WITHPTHREADS
        AN.FullProto = T->prototype[identity];
#else
        AN.FullProto = T->prototype;
#endif
        AN.WildValue = AN.FullProto + SUBEXPSIZE;
        AN.WildStop = AN.FullProto+AN.FullProto[1];
```

```
        ClearWild(BHEAD0);
        AN.RepFunNum = 0;
        AN.RepFunList = AN.EndNest;
        AT.WorkPointer = (WORD *)(((UBYTE *)(AN.EndNest)) + AM.MaxTer/2);
/*
        The RepFunList is after the term but not very relevant.
        We need because MatchFunction uses it
*/
        if ( AT.WorkPointer + t1[1] >= AT.WorkTop ) { MesWork(); }
        wilds = 0;
        w = AT.WorkPointer;
        *w++ = -t1[FUNHEAD];
        *w++ = t1[1] - 1;
        for ( i = 2; i < FUNHEAD; i++ ) *w++ = t1[i];
        t = t1 + FUNHEAD+1;
        while ( t < r ) *w++ = *t++;
        t = AT.WorkPointer;
        AT.WorkPointer = w;
        if ( MatchFunction(BHEAD Tpattern,t,&wilds) > 0 ) {
/*
            Here we caught one. Now we should worry about:
            1: inserting the subexpression pointer with its wildcards
            2: NestPoin because we may not be at the lowest level
            The function starts at t1.
*/
#ifdef WITHPTHREADS
            m1 = T->prototype[identity];
#else
            m1 = T->prototype;
#endif
            m1[2] = rhsnumber;
            m1[4] = tbufnum;
            t = t1;
            j = t[1];
            i = m1[1];
            if ( j > i ) {
                j = i - j;
                NCOPY(t,m1,i);
                m1 = AN.EndNest;
                while ( r < m1 ) *t++ = *r++;
                AN.EndNest = t;
                *term += j;
                NN = AT.NestPoin;
                while ( NN > AT.Nest ) {
                    NN--;
                    NN->termsize[0] += j;
                    NN->funsize[1] += j;
                    NN->argsize[0] += j;
                    NN->funsize[2] |= DIRTYFLAG;
                    NN->argsize[1] |= DIRTYFLAG;
                }
                m += j;
            }
            else if ( j < i ) {
                j = i-j;
                t = AN.EndNest;
                while ( t >= r ) { t[j] = *t; t--; }
                t = t1;
                NCOPY(t,m1,i);
                AN.EndNest += j;
                *term += j;
                NN = AT.NestPoin;
                while ( NN > AT.Nest ) {
                    NN--;
                    NN->termsize[0] += j;
                    NN->funsize[1] += j;
                    NN->argsize[0] += j;
                    NN->funsize[2] |= DIRTYFLAG;
                    NN->argsize[1] |= DIRTYFLAG;
                }
                m += j;
            }
            else {
                NCOPY(t,m1,j);
            }
            r = t1 + t1[1];
            maxtogo--;
            if ( maxtogo <= 0 ) return(maxtogo);
        }
        t = r;
    }
    return(maxtogo);
}

/*
    #] ApplyExec :
    #[ ApplyReset :
```

```
*/

WORD ApplyReset(WORD level)
{
    WORD *funs, numfuns;
    TABLES T;
    int i, j;
    CBUF *C = cbuf+AM.rbufnum;

    numfuns = C->lhs[level][1] - 2;
    funs = C->lhs[level] + 2;
    if ( numfuns > 0 ) {
        for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
            if ( ( T = functions[i].tabl ) != 0 ) {
                for ( j = 0; j < numfuns; j++ ) {
                    if ( i == (funs[j]-FUNCTION) && T->spare ) {
                        FlipTable(&(functions[i]),1);
                        break;
                    }
                }
            }
        }
    }
    else {
        for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
            if ( ( T = functions[i].tabl ) != 0 ) {
                if ( T->spare ) FlipTable(&(functions[i]),1);
            }
        }
    }
    return(0);
}

/*
    #] ApplyReset :
    #[ TableReset :
*/

WORD TableReset()
{
    TABLES T;
    int i;

    for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
        if ( ( T = functions[i].tabl ) != 0 && T->spare && T->mode == 0 ) {
            functions[i].tabl = T->spare;
        }
    }
    return(0);
}

/*
    #] TableReset :
    #[ LoadTableElement :
?????
int LoadTableElement(DBASE *d, TABLE *T, WORD num)
{
}

    #] LoadTableElement :
    #[ ReleaseTB :

    Releases all TableBases
*/

int ReleaseTB()
{
    DBASE *d;
    int i;
    for ( i = NumTableBases - 1; i >= 0; i-- ) {
        d = tablebases+i;
        fclose(d->handle);
        FreeTableBase(d);
    }
    return(0);
}

/*
    #] ReleaseTB :
*/
```

4.114 threads.c File Reference

4.114.1 Detailed Description

Routines for the interface of FORM with the pthreads library

This is the main part of the parallelization of TFORM. It is important to also look in the files structs.h and variable.h because the treatment of the A and B structs is essential (these structs are used by means of the macros AM, AP, AC, AS, AR, AT, AN, AO and AX). Also the definitions and use of the macros BHEAD and PHEAD should be looked up.

The sources are set up in such a way that if WITHPTHREADS isn't defined there is no trace of pthread parallelization. The reason is that TFORM is far more memory hungry than sequential FORM.

Special attention should also go to the locks. The proper use of the locks is essential and determines whether TFORM can work at all. We use the LOCK/UNLOCK macros which are empty in the case of sequential FORM These locks are at many places in the source files when workers can interfere with each others data or with the data of the master.

Definition in file threads.c.

4.115 threads.c

Go to the documentation of this file.

```
00001
00022 /* #[ License : */
00023 /*
00024  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00025  *   When using this file you are requested to refer to the publication
00026  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00027  *   This is considered a matter of courtesy as the development was paid
00028  *   for by FOM the Dutch physics granting agency and we would like to
00029  *   be able to track its scientific use to convince FOM of its value
00030  *   for the community.
00031  *
00032  *   This file is part of FORM.
00033  *
00034  *   FORM is free software: you can redistribute it and/or modify it under the
00035  *   terms of the GNU General Public License as published by the Free Software
00036  *   Foundation, either version 3 of the License, or (at your option) any later
00037  *   version.
00038  *
00039  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00040  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00041  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00042  *   details.
00043  *
00044  *   You should have received a copy of the GNU General Public License along
00045  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00046  */
00047 /* #] License : */
00048
00049 #ifdef WITHPTHREADS
00050
00051 #define WHOLEBRACKETS
00052 /*
00053     #[ Variables :
00054
00055     The sortbot additions are from 17-may-2007 and after. They consitute
00056     an attempt to make the final merge sorting faster for the master.
00057     This way the master has only one compare per term.
00058     It does add some complexity, but the final merge routine (MasterMerge)
00059     is much simpler for the sortbots. On the other hand the original merging is
00060     for a large part a copy of the MergePatches routine in sort.c and hence
00061     even though complex the bad part has been thoroughly debugged.
00062 */
00063
00064 #include "form3.h"
00065
00066 static int numberofthreads;
```

```
static int numberofworkers;
static int identityofthreads = 0;
static int *listofavailables;
static int topofavailables = 0;
static pthread_key_t identitykey;
static INILOCK(numberofthreadslock);
static INILOCK(availabilitylock);
static pthread_t *threadpointers = 0;
static pthread_mutex_t *wakeuplocks;
static pthread_mutex_t *wakeupmasterthreadlocks;
static pthread_cond_t *wakeupconditions;
static pthread_condattr_t *wakeupconditionattributes;
static int *wakeup;
static int *wakeupmasterthread;
static INILOCK(wakeupmasterlock);
static pthread_cond_t wakeupmasterconditions = PTHREAD_COND_INITIALIZER;
static pthread_cond_t *wakeupmasterthreadconditions;
static int wakeupmaster = 0;
static int identityretval;
/* static INILOCK(clearclocklock); */
static LONG *timerinfo;
static LONG *sumtimerinfo;
static int numberclaimed;

static THREADBUCKET **threadbuckets, **freebuckets;
static int numthreadbuckets;
static int numberoffullbuckets;

/* static int numberbusy = 0; */

INILOCK(dummylock);
INIRWLOCK(dummyrwlock);
static pthread_cond_t dummywakeupcondition = PTHREAD_COND_INITIALIZER;

#ifdef WITHSORTBOTS
static POSITION SortBotPosition;
static int numberofsortbots;
static INILOCK(wakeupsortbotlock);
static pthread_cond_t wakeupsortbotconditions = PTHREAD_COND_INITIALIZER;
static int topsortbotavailables = 0;
static LONG numberofterms;
#endif

/*
    #] Variables :
    #[ Identity :
        #[ StartIdentity :
*/
void StartIdentity()
{
    pthread_key_create(&identitykey,FinishIdentity);
}

/*
        #] StartIdentity :
        #[ FinishIdentity :
*/
void FinishIdentity(void *keyp)
{
    DUMMYUSE(keyp);
/*  free(keyp); */
}

/*
        #] FinishIdentity :
        #[ SetIdentity :
*/
int SetIdentity(int *identityretval)
{
/*
#ifdef _MSC_VER
    printf("addr %d\n",&numberofthreadslock);
    printf("size %d\n",sizeof(numberofthreadslock));
#endif
*/
    LOCK(numberofthreadslock);
    *identityretval = identityofthreads++;
    UNLOCK(numberofthreadslock);
    pthread_setspecific(identitykey,(void *)identityretval);
    return(*identityretval);
}

/*
        #] SetIdentity :
        #[ WhoAmI :
*/

```

```
int WhoAmI()
{
    int *identity;
/*
    First a fast exit for when there is at most one thread
*/
    if ( identityofthreads <= 1 ) return(0);
/*
    Now the reading of the key.
    According to the book the statement should read:

    pthread_getspecific(identitykey,(void **)(&identity));

    but according to the information in pthread.h it is:
*/
    identity = (int *)pthread_getspecific(identitykey);
    return(*identity);
}

/*
        #] WhoAmI :
        #[ BeginIdentities :
*/
VOID BeginIdentities()
{
    StartIdentity();
    SetIdentity(&identityretval);
}

/*
        #] BeginIdentities :
    #] Identity :
    #[ StartHandleLock :
*/
void StartHandleLock()
{
    AM.handlelock = dummyrwlock;
}

/*
    #] StartHandleLock :
    #[ StartAllThreads :
*/
int StartAllThreads(int number)
{
    int identity, j, dummy, mul;
    ALLPRIVATES *B;
    pthread_t thethread;
    identity = WhoAmI();

#ifdef WITHSORTBOTS
    timerinfo = (LONG *)Malloc1(sizeof(LONG)*number*2,"timerinfo");
    sumtimerinfo = (LONG *)Malloc1(sizeof(LONG)*number*2,"sumtimerinfo");
    for ( j = 0; j < number*2; j++ ) { timerinfo[j] = 0; sumtimerinfo[j] = 0; }
    mul = 2;
#else
    timerinfo = (LONG *)Malloc1(sizeof(LONG)*number,"timerinfo");
    sumtimerinfo = (LONG *)Malloc1(sizeof(LONG)*number,"sumtimerinfo");
    for ( j = 0; j < number; j++ ) { timerinfo[j] = 0; sumtimerinfo[j] = 0; }
    mul = 1;
#endif

    listofavailables = (int *)Malloc1(sizeof(int)*(number+1),"listofavailables");
    threadpointers = (pthread_t *)Malloc1(sizeof(pthread_t)*number*mul,"threadpointers");
    AB = (ALLPRIVATES **)Malloc1(sizeof(ALLPRIVATES *)*number*mul,"Private structs");

    wakeup = (int *)Malloc1(sizeof(int)*number*mul,"wakeup");
    wakeuplocks = (pthread_mutex_t *)Malloc1(sizeof(pthread_mutex_t)*number*mul,"wakeuplocks");
    wakeupconditions = (pthread_cond_t *)Malloc1(sizeof(pthread_cond_t)*number*mul,"wakeupconditions");
    wakeupconditionattributes = (pthread_condattr_t *)
            Malloc1(sizeof(pthread_condattr_t)*number*mul,"wakeupconditionattributes");

    wakeupmasterthread = (int *)Malloc1(sizeof(int)*number*mul,"wakeupmasterthread");
    wakeupmasterthreadlocks = (pthread_mutex_t *)Malloc1(sizeof(pthread_mutex_t)*number*mul,"wakeupmasterthreadlocks");
    wakeupmasterthreadconditions = (pthread_cond_t *)Malloc1(sizeof(pthread_cond_t)*number*mul,"wakeupmasterthread");

    numberofthreads = number;
    numberofworkers = number - 1;
    threadpointers[identity] = pthread_self();
    topofavailables = 0;
    for ( j = 1; j < number; j++ ) {
        if ( pthread_create(&thethread,NULL,RunThread,(void *)(&dummy)) )
            goto failure;
    }
```

```
/*
    Now we initialize the master at the same time that the workers are doing so.
*/
    B = InitializeOneThread(identity);
    AR.infile = &(AR.Fscr[0]);
    AR.outfile = &(AR.Fscr[1]);
    AR.hidefile = &(AR.Fscr[2]);
    AM.sbuflock = dummylock;
    AS.inputslock = dummylock;
    AS.outputslock = dummylock;
    AS.MaxExprSizeLock = dummylock;
    AP.PreVarLock = dummylock;
    AC.halfmodlock = dummylock;
    MakeThreadBuckets(number,0);
/*
    Now we wait for the workers to finish their startup.
    We don't want to initialize the sortbots yet and run the risk that
    some of them may end up with a lower number than one of the workers.
*/
    MasterWaitAll();
#ifdef WITHSORTBOTS
    if ( numberofworkers > 2 ) {
        numberofsortbots = numberofworkers-2;
        for ( j = numberofworkers+1; j < 2*numberofworkers-1; j++ ) {
            if ( pthread_create(&thethread,NULL,RunSortBot,(void *)(&dummy)) )
                goto failure;
        }
    }
    else {
        numberofsortbots = 0;
    }
    MasterWaitAllSortBots();
    DefineSortBotTree();
#endif
    IniSortBlocks(number-1);
    AS.MasterSort = 0;
    AM.storefilelock = dummylock;
/*
MesPrint("AB = %x %x %x  %d",AB[0],AB[1],AB[2], identityofthreads);
*/
    return(0);
failure:
    MesPrint("Cannot start %d threads",number);
    Terminate(-1);
    return(-1);
}

/*
    #] StartAllThreads :
    #[ InitializeOneThread :
*/
UBYTE *scratchname[] = { (UBYTE *)"scratchsize",
                         (UBYTE *)"scratchsize",
                         (UBYTE *)"hidesize" };
ALLPRIVATES *InitializeOneThread(int identity)
{
    WORD *t, *ScratchBuf;
    int i, j, bsize, *bp;
    LONG ScratchSize[3], IOsize;
    ALLPRIVATES *B;
    UBYTE *s;

    wakeup[identity] = 0;
    wakeuplocks[identity] = dummylock;
    pthread_condattr_init(&(wakeupconditionattributes[identity]));
    pthread_condattr_setpshared(&(wakeupconditionattributes[identity]),PTHREAD_PROCESS_PRIVATE);
    wakeupconditions[identity] = dummywakeupcondition;
    pthread_cond_init(&(wakeupconditions[identity]),&(wakeupconditionattributes[identity]));
    wakeupmasterthread[identity] = 0;
    wakeupmasterthreadlocks[identity] = dummylock;
    wakeupmasterthreadconditions[identity] = dummywakeupcondition;

    bsize = sizeof(ALLPRIVATES);
    bsize = (bsize+sizeof(int)-1)/sizeof(int);
    B = (ALLPRIVATES *)Malloc1(sizeof(int)*bsize,"B struct");
    for ( bp = (int *)B, j = 0; j < bsize; j++ ) *bp++ = 0;

    AB[identity] = B;
/*
            12-jun-2007 JV:
    For the timing one has to know a few things:
    The POSIX standard is that there is only a single process ID and that
    getrusage returns the time of all the threads together.
    Under Linux there are two methods though: The older LinuxThreads and NPTL.
    LinuxThreads gives each thread its own process id. This makes that we
    can time the threads with getrusage, and hence this was done. Under NPTL
    this has been 'corrected' and suddenly getruage doesn't work anymore the
```

```
    way it used to. Now we need
        clock_gettime(CLOCK_THREAD_CPUTIME_ID,&timing)
    which is declared in <time.h> and we need -lrt extra in the link statement.
    (this is at least the case on blade02 at DESY-Zeuthen).
    See also the code in tools.c at the routine Timer.
    We may still have to include more stuff there.
*/
    if ( identity > 0 ) TimeCPU(0);

#ifdef WITHSORTBOTS

    if ( identity > numberofworkers ) {
/*
        Some workspace is needed when we have a PolyFun and we have to add
        two terms and the new result is going to be longer than the old result.
*/
        LONG length = AM.WorkSize*sizeof(WORD)/8+AM.MaxTer*2;
        AT.WorkSpace = (WORD *)Malloc1(length,"WorkSpace");
        AT.WorkTop = AT.WorkSpace + length/sizeof(WORD);
        AT.WorkPointer = AT.WorkSpace;
        AT.identity = identity;
/*
        The SB struct gets treated in IniSortBlocks.
        The SortBotIn variables will be defined DefineSortBotTree.
*/
        if ( AN.SoScratC == 0 ) {
            AN.SoScratC = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"Scratch in SortBot");
        }
        AT.SS = (SORTING *)Malloc1(sizeof(SORTING),"dummy sort buffer");
        AT.SS->PolyFlag = 0;

        AT.comsym[0] = 8;
        AT.comsym[1] = SYMBOL;
        AT.comsym[2] = 4;
        AT.comsym[3] = 0;
        AT.comsym[4] = 1;
        AT.comsym[5] = 1;
        AT.comsym[6] = 1;
        AT.comsym[7] = 3;
        AT.comnum[0] = 4;
        AT.comnum[1] = 1;
        AT.comnum[2] = 1;
        AT.comnum[3] = 3;
        AT.comfun[0] = FUNHEAD+4;
        AT.comfun[1] = FUNCTION;
        AT.comfun[2] = FUNHEAD;
        AT.comfun[3] = 0;
#if FUNHEAD > 3
        for ( i = 4; i <= FUNHEAD; i++ )
            AT.comfun[i] = 0;
#endif
        AT.comfun[FUNHEAD+1] = 1;
        AT.comfun[FUNHEAD+2] = 1;
        AT.comfun[FUNHEAD+3] = 3;
        AT.comind[0] = 7;
        AT.comind[1] = INDEX;
        AT.comind[2] = 3;
        AT.comind[3] = 0;
        AT.comind[4] = 1;
        AT.comind[5] = 1;
        AT.comind[6] = 3;

        AT.inprimelist = -1;
        AT.sizeprimelist = 0;
        AT.primelist = 0;
        AT.bracketinfo = 0;

        AR.CompareRoutine = &Compare1;

        AR.sLevel = 0;
        AR.wranfia = 0;
        AR.wranfcall = 0;
        AR.wranfnpair1 = NPAIR1;
        AR.wranfnpair2 = NPAIR2;
        AN.NumFunSorts = 5;
        AN.MaxFunSorts = 5;
        AN.SplitScratch = 0;
        AN.SplitScratchSize = AN.InScratch = 0;
        AN.SplitScratch1 = 0;
        AN.SplitScratchSize1 = AN.InScratch1 = 0;

        AN.FunSorts = (SORTING **)Malloc1((AN.NumFunSorts+1)*sizeof(SORTING *),"FunSort pointers");
        for ( i = 0; i <= AN.NumFunSorts; i++ ) AN.FunSorts[i] = 0;
        AN.FunSorts[0] = AT.S0 = AT.SS;
        AN.idfunctionflag = 0;
        AN.tryterm = 0;

```

```
        return(B);
    }
    if ( identity == 0 && AN.SoScratC == 0 ) {
        AN.SoScratC = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"Scratch in SortBot");
    }
#endif
    AR.CurDum = AM.IndDum;
    for ( j = 0; j < 3; j++ ) {
        if ( identity == 0 ) {
            if ( j == 2 ) {
                ScratchSize[j] = AM.HideSize;
            }
            else {
                ScratchSize[j] = AM.ScratSize;
            }
            if ( ScratchSize[j] < 10*AM.MaxTer ) ScratchSize[j] = 10 * AM.MaxTer;
        }
        else {
/*
            ScratchSize[j] = AM.ScratSize / (numberofthreads-1);
            ScratchSize[j] = ScratchSize[j] / 20;
            if ( ScratchSize[j] < 10*AM.MaxTer ) ScratchSize[j] = 10 * AM.MaxTer;
*/
            if ( j == 1 ) ScratchSize[j] = AM.ThreadScratOutSize;
            else          ScratchSize[j] = AM.ThreadScratSize;
            if ( ScratchSize[j] < 4*AM.MaxTer ) ScratchSize[j] = 4 * AM.MaxTer;
            AR.Fscr[j].name = 0;
        }
        ScratchSize[j] = ( ScratchSize[j] + 255 ) / 256;
        ScratchSize[j] = ScratchSize[j] * 256;
        ScratchBuf = (WORD *)Malloc1(ScratchSize[j]*sizeof(WORD),(char *)(scratchname[j]));
        AR.Fscr[j].POsize = ScratchSize[j] * sizeof(WORD);
        AR.Fscr[j].POfull = AR.Fscr[j].POfill = AR.Fscr[j].PObuffer = ScratchBuf;
        AR.Fscr[j].POstop = AR.Fscr[j].PObuffer + ScratchSize[j];
        PUTZERO(AR.Fscr[j].POposition);
        AR.Fscr[j].pthreadslock = dummylock;
        AR.Fscr[j].wPOsize = AR.Fscr[j].POsize;
        AR.Fscr[j].wPObuffer = AR.Fscr[j].PObuffer;
        AR.Fscr[j].wPOfill = AR.Fscr[j].POfill;
        AR.Fscr[j].wPOfull = AR.Fscr[j].POfull;
        AR.Fscr[j].wPOstop = AR.Fscr[j].POstop;
    }
    AR.InInBuf = 0;
    AR.InHiBuf = 0;
    AR.Fscr[0].handle = -1;
    AR.Fscr[1].handle = -1;
    AR.Fscr[2].handle = -1;
    AR.FoStage4[0].handle = -1;
    AR.FoStage4[1].handle = -1;
    IOsize = AM.S0->file.POsize;
#ifdef WITHZLIB
    AR.FoStage4[0].ziosize = IOsize;
    AR.FoStage4[1].ziosize = IOsize;
#endif
    AR.FoStage4[0].POsize  = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
    AR.FoStage4[1].POsize  = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);

    AR.hidefile = &(AR.Fscr[2]);
    AR.StoreData.Handle = -1;
    AR.SortType = AC.SortType;

    AN.IndDum = AM.IndDum;

    if ( identity > 0 ) {
        s = (UBYTE *)(FG.fname); i = 0;
        while ( *s ) { s++; i++; }
        s = (UBYTE *)Malloc1(sizeof(char)*(i+12),"name for Fscr[0] file");
        sprintf((char *)s,"%s.%d",FG.fname,identity);
        s[i-3] = 's'; s[i-2] = 'c'; s[i-1] = '0';
        AR.Fscr[0].name = (char *)s;
        s = (UBYTE *)(FG.fname); i = 0;
        while ( *s ) { s++; i++; }
        s = (UBYTE *)Malloc1(sizeof(char)*(i+12),"name for Fscr[1] file");
        sprintf((char *)s,"%s.%d",FG.fname,identity);
        s[i-3] = 's'; s[i-2] = 'c'; s[i-1] = '1';
        AR.Fscr[1].name = (char *)s;
    }

    AR.CompressBuffer = (WORD *)Malloc1((AM.CompressSize+10)*sizeof(WORD),"compresssize");
    AR.ComprTop = AR.CompressBuffer + AM.CompressSize;
    AR.CompareRoutine = &Compare1;
/*
    Here we make all allocations for the struct AT
    (which is AB[identity].T or B->T with B = AB+identity).
*/
    AT.WorkSpace = (WORD *)Malloc1(AM.WorkSize*sizeof(WORD),"WorkSpace");
    AT.WorkTop = AT.WorkSpace + AM.WorkSize;
```

```
    AT.WorkPointer = AT.WorkSpace;

    AT.Nest = (NESTING)Malloc1((LONG)sizeof(struct NeStInG)*AM.maxFlevels,"functionlevels");
    AT.NestStop = AT.Nest + AM.maxFlevels;
    AT.NestPoin = AT.Nest;

    AT.WildMask = (WORD *)Malloc1((LONG)AM.MaxWildcards*sizeof(WORD),"maxwildcards");

    LOCK(availabilitylock);
    AT.ebufnum = inicbufs();        /* Buffer for extras during execution */
    AT.fbufnum = inicbufs();        /* Buffer for caching in factorization */
    AT.allbufnum = inicbufs();      /* Buffer for id,all */
    AT.aebufnum = inicbufs();       /* Buffer for id,all */
    UNLOCK(availabilitylock);

    AT.RepCount = (int *)Malloc1((LONG)((AM.RepMax+3)*sizeof(int)),"repeat buffers");
    AN.RepPoint = AT.RepCount;
    AN.polysortflag = 0;
    AN.subsubveto = 0;
    AN.tryterm = 0;
    AT.RepTop = AT.RepCount + AM.RepMax;

    AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
                ,"argument list names");
    AT.WildcardBufferSize = AC.WildcardBufferSize;
    AT.previousEfactor = 0;

    AT.identity = identity;
    AT.LoadBalancing = 0;
/*
    Still to do: the SS stuff.
                 the Fscr[3]
                 the FoStage4[2]
*/
    if ( AT.WorkSpace == 0 ||
         AT.Nest == 0 ||
         AT.WildMask == 0 ||
         AT.RepCount == 0 ||
         AT.WildArgTaken == 0 ) goto OnError;
/*
    And initializations
*/
    AT.comsym[0] = 8;
    AT.comsym[1] = SYMBOL;
    AT.comsym[2] = 4;
    AT.comsym[3] = 0;
    AT.comsym[4] = 1;
    AT.comsym[5] = 1;
    AT.comsym[6] = 1;
    AT.comsym[7] = 3;
    AT.comnum[0] = 4;
    AT.comnum[1] = 1;
    AT.comnum[2] = 1;
    AT.comnum[3] = 3;
    AT.comfun[0] = FUNHEAD+4;
    AT.comfun[1] = FUNCTION;
    AT.comfun[2] = FUNHEAD;
    AT.comfun[3] = 0;
#if FUNHEAD > 3
    for ( i = 4; i <= FUNHEAD; i++ )
        AT.comfun[i] = 0;
#endif
    AT.comfun[FUNHEAD+1] = 1;
    AT.comfun[FUNHEAD+2] = 1;
    AT.comfun[FUNHEAD+3] = 3;
    AT.comind[0] = 7;
    AT.comind[1] = INDEX;
    AT.comind[2] = 3;
    AT.comind[3] = 0;
    AT.comind[4] = 1;
    AT.comind[5] = 1;
    AT.comind[6] = 3;
    AT.locwildvalue[0] = SUBEXPRESSION;
    AT.locwildvalue[1] = SUBEXPSIZE;
    for ( i = 2; i < SUBEXPSIZE; i++ ) AT.locwildvalue[i] = 0;
    AT.mulpat[0] = TYPEMULT;
    AT.mulpat[1] = SUBEXPSIZE+3;
    AT.mulpat[2] = 0;
    AT.mulpat[3] = SUBEXPRESSION;
    AT.mulpat[4] = SUBEXPSIZE;
    AT.mulpat[5] = 0;
    AT.mulpat[6] = 1;
    for ( i = 7; i < SUBEXPSIZE+5; i++ ) AT.mulpat[i] = 0;
    AT.proexp[0] = SUBEXPSIZE+4;
    AT.proexp[1] = EXPRESSION;
    AT.proexp[2] = SUBEXPSIZE;
    AT.proexp[3] = -1;
```

```
    AT.proexp[4] = 1;
    for ( i = 5; i < SUBEXPSIZE+1; i++ ) AT.proexp[i] = 0;
    AT.proexp[SUBEXPSIZE+1] = 1;
    AT.proexp[SUBEXPSIZE+2] = 1;
    AT.proexp[SUBEXPSIZE+3] = 3;
    AT.proexp[SUBEXPSIZE+4] = 0;
    AT.dummysubexp[0] = SUBEXPRESSION;
    AT.dummysubexp[1] = SUBEXPSIZE+4;
    for ( i = 2; i < SUBEXPSIZE; i++ ) AT.dummysubexp[i] = 0;
    AT.dummysubexp[SUBEXPSIZE] = WILDDUMMY;
    AT.dummysubexp[SUBEXPSIZE+1] = 4;
    AT.dummysubexp[SUBEXPSIZE+2] = 0;
    AT.dummysubexp[SUBEXPSIZE+3] = 0;

    AT.MinVecArg[0] = 7+ARGHEAD;
    AT.MinVecArg[ARGHEAD] = 7;
    AT.MinVecArg[1+ARGHEAD] = INDEX;
    AT.MinVecArg[2+ARGHEAD] = 3;
    AT.MinVecArg[3+ARGHEAD] = 0;
    AT.MinVecArg[4+ARGHEAD] = 1;
    AT.MinVecArg[5+ARGHEAD] = 1;
    AT.MinVecArg[6+ARGHEAD] = -3;
    t = AT.FunArg;
    *t++ = 4+ARGHEAD+FUNHEAD;
    for ( i = 1; i < ARGHEAD; i++ ) *t++ = 0;
    *t++ = 4+FUNHEAD;
    *t++ = 0;
    *t++ = FUNHEAD;
    for ( i = 2; i < FUNHEAD; i++ ) *t++ = 0;
    *t++ = 1; *t++ = 1; *t++ = 3;

    AT.inprimelist = -1;
    AT.sizeprimelist = 0;
    AT.primelist = 0;
    AT.nfac = AT.nBer = 0;
    AT.factorials = 0;
    AT.bernoullis = 0;
    AR.wranfia = 0;
    AR.wranfcall = 0;
    AR.wranfnpair1 = NPAIR1;
    AR.wranfnpair2 = NPAIR2;
    AR.wranfseed = 0;
    AN.SplitScratch = 0;
    AN.SplitScratchSize = AN.InScratch = 0;
    AN.SplitScratch1 = 0;
    AN.SplitScratchSize1 = AN.InScratch1 = 0;
/*
    Now the sort buffers. They depend on which thread. The master
    inherits the sortbuffer from AM.S0
*/
    if ( identity == 0 ) {
        AT.S0 = AM.S0;
    }
    else {
/*
        For the moment we don't have special settings.
        They may become costly in virtual memory.
*/
        AT.S0 = AllocSort(AM.S0->LargeSize*sizeof(WORD)/numberofworkers
                          ,AM.S0->SmallSize*sizeof(WORD)/numberofworkers
                          ,AM.S0->SmallEsize*sizeof(WORD)/numberofworkers
                          ,AM.S0->TermsInSmall
                          ,AM.S0->MaxPatches
/*                        ,AM.S0->MaxPatches/numberofworkers  */
                          ,AM.S0->MaxFpatches/numberofworkers
                          ,AM.S0->file.POsize);
    }
    AR.CompressPointer = AR.CompressBuffer;
/*
    Install the store caches (15-aug-2006 JV)
*/
    AT.StoreCache = AT.StoreCacheAlloc = 0;
    if ( AM.NumStoreCaches > 0 ) {
        STORECACHE sa, sb;
        LONG size;
        size = sizeof(struct StOrEcAcHe)+AM.SizeStoreCache;
        size = ((size-1)/sizeof(size_t)+1)*sizeof(size_t);
        AT.StoreCacheAlloc = (STORECACHE)Malloc1(size*AM.NumStoreCaches,"StoreCaches");
        sa = AT.StoreCache = AT.StoreCacheAlloc;
        for ( i = 0; i < AM.NumStoreCaches; i++ ) {
            sb = (STORECACHE)(VOID *)((UBYTE *)sa+size);
            if ( i == AM.NumStoreCaches-1 ) {
                sa->next = 0;
            }
            else {
                sa->next = sb;
            }
```

```
            SETBASEPOSITION(sa->position,-1);
            SETBASEPOSITION(sa->toppos,-1);
            sa = sb;
        }
    }

    ReserveTempFiles(2);
    return(B);
OnError:;
    MLOCK(ErrorMessageLock);
    MesPrint("Error initializing thread %d",identity);
    MUNLOCK(ErrorMessageLock);
    Terminate(-1);
    return(B);
}

/*
    #] InitializeOneThread :
    #[ FinalizeOneThread :
*/
void FinalizeOneThread(int identity)
{
    timerinfo[identity] = TimeCPU(1);
}

/*
    #] FinalizeOneThread :
    #[ ClearAllThreads :
*/
VOID ClearAllThreads()
{
    int i;
    MasterWaitAll();
    for ( i = 1; i <= numberofworkers; i++ ) {
        WakeupThread(i,CLEARCLOCK);
    }
#ifdef WITHSORTBOTS
    for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
        WakeupThread(i,CLEARCLOCK);
    }
#endif
}

/*
    #] ClearAllThreads :
    #[ TerminateAllThreads :
*/
VOID TerminateAllThreads()
{
    int i;
    for ( i = 1; i <= numberofworkers; i++ ) {
        GetThread(i);
        WakeupThread(i,TERMINATETHREAD);
    }
#ifdef WITHSORTBOTS
    for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
        WakeupThread(i,TERMINATETHREAD);
    }
#endif
    for ( i = 1; i <= numberofworkers; i++ ) {
        pthread_join(threadpointers[i],NULL);
    }
#ifdef WITHSORTBOTS
    for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
        pthread_join(threadpointers[i],NULL);
    }
#endif
}

/*
    #] TerminateAllThreads :
    #[ MakeThreadBuckets :
*/
int MakeThreadBuckets(int number, int par)
{
    int i;
    LONG sizethreadbuckets;
    THREADBUCKET *thr;
/*
    First we need a decent estimate. Not all terms should be maximal.
    Note that AM.MaxTer is in bytes!!!
    Maybe we should try to limit the size here a bit more effectively.
    This is a great consumer of memory.
*/
    sizethreadbuckets = ( AC.ThreadBucketSize + 1 ) * AM.MaxTer + 2*sizeof(WORD);
    if ( AC.ThreadBucketSize >= 250 )      sizethreadbuckets /= 4;
    else if ( AC.ThreadBucketSize >= 90 )  sizethreadbuckets /= 3;
```

```
    else if ( AC.ThreadBucketSize >= 40 )  sizethreadbuckets /= 2;
    sizethreadbuckets /= sizeof(WORD);

    if ( par == 0 ) {
        numthreadbuckets = 2*(number-1);
        threadbuckets = (THREADBUCKET **)Malloc1(numthreadbuckets*sizeof(THREADBUCKET *),"threadbuckets");
        freebuckets = (THREADBUCKET **)Malloc1(numthreadbuckets*sizeof(THREADBUCKET *),"threadbuckets");
    }
    if ( par > 0 ) {
        if ( sizethreadbuckets <= threadbuckets[0]->threadbuffersize ) return(0);
        for ( i = 0; i < numthreadbuckets; i++ ) {
            thr = threadbuckets[i];
            M_free(thr->deferbuffer,"deferbuffer");
        }
    }
    else {
        for ( i = 0; i < numthreadbuckets; i++ ) {
            threadbuckets[i] = (THREADBUCKET *)Malloc1(sizeof(THREADBUCKET),"threadbuckets");
            threadbuckets[i]->lock = dummylock;
        }
    }
    for ( i = 0; i < numthreadbuckets; i++ ) {
        thr = threadbuckets[i];
        thr->threadbuffersize = sizethreadbuckets;
        thr->free = BUCKETFREE;
        thr->deferbuffer = (POSITION *)Malloc1(2*sizethreadbuckets*sizeof(WORD)
                    +(AC.ThreadBucketSize+1)*sizeof(POSITION),"deferbuffer");
        thr->threadbuffer = (WORD *)(thr->deferbuffer+AC.ThreadBucketSize+1);
        thr->compressbuffer = (WORD *)(thr->threadbuffer+sizethreadbuckets);
        thr->busy = BUCKETPREPARINGTERM;
        thr->usenum = thr->totnum = 0;
        thr->type = BUCKETDOINGTERMS;
    }
    return(0);
}

/*
    #] MakeThreadBuckets :
    #[ GetTimerInfo :
*/

int GetTimerInfo(LONG** ti,LONG** sti)
{
    *ti = timerinfo;
    *sti = sumtimerinfo;
#ifdef WITHSORTBOTS
    return AM.totalnumberofthreads*2;
#else
    return AM.totalnumberofthreads;
#endif
}

/*
    #] GetTimerInfo :
    #[ WriteTimerInfo :
*/

void WriteTimerInfo(LONG* ti,LONG* sti)
{
    int i;
#ifdef WITHSORTBOTS
    int max = AM.totalnumberofthreads*2;
#else
    int max = AM.totalnumberofthreads;
#endif
    for ( i=0; i<max; ++i ) {
        timerinfo[i] = ti[i];
        sumtimerinfo[i] = sti[i];
    }
}

/*
    #] WriteTimerInfo :
    #[ GetWorkerTimes :
*/
LONG GetWorkerTimes()
{
    LONG retval = 0;
    int i;
    for ( i = 1; i <= numberofworkers; i++ ) retval += timerinfo[i] + sumtimerinfo[i];
#ifdef WITHSORTBOTS
    for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ )
        retval += timerinfo[i] + sumtimerinfo[i];
#endif
    return(retval);
```

```
}

/*
    #] GetWorkerTimes :
    #[ UpdateOneThread :
*/
int UpdateOneThread(int identity)
{
    ALLPRIVATES *B = AB[identity], *B0 = AB[0];
    AR.GetFile = AR0.GetFile;
    AR.KeptInHold = AR0.KeptInHold;
    AR.CurExpr = AR0.CurExpr;
    AR.SortType = AC.SortType;
    if ( AT.WildcardBufferSize < AC.WildcardBufferSize ) {
        M_free(AT.WildArgTaken,"argument list names");
        AT.WildcardBufferSize = AC.WildcardBufferSize;
        AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
                ,"argument list names");
        if ( AT.WildArgTaken == 0 ) return(-1);
    }
    return(0);
}

/*
    #] UpdateOneThread :
    #[ LoadOneThread :
*/
int LoadOneThread(int from, int identity, THREADBUCKET *thr, int par)
{
    WORD *t1, *t2;
    ALLPRIVATES *B = AB[identity], *B0 = AB[from];

    AR.DefPosition = AR0.DefPosition;
    AR.NoCompress = AR0.NoCompress;
    AR.gzipCompress = AR0.gzipCompress;
    AR.BracketOn = AR0.BracketOn;
    AR.CurDum = AR0.CurDum;
    AR.DeferFlag = AR0.DeferFlag;
    AR.TePos = 0;
    AR.sLevel = AR0.sLevel;
    AR.Stage4Name = AR0.Stage4Name;
    AR.GetOneFile = AR0.GetOneFile;
    AR.PolyFun = AR0.PolyFun;
    AR.PolyFunInv = AR0.PolyFunInv;
    AR.PolyFunType = AR0.PolyFunType;
    AR.PolyFunExp = AR0.PolyFunExp;
    AR.PolyFunVar = AR0.PolyFunVar;
    AR.PolyFunPow = AR0.PolyFunPow;
    AR.Eside = AR0.Eside;
    AR.Cnumlhs = AR0.Cnumlhs;
/*
    AR.MaxBracket = AR0.MaxBracket;

    The compressbuffer contents are mainly relevant for keep brackets
    We should do this only if there is a keep brackets statement
    We may however still need the compressbuffer for expressions in the rhs.
*/
    if ( par >= 1 ) {
/*
        We may not need this %%%%% 7-apr-2006
*/
        t1 = AR.CompressBuffer; t2 = AR0.CompressBuffer;
        while ( t2 < AR0.CompressPointer ) *t1++ = *t2++;
        AR.CompressPointer = t1;

    }
    else {
        AR.CompressPointer = AR.CompressBuffer;
    }
    if ( AR.DeferFlag ) {
        if ( AR.infile->handle < 0 ) {
            AR.infile->POfill = AR0.infile->POfill;
        }
        else {
/*
            We have to set the value of POposition to something that will
            force a read in the first try.
*/
            AR.infile->POfull = AR.infile->POfill = AR.infile->PObuffer;
        }
    }
    if ( par == 0 ) {
        AN.threadbuck = thr;
        AN.ninterms = thr->firstterm;
    }
    else if ( par == 1 ) {
        WORD *tstop;
```

```
        t1 = thr->threadbuffer; tstop = t1 + *t1;
        t2 = AT.WorkPointer;
        while ( t1 < tstop ) *t2++ = *t1++;
        AN.ninterms = thr->firstterm;
    }
    AN.TeInFun = 0;
    AN.ncmod = AC.ncmod;
    AT.BrackBuf = AT0.BrackBuf;
    AT.bracketindexflag = AT0.bracketindexflag;
    AN.PolyFunTodo = 0;
/*
    The relevant variables and the term are in their place.
    There is nothing more to do.
*/
    return(0);
}

/*
    #] LoadOneThread :
    #[ BalanceRunThread :
*/
int BalanceRunThread(PHEAD int identity, WORD *term, WORD level)
{
    GETBIDENTITY
    ALLPRIVATES *BB;
    WORD *t, *tt;
    int i, *ti, *tti;

    LoadOneThread(AT.identity,identity,0,2);
/*
    Extra loading if needed. Quantities changed in Generator.
    Like the level that has to be passed.
*/
    BB = AB[identity];
    BB->R.level = level;
    BB->T.TMbuff = AT.TMbuff;
    ti = AT.RepCount; tti = BB->T.RepCount;
    i = AN.RepPoint - AT.RepCount;
    BB->N.RepPoint = BB->T.RepCount + i;
    for ( ; i >= 0; i-- ) tti[i] = ti[i];

    t = term; i = *term;
    tt = BB->T.WorkSpace;
    NCOPY(tt,t,i);
    BB->T.WorkPointer = tt;

    WakeupThread(identity,HIGHERLEVELGENERATION);

    return(0);
}

/*
    #] BalanceRunThread :
    #[ SetWorkerFiles :
*/
void SetWorkerFiles()
{
    int id;
    ALLPRIVATES *B, *B0 = AB[0];
    for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
        B = AB[id];
        AR.infile = &(AR.Fscr[0]);
        AR.outfile = &(AR.Fscr[1]);
        AR.hidefile = &(AR.Fscr[2]);
        AR.infile->handle = AR0.infile->handle;
        AR.hidefile->handle = AR0.hidefile->handle;
        if ( AR.infile->handle < 0 ) {
            AR.infile->PObuffer = AR0.infile->PObuffer;
            AR.infile->POstop = AR0.infile->POstop;
            AR.infile->POfill = AR0.infile->POfill;
            AR.infile->POfull = AR0.infile->POfull;
            AR.infile->POsize = AR0.infile->POsize;
            AR.InInBuf = AR0.InInBuf;
            AR.infile->POposition = AR0.infile->POposition;
            AR.infile->filesize = AR0.infile->filesize;
        }
        else {
            AR.infile->PObuffer = AR.infile->wPObuffer;
            AR.infile->POstop = AR.infile->wPOstop;
            AR.infile->POfill = AR.infile->wPOfill;
            AR.infile->POfull = AR.infile->wPOfull;
            AR.infile->POsize = AR.infile->wPOsize;
            AR.InInBuf = 0;
            PUTZERO(AR.infile->POposition);
        }
/*
        If there is some writing, it betters happens to ones own outfile.
```

```
        Currently this is to be done only for InParallel.
        Merging of the outputs is then done by the CopyExpression routine.
*/
        {
            AR.outfile->PObuffer = AR.outfile->wPObuffer;
            AR.outfile->POstop = AR.outfile->wPOstop;
            AR.outfile->POfill = AR.outfile->wPOfill;
            AR.outfile->POfull = AR.outfile->wPOfull;
            AR.outfile->POsize = AR.outfile->wPOsize;
            PUTZERO(AR.outfile->POposition);
        }
        if ( AR.hidefile->handle < 0 ) {
            AR.hidefile->PObuffer = AR0.hidefile->PObuffer;
            AR.hidefile->POstop = AR0.hidefile->POstop;
            AR.hidefile->POfill = AR0.hidefile->POfill;
            AR.hidefile->POfull = AR0.hidefile->POfull;
            AR.hidefile->POsize = AR0.hidefile->POsize;
            AR.InHiBuf = AR0.InHiBuf;
            AR.hidefile->POposition = AR0.hidefile->POposition;
            AR.hidefile->filesize = AR0.hidefile->filesize;
        }
        else {
            AR.hidefile->PObuffer = AR.hidefile->wPObuffer;
            AR.hidefile->POstop = AR.hidefile->wPOstop;
            AR.hidefile->POfill = AR.hidefile->wPOfill;
            AR.hidefile->POfull = AR.hidefile->wPOfull;
            AR.hidefile->POsize = AR.hidefile->wPOsize;
            AR.InHiBuf = 0;
            PUTZERO(AR.hidefile->POposition);
        }
    }
    if ( AR0.StoreData.dirtyflag ) {
        for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
            B = AB[id];
            AR.StoreData = AR0.StoreData;
        }
    }
}

/*
    #] SetWorkerFiles :
    #[ RunThread :
*/
void *RunThread(void *dummy)
{
    WORD *term, *ttin, *tt, *ttco, *oldwork;
    int identity, wakeupsignal, identityretv, i, tobereleased, errorcode;
    ALLPRIVATES *B;
    THREADBUCKET *thr;
    POSITION *ppdef;
    EXPRESSIONS e;
    DUMMYUSE(dummy);
    identity = SetIdentity(&identityretv);
    threadpointers[identity] = pthread_self();
    B = InitializeOneThread(identity);
    while ( ( wakeupsignal = ThreadWait(identity) ) > 0 ) {
        switch ( wakeupsignal ) {
/*
            #[ STARTNEWEXPRESSION :
*/
            case STARTNEWEXPRESSION:
/*
                Set up the sort routines etc.
                Start with getting some buffers synchronized with the compiler
*/
                if ( UpdateOneThread(identity) ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Update error in starting expression in thread %d in module
%d",identity,AC.CModule);
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                AR.DeferFlag = AC.ComDefer;
                AR.sLevel = AS.sLevel;
                AR.MaxDum = AM.IndDum;
                AR.expchanged = AB[0]->R.expchanged;
                AR.expflags = AB[0]->R.expflags;
                AR.PolyFun = AB[0]->R.PolyFun;
                AR.PolyFunInv = AB[0]->R.PolyFunInv;
                AR.PolyFunType = AB[0]->R.PolyFunType;
                AR.PolyFunExp = AB[0]->R.PolyFunExp;
                AR.PolyFunVar = AB[0]->R.PolyFunVar;
                AR.PolyFunPow = AB[0]->R.PolyFunPow;
/*
                Now fire up the sort buffer.
*/
                NewSort(BHEAD0);
```

```
                break;
/*
            #] STARTNEWEXPRESSION :
            #[ LOWESTLEVELGENERATION :
*/
            case LOWESTLEVELGENERATION:
#ifdef INNERTEST
                if ( AC.InnerTest ) {
                    if ( StrCmp(AC.TestValue,(UBYTE *)INNERTEST) == 0 ) {
                        MesPrint("Testing(Worker%d): value = %s",AT.identity,AC.TestValue);
                    }
                }
#endif
                e = Expressions + AR.CurExpr;
                thr = AN.threadbuck;
                ppdef = thr->deferbuffer;
                ttin = thr->threadbuffer;
                ttco = thr->compressbuffer;
                term = AT.WorkPointer;
                thr->usenum = 0;
                tobereleased = 0;
                AN.inputnumber = thr->firstterm;
                AN.ninterms = thr->firstterm;
                do {
                  thr->usenum++;    /* For if the master wants to steal the bucket */
                  tt = term; i = *ttin;
                  NCOPY(tt,ttin,i);
                  AT.WorkPointer = tt;
                  if ( AR.DeferFlag ) {
                    tt = AR.CompressBuffer; i = *ttco;
                    NCOPY(tt,ttco,i);
                    AR.CompressPointer = tt;
                    AR.DefPosition = ppdef[0]; ppdef++;
                  }
                  if ( thr->free == BUCKETTERMINATED ) {
/*
                    The next statement allows the master to steal the bucket
                    for load balancing purposes. We do still execute the current
                    term, but afterwards we drop out.
                    Once we have written the release code, we cannot use this
                    bucket anymore. Hence the exit to the label bucketstolen.
*/
                    if ( thr->usenum == thr->totnum ) {
                        thr->free = BUCKETCOMINGFREE;
                    }
                    else {
                        thr->free = BUCKETRELEASED;
                        tobereleased = 1;
                    }
                  }
/*
                    What if we want to steal and we set thr->free while
                    the thread is inside the next code for a long time?
                  if ( AT.LoadBalancing ) {
*/
                    LOCK(thr->lock);
                    thr->busy = BUCKETDOINGTERM;
                    UNLOCK(thr->lock);
/*
                  }
                  else {
                    thr->busy = BUCKETDOINGTERM;
                  }
*/
                  AN.RepPoint = AT.RepCount + 1;

                  if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
                    StoreTerm(BHEAD term);
                  }
                  else {
                  if ( AR.DeferFlag ) {
                    AR.CurDum = AN.IndDum = Expressions[AR.CurExpr].numdummies + AM.IndDum;
                  }
                  else {
                    AN.IndDum = AM.IndDum;
                    AR.CurDum = ReNumber(BHEAD term);
                  }
                  if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
                  if ( AN.ncmod ) {
                    if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
                    else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
                  }
                  else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
                  if ( ( AP.PreDebug & THREADSDEBUG ) != 0 ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Thread %w executing term:");
                    PrintTerm(term,"LLG");
```

```
                    MUNLOCK(ErrorMessageLock);
                  }
                  if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
                        && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
                        PolyFunClean(BHEAD term);
                  }
                  if ( Generator(BHEAD term,0) ) {
                    LowerSortLevel();
                    MLOCK(ErrorMessageLock);
                    MesPrint("Error in processing one term in thread %d in module
%d",identity,AC.CModule);
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                  }
                  AN.ninterms++;
                  }
/*                if ( AT.LoadBalancing ) { */
                    LOCK(thr->lock);
                    thr->busy = BUCKETPREPARINGTERM;
                    UNLOCK(thr->lock);
/*
                  }
                  else {
                    thr->busy = BUCKETPREPARINGTERM;
                  }
*/
                  if ( thr->free == BUCKETTERMINATED ) {
                    if ( thr->usenum == thr->totnum ) {
                        thr->free = BUCKETCOMINGFREE;
                    }
                    else {
                        thr->free = BUCKETRELEASED;
                        tobereleased = 1;
                    }
                  }
                  if ( tobereleased ) goto bucketstolen;
                } while ( *ttin );
                thr->free = BUCKETCOMINGFREE;
bucketstolen:;
/*              if ( AT.LoadBalancing ) { */
                    LOCK(thr->lock);
                    thr->busy = BUCKETTOBERELEASED;
                    UNLOCK(thr->lock);
/*              }
                else {
                    thr->busy = BUCKETTOBERELEASED;
                }
*/
                AT.WorkPointer = term;
                break;
/*
            #] LOWESTLEVELGENERATION :
            #[ FINISHEXPRESSION :
*/
#ifdef WITHSORTBOTS
            case CLAIMOUTPUT:
                LOCK(AT.SB.MasterBlockLock[1]);
                break;
#endif
            case FINISHEXPRESSION:
/*
                Finish the sort

                Start with claiming the first block
                Once we have claimed it we can let the master know that
                everything is all right.
*/
                LOCK(AT.SB.MasterBlockLock[1]);
                ThreadClaimedBlock(identity);
/*
                Entry for when we work with sortbots
*/
#ifdef WITHSORTBOTS
            case FINISHEXPRESSION2:
#endif
/*
                Now we may need here an fsync on the sort file
*/
                if ( AC.ThreadSortFileSynch ) {
                  if ( AT.S0->file.handle >= 0 ) {
                    SynchFile(AT.S0->file.handle);
                  }
                }
                AT.SB.FillBlock = 1;
                AT.SB.MasterFill[1] = AT.SB.MasterStart[1];
                errorcode = EndSort(BHEAD AT.S0->sBuffer,0);
                UNLOCK(AT.SB.MasterBlockLock[AT.SB.FillBlock]);
```

```
                UpdateMaxSize();
                if ( errorcode ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Error terminating sort in thread %d in module %d",identity,AC.CModule);
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                break;
/*
            #] FINISHEXPRESSION :
            #[ CLEANUPEXPRESSION :
*/
            case CLEANUPEXPRESSION:
/*
                Cleanup everything and wait for the next expression
*/
                if ( AR.outfile->handle >= 0 ) {
                    CloseFile(AR.outfile->handle);
                    AR.outfile->handle = -1;
                    remove(AR.outfile->name);
                    AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
                    PUTZERO(AR.outfile->POposition);
                    PUTZERO(AR.outfile->filesize);
                }
                else {
                    AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
                    PUTZERO(AR.outfile->POposition);
                    PUTZERO(AR.outfile->filesize);
                }
                {
                    CBUF *C = cbuf+AT.ebufnum;
                    WORD **w, ii;
                    if ( C->numrhs > 0 || C->numlhs > 0 ) {
                        if ( C->rhs ) {
                            w = C->rhs; ii = C->numrhs;
                            do { *w++ = 0; } while ( --ii > 0 );
                        }
                        if ( C->lhs ) {
                            w = C->lhs; ii = C->numlhs;
                            do { *w++ = 0; } while ( --ii > 0 );
                        }
                        C->numlhs = C->numrhs = 0;
                        ClearTree(AT.ebufnum);
                        C->Pointer = C->Buffer;
                    }
                }
                break;
/*
            #] CLEANUPEXPRESSION :
            #[ HIGHERLEVELGENERATION :
*/
            case HIGHERLEVELGENERATION:
/*
                When foliating halfway the tree.
                This should only be needed in a second level load balancing
*/
                term = AT.WorkSpace; AT.WorkPointer = term + *term;
                if ( Generator(BHEAD term,AR.level) ) {
                    LowerSortLevel();
                    MLOCK(ErrorMessageLock);
                    MesPrint("Error in load balancing one term at level %d in thread %d in module
%d",AR.level,AT.identity,AC.CModule);
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                AT.WorkPointer = term;
                break;
/*
            #] HIGHERLEVELGENERATION :
            #[ STARTNEWMODULE :
*/
            case STARTNEWMODULE:
/*
                For resetting variables.
*/
                SpecialCleanup(B);
                break;
/*
            #] STARTNEWMODULE :
            #[ TERMINATETHREAD :
*/
            case TERMINATETHREAD:
                goto EndOfThread;
/*
            #] TERMINATETHREAD :
            #[ DOONEEXPRESSION :

```

```
                When a thread has to do a complete (not too big) expression.
                The number of the expression to be done is in AR.exprtodo.
                The code is mostly taken from Processor. The only difference
                is with what to do with the output.
                The output should go to the scratch buffer of the worker
                (which is free at the right moment). If this buffer is too
                small we have a problem. We could write to file or give the
                master what we have and from now on the master has to collect
                pieces until things are complete.
                Note: this assumes that the expressions don't keep their order.
                If they have to keep their order, don't use this feature.
*/
            case DOONEEXPRESSION: {

                POSITION position, outposition;
                FILEHANDLE *fi, *fout, *oldoutfile;
                LONG dd = 0;
                WORD oldBracketOn = AR.BracketOn;
                WORD *oldBrackBuf = AT.BrackBuf;
                WORD oldbracketindexflag = AT.bracketindexflag;
                WORD fromspectator = 0;
                e = Expressions + AR.exprtodo;
                i = AR.exprtodo;
                AR.CurExpr = i;
                AR.SortType = AC.SortType;
                AR.expchanged = 0;
                if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
                    AR.BracketOn = 1;
                    AT.BrackBuf = AM.BracketFactors;
                    AT.bracketindexflag = 1;
                }

                position = AS.OldOnFile[i];
                if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENGEXPRESSION ) {
                    AR.GetFile = 2; fi = AR.hidefile;
                }
                else {
                    AR.GetFile = 0; fi = AR.infile;
                }
/*
                PUTZERO(fi->POposition);
                if ( fi->handle >= 0 ) {
                    fi->POfill = fi->POfull = fi->PObuffer;
                }
*/
                SetScratch(fi,&position);
                term = oldwork = AT.WorkPointer;
                AR.CompressPointer = AR.CompressBuffer;
                AR.CompressPointer[0] = 0;
                AR.KeptInHold = 0;
                if ( GetTerm(BHEAD term) <= 0 ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Expression %d has problems in scratchfile (t)",i);
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
                term[3] = i;
                if ( term[5] < 0 ) {
                    fromspectator = -term[5];
                    PUTZERO(AM.SpectatorFiles[fromspectator-1].readpos);
                    term[5] = AC.cbufnum;
                }
                PUTZERO(outposition);
                fout = AR.outfile;
                fout->POfill = fout->POfull = fout->PObuffer;
                fout->POposition = outposition;
                if ( fout->handle >= 0 ) {
                    fout->POposition = outposition;
                }
/*
                The next statement is needed because we need the system
                to believe that the expression is at position zero for
                the moment. In this worker, with no memory of other expressions,
                it is. This is needed for when a bracket index is made
                because there e->onfile is an offset. Afterwards, when the
                expression is written to its final location in the masters
                output e->onfile will get its real value.
*/
                PUTZERO(e->onfile);
                if ( PutOut(BHEAD term,&outposition,fout,0) < 0 ) goto ProcErr;

                AR.DeferFlag = AC.ComDefer;

                AR.sLevel = AB[0]->R.sLevel;
                term = AT.WorkPointer;
                NewSort(BHEAD0);
```

```
                AR.MaxDum = AM.IndDum;
                AN.ninterms = 0;
                if ( fromspectator ) {
                 while ( GetFromSpectator(term,fromspectator-1) ) {
                  AT.WorkPointer = term + *term;
                  AN.RepPoint = AT.RepCount + 1;
                  AN.IndDum = AM.IndDum;
                  AR.CurDum = ReNumber(BHEAD term);
                  if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
                  if ( AN.ncmod ) {
                    if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
                    else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
                  }
                  else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
                  if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
                        && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
                        PolyFunClean(BHEAD term);
                  }
                  if ( Generator(BHEAD term,0) ) {
                    LowerSortLevel(); goto ProcErr;
                  }
                 }
                }
                else {
                 while ( GetTerm(BHEAD term) ) {
                  SeekScratch(fi,&position);
                  AN.ninterms++; dd = AN.deferskipped;
                  if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
                      StoreTerm(BHEAD term);
                  }
                  else {
                  if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)sizeof(WORD))) ) {
                    if ( GetMoreTerms(term) < 0 ) {
                      LowerSortLevel(); goto ProcErr;
                    }
                    SeekScratch(fi,&position);
                  }
                  AT.WorkPointer = term + *term;
                  AN.RepPoint = AT.RepCount + 1;
                  if ( AR.DeferFlag ) {
                    AR.CurDum = AN.IndDum = Expressions[AR.exprtodo].numdummies;
                  }
                  else {
                    AN.IndDum = AM.IndDum;
                    AR.CurDum = ReNumber(BHEAD term);
                  }
                  if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
                  if ( AN.ncmod ) {
                    if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
                    else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
                  }
                  else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
                  if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
                        && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
                        PolyFunClean(BHEAD term);
                  }
                  if ( Generator(BHEAD term,0) ) {
                    LowerSortLevel(); goto ProcErr;
                  }
                  AN.ninterms += dd;
                  }
                  SetScratch(fi,&position);
                  if ( fi == AR.hidefile ) {
                    AR.InHiBuf = (fi->POfull-fi->PObuffer)
                        -DIFBASE(position,fi->POposition)/sizeof(WORD);
                  }
                  else {
                    AR.InInBuf = (fi->POfull-fi->PObuffer)
                        -DIFBASE(position,fi->POposition)/sizeof(WORD);
                  }
                 }
                }
                AN.ninterms += dd;
                if ( EndSort(BHEAD AT.S0->sBuffer,0) < 0 ) goto ProcErr;
                e->numdummies = AR.MaxDum - AM.IndDum;
                AR.BracketOn = oldBracketOn;
                AT.BrackBuf = oldBrackBuf;
                if ( ( e->vflags & TOBEFACTORED ) != 0 )
                        poly_factorize_expression(e);
                else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
                 && ( ( e->vflags & ISFACTORIZED ) != 0 ) )
                        poly_unfactorize_expression(e);
                if ( AT.S0->TermsLeft )   e->vflags &= ~ISZERO;
                else                      e->vflags |= ISZERO;
                if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
                if ( AT.S0->TermsLeft ) AR.expflags |= ISZERO;
                if ( AR.expchanged )    AR.expflags |= ISUNMODIFIED;
```

```
                AR.GetFile = 0;
                AT.bracketindexflag = oldbracketindexflag;
/*
                Now copy the whole thing from fout to AR0.outfile
                Do this in one go to keep the lock occupied as short as possible
*/
                SeekScratch(fout,&outposition);
                LOCK(AS.outputslock);
                oldoutfile = AB[0]->R.outfile;
                if ( e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION ) {
                    AB[0]->R.outfile = AB[0]->R.hidefile;
                }
                SeekScratch(AB[0]->R.outfile,&position);
                e->onfile = position;
                if ( CopyExpression(fout,AB[0]->R.outfile) < 0 ) {
                    AB[0]->R.outfile = oldoutfile;
                    UNLOCK(AS.outputslock);
                    MLOCK(ErrorMessageLock);
                    MesPrint("Error copying output of 'InParallel' expression to master. Thread: %d",identity);
                    MUNLOCK(ErrorMessageLock);
                    goto ProcErr;
                }
                AB[0]->R.outfile = oldoutfile;
                AB[0]->R.hidefile->POfull = AB[0]->R.hidefile->POfill;
                AB[0]->R.expflags = AR.expflags;
                UNLOCK(AS.outputslock);

                if ( fout->handle >= 0 ) {    /* Now get rid of the file */
                    CloseFile(fout->handle);
                    fout->handle = -1;
                    remove(fout->name);
                    PUTZERO(fout->POposition);
                    PUTZERO(fout->filesize);
                    fout->POfill = fout->POfull = fout->PObuffer;
                }
                UpdateMaxSize();

                AT.WorkPointer = oldwork;

                } break;
/*
            #] DOONEEXPRESSION :
            #[ DOBRACKETS :

                In case we have a bracket index we can have the worker treat
                one or more of the entries in the bracket index.
                The advantage is that identical terms will meet each other
                sooner in the sorting and hence fewer compares will be needed.
                Also this way the master doesn't need to fill the buckets.
                The main problem is the load balancing which can become very
                bad when there is a long tail without things outside the bracket.

                We get sent:
                1: The number of the first bracket to be done
                2: The number of the last bracket to be done
*/
            case DOBRACKETS: {
                BRACKETINFO *binfo;
                BRACKETINDEX *bi;
                FILEHANDLE *fi;
                POSITION stoppos,where;
                e = Expressions + AR.CurExpr;
                binfo = e->bracketinfo;
                thr = AN.threadbuck;
                bi = &(binfo->indexbuffer[thr->firstbracket]);
                if ( AR.GetFile == 2 ) fi = AR.hidefile;
                else                   fi = AR.infile;
                where = bi->start;
                ADD2POS(where,AS.OldOnFile[AR.CurExpr]);
                SetScratch(fi,&(where));
                stoppos = binfo->indexbuffer[thr->lastbracket].next;
                ADD2POS(stoppos,AS.OldOnFile[AR.CurExpr]);
                AN.ninterms = thr->firstterm;
/*
                Now we have to put the 'value' of the bracket in the
                Compress buffer.
*/
                ttco = AR.CompressBuffer;
                tt = binfo->bracketbuffer + bi->bracket;
                i = *tt;
                NCOPY(ttco,tt,i)
                AR.CompressPointer = ttco;
                term = AT.WorkPointer;
                while ( GetTerm(BHEAD term) ) {
                    SeekScratch(fi,&where);
                    AT.WorkPointer = term + *term;
```

```
                    AN.IndDum = AM.IndDum;
                    AR.CurDum = ReNumber(BHEAD term);
                    if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
                    if ( AN.ncmod ) {
                        if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
                        else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
                    }
                    else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
                    if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
                        && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
                        PolyFunClean(BHEAD term);
                    }
                    if ( ( AP.PreDebug & THREADSDEBUG ) != 0 ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Thread %w executing term:");
                        PrintTerm(term,"DoBrackets");
                        MUNLOCK(ErrorMessageLock);
                    }
                    AT.WorkPointer = term + *term;
                    if ( Generator(BHEAD term,0) ) {
                        LowerSortLevel();
                        MLOCK(ErrorMessageLock);
                        MesPrint("Error in processing one term in thread %d in module
%d",identity,AC.CModule);
                        MUNLOCK(ErrorMessageLock);
                        Terminate(-1);
                    }
                    AN.ninterms++;
                    SetScratch(fi,&(where));
                    if ( ISGEPOS(where,stoppos) ) break;
                }
                AT.WorkPointer = term;
                thr->free = BUCKETCOMINGFREE;
                break;
            }
/*
            #] DOBRACKETS :
            #[ CLEARCLOCK :

            The program only comes here after a .clear
*/
            case CLEARCLOCK:
/*              LOCK(clearclocklock); */
                sumtimerinfo[identity] += TimeCPU(1);
                timerinfo[identity] = TimeCPU(0);
/*              UNLOCK(clearclocklock); */
                break;
/*
            #] CLEARCLOCK :
            #[ MCTSEXPANDTREE :
*/
            case MCTSEXPANDTREE:
                AT.optimtimes = AB[0]->T.optimtimes;
                find_Horner_MCTS_expand_tree();
                break;
/*
            #] MCTSEXPANDTREE :
            #[ OPTIMIZEEXPRESSION :
*/
            case OPTIMIZEEXPRESSION:
                optimize_expression_given_Horner();
                break;
/*
            #] OPTIMIZEEXPRESSION :
*/
            default:
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal wakeup signal %d for thread %d",wakeupsignal,identity);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
                break;
        }
        /* we need the following update in case we are using checkpoints. then we
           need to readjust the clocks when recovering using this information */
        timerinfo[identity] = TimeCPU(1);
    }
EndOfThread:;
/*
    This is the end of the thread. We cleanup and exit.
*/
    FinalizeOneThread(identity);
    return(0);
ProcErr:
    Terminate(-1);
    return(0);
}
```

```
/*
    #] RunThread :
    #[ RunSortBot :
*/
#ifdef WITHSORTBOTS

void *RunSortBot(void *dummy)
{
    int identity, wakeupsignal, identityretv;
    ALLPRIVATES *B, *BB;
    DUMMYUSE(dummy);
    identity = SetIdentity(&identityretv);
    threadpointers[identity] = pthread_self();
    B = InitializeOneThread(identity);
    while ( ( wakeupsignal = SortBotWait(identity) ) > 0 ) {
        switch ( wakeupsignal ) {
/*
            #[ INISORTBOT :
*/
            case INISORTBOT:
                AR.CurExpr = AB[0]->R.CurExpr;
                AR.PolyFun = AB[0]->R.PolyFun;
                AR.PolyFunInv = AB[0]->R.PolyFunInv;
                AR.PolyFunType = AB[0]->R.PolyFunType;
                AR.PolyFunExp = AB[0]->R.PolyFunExp;
                AR.PolyFunVar = AB[0]->R.PolyFunVar;
                AR.PolyFunPow = AB[0]->R.PolyFunPow;
                AR.SortType = AC.SortType;
                if ( AR.PolyFun == 0 ) { AT.SS->PolyFlag = 0; }
                else if ( AR.PolyFunType == 1 ) { AT.SS->PolyFlag = 1; }
                else if ( AR.PolyFunType == 2 ) {
                    if ( AR.PolyFunExp == 2
                      || AR.PolyFunExp == 3 ) AT.SS->PolyFlag = 1;
                    else                      AT.SS->PolyFlag = 2;
                }
                AT.SS->PolyWise = 0;
                AN.ncmod = AC.ncmod;
                LOCK(AT.SB.MasterBlockLock[1]);
                BB = AB[AT.SortBotIn1];
                LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
                BB = AB[AT.SortBotIn2];
                LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
                AT.SB.FillBlock = 1;
                AT.SB.MasterFill[1] = AT.SB.MasterStart[1];
                SETBASEPOSITION(AN.theposition,0);
                break;
/*
            #] INISORTBOT :
            #[ RUNSORTBOT :
*/
            case RUNSORTBOT:
                SortBotMerge(B);
                break;
/*
            #] RUNSORTBOT :
            #[ TERMINATETHREAD :
*/
            case TERMINATETHREAD:
                goto EndOfThread;
/*
            #] TERMINATETHREAD :
            #[ CLEARCLOCK :

            The program only comes here after a .clear
*/
            case CLEARCLOCK:
/*              LOCK(clearclocklock); */
                sumtimerinfo[identity] += TimeCPU(1);
                timerinfo[identity] = TimeCPU(0);
/*              UNLOCK(clearclocklock); */
                break;
/*
            #] CLEARCLOCK :
*/
            default:
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal wakeup signal %d for thread %d",wakeupsignal,identity);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
                break;
        }
    }
EndOfThread:;
/*
    This is the end of the thread. We cleanup and exit.
*/
    FinalizeOneThread(identity);
```

```
    return(0);
}

#endif

/*
    #] RunSortBot :
    #[ IAmAvailable :
*/
void IAmAvailable(int identity)
{
    int top;
    LOCK(availabilitylock);
    top = topofavailables;
    listofavailables[topofavailables++] = identity;
    if ( top == 0 ) {
        UNLOCK(availabilitylock);
        LOCK(wakeupmasterlock);
        wakeupmaster = identity;
        pthread_cond_signal(&wakeupmasterconditions);
        UNLOCK(wakeupmasterlock);
    }
    else {
        UNLOCK(availabilitylock);
    }
}

/*
    #] IAmAvailable :
    #[ GetAvailableThread :
*/
int GetAvailableThread()
{
    int retval = -1;
    LOCK(availabilitylock);
    if ( topofavailables > 0 ) retval = listofavailables[--topofavailables];
    UNLOCK(availabilitylock);
    if ( retval >= 0 ) {
/*
        Make sure the thread is indeed waiting and not between
        saying that it is available and starting to wait.
*/
        LOCK(wakeuplocks[retval]);
        UNLOCK(wakeuplocks[retval]);
    }
    return(retval);
}

/*
    #] GetAvailableThread :
    #[ ConditionalGetAvailableThread :
*/
int ConditionalGetAvailableThread()
{
    int retval = -1;
    if ( topofavailables > 0 ) {
        LOCK(availabilitylock);
        if ( topofavailables > 0 ) {
            retval = listofavailables[--topofavailables];
        }
        UNLOCK(availabilitylock);
        if ( retval >= 0 ) {
/*
            Make sure the thread is indeed waiting and not between
            saying that it is available and starting to wait.
*/
            LOCK(wakeuplocks[retval]);
            UNLOCK(wakeuplocks[retval]);
        }
    }
    return(retval);
}

/*
    #] ConditionalGetAvailableThread :
    #[ GetThread :
*/
int GetThread(int identity)
{
    int retval = -1, j;
    LOCK(availabilitylock);
    for ( j = 0; j < topofavailables; j++ ) {
        if ( identity == listofavailables[j] ) break;
    }
    if ( j < topofavailables ) {
        --topofavailables;
        for ( ; j < topofavailables; j++ ) {
```

```
            listofavailables[j] = listofavailables[j+1];
        }
        retval = identity;
    }
    UNLOCK(availabilitylock);
    return(retval);
}

/*
    #] GetThread :
    #[ ThreadWait :
*/
int ThreadWait(int identity)
{
    int retval, top, j;
    LOCK(wakeuplocks[identity]);
    LOCK(availabilitylock);
    top = topofavailables;
    for ( j = topofavailables; j > 0; j-- )
        listofavailables[j] = listofavailables[j-1];
    listofavailables[0] = identity;
    topofavailables++;
    if ( top == 0 || topofavailables == numberofworkers ) {
        UNLOCK(availabilitylock);
        LOCK(wakeupmasterlock);
        wakeupmaster = identity;
        pthread_cond_signal(&wakeupmasterconditions);
        UNLOCK(wakeupmasterlock);
    }
    else {
        UNLOCK(availabilitylock);
    }
    while ( wakeup[identity] == 0 ) {
        pthread_cond_wait(&(wakeupconditions[identity]),&(wakeuplocks[identity]));
    }
    retval = wakeup[identity];
    wakeup[identity] = 0;
    UNLOCK(wakeuplocks[identity]);
    return(retval);
}

/*
    #] ThreadWait :
    #[ SortBotWait :
*/

#ifdef WITHSORTBOTS
int SortBotWait(int identity)
{
    int retval;
    LOCK(wakeuplocks[identity]);
    LOCK(availabilitylock);
    topsortbotavailables++;
    if ( topsortbotavailables >= numberofsortbots ) {
        UNLOCK(availabilitylock);
        LOCK(wakeupsortbotlock);
        wakeupmaster = identity;
        pthread_cond_signal(&wakeupsortbotconditions);
        UNLOCK(wakeupsortbotlock);
    }
    else {
        UNLOCK(availabilitylock);
    }
    while ( wakeup[identity] == 0 ) {
        pthread_cond_wait(&(wakeupconditions[identity]),&(wakeuplocks[identity]));
    }
    retval = wakeup[identity];
    wakeup[identity] = 0;
    UNLOCK(wakeuplocks[identity]);
    return(retval);
}

#endif

/*
    #] SortBotWait :
    #[ ThreadClaimedBlock :
*/
int ThreadClaimedBlock(int identity)
{
    LOCK(availabilitylock);
    numberclaimed++;
    if ( numberclaimed >= numberofworkers ) {
        UNLOCK(availabilitylock);
        LOCK(wakeupmasterlock);
        wakeupmaster = identity;
        pthread_cond_signal(&wakeupmasterconditions);
```

```
        UNLOCK(wakeupmasterlock);
    }
    else {
        UNLOCK(availabilitylock);
    }
    return(0);
}

/*
    #] ThreadClaimedBlock :
    #[ MasterWait :
*/
int MasterWait()
{
    int retval;
    LOCK(wakeupmasterlock);
    while ( wakeupmaster == 0 ) {
        pthread_cond_wait(&wakeupmasterconditions,&wakeupmasterlock);
    }
    retval = wakeupmaster;
    wakeupmaster = 0;
    UNLOCK(wakeupmasterlock);
    return(retval);
}

/*
    #] MasterWait :
    #[ MasterWaitThread :
*/
int MasterWaitThread(int identity)
{
    int retval;
    LOCK(wakeupmasterthreadlocks[identity]);
    while ( wakeupmasterthread[identity] == 0 ) {
        pthread_cond_wait(&(wakeupmasterthreadconditions[identity])
                ,&(wakeupmasterthreadlocks[identity]));
    }
    retval = wakeupmasterthread[identity];
    wakeupmasterthread[identity] = 0;
    UNLOCK(wakeupmasterthreadlocks[identity]);
    return(retval);
}

/*
    #] MasterWaitThread :
    #[ MasterWaitAll :
*/
void MasterWaitAll()
{
    LOCK(wakeupmasterlock);
    while ( topofavailables < numberofworkers ) {
        pthread_cond_wait(&wakeupmasterconditions,&wakeupmasterlock);
    }
    UNLOCK(wakeupmasterlock);
    return;
}

/*
    #] MasterWaitAll :
    #[ MasterWaitAllSortBots :
*/

#ifdef WITHSORTBOTS

void MasterWaitAllSortBots()
{
    LOCK(wakeupsortbotlock);
    while ( topsortbotavailables < numberofsortbots ) {
        pthread_cond_wait(&wakeupsortbotconditions,&wakeupsortbotlock);
    }
    UNLOCK(wakeupsortbotlock);
    return;
}

#endif

/*
    #] MasterWaitAllSortBots :
    #[ MasterWaitAllBlocks :
*/
void MasterWaitAllBlocks()
{
    LOCK(wakeupmasterlock);
    while ( numberclaimed < numberofworkers ) {
        pthread_cond_wait(&wakeupmasterconditions,&wakeupmasterlock);
    }
    UNLOCK(wakeupmasterlock);
```

```
    return;
}

/*
    #] MasterWaitAllBlocks :
    #[ WakeupThread :
*/
void WakeupThread(int identity, int signalnumber)
{
    if ( signalnumber == 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal wakeup signal for thread %d",identity);
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    LOCK(wakeuplocks[identity]);
    wakeup[identity] = signalnumber;
    pthread_cond_signal(&(wakeupconditions[identity]));
    UNLOCK(wakeuplocks[identity]);
}

/*
    #] WakeupThread :
    #[ WakeupMasterFromThread :
*/
void WakeupMasterFromThread(int identity, int signalnumber)
{
    if ( signalnumber == 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal wakeup signal for master %d",identity);
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    LOCK(wakeupmasterthreadlocks[identity]);
    wakeupmasterthread[identity] = signalnumber;
    pthread_cond_signal(&(wakeupmasterthreadconditions[identity]));
    UNLOCK(wakeupmasterthreadlocks[identity]);
}

/*
    #] WakeupMasterFromThread :
    #[ SendOneBucket :
*/
int SendOneBucket(int type)
{
    ALLPRIVATES *B0 = AB[0];
    THREADBUCKET *thr = 0;
    int j, k, id;
    for ( j = 0; j < numthreadbuckets; j++ ) {
        if ( threadbuckets[j]->free == BUCKETFILLED ) {
            thr = threadbuckets[j];
            for ( k = j+1; k < numthreadbuckets; k++ )
                threadbuckets[k-1] = threadbuckets[k];
            threadbuckets[numthreadbuckets-1] = thr;
            break;
        }
    }
    AN0.ninterms++;
    while ( ( id = GetAvailableThread() ) < 0 ) { MasterWait(); }
/*
    Prepare the thread. Give it the term and variables.
*/
    LoadOneThread(0,id,thr,0);
    thr->busy = BUCKETASSIGNED;
    thr->free = BUCKETINUSE;
    numberoffullbuckets--;
/*
    And signal the thread to run.
    Form now on we may only interfere with this bucket
    1: after it has been marked BUCKETCOMINGFREE
    2: when thr->busy == BUCKETDOINGTERM and then only when protected by
       thr->lock. This would be for load balancing.
*/
    WakeupThread(id,type);
/*  AN0.ninterms += thr->ddterms; */
    return(0);
}

/*
    #] SendOneBucket :
    #[ InParallelProcessor :
*/
int InParallelProcessor()
{
    GETIDENTITY
    int i, id, retval = 0, num = 0;
    EXPRESSIONS e;
```

```
    if ( numberofworkers >= 2 ) {
        SetWorkerFiles();
        for ( i = 0; i < NumExpressions; i++ ) {
            e = Expressions+i;
            if ( e->partodo <= 0 ) continue;
            if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
            || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
            || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION ) {
            }
            else {
                e->partodo = 0;
                continue;
            }
            if ( e->counter == 0 ) { /* Expression with zero terms */
                e->partodo = 0;
                continue;
            }
/*
            This expression should go to an idle worker
*/
            while ( ( id = GetAvailableThread() ) < 0 ) { MasterWait(); }
            LoadOneThread(0,id,0,-1);
            AB[id]->R.exprtodo = i;
            WakeupThread(id,DOONEEXPRESSION);
            num++;
        }
/*
        Now we have to wait for all workers to finish
*/
        if ( num > 0 ) MasterWaitAll();

        if ( AC.CollectFun ) AR.DeferFlag = 0;
    }
    else {
        for ( i = 0; i < NumExpressions; i++ ) {
            Expressions[i].partodo = 0;
        }
    }
    return(retval);
}

/*
    #] InParallelProcessor :
    #[ ThreadsProcessor :
*/
int ThreadsProcessor(EXPRESSIONS e, WORD LastExpression, WORD fromspectator)
{
    ALLPRIVATES *B0 = AB[0], *B = B0;
    int id, oldgzipCompress, endofinput = 0, j, still, k, defcount = 0, bra = 0, first = 1;
    LONG dd = 0, ddd, thrbufsiz, thrbufsiz0, thrbufsiz2, numbucket = 0, numpasses;
    LONG num, i;
    WORD *oldworkpointer = AT0.WorkPointer, *tt, *ttco = 0, *t1 = 0, ter, *tstop = 0, *t2;
    THREADBUCKET *thr = 0;
    FILEHANDLE *oldoutfile = AR0.outfile;
    GETTERM GetTermP = &GetTerm;
    POSITION eonfile = AS.OldOnFile[e-Expressions];
    numberoffullbuckets = 0;
/*
    Start up all threads. The lock needs to be around the whole loop
    to keep processes from terminating quickly and putting themselves
    in the list of available threads again.
*/
    AM.tracebackflag = 1;

    AS.sLevel = AR0.sLevel;
    LOCK(availabilitylock);
    topofavailables = 0;
    for ( id = 1; id <= numberofworkers; id++ ) {
        WakeupThread(id,STARTNEWEXPRESSION);
    }
    UNLOCK(availabilitylock);
    NewSort(BHEAD0);
    AN0.ninterms = 1;
/*
    Now for redefine
*/
    if ( AC.numpfirstnum > 0 ) {
        for ( j = 0; j < AC.numpfirstnum; j++ ) {
            AC.inputnumbers[j] = -1;
        }
    }
    MasterWaitAll();
/*
    Determine a reasonable bucketsize.
    This is based on the value of AC.ThreadBucketSize and the number
    of terms. We want at least 5 buckets per worker at the moment.
    Some research should show whether this is reasonable.
```

```
    The number of terms in the expression is in e->counter
*/
    thrbufsiz2 = thrbufsiz = AC.ThreadBucketSize-1;
    if ( ( e->counter / ( numberofworkers * 5 ) ) < thrbufsiz ) {
        thrbufsiz = e->counter / ( numberofworkers * 5 ) - 1;
        if ( thrbufsiz < 0 ) thrbufsiz = 0;
    }
    thrbufsiz0 = thrbufsiz;
    numpasses = 5; /* this is just for trying */
    thrbufsiz = thrbufsiz0 / (2 « numpasses);
/*
    Mark all buckets as free and take the first.
*/
    for ( j = 0; j < numthreadbuckets; j++ )
        threadbuckets[j]->free = BUCKETFREE;
    thr = threadbuckets[0];
/*
    #[ Whole brackets :

    First we look whether we have to work with entire brackets
    This is the case when there is a non-NULL address in e->bracketinfo.
    Of course we shouldn't have interference from a collect or keep statement.
*/
#ifdef WHOLEBRACKETS
    if ( e->bracketinfo && AC.CollectFun == 0 && AR0.DeferFlag == 0 ) {
        FILEHANDLE *curfile;
        int didone = 0;
        LONG num, n;
        AN0.expr = e;
        for ( n = 0; n < e->bracketinfo->indexfill; n++ ) {
            num = TreatIndexEntry(B0,n);
            if ( num > 0 ) {
                didone = 1;
/*
                This bracket can be sent off.
                1: Look for an empty bucket
*/
ReTry:;
                for ( j = 0; j < numthreadbuckets; j++ ) {
                    switch ( threadbuckets[j]->free ) {
                        case BUCKETFREE:
                            thr = threadbuckets[j];
                            goto Found1;
                        case BUCKETCOMINGFREE:
                            thr = threadbuckets[j];
                            thr->free = BUCKETFREE;
                            for ( k = j+1; k < numthreadbuckets; k++ )
                                threadbuckets[k-1] = threadbuckets[k];
                            threadbuckets[numthreadbuckets-1] = thr;
                            j--;
                            break;
                        default:
                            break;
                    }
                }
Found1:;
                if ( j < numthreadbuckets ) {
/*
                    Found an empty bucket. Fill it.
*/
                    thr->firstbracket = n;
                    thr->lastbracket = n + num - 1;
                    thr->type = BUCKETDOINGBRACKET;
                    thr->free = BUCKETFILLED;
                    thr->firstterm = AN0.ninterms;
                    for ( j = n; j < n+num; j++ ) {
                        AN0.ninterms += e->bracketinfo->indexbuffer[j].termsinbracket;
                    }
                    n += num-1;
                    numberoffullbuckets++;
                    if ( topofavailables > 0 ) {
                        SendOneBucket(DOBRACKETS);
                    }
                }
/*
                    All buckets are in use.
                    Look/wait for an idle worker. Give it a bucket.
                    After that, retry for a bucket
*/
                else {
                    while ( topofavailables <= 0 ) {
                        MasterWait();
                    }
                    SendOneBucket(DOBRACKETS);
                    goto ReTry;
                }
```

```
            }
        }
        if ( didone ) {
/*
            And now put the input back in the original position.
*/
            switch ( e->status ) {
                case UNHIDELEXPRESSION:
                case UNHIDEGEXPRESSION:
                case DROPHLEXPRESSION:
                case DROPHGEXPRESSION:
                case HIDDENLEXPRESSION:
                case HIDDENGEXPRESSION:
                    curfile = AR0.hidefile;
                    break;
                default:
                    curfile = AR0.infile;
                    break;
            }
            SetScratch(curfile,&eonfile);
            GetTerm(B0,AT0.WorkPointer);
/*
            Now we point the GetTerm that is used to the one that is selective
*/
            GetTermP = &GetTerm2;
/*
            Next wait till there is a bucket available and initialize thr to it.
*/
            for(;;) {
                for ( j = 0; j < numthreadbuckets; j++ ) {
                    switch ( threadbuckets[j]->free ) {
                        case BUCKETFREE:
                            thr = threadbuckets[j];
                            goto Found2;
                        case BUCKETCOMINGFREE:
                            thr = threadbuckets[j];
                            thr->free = BUCKETFREE;
                            for ( k = j+1; k < numthreadbuckets; k++ )
                                threadbuckets[k-1] = threadbuckets[k];
                            threadbuckets[numthreadbuckets-1] = thr;
                            j--;
                            break;
                        default:
                            break;
                    }
                }
                while ( topofavailables <= 0 ) {
                    MasterWait();
                }
                while ( topofavailables > 0 && numberoffullbuckets > 0 ) {
                    SendOneBucket(DOBRACKETS);
                }
            }
Found2:;
            while ( numberoffullbuckets > 0 ) {
                while ( topofavailables <= 0 ) {
                    MasterWait();
                }
                while ( topofavailables > 0 && numberoffullbuckets > 0 ) {
                    SendOneBucket(DOBRACKETS);
                }
            }
/*
            Disable the 'warming up' with smaller buckets.

            numpasses = 0;
            thrbufsiz = thrbufsiz0;
*/
            AN0.lastinindex = -1;
        }
        MasterWaitAll();
    }
#endif
/*
    #] Whole brackets :

    Now the loop to start a bucket
*/
    for(;;) {
        if ( fromspectator ) {
            ter = GetFromSpectator(thr->threadbuffer,fromspectator-1);
            if ( ter == 0 ) fromspectator = 0;
        }
        else {
            ter = GetTermP(B0,thr->threadbuffer);
        }
        if ( ter < 0 ) break;
```

```
        if ( ter == 0 ) { endofinput = 1; goto Finalize; }
        dd = AN0.deferskipped;
        if ( AR0.DeferFlag ) {
            defcount = 0;
            thr->deferbuffer[defcount++] = AR0.DefPosition;
            ttco = thr->compressbuffer; t1 = AR0.CompressBuffer; j = *t1;
            NCOPY(ttco,t1,j);
        }
        else if ( first && ( AC.CollectFun == 0 ) ) { /* Brackets ? */
            first = 0;
            t1 = tstop = thr->threadbuffer;
            tstop += *tstop; tstop -= ABS(tstop[-1]);
            t1++;
            while ( t1 < tstop ) {
                if ( t1[0] == HAAKJE ) { bra = 1; break; }
                t1 += t1[1];
            }
            t1 = thr->threadbuffer;
        }
/*
        Check whether we have a collect,function going. If so execute it.
*/
        if ( AC.CollectFun && *(thr->threadbuffer) < (AM.MaxTer/((LONG)sizeof(WORD))-10) ) {
            if ( ( dd = GetMoreTerms(thr->threadbuffer) ) < 0 ) {
                LowerSortLevel(); goto ProcErr;
            }
        }
/*
        Check whether we have a priority task:
*/
        if ( topofavailables > 0 && numberoffullbuckets > 0 ) SendOneBucket(LOWESTLEVELGENERATION);
/*
        Now put more terms in the bucket. Position tt after the first term
*/
        tt = thr->threadbuffer; tt += *tt;
        thr->totnum = 1;
        thr->usenum = 0;
/*
        Next we worry about the 'slow startup' in which we make the initial
        buckets smaller, so that we get all threads busy as soon as possible.
*/
        if ( numpasses > 0 ) {
            numbucket++;
            if ( numbucket >= numberofworkers ) {
                numbucket = 0;
                numpasses--;
                if ( numpasses == 0 ) thrbufsiz = thrbufsiz0;
                else                  thrbufsiz = thrbufsiz0 / (2 « numpasses);
            }
            thrbufsiz2 = thrbufsiz + thrbufsiz/5; /* for completing brackets */
        }
/*
        we have already 1+dd terms
*/
        while ( ( dd < thrbufsiz ) &&
            ( tt - thr->threadbuffer ) < ( thr->threadbuffersize - AM.MaxTer/((LONG)sizeof(WORD)) - 2 ) ) {
/*
            First check:
*/
            if ( topofavailables > 0 && numberoffullbuckets > 0 ) SendOneBucket(LOWESTLEVELGENERATION);
/*
            There is room in the bucket. Fill yet another term.
*/
            if ( GetTermP(B0,tt) == 0 ) { endofinput = 1; break; }
            dd++;
            thr->totnum++;
            dd += AN0.deferskipped;
            if ( AR0.DeferFlag ) {
                thr->deferbuffer[defcount++] = AR0.DefPosition;
                t1 = AR0.CompressBuffer; j = *t1;
                NCOPY(ttco,t1,j);
            }
            if ( AC.CollectFun && *tt < (AM.MaxTer/((LONG)sizeof(WORD))-10) ) {
                if ( ( ddd = GetMoreTerms(tt) ) < 0 ) {
                    LowerSortLevel(); goto ProcErr;
                }
                dd += ddd;
            }
            t1 = tt;
            tt += *tt;
        }
/*
        Check whether there are regular brackets and if we have no DeferFlag
        and no collect, we try to add more terms till we finish the current
        bracket. We should however not overdo it. Let us say: up to 20%
```

```
        more terms are allowed.
*/
        if ( bra ) {
            tstop = t1 + *t1; tstop -= ABS(tstop[-1]);
            t2 = t1+1;
            while ( t2 < tstop ) {
                if ( t2[0] == HAAKJE ) { break; }
                t2 += t2[1];
            }
            if ( t2[0] == HAAKJE ) {
              t2 += t2[1]; num = t2 - t1;
              while ( ( dd < thrbufsiz2 ) &&
                ( tt - thr->threadbuffer ) < ( thr->threadbuffersize - AM.MaxTer - 2 ) ) {
/*
                First check:
*/
                if ( topofavailables > 0 && numberoffullbuckets > 0 ) SendOneBucket(LOWESTLEVELGENERATION);
/*
                There is room in the bucket. Fill yet another term.
*/
                if ( GetTermP(B0,tt) == 0 ) { endofinput = 1; break; }
/*
                Same bracket?
*/
                tstop = tt + *tt; tstop -= ABS(tstop[-1]);
                if ( tstop-tt < num ) { /* Different: abort */
                    AR0.KeptInHold = 1;
                    break;
                }
                for ( i = 1; i < num; i++ ) {
                    if ( t1[i] != tt[i] ) break;
                }
                if ( i < num ) { /* Different: abort */
                    AR0.KeptInHold = 1;
                    break;
                }
/*
                Same bracket. We need this term.
*/
                dd++;
                thr->totnum++;
                tt += *tt;
              }
            }
        }
        thr->ddterms = dd; /* total number of terms including keep brackets */
        thr->firstterm = AN0.ninterms;
        AN0.ninterms += dd;
        *tt = 0;            /* mark end of bucket */
        thr->free = BUCKETFILLED;
        thr->type = BUCKETDOINGTERMS;
        numberoffullbuckets++;
        if ( topofavailables <= 0 && endofinput == 0 ) {
/*
            Problem: topofavailables may already be > 0, but the
            thread has not yet gone into waiting. Can the signal get lost?
            How can we tell that a thread is waiting for a signal?

            All threads are busy. Try to load up another bucket.
            In the future we could be more sophisticated.
            At the moment we load a complete bucket which could be
            1000 terms or even more.
            In principle it is better to keep a full bucket ready
            and check after each term we put in the next bucket. That
            way we don't waste time of the workers.
*/
            for ( j = 0; j < numthreadbuckets; j++ ) {
                switch ( threadbuckets[j]->free ) {
                    case BUCKETFREE:
                        thr = threadbuckets[j];
                        if ( !endofinput ) goto NextBucket;
/*
                        If we are at the end of the input we mark
                        the free buckets in a special way. That way
                        we don't keep running into them.
*/
                        thr->free = BUCKETATEND;
                        break;
                    case BUCKETCOMINGFREE:
                        thr = threadbuckets[j];
                        thr->free = BUCKETFREE;
/*
                        Bucket has just been finished.
                        Put at the end of the list. We don't want
                        an early bucket to wait to be treated last.
*/
```

```
                        for ( k = j+1; k < numthreadbuckets; k++ )
                            threadbuckets[k-1] = threadbuckets[k];
                        threadbuckets[numthreadbuckets-1] = thr;
                        j--;   /* we have to redo the same number j. */
                        break;
                    default:
                        break;
                }
            }
/*
            We have no free bucket or we are at the end.
            The only thing we can do now is wait for a worker to come free,
            provided there are still buckets to send.
*/
        }
/*
        Look for the next bucket to send. There is at least one full bucket!
*/
        for ( j = 0; j < numthreadbuckets; j++ ) {
            if ( threadbuckets[j]->free == BUCKETFILLED ) {
                thr = threadbuckets[j];
                for ( k = j+1; k < numthreadbuckets; k++ )
                    threadbuckets[k-1] = threadbuckets[k];
                threadbuckets[numthreadbuckets-1] = thr;
                break;
            }
        }
/*
        Wait for a thread to become available
        The bucket we are going to use is in thr.
*/
DoBucket:;
        AN0.ninterms++;
        while ( ( id = GetAvailableThread() ) < 0 ) { MasterWait(); }
/*
        Prepare the thread. Give it the term and variables.
*/
        LoadOneThread(0,id,thr,0);
        LOCK(thr->lock);
        thr->busy = BUCKETASSIGNED;
        UNLOCK(thr->lock);
        thr->free = BUCKETINUSE;
        numberoffullbuckets--;
/*
        And signal the thread to run.
        Form now on we may only interfere with this bucket
        1: after it has been marked BUCKETCOMINGFREE
        2: when thr->busy == BUCKETDOINGTERM and then only when protected by
           thr->lock. This would be for load balancing.
*/
        WakeupThread(id,LOWESTLEVELGENERATION);
/*      AN0.ninterms += thr->ddterms; */
/*
        Now look whether there is another bucket filled and a worker available
*/
        if ( topofavailables > 0 ) {  /* there is a worker */
            for ( j = 0; j < numthreadbuckets; j++ ) {
                if ( threadbuckets[j]->free == BUCKETFILLED ) {
                    thr = threadbuckets[j];
                    for ( k = j+1; k < numthreadbuckets; k++ )
                        threadbuckets[k-1] = threadbuckets[k];
                    threadbuckets[numthreadbuckets-1] = thr;
                    goto DoBucket; /* and we found a bucket */
                }
            }
/*
            no bucket is loaded but there is a thread available
            find a bucket to load. If there is none (all are USED or ATEND)
            we jump out of the loop.
*/
            for ( j = 0; j < numthreadbuckets; j++ ) {
                switch ( threadbuckets[j]->free ) {
                    case BUCKETFREE:
                        thr = threadbuckets[j];
                        if ( !endofinput ) goto NextBucket;
                        thr->free = BUCKETATEND;
                        break;
                    case BUCKETCOMINGFREE:
                        thr = threadbuckets[j];
                        if ( endofinput ) {
                            thr->free = BUCKETATEND;
                        }
                        else {
                            thr->free = BUCKETFREE;
                            for ( k = j+1; k < numthreadbuckets; k++ )
                                threadbuckets[k-1] = threadbuckets[k];
                            threadbuckets[numthreadbuckets-1] = thr;
```

```
                            j--;
                        }
                        break;
                    default:
                        break;
                }
            }
            if ( j >= numthreadbuckets ) break;
        }
        else {
/*
            No worker available.
            Look for a bucket to load.
            Its number will be in "still"
*/
Finalize:;
            still = -1;
            for ( j = 0; j < numthreadbuckets; j++ ) {
                switch ( threadbuckets[j]->free ) {
                    case BUCKETFREE:
                        thr = threadbuckets[j];
                        if ( !endofinput ) goto NextBucket;
                        thr->free = BUCKETATEND;
                        break;
                    case BUCKETCOMINGFREE:
                        thr = threadbuckets[j];
                        if ( endofinput ) thr->free = BUCKETATEND;
                        else {
                            thr->free = BUCKETFREE;
                            for ( k = j+1; k < numthreadbuckets; k++ )
                                threadbuckets[k-1] = threadbuckets[k];
                            threadbuckets[numthreadbuckets-1] = thr;
                            j--;
                        }
                        break;
                    case BUCKETFILLED:
                        if ( still < 0 ) still = j;
                        break;
                    default:
                        break;
                }
            }
            if ( still < 0 ) {
/*
                No buckets to be executed and no buckets FREE.
                We must be at the end. Break out of the loop.
*/
                break;
            }
            thr = threadbuckets[still];
            for ( k = still+1; k < numthreadbuckets; k++ )
                threadbuckets[k-1] = threadbuckets[k];
            threadbuckets[numthreadbuckets-1] = thr;
            goto DoBucket;
        }
NextBucket:;
    }
/*
    Now the stage one load balancing.
    If the load has been readjusted we have again filled buckets.
    In that case we jump back in the loop.

    Tricky point: when do the workers see the new value of AT.LoadBalancing?
    It should activate the locks on thr->busy
*/
    if ( AC.ThreadBalancing ) {
        for ( id = 1; id <= numberofworkers; id++ ) {
            AB[id]->T.LoadBalancing = 1;
        }
        if ( LoadReadjusted() ) goto Finalize;
        for ( id = 1; id <= numberofworkers; id++ ) {
            AB[id]->T.LoadBalancing = 0;
        }
    }
    if ( AC.ThreadBalancing ) {
/*
        The AS.Balancing flag should have Generator look for
        free workers and apply the "buro" method.

        There is still a serious problem.
        When for instance a sum_, there may be space created in a local
        compiler buffer for a wildcard substitution or whatever.
        Compiler buffer execution scribble space.....
        This isn't copied along?
        Look up ebufnum. There are 12 places with AddRHS!
        Problem: one process alloces in ebuf. Then term is given to
        other process. It would like to use from this ebuf, but the sender
```

```
        finishes first and removes the ebuf (and/or overwrites it).

        Other problem: local $ variables aren't copied along.
*/
        AS.Balancing = 0;
    }
    MasterWaitAll();
    AS.Balancing = 0;
/*
    When we deal with the last expression we can now remove the input
    scratch file. This saves potentially much disk space (up to 1/3)
*/
    if ( LastExpression ) {
        UpdateMaxSize();
        if ( AR0.infile->handle >= 0 ) {
            CloseFile(AR0.infile->handle);
            AR0.infile->handle = -1;
            remove(AR0.infile->name);
            PUTZERO(AR0.infile->POposition);
            AR0.infile->POfill = AR0.infile->POfull = AR0.infile->PObuffer;
        }
    }
/*
    We order the threads to finish in the MasterMerge routine
    It will start with waiting for all threads to finish.
    One could make an administration in which threads that have
    finished can start already with the final sort but
    1: The load balancing should not make this super urgent
    2: It would definitely not be very compatible with the second
       stage load balancing.
*/
    oldgzipCompress = AR0.gzipCompress;
    AR0.gzipCompress = 0;
    if ( AR0.outtohide ) AR0.outfile = AR0.hidefile;
    if ( MasterMerge() < 0 ) {
        if ( AR0.outtohide ) AR0.outfile = oldoutfile;
        AR0.gzipCompress = oldgzipCompress;
        goto ProcErr;
    }
    if ( AR0.outtohide ) AR0.outfile = oldoutfile;
    AR0.gzipCompress = oldgzipCompress;
/*
    Now wait for all threads to be ready to give them the cleaning up signal.
    With the new MasterMerge routine we can do the cleanup already automatically
    avoiding having to send these signals.
*/
    MasterWaitAll();
    AR0.sLevel--;
    for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
        if ( GetThread(id) > 0 ) WakeupThread(id,CLEANUPEXPRESSION);
    }
    e->numdummies = 0;
    for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
        if ( AB[id]->R.MaxDum - AM.IndDum > e->numdummies )
            e->numdummies = AB[id]->R.MaxDum - AM.IndDum;
        AR0.expchanged |= AB[id]->R.expchanged;
    }
/*
    And wait for all to be clean.
*/
    MasterWaitAll();
    AT0.WorkPointer = oldworkpointer;
    return(0);
ProcErr:;
    return(-1);
}

/*
    #] ThreadsProcessor :
    #[ LoadReadjusted :
*/
int LoadReadjusted()
{
    ALLPRIVATES *B0 = AB[0];
    THREADBUCKET *thr = 0, *thrtogo = 0;
    int numtogo, numfree, numbusy, n, nperbucket, extra, i, j, u, bus;
    LONG numinput;
    WORD *t1, *c1, *t2, *c2, *t3;
/*
    Start with waiting for at least one free processor.
    We don't want the master competing for time when all are busy.
*/
    while ( topofavailables <= 0 ) MasterWait();
/*
    Now look for the fullest bucket and make a list of free buckets
    The bad part is that most numbers can change at any moment.
*/
```

```
restart:;
    numtogo = 0;
    numfree = 0;
    numbusy = 0;
    for ( j = 0; j < numthreadbuckets; j++ ) {
        thr = threadbuckets[j];
        if ( thr->free == BUCKETFREE || thr->free == BUCKETATEND
        || thr->free == BUCKETCOMINGFREE ) {
            freebuckets[numfree++] = thr;
        }
        else if ( thr->type != BUCKETDOINGTERMS ) {}
        else if ( thr->totnum > 1 ) { /* never steal from a bucket with one term */
            LOCK(thr->lock);
            bus = thr->busy;
            UNLOCK(thr->lock);
            if ( thr->free == BUCKETINUSE ) {
                n = thr->totnum-thr->usenum;
                if ( bus == BUCKETASSIGNED ) numbusy++;
                else if ( ( bus != BUCKETASSIGNED )
                        && ( n > numtogo ) ) {
                    numtogo = n;
                    thrtogo = thr;
                }
            }
            else if ( bus == BUCKETTOBERELEASED
             && thr->free == BUCKETRELEASED ) {
                freebuckets[numfree++] = thr;
                thr->free = BUCKETATEND;
                LOCK(thr->lock);
                thr->busy = BUCKETPREPARINGTERM;
                UNLOCK(thr->lock);
            }
        }
    }
    if ( numfree == 0 ) return(0); /* serious problem */
    if ( numtogo > 0 ) {   /* provisionally there is something to be stolen */
        thr = thrtogo;
/*
        If the number has changed there is good progress.
        Maybe there is another thread that needs assistence.
        We start all over.
*/
        if ( thr->totnum-thr->usenum < numtogo ) goto restart;
/*
        If the thread is in the term loading phace
        (thr->busy == BUCKETPREPARINGTERM) we better stay away from it.
        We wait now for the thread to be busy, and don't allow it
        now to drop out of this state till we are done here.
        This all depends on whether AT.LoadBalancing == 1 is seen by
        the thread.
*/
        LOCK(thr->lock);
        if ( thr->busy != BUCKETDOINGTERM ) {
            UNLOCK(thr->lock);
            goto restart;
        }
        if ( thr->totnum-thr->usenum < numtogo ) {
            UNLOCK(thr->lock);
            goto restart;
        }
        thr->free = BUCKETTERMINATED;
/*
        The above will signal the thread we want to terminate.
        Next all effort goes into making sure the landing is soft.
        Unfortunately we don't want to wait for a signal, because the thread
        may be working for a long time on a single term.
*/
        if ( thr->usenum == thr->totnum ) {
/*
            Terminated in the mean time or by now working on the
            last term. Try again.
*/
            thr->free = BUCKETATEND;
            UNLOCK(thr->lock);
            goto restart;
        }
        goto intercepted;
    }
/*  UNLOCK(thr->lock); */
    if ( numbusy > 0 ) return(1); /* Wait a bit.... */
    return(0);
intercepted:;
/*
    We intercepted one successfully. Now it becomes interesting. Action:
    1: determine how many terms per free bucket.
    2: find the first untreated term.
    3: put the terms in the free buckets.
```

```

    Remember: we have the lock to avoid interference from the thread
    that is being robbed.
*/
    numinput = thr->firstterm + thr->usenum;
    nperbucket = numtogo / numfree;
    extra = numtogo - nperbucket*numfree;
    if ( AR0.DeferFlag ) {
        t1 = thr->threadbuffer; c1 = thr->compressbuffer; u = thr->usenum;
        for ( n = 0; n < thr->usenum; n++ ) { t1 += *t1; c1 += *c1; }
        t3 = t1;
        if ( extra > 0 ) {
          for ( i = 0; i < extra; i++ ) {
            thrtogo = freebuckets[i];
            t2 = thrtogo->threadbuffer;
            c2 = thrtogo->compressbuffer;
            thrtogo->free = BUCKETFILLED;
            thrtogo->type = BUCKETDOINGTERMS;
            thrtogo->totnum = nperbucket+1;
            thrtogo->ddterms = 0;
            thrtogo->usenum = 0;
            thrtogo->busy = BUCKETASSIGNED;
            thrtogo->firstterm = numinput;
            numinput += nperbucket+1;
            for ( n = 0; n <= nperbucket; n++ ) {
                j = *t1; NCOPY(t2,t1,j);
                j = *c1; NCOPY(c2,c1,j);
                thrtogo->deferbuffer[n] = thr->deferbuffer[u++];
            }
            *t2 = *c2 = 0;
          }
        }
        if ( nperbucket > 0 ) {
          for ( i = extra; i < numfree; i++ ) {
            thrtogo = freebuckets[i];
            t2 = thrtogo->threadbuffer;
            c2 = thrtogo->compressbuffer;
            thrtogo->free = BUCKETFILLED;
            thrtogo->type = BUCKETDOINGTERMS;
            thrtogo->totnum = nperbucket;
            thrtogo->ddterms = 0;
            thrtogo->usenum = 0;
            thrtogo->busy = BUCKETASSIGNED;
            thrtogo->firstterm = numinput;
            numinput += nperbucket;
            for ( n = 0; n < nperbucket; n++ ) {
                j = *t1; NCOPY(t2,t1,j);
                j = *c1; NCOPY(c2,c1,j);
                thrtogo->deferbuffer[n] = thr->deferbuffer[u++];
            }
            *t2 = *c2 = 0;
          }
        }
    }
    else {
        t1 = thr->threadbuffer;
        for ( n = 0; n < thr->usenum; n++ ) { t1 += *t1; }
        t3 = t1;
        if ( extra > 0 ) {
          for ( i = 0; i < extra; i++ ) {
            thrtogo = freebuckets[i];
            t2 = thrtogo->threadbuffer;
            thrtogo->free = BUCKETFILLED;
            thrtogo->type = BUCKETDOINGTERMS;
            thrtogo->totnum = nperbucket+1;
            thrtogo->ddterms = 0;
            thrtogo->usenum = 0;
            thrtogo->busy = BUCKETASSIGNED;
            thrtogo->firstterm = numinput;
            numinput += nperbucket+1;
            for ( n = 0; n <= nperbucket; n++ ) {
                j = *t1; NCOPY(t2,t1,j);
            }
            *t2 = 0;
          }
        }
        if ( nperbucket > 0 ) {
          for ( i = extra; i < numfree; i++ ) {
            thrtogo = freebuckets[i];
            t2 = thrtogo->threadbuffer;
            thrtogo->free = BUCKETFILLED;
            thrtogo->type = BUCKETDOINGTERMS;
            thrtogo->totnum = nperbucket;
            thrtogo->ddterms = 0;
            thrtogo->usenum = 0;
            thrtogo->busy = BUCKETASSIGNED;
            thrtogo->firstterm = numinput;
```

```
            numinput += nperbucket;
            for ( n = 0; n < nperbucket; n++ ) {
                j = *t1; NCOPY(t2,t1,j);
            }
            *t2 = 0;
          }
        }
    }
    *t3 = 0;   /* This is some form of extra insurance */
    if ( thr->free == BUCKETRELEASED && thr->busy == BUCKETTOBERELEASED ) {
        thr->free = BUCKETATEND; thr->busy = BUCKETPREPARINGTERM;
    }
    UNLOCK(thr->lock);
    return(1);
}

/*
    #] LoadReadjusted :
    #[ SortStrategy :
*/
/*
    #] SortStrategy :
    #[ PutToMaster :
*/
int PutToMaster(PHEAD WORD *term)
{
    int i,j,nexti,ret = 0;
    WORD *t, *fill, *top, zero = 0;
    if ( term == 0 ) { /* Mark the end of the expression */
        t = &zero; j = 1;
    }
    else {
        t = term; ret = j = *term;
        if ( j == 0 ) { j = 1; } /* Just in case there is a spurious end */
    }
    i = AT.SB.FillBlock;          /* The block we are working at */
    fill = AT.SB.MasterFill[i];     /* Where we are filling */
    top = AT.SB.MasterStop[i];      /* End of the block */
    while ( j > 0 ) {
        while ( j > 0 && fill < top ) {
            *fill++ = *t++; j--;
        }
        if ( j > 0 ) {
/*
            We reached the end of the block.
            Get the next block and release this block.
            The order is important. This is why there should be at least
            4 blocks or deadlocks can occur.
*/
            nexti = i+1;
            if ( nexti > AT.SB.MasterNumBlocks ) {
                nexti = 1;
            }
            LOCK(AT.SB.MasterBlockLock[nexti]);
            UNLOCK(AT.SB.MasterBlockLock[i]);
            AT.SB.MasterFill[i] = AT.SB.MasterStart[i];
            AT.SB.FillBlock = i = nexti;
            fill = AT.SB.MasterStart[i];
            top = AT.SB.MasterStop[i];
        }
    }
    AT.SB.MasterFill[i] = fill;
    return(ret);
}

/*
    #] PutToMaster :
    #[ SortBotOut :
*/

#ifdef WITHSORTBOTS

int
SortBotOut(PHEAD WORD *term)
{
    WORD im;

    if ( AT.identity != 0 ) return(PutToMaster(BHEAD term));

    if ( term == 0 ) {
        if ( FlushOut(&SortBotPosition,AR.outfile,1) ) return(-1);
        ADDPOS(AT.SS->SizeInFile[0],1);
        return(0);
    }
    else {
        numberofterms++;
        if ( ( im = PutOut(BHEAD term,&SortBotPosition,AR.outfile,1) ) < 0 ) {
```

```
            MLOCK(ErrorMessageLock);
            MesPrint("Called from MasterMerge/SortBotOut");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        ADDPOS(AT.SS->SizeInFile[0],im);
        return(im);
    }
}

#endif

/*
    #] SortBotOut :
    #[ MasterMerge :
*/
int MasterMerge()
{
    ALLPRIVATES *B0 = AB[0], *B = 0;
    SORTING *S = AT0.SS;
    WORD **poin, **poin2, ul, k, i, im, *m1, j;
    WORD lpat, mpat, level, l1, l2, r1, r2, r3, c;
    WORD *m2, *m3, r31, r33, ki, *rr;
    UWORD *coef;
    POSITION position;
    FILEHANDLE *fin, *fout;
#ifdef WITHSORTBOTS
    if ( numberofworkers > 2 ) return(SortBotMasterMerge());
#endif
    fin = &S->file;
    if ( AR0.PolyFun == 0 ) { S->PolyFlag = 0; }
    else if ( AR0.PolyFunType == 1 ) { S->PolyFlag = 1; }
    else if ( AR0.PolyFunType == 2 ) {
        if ( AR0.PolyFunExp == 2
          || AR0.PolyFunExp == 3 ) S->PolyFlag = 1;
        else                       S->PolyFlag = 2;
    }
    S->TermsLeft = 0;
    coef = AN0.SoScratC;
    poin = S->poina; poin2 = S->poin2a;
    rr = AR0.CompressPointer;
    *rr = 0;
/*
        #[ Setup :
*/
    S->inNum = numberofthreads;
    fout = AR0.outfile;
/*
    Load the patches. The threads have to finish their sort first.
*/
    S->lPatch = S->inNum - 1;
/*
    Claim all zero blocks. We need them anyway.
    In principle the workers should never get into these.
    We also claim all last blocks. This is a safety procedure that
    should prevent the workers from working their way around the clock
    before the master gets started again.
*/
    AS.MasterSort = 1;
    numberclaimed = 0;
    for ( i = 1; i <= S->lPatch; i++ ) {
        B = AB[i];
        LOCK(AT.SB.MasterBlockLock[0]);
        LOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
    }
/*
    Now wake up the threads and have them start their final sorting.
    They should start with claiming their block and the master is
    not allowed to continue until that has been done.
    This waiting of the master will be done below in MasterWaitAllBlocks
*/
    for ( i = 0; i < S->lPatch; i++ ) {
        GetThread(i+1);
        WakeupThread(i+1,FINISHEXPRESSION);
    }
/*
    Prepare the output file.
*/
    if ( fout->handle >= 0 ) {
        PUTZERO(position);
        SeekFile(fout->handle,&position,SEEK_END);
        ADDPOS(position,((fout->POfill-fout->PObuffer)*sizeof(WORD)));
    }
    else {
        SETBASEPOSITION(position,(fout->POfill-fout->PObuffer)*sizeof(WORD));
    }
/*
```

```
    Wait for all threads to finish loading their first block.
*/
    MasterWaitAllBlocks();
/*
    Claim all first blocks.
    We don't release the last blocks.
    The strategy is that we always keep the previous block.
    In principle it looks like it isn't needed for the last block but
    actually it is to keep the front from overrunning the tail when writing.
*/
    for ( i = 1; i <= S->lPatch; i++ ) {
        B = AB[i];
        LOCK(AT.SB.MasterBlockLock[1]);
        AT.SB.MasterBlock = 1;
    }
/*
        #] Setup :

    Now construct the tree:
*/
    lpat = 1;
    do { lpat «= 1; } while ( lpat < S->lPatch );
    mpat = ( lpat » 1 ) - 1;
    k = lpat - S->lPatch;
/*
    k is the number of empty places in the tree. they will
    be at the even positions from 2 to 2*k
*/
    for ( i = 1; i < lpat; i++ ) { S->tree[i] = -1; }
    for ( i = 1; i <= k; i++ ) {
        im = ( i « 1 ) - 1;
        poin[im] = AB[i]->T.SB.MasterStart[AB[i]->T.SB.MasterBlock];
        poin2[im] = poin[im] + *(poin[im]);
        S->used[i] = im;
        S->ktoi[im] = i-1;
        S->tree[mpat+i] = 0;
        poin[im-1] = poin2[im-1] = 0;
    }
    for ( i = (k«1)+1; i <= lpat; i++ ) {
        S->used[i-k] = i;
        S->ktoi[i] = i-k-1;
        poin[i] = AB[i-k]->T.SB.MasterStart[AB[i-k]->T.SB.MasterBlock];
        poin2[i] = poin[i] + *(poin[i]);
    }
/*
    the array poin tells the position of the i-th element of the S->tree
    'S->used' is a stack with the S->tree elements that need to be entered
    into the S->tree. at the beginning this is S->lPatch. during the
    sort there will be only very few elements.
    poin2 is the next value of poin. it has to be determined
    before the comparisons as the position or the size of the
    term indicated by poin may change.
    S->ktoi translates a S->tree element back to its stream number.

    start the sort
*/
    level = S->lPatch;
/*
    introduce one term
*/
OneTerm:
    k = S->used[level];
    i = k + lpat - 1;
    if ( !*(poin[k]) ) {
        do { if ( !( i »= 1 ) ) goto EndOfMerge; } while ( !S->tree[i] );
        if ( S->tree[i] == -1 ) {
            S->tree[i] = 0;
            level--;
            goto OneTerm;
        }
        k = S->tree[i];
        S->used[level] = k;
        S->tree[i] = 0;
    }
/*
    move terms down the tree
*/
    while ( i »= 1 ) {
        if ( S->tree[i] > 0 ) {
            if ( ( c = CompareTerms(B0,poin[S->tree[i]],poin[k],(WORD)0) ) > 0 ) {
/*
                S->tree[i] is the smaller. Exchange and go on.
*/
                S->used[level] = S->tree[i];
                S->tree[i] = k;
                k = S->used[level];
            }
```

```
            else if ( !c ) {    /* Terms are equal */
/*
                S->TermsLeft--;
                    Here the terms are equal and their coefficients
                    have to be added.
*/
                l1 = *( m1 = poin[S->tree[i]] );
                l2 = *( m2 = poin[k] );
                if ( S->PolyWise ) {  /* Here we work with PolyFun */
                    WORD *tt1, *w;
                    tt1 = m1;
                    m1 += S->PolyWise;
                    m2 += S->PolyWise;
                    if ( S->PolyFlag == 2 ) {
                        w = poly_ratfun_add(B0,m1,m2);
                        if ( *tt1 + w[1] - m1[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l is too small",AM.MaxTer);
                            MUNLOCK(ErrorMessageLock);
                            Terminate(-1);
                        }
                        AT0.WorkPointer = w;
                        if ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 && w[1] > FUNHEAD ) {
                            goto cancelled;
                        }
                    }
                    else {
                        w = AT0.WorkPointer;
                        if ( w + m1[1] + m2[1] > AT0.WorkTop ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("MasterMerge: A WorkSpace of %10l is too small",AM.WorkSize);
                            MUNLOCK(ErrorMessageLock);
                            Terminate(-1);
                        }
                        AddArgs(B0,m1,m2,w);
                    }
                    r1 = w[1];
                    if ( r1 <= FUNHEAD
                        || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) )
                             { goto cancelled; }
                    if ( r1 == m1[1] ) {
                        NCOPY(m1,w,r1);
                    }
                    else if ( r1 < m1[1] ) {
                        r2 = m1[1] - r1;
                        m2 = w + r1;
                        m1 += m1[1];
                        while ( --r1 >= 0 ) *--m1 = *--m2;
                        m2 = m1 - r2;
                        r1 = S->PolyWise;
                        while ( --r1 >= 0 ) *--m1 = *--m2;
                        *m1 -= r2;
                        poin[S->tree[i]] = m1;
                    }
                    else {
                        r2 = r1 - m1[1];
                        m2 = tt1 - r2;
                        r1 = S->PolyWise;
                        m1 = tt1;
                        *m1 += r2;
                        poin[S->tree[i]] = m2;
                        NCOPY(m2,m1,r1);
                        r1 = w[1];
                        NCOPY(m2,w,r1);
                    }
                }
                else {
                    r1 = *( m1 += l1 - 1 );
                    m1 -= ABS(r1) - 1;
                    r1 = ( ( r1 > 0 ) ? (r1-1) : (r1+1) ) » 1;
                    r2 = *( m2 += l2 - 1 );
                    m2 -= ABS(r2) - 1;
                    r2 = ( ( r2 > 0 ) ? (r2-1) : (r2+1) ) » 1;

                    if ( AddRat(B0,(UWORD *)m1,r1,(UWORD *)m2,r2,coef,&r3) ) {
                        MLOCK(ErrorMessageLock);
                        MesCall("MasterMerge");
                        MUNLOCK(ErrorMessageLock);
                        SETERROR(-1)
                    }

                    if ( AN.ncmod != 0 ) {
                        if ( ( AC.modmode & POSNEG ) != 0 ) {
                            NormalModulus(coef,&r3);
                        }
                        else if ( BigLong(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
```

```
                            WORD ii;
                            SubPLon(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod),coef,&r3);
                            coef[r3] = 1;
                            for ( ii = 1; ii < r3; ii++ ) coef[r3+ii] = 0;
                        }
                    }
                    r3 «= 1;
                    r33 = ( r3 > 0 ) ? ( r3 + 1 ) : ( r3 - 1 );
                    if ( r3 < 0 ) r3 = -r3;
                    if ( r1 < 0 ) r1 = -r1;
                    r1 «= 1;
                    r31 = r3 - r1;
                    if ( !r3 ) {        /* Terms cancel */
cancelled:
                        ul = S->used[level] = S->tree[i];
                        S->tree[i] = -1;
/*
                        We skip to the next term in stream ul
*/
                        im = *poin2[ul];
                        poin[ul] = poin2[ul];
                        ki = S->ktoi[ul];
                        if ( (poin[ul] + im + COMPINC) >=
                        AB[ki+1]->T.SB.MasterStop[AB[ki+1]->T.SB.MasterBlock]
                        && im > 0 ) {
/*
                            We made it to the end of the block. We have to
                            release the previous block and claim the next.
*/
                            B = AB[ki+1];
                            i = AT.SB.MasterBlock;
                            if ( i == 1 ) {
                                UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
                            }
                            else {
                                UNLOCK(AT.SB.MasterBlockLock[i-1]);
                            }
                            if ( i == AT.SB.MasterNumBlocks ) {
/*
                                Move the remainder down into block 0
*/
                                WORD *from, *to;
                                to = AT.SB.MasterStart[1];
                                from = AT.SB.MasterStop[i];
                                while ( from > poin[ul] ) *--to = *--from;
                                poin[ul] = to;
                                i = 1;
                            }
                            else { i++; }
                            LOCK(AT.SB.MasterBlockLock[i]);
                            AT.SB.MasterBlock = i;
                            poin2[ul] = poin[ul] + im;
                        }
                        else {
                            poin2[ul] += im;
                        }
                        S->used[++level] = k;
/*                      S->TermsLeft--; */
                    }
                    else if ( !r31 ) {      /* copy coef into term1 */
                        goto CopCof2;
                    }
                    else if ( r31 < 0 ) {       /* copy coef into term1
                                                and adjust the length of term1 */
                        goto CopCoef;
                    }
                    else {
/*
                            this is the dreaded calamity.
                            is there enough space?
*/
                        if( (poin[S->tree[i]]+l1+r31) >= poin2[S->tree[i]] ) {
/*
                                no space! now the special trick for which
                                we left 2*maxlng spaces open at the beginning
                                of each patch.
*/
                            if ( (l1 + r31)*((LONG)sizeof(WORD)) >= AM.MaxTer ) {
                                MLOCK(ErrorMessageLock);
                                MesPrint("MasterMerge: Coefficient overflow during sort");
                                MUNLOCK(ErrorMessageLock);
                                goto ReturnError;
                            }
                            m2 = poin[S->tree[i]];
                            m3 = ( poin[S->tree[i]] -= r31 );
                            do { *m3++ = *m2++; } while ( m2 < m1 );
                            m1 = m3;
```

```
                        }
CopCoef:
                        *(poin[S->tree[i]]) += r31;
CopCof2:
                        m2 = (WORD *)coef; im = r3;
                        NCOPY(m1,m2,im);
                        *m1 = r33;
                    }
                }
/*
                Now skip to the next term in stream k.
*/
NextTerm:
                im = poin2[k][0];
                poin[k] = poin2[k];
                ki = S->ktoi[k];
                if ( (poin[k] + im + COMPINC) >=
                AB[ki+1]->T.SB.MasterStop[AB[ki+1]->T.SB.MasterBlock]
                && im > 0 ) {
/*
                We made it to the end of the block. We have to
                release the previous block and claim the next.
*/
                    B = AB[ki+1];
                    i = AT.SB.MasterBlock;
                    if ( i == 1 ) {
                        UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
                    }
                    else {
                        UNLOCK(AT.SB.MasterBlockLock[i-1]);
                    }
                    if ( i == AT.SB.MasterNumBlocks ) {
/*
                        Move the remainder down into block 0
*/
                        WORD *from, *to;
                        to = AT.SB.MasterStart[1];
                        from = AT.SB.MasterStop[i];
                        while ( from > poin[k] ) *--to = *--from;
                        poin[k] = to;
                        i = 1;
                    }
                    else { i++; }
                    LOCK(AT.SB.MasterBlockLock[i]);
                    AT.SB.MasterBlock = i;
                    poin2[k] = poin[k] + im;
                }
                else {
                    poin2[k] += im;
                }
                goto OneTerm;
            }
        }
        else if ( S->tree[i] < 0 ) {
            S->tree[i] = k;
            level--;
            goto OneTerm;
        }
    }
/*
    found the smallest in the set. indicated by k.
    write to its destination.
*/
    S->TermsLeft++;
    if ( ( im = PutOut(B0,poin[k],&position,fout,1) ) < 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Called from MasterMerge with k = %d (stream %d)",k,S->ktoi[k]);
        MUNLOCK(ErrorMessageLock);
        goto ReturnError;
    }
    ADDPOS(S->SizeInFile[0],im);
    goto NextTerm;
EndOfMerge:
    if ( FlushOut(&position,fout,1) ) goto ReturnError;
    ADDPOS(S->SizeInFile[0],1);
    CloseFile(fin->handle);
    remove(fin->name);
    fin->handle = -1;
    position = S->SizeInFile[0];
    MULPOS(position,sizeof(WORD));
    S->GenTerms = 0;
    for ( j = 1; j <= numberofworkers; j++ ) {
        S->GenTerms += AB[j]->T.SS->GenTerms;
    }
    WriteStats(&position,2);
    Expressions[AR0.CurExpr].counter = S->TermsLeft;
    Expressions[AR0.CurExpr].size = position;
```

```
/*
    Release all locks
*/
    for ( i = 1; i <= S->lPatch; i++ ) {
        B = AB[i];
        UNLOCK(AT.SB.MasterBlockLock[0]);
        if ( AT.SB.MasterBlock == 1 ) {
            UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
        }
        else {
            UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock-1]);
        }
        UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock]);
    }
    AS.MasterSort = 0;
    return(0);
ReturnError:
    for ( i = 1; i <= S->lPatch; i++ ) {
        B = AB[i];
        UNLOCK(AT.SB.MasterBlockLock[0]);
        if ( AT.SB.MasterBlock == 1 ) {
            UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
        }
        else {
            UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock-1]);
        }
        UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock]);
    }
    AS.MasterSort = 0;
    return(-1);
}

/*
    #] MasterMerge :
    #[ SortBotMasterMerge :
*/

#ifdef WITHSORTBOTS

int SortBotMasterMerge()
{
    FILEHANDLE *fin, *fout;
    ALLPRIVATES *B = AB[0], *BB;
    POSITION position;
    SORTING *S = AT.SS;
    int i, j;
/*
    Get the sortbots get to claim their writing blocks.
    We have to wait till all have been claimed because they also have to
    claim the last writing blocks of the workers to prevent the head of
    the circular buffer to overrun the tail.

    Before waiting we can do some needed initializations.
    Also the master has to claim the last writing blocks of its input.
*/
    topsortbotavailables = 0;
    for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
        WakeupThread(i,INISORTBOT);
    }

    AS.MasterSort = 1;
    fout = AR.outfile;
    numberofterms = 0;
    AR.CompressPointer[0] = 0;
    numberclaimed = 0;
    BB = AB[AT.SortBotIn1];
    LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
    BB = AB[AT.SortBotIn2];
    LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);

    MasterWaitAllSortBots();
/*
    Now we can start up the workers. They will claim their writing blocks.
    Here the master will wait till all writing blocks have been claimed.
*/
    for ( i = 1; i <= numberofworkers; i++ ) {
        j = GetThread(i);
        WakeupThread(i,FINISHEXPRESSION);
    }
/*
    Prepare the output file in the mean time.
*/
    if ( fout->handle >= 0 ) {
        PUTZERO(SortBotPosition);
        SeekFile(fout->handle,&SortBotPosition,SEEK_END);
        ADDPOS(SortBotPosition,((fout->POfill-fout->PObuffer)*sizeof(WORD)));
    }
```

```
    else {
        SETBASEPOSITION(SortBotPosition,(fout->POfill-fout->PObuffer)*sizeof(WORD));
    }
    MasterWaitAllBlocks();
/*
    Now we can start the sortbots after which the master goes in
    sortbot mode to do its part of the job (the very final merge and
    the writing to output file).
*/
    topsortbotavailables = 0;
    for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
        WakeupThread(i,RUNSORTBOT);
    }
    if ( SortBotMerge(BHEAD0) ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Called from SortBotMasterMerge");
        MUNLOCK(ErrorMessageLock);
        AS.MasterSort = 0;
        return(-1);
    }
/*
    And next the cleanup
*/
    if ( S->file.handle >= 0 )
    {
        fin = &S->file;
        CloseFile(fin->handle);
        remove(fin->name);
        fin->handle = -1;
    }
    position = S->SizeInFile[0];
    MULPOS(position,sizeof(WORD));
    S->GenTerms = 0;
    for ( j = 1; j <= numberofworkers; j++ ) {
        S->GenTerms += AB[j]->T.SS->GenTerms;
    }
    S->TermsLeft = numberofterms;
    WriteStats(&position,2);
    Expressions[AR.CurExpr].counter = S->TermsLeft;
    Expressions[AR.CurExpr].size = position;
    AS.MasterSort = 0;
/*
    The next statement is to prevent one of the sortbots not having
    completely cleaned up before the next module starts.
    If this statement is omitted every once in a while one of the sortbots
    is still running when the next expression starts and misses its
    initialization. The result is usually disastrous.
*/
    MasterWaitAllSortBots();

    return(0);
}

#endif

/*
    #] SortBotMasterMerge :
    #[ SortBotMerge :
*/

#ifdef WITHSORTBOTS

int SortBotMerge(PHEAD0)
{
    GETBIDENTITY
    ALLPRIVATES *Bin1 = AB[AT.SortBotIn1],*Bin2 = AB[AT.SortBotIn2];
    WORD *term1, *term2, *next, *wp;
    int blin1, blin2;   /* Current block numbers */
    int error = 0;
    WORD l1, l2, *m1, *m2, *w, r1, r2, r3, r33, r31, *tt1, ii;
    WORD *to, *from, im, c;
    UWORD *coef;
    SORTING *S = AT.SS;
/*
    Set the pointers to the input terms and the output space
*/
    coef = AN.SoScratC;
    blin1 = 1;
    blin2 = 1;
    if ( AT.identity == 0 ) {
        wp = AT.WorkPointer;
    }
    else {
        wp = AT.WorkPointer = AT.WorkSpace;
    }
/*
    Get the locks for reading the input
```

```
    This means that we can start once these locks have been cleared
    which means that there will be input.
*/
    LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
    LOCK(Bin2->T.SB.MasterBlockLock[blin2]);

    term1 = Bin1->T.SB.MasterStart[blin1];
    term2 = Bin2->T.SB.MasterStart[blin2];
    AT.SB.FillBlock = 1;
/*
    Now the main loop. Keep going until one of the two hits the end.
*/
    while ( *term1 && *term2 ) {
        if ( ( c = CompareTerms(BHEAD term1,term2,(WORD)0) ) > 0 ) {
/*
            #[ One is smallest :
*/
            if ( SortBotOut(BHEAD term1) < 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
                MUNLOCK(ErrorMessageLock);
                error = -1;
                goto ReturnError;
            }
            im = *term1;
            next = term1 + im;
            if ( next >= Bin1->T.SB.MasterStop[blin1] || ( *next &&
            next+*next+COMPINC > Bin1->T.SB.MasterStop[blin1] ) ) {
                if ( blin1 == 1 ) {
                    UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
                }
                else {
                    UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
                }
                if ( blin1 == Bin1->T.SB.MasterNumBlocks ) {
/*
                    Move the remainder down into block 0
*/
                    to = Bin1->T.SB.MasterStart[1];
                    from = Bin1->T.SB.MasterStop[Bin1->T.SB.MasterNumBlocks];
                    while ( from > next ) *--to = *--from;
                    next = to;
                    blin1 = 1;
                }
                else {
                    blin1++;
                }
                LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
                Bin1->T.SB.MasterBlock = blin1;
            }
            term1 = next;
/*
            #] One is smallest :
*/
        }
        else if ( c < 0 ) {
/*
            #[ Two is smallest :
*/
            if ( SortBotOut(BHEAD term2) < 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
                MUNLOCK(ErrorMessageLock);
                error = -1;
                goto ReturnError;
            }
next2:      im = *term2;
            next = term2 + im;
            if ( next >= Bin2->T.SB.MasterStop[blin2] || ( *next
            && next+*next+COMPINC > Bin2->T.SB.MasterStop[blin2] ) ) {
                if ( blin2 == 1 ) {
                    UNLOCK(Bin2->T.SB.MasterBlockLock[Bin2->T.SB.MasterNumBlocks]);
                }
                else {
                    UNLOCK(Bin2->T.SB.MasterBlockLock[blin2-1]);
                }
                if ( blin2 == Bin2->T.SB.MasterNumBlocks ) {
/*
                    Move the remainder down into block 0
*/
                    to = Bin2->T.SB.MasterStart[1];
                    from = Bin2->T.SB.MasterStop[Bin2->T.SB.MasterNumBlocks];
                    while ( from > next ) *--to = *--from;
                    next = to;
                    blin2 = 1;
                }
                else {
```

```
                    blin2++;
                }
                LOCK(Bin2->T.SB.MasterBlockLock[blin2]);
                Bin2->T.SB.MasterBlock = blin2;
            }
            term2 = next;
/*
            #] Two is smallest :
*/
        }
        else {
/*
            #[ Equal :
*/
            l1 = *( m1 = term1 );
            l2 = *( m2 = term2 );
            if ( S->PolyWise ) {  /* Here we work with PolyFun */
                tt1 = m1;
                m1 += S->PolyWise;
                m2 += S->PolyWise;
                if ( S->PolyFlag == 2 ) {
                    AT.WorkPointer = wp;
                    w = poly_ratfun_add(BHEAD m1,m2);
                    if ( *tt1 + w[1] - m1[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l is too small",AM.MaxTer);
                        MUNLOCK(ErrorMessageLock);
                        Terminate(-1);
                    }
                    AT.WorkPointer = wp;
                    if ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 && w[1] > FUNHEAD ) {
                        goto cancelled;
                    }
                }
                else {
                    w = wp;
                    if ( w + m1[1] + m2[1] > AT.WorkTop ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("SortBotMerge(%d): A Maxtermsize of %10l is too small",
                                AT.identity,AM.MaxTer/sizeof(WORD));
                        MesPrint("m1[1] = %d, m2[1] = %d, Space = %l",m1[1],m2[1],(LONG)(AT.WorkTop-wp));
                        PrintTerm(term1,"term1");
                        PrintTerm(term2,"term2");
                        MesPrint("PolyWise = %d",S->PolyWise);
                        MUNLOCK(ErrorMessageLock);
                        Terminate(-1);
                    }
                    AddArgs(BHEAD m1,m2,w);
                }
                r1 = w[1];
                if ( r1 <= FUNHEAD
                    || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) )
                        { goto cancelled; }
                if ( r1 == m1[1] ) {
                    NCOPY(m1,w,r1);
                }
                else if ( r1 < m1[1] ) {
                    r2 = m1[1] - r1;
                    m2 = w + r1;
                    m1 += m1[1];
                    while ( --r1 >= 0 ) *--m1 = *--m2;
                    m2 = m1 - r2;
                    r1 = S->PolyWise;
                    while ( --r1 >= 0 ) *--m1 = *--m2;
                    *m1 -= r2;
                    term1 = m1;
                }
                else {
                    r2 = r1 - m1[1];
                    m2 = tt1 - r2;
                    r1 = S->PolyWise;
                    m1 = tt1;
                    *m1 += r2;
                    term1 = m2;
                    NCOPY(m2,m1,r1);
                    r1 = w[1];
                    NCOPY(m2,w,r1);
                }
            }
            else {
                r1 = *( m1 += l1 - 1 );
                m1 -= ABS(r1) - 1;
                r1 = ( ( r1 > 0 ) ? (r1-1) : (r1+1) ) » 1;
                r2 = *( m2 += l2 - 1 );
                m2 -= ABS(r2) - 1;
```

```
                r2 = ( ( r2 > 0 ) ? (r2-1) : (r2+1) ) » 1;

                if ( AddRat(BHEAD (UWORD *)m1,r1,(UWORD *)m2,r2,coef,&r3) ) {
                    MLOCK(ErrorMessageLock);
                    MesCall("SortBotMerge");
                    MUNLOCK(ErrorMessageLock);
                    SETERROR(-1)
                }

                if ( AN.ncmod != 0 ) {
                    if ( ( AC.modmode & POSNEG ) != 0 ) {
                        NormalModulus(coef,&r3);
                    }
                    else if ( BigLong(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
                        SubPLon(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod),coef,&r3);
                        coef[r3] = 1;
                        for ( ii = 1; ii < r3; ii++ ) coef[r3+ii] = 0;
                    }
                }
                if ( !r3 ) { goto cancelled; }
                r3 «= 1;
                r33 = ( r3 > 0 ) ? ( r3 + 1 ) : ( r3 - 1 );
                if ( r3 < 0 ) r3 = -r3;
                if ( r1 < 0 ) r1 = -r1;
                r1 «= 1;
                r31 = r3 - r1;
                if ( !r31 ) {       /* copy coef into term1 */
                    m2 = (WORD *)coef; im = r3;
                    NCOPY(m1,m2,im);
                    *m1 = r33;
                }
/*
                else if ( r31 < 0 ) {
                    *term1 += r31;
                    m2 = (WORD *)coef; im = r3;
                    NCOPY(m1,m2,im);
                    *m1 = r33;
                }
*/
                else {
                    to = wp; from = term1;
                    while ( from < m1 ) *to++ = *from++;
                    from = (WORD *)coef; im = r3;
                    NCOPY(to,from,im);
                    *to++ = r33;
                    wp[0] = to - wp;
                    if ( SortBotOut(BHEAD wp) < 0 ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
                        MUNLOCK(ErrorMessageLock);
                        error = -1;
                        goto ReturnError;
                    }
                    goto cancelled;
                }
            }
            if ( SortBotOut(BHEAD term1) < 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
                MUNLOCK(ErrorMessageLock);
                error = -1;
                goto ReturnError;
            }
cancelled:;     /* Now we need two new terms */
            im = *term1;
            next = term1 + im;
            if ( next >= Bin1->T.SB.MasterStop[blin1] || ( *next &&
            next++*next+COMPINC > Bin1->T.SB.MasterStop[blin1] ) ) {
                if ( blin1 == 1 ) {
                    UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
                }
                else {
                    UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
                }
                if ( blin1 == Bin1->T.SB.MasterNumBlocks ) {
/*
                    Move the remainder down into block 0

*/
                    to = Bin1->T.SB.MasterStart[1];
                    from = Bin1->T.SB.MasterStop[Bin1->T.SB.MasterNumBlocks];
                    while ( from > next ) *--to = *--from;
                    next = to;
                    blin1 = 1;
                }
                else {
                    blin1++;
                }
```

```
                LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
                Bin1->T.SB.MasterBlock = blin1;
            }
            term1 = next;
            goto next2;
/*
            #] Equal :
*/
        }
    }
/*
    Copy the tail
*/
    if ( *term1 ) {
/*
            #[ Tail in one :
*/
        while ( *term1 ) {
            if ( SortBotOut(BHEAD term1) < 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
                MUNLOCK(ErrorMessageLock);
                error = -1;
                goto ReturnError;
            }
            im = *term1;
            next = term1 + im;
            if ( next >= Bin1->T.SB.MasterStop[blin1] || ( *next &&
            next+*next+COMPINC > Bin1->T.SB.MasterStop[blin1] ) ) {
                if ( blin1 == 1 ) {
                    UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
                }
                else {
                    UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
                }
                if ( blin1 == Bin1->T.SB.MasterNumBlocks ) {
/*
                    Move the remainder down into block 0
*/
                    to = Bin1->T.SB.MasterStart[1];
                    from = Bin1->T.SB.MasterStop[Bin1->T.SB.MasterNumBlocks];
                    while ( from > next ) *--to = *--from;
                    next = to;
                    blin1 = 1;
                }
                else {
                    blin1++;
                }
                LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
                Bin1->T.SB.MasterBlock = blin1;
            }
            term1 = next;
        }
/*
            #] Tail in one :
*/
    }
    else if ( *term2 ) {
/*
            #[ Tail in two :
*/
        while ( *term2 ) {
            if ( SortBotOut(BHEAD term2) < 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
                MUNLOCK(ErrorMessageLock);
                error = -1;
                goto ReturnError;
            }
            im = *term2;
            next = term2 + im;
            if ( next >= Bin2->T.SB.MasterStop[blin2] || ( *next
            && next+*next+COMPINC > Bin2->T.SB.MasterStop[blin2] ) ) {
                if ( blin2 == 1 ) {
                    UNLOCK(Bin2->T.SB.MasterBlockLock[Bin2->T.SB.MasterNumBlocks]);
                }
                else {
                    UNLOCK(Bin2->T.SB.MasterBlockLock[blin2-1]);
                }
                if ( blin2 == Bin2->T.SB.MasterNumBlocks ) {
/*
                    Move the remainder down into block 0
*/
                    to = Bin2->T.SB.MasterStart[1];
                    from = Bin2->T.SB.MasterStop[Bin2->T.SB.MasterNumBlocks];
                    while ( from > next ) *--to = *--from;
                    next = to;
```

```
                    blin2 = 1;
                }
                else {
                    blin2++;
                }
                LOCK(Bin2->T.SB.MasterBlockLock[blin2]);
                Bin2->T.SB.MasterBlock = blin2;
            }
            term2 = next;
        }
/*
            #] Tail in two :
*/
    }
    SortBotOut(BHEAD 0);
ReturnError:;
/*
    Release all locks
*/
    UNLOCK(Bin1->T.SB.MasterBlockLock[blin1]);
    if ( blin1 > 1 ) {
        UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
    }
    else {
        UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
    }
    UNLOCK(Bin2->T.SB.MasterBlockLock[blin2]);
    if ( blin2 > 1 ) {
        UNLOCK(Bin2->T.SB.MasterBlockLock[blin2-1]);
    }
    else {
        UNLOCK(Bin2->T.SB.MasterBlockLock[Bin2->T.SB.MasterNumBlocks]);
    }
    if ( AT.identity > 0 ) {
        UNLOCK(AT.SB.MasterBlockLock[AT.SB.FillBlock]);
    }
/*
    And that was all folks
*/
    return(error);
}

#endif

/*
    #] SortBotMerge :
    #[ IniSortBlocks :
*/

static int SortBlocksInitialized = 0;

int IniSortBlocks(int numworkers)
{
    ALLPRIVATES *B;
    SORTING *S;
    LONG totalsize, workersize, blocksize, numberofterms;
    int maxter, id, j;
    int numberofblocks = NUMBEROFBLOCKSINSORT, numparts;
    WORD *w;

    if ( SortBlocksInitialized ) return(0);
    SortBlocksInitialized = 1;
    if ( numworkers == 0 ) return(0);

#ifdef WITHSORTBOTS
    if ( numworkers > 2 ) {
        numparts = 2*numworkers - 2;
        numberofblocks = numberofblocks/2;
    }
    else {
        numparts = numworkers;
    }
#else
    numparts = numworkers;
#endif
    S = AM.S0;
    totalsize = S->LargeSize + S->SmallEsize;
    workersize = totalsize / numparts;
    maxter = AM.MaxTer/sizeof(WORD);
    blocksize = ( workersize - maxter )/numberofblocks;
    numberofterms = blocksize / maxter;
    if ( numberofterms < MINIMUMNUMBEROFTERMS ) {
/*
        This should have been taken care of in RecalcSetups.
*/
        MesPrint("We have a problem with the size of the blocks in IniSortBlocks");
        Terminate(-1);
```

```
    }
/*
    Layout:  For each worker
                block 0: size is maxter WORDS
                numberofblocks blocks of size blocksize WORDS
*/
    w = S->lBuffer;
    if ( w == 0 ) w = S->sBuffer;
    for ( id = 1; id <= numparts; id++ ) {
        B = AB[id];
        AT.SB.MasterBlockLock = (pthread_mutex_t *)Malloc1(
            sizeof(pthread_mutex_t)*(numberofblocks+1),"MasterBlockLock");
        AT.SB.MasterStart = (WORD **)Malloc1(sizeof(WORD *)*(numberofblocks+1)*3,"MasterBlock");
        AT.SB.MasterFill = AT.SB.MasterStart + (numberofblocks+1);
        AT.SB.MasterStop = AT.SB.MasterFill  + (numberofblocks+1);
        AT.SB.MasterNumBlocks = numberofblocks;
        AT.SB.MasterBlock = 0;
        AT.SB.FillBlock = 0;
        AT.SB.MasterFill[0] = AT.SB.MasterStart[0] = w;
        w += maxter;
        AT.SB.MasterStop[0] = w;
        AT.SB.MasterBlockLock[0] = dummylock;
        for ( j = 1; j <= numberofblocks; j++ ) {
            AT.SB.MasterFill[j] = AT.SB.MasterStart[j] = w;
            w += blocksize;
            AT.SB.MasterStop[j] = w;
            AT.SB.MasterBlockLock[j] = dummylock;
        }
    }
    if ( w > S->sTop2 ) {
        MesPrint("Counting problem in IniSortBlocks");
        Terminate(-1);
    }
    return(0);
}

/*
    #] IniSortBlocks :
    #[ DefineSortBotTree :
*/

#ifdef WITHSORTBOTS

void DefineSortBotTree()
{
    ALLPRIVATES *B;
    int n, i, from;
    if ( numberofworkers <= 2 ) return;
    n = numberofworkers*2-2;
    for ( i = numberofworkers+1, from = 1; i <= n; i++ ) {
        B = AB[i];
        AT.SortBotIn1 = from++;
        AT.SortBotIn2 = from++;
    }
    B = AB[0];
    AT.SortBotIn1 = from++;
    AT.SortBotIn2 = from++;
}

#endif

/*
    #] DefineSortBotTree :
    #[ GetTerm2 :

    Routine does a GetTerm but only when a bracket index is involved and
    only from brackets that have been judged not suitable for treatment
    as complete brackets by a single worker. Whether or not a bracket should
    be treated by a single worker is decided by TreatIndexEntry
*/

WORD GetTerm2(PHEAD WORD *term)
{
    WORD *ttco, *tt, retval;
    LONG n,i;
    FILEHANDLE *fi;
    EXPRESSIONS e = AN.expr;
    BRACKETINFO *b  = e->bracketinfo;
    BRACKETINDEX *bi = b->indexbuffer;
    POSITION where, eonfile = AS.OldOnFile[e-Expressions], bstart, bnext;
/*
    1: Get the current position.
*/
    switch ( e->status ) {
        case UNHIDELEXPRESSION:
        case UNHIDEGEXPRESSION:
        case DROPHLEXPRESSION:
```

```
        case DROPHGEXPRESSION:
        case HIDDENLEXPRESSION:
        case HIDDENGEXPRESSION:
            fi = AR.hidefile;
            break;
        default:
            fi = AR.infile;
            break;
    }
    if ( AR.KeptInHold ) {
        retval = GetTerm(BHEAD term);
        return(retval);
    }
    SeekScratch(fi,&where);
    if ( AN.lastinindex < 0 ) {
/*
        We have to test whether we have to do the first bracket
*/
        if ( ( n = TreatIndexEntry(BHEAD 0) ) <= 0 ) {
            AN.lastinindex = n;
            where = bi[n].start;
            ADD2POS(where,eonfile);
            SetScratch(fi,&where);
/*
            Put the bracket in the Compress buffer.
*/
            ttco = AR.CompressBuffer;
            tt = b->bracketbuffer + bi[0].bracket;
            i = *tt;
            NCOPY(ttco,tt,i)
            AR.CompressPointer = ttco;
            retval = GetTerm(BHEAD term);
            return(retval);
        }
        else AN.lastinindex = n-1;
    }
/*
    2: Find the corresponding index number
       a: test whether it is in the current bracket
*/
    n = AN.lastinindex;
    bstart = bi[n].start;
    ADD2POS(bstart,eonfile);
    bnext = bi[n].next;
    ADD2POS(bnext,eonfile);
    if ( ISLESSPOS(bstart,where) && ISLESSPOS(where,bnext) ) {
        retval = GetTerm(BHEAD term);
        return(retval);
    }
    for ( n++ ; n < b->indexfill; n++ ) {
        i = TreatIndexEntry(BHEAD n);
        if ( i <= 0 ) {
/*
            Put the bracket in the Compress buffer.
*/
            ttco = AR.CompressBuffer;
            tt = b->bracketbuffer + bi[n].bracket;
            i = *tt;
            NCOPY(ttco,tt,i)
            AR.CompressPointer = ttco;
            AN.lastinindex = n;
            where = bi[n].start;
            ADD2POS(where,eonfile);
            SetScratch(fi,&(where));
            retval = GetTerm(BHEAD term);
            return(retval);
        }
        else n += i - 1;
    }
    return(0);
}

/*
    #] GetTerm2 :
    #[ TreatIndexEntry :
*/
int TreatIndexEntry(PHEAD LONG n)
{
    BRACKETINFO *b  = AN.expr->bracketinfo;
    LONG numbra = b->indexfill - 1, i;
    LONG totterms;
    BRACKETINDEX *bi;
    POSITION pos1, average;
/*
    1: number of the bracket should be such that there is one bucket
       for each worker remaining.
*/
```

```
    if ( ( numbra - n ) <= numberofworkers ) return(0);
/*
    2: size of the bracket contents should be less than what remains in
       the expression divided by the number of workers.
*/
    bi = b->indexbuffer;
    DIFPOS(pos1,bi[numbra].next,bi[n].next);  /* Size of what remains */
    BASEPOSITION(average) = DIVPOS(pos1,(3*numberofworkers));
    DIFPOS(pos1,bi[n].next,bi[n].start);      /* Size of the current bracket */
    if ( ISLESSPOS(average,pos1) ) return(0);
/*
    It passes.
    Now check whether we can do more brackets
*/
    totterms = bi->termsinbracket;
    if ( totterms > 2*AC.ThreadBucketSize ) return(1);
    for ( i = 1; i < numbra-n; i++ ) {
        DIFPOS(pos1,bi[n+i].next,bi[n].start); /* Size of the combined brackets */
        if ( ISLESSPOS(average,pos1) ) return(i);
        totterms += bi->termsinbracket;
        if ( totterms > 2*AC.ThreadBucketSize ) return(i+1);
    }
/*
    We have a problem at the end of the system. Just do one.
*/
    return(1);
}

/*
    #] TreatIndexEntry :
    #[ SetHideFiles :
*/

void SetHideFiles() {
    int i;
    ALLPRIVATES *B, *B0 = AB[0];
    for ( i = 1; i <= numberofworkers; i++ ) {
        B = AB[i];
        AR.hidefile->handle = AR0.hidefile->handle;
        if ( AR.hidefile->handle < 0 ) {
            AR.hidefile->PObuffer = AR0.hidefile->PObuffer;
            AR.hidefile->POstop = AR0.hidefile->POstop;
            AR.hidefile->POfill = AR0.hidefile->POfill;
            AR.hidefile->POfull = AR0.hidefile->POfull;
            AR.hidefile->POsize = AR0.hidefile->POsize;
            AR.hidefile->POposition = AR0.hidefile->POposition;
            AR.hidefile->filesize = AR0.hidefile->filesize;
        }
        else {
            AR.hidefile->PObuffer = AR.hidefile->wPObuffer;
            AR.hidefile->POstop = AR.hidefile->wPOstop;
            AR.hidefile->POfill = AR.hidefile->wPOfill;
            AR.hidefile->POfull = AR.hidefile->wPOfull;
            AR.hidefile->POsize = AR.hidefile->wPOsize;
            PUTZERO(AR.hidefile->POposition);
        }
    }
}

/*
    #] SetHideFiles :
    #[ IniFbufs :
*/

void IniFbufs(VOID)
{
    int i;
    for ( i = 0; i < AM.totalnumberofthreads; i++ ) {
        IniFbuffer(AB[i]->T.fbufnum);
    }
}

/*
    #] IniFbufs :
    #[ SetMods :
*/

void SetMods()
{
    ALLPRIVATES *B;
    int i, n, j;
    for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
        B = AB[j];
        AN.ncmod = AC.ncmod;
        if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
        n = ABS(AN.ncmod);
        AN.cmod = (UWORD *)Malloc1(sizeof(WORD)*n,"AN.cmod");
```

```
        for ( i = 0; i < n; i++ ) AN.cmod[i] = AC.cmod[i];
    }
}

/*
    #] SetMods :
    #[ UnSetMods :
*/

void UnSetMods()
{
    ALLPRIVATES *B;
    int j;
    for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
        B = AB[j];
        if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
        AN.cmod = 0;
    }
}

/*
    #] UnSetMods :
    #[ find_Horner_MCTS_expand_tree_threaded :
*/

void find_Horner_MCTS_expand_tree_threaded() {
    int id;
    while (( id = GetAvailableThread() ) < 0)
        MasterWait();
    WakeupThread(id,MCTSEXPANDTREE);
}

/*
    #] find_Horner_MCTS_expand_tree_threaded :
    #[ optimize_expression_given_Horner_threaded :
*/

extern void optimize_expression_given_Horner_threaded() {
    int id;
    while (( id = GetAvailableThread() ) < 0)
        MasterWait();
    WakeupThread(id,OPTIMIZEEXPRESSION);
}

/*
    #] optimize_expression_given_Horner_threaded :
*/

#endif
```

4.116 token.c File Reference

`#include "form3.h"`

Include dependency graph for token.c:

Macros

- #define CHECKPOLY {if(polyflag)MesPrint("&Illegal use of polynomial function"); polyflag = 0; }

Functions

- int tokenize (UBYTE ∗in, WORD leftright)
- void WriteTokens (SBYTE ∗in)
- int simp1token (SBYTE ∗s)
- int simpwtoken (SBYTE ∗s)
- int simp2token (SBYTE ∗s)
- int simp3atoken (SBYTE ∗s, int mode)
- int simp3btoken (SBYTE ∗s, int mode)
- int simp4token (SBYTE ∗s)
- int simp5token (SBYTE ∗s, int mode)
- int simp6token (SBYTE ∗tokens, int mode)

Variables

- char ∗ ttypes [ ]

4.116.1 Detailed Description

The tokenizer. This is a part of the compiler that does an intermediate type of translation. It does look up the names etc and can do a number of optimizations. The resulting output is a stream of bytes which can be processed by the code generator (in the file compiler.c)

Definition in file token.c.

4.116.2 Macro Definition Documentation

4.116.2.1 CHECKPOLY

```
#define CHECKPOLY {if(polyflag)MesPrint("&Illegal use of polynomial function"); polyflag = 0;
}
```

Definition at line 56 of file token.c.

4.116.3 Function Documentation

4.116.3.1 tokenize()

```
int tokenize (
            UBYTE * in,
            WORD leftright )
```

Definition at line 58 of file token.c.

4.116.3.2 WriteTokens()

```
void WriteTokens (
            SBYTE * in )
```

Definition at line 592 of file token.c.

4.116.3.3 simp1token()

```
int simp1token (
            SBYTE * s )
```

Definition at line 640 of file token.c.

4.116.3.4 simpwtoken()

```
int simpwtoken (
            SBYTE * s )
```

Definition at line 729 of file token.c.

4.116.3.5 simp2token()

```
int simp2token (
            SBYTE * s )
```

Definition at line 883 of file token.c.

4.116.3.6 simp3atoken()

```
int simp3atoken (
            SBYTE * s,
            int mode )
```

Definition at line 1127 of file token.c.

4.116.3.7 simp3btoken()

```
int simp3btoken (
            SBYTE * s,
            int mode )
```

Definition at line 1350 of file token.c.

4.116.3.8 simp4token()

```
int simp4token (
            SBYTE * s )
```

Definition at line 1726 of file token.c.

4.116.3.9 simp5token()

```
int simp5token (
            SBYTE * s,
            int mode )
```

Definition at line 1883 of file token.c.

4.116.3.10 simp6token()

```
int simp6token (
            SBYTE * tokens,
            int mode )
```

Definition at line 1929 of file token.c.

4.116.4 Variable Documentation

4.116.4.1 ttypes

```
char* ttypes[]
```

Initial value:

```
= { "\n", "S", "I", "V", "F", "set", "E", "dotp", "#",
   "sub", "d_", "$", "dub", "(", ")", "?", "??", ".", "[", "]",
   ",", "((", ))", "*", "/", "^", "+", "-", "!", "end", "{{", "}}",
   "N_?", "conj", "()", "#d", "^d", "_", "snum"  }
```

Definition at line 587 of file token.c.

4.117 token.c

Go to the documentation of this file.

```
00001
00008 /* #[ License : */
00009 /*
00010  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00011  *   When using this file you are requested to refer to the publication
00012  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013  *   This is considered a matter of courtesy as the development was paid
00014  *   for by FOM the Dutch physics granting agency and we would like to
00015  *   be able to track its scientific use to convince FOM of its value
00016  *   for the community.
00017  *
00018  *   This file is part of FORM.
00019  *
00020  *   FORM is free software: you can redistribute it and/or modify it under the
00021  *   terms of the GNU General Public License as published by the Free Software
00022  *   Foundation, either version 3 of the License, or (at your option) any later
00023  *   version.
00024  *
00025  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028  *   details.
00029  *
00030  *   You should have received a copy of the GNU General Public License along
00031  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032  */
00033 /* #] License : */
00034 /*
00035     #[ Includes :
```

```
*/

#include "form3.h"

/*
    #] Includes :
    #[ Compiler :
        #[ tokenize :

        Takes the input in 'in' and translates it into tokens.
        The tokens are put in the token buffer which starts at 'AC.tokens'
        and runs till 'AC.toptokens'
        We may assume that the various types of brackets match properly.
        object = -1: after , or (
        object = 0: name/variable/number etc is allowed
        object = 1: variable.
        object = 2: number
        object = 3: ) after subexpression
*/

#define CHECKPOLY {if(polyflag)MesPrint("&Illegal use of polynomial function"); polyflag = 0; }

int tokenize(UBYTE *in, WORD leftright)
{
    int error = 0, object, funlevel = 0, bracelevel = 0, explevel = 0, numexp;
    int polyflag = 0;
    WORD number, type;
    UBYTE *s = in, c;
    SBYTE *out, *outtop, num[MAXNUMSIZE], *t;
    LONG i;
    if ( AC.tokens == 0 ) {
        SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
        SBYTE **pppp = &(AC.toptokens);
        DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"start tokens");
    }
    out = AC.tokens;
    outtop = AC.toptokens - MAXNUMSIZE;
    AC.dumnumflag = 0;
    object = 0;
    while ( *in ) {
        if ( out > outtop ) {
            LONG oldsize = (LONG)(out - AC.tokens);
            SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
            SBYTE **pppp = &(AC.toptokens);
            DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"expand tokens");
            out = AC.tokens + oldsize;
            outtop = AC.toptokens - MAXNUMSIZE;
        }
        switch ( FG.cTable[*in] ) {
            case 0:     /* a-zA-Z */
                CHECKPOLY
                s = in++;
                while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1
                || *in == '_' ) in++;
dovariable:     c = *in; *in = 0;
                if ( object > 0 ) {
                    MesPrint("&Illegal position for %s",s);
                    if ( !error ) error = 1;
                }
                if ( out > AC.tokens && ( out[-1] == TWILDCARD || out[-1] == TNOT ) ) {
                    type = GetName(AC.varnames,s,&number,NOAUTO);
                }
                else {
                    type = GetName(AC.varnames,s,&number,WITHAUTO);
                }
                if ( type < 0 )
                    type = GetName(AC.exprnames,s,&number,NOAUTO);
                switch ( type ) {
                    case CSYMBOL:       *out++ = TSYMBOL;     break;
                    case CINDEX:
                        if ( number >= (AM.IndDum-AM.OffsetIndex) ) {
                            if ( c != '?' ) {
                                MesPrint("&Generated indices should be of the type Nnumber_?");
                                error = 1;
                            }
                            else {
                                *in++ = c; c = *in; *in = 0;
                                AC.dumnumflag = 1;
                            }
                        }
                        *out++ = TINDEX;
                        break;
                    case CVECTOR:       *out++ = TVECTOR;     break;
                    case CFUNCTION:
#ifdef WITHMPI
                        /*
                         * In the preprocessor, random functions in #$var=... and #inside
```

```
                         * may cause troubles, because the program flow on a slave may be
                         * different from those on others. We set AC.RhsExprInModuleFlag in order
                         * to make the change of $-variable be done on the master and thus keep the
                         * consistency among the master and all slave processes. The previous value
                         * of AC.RhsExprInModuleFlag will be restored after #$var=... and #inside.
                         */
                        if ( AP.PreAssignFlag || AP.PreInsideLevel ) {
                            switch ( number + FUNCTION ) {
                                case RANDOMFUNCTION:
                                case RANPERM:
                                    AC.RhsExprInModuleFlag = 1;
                            }
                        }
#endif
                        *out++ = TFUNCTION;
                        break;
                    case CSET:          *out++ = TSET;        break;
                    case CEXPRESSION:   *out++ = TEXPRESSION;
                                        if ( leftright == LHSIDE ) {
                                            if ( !error ) error = 1;
                                            MesPrint("&Expression not allowed in LH-side of substitution: %s",s);
                                        }
/*[06nov2003 mt]:*/
#ifdef WITHMPI
                                        else/*RHSide*/
                                            /* NOTE: We always set AC.RhsExprInModuleFlag regardless of
                                             *       AP.PreAssignFlag or AP.PreInsideLevel because we have to detect
                                             *       RHS expressions even in those cases. */
                                            AC.RhsExprInModuleFlag = 1;
                                            if ( !AP.PreAssignFlag && !AP.PreInsideLevel )
                                                Expressions[number].vflags |= ISINRHS;
#endif
/*:[06nov2003 mt]*/
                                        if ( AC.exprfillwarning == 0 ) {
                                                AC.exprfillwarning = 1;
                                        }
                                        break;
                    case CDELTA:        *out++ = TDELTA;      *in = c;
                                        object = 1; continue;
                    case CDUBIOUS:      *out++ = TDUBIOUS;    break;
                    default:            *out++ = TDUBIOUS;
                                        if ( !error ) error = 1;
                                        MesPrint("&Undeclared variable %s",s);
                                        number = AddDubious(s);
                                        break;
                }
                object = 1;
donumber:       i = 0;
                do { num[i++] = (SBYTE)(number & 0x7F); number >= 7; } while ( number );
                while ( --i >= 0 ) *out++ = num[i];
                *in = c;
                break;
            case 1:     /* 0-9 */
                CHECKPOLY
                s = in;
                while ( *s == '0' && FG.cTable[s[1]] == 1 ) s++;
                in = s+1; i = 1;
                while ( FG.cTable[*in] == 1 ) { in++; i++; }
                if ( object > 0 ) {
                    c = *in; *in = 0;
                    MesPrint("&Illegal position for %s",s);
                    *in = c;
                    if ( !error ) error = 1;
                }
                if ( i == 1 && *in == '_' && ( *s == '5' || *s == '6'
                || *s == '7' ) ) {
                    in++; *out++ = TSGAMMA; *out++ = (SBYTE)(*s - '4');
                    object = 1;
                    break;
                }
                *out++ = TNUMBER;
                if ( ( i & 1 ) != 0 ) *out++ = (SBYTE)(*s++ - '0');
                while ( out + (in-s)/2 >= AC.toptokens ) {
                    LONG oldsize = (LONG)(out - AC.tokens);
                    SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
                    SBYTE **pppp = &(AC.toptokens);
                    DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"more tokens");
                    out = AC.tokens + oldsize;
                    outtop = AC.toptokens - MAXNUMSIZE;
                }
                while ( s < in ) {   /* We store in base 100 */
                    *out++ = (SBYTE)(( *s - '0' ) * 10 + ( s[1] - '0' ));
                    s += 2;
                }
```

```
                object = 2;
                break;
            case 2:     /* . $ _ ? # ' */
                CHECKPOLY
                if ( *in == '?' ) {
                    if ( leftright == LHSIDE ) {
                        if ( object == 1 ) {    /* follows a name */
                            in++; *out++ = TWILDCARD;
                            if ( FG.cTable[in[0]] == 0 || in[0] == '[' || in[0] == '{' ) object = 0;
                        }
                        else if ( object == -1 ) {  /* follows comma or ( */
                            in++; s = in;
                            while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
                            c = *in; *in = 0;
                            if ( FG.cTable[*s] != 0 ) {
                                MesPrint("&Illegal name for argument list variable %s",s);
                                error = 1;
                            }
                            else {
                                i = AddWildcardName((UBYTE *)s);
                                *in = c;
                                *out++ = TWILDARG;
                                *out++ = (SBYTE)i;
                            }
                            object = 1;
                        }
                        else {
                            MesPrint("&Illegal position for ?");
                            error = 1;
                            in++;
                        }
                    }
                    else {
                        if ( object != -1 ) goto IllPos;
                        in++;
                        if ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) {
                            s = in;
                            while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
                            c = *in; *in = 0;
                            i = GetWildcardName((UBYTE *)s);
                            if ( i <= 0 ) {
                                MesPrint("&Undefined argument list variable %s",s);
                                error = 1;
                            }
                            *in = c;
                            *out++ = TWILDARG;
                            *out++ = (SBYTE)i;
                        }
                        else {
                            if ( AC.vectorlikeLHS == 0 ) {
                                MesPrint("&Generated index ? only allowed in vector substitution",s);
                                error = 1;
                            }
                            *out++ = TGENINDEX;
                        }
                        object = 1;
                    }
                }
                else if ( *in == '.' ) {
                    if ( object == 1 ) {    /* follows a name */
                        *out++ = TDOT;
                        object = 0;
                        in++;
                    }
                    else goto IllPos;
                }
                else if ( *in == '$' ) {    /* $ variable */
                    in++;
                    s = in;
                    if ( FG.cTable[*in] == 0 ) {
                        while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
                        if ( *in == '_' && AP.PreAssignFlag == 2 ) in++;
                        c = *in; *in = 0;
                        if ( object > 0 ) {
                            if ( object != 1 || leftright == RHSIDE ) {
                                MesPrint("&Illegal position for $%s",s);
                                if ( !error ) error = 1;
                            }   /* else can be assignment in wildcard */
                            else {
                                if ( ( number = GetDollar(s) ) < 0 ) {
                                    number = AddDollar(s,0,0,0);
                                }
                            }
                        }
                        else if ( ( number = GetDollar(s) ) < 0 ) {
                            MesPrint("&Undefined variable $%s",s);
                            if ( !error ) error = 1;
```

```
                        number = AddDollar(s,0,0,0);
                    }
                    *out++ = TDOLLAR;
                    object = 1;
                    if ( ( AC.exprfillwarning == 0 ) &&
                         ( ( out > AC.tokens+1 ) && ( out[-2] != TWILDCARD ) ) ) {
                        AC.exprfillwarning = 1;
                    }
                    goto donumber;
                }
                else {
                    MesPrint("Illegal name for $ variable after %s",in);
                    if ( !error ) error = 1;
                }
            }
            else if ( *in == '#' ) {
                if ( object == 1 ) {    /* follows a name */
                    *out++ = TCONJUGATE;
                }
            }
            else goto IllPos;
            break;
        case 3:     /* [ ] */
            CHECKPOLY
            if ( *in == '[' ) {
                if ( object == 1 ) {    /* after name */
                    t = out-1;
                    if ( *t == RPARENTHESIS ) {
                        *out++ = LBRACE; *out++ = LPARENTHESIS;
                        bracelevel++; explevel = bracelevel;
                    }
                    else {
                        while ( *t >= 0 && t > AC.tokens ) t--;
                        if ( *t == TEXPRESSION ) {
                            *out++ = LBRACE; *out++ = LPARENTHESIS;
                            bracelevel++; explevel = bracelevel;
                        }
                        else {*out++ = LBRACE; bracelevel++; }
                    }
                    object = 0;
                }
                else {                  /* name. find matching ] */
                    s = in;
                    in = SkipAName(in);
                    goto dovariable;
                }
            }
            else {
                if ( explevel > 0 && explevel == bracelevel ) {
                    *out++ = RPARENTHESIS; explevel = 0;
                }
                *out++ = RBRACE; object = 1; bracelevel--;
            }
            in++;
            break;
        case 4:     /* ( ) = ; , */
            if ( *in == '(' ) {
                if ( funlevel >= AM.MaxParLevel ) {
                    MesPrint("&More than %d levels of parentheses",AM.MaxParLevel);
                    return(-1);
                }
                if ( object == 1 ) {    /* After name -> function,vector */
                    AC.tokenarglevel[funlevel++] = TYPEISFUN;
                    *out++ = TFUNOPEN;
                    if ( polyflag ) {
                        if ( in[1] != ')' && in[1] != ',' ) {
                            *out++ = TNUMBER; *out++ = (SBYTE)(polyflag);
                            *out++ = TCOMMA;
                            *out++ = LPARENTHESIS;
                        }
                        else {
                            *out++ = LPARENTHESIS;
                            *out++ = TNUMBER; *out++ = (SBYTE)(polyflag);
                        }
                        polyflag = 0;
                    }
                    else if ( in[1] != ')' && in[1] != ',' ) {
                        *out++ = LPARENTHESIS;
                    }
                }
                else if ( object <= 0 ) {
                    CHECKPOLY
                    AC.tokenarglevel[funlevel++] = TYPEISSUB;
                    *out++ = LPARENTHESIS;
                }
                else {
                    polyflag = 0;
```

```
                        AC.tokenarglevel[funlevel++] = TYPEISMYSTERY;
                        MesPrint("&Illegal position for (: %s",in);
                        if ( error >= 0 ) error = -1;
                    }
                    object = -1;
                }
                else if ( *in == ')' ) {
                    funlevel--;
                    if ( funlevel < 0 ) {
/*                      if ( funflag == 0 ) { */
                            MesPrint("&There is an unmatched parenthesis");
                            if ( error >= 0 ) error = -1;
/*                      } */
                    }
                    else if ( object <= 0
                    && ( AC.tokenarglevel[funlevel] != TYPEISFUN
                    || out[-1] != TFUNOPEN ) ) {
                        MesPrint("&Illegal position for closing parenthesis.");
                        if ( error >= 0 ) error = -1;
                        if ( AC.tokenarglevel[funlevel] == TYPEISFUN ) object = 1;
                        else object = 3;
                    }
                    else {
                        if ( AC.tokenarglevel[funlevel] == TYPEISFUN ) {
                            if ( out[-1] == TFUNOPEN ) out--;
                            else {
                                if ( out[-1] != TCOMMA ) *out++ = RPARENTHESIS;
                                *out++ = TFUNCLOSE;
                            }
                            object = 1;
                        }
                        else if ( AC.tokenarglevel[funlevel] == TYPEISSUB ) {
                            *out++ = RPARENTHESIS;
                            object = 3;
                        }
                    }
                }
                else if ( *in == ',' ) {
                    if ( /* object > 0 && */ funlevel > 0 &&
                    AC.tokenarglevel[funlevel-1] == TYPEISFUN ) {
                        if ( out[-1] != TFUNOPEN && out[-1] != TCOMMA )
                            *out++ = RPARENTHESIS;
                        else { *out++ = TNUMBER; *out++ = 0; }
                        *out++ = TCOMMA;
                        if ( in[1] != ',' && in[1] != ')' )
                            *out++ = LPARENTHESIS;
                        else if ( in[1] == ')' ) {
                            *out++ = TNUMBER; *out++ = 0;
                        }
                    }
/*
                    else if ( object > 0 ) {
                    }
*/
                    else {
                        MesPrint("&Illegal position for comma: %s",in);
                        MesPrint("&Forgotten ; ?");
                        if ( error >= 0 ) error = -1;
                    }
                    object = -1;
                }
                else goto IllPos;
                in++;
                break;
            case 5:     /* + - * % / ^ : */
                CHECKPOLY
                if ( *in == ':' || *in == '%' ) goto IllPos;
                if ( *in == '*' || *in == '/' || *in == '^' ) {
                    if ( object <= 0 ) {
                        MesPrint("&Illegal position for operator: %s",in);
                        if ( error >= 0 ) error = -1;
                    }
                    else if ( *in == '*' ) *out++ = TMULTIPLY;
                    else if ( *in == '/' ) *out++ = TDIVIDE;
                    else                   *out++ = TPOWER;
                    in++;
                }
                else {
                    i = 1;
                    while ( *in == '+' || *in == '-' ) {
                        if ( *in == '-' ) i = -i;
                        in++;
                    }
                    if ( i == 1 ) {
                        if ( out > AC.tokens && out[-1] != TFUNOPEN &&
                        out[-1] != LPARENTHESIS && out[-1] != TCOMMA
                        && out[-1] != LBRACE )
```

```
                            *out++ = TPLUS;
                        }
                        else *out++ = TMINUS;
                    }
                    object = 0;
                    break;
                case 6:     /* Whitespace */
                    in++; break;
                case 7:     /* { | } */
                    CHECKPOLY
                    if ( *in == '{' ) {
                        if ( object > 0 ) {
                            MesPrint("&Illegal position for %s",in);
                            if ( !error ) error = 1;
                        }
                        s = in+1;
                        SKIPBRA2(in)
                        number = DoTempSet(s,in);
                        in++;
                        if ( number >= 0 ) {
                            *out++ = TSET;
                            i = 0;
                            do { num[i++] = (SBYTE)(number & 0x7F); number >>= 7; } while ( number );
                            while ( --i >= 0 ) *out++ = num[i];
                        }
                        else if ( error == 0 ) error = 1;
                        object = 1;
                    }
                    else goto IllPos;
                    break;
                case 8:     /* ! & < > */
                    CHECKPOLY
                    if ( *in != '!' || leftright == RHSIDE
                    || object != 1 || out[-1] != TWILDCARD ) goto IllPos;
                    *out++ = TNOT;
                    if ( FG.cTable[in[1]] == 0 || in[1] == '[' || in[1] == '{' ) object = 0;
                    in++;
                    break;
                default:
IllPos:             MesPrint("&Illegal character at this position: %s",in);
                    if ( error >= 0 ) error = -1;
                    in++;
                    polyflag = 0;
                    break;
            }
        }
        *out++ = TENDOFIT;
        AC.endoftokens = out;
        if ( funlevel > 0 || bracelevel != 0 ) {
            if ( funlevel > 0 ) MesPrint("&Unmatched parentheses");
            if ( bracelevel != 0 ) MesPrint("&Unmatched braces");
            return(-1);
        }
        if ( AC.TokensWriteFlag ) WriteTokens(AC.tokens);
/*
        Simplify fixed set elements
*/
        if ( error == 0 && simp1token(AC.tokens) ) error = 1;
/*
        Collect wildcards for the prototype. Symplify the leftover wildcards
*/
        if ( error == 0 && leftright == LHSIDE && simpwtoken(AC.tokens) )
                    error = 1;
/*
        Now prepare the set[n] objects in the RHS.
*/
        if ( error == 0 && leftright == RHSIDE && simp4token(AC.tokens) )
                    error = 1;
/*
        Simplify simple function arguments (and 1/fac_ and 1/invfac_)
*/
        if ( error == 0 && simp2token(AC.tokens) ) error = 1;
/*
        Next we try to remove composite denominators or exponents and
        replace them by their internal functions. This may involve expanding
        the buffer. The return code of 3a is negative if there is an error
        and positive if indeed we need to do some work.
        simp3btoken does the work
*/
        numexp = 0;
        if ( error == 0 && ( numexp = simp3atoken(AC.tokens,leftright) ) < 0 )
            error = 1;
        if ( numexp > 0 ) {
            SBYTE *tt;
            out = AC.tokens;
            while ( *out != TENDOFIT ) out++;
            while ( out+numexp*9+20 > outtop ) {
```

```
            LONG oldsize = (LONG)(out - AC.tokens);
            SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
            SBYTE **pppp = &(AC.toptokens);
            DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"out tokens");
            out = AC.tokens + oldsize;
            outtop = AC.toptokens - MAXNUMSIZE;
        }
        tt = out + numexp*9+20;
        while ( out >= AC.tokens ) { *tt-- = *out--; }
        while ( tt >= AC.tokens ) { *tt-- = TEMPTY; }
        if ( error == 0 && simp3btoken(AC.tokens,leftright) ) error = 1;
        if ( error == 0 && simp2token(AC.tokens) ) error = 1;
    }
/*
    In simp5token we test for special cases like sumvariables that are
    already wildcards, etc.
*/
    if ( error == 0 && simp5token(AC.tokens,leftright) ) error = 1;
/*
    In simp6token we test for special cases like factorized expressions
    that occur in the RHS in an improper way.
*/
    if ( error == 0 && simp6token(AC.tokens,leftright) ) error = 1;

    return(error);
}

/*
        #] tokenize :
        #[ WriteTokens :
*/

char *ttypes[] = { "\n", "S", "I", "V", "F", "set", "E", "dotp", "#",
   "sub", "d_", "$", "dub", "(", ")", "?", "??", ".", "[", "]",
   ",", "((", ))", "*", "/", "^", "+", "-", "!", "end", "{{", "}}",
   "N_?", "conj", "()", "#d", "^d", "_", "snum"  };

void WriteTokens(SBYTE *in)
{
    int numinline = 0, x, n = sizeof(ttypes)/sizeof(char *);
    char outbuf[81], *s, *out, c;
    out = outbuf;
    while ( *in != TENDOFIT ) {
        if ( *in < 0 ) {
            if ( *in >= -n ) {
                s = ttypes[-*in];
                while ( *s ) { *out++ = *s++; numinline++; }
            }
            else {
                *out++ = '-'; x = -*in; numinline++;
                goto writenumber;
            }
        }
        else {
            x = *in;
writenumber:
            s = out;
            do {
                *out++ = (char)(( x % 10 ) + '0');
                numinline++;
                x = x / 10;
            } while ( x );
            c = out[-1]; out[-1] = *s; *s = c;
        }
        if ( numinline > 70 ) {
            *out = 0;
            MesPrint("%s",outbuf);
            out = outbuf; numinline = 0;
        }
        else {
            *out++ = ' '; numinline++;
        }
        in++;
    }
    if ( numinline > 0 ) { *out = 0; MesPrint("%s",outbuf); }
}

/*
        #] WriteTokens :
        #[ simp1token :

        Routine substitutes set elements if possible.
        This means sets with a fixed argument like setname[3].
*/

int simp1token(SBYTE *s)
{
```

```
    int error = 0, n, i, base;
    WORD numsub;
    SBYTE *fill = s, *start, *t, numtab[10];
    SETS set;
    while ( *s != TENDOFIT ) {
        if ( *s == RBRACE ) {
            start = fill-1;
            while ( *start != LBRACE ) start--;
            t = start - 1;
            while ( *t >= 0 ) t--;
            if ( *t == TSET && ( start[1] == TNUMBER || start[1] == TNUMBER1 ) ) {
                base = start[1] == TNUMBER ? 100: 128;
                start += 2;
                numsub = *start++;
                while ( *start >= 0 && start < fill )
                    { numsub = base*numsub + *start++; }
                if ( start == fill ) {
                    start = t;
                    t++; n = *t++; while ( *t >= 0 ) { n = 128*n + *t++; }
                    set = Sets+n;
                    if ( ( set->type != CRANGE )
                    && ( numsub > 0 && numsub <= set->last-set->first ) ) {
                        fill = start;
                        n = SetElements[set->first+numsub-1];
                        switch (set->type) {
                            case CSYMBOL:
                                if ( n > MAXPOWER ) {
                                    n -= 2*MAXPOWER;
                                    if ( n < 0 ) { n = -n; *fill++ = TMINUS; }
                                    *fill++ = TNUMBER1;
                                }
                                else *fill++ = TSYMBOL;
                                break;
                            case CINDEX:
                                if ( n < AM.OffsetIndex ) *fill++ = TNUMBER1;
                                else {
                                    *fill++ = TINDEX;
                                    n -= AM.OffsetIndex;
                                }
                                break;
                            case CVECTOR:   *fill++ = TVECTOR;
                                n -= AM.OffsetVector;   break;
                            case CFUNCTION: *fill++ = TFUNCTION;
                                n -= FUNCTION; break;
                            case CNUMBER:   *fill++ = TNUMBER1;  break;
                            case CDUBIOUS:  *fill++ = TDUBIOUS; n = 1; break;
                        }
                        i = 0;
if ( n < 0 ) {
    MesPrint("Value of n = %d",n);
}
                        do { numtab[i++] = (SBYTE)(n & 0x7F); n >>= 7; } while ( n );
                        while ( --i >= 0 ) *fill++ = numtab[i];
                    }
                    else {
                        MesPrint("&Illegal element %d in set",numsub);
                        error++;
                    }
                    s++; continue;
                }
            }
            *fill++ = *s++;
        }
        else *fill++ = *s++;
    }
    *fill++ = TENDOFIT;
    return(error);
}

/*
        #] simp1token :
        #[ simpwtoken :

        Only to be called in the LHS.
        Hunts down the wildcards and writes them to the wildcardbuffer.
        Next it causes the ProtoType to be constructed.
        All wildcards are simplified into the trailing TWILDCARD,
        because the specifics are stored in the prototype.
        These specifics also include the transfer of wildcard values
        to $variables.

        Types of wildcards:
        a?, a?set, a?!set, a?set[i], A?set1?set2, ?a
        After this we can strip the set information.
        We still need the ? because of the wildcarding offset in code generation
*/
```

```
int simpwtoken(SBYTE *s)
{
    int error = 0, first = 1, notflag;
    WORD num, numto, numdollar, *w = AC.WildC, *wstart, *wtop;
    SBYTE *fill = s, *t, *v, *s0 = s;
    while ( *s != TENDOFIT ) {
        if ( *s == TWILDCARD ) {
            notflag = 0; t = fill;
            while ( t > s0 && t[-1] >= 0 ) t--;
            v = t; num = 0; *fill++ = *s++;
            while ( *v >= 0 ) num = 128*num + *v++;
            if ( t > s0 ) t--;
            AC.NwildC += 4;
            if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
            switch ( *t ) {
                case TSYMBOL:
                case TDUBIOUS:
                    *w++ = SYMTOSYM; *w++ = 4; *w++ = num; *w++ = num; break;
                case TINDEX:
                    num += AM.OffsetIndex;
                    *w++ = INDTOIND; *w++ = 4; *w++ = num; *w++ = num;  break;
                case TVECTOR:
                    num += AM.OffsetVector;
                    *w++ = VECTOVEC; *w++ = 4; *w++ = num; *w++ = num;  break;
                case TFUNCTION:
                    num += FUNCTION;
                    *w++ = FUNTOFUN; *w++ = 4; *w++ = num; *w++ = num;  break;
                default:
                    MesPrint("&Illegal type of wildcard in LHS");
                    error = -1;
                    *w++ = SYMTOSYM; *w++ = 4; *w++ = num; *w++ = num;  break;
                    break;
            }
/*
            Now the sets. The s pointer sits after the ?
*/
            wstart = w;
            if ( *s == TNOT && s[1] == TSET ) { notflag = 1; s++; }
            if ( *s == TSET ) {
                s++; num = 0; while ( *s >= 0 ) num = 128*num + *s++;
                if ( notflag == 0 && *s == TWILDCARD && s[1] == TSET ) {
                    s += 2; numto = 0; while ( *s >= 0 ) numto = 128*numto + *s++;
                    if ( num < AM.NumFixedSets || numto < AM.NumFixedSets
                    || Sets[num].type == CRANGE || Sets[numto].type == CRANGE ) {
                        MesPrint("&This type of set not allowed in this wildcard construction");
                        error = 1;
                    }
                    else {
                    AC.NwildC += 4;
                    if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
                    *w++ = FROMSET; *w++ = 4; *w++ = num; *w++ = numto;
                    wstart = w;
                    }
                }
                else if ( notflag == 0 && *s == LBRACE && s[1] == TSYMBOL ) {
                    if ( num < AM.NumFixedSets || Sets[num].type == CRANGE ) {
                        MesPrint("&This type of set not allowed in this wildcard construction");
                        error = 1;
                    }
                    v = s; s += 2;
                    numto = 0; while ( *s >= 0 ) numto = 128*numto + *s++;
                    if ( *s == TWILDCARD ) s++; /* most common mistake */
                    if ( *s == RBRACE ) {
                        s++;
                        AC.NwildC += 8;
                        if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
                        *w++ = SETTONUM; *w++ = 4; *w++ = num; *w++ = numto;
                        wstart = w;
                        *w++ = SYMTOSYM; *w++ = 4; *w++ = numto; *w++ = 0;
                    }
                    else if ( *s == TDOLLAR ) {
                        s++; numdollar = 0;
                        while ( *s >= 0 ) numdollar = 128*numdollar + *s++;
                        if ( *s == RBRACE ) {
                            s++;
                            AC.NwildC += 12;
                            if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
                            *w++ = SETTONUM; *w++ = 4; *w++ = num; *w++ = numto;
                            wstart = w;
                            *w++ = SYMTOSYM; *w++ = 4; *w++ = numto; *w++ = 0;
                            *w++ = LOADDOLLAR; *w++ = 4; *w++ = numdollar;
                            *w++ = numdollar;
                        }
                        else { s = v; goto singlewild; }
                    }
                    else { s = v; goto singlewild; }
                }
```

```
                else {
singlewild:         num += notflag * 2*WILDOFFSET;
                    AC.NwildC += 4;
                    if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
                    *w++ = FROMSET; *w++ = 4; *w++ = num; *w++ = -WILDOFFSET;
                    wstart = w;
                }
            }
            else if ( *s != TDOLLAR && *s != TENDOFIT && *s != RPARENTHESIS
            && *s != RBRACE && *s != TCOMMA && *s != TFUNCLOSE && *s != TMULTIPLY
            && *s != TPOWER && *s != TDIVIDE && *s != TPLUS && *s != TMINUS
            && *s != TPOWER1 && *s != TEMPTY && *s != TFUNOPEN && *s != TDOT ) {
                MesPrint("&Illegal type of wildcard in LHS");
                error = -1;
            }
            if ( *s == TDOLLAR ) {
                s++; numdollar = 0;
                while ( *s >= 0 ) numdollar = 128*numdollar + *s++;
                AC.NwildC += 4;
                if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
                wtop = w + 4;
                if ( wstart < w ) {
                    while ( w > wstart ) { w[4] = w[0]; w--; }
                }
                *w++ = LOADDOLLAR; *w++ = 4; *w++ = numdollar; *w++ = numdollar;
                w = wtop;
            }
        }
        else if ( *s == TWILDARG ) {
            *fill++ = *s++;
            num = 0;
            while ( *s >= 0 ) { num = 128*num + *s;  *fill++ = *s++; }
            AC.NwildC += 4;
            if ( AC.NwildC > 4*AM.MaxWildcards ) {
firsterr:       if ( first ) {
                    MesPrint("&More than %d wildcards",AM.MaxWildcards);
                    error = -1;
                    first = 0;
                }
            }
            else { *w++ = ARGTOARG; *w++ = 4; *w++ = num; *w++ = -1; }
            if ( *s == TDOLLAR ) {
                s++; num = 0; while ( *s >= 0 ) num = 128*num + *s++;
                AC.NwildC += 4;
                if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
                *w++ = LOADDOLLAR; *w++ = 4; *w++ = num; *w++ = num;
            }
        }
        else *fill++ = *s++;
    }
    *fill++ = TENDOFIT;
    AC.WildC = w;
    return(error);
}

/*
        #] simpwtoken :
        #[ simp2token :

        Deals with function arguments.
        The tokenizer has given function arguments extra parentheses.
        We remove the double parentheses.
        Next we remove the parentheses around the simple arguments.

        It also replaces /fac_() by *invfac_() and /invfac_() by *fac_()
*/

int simp2token(SBYTE *s)
{
    SBYTE *to, *fill, *t, *v, *w, *s0 = s, *vv;
    int error = 0, n;
/*
    Set substitutions
*/
    fill = to = s;
    while ( *s != TENDOFIT ) {
        if ( *s == LPARENTHESIS && s[1] == LPARENTHESIS ) {
            t = s+1; n = 0;
            while ( n >= 0 ) {
                t++;
                if ( *t == LPARENTHESIS ) n++;
                else if ( *t == RPARENTHESIS ) n--;
            }
            if ( t[1] == RPARENTHESIS ) {
                *t = TEMPTY; s++;
            }
            *fill++ = *s++;
```

```
        }
        else if ( *s == TEMPTY ) s++;
        else if ( *s == AM.facnum && ( fill > (s0+1) ) && fill[-2] == TDIVIDE
         && fill[-1] == TFUNCTION ) {
            fill[-2] = TMULTIPLY; *fill++ = (SBYTE)(AM.invfacnum); s++;
        }
        else if ( *s == AM.invfacnum && ( fill > (s0+1) ) && fill[-2] == TDIVIDE
         && fill[-1] == TFUNCTION ) {
            fill[-2] = TMULTIPLY; *fill++ = (SBYTE)(AM.facnum); s++;
        }
        else *fill++ = *s++;
    }
    *fill++ = TENDOFIT;
/*
    Second round: try to locate 'simple' arguments and strip their brackets

    We add (9-feb-2010) to the simple arguments integers of any size
*/
    fill = s = to;
    while ( *s != TENDOFIT ) {
        if ( *s == LPARENTHESIS ) {
            t = s; n = 0;
            while ( n >= 0 ) {
                t++;
                if ( *t == LPARENTHESIS ) n++;
                else if ( *t == RPARENTHESIS ) n--;
            }
            if ( t[1] == TFUNCLOSE && s[1] != TWILDARG ) { /* Check for last argument in sum */
                v = fill - 1; n = 0;
                while ( n >= 0 && v >= to ) {
                    if ( *v == TFUNOPEN ) n--;
                    else if ( *v == TFUNCLOSE ) n++;
                    v--;
                }
                if ( v > to ) {
                    while ( *v >= 0 ) v--;
                    if ( *v == TFUNCTION ) { v++;
                        n = 0; while ( *v >= 0 && v < fill ) n = 128*n + *v++;
                        if ( n == AM.sumnum || n == AM.sumpnum ) {
                            *fill++ = *s++; continue;
                        }
                        else if ( ( n == (FIRSTBRACKET-FUNCTION)
                        || n == (TERMSINEXPR-FUNCTION)
                        || n == (NUMFACTORS-FUNCTION)
                        || n == (GCDFUNCTION-FUNCTION)
                        || n == (DIVFUNCTION-FUNCTION)
                        || n == (REMFUNCTION-FUNCTION)
                        || n == (INVERSEFUNCTION-FUNCTION)
                        || n == (MULFUNCTION-FUNCTION)
                        || n == (FACTORIN-FUNCTION)
                        || n == (FIRSTTERM-FUNCTION)
                        || n == (CONTENTTERM-FUNCTION) )
                        && fill[-1] == TFUNOPEN ) {
                            v = s+1;
                            if ( *v == TEXPRESSION ) {
                                v++;
                                n = 0; while ( *v >= 0 ) n = 128*n + *v++;
                                if ( v == t ) {
                                    *t = TEMPTY; s++;
                                }
                            }
                        }
                    }
                }
            }
            if ( ( fill > to )
             && ( ( fill[-1] == TFUNOPEN || fill[-1] == TCOMMA )
               && ( t[1] == TFUNCLOSE || t[1] == TCOMMA ) ) ) {
                v = s + 1;
                switch ( *v ) {
                    case TMINUS:
                        v++;
                        if ( *v == TVECTOR ) {
                            w = v+1; while ( *w >= 0 ) w++;
                            if ( w == t ) {
                                *t = TEMPTY; s++;
                            }
                        }
                        else {
                            if ( *v == TNUMBER || *v == TNUMBER1 ) {
                              if ( BITSINWORD == 16 ) { LONG x; WORD base;
                                base = ( *v == TNUMBER ) ? 100: 128;
                                vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
                                if ( ( vv != t ) || ( ( vv - v ) > 4 ) || ( x > (MAXPOSITIVE+1) ) )
                                    *fill++ = *s++;
                                else { *t = TEMPTY; s++; break; }
                              }
```

```
                                else if ( BITSINWORD == 32 ) { LONG x; WORD base;
                                  base = ( *v == TNUMBER ) ? 100: 128;
                                  vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
                                  if ( ( vv != t ) || ( ( vv - v ) > 6 ) || ( x > (MAXPOSITIVE+1) ) )
                                      *fill++ = *s++;
                                  else { *t = TEMPTY; s++; break; }
                                }
                                else {
                                  if ( ( v+2 == t ) || ( v+3 == t && v[2] >= 0 ) )
                                      { *t = TEMPTY; s++; break; }
                                  else *fill++ = *s++;
                                }
                              }
                              else if ( *v == LPARENTHESIS && t[-1] == RPARENTHESIS ) {
                                  w = v; n = 0;
                                  while ( n >= 0 ) {
                                      w++;
                                      if ( *w == LPARENTHESIS ) n++;
                                      else if ( *w == RPARENTHESIS ) n--;
                                  }
                                  if ( w == ( t-1 ) ) { *t = TEMPTY; s++; }
                                  else *fill++ = *s++;
                              }
                              else *fill++ = *s++;
                              break;
                          }
                      case TSETNUM:
                          v++; while ( *v >= 0 ) v++;
                          goto tcommon;
                      case TSYMBOL:
                          if ( ( v[1] == COEFFSYMBOL || v[1] == NUMERATORSYMBOL
                          || v[1] == DENOMINATORSYMBOL ) && v[2] < 0 ) {
                              *fill++ = *s++; break;
                          }
                      case TVECTOR:
                      case TINDEX:
                      case TFUNCTION:
                      case TDOLLAR:
                      case TDUBIOUS:
                      case TSGAMMA:
tcommon:                  v++; while ( *v >= 0 ) v++;
                          if ( v == t || ( v[0] == TWILDCARD && v+1 == t ) )
                          { *t = TEMPTY; s++; }
                          else *fill++ = *s++;
                          break;
                      case TGENINDEX:
                          v++;
                          if ( v == t ) { *t = TEMPTY; s++; }
                          else *fill++ = *s++;
                          break;
                      case TNUMBER:
                      case TNUMBER1:
                          if ( BITSINWORD == 16 ) { LONG x; WORD base;
                              base = ( *v == TNUMBER ) ? 100: 128;
                              vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
                              if ( ( vv != t ) || ( ( vv - v ) > 4 ) || ( x > MAXPOSITIVE ) )
                                  *fill++ = *s++;
                              else { *t = TEMPTY; s++; break; }
                          }
                          else if ( BITSINWORD == 32 ) { LONG x; WORD base;
                              base = ( *v == TNUMBER ) ? 100: 128;
                              vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
                              if ( ( vv != t ) || ( ( vv - v ) > 6 ) || ( x > MAXPOSITIVE ) )
                                  *fill++ = *s++;
                              else { *t = TEMPTY; s++; break; }
                          }
                          else {
                              if ( ( v+2 == t ) || ( v+3 == t && v[2] >= 0 ) )
                                  { *t = TEMPTY; s++; break; }
                              else *fill++ = *s++;
                          }
                          break;
                      case TWILDARG:
                          v++; while ( *v >= 0 ) v++;
                          if ( v == t ) { *t = TEMPTY; s++; }
                          else *fill++ = *s++;
                          break;
                      case TEXPRESSION:
/*
                          First establish that there is only the expression
                          in this argument.
*/
                          vv = s+1;
                          while ( vv < t ) {
                              if ( *vv != TEXPRESSION ) break;
                              vv++; while ( *vv >= 0 ) vv++;
                          }
```

```
                        if ( vv < t ) { *fill++ = *s++; break; }
/*
                        Find the function
*/
                        w = fill-1; n = 0;
                        while ( n >= 0 && w >= to ) {
                            if ( *w == TFUNOPEN ) n--;
                            else if ( *w == TFUNCLOSE ) n++;
                            w--;
                        }
                        w--; while ( w > to && *w >= 0 ) w--;
                        if ( *w != TFUNCTION ) { *fill++ = *s++; break; }
                        w++; n = 0;
                        while ( *w >= 0 ) { n = 128*n + *w++; }
                        if ( n == GCDFUNCTION-FUNCTION
                        || n == DIVFUNCTION-FUNCTION
                        || n == REMFUNCTION-FUNCTION
                        || n == INVERSEFUNCTION-FUNCTION
                        || n == MULFUNCTION-FUNCTION ) {
                            *t = TEMPTY; s++;
                        }
                        else *fill++ = *s++;
                        break;
                    default: *fill++ = *s++; break;
                }
            }
            else *fill++ = *s++;
        }
        else if ( *s == TEMPTY ) s++;
        else *fill++ = *s++;
    }
    *fill++ = TENDOFIT;
    return(error);
}

/*
        #] simp2token :
        #[ simp3atoken :

        We hunt for denominators and exponents that seem hidden.
        For the denominators we have to recognize:
            /fun  /fun()  /fun^power  /fun()^power
            /set[n]  /set[n]()  /set[n]^power  /set[n]()^power
            /symbol^power (power no number or symbol wildcard)
            /dotpr^power (id)
            /#^power (id)
            /()  /()^power
            /vect /index /vect(anything) /vect(anything)^power
*/

int simp3atoken(SBYTE *s, int mode)
{
    int error = 0, n, numexp = 0, denom, base, numprot, i;
    SBYTE *t, c;
    LONG num;
    WORD *prot;
    if ( mode == RHSIDE ) {
        prot = AC.ProtoType;
        numprot = prot[1] - SUBEXPSIZE;
        prot += SUBEXPSIZE;
    }
    else { prot = 0; numprot = 0; }
    while ( *s != TENDOFIT ) {
        denom = 1;
        if ( *s == TDIVIDE ) { denom = -1; s++; }
        c = *s;
        switch(c) {
            case TSYMBOL:
            case TNUMBER:
            case TNUMBER1:
                s++; while ( *s >= 0 ) s++; /* skip the object */
                if ( *s == TWILDCARD ) s++; /* and the possible wildcard */
dosymbol:
                if ( *s != TPOWER ) continue;   /* No power -> done */
                s++;                            /* Skip the power */
                if ( *s == TMINUS ) s++;        /* negative: no difference here */
                if ( *s == TNUMBER || *s == TNUMBER1 ) {
                    base = *s == TNUMBER ? 100: 128; /* NUMBER = base 100 */
                    s++;                /* Now we compose the power */
                    num = *s++;         /* If the number is way too large */
                    while ( *s >= 0 ) { /* it may look like not too big */
                        if ( num > MAXPOWER ) break;    /* Hence... */
                        num = base*num + *s++;
                    }
                    while ( *s >= 0 ) s++;  /* Finish the number if needed */
                    if ( *s == TPOWER ) goto doublepower;
                    if ( num <= MAXPOWER ) continue;    /* Simple case */
```

```
                }
                else if ( *s == TSYMBOL && c != TNUMBER && c != TNUMBER1 ) {
                    s++; n = 0; while ( *s >= 0 ) { n = 128*n + *s++; }
                    if ( *s == TWILDCARD ) { s++;
                        if ( *s == TPOWER ) goto doublepower;
                        continue; }
/*
                    Now we have to test whether n happens to be a wildcard
*/
                    if ( mode == RHSIDE ) {
                        n += 2*MAXPOWER;
                        for ( i = 0; i < numprot; i += 4 ) {
                            if ( prot[i+2] == n && prot[i] == SYMTOSYM ) break;
                        }
                        if ( i < numprot ) break;
                    }
                    if ( *s == TPOWER ) goto doublepower;
                }
                numexp++;
                break;
            case TINDEX:
                s++; while ( *s >= 0 ) s++;
                if ( *s == TWILDCARD ) s++;
doindex:
                if ( denom < 0 || *s == TPOWER ) {
                    MesPrint("&Index to a power or in denominator is illegal");
                    error = 1;
                }
                break;
            case TVECTOR:
                s++; while ( *s >= 0 ) s++;
                if ( *s == TWILDCARD ) s++;
dovector:
                if ( *s == TFUNOPEN ) {
                    s++; n = 1;
                    for(;;) {
                        if ( *s == TFUNOPEN ) {
                            n++;
                            MesPrint("&Illegal vector index");
                            error = 1;
                        }
                        else if ( *s == TFUNCLOSE ) {
                            n--;
                            if ( n <= 0 ) break;
                        }
                        s++;
                    }
                    s++;
                }
                else if ( *s == TDOT ) goto dodot;
                if ( denom < 0 || *s == TPOWER || *s == TPOWER1 ) numexp++;
                break;
            case TFUNCTION:
                s++; while ( *s >= 0 ) s++;
                if ( *s == TWILDCARD ) s++;
dofunction:
                t = s;
                if ( *t == TFUNOPEN ) {
                    t++; n = 1;
                    for(;;) {
                        if ( *t == TFUNOPEN ) n++;
                        else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
                        t++;
                    }
                    t++; s++;
                }
                if ( denom < 0 || *t == TPOWER || *t == TPOWER1 ) numexp++;
                break;
            case TEXPRESSION:
                s++; while ( *s >= 0 ) s++;
                t = s;
                if ( *t == TFUNOPEN ) {
                    t++; n = 1;
                    for(;;) {
                        if ( *t == TFUNOPEN ) n++;
                        else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
                        t++;
                    }
                    t++;
                }
                if ( *t == LBRACE ) {
                    t++; n = 1;
                    for(;;) {
                        if ( *t == LBRACE ) n++;
                        else if ( *t == RBRACE ) { if ( --n <= 0 ) break; }
                        t++;
                    }
```

```
                    t++;
                }
                if ( denom < 0 || ( ( *t == TPOWER || *t == TPOWER1 )
                && t[1] == TMINUS ) ) numexp++;
                break;
            case TDOLLAR:
                s++; while ( *s >= 0 ) s++;
                if ( denom < 0 || ( ( *s == TPOWER || *s == TPOWER1 )
                && s[1] == TMINUS ) ) numexp++;
                break;
            case LPARENTHESIS:
                s++; n = 1; t = s;
                for(;;) {
                    if ( *t == LPARENTHESIS ) n++;
                    else if ( *t == RPARENTHESIS ) { if ( --n <= 0 ) break; }
                    t++;
                }
                t++;
                if ( denom > 0 && ( *t == TPOWER || *t == TPOWER1 ) ) {
                    if ( ( t[1] == TNUMBER || t[1] == TNUMBER1 ) && t[2] >= 0
                    && t[3] < 0 ) break;
                    numexp++;
                }
                else if ( denom < 0 && ( *t == TPOWER || *t == TPOWER1 ) ) {
                    if ( t[1] == TMINUS && ( t[2] == TNUMBER
                     || t[2] == TNUMBER1 ) && t[3] >= 0
                    && t[4] < 0 ) break;
                    numexp++;
                }
                else if ( denom < 0 || ( ( *t == TPOWER || *t == TPOWER1 )
                && ( t[1] == TMINUS || t[1] == LPARENTHESIS ) ) ) numexp++;
                break;
            case TSET:
                s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
                n = Sets[n].type;
                switch ( n ) {
                    case CSYMBOL: goto dosymbol;
                    case CINDEX: goto doindex;
                    case CVECTOR: goto dovector;
                    case CFUNCTION: goto dofunction;
                    case CNUMBER: goto dosymbol;
                    default: error = 1; break;
                }
                break;
            case TDOT:
dodot:          s++;
                if ( *s == TVECTOR ) { s++; while ( *s >= 0 ) s++; }
                else if ( *s == TSET ) {
                    s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
                    if ( Sets[n].type != CVECTOR ) {
                        MesPrint("&Set in dotproduct is not a set of vectors");
                        error = 1;
                    }
                    if ( *s == LBRACE ) {
                        s++; n = 1;
                        for(;;) {
                            if ( *s == LBRACE ) n++;
                            else if ( *s == RBRACE ) { if ( --n <= 0 ) break; }
                            s++;
                        }
                        s++;
                    }
                    else {
                        MesPrint("&Set without argument in dotproduct");
                        error = 1;
                    }
                }
                else if ( *s == TSETNUM ) {
                    s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
                    if ( *s != TVECTOR ) goto nodot;
                    s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
                    if ( Sets[n].type != CVECTOR ) {
                        MesPrint("&Set in dotproduct is not a set of vectors");
                        error = 1;
                    }
                }
                else {
nodot:              MesPrint("&Illegal second element in dotproduct");
                    error = 1;
                    s++; while ( *s >= 0 ) s++;
                }
                goto dosymbol;
            default:
                s++; while ( *s >= 0 ) s++;
                break;
        }
    }
```

```
    if ( error ) return(-1);
    return(numexp);
doublepower:
    MesPrint("&Dubious notation with object^power1^power2");
    return(-1);
}

/*
        #] simp3atoken :
        #[ simp3btoken :
*/

int simp3btoken(SBYTE *s, int mode)
{
    int error = 0, i, numprot, n, denom, base, inset = 0, dotp, sube = 0;
    SBYTE *t, c, *fill, *ff, *ss;
    LONG num;
    WORD *prot;
    if ( mode == RHSIDE ) {
        prot = AC.ProtoType;
        numprot = prot[1] - SUBEXPSIZE;
        prot += SUBEXPSIZE;
    }
    else { prot = 0; numprot = 0; }
    fill = s;
    while ( *s == TEMPTY ) s++;
    while ( *s != TENDOFIT ) {
        if ( *s == TEMPTY ) { s++; continue; }
        denom = 1;
        if ( *s == TDIVIDE ) { denom = -1; *fill++ = *s++; }
        ff = fill; ss = s; c = *s;
        if ( c == TSETNUM ) {
            *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
            c = *s;
        }
        dotp = 0;
        switch(c) {
            case TSYMBOL:
            case TNUMBER:
            case TNUMBER1:
                *fill++ = *s++;
                while ( *s >= 0 ) *fill++ = *s++;
                if ( *s == TWILDCARD ) *fill++ = *s++;
dosymbol:
                t = s;
                if ( *s != TPOWER ) continue;
                *fill++ = *s++;
                if ( *s == TMINUS ) *fill++ = *s++;
                if ( *s == TPLUS ) s++;
                if ( *s == TSETNUM ) {
                    *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
                    inset = 1;
                }
                else inset = 0;
                if ( *s == TNUMBER || *s == TNUMBER1 ) {
                    base = *s == TNUMBER ? 100: 128;
                    *fill++ = *s++;
                    num = *s++; *fill++ = num;
                    while ( *s >= 0 ) {
                        if ( num > MAXPOWER ) break;
                        *fill++ = *s;
                        num = base*num + *s++;
                    }
                    while ( *s >= 0 ) *fill++ = *s++;
                    if ( num <= MAXPOWER ) continue;
                    goto putexp1;
                }
                else if ( *s == TSYMBOL && c != TNUMBER && c != TNUMBER1 ) {
                    *fill++ = *s++;
                    n = 0; while ( *s >= 0 ) { n = 128*n + *s; *fill++ = *s++; }
                    if ( *s == TWILDCARD ) { *fill++ = *s++;
                        if ( *s == TPOWER ) goto doublepower;
                    break; }
/*
                    Now we have to test whether n happens to be a wildcard
*/
                    if ( mode == RHSIDE && inset == 0 ) {
/*                      n += WILDOFFSET;*/
                        for ( i = 0; i < numprot; i += 4 ) {
                            if ( prot[i+2] == n && prot[i] == SYMTOSYM ) break;
                        }
                        if ( i < numprot ) break;
                    }

putexp1:            fill = ff;
                    if ( denom < 0 ) fill[-1] = TMULTIPLY;
                    *fill++ = TFUNCTION; *fill++ = (SBYTE)(AM.expnum); *fill++ = TFUNOPEN;
```

```
                    if ( dotp ) *fill++ = LPARENTHESIS;
                    while ( ss < t ) *fill++ = *ss++;
                    if ( dotp ) *fill++ = RPARENTHESIS;
                    *fill++ = TCOMMA;
                    ss++;                       /* Skip TPOWER */
                    if ( *ss == TMINUS ) { denom = -denom; ss++; }
                    if ( denom < 0 ) {
                        *fill++ = LPARENTHESIS;
                        *fill++ = TMINUS;
                        while ( ss < s ) *fill++ = *ss++;
                        *fill++ = RPARENTHESIS;
                    }
                    else {
                        while ( ss < s ) *fill++ = *ss++;
                    }
                    *fill++ = TFUNCLOSE;
                    if ( *ss == TPOWER ) goto doublepower;
                }
                else {  /* other objects can be composite */
                    goto dofunpower;
                }
                break;
            case TINDEX:
                *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
                if ( *s == TWILDCARD ) *fill++ = *s++;
                break;
            case TVECTOR:
                *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
                if ( *s == TWILDCARD ) *fill++ = *s++;
dovector:
                if ( *s == TFUNOPEN ) {
                    while ( *s != TFUNCLOSE ) *fill++ = *s++;
                    *fill++ = *s++;
                }
                else if ( *s == TDOT ) goto dodot;
                t = s;
                goto dofunpower;
            case TFUNCTION:
                *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
                if ( *s == TWILDCARD ) *fill++ = *s++;
dofunction:
                t = s;
                if ( *t == TFUNOPEN ) {
                    t++; n = 1;
                    for(;;) {
                        if ( *t == TFUNOPEN ) n++;
                        else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
                        t++;
                    }
                    t++; *fill++ = *s++;
                }
                sube = 0;
dofunpower:
                if ( *t == TPOWER || *t == TPOWER1 ) {
                    if ( sube ) {
                        if ( ( t[1] == TNUMBER || t[1] == TNUMBER1 )
                         && denom > 0 ) {
                            if ( t[2] >= 0 && t[3] < 0 ) { sube = 0; break; }
                        }
                        else if ( t[1] == TMINUS && denom < 0 &&
                        ( t[2] == TNUMBER || t[2] == TNUMBER1 ) ) {
                            if ( t[2] >= 0 && t[3] < 0 ) { sube = 0; break; }
                        }
                        sube = 0;
                    }
                    fill = ff;
                    *fill++ = TFUNCTION; *fill++ = (SBYTE)(AM.expnum); *fill++ = TFUNOPEN;
                    *fill++ = LPARENTHESIS;
                    while ( ss < t ) *fill++ = *ss++;
                    t++;
                    *fill++ = RPARENTHESIS; *fill++ = TCOMMA;
                    if ( *t == TMINUS ) { t++; denom = -denom; }
                    *fill++ = LPARENTHESIS;
                    if ( denom < 0 ) *fill++ = TMINUS;
                    if ( *t == LPARENTHESIS ) {
                        *fill++ = *t++; n = 0;
                        while ( n >= 0 ) {
                            if ( *t == LPARENTHESIS ) n++;
                            else if ( *t == RPARENTHESIS ) n--;
                            *fill++ = *t++;
                        }
                    }
                    else if ( *t == TFUNCTION || *t == TDUBIOUS ) {
                        *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
                        if ( *t == TWILDCARD ) *fill++ = *t++;
                        if ( *t == TFUNOPEN ) {
                            *fill++ = *t++; n = 0;
```

```
                        while ( n >= 0 ) {
                            if ( *t == TFUNOPEN ) n++;
                            else if ( *t == TFUNCLOSE ) n--;
                            *fill++ = *t++;
                        }
                    }
                }
                else if ( *t == TSET ) {
                    *fill++ = *t++; n = 0;
                    while ( *t >= 0 ) { n = 128*n + *t; *fill++ = *t++; }
                    if ( *t == LBRACE ) {
                        if ( n < AM.NumFixedSets || Sets[n].type == CRANGE ) {
                            MesPrint("&This type of usage of sets is not allowed");
                            error = 1;
                        }
                        *fill++ = *t++; n = 0;
                        while ( n >= 0 ) {
                            if ( *t == LBRACE ) n++;
                            else if ( *t == RBRACE ) n--;
                            *fill++ = *t++;
                        }
                    }
                }
                else if ( *t == TEXPRESSION ) {
                    *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
                    if ( *t == TFUNOPEN ) {
                        *fill++ = *t++; n = 0;
                        while ( n >= 0 ) {
                            if ( *t == TFUNOPEN ) n++;
                            else if ( *t == TFUNCLOSE ) n--;
                            *fill++ = *t++;
                        }
                    }
                    if ( *t == LBRACE ) {
                        *fill++ = *t++; n = 0;
                        while ( n >= 0 ) {
                            if ( *t == LBRACE ) n++;
                            else if ( *t == RBRACE ) n--;
                            *fill++ = *t++;
                        }
                    }
                }
                else if ( *t == TVECTOR ) {
                    *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
                    if ( *t == TFUNOPEN ) {
                        *fill++ = *t++; n = 0;
                        while ( n >= 0 ) {
                            if ( *t == TFUNOPEN ) n++;
                            else if ( *t == TFUNCLOSE ) n--;
                            *fill++ = *t++;
                        }
                    }
                    else if ( *t == TDOT ) {
                        *fill++ = *t++;
                        if ( *t == TVECTOR || *t == TDUBIOUS ) {
                            *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
                        }
                        else if ( *t == TSET ) {
                            *fill++ = *t++; num = 0;
                            while ( *t >= 0 ) { num = 128*num + *t; *fill++ = *t++; }
                            if ( Sets[num].type != CVECTOR ) {
                                MesPrint("&Illegal set type in dotproduct");
                                error = 1;
                            }
                            if ( *t == LBRACE ) {
                                *fill++ = *t++; n = 0;
                                while ( n >= 0 ) {
                                    if ( *t == LBRACE ) n++;
                                    else if ( *t == RBRACE ) n--;
                                    *fill++ = *t++;
                                }
                            }
                        }
                        else if ( *t == TSETNUM ) {
                            *fill++ = *t++;
                            while ( *t >= 0 ) { *fill++ = *t++; }
                            *fill++ = *t++;
                            while ( *t >= 0 ) { *fill++ = *t++; }
                        }
                    }
                    else {
                        MesPrint("&Illegal second element in dotproduct");
                        error = 1;
                    }
                }
                else {
                    *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
```

```
                        if ( *t == TWILDCARD ) *fill++ = *t++;
                    }
                    *fill++ = RPARENTHESIS; *fill++ = TFUNCLOSE;
                    if ( *t == TPOWER ) goto doublepower;
                    while ( fill > ff ) *--t = *--fill;
                    s = t;
                }
                else if ( denom < 0 ) {
                    fill = ff; ff[-1] = TMULTIPLY;
                    *fill++ = TFUNCTION; *fill++ = (SBYTE)(AM.denomnum);
                    *fill++ = TFUNOPEN; *fill++ = LPARENTHESIS;
                    while ( ss < t ) *fill++ = *ss++;
                    *fill++ = RPARENTHESIS; *fill++ = TFUNCLOSE;
                    while ( fill > ff ) *--t = *--fill;
                    s = t; denom = 1; sube = 0;
                    break;
                }
                sube = 0;
                break;
            case TEXPRESSION:
                *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
                t = s;
                if ( *t == TFUNOPEN ) {
                    t++; n = 1;
                    for(;;) {
                        if ( *t == TFUNOPEN ) n++;
                        else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
                        t++;
                    }
                    t++;
                }
                if ( *t == LBRACE ) {
                    t++; n = 1;
                    for(;;) {
                        if ( *t == LBRACE ) n++;
                        else if ( *t == RBRACE ) { if ( --n <= 0 ) break; }
                        t++;
                    }
                    t++;
                }
                if ( t > s || denom < 0 || ( ( *t == TPOWER || *t == TPOWER1 )
                && t[1] == TMINUS ) ) goto dofunpower;
                else goto dosymbol;
            case TDOLLAR:
                *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
                goto dosymbol;
            case LPARENTHESIS:
                *fill++ = *s++; n = 1; t = s;
                for(;;) {
                    if ( *t == LPARENTHESIS ) n++;
                    else if ( *t == RPARENTHESIS ) { if ( --n <= 0 ) break; }
                    t++;
                }
                t++; sube = 1;
                goto dofunpower;
            case TSET:
                *fill++ = *s++; n = *s++; *fill++ = (SBYTE)n;
                while ( *s >= 0 ) { *fill++ = *s; n = 128*n + *s++; }
                n = Sets[n].type;
                switch ( n ) {
                    case CSYMBOL: goto dosymbol;
                    case CINDEX: break;
                    case CVECTOR: goto dovector;
                    case CFUNCTION: goto dofunction;
                    case CNUMBER: goto dosymbol;
                    default: error = 1; break;
                }
                break;
            case TDOT:
dodot:          *fill++ = *s++;
                if ( *s == TVECTOR ) {
                    *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
                }
                else if ( *s == TSET ) {
                    *fill++ = *s++; n = *s++; *fill++ = (SBYTE)n;
                    while ( *s >= 0 ) { *fill++ = *s; n = 128*n + *s++; }
                    if ( *s == LBRACE ) {
                        if ( n < AM.NumFixedSets || Sets[n].type == CRANGE ) {
                            MesPrint("&This type of usage of sets is not allowed");
                            error = 1;
                        }
                        *fill++ = *s++; n = 1;
                        for(;;) {
                            if ( *s == LBRACE ) n++;
                            else if ( *s == RBRACE ) { if ( --n <= 0 ) break; }
                            *fill++ = *s++;
                        }
```

```
                        *fill++ = *s++;
                    }
                    else {
                        MesPrint("&Set without argument in dotproduct");
                        error = 1;
                    }
                }
                else if ( *s == TSETNUM ) {
                    *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
                    if ( *s != TVECTOR ) goto nodot;
                    *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
                }
                else {
nodot:              MesPrint("&Illegal second element in dotproduct");
                    error = 1;
                    *fill++ = *s++;
                    while ( *s >= 0 ) *fill++ = *s++;
                }
                dotp = 1;
                goto dosymbol;
            default:
                *fill++ = *s++;
                while ( *s >= 0 ) *fill++ = *s++;
                break;
        }
    }
    *fill = TENDOFIT;
    return(error);
doublepower:;
    MesPrint("&Dubious notation with power of power");
    return(-1);
}

/*
        #] simp3btoken :
        #[ simp4token :

        Deal with the set[n] objects in the RHS.
*/

int simp4token(SBYTE *s)
{
    int error = 0, n, nsym, settype;
    WORD i, *w, *wstop, level;
    SBYTE *const s0 = s;
    SBYTE *fill = s, *s1, *s2, *s3, type, s1buf[10];
    SBYTE *tbuf = s, *t, *t1;

    while ( *s != TENDOFIT ) {
        if ( *s != TSET ) {
            if ( *s == TEMPTY ) s++;
            else *fill++ = *s++;
            continue;
        }
        if ( fill >= (s0+1) && fill[-1] == TWILDCARD ) { *fill++ = *s++; continue; }
        if ( fill >= (s0+2) && fill[-1] == TNOT && fill[-2] == TWILDCARD ) { *fill++ = *s++; continue;
    }
        s1 = s++; n = 0; while ( *s >= 0 ) { n = 128*n + *s++; }
        i = Sets[n].type;
        if ( *s != LBRACE ) { while ( s1 < s ) *fill++ = *s1++; continue; }
        if ( n < AM.NumFixedSets || i == CRANGE ) {
            MesPrint("&It is not allowed to refer to individual elements of built in or ranged sets");
            error = 1;
        }
        s++;
        if ( *s != TSYMBOL && *s != TDOLLAR ) {
            MesPrint("&Set index in RHS is not a wildcard symbol or $-variable");
            error = 1;
            while ( s1 < s ) *fill++ = *s1++; continue;
        }
        settype = ( *s == TDOLLAR );
        s++; nsym = 0; s2 = s;
        while ( *s >= 0 ) nsym = 128*nsym + *s++;
        if ( *s != RBRACE ) {
            MesPrint("&Improper set argument in RHS");
            error = 1;
            while ( s1 < s ) *fill++ = *s1++; continue;
        }
        s++;
/*
        Verify that nsym is a wildcard
*/
        if ( !settype ) {
            w = AC.ProtoType; wstop = w + w[1]; w += SUBEXPSIZE;
            while ( w < wstop ) {
                if ( *w == SYMTOSYM && w[2] == nsym ) break;
                w += w[1];
```

```
            }
            if ( w >= wstop ) {
/*
                It could still be a summation parameter!
*/
                t = fill - 1;
                while ( t >= tbuf ) {
                    if ( *t == TFUNCLOSE ) {
                        level = 1; t--;
                        while ( t >= tbuf ) {
                            if ( *t == TFUNCLOSE ) level++;
                            else if ( *t == TFUNOPEN ) {
                                level--;
                                if ( level == 0 ) break;
                            }
                            t--;
                        }
                    }
                    else if ( *t == RBRACE ) {
                        level = 1; t--;
                        while ( t >= tbuf ) {
                            if ( *t == RBRACE ) level++;
                            else if ( *t == LBRACE ) {
                                level--;
                                if ( level == 0 ) break;
                            }
                            t--;
                        }
                    }
                    else if ( *t == RPARENTHESIS ) {
                        level = 1; t--;
                        while ( t >= tbuf ) {
                            if ( *t == RPARENTHESIS ) level++;
                            else if ( *t == LPARENTHESIS ) {
                                level--;
                                if ( level == 0 ) break;
                            }
                            t--;
                        }
                    }
                    else if ( *t == TFUNOPEN ) {
                        t1 = t-1;
                        while ( *t1 > 0 && t1 > tbuf ) t1--;
                        if ( *t1 == TFUNCTION ) {
                            t1++; level = 0;
                            while ( *t1 > 0 ) level = level*128+*t1++;
                            if ( level == (SUMF1-FUNCTION)
                             || level == (SUMF2-FUNCTION) ) {
                                t1 = t + 1;
                                if ( *t1 == LPARENTHESIS ) t1++;
                                if ( *t1 == TSYMBOL ) {
                                  if ( ( t1[1] == COEFFSYMBOL
                                  || t1[1] == NUMERATORSYMBOL
                                  || t1[1] == DENOMINATORSYMBOL )
                                  && t1[2] < 0 ) {}
                                  else {
                                    t1++; level = 0;
                                    while ( *t1 >= 0 && t1 < fill ) level = 128*level + *t1++;
                                    if ( level == nsym && t1 < fill ) {
                                        if ( t[1] == LPARENTHESIS
                                         && *t1 == RPARENTHESIS && t1[1] == TCOMMA ) break;
                                        if ( t[1] != LPARENTHESIS && *t1 == TCOMMA ) break;
                                  }
                                    }
                                }
                            }
                        }
                    }
                    t--;
                }
                if ( t < tbuf ) {
                    fill--;
                    MesPrint("&Set index in RHS is not a wildcard symbol");
                    error = 1;
                    while ( s1 < s ) *fill++ = *s1++; continue;
                }
            }
        }
/*
        Now replace by a set marker: TSETNUM,nsym,TYPE,setnumber
*/
        switch ( i ) {
            case CSYMBOL: type = TSYMBOL; break;
            case CINDEX: type = TINDEX; break;
            case CVECTOR: type = TVECTOR; break;
            case CFUNCTION: type = TFUNCTION; break;
            case CNUMBER: type = TNUMBER1; break;
```

```
            case CDUBIOUS: type = TDUBIOUS; break;
            default:
                MesPrint("&Unknown set type in simp4token");
                error = 1; type = CDUBIOUS; break;
        }
        s3 = s1buf; s1++;
        while ( *s1 >= 0 ) *s3++ = *s1++;
        *s3 = -1; s1 = s1buf;
        if ( settype ) *fill++ = TSETDOL;
        else           *fill++ = TSETNUM;
        while ( *s2 >= 0 ) *fill++ = *s2++;
        *fill++ = type; while ( *s1 >= 0 ) *fill++ = *s1++;
    }
    *fill++ = TENDOFIT;
    return(error);
}

/*
        #] simp4token :
        #[ simp5token :

    Making sure that first argument of sumfunction is not a wildcard already
*/

int simp5token(SBYTE *s, int mode)
{
    int error = 0, n, type;
    WORD *w, *wstop;
    if ( mode == RHSIDE ) {
        while ( *s != TENDOFIT ) {
            if ( *s == TFUNCTION ) {
                s++; n = 0; while ( *s >= 0 ) n = 128*n + *s++;
                if ( n == AM.sumnum || n == AM.sumpnum ) {
                    if ( *s != TFUNOPEN ) continue;
                    s++;
                    if ( *s != TSYMBOL && *s != TINDEX ) continue;
                    type = *s++;
                    n = 0; while ( *s >= 0 ) n = 128*n + *s++;
                    if ( type == TINDEX ) n += AM.OffsetIndex;
                    if ( *s != TCOMMA ) continue;
                    w = AC.ProtoType;
                    wstop = w + w[1];
                    w += SUBEXPSIZE;
                    while ( w < wstop ) {
                        if ( w[2] == n ) {
                            if ( ( type == TSYMBOL && ( w[0] == SYMTOSYM
                            || w[0] == SYMTONUM || w[0] == SYMTOSUB ) ) || (
                            type == TINDEX && ( w[0] == INDTOIND
                            || w[0] == INDTOSUB ) ) ) {
                                error = 1;
                                MesPrint("&Parameter of sum function is already a wildcard");
                            }
                        }
                        w += w[1];
                    }
                }
            }
            else s++;
        }
    }
    return(error);
}

/*
        #] simp5token :
        #[ simp6token :

    Making sure that factorized expressions are used properly
*/

int simp6token(SBYTE *tokens, int mode)
{
/*  EXPRESSIONS e = Expressions; */
    int error = 0, n;
    int level = 0, haveone = 0;
    SBYTE *s = tokens, *ss;
    LONG numterms;
    WORD funnum = 0;
    GETIDENTITY
    if ( mode == RHSIDE ) {
        while ( *s == TPLUS || *s == TMINUS ) s++;
        numterms = 1;
        while ( *s != TENDOFIT ) {
            if ( *s == LPARENTHESIS ) level++;
            else if ( *s == RPARENTHESIS ) level--;
            else if ( *s == TFUNOPEN ) level++;
            else if ( *s == TFUNCLOSE ) level--;
```

```
            else if ( ( *s == TPLUS || *s == TMINUS ) && level == 0 ) {
/*
                Special exception: x^-1 etc.
*/
                if ( s[-1] != TPOWER && s[-1] != TPLUS && s[-1] != TMINUS ) {
                    numterms++;
                }
            }
            else if ( *s == TEXPRESSION ) {
                ss = s;
                s++; n = 0; while ( *s >= 0 ) n = 128*n + *s++;

                if ( Expressions[n].status == STOREDEXPRESSION ) {
                    POSITION position;
/*
#ifdef WITHPTHREADS
                    RENUMBER renumber;
#endif
*/
                    RENUMBER renumber;

                    WORD TMproto[SUBEXPSIZE];
                    TMproto[0] = EXPRESSION;
                    TMproto[1] = SUBEXPSIZE;
                    TMproto[2] = n;
                    TMproto[3] = 1;
                    { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
                    AT.TMaddr = TMproto;
                    PUTZERO(position);
/*
                    if ( (
#ifdef WITHPTHREADS
                        renumber =
#endif
                            GetTable(n,&position,0) ) == 0 )
*/
                    if ( ( renumber = GetTable(n,&position,0) ) == 0 )
                    {
                        error = 1;
                        MesPrint("&Problems getting information about stored expression %s(4)"
                        ,EXPRNAME(n));
                    }
/*
#ifdef WITHPTHREADS
*/
                    if ( renumber->symb.lo != AN.dummyrenumlist )
                        M_free(renumber->symb.lo,"VarSpace");
                    M_free(renumber,"Renumber");
/*
#endif
*/
                }

                if ( ( ( AS.Oldvflags[n] & ISFACTORIZED ) != 0 ) && *s != LBRACE ) {
                    if ( level == 0 ) {
                        haveone = 1;
                    }
                    else if ( error == 0 ) {
                        if ( ss[-1] != TFUNOPEN || funnum != NUMFACTORS-FUNCTION ) {
                            MesPrint("&Illegal use of factorized expression(s) in RHS");
                            error = 1;
                        }
                    }
                }
                continue;
            }
            else if ( *s == TFUNCTION ) {
                s++; funnum = 0; while ( *s >= 0 ) funnum = 128*funnum + *s++;
                continue;
            }
            s++;
        }
        if ( haveone ) {
            if ( numterms > 1 ) {
                MesPrint("&Factorized expression in RHS in an expression of more than one term.");
                error = 1;
            }
            else if ( AC.ToBeInFactors == 0 ) {
                MesPrint("&Attempt to put a factorized expression inside an unfactorized
    expression.");
                error = 1;
            }
        }
    }
    return(error);
}
```

```
/*
        #] simp6token :
    #] Compiler :
*/
```

4.118 tools.c File Reference

```
#include "form3.h"
#include <sys/timeb.h>
```

Include dependency graph for tools.c:

Macros

- #define FILLVALUE 0
- #define **COPYFILEBUFSIZE** 40960L
- #define TERMMEMSTARTNUM 16
- #define TERMEXTRAWORDS 10
- #define NUMBERMEMSTARTNUM 16
- #define NUMBEREXTRAWORDS 10L
- #define DODOUBLE(x)
- #define DOEXPAND(x)

Functions

- UBYTE * LoadInputFile (UBYTE *filename, int type)
- UBYTE ReadFromStream (STREAM *stream)
- UBYTE GetFromStream (STREAM *stream)
- UBYTE LookInStream (STREAM *stream)
- STREAM * OpenStream (UBYTE *name, int type, int prevarmode, int raiselow)
- int LocateFile (UBYTE **name, int type)
- STREAM * CloseStream (STREAM *stream)
- STREAM * CreateStream (UBYTE *where)
- LONG GetStreamPosition (STREAM *stream)
- VOID PositionStream (STREAM *stream, LONG position)
- int ReverseStatements (STREAM *stream)
- VOID StartFiles ()
- int OpenFile (char *name)
- int OpenAddFile (char *name)
- int ReOpenFile (char *name)
- int CreateFile (char *name)

- int CreateLogFile (char ∗name)
- VOID CloseFile (int handle)
- int CopyFile (char ∗source, char ∗dest)
- int CreateHandle ()
- LONG ReadFile (int handle, UBYTE ∗buffer, LONG size)
- LONG ReadPosFile (PHEAD FILEHANDLE ∗fi, UBYTE ∗buffer, LONG size, POSITION ∗pos)
- LONG WriteFileToFile (int handle, UBYTE ∗buffer, LONG size)
- VOID SeekFile (int handle, POSITION ∗offset, int origin)
- LONG TellFile (int handle)
- VOID TELLFILE (int handle, POSITION ∗position)
- void FlushFile (int handle)
- int GetPosFile (int handle, fpos_t ∗pospointer)
- int SetPosFile (int handle, fpos_t ∗pospointer)
- VOID SynchFile (int handle)
- VOID TruncateFile (int handle)
- int GetChannel (char ∗name)
- int GetAppendChannel (char ∗name)
- int CloseChannel (char ∗name)
- void UpdateMaxSize ()
- int StrCmp (UBYTE ∗s1, UBYTE ∗s2)
- int StrICmp (UBYTE ∗s1, UBYTE ∗s2)
- int StrHICmp (UBYTE ∗s1, UBYTE ∗s2)
- int StrICont (UBYTE ∗s1, UBYTE ∗s2)
- int ConWord (UBYTE ∗s1, UBYTE ∗s2)
- int StrLen (UBYTE ∗s)
- VOID NumToStr (UBYTE ∗s, LONG x)
- VOID WriteString (int type, UBYTE ∗str, int num)
- VOID WriteUnfinString (int type, UBYTE ∗str, int num)
- UBYTE ∗ AddToString (UBYTE ∗outstring, UBYTE ∗extrastring, int par)
- UBYTE ∗ strDup1 (UBYTE ∗instring, char ∗ifwrong)
- UBYTE ∗ EndOfToken (UBYTE ∗s)
- UBYTE ∗ ToToken (UBYTE ∗s)
- UBYTE ∗ SkipField (UBYTE ∗s, int level)
- WORD ReadSnum (UBYTE ∗∗p)
- UBYTE ∗ NumCopy (WORD y, UBYTE ∗to)
- char ∗ LongCopy (LONG y, char ∗to)
- char ∗ LongLongCopy (off_t ∗y, char ∗to)
- UBYTE ∗ MakeDate ()
- int set_in (UBYTE ch, set_of_char set)
- one_byte set_set (UBYTE ch, set_of_char set)
- one_byte set_del (UBYTE ch, set_of_char set)
- one_byte set_sub (set_of_char set, set_of_char set1, set_of_char set2)
- VOID iniTools (VOID)
- VOID ∗ Malloc (LONG size)
- VOID ∗ Malloc1 (LONG size, const char ∗messageifwrong)
- void M_free (VOID ∗x, const char ∗where)
- void M_check1 ()
- void M_print ()
- VOID TermMallocAddMemory (PHEAD0)
- VOID NumberMallocAddMemory (PHEAD0)
- VOID CacheNumberMallocAddMemory (PHEAD0)
- VOID ∗ FromList (LIST ∗L)
- VOID ∗ From0List (LIST ∗L)
- VOID ∗ FromVarList (LIST ∗L)
- int DoubleList (VOID ∗∗∗lijst, int ∗oldsize, int objectsize, char ∗nameoftype)

- int DoubleLList (VOID ∗∗∗lijst, LONG ∗oldsize, int objectsize, char ∗nameoftype)
- void DoubleBuffer (void ∗∗start, void ∗∗stop, int size, char ∗text)
- void ExpandBuffer (void ∗∗buffer, LONG ∗oldsize, int type)
- LONG iexp (LONG x, int p)
- void ToGeneral (WORD ∗r, WORD ∗m, WORD par)
- int ToFast (WORD ∗r, WORD ∗m)
- WORD ToPolyFunGeneral (PHEAD WORD ∗term)
- int IsLikeVector (WORD ∗arg)
- int AreArgsEqual (WORD ∗arg1, WORD ∗arg2)
- int CompareArgs (WORD ∗arg1, WORD ∗arg2)
- int CompArg (WORD ∗s1, WORD ∗s2)
- LONG TimeWallClock (WORD par)
- LONG TimeChildren (WORD par)
- LONG TimeCPU (WORD par)
- int Crash ()
- int TestTerm (WORD ∗term)

Variables

- FILES ∗∗ filelist
- int numinfilelist = 0
- int filelistsize = 0
- WRITEFILE WriteFile = &WriteFileToFile

4.118.1 Detailed Description

Low level routines for many types of task. There are routines for manipulating the input system (streams and files) routines for string manipulation, the memory allocation interface, and the clock. The last is the most sensitive to ports. In the past nearly every port to another OS or computer gave trouble. Nowadays it is slightly better but the poor POSIX compliance of LINUX again gave problems for the multithreaded version.

Definition in file tools.c.

4.118.2 Macro Definition Documentation

4.118.2.1 FILLVALUE

```
#define FILLVALUE 0
```

Definition at line 49 of file tools.c.

4.118.2.2 TERMMEMSTARTNUM

```
#define TERMMEMSTARTNUM 16
```

Provides memory for one term (or one small polynomial) This means that the memory is limited to a buffer of size AM.MaxTer plus a few extra words. In parallel versions, each worker has its own memory pool.

The way we use the memory is by: term = TermMalloc(BHEAD0); and later we free it by TermFree(BHEAD term);

Layout: We have a list of available pointers to buffers: AT.TermMemHeap Its size is AT.TermMemMax We take from the top (indicated by AT.TermMemTop). When we run out of buffers we assign new ones (doubling the amount) and we have to extend the AT.TermMemHeap array. Important: There is no checking that the returned memory is legal, ie is memory that was handed out earlier.

Definition at line 2460 of file tools.c.

4.118.2.3 TERMEXTRAWORDS

```
#define TERMEXTRAWORDS 10
```

Definition at line 2461 of file tools.c.

4.118.2.4 NUMBERMEMSTARTNUM

```
#define NUMBERMEMSTARTNUM 16
```

Provides memory for one Long number This means that the memory is limited to a buffer of size AM.MaxTal In parallel versions, each worker has its own memory pool.

The way we use the memory is by: num = NumberMalloc(BHEAD0); Number = AT.NumberMemHeap[num]; and later we free it by NumberFree(BHEAD num);

Layout: We have a list of available pointers to buffers: AT.NumberMemHeap Its size is AT.NumberMemMax We take from the top (indicated by AT.NumberMemTop). When we run out of buffers we assign new ones (doubling the amount) and we have to extend the AT.NumberMemHeap array. Important: There is no checking on the returned memory!!!!

Definition at line 2560 of file tools.c.

4.118.2.5 NUMBEREXTRAWORDS

```
#define NUMBEREXTRAWORDS 10L
```

Definition at line 2561 of file tools.c.

4.118.2.6 DODOUBLE

```
#define DODOUBLE(
               x )
```

Value:

```
{ x *s, *t, *u; if ( *start ) { \
oldsize = *(x **)stop - *(x **)start; newsize = 2*oldsize; \
t = u = (x *)Malloc1(newsize*sizeof(x),text); s = *(x **)start; \
for ( i = 0; i < oldsize; i++ ) *t++ = *s++; M_free(*start,"double"); } \
else { newsize = 100; u = (x *)Malloc1(newsize*sizeof(x),text); } \
*start = (void *)u; *stop = (void *)(u+newsize); }
```

Definition at line 2883 of file tools.c.

4.118.2.7 DOEXPAND

```
#define DOEXPAND(
               x )
```

Value:

```
{ x *newbuffer, *t, *m;                                \
t = newbuffer = (x *)Malloc1((newsize+2)*type,"ExpandBuffer");      \
if ( *buffer ) { m = (x *)*buffer; i = *oldsize;                    \
    while ( --i >= 0 ) *t++ = *m++; M_free(*buffer,"ExpandBuffer"); \
} *buffer = newbuffer; *oldsize = newsize; }
```

Definition at line 2909 of file tools.c.

4.118.3 Function Documentation

4.118.3.1 LoadInputFile()

```
UBYTE * LoadInputFile (
            UBYTE * filename,
            int type )
```

Definition at line 112 of file tools.c.

4.118.3.2 ReadFromStream()

```
UBYTE ReadFromStream (
            STREAM * stream )
```

Definition at line 151 of file tools.c.

4.118.3.3 GetFromStream()

```
UBYTE GetFromStream (
            STREAM * stream )
```

Definition at line 255 of file tools.c.

4.118.3.4 LookInStream()

```
UBYTE LookInStream (
            STREAM * stream )
```

Definition at line 278 of file tools.c.

4.118.3.5 OpenStream()

```
STREAM * OpenStream (
            UBYTE * name,
            int type,
            int prevarmode,
            int raiselow )
```

Definition at line 290 of file tools.c.

4.118.3.6 LocateFile()

```
int LocateFile (
            UBYTE ** name,
            int type )
```

Definition at line 513 of file tools.c.

4.118.3.7 CloseStream()

```
STREAM * CloseStream (
            STREAM * stream )
```

Definition at line 600 of file tools.c.

4.118.3.8 CreateStream()

```
STREAM * CreateStream (
            UBYTE * where )
```

Definition at line 710 of file tools.c.

4.118.3.9 GetStreamPosition()

```
LONG GetStreamPosition (
            STREAM * stream )
```

Definition at line 741 of file tools.c.

4.118.3.10 PositionStream()

```
VOID PositionStream (
            STREAM * stream,
            LONG position )
```

Definition at line 751 of file tools.c.

4.118.3.11 ReverseStatements()

```
int ReverseStatements (
            STREAM * stream )
```

Definition at line 783 of file tools.c.

4.118.3.12 StartFiles()

```
VOID StartFiles ( )
```

Definition at line 884 of file tools.c.

4.118.3.13 OpenFile()

```
int OpenFile (
            char * name )
```

Definition at line 911 of file tools.c.

4.118.3.14 OpenAddFile()

```
int OpenAddFile (
            char * name )
```

Definition at line 930 of file tools.c.

4.118.3.15 ReOpenFile()

```
int ReOpenFile (
            char * name )
```

Definition at line 951 of file tools.c.

4.118.3.16 CreateFile()

```
int CreateFile (
            char * name )
```

Definition at line 971 of file tools.c.

4.118.3.17 CreateLogFile()

```
int CreateLogFile (
            char * name )
```

Definition at line 988 of file tools.c.

4.118.3.18 CloseFile()

```
VOID CloseFile (
            int handle )
```

Definition at line 1006 of file tools.c.

4.118.3.19 CopyFile()

```
int CopyFile (
            char * source,
            char * dest )
```

Copies a file with name ∗source to a file named ∗dest. The involved files must not be open. Returns non-zero if an error occurred. Uses if possible the combined large and small sorting buffers as cache.

Definition at line 1029 of file tools.c.

Referenced by DoRecovery().

4.118.3.20 CreateHandle()

```
int CreateHandle ( )
```

Definition at line 1090 of file tools.c.

4.118.3.21 ReadFile()

```
LONG ReadFile (
            int handle,
            UBYTE * buffer,
            LONG size )
```

Definition at line 1145 of file tools.c.

4.118.3.22 ReadPosFile()

```
LONG ReadPosFile (
            PHEAD FILEHANDLE * fi,
            UBYTE * buffer,
            LONG size,
            POSITION * pos )
```

Definition at line 1195 of file tools.c.

4.118.3.23 WriteFileToFile()

```
LONG WriteFileToFile (
            int handle,
            UBYTE * buffer,
            LONG size )
```

Definition at line 1261 of file tools.c.

4.118.3.24 SeekFile()

```
VOID SeekFile (
            int handle,
            POSITION * offset,
            int origin )
```

Definition at line 1299 of file tools.c.

4.118.3.25 TellFile()

```
LONG TellFile (
            int handle )
```

Definition at line 1324 of file tools.c.

4.118.3.26 TELLFILE()

```
VOID TELLFILE (
            int handle,
            POSITION * position )
```

Definition at line 1334 of file tools.c.

4.118.3.27 FlushFile()

```
void FlushFile (
            int handle )
```

Definition at line 1348 of file tools.c.

4.118.3.28 GetPosFile()

```
int GetPosFile (
            int handle,
            fpos_t * pospointer )
```

Definition at line 1362 of file tools.c.

4.118.3.29 SetPosFile()

```
int SetPosFile (
            int handle,
            fpos_t * pospointer )
```

Definition at line 1376 of file tools.c.

4.118.3.30 SynchFile()

```
VOID SynchFile (
            int handle )
```

Definition at line 1396 of file tools.c.

4.118.3.31 TruncateFile()

```
VOID TruncateFile (
            int handle )
```

Definition at line 1418 of file tools.c.

4.118.3.32 GetChannel()

```
int GetChannel (
            char * name )
```

Definition at line 1437 of file tools.c.

4.118.3.33 GetAppendChannel()

```
int GetAppendChannel (
            char * name )
```

Definition at line 1468 of file tools.c.

4.118.3.34 CloseChannel()

```
int CloseChannel (
            char * name )
```

Definition at line 1498 of file tools.c.

4.118.3.35 UpdateMaxSize()

```
void UpdateMaxSize ( )
```

Definition at line 1532 of file tools.c.

4.118.3.36 StrCmp()

```
int StrCmp (
            UBYTE * s1,
            UBYTE * s2 )
```

Definition at line 1622 of file tools.c.

4.118.3.37 StrICmp()

```
int StrICmp (
            UBYTE * s1,
            UBYTE * s2 )
```

Definition at line 1633 of file tools.c.

4.118.3.38 StrHICmp()

```
int StrHICmp (
            UBYTE * s1,
            UBYTE * s2 )
```

Definition at line 1644 of file tools.c.

4.118.3.39 StrICont()

```
int StrICont (
            UBYTE * s1,
            UBYTE * s2 )
```

Definition at line 1655 of file tools.c.

4.118.3.40 ConWord()

```
int ConWord (
            UBYTE * s1,
            UBYTE * s2 )
```

Definition at line 1667 of file tools.c.

4.118.3.41 StrLen()

```
int StrLen (
            UBYTE * s )
```

Definition at line 1679 of file tools.c.

4.118.3.42 NumToStr()

```
VOID NumToStr (
            UBYTE * s,
            LONG x )
```

Definition at line 1691 of file tools.c.

4.118.3.43 WriteString()

```
VOID WriteString (
            int type,
            UBYTE * str,
            int num )
```

Definition at line 1715 of file tools.c.

4.118.3.44 WriteUnfinString()

```
VOID WriteUnfinString (
            int type,
            UBYTE * str,
            int num )
```

Definition at line 1749 of file tools.c.

4.118.3.45 AddToString()

```
UBYTE * AddToString (
            UBYTE * outstring,
            UBYTE * extrastring,
            int par )
```

Definition at line 1774 of file tools.c.

4.118.3.46 strDup1()

```
UBYTE * strDup1 (
            UBYTE * instring,
            char * ifwrong )
```

Definition at line 1812 of file tools.c.

4.118.3.47 EndOfToken()

```
UBYTE * EndOfToken (
            UBYTE * s )
```

Definition at line 1827 of file tools.c.

4.118.3.48 ToToken()

```
UBYTE * ToToken (
            UBYTE * s )
```

Definition at line 1839 of file tools.c.

4.118.3.49 SkipField()

```
UBYTE * SkipField (
            UBYTE * s,
            int level )
```

Definition at line 1854 of file tools.c.

4.118.3.50 ReadSnum()

```
WORD ReadSnum (
            UBYTE ** p )
```

Definition at line 1881 of file tools.c.

4.118.3.51 NumCopy()

```
UBYTE * NumCopy (
            WORD y,
            UBYTE * to )
```

Definition at line 1905 of file tools.c.

4.118.3.52 LongCopy()

```
char * LongCopy (
            LONG y,
            char * to )
```

Definition at line 1935 of file tools.c.

4.118.3.53 LongLongCopy()

```
char * LongLongCopy (
            off_t * y,
            char * to )
```

Definition at line 1965 of file tools.c.

4.118.3.54 MakeDate()

```
UBYTE * MakeDate ( )
```

Definition at line 2004 of file tools.c.

4.118.3.55 set_in()

```
int set_in (
            UBYTE ch,
            set_of_char set )
```

Definition at line 2026 of file tools.c.

4.118.3.56 set_set()

```
one_byte set_set (
            UBYTE ch,
            set_of_char set )
```

Definition at line 2046 of file tools.c.

4.118.3.57 set_del()

```
one_byte set_del (
            UBYTE ch,
            set_of_char set )
```

Definition at line 2067 of file tools.c.

4.118.3.58 set_sub()

```
one_byte set_sub (
            set_of_char set,
            set_of_char set1,
            set_of_char set2 )
```

Definition at line 2089 of file tools.c.

4.118.3.59 iniTools()

```
VOID iniTools (
            VOID  )
```

Definition at line 2115 of file tools.c.

4.118.3.60 Malloc()

```
VOID * Malloc (
            LONG size )
```

Definition at line 2135 of file tools.c.

4.118.3.61 Malloc1()

```
VOID * Malloc1 (
            LONG size,
            const char * messageifwrong )
```

Definition at line 2217 of file tools.c.

4.118.3.62 M_free()

```
void M_free (
            VOID * x,
            const char * where )
```

Definition at line 2296 of file tools.c.

4.118.3.63 M_check1()

```
void M_check1 ( )
```

Definition at line 2429 of file tools.c.

4.118.3.64 M_print()

```
void M_print ( )
```

Definition at line 2430 of file tools.c.

4.118.3.65 TermMallocAddMemory()

```
VOID TermMallocAddMemory (
            PHEAD0  )
```

Definition at line 2463 of file tools.c.

4.118.3.66 NumberMallocAddMemory()

```
VOID NumberMallocAddMemory (
            PHEAD0  )
```

Definition at line 2567 of file tools.c.

4.118.3.67 CacheNumberMallocAddMemory()

```
VOID CacheNumberMallocAddMemory (
            PHEAD0  )
```

Definition at line 2641 of file tools.c.

4.118.3.68 FromList()

```
VOID * FromList (
            LIST * L )
```

Definition at line 2693 of file tools.c.

4.118.3.69 From0List()

```
VOID * From0List (
            LIST * L )
```

Definition at line 2719 of file tools.c.

4.118.3.70 FromVarList()

```
VOID * FromVarList (
            LIST * L )
```

Definition at line 2748 of file tools.c.

4.118.3.71 DoubleList()

```
int DoubleList (
            VOID *** lijst,
            int * oldsize,
            int objectsize,
            char * nameoftype )
```

Definition at line 2790 of file tools.c.

4.118.3.72 DoubleLList()

```
int DoubleLList (
            VOID *** lijst,
            LONG * oldsize,
            int objectsize,
            char * nameoftype )
```

Definition at line 2842 of file tools.c.

4.118.3.73 DoubleBuffer()

```
void DoubleBuffer (
            void ** start,
            void ** stop,
            int size,
            char * text )
```

Definition at line 2890 of file tools.c.

4.118.3.74 ExpandBuffer()

```
void ExpandBuffer (
            void ** buffer,
            LONG * oldsize,
            int type )
```

Definition at line 2915 of file tools.c.

4.118.3.75 iexp()

```
LONG iexp (
            LONG x,
            int p )
```

Definition at line 2939 of file tools.c.

4.118.3.76 ToGeneral()

```
void ToGeneral (
            WORD * r,
            WORD * m,
            WORD par )
```

Definition at line 2969 of file tools.c.

4.118.3.77 ToFast()

```
int ToFast (
            WORD * r,
            WORD * m )
```

Definition at line 3009 of file tools.c.

4.118.3.78 ToPolyFunGeneral()

```
WORD ToPolyFunGeneral (
            PHEAD WORD * term )
```

Definition at line 3062 of file tools.c.

4.118.3.79 IsLikeVector()

```
int IsLikeVector (
            WORD * arg )
```

Definition at line 3137 of file tools.c.

4.118.3.80 AreArgsEqual()

```
int AreArgsEqual (
            WORD * arg1,
            WORD * arg2 )
```

Definition at line 3165 of file tools.c.

4.118.3.81 CompareArgs()

```
int CompareArgs (
            WORD * arg1,
            WORD * arg2 )
```

Definition at line 3184 of file tools.c.

4.118.3.82 CompArg()

```
int CompArg (
            WORD * s1,
            WORD * s2 )
```

Definition at line 3215 of file tools.c.

4.118.3.83 TimeWallClock()

```
LONG TimeWallClock (
            WORD par )
```

Returns the wall-clock time.

Parameters

par	If zero, the wall-clock time will be reset to 0.

Returns

The wall-clock time in centiseconds.

Definition at line 3377 of file tools.c.

Referenced by DoCheckpoint(), DoRecovery(), and WriteStats().

4.118.3.84 TimeChildren()

```
LONG TimeChildren (
            WORD par )
```

Definition at line 3400 of file tools.c.

4.118.3.85 TimeCPU()

```
LONG TimeCPU (
            WORD par )
```

Returns the CPU time.

Parameters

par	If zero, the CPU time will be reset to 0.

Returns

The CPU time in milliseconds.

Definition at line 3418 of file tools.c.

Referenced by PF_GetSlaveTimes(), PF_Processor(), and WriteStats().

4.118.3.86 Crash()

```
int Crash ( )
```

Definition at line 3701 of file tools.c.

4.118.3.87 TestTerm()

```
int TestTerm (
            WORD * term )
```

Tests the consistency of the term. Returns 0 when the term is OK. Any nonzero value is trouble. In the current version the testing isn't 100% complete. For instance, we don't check the validity of the symbols nor do we check the range of their powers. Etc. This should be extended when the need is there.

Parameters

term	the term to be tested

Definition at line 3729 of file tools.c.

References TestTerm().

Referenced by TestTerm().

Here is the call graph for this function:

4.118.4 Variable Documentation

4.118.4.1 filelist

```
FILES** filelist
```

Definition at line 65 of file tools.c.

4.118.4.2 numinfilelist

```
int numinfilelist = 0
```

Definition at line 66 of file tools.c.

4.118.4.3 filelistsize

```
int filelistsize = 0
```

Definition at line 67 of file tools.c.

4.118.4.4 WriteFile

```
WRITEFILE WriteFile = &WriteFileToFile
```

Definition at line 1285 of file tools.c.

4.119 tools.c

Go to the documentation of this file.

```
00001
00011 /* #[ License : */
00012 /*
00013  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00014  *   When using this file you are requested to refer to the publication
00015  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016  *   This is considered a matter of courtesy as the development was paid
00017  *   for by FOM the Dutch physics granting agency and we would like to
00018  *   be able to track its scientific use to convince FOM of its value
00019  *   for the community.
00020  *
00021  *   This file is part of FORM.
00022  *
00023  *   FORM is free software: you can redistribute it and/or modify it under the
00024  *   terms of the GNU General Public License as published by the Free Software
00025  *   Foundation, either version 3 of the License, or (at your option) any later
00026  *   version.
00027  *
00028  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00031  *   details.
00032  *
00033  *   You should have received a copy of the GNU General Public License along
00034  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00035  */
00036 /* #] License : */
00037 /*
00038     #[ Includes :
00039     Note: TERMMALLOCDEBUG tests part of the TermMalloc and NumberMalloc
00040           system. To work properly it needs MEMORYMACROS in declare.h
```

```
          not to be defined to make sure that all calls will be diverted
          to the routines here.
#define TERMMALLOCDEBUG
#define FILLVALUE 126
#define MALLOCDEBUGOUTPUT
#define MALLOCDEBUG 1
*/
#ifndef FILLVALUE
    #define FILLVALUE 0
#endif

/*
    The enhanced malloc debugger, see comments in the beginning of the
    file mallocprotect.h
    MALLOCPROTECT == -1  -- protect left side, used block is left-aligned.
    MALLOCPROTECT == 0  -- protect both sides, used block is left-aligned;
    MALLOCPROTECT == 1  -- protect both sides, used block is right-aligned;
    ATTENTION! The macro MALLOCPROTECT must be defined
    BEFORE #include mallocprotect.h
#define MALLOCPROTECT 1
*/

#include "form3.h"

FILES **filelist;
int numinfilelist = 0;
int filelistsize = 0;
#ifdef MALLOCDEBUG
#define BANNER (4*sizeof(LONG))
void *malloclist[60000];
LONG mallocsizes[60000];
char *mallocstrings[60000];
int nummalloclist = 0;
#endif

#ifdef GPP
extern "C" getdtablesize();
#endif

#ifdef WITHSTATS
LONG numwrites = 0;
LONG numreads = 0;
LONG numseeks = 0;
LONG nummallocs = 0;
LONG numfrees = 0;
#endif

#ifdef MALLOCPROTECT
#ifdef TRAPSIGNALS
#error "MALLOCPROTECT":  undefine "TRAPSIGNALS" in unix.h first!
#endif
#include "mallocprotect.h"

#ifdef M_alloc
#undef M_alloc
#endif

#define M_alloc mprotectMalloc

#endif

#ifdef TERMMALLOCDEBUG
WORD **DebugHeap1, **DebugHeap2;
#endif

/*
    #] Includes :
    #[ Streams :
        #[ LoadInputFile :
*/

UBYTE *LoadInputFile(UBYTE *filename, int type)
{
    int handle;
    LONG filesize;
    UBYTE *buffer, *name = filename;
    POSITION scrpos;
    handle = LocateFile(&name,type);
    if ( handle < 0 ) return(0);
    PUTZERO(scrpos);
    SeekFile(handle,&scrpos,SEEK_END);
    TELLFILE(handle,&scrpos);
    filesize = BASEPOSITION(scrpos);
    PUTZERO(scrpos);
    SeekFile(handle,&scrpos,SEEK_SET);
    buffer = (UBYTE *)Malloc1(filesize+2,"LoadInputFile");
    if ( ReadFile(handle,buffer,filesize) != filesize ) {
```

```
        Error1("Read error for file ",name);
        M_free(buffer,"LoadInputFile");
        if ( name != filename ) M_free(name,"FromLoadInputFile");
        CloseFile(handle);
        return(0);
    }
    CloseFile(handle);
    if ( type == PROCEDUREFILE || type == SETUPFILE ) {
        buffer[filesize] = '\n';
        buffer[filesize+1] = 0;
    }
    else {
        buffer[filesize] = 0;
    }
    if ( name != filename ) M_free(name,"FromLoadInputFile");
    return(buffer);
}

/*
        #] LoadInputFile :
        #[ ReadFromStream :
*/

UBYTE ReadFromStream(STREAM *stream)
{
    UBYTE c;
    POSITION scrpos;
#ifdef WITHPIPE
    if ( stream->type == PIPESTREAM ) {
#ifndef WITHMPI
        FILE *f;
        int cc;
        RWLOCKR(AM.handlelock);
        f = (FILE *)(filelist[stream->handle]);
        UNRWLOCK(AM.handlelock);
        cc = getc(f);
        if ( cc == EOF ) return(ENDOFSTREAM);
        c = (UBYTE)cc;
#else
        if ( stream->pointer >= stream->top ) {
            /* The master reads the pipe and broadcasts it to the slaves. */
            LONG len;
            if ( PF.me == MASTER ) {
                FILE *f;
                UBYTE *p, *end;
                RWLOCKR(AM.handlelock);
                f = (FILE *)filelist[stream->handle];
                UNRWLOCK(AM.handlelock);
                p = stream->buffer;
                end = stream->buffer + stream->buffersize;
                while ( p < end ) {
                    int cc = getc(f);
                    if ( cc == EOF ) {
                        break;
                    }
                    *p++ = (UBYTE)cc;
                }
                len = p - stream->buffer;
                PF_BroadcastNumber(len);
            }
            else {
                len = PF_BroadcastNumber(0);
            }
            if ( len > 0 ) {
                PF_Bcast(stream->buffer, len);
            }
            stream->pointer = stream->buffer;
            stream->inbuffer = len;
            stream->top = stream->buffer + stream->inbuffer;
            if ( stream->pointer == stream->top ) return ENDOFSTREAM;
        }
        c = (UBYTE)*stream->pointer++;
#endif
        if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
        if ( c == LINEFEED ) stream->eqnum = 1;
        return(c);
    }
#endif
/*[14apr2004 mt]:*/
#ifdef WITHEXTERNALCHANNEL
    if ( stream->type == EXTERNALCHANNELSTREAM ) {
        int cc;
        cc = getcFromExtChannel();
        /*[18may20006 mt]:*/
        /*if ( cc == EOF ) return(ENDOFSTREAM);*/
        if ( cc < 0 ){
            if( cc == EOF )
```

```
            return(ENDOFSTREAM);
            else{
                Error0("No current external channel");
                Terminate(-1);
            }
        }/*if ( cc < 0 )*/
        /*:[18may20006 mt]*/
        c = (UBYTE)cc;
        if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
        if ( c == LINEFEED ) stream->eqnum = 1;
        return(c);
    }
#endif /*ifdef WITHEXTERNALCHANNEL*/
/*:[14apr2004 mt]*/
    if ( stream->pointer >= stream->top ) {
        if ( stream->type != FILESTREAM ) return(ENDOFSTREAM);
        if ( stream->fileposition != stream->bufferposition+stream->inbuffer ) {
            stream->fileposition = stream->bufferposition+stream->inbuffer;
            SETBASEPOSITION(scrpos,stream->fileposition);
            SeekFile(stream->handle,&scrpos,SEEK_SET);
        }
        stream->bufferposition = stream->fileposition;
        stream->inbuffer = ReadFile(stream->handle,
                stream->buffer,stream->buffersize);
        if ( stream->inbuffer <= 0 ) return(ENDOFSTREAM);
        stream->top = stream->buffer + stream->inbuffer;
        stream->pointer = stream->buffer;
        stream->fileposition = stream->bufferposition + stream->inbuffer;
    }
    if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
    c = *(stream->pointer)++;
    if ( c == LINEFEED ) stream->eqnum = 1;
    return(c);
}

/*
        #] ReadFromStream :
        #[ GetFromStream :
*/

UBYTE GetFromStream(STREAM *stream)
{
    UBYTE c1, c2;
    if ( stream->isnextchar > 0 ) {
        return(stream->nextchar[--stream->isnextchar]);
    }
    c1 = ReadFromStream(stream);
    if ( c1 == LINEFEED || c1 == CARRIAGERETURN ) {
        c2 = ReadFromStream(stream);
        if ( c2 == c1 || ( c2 != LINEFEED && c2 != CARRIAGERETURN ) ) {
            stream->isnextchar = 1;
            stream->nextchar[0] = c2;
        }
        return(LINEFEED);
    }
    else return(c1);
}

/*
        #] GetFromStream :
        #[ LookInStream :
*/

UBYTE LookInStream(STREAM *stream)
{
    UBYTE c = GetFromStream(stream);
    UngetFromStream(stream,c);
    return(c);
}

/*
        #] LookInStream :
        #[ OpenStream :
*/

STREAM *OpenStream(UBYTE *name, int type, int prevarmode, int raiselow)
{
    STREAM *stream;
    UBYTE *rhsofvariable, *s, *newname, c;
    POSITION scrpos;
    int handle, num;
    LONG filesize;
    switch ( type ) {
        case REVERSEFILESTREAM:
        case FILESTREAM:
/*
            Notice that FILESTREAM is only used for text files:
```

```
            The #include files and the main input file (.frm)
            Hence we do not worry about files longer than 2 Gbytes.
*/
            newname = name;
            handle = LocateFile(&newname,-1);
            if ( handle < 0 ) return(0);
            PUTZERO(scrpos);
            SeekFile(handle,&scrpos,SEEK_END);
            TELLFILE(handle,&scrpos);
            filesize = BASEPOSITION(scrpos);
            PUTZERO(scrpos);
            SeekFile(handle,&scrpos,SEEK_SET);
            if ( filesize > AM.MaxStreamSize && type == FILESTREAM )
                    filesize = AM.MaxStreamSize;
            stream = CreateStream((UBYTE *)"filestream");
/*
            The extra +1 in the Malloc1 is potentially needed in ReverseStatements!
*/
            stream->buffer = (UBYTE *)Malloc1(filesize+1,"name of input stream");
            stream->inbuffer = ReadFile(handle,stream->buffer,filesize);
            if ( type == REVERSEFILESTREAM ) {
                if ( ReverseStatements(stream) ) {
                    M_free(stream->buffer,"name of input stream");
                    return(0);
                }
            }
            stream->top = stream->buffer + stream->inbuffer;
            stream->pointer = stream->buffer;
            stream->handle = handle;
            stream->buffersize = filesize;
            stream->fileposition = stream->inbuffer;
            if ( newname != name ) stream->name = newname;
            else if ( name ) stream->name = strDup1(name,"name of input stream");
            else
                stream->name = 0;
            stream->prevline = stream->linenumber = 1;
            stream->eqnum = 0;
            break;
        case PREVARSTREAM:
            if ( ( rhsofvariable = GetPreVar(name,WITHERROR) ) == 0 ) return(0);
            stream = CreateStream((UBYTE *)"var-stream");
            stream->buffer = stream->pointer = s = rhsofvariable;
            while ( *s ) s++;
            stream->top = s;
            stream->inbuffer = s - stream->buffer;
            stream->name = AC.CurrentStream->name;
            stream->linenumber = AC.CurrentStream->linenumber;
            stream->prevline = AC.CurrentStream->prevline;
            stream->eqnum = AC.CurrentStream->eqnum;
            stream->pname = strDup1(name,"stream->pname");
            stream->olddelay = AP.AllowDelay;
            s = stream->pname; while ( *s ) s++;
            while ( s[-1] == '+' || s[-1] == '-' ) s--;
            *s = 0;
            UnsetAllowDelay();
            break;
        case DOLLARSTREAM:
            if ( ( num = GetDollar(name) ) < 0 ) {
                WORD numfac = 0;
/*
                Here we have to test first whether we have $x[1], $x[0]
                or just an undefined $x.
*/
                s = name; while ( *s && *s != '[' ) s++;
                if ( *s == 0 ) return(0);
                c = *s; *s = 0;
                if ( ( num = GetDollar(name) ) < 0 ) return(0);
                *s = c;
                s++;
                if ( *s == 0 || FG.cTable[*s] != 1 || *s == ']' ) {
                    MesPrint("@Illegal factor number for dollar variable");
                    return(0);
                }
                while ( *s && FG.cTable[*s] == 1 ) {
                    numfac = 10*numfac+*s++-'0';
                }
                if ( *s != ']' || s[1] != 0 ) {
                    MesPrint("@Illegal factor number for $ variable");
                    return(0);
                }
                stream = CreateStream((UBYTE *)"dollar-stream");
                stream->buffer = stream->pointer = s = WriteDollarFactorToBuffer(num,numfac,1);
            }
            else {
                stream = CreateStream((UBYTE *)"dollar-stream");
                stream->buffer = stream->pointer = s = WriteDollarToBuffer(num,1);
            }
```

```
            while ( *s ) s++;
            stream->top = s;
            stream->inbuffer = s - stream->buffer;
            stream->name = AC.CurrentStream->name;
            stream->linenumber = AC.CurrentStream->linenumber;
            stream->prevline= AC.CurrentStream->prevline;
            stream->eqnum = AC.CurrentStream->eqnum;
            stream->pname = strDup1(name,"stream->pname");
            s = stream->pname; while ( *s ) s++;
            while ( s[-1] == '+' || s[-1] == '-' ) s--;
            *s = 0;
            /* We 'stole' the buffer. Later we can free it. */
            AO.DollarOutSizeBuffer = 0;
            AO.DollarOutBuffer = 0;
            AO.DollarInOutBuffer = 0;
            break;
        case PREREADSTREAM:
        case PREREADSTREAM2:
        case PREREADSTREAM3:
        case PRECALCSTREAM:
            stream = CreateStream((UBYTE *)"calculator");
            stream->buffer = stream->pointer = s = name;
            while ( *s ) s++;
            stream->top = s;
            stream->inbuffer = s - stream->buffer;
            stream->name = AC.CurrentStream->name;
            stream->linenumber = AC.CurrentStream->linenumber;
            stream->prevline = AC.CurrentStream->prevline;
            stream->eqnum = 0;
            break;
#ifdef WITHPIPE
        case PIPESTREAM:
            stream = CreateStream((UBYTE *)"pipe");
#ifndef WITHMPI
            {
                FILE *f;
                if ( ( f = popen((char *)name,"r") ) == 0 ) {
                    Error0("@Cannot create pipe");
                }
                stream->handle = CreateHandle();
                RWLOCKW(AM.handlelock);
                filelist[stream->handle] = (FILES *)f;
                UNRWLOCK(AM.handlelock);
            }
            stream->buffer = stream->top = 0;
            stream->inbuffer = 0;
#else
            {
                /* Only the master opens the pipe. */
                FILE *f;
                if ( PF.me == MASTER ) {
                    f = popen((char *)name, "r");
                    PF_BroadcastNumber(f == 0);
                    if ( f == 0 ) Error0("@Cannot create pipe");
                }
                else {
                    if ( PF_BroadcastNumber(0) ) Error0("@Cannot create pipe");
                    f = (FILE *)123;  /* dummy */
                }
                stream->handle = CreateHandle();
                RWLOCKW(AM.handlelock);
                filelist[stream->handle] = (FILES *)f;
                UNRWLOCK(AM.handlelock);
            }
            /* stream->buffer as a send/receive buffer. */
            stream->buffersize = AM.MaxStreamSize;
            stream->buffer = (UBYTE *)Malloc1(stream->buffersize, "pipe buffer");
            stream->inbuffer = 0;
            stream->top = stream->buffer;
            stream->pointer = stream->buffer;
#endif
            stream->name = strDup1((UBYTE *)"pipe","pipe");
            stream->prevline = stream->linenumber = 1;
            stream->eqnum = 0;
            break;
#endif
/*[14apr2004 mt]:*/
#ifdef WITHEXTERNALCHANNEL
        case EXTERNALCHANNELSTREAM:
            {/*Block*/
                int n, *tmpn;
                if( (n=getCurrentExternalChannel()) == 0 )
                    Error0("@No current extrenal channel");
                stream = CreateStream((UBYTE *)"externalchannel");
                stream->handle = CreateHandle();
                tmpn = (int *)Malloc1(sizeof(int),"external channel handle");
                *tmpn = n;
```

```
                RWLOCKW(AM.handlelock);
                filelist[stream->handle] = (FILES *)tmpn;
                UNRWLOCK(AM.handlelock);
            }/*Block*/
            stream->buffer = stream->top = 0;
            stream->inbuffer = 0;
            stream->name = strDup1((UBYTE *)"externalchannel","externalchannel");
            stream->prevline = stream->linenumber = 1;
            stream->eqnum = 0;
            break;
#endif /*ifdef WITHEXTERNALCHANNEL*/
/*:[14apr2004 mt]*/
        default:
            return(0);
    }
    stream->bufferposition = 0;
    stream->isnextchar = 0;
    stream->type = type;
    stream->previousNoShowInput = AC.NoShowInput;
    stream->afterwards = raiselow;
    if ( AC.CurrentStream ) stream->previous = AC.CurrentStream - AC.Streams;
    else stream->previous = -1;
    stream->FoldName = 0;
    if ( prevarmode == 0 ) stream->prevars = -1;
    else if ( prevarmode > 0 ) stream->prevars = NumPre;
    else if ( prevarmode < 0 ) stream->prevars = -prevarmode-1;
    AC.CurrentStream = stream;
    if ( type == PREREADSTREAM || type == PREREADSTREAM3 || type == PRECALCSTREAM
        || type == DOLLARSTREAM ) AC.NoShowInput = 1;
    return(stream);
}

/*
        #] OpenStream :
        #[ LocateFile :
*/

int LocateFile(UBYTE **name, int type)
{
    int handle, namesize, i;
    UBYTE *s, *to, *u1, *u2, *newname, *indir;
    handle = OpenFile((char *)(*name));
    if ( handle >= 0 ) return(handle);
    if ( type == SETUPFILE && AM.SetupFile ) {
        handle = OpenFile((char *)(AM.SetupFile));
        if ( handle >= 0 ) return(handle);
        MesPrint("Could not open setup file %s",(char *)(AM.SetupFile));
    }
    namesize = 4; s = *name;
    while ( *s ) { s++; namesize++; }
    if ( type == SETUPFILE ) indir = AM.SetupDir;
    else indir = AM.IncDir;
    if ( indir ) {

        s = indir; i = 0;
        while ( *s ) { s++; i++; }
        newname = (UBYTE *)Malloc1(namesize+i,"LocateFile");
        s = indir; to = newname;
        while ( *s ) *to++ = *s++;
        if ( to > newname && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
        s = *name;
        while ( *s ) *to++ = *s++;
        *to = 0;
        handle = OpenFile((char *)newname);
        if ( handle >= 0 ) {
            *name = newname;
            return(handle);
        }
        M_free(newname,"LocateFile, incdir/file");
    }
    if ( type == SETUPFILE ) {
        handle = OpenFile(setupfilename);
        if ( handle >= 0 ) return(handle);
        s = (UBYTE *)getenv("FORMSETUP");
        if ( s ) {
            handle = OpenFile((char *)s);
            if ( handle >= 0 ) return(handle);
            MesPrint("Could not open setup file %s",s);
        }
    }
    if ( type != SETUPFILE && AM.Path ) {
        u1 = AM.Path;
        while ( *u1 ) {
            u2 = u1; i = 0;
#ifdef WINDOWS
            while ( *u1 && *u1 != ';' ) {
                u1++; i++;
```

```
            }
#else
            while ( *u1 && *u1 != ':' ) {
                if ( *u1 == '\\' ) u1++;
                u1++; i++;
            }
#endif
            newname = (UBYTE *)Malloc1(namesize+i,"LocateFile");
            s = u2; to = newname;
            while ( s < u1 ) {
#ifndef WINDOWS
                if ( *s == '\\' ) s++;
#endif
                *to++ = *s++;
            }
            if ( to > newname && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
            s = *name;
            while ( *s ) *to++ = *s++;
            *to = 0;
            handle = OpenFile((char *)newname);
            if ( handle >= 0 ) {
                *name = newname;
                return(handle);
            }
            M_free(newname,"LocateFile Path/file");
            if ( *u1 ) u1++;
        }
    }
    if ( type != SETUPFILE ) Error1("LocateFile: Cannot find file",*name);
    return(-1);
}

/*
        #] LocateFile :
        #[ CloseStream :
*/

STREAM *CloseStream(STREAM *stream)
{
    int newstr = stream->previous, sgn;
    UBYTE *t, numbuf[24];
    LONG x;
    if ( stream->FoldName ) {
        M_free(stream->FoldName,"stream->FoldName");
        stream->FoldName = 0;
    }
    if ( stream->type == FILESTREAM || stream->type == REVERSEFILESTREAM ) {
        CloseFile(stream->handle);
        if ( stream->buffer != 0 ) M_free(stream->buffer,"name of input stream");
        stream->buffer = 0;
    }
#ifdef WITHPIPE
    else if ( stream->type == PIPESTREAM ) {
        RWLOCKW(AM.handlelock);
#ifdef WITHMPI
        if ( PF.me == MASTER )
#endif
        pclose((FILE *)(filelist[stream->handle]));
        filelist[stream->handle] = 0;
        numinfilelist--;
        UNRWLOCK(AM.handlelock);
#ifdef WITHMPI
        if ( stream->buffer != 0 ) {
            M_free(stream->buffer, "pipe buffer");
            stream->buffer = 0;
        }
#endif
    }
#endif
/*[14apr2004 mt]:*/
#ifdef WITHEXTERNALCHANNEL
    else if ( stream->type == EXTERNALCHANNELSTREAM ) {
        int *tmpn;
        RWLOCKW(AM.handlelock);
        tmpn = (int *)(filelist[stream->handle]);
        filelist[stream->handle] = 0;
        numinfilelist--;
        UNRWLOCK(AM.handlelock);
        M_free(tmpn,"external channel handle");
    }
#endif /*ifdef WITHEXTERNALCHANNEL*/
/*:[14apr2004 mt]*/
    else if ( stream->type == PREVARSTREAM && (
    stream->afterwards == PRERAISEAFTER || stream->afterwards == PRELOWERAFTER ) ) {
        t = stream->buffer; x = 0; sgn = 1;
        while ( *t == '-' || *t == '+' ) {
            if ( *t == '-' ) sgn = -sgn;
```

```
            t++;
        }
        if ( FG.cTable[*t] == 1 ) {
            while ( *t && FG.cTable[*t] == 1 ) x = 10*x + *t++ - '0';
            if ( *t == 0 ) {
                if ( stream->afterwards == PRERAISEAFTER ) x = sgn*x + 1;
                else x = sgn*x - 1;
                NumToStr(numbuf,x);
                PutPreVar(stream->pname,numbuf,0,1);
            }
        }
    }
    else if ( stream->type == DOLLARSTREAM && (
    stream->afterwards == PRERAISEAFTER || stream->afterwards == PRELOWERAFTER ) ) {
        if ( stream->afterwards == PRERAISEAFTER ) x = 1;
        else x = -1;
        DollarRaiseLow(stream->pname,x);
    }
    else if ( stream->type == PRECALCSTREAM || stream->type == DOLLARSTREAM ) {
        if ( stream->buffer ) M_free(stream->buffer,"stream->buffer");
        stream->buffer = 0;
    }
    if ( stream->name && stream->type != PREVARSTREAM
    && stream->type != PREREADSTREAM && stream->type != PREREADSTREAM2 && stream->type !=
PREREADSTREAM3
    && stream->type != PRECALCSTREAM && stream->type != DOLLARSTREAM ) {
        M_free(stream->name,"stream->name");
    }
    stream->name = 0;
/*  if ( stream->type != FILESTREAM )  */
    AC.NoShowInput = stream->previousNoShowInput;
    stream->buffer = 0;       /* To make sure we will not reuse it */
    stream->pointer = 0;
/*
    Look whether we have to pop preprocessor variables.
*/
    if ( stream->prevars >= 0 ) {
        while ( NumPre > stream->prevars ) {
            NumPre--;
            M_free(PreVar[NumPre].name,"PreVar[NumPre].name");
            PreVar[NumPre].name = PreVar[NumPre].value = 0;
        }
    }
    if ( stream->type == PREVARSTREAM ) {
        AP.AllowDelay = stream->olddelay;
        ClearMacro(stream->pname);
        M_free(stream->pname,"stream->pname");
    }
    else if ( stream->type == DOLLARSTREAM ) {
        M_free(stream->pname,"stream->pname");
    }
    AC.NumStreams--;
    if ( newstr >= 0 ) return(AC.Streams + newstr);
    else return(0);
}

/*
        #] CloseStream :
        #[ CreateStream :
*/

STREAM *CreateStream(UBYTE *where)
{
    STREAM *newstreams;
    int numnewstreams,i;
    int offset;
    if ( AC.NumStreams >= AC.MaxNumStreams ) {
        if ( AC.MaxNumStreams == 0 ) numnewstreams = 10;
        else                         numnewstreams = 2*AC.MaxNumStreams;
        newstreams = (STREAM *)Malloc1(sizeof(STREAM)*(numnewstreams+1),"CreateStream");
        if ( AC.MaxNumStreams > 0 ) {
            offset = AC.CurrentStream - AC.Streams;
            for ( i = 0; i < AC.MaxNumStreams; i++ ) {
                newstreams[i] = AC.Streams[i];
            }
            AC.CurrentStream = newstreams + offset;
        }
        else newstreams[0].previous = -1;
        AC.MaxNumStreams = numnewstreams;
        if ( AC.Streams ) M_free(AC.Streams,(char *)where);
        AC.Streams = newstreams;
    }
    newstreams = AC.Streams+AC.NumStreams++;
    newstreams->name = 0;
    return(newstreams);
}
```

```
/*
        #] CreateStream :
        #[ GetStreamPosition :
*/

LONG GetStreamPosition(STREAM *stream)
{
    return(stream->bufferposition + ((LONG)stream->pointer-(LONG)stream->buffer));
}

/*
        #] GetStreamPosition :
        #[ PositionStream :
*/

VOID PositionStream(STREAM *stream, LONG position)
{
    POSITION scrpos;
    if ( position >= stream->bufferposition
    && position < stream->bufferposition + stream->inbuffer ) {
        stream->pointer = stream->buffer + (position-stream->bufferposition);
    }
    else if ( stream->type == FILESTREAM ) {
        SETBASEPOSITION(scrpos,position);
        SeekFile(stream->handle,&scrpos,SEEK_SET);
        stream->inbuffer = ReadFile(stream->handle,stream->buffer,stream->buffersize);
        stream->pointer = stream->buffer;
        stream->top = stream->buffer + stream->inbuffer;
        stream->bufferposition = position;
        stream->fileposition = position + stream->inbuffer;
        stream->isnextchar = 0;
    }
    else {
        Error0("Illegal position for stream");
        Terminate(-1);
    }
}

/*
        #] PositionStream :
        #[ ReverseStatements :

        Reverses the order of the statements in the buffer.
        We allocate an extra buffer and copy a bit to and fro.
        Note that there are some nasties that cannot be resolved.
*/

int ReverseStatements(STREAM *stream)
{
    UBYTE *spare = (UBYTE *)Malloc1((stream->inbuffer+1)*sizeof(UBYTE),"Reverse copy");
    UBYTE *top = stream->buffer + stream->inbuffer, *in, *s, *ss, *out;
    out = spare+stream->inbuffer+1;
    in = stream->buffer;
    while ( in < top ) {
        s = in;
        if ( *s == AP.ComChar ) {
toeol:;
            for(;;) {
                if ( s == top ) { *--out = '\n'; break; }
                if ( *s == '\\' ) {
                    s++;
                    if ( s >= top ) { /* This is an error! */
irrend:                 MesPrint("@Irregular end of reverse include file.");
                        return(1);
                    }
                }
                else if ( *s == '\n' ) {
                    s++; ss = s;
                    while ( ss > in ) *--out = *--ss;
                    in = s;
                    if ( out[0] == AP.ComChar && ss+6 < s && out[3] == '#' ) {
/*
                        For folds we have to exchange begin and end
*/
                        if ( out[4] == '[' ) out[4] = ']';
                        else if ( out[4] == ']' ) out[4] = '[';
                    }
                    break;
                }
                s++;
            }
            continue;
        }
        while ( s < top && ( *s == ' ' || *s == '\t' ) ) s++;
        if ( *s == '#' ) {  /* preprocessor instruction */
            goto toeol;     /* read to end of line */
        }
```

```
        if ( *s == '.' ) {  /* end-of-module instruction */
            goto toeol;     /* read to end of line */
        }
/*
        Here we have a regular statement. In principle we scan to ; and its \n
        but there are special cases.
        1: ; inside a string (in print "......;";)
        2: multiple statements on one line.
        3: ; + commentary after some blanks.
        4: `var' can cause problems.....
*/
        while ( s < top ) {
            if ( *s == ';' ) {
                s++;
                while ( s < top && ( *s == ' ' || *s == '\t' ) ) s++;
                while ( s < top && *s == '\n' ) s++;
                if ( s >= top && s[-1] != '\n' ) *s++ = '\n';
                ss = s;
                while ( ss > in ) *--out = *--ss;
                in = s;
                break;
            }
            else if ( *s == '"' ) {
                s++;
                while ( s < top ) {
                    if ( *s == '"' ) break;
                    if ( *s == '\\' ) { s++; }
                    s++;
                }
                if ( s >= top ) goto irrend;
            }
            else if ( *s == '\\' ) {
                s++;
                if ( s >= top ) goto irrend;
            }
            s++;
        }
        if ( in < top ) { /* Like blank lines at the end */
            if ( s >= top && s[-1] != '\n' ) *s++ = '\n';
            ss = s;
            while ( ss > in ) *--out = *--ss;
            in = s;
        }
    }
    if ( out == spare ) stream->inbuffer++;
    if ( out > spare+1 ) {
        MesPrint("@Internal error in #reverseinclude instruction.");
        return(1);
    }
    memcpy((void *)(stream->buffer),(void *)out,(size_t)(stream->inbuffer*sizeof(UBYTE)));
    M_free(spare,"Reverse copy");
    return(0);
}

/*
        #] ReverseStatements :
    #] Streams :
    #[ Files :
        #[ StartFiles :
*/

VOID StartFiles()
{
    int i = CreateHandle();
    filelist[i] = Ustdout;
    AM.StdOut = i;
    AC.StoreHandle = -1;
    AC.LogHandle = -1;
#ifndef WITHPTHREADS
    AR.Fscr[0].handle = -1;
    AR.Fscr[1].handle = -1;
    AR.Fscr[2].handle = -1;
    AR.FoStage4[0].handle = -1;
    AR.FoStage4[1].handle = -1;
    AR.infile = &(AR.Fscr[0]);
    AR.outfile = &(AR.Fscr[1]);
    AR.hidefile = &(AR.Fscr[2]);
    AR.StoreData.Handle = -1;
#endif
    AC.Streams = 0;
    AC.MaxNumStreams = 0;
}

/*
        #] StartFiles :
        #[ OpenFile :
*/
```

```c

int OpenFile(char *name)
{
    FILES *f;
    int i;

    if ( ( f = Uopen(name,"rb") ) == 0 ) return(-1);
/*  Usetbuf(f,0); */
    i = CreateHandle();
    RWLOCKW(AM.handlelock);
    filelist[i] = f;
    UNRWLOCK(AM.handlelock);
    return(i);
}

/*
        #] OpenFile :
        #[ OpenAddFile :
*/

int OpenAddFile(char *name)
{
    FILES *f;
    int i;
    POSITION scrpos;
    if ( ( f = Uopen(name,"a+b") ) == 0 ) return(-1);
/*  Usetbuf(f,0); */
    i = CreateHandle();
    RWLOCKW(AM.handlelock);
    filelist[i] = f;
    UNRWLOCK(AM.handlelock);
    TELLFILE(i,&scrpos);
    SeekFile(i,&scrpos,SEEK_SET);
    return(i);
}

/*
        #] OpenAddFile :
        #[ ReOpenFile :
*/

int ReOpenFile(char *name)
{
    FILES *f;
    int i;
    POSITION scrpos;
    if ( ( f = Uopen(name,"r+b") ) == 0 ) return(-1);
    i = CreateHandle();
    RWLOCKW(AM.handlelock);
    filelist[i] = f;
    UNRWLOCK(AM.handlelock);
    TELLFILE(i,&scrpos);
    SeekFile(i,&scrpos,SEEK_SET);
    return(i);
}

/*
        #] ReOpenFile :
        #[ CreateFile :
*/

int CreateFile(char *name)
{
    FILES *f;
    int i;
    if ( ( f = Uopen(name,"w+b") ) == 0 ) return(-1);
    i = CreateHandle();
    RWLOCKW(AM.handlelock);
    filelist[i] = f;
    UNRWLOCK(AM.handlelock);
    return(i);
}

/*
        #] CreateFile :
        #[ CreateLogFile :
*/

int CreateLogFile(char *name)
{
    FILES *f;
    int i;
    if ( ( f = Uopen(name,"w+b") ) == 0 ) return(-1);
    Usetbuf(f,0);
    i = CreateHandle();
    RWLOCKW(AM.handlelock);
    filelist[i] = f;
```

```
    UNRWLOCK(AM.handlelock);
    return(i);
}

/*
        #] CreateLogFile :
        #[ CloseFile :
*/

VOID CloseFile(int handle)
{
    if ( handle >= 0 ) {
        FILES *f;   /* we need this variable to be thread-safe */
        RWLOCKW(AM.handlelock);
        f = filelist[handle];
        filelist[handle] = 0;
        numinfilelist--;
        UNRWLOCK(AM.handlelock);
        Uclose(f);
    }
}

/*
        #] CloseFile :
        #[ CopyFile :
*/

int CopyFile(char *source, char *dest)
{
    #define COPYFILEBUFSIZE 40960L
    FILE *in, *out;
    size_t countin, countout, sumcount;
    char *buffer = NULL;

    sumcount = (AM.S0->LargeSize+AM.S0->SmallEsize)*sizeof(WORD);
    if ( sumcount <= COPYFILEBUFSIZE ) {
        sumcount = COPYFILEBUFSIZE;
        buffer = (char*)Malloc1(sumcount, "file copy buffer");
    }
    else {
        buffer = (char *)(AM.S0->lBuffer);
    }

    in = fopen(source, "rb");
    if ( in == NULL ) {
        perror("CopyFile: ");
        return(1);
    }
    out = fopen(dest, "wb");
    if ( out == NULL ) {
        perror("CopyFile: ");
        return(2);
    }

    while ( !feof(in) ) {
        countin = fread(buffer, 1, sumcount, in);
        if ( countin != sumcount ) {
            if ( ferror(in) ) {
                perror("CopyFile: ");
                return(3);
            }
        }
        countout = fwrite(buffer, 1, countin, out);
        if ( countin != countout ) {
            perror("CopyFile: ");
            return(4);
        }
    }

    fclose(in);
    fclose(out);
    if ( sumcount <= COPYFILEBUFSIZE ) {
        M_free(buffer, "file copy buffer");
    }
    return(0);
}

/*
        #] CopyFile :
        #[ CreateHandle :

        We need a lock here.
        Problem: the same lock is needed inside Malloc1 and M_free which
        is used in DoubleList when we use MALLOCDEBUG

        Conclusion: MALLOCDEBUG will have to be a bit unsafe
*/
```

```c

int CreateHandle()
{
    int i, j;
#ifndef MALLOCDEBUG
    RWLOCKW(AM.handlelock);
#endif
    if ( filelistsize == 0 ) {
        filelistsize = 10;
        filelist = (FILES **)Malloc1(sizeof(FILES *)*filelistsize,"file handle");
        for ( j = 0; j < filelistsize; j++ ) filelist[j] = 0;
        numinfilelist = 1;
        i = 0;
    }
    else if ( numinfilelist >= filelistsize ) {
        VOID **fl = (VOID **)filelist;
        i = filelistsize;
        if ( DoubleList((VOID ***)(&fl),&filelistsize,(int)sizeof(FILES *),
            "list of open files") != 0 ) Terminate(-1);
        filelist = (FILES **)fl;
        for ( j = i; j < filelistsize; j++ ) filelist[j] = 0;
        numinfilelist = i + 1;
    }
    else {
        i = filelistsize;
        for ( j = 0; j < filelistsize; j++ ) {
            if ( filelist[j] == 0 ) { i = j; break; }
        }
        numinfilelist++;
    }
    filelist[i] = (FILES *)(filelist); /* Just for now to not get into problems */
/*
    The next code is not needed when we use open.
    It may be needed when we use fopen.
    fopen is used in minos.c without this central administration.
*/
    if ( numinfilelist > MAX_OPEN_FILES ) {
#ifndef MALLOCDEBUG
        UNRWLOCK(AM.handlelock);
#endif
        MesPrint("More than %d open files",MAX_OPEN_FILES);
        Error0("System limit. This limit is not due to FORM!");
    }
    else {
#ifndef MALLOCDEBUG
        UNRWLOCK(AM.handlelock);
#endif
    }
    return(i);
}

/*
        #] CreateHandle :
        #[ ReadFile :
*/

LONG ReadFile(int handle, UBYTE *buffer, LONG size)
{
    LONG inbuf = 0, r;
    FILES *f;
    char *b;
    b = (char *)buffer;
    for(;;) {   /* Gotta do difficult because of VMS! */
        RWLOCKR(AM.handlelock);
        f = filelist[handle];
        UNRWLOCK(AM.handlelock);
#ifdef WITHSTATS
        numreads++;
#endif
        r = Uread(b,1,size,f);
        if ( r < 0 ) return(r);
        if ( r == 0 ) return(inbuf);
        inbuf += r;
        if ( r == size ) return(inbuf);
        if ( r > size ) return(-1);
        size -= r;
        b += r;
    }
}

/*
        #] ReadFile :
        #[ ReadPosFile :

        Gets words from a file(handle).
        First tries to get the information from the buffers.
        Reads a file at a position. Updates the position.
```

```
        Places a lock in the case of multithreading.
        Exists for multiple reading from the same file.
        size is the number of WORDs to read!!!!

        We may need some strategy in the caching. This routine is used from
        GetOneTerm only. The problem is when it reads brackets and the
        brackets are read backwards. This is very uneconomical because
        each time it may read a large buffer.
        On the other hand, reading piece by piece in GetOneTerm takes
        much overhead as well.
        Two strategies come to mind:
        1: keep things as they are but limit the size of the buffers.
        2: have the position of 'pos' at about 1/3 of the buffer.
           this is of course guess work.
        Currently we have implemented the first method by creating the
        setup parameter threadscratchsize with the default value 100K.
        In the test program much bigger values gave a slower program.
*/

LONG ReadPosFile(PHEAD FILEHANDLE *fi, UBYTE *buffer, LONG size, POSITION *pos)
{
    GETBIDENTITY
    LONG i, retval = 0;
    WORD *b = (WORD *)buffer, *t;

    if ( fi->handle < 0 ) {
        fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + BASEPOSITION(*pos));
        t = fi->POfill;
        while ( size > 0 && fi->POfill < fi->POfull ) { *b++ = *t++; size--; }
    }
    else {
        if ( ISLESSPOS(*pos,fi->POposition) || ISGEPOSINC(*pos,fi->POposition,
            ((UBYTE *)(fi->POfull)-(UBYTE *)(fi->PObuffer))) ) {
/*
            The start is not inside the buffer. Fill the buffer.
*/

            fi->POposition = *pos;
            LOCK(AS.inputslock);
            SeekFile(fi->handle,pos,SEEK_SET);
            retval = ReadFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize);
            UNLOCK(AS.inputslock);
            fi->POfull = fi->PObuffer+retval/sizeof(WORD);
            fi->POfill = fi->PObuffer;
            if ( fi != AR.hidefile ) AR.InInBuf = retval/sizeof(WORD);
            else                     AR.InHiBuf = retval/sizeof(WORD);
        }
        else {
            fi->POfill = (WORD *)((UBYTE *)(fi->PObuffer) + DIFBASE(*pos,fi->POposition));
        }
        if ( fi->POfill + size <= fi->POfull ) {
            t = fi->POfill;
            while ( size > 0 ) { *b++ = *t++; size--; }
        }
        else {
          for (;;) {
            i = fi->POfull - fi->POfill; t = fi->POfill;
            if ( i > size ) i = size;
            size -= i;
            while ( --i >= 0 ) *b++ = *t++;
            if ( size == 0 ) break;
            ADDPOS(fi->POposition,(UBYTE *)(fi->POfull)-(UBYTE *)(fi->PObuffer));
            LOCK(AS.inputslock);
            SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
            retval = ReadFile(fi->handle,(UBYTE *)(fi->PObuffer),fi->POsize);
            UNLOCK(AS.inputslock);
            fi->POfull = fi->PObuffer+retval/sizeof(WORD);
            fi->POfill = fi->PObuffer;
            if ( fi != AR.hidefile ) AR.InInBuf = retval/sizeof(WORD);
            else                     AR.InHiBuf = retval/sizeof(WORD);
            if ( retval == 0 ) { t = fi->POfill; break; }
          }
        }
    }
    retval = (UBYTE *)b - buffer;
    fi->POfill = t;
    ADDPOS(*pos,retval);
    return(retval);
}

/*
        #] ReadPosFile :
        #[ WriteFile :
*/

LONG WriteFileToFile(int handle, UBYTE *buffer, LONG size)
{
```

```
    FILES *f;
    LONG retval, totalwritten = 0, stilltowrite;
    RWLOCKR(AM.handlelock);
    f = filelist[handle];
    UNRWLOCK(AM.handlelock);
    while ( totalwritten < size ) {
        stilltowrite = size - totalwritten;
#ifdef WITHSTATS
        numwrites++;
#endif
        retval = Uwrite((char *)buffer+totalwritten,1,stilltowrite,f);
        if ( retval < 0 ) return(retval);
        if ( retval == 0 ) return(totalwritten);
        totalwritten += retval;
    }
/*
if ( handle == AC.LogHandle || handle == ERROROUT ) FlushFile(handle);
*/
    return(totalwritten);
}
#ifndef WITHMPI
/*[17nov2005]:*/
WRITEFILE WriteFile = &WriteFileToFile;
/*
LONG (*WriteFile)(int handle, UBYTE *buffer, LONG size) = &WriteFileToFile;
*/
/*:[17nov2005]*/
#else
WRITEFILE WriteFile = &PF_WriteFileToFile;
#endif

/*
        #] WriteFile :
        #[ SeekFile :
*/

VOID SeekFile(int handle, POSITION *offset, int origin)
{
    FILES *f;
    RWLOCKR(AM.handlelock);
    f = filelist[handle];
    UNRWLOCK(AM.handlelock);
#ifdef WITHSTATS
    numseeks++;
#endif
    if ( origin == SEEK_SET ) {
        Useek(f,BASEPOSITION(*offset),origin);
        SETBASEPOSITION(*offset,(Utell(f)));
        return;
    }
    else if ( origin == SEEK_END ) {
        Useek(f,0,origin);
    }
    SETBASEPOSITION(*offset,(Utell(f)));
}

/*
        #] SeekFile :
        #[ TellFile :
*/

LONG TellFile(int handle)
{
    POSITION pos;
    TELLFILE(handle,&pos);
#ifdef WITHSTATS
    numseeks++;
#endif
    return(BASEPOSITION(pos));
}

VOID TELLFILE(int handle, POSITION *position)
{
    FILES *f;
    RWLOCKR(AM.handlelock);
    f = filelist[handle];
    UNRWLOCK(AM.handlelock);
    SETBASEPOSITION(*position,(Utell(f)));
}

/*
        #] TellFile :
        #[ FlushFile :
*/

void FlushFile(int handle)
{
```

```
    FILES *f;
    RWLOCKR(AM.handlelock);
    f = filelist[handle];
    UNRWLOCK(AM.handlelock);
    Uflush(f);
}

/*
        #] FlushFile :
        #[ GetPosFile :
*/

int GetPosFile(int handle, fpos_t *pospointer)
{
    FILES *f;
    RWLOCKR(AM.handlelock);
    f = filelist[handle];
    UNRWLOCK(AM.handlelock);
    return(Ugetpos(f,pospointer));
}

/*
        #] GetPosFile :
        #[ SetPosFile :
*/

int SetPosFile(int handle, fpos_t *pospointer)
{
    FILES *f;
    RWLOCKR(AM.handlelock);
    f = filelist[handle];
    UNRWLOCK(AM.handlelock);
    return(Usetpos(f,(fpos_t *)pospointer));
}

/*
        #] SetPosFile :
        #[ SynchFile :

        It may be that when we use many sort files at the same time there
        is a big traffic jam in the cache. This routine is experimental,
        just to see whether this improves the situation.
        It could also be that the internal disk of the Quad opteron norma
        is very slow.
*/

VOID SynchFile(int handle)
{
    FILES *f;
    if ( handle >= 0 ) {
        RWLOCKR(AM.handlelock);
        f = filelist[handle];
        UNRWLOCK(AM.handlelock);
        Usync(f);
    }
}

/*
        #] SynchFile :
        #[ TruncateFile :

        It may be that when we use many sort files at the same time there
        is a big traffic jam in the cache. This routine is experimental,
        just to see whether this improves the situation.
        It could also be that the internal disk of the Quad opteron norma
        is very slow.
*/

VOID TruncateFile(int handle)
{
    FILES *f;
    if ( handle >= 0 ) {
        RWLOCKR(AM.handlelock);
        f = filelist[handle];
        UNRWLOCK(AM.handlelock);
        Utruncate(f);
    }
}

/*
        #] TruncateFile :
        #[ GetChannel :

        Checks whether we have this file already. If so, we return its
        handle. If not, we open the file first and add it to the buffers.
*/

```

```c
int GetChannel(char *name)
{
    CHANNEL *ch;
    int i;
    FILES *f;
    for ( i = 0; i < NumOutputChannels; i++ ) {
        if ( channels[i].name == 0 ) continue;
        if ( StrCmp((UBYTE *)name,(UBYTE *)(channels[i].name)) == 0 ) return(channels[i].handle);
    }
    for ( i = 0; i < NumOutputChannels; i++ ) {
        if ( channels[i].name == 0 ) break;
    }
    if ( i < NumOutputChannels ) { ch = &(channels[i]); }
    else { ch = (CHANNEL *)FromList(&AC.ChannelList); }
    ch->name = (char *)strDup1((UBYTE *)name,"name of channel");
    ch->handle = CreateFile(name);
    RWLOCKR(AM.handlelock);
    f = filelist[ch->handle];
    UNRWLOCK(AM.handlelock);
    Usetbuf(f,0);    /* We turn the buffer off!!!!!!*/
    return(ch->handle);
}

/*
        #] GetChannel :
        #[ GetAppendChannel :

        Checks whether we have this file already. If so, we return its
        handle. If not, we open the file first and add it to the buffers.
*/

int GetAppendChannel(char *name)
{
    CHANNEL *ch;
    int i;
    FILES *f;
    for ( i = 0; i < NumOutputChannels; i++ ) {
        if ( channels[i].name == 0 ) continue;
        if ( StrCmp((UBYTE *)name,(UBYTE *)(channels[i].name)) == 0 ) return(channels[i].handle);
    }
    for ( i = 0; i < NumOutputChannels; i++ ) {
        if ( channels[i].name == 0 ) break;
    }
    if ( i < NumOutputChannels ) { ch = &(channels[i]); }
    else { ch = (CHANNEL *)FromList(&AC.ChannelList); }
    ch->name = (char *)strDup1((UBYTE *)name,"name of channel");
    ch->handle = OpenAddFile(name);
    RWLOCKR(AM.handlelock);
    f = filelist[ch->handle];
    UNRWLOCK(AM.handlelock);
    Usetbuf(f,0);    /* We turn the buffer off!!!!!!*/
    return(ch->handle);
}

/*
        #] GetAppendChannel :
        #[ CloseChannel :

        Checks whether we have this file already. If so, we close it.
*/

int CloseChannel(char *name)
{
    int i;
    for ( i = 0; i < NumOutputChannels; i++ ) {
        if ( channels[i].name == 0 ) continue;
        if ( channels[i].name[0] == 0 ) continue;
        if ( StrCmp((UBYTE *)name,(UBYTE *)(channels[i].name)) == 0 ) {
            CloseFile(channels[i].handle);
            M_free(channels[i].name,"CloseChannel");
            channels[i].name = 0;
            return(0);
        }
    }
    return(0);
}

/*
        #] CloseChannel :
        #[ UpdateMaxSize :

        Updates the maximum size of the combined input/output/hide scratch
        files, the sort files and the .str file.
        The result becomes only visible with either
            ON totalsize;
            #: totalsize ON;
        or the -T in the command tail.
```

```
        To be called, whenever a file is closed/removed or truncated to zero.

        We have no provisions yet for expressions that remain inside the
        small or large buffer during the sort. The space they use there is
        currently ignored.
*/

void UpdateMaxSize()
{
    POSITION position, sumsize;
    int i;
    FILEHANDLE *scr;
#ifdef WITHMPI
    /* Currently, it works only on the master. The sort files on the slaves
     * are ignored. (TU 11 Oct 2011) */
    if ( PF.me != MASTER ) return;
#endif
    PUTZERO(sumsize);
    if ( AM.PrintTotalSize ) {
/*
        First the three scratch files
*/
#ifdef WITHPTHREADS
    scr = AB[0]->R.Fscr;
#else
    scr = AR.Fscr;
#endif
    for ( i = 0; i <=2; i++ ) {
        if ( scr[i].handle < 0 ) {
            SETBASEPOSITION(position,(scr[i].POfull-scr[i].PObuffer)*sizeof(WORD));
        }
        else {
            position = scr[i].filesize;
        }
        ADD2POS(sumsize,position);
    }
/*
        Now the sort file(s)
*/
#ifdef WITHPTHREADS
    {
        int j;
        ALLPRIVATES *B;
        for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
            B = AB[j];
            if ( AT.SS && AT.SS->file.handle >= 0 ) {
                position = AT.SS->file.filesize;
/*
MLOCK(ErrorMessageLock);
MesPrint("%d: %10p",j,&(AT.SS->file.filesize));
MUNLOCK(ErrorMessageLock);
*/
                ADD2POS(sumsize,position);
            }
            if ( AR.FoStage4[0].handle >= 0 ) {
                position = AR.FoStage4[0].filesize;
                ADD2POS(sumsize,position);
            }
        }
    }
#else
    if ( AT.SS && AT.SS->file.handle >= 0 ) {
        position = AT.SS->file.filesize;
        ADD2POS(sumsize,position);
    }
    if ( AR.FoStage4[0].handle >= 0 ) {
        position = AR.FoStage4[0].filesize;
        ADD2POS(sumsize,position);
    }
#endif
/*
        And of course the str file.
*/
    ADD2POS(sumsize,AC.StoreFileSize);
/*
        Finally the test whether it is bigger
*/
    if ( ISLESSPOS(AS.MaxExprSize,sumsize) ) {
#ifdef WITHPTHREADS
        LOCK(AS.MaxExprSizeLock);
        if ( ISLESSPOS(AS.MaxExprSize,sumsize) ) AS.MaxExprSize = sumsize;
        UNLOCK(AS.MaxExprSizeLock);
#else
        AS.MaxExprSize = sumsize;
#endif
    }
```

```
    }
    return;
}

/*
        #] UpdateMaxSize :
    #] Files :
    #[ Strings :
        #[ StrCmp :
*/

int StrCmp(UBYTE *s1, UBYTE *s2)
{
    while ( *s1 && *s1 == *s2 ) { s1++; s2++; }
    return((int)*s1-(int)*s2);
}

/*
        #] StrCmp :
        #[ StrICmp :
*/

int StrICmp(UBYTE *s1, UBYTE *s2)
{
    while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
    return((int)tolower(*s1)-(int)tolower(*s2));
}

/*
        #] StrICmp :
        #[ StrHICmp :
*/

int StrHICmp(UBYTE *s1, UBYTE *s2)
{
    while ( *s1 && tolower(*s1) == *s2 ) { s1++; s2++; }
    return((int)tolower(*s1)-(int)(*s2));
}

/*
        #] StrHICmp :
        #[ StrICont :
*/

int StrICont(UBYTE *s1, UBYTE *s2)
{
    while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
    if ( *s1 == 0 ) return(0);
    return((int)tolower(*s1)-(int)tolower(*s2));
}

/*
        #] StrICont :
        #[ ConWord :
*/

int ConWord(UBYTE *s1, UBYTE *s2)
{
    while ( *s1 && ( tolower(*s1) == tolower(*s2) ) ) { s1++; s2++; }
    if ( *s1 == 0 ) return(1);
    return(0);
}

/*
        #] ConWord :
        #[ StrLen :
*/

int StrLen(UBYTE *s)
{
    int i = 0;
    while ( *s ) { s++; i++; }
    return(i);
}

/*
        #] StrLen :
        #[ NumToStr :
*/

VOID NumToStr(UBYTE *s, LONG x)
{
    UBYTE *t, str[24];
    ULONG xx;
    t = str;
    if ( x < 0 ) { *s++ = '-'; xx = -x; }
    else xx = x;
```

```
    do {
        *t++ = xx % 10 + '0';
        xx /= 10;
    } while ( xx );
    while ( t > str ) *s++ = *--t;
    *s = 0;
}

/*
        #] NumToStr :
        #[ WriteString :

        Writes a characterstring to the various outputs.
        The action may depend on the flags involved.
        The type of output is given by type, the string by str and the
        number of characters in it by num
*/
VOID WriteString(int type, UBYTE *str, int num)
{
    int error = 0;

    if ( num > 0 && str[num-1] == 0 ) { num--; }
    else if ( num <= 0 || str[num-1] != LINEFEED ) {
        AddLineFeed(str,num);
    }
    /*[15apr2004 mt]:*/
    if(type == EXTERNALCHANNELOUT){
        if(WriteFile(0,str,num) != num) error = 1;
    }else
    /*:[15apr2004 mt]*/
    if ( AM.silent == 0 || type == ERROROUT ) {
        if ( type == INPUTOUT ) {
            if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut,(UBYTE *)"    ",4) != 4 ) error = 1;
            if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,(UBYTE *)"    ",4) != 4 ) error = 1;
        }
        if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut,str,num) != num ) error = 1;
        if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,str,num) != num ) error = 1;
    }
    if ( error ) Terminate(-1);
}

/*
        #] WriteString :
        #[ WriteUnfinString :

        Writes a characterstring to the various outputs.
        The action may depend on the flags involved.
        The type of output is given by type, the string by str and the
        number of characters in it by num
*/

VOID WriteUnfinString(int type, UBYTE *str, int num)
{
    int error = 0;

    /*[15apr2004 mt]:*/
    if(type == EXTERNALCHANNELOUT){
        if(WriteFile(0,str,num) != num) error = 1;
    }else
    /*:[15apr2004 mt]*/
    if ( AM.silent == 0 || type == ERROROUT ) {
        if ( type == INPUTOUT ) {
            if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut,(UBYTE *)"    ",4) != 4 ) error = 1;
            if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,(UBYTE *)"    ",4) != 4 ) error = 1;
        }
        if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut,str,num) != num ) error = 1;
        if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,str,num) != num ) error = 1;
    }
    if ( error ) Terminate(-1);
}

/*
        #] WriteUnfinString :
        #[ AddToString :
*/

UBYTE *AddToString(UBYTE *outstring, UBYTE *extrastring, int par)
{
    UBYTE *s = extrastring, *t, *newstring;
    int n, nn;
    while ( *s ) { s++; }
    n = s-extrastring;
    if ( outstring == 0 ) {
        s = extrastring;
        t = outstring = (UBYTE *)Malloc1(n+1,"AddToString");
        NCOPY(t,s,n)
        *t++ = 0;
```

```
        return(outstring);
    }
    else {
        t = outstring;
        while ( *t ) t++;
        nn = t - outstring;
        t = newstring = (UBYTE *)Malloc1(n+nn+2,"AddToString");
        s = outstring;
        NCOPY(t,s,nn)
        if ( par == 1 ) *t++ = ',';
        s = extrastring;
        NCOPY(t,s,n)
        *t = 0;
        M_free(outstring,"AddToString");
        return(newstring);
    }
}

/*
        #] AddToString :
        #[ strDup1 :

        string duplication with message passing for Malloc1, allowing
        this routine to give a more detailed error message if there
        is not enough memory.
*/

UBYTE *strDup1(UBYTE *instring, char *ifwrong)
{
    UBYTE *s = instring, *to;
    while ( *s ) s++;
    to = s = (UBYTE *)Malloc1((s-instring)+1,ifwrong);
    while ( *instring ) *to++ = *instring++;
    *to = 0;
    return(s);
}

/*
        #] strDup1 :
        #[ EndOfToken :
*/

UBYTE *EndOfToken(UBYTE *s)
{
    UBYTE c;
    while ( ( c = (UBYTE)(FG.cTable[*s]) ) == 0 || c == 1 ) s++;
    return(s);
}

/*
        #] EndOfToken :
        #[ ToToken :
*/

UBYTE *ToToken(UBYTE *s)
{
    UBYTE c;
    while ( *s && ( c = (UBYTE)(FG.cTable[*s]) ) != 0 && c != 1 ) s++;
    return(s);
}

/*
        #] ToToken :
        #[ SkipField :

    Skips from s to the end of a declaration field.
    par is the number of parentheses that still has to be closed.
*/

UBYTE *SkipField(UBYTE *s, int level)
{
    while ( *s ) {
        if ( *s == ',' && level == 0 ) return(s);
        if ( *s == '(' ) level++;
        else if ( *s == ')' ) { level--; if ( level < 0 ) level = 0; }
        else if ( *s == '[' ) {
            SKIPBRA1(s)
        }
        else if ( *s == '{' ) {
            SKIPBRA2(s)
        }
        s++;
    }
    return(s);
}

/*
```

```
        #] SkipField :
        #[ ReadSnum :           WORD ReadSnum(p)

        Reads a number that should fit in a word.
        The number should be unsigned and a negative return value
        indicates an irregularity.

*/

WORD ReadSnum(UBYTE **p)
{
    LONG x = 0;
    UBYTE *s;
    s = *p;
    if ( FG.cTable[*s] == 1 ) {
        do {
            x = ( x « 3 ) + ( x « 1 ) + ( *s++ - '0' );
            if ( x > MAXPOSITIVE ) return(-1);
        } while ( FG.cTable[*s] == 1 );
        *p = s;
        return((WORD)x);
    }
    else return(-1);
}

/*
        #] ReadSnum :
        #[ NumCopy :

    Adds the decimal representation of a number to a string.

*/

UBYTE *NumCopy(WORD y, UBYTE *to)
{
    UBYTE *s;
    WORD i = 0, j;
    UWORD x;
    if ( y < 0 ) {
        x = (UWORD)(-y);
        *to++ = '-';
    }
    else {
        x = (UWORD)y;
    }
    s = to;
    do { *s++ = (UBYTE)((x % 10)+'0'); i++; } while ( ( x /= 10 ) != 0 );
    *s-- = '\0';
    j = ( i - 1 ) » 1;
    while ( j >= 0 ) {
        i = to[j]; to[j] = s[-j]; s[-j] = (UBYTE)i; j--;
    }
    return(s+1);
}

/*
        #] NumCopy :
        #[ LongCopy :

    Adds the decimal representation of a number to a string.

*/

char *LongCopy(LONG y, char *to)
{
    char *s;
    WORD i = 0, j;
    ULONG x;
    if ( y < 0 ) {
        x = (ULONG)(-y);
        *to++ = '-';
    }
    else {
        x = (ULONG)y;
    }
    s = to;
    do { *s++ = (x % 10)+'0'; i++; } while ( ( x /= 10 ) != 0 );
    *s-- = '\0';
    j = ( i - 1 ) » 1;
    while ( j >= 0 ) {
        i = to[j]; to[j] = s[-j]; s[-j] = (char)i; j--;
    }
    return(s+1);
}

/*
        #] LongCopy :
```

```
        #[ LongLongCopy :

    Adds the decimal representation of a number to a string.
    Bugfix feb 2003. y was not pointer!
*/

char *LongLongCopy(off_t *y, char *to)
{
    /*
     * This code fails to print the maximum negative value on systems with two's
     * complement. To fix this, we need the unsigned version of off_t with the
     * same size, but unfortunately it is undefined. On the other hand, if a
     * system is configured with a 64-bit off_t, in practice one never reaches
     * 2^63 ~ 10^18 as of 2016. If one really reach such a big number, then it
     * would be the time to move on a 128-bit off_t.
     */
    off_t x = *y;
    char *s;
    WORD i = 0, j;
    if ( x < 0 ) { x = -x; *to++ = '-'; }
    s = to;
    do { *s++ = (x % 10)+'0'; i++; } while ( ( x /= 10 ) != 0 );
    *s-- = '\0';
    j = ( i - 1 ) » 1;
    while ( j >= 0 ) {
        i = to[j]; to[j] = s[-j]; s[-j] = (char)i; j--;
    }
    return(s+1);
}

/*
        #] LongLongCopy :
        #[ MakeDate :

        Routine produces a string with the date and time of the run
*/

#ifdef ANSI
#else
#ifdef mBSD
#else
static char notime[] = "";
#endif
#endif

UBYTE *MakeDate()
{
#ifdef ANSI
    time_t tp;
    time(&tp);
    return((UBYTE *)ctime(&tp));
#else
#ifdef mBSD
    time_t tp;
    time(&tp);
    return((UBYTE *)ctime(&tp));
#else
    return((UBYTE *)notime);
#endif
#endif
}

/*
        #] MakeDate :
        #[ set_in :
         Returns 1 if ch is in set ; 0 if ch is not in set:
*/
int set_in(UBYTE ch, set_of_char set)
{
    set += ch/8;
    switch (ch % 8){
        case 0: return(set->bit_0);
        case 1: return(set->bit_1);
        case 2: return(set->bit_2);
        case 3: return(set->bit_3);
        case 4: return(set->bit_4);
        case 5: return(set->bit_5);
        case 6: return(set->bit_6);
        case 7: return(set->bit_7);
    }/*switch (ch % 8)*/
    return(-1);
}/*set_in*/
/*
        #] set_in :
        #[ set_set :
            sets ch into set; returns *set:
*/
```

```
one_byte set_set(UBYTE ch, set_of_char set)
{
    one_byte tmp=(one_byte)set;
    set += ch/8;
    switch (ch % 8){
        case 0: set->bit_0=1;break;
        case 1: set->bit_1=1;break;
        case 2: set->bit_2=1;break;
        case 3: set->bit_3=1;break;
        case 4: set->bit_4=1;break;
        case 5: set->bit_5=1;break;
        case 6: set->bit_6=1;break;
        case 7: set->bit_7=1;break;
    }
    return(tmp);
}/*set_set*/
/*
        #] set_set :
        #[ set_del :
            deletes ch from set; returns *set:
*/
one_byte set_del(UBYTE ch, set_of_char set)
{
    one_byte tmp=(one_byte)set;
    set += ch/8;
    switch (ch % 8){
        case 0: set->bit_0=0;break;
        case 1: set->bit_1=0;break;
        case 2: set->bit_2=0;break;
        case 3: set->bit_3=0;break;
        case 4: set->bit_4=0;break;
        case 5: set->bit_5=0;break;
        case 6: set->bit_6=0;break;
        case 7: set->bit_7=0;break;
    }
    return(tmp);
}/*set_del*/
/*
        #] set_del :
        #[ set_sub :
            returns *set = set1\set2. This function may be usd for initialising,
                set_sub(a,a,a) => now a is empty set :
*/
one_byte set_sub(set_of_char set, set_of_char set1, set_of_char set2)
{
    one_byte tmp=(one_byte)set;
    int i=0,j=0;
    while(j=0,i++<32)
    while(j<9)
        switch (j++){
            case 0: set->bit_0=(set1->bit_0&&(!set2->bit_0));break;
            case 1: set->bit_1=(set1->bit_1&&(!set2->bit_1));break;
            case 2: set->bit_2=(set1->bit_2&&(!set2->bit_2));break;
            case 3: set->bit_3=(set1->bit_3&&(!set2->bit_3));break;
            case 4: set->bit_4=(set1->bit_4&&(!set2->bit_4));break;
            case 5: set->bit_5=(set1->bit_5&&(!set2->bit_5));break;
            case 6: set->bit_6=(set1->bit_6&&(!set2->bit_6));break;
            case 7: set->bit_7=(set1->bit_7&&(!set2->bit_7));break;
            case 8: set++;set1++;set2++;
     };
    return(tmp);
}/*set_sub*/
/*
        #] set_sub :
    #] Strings :
    #[ Mixed :
        #[ iniTools :
*/

VOID iniTools(VOID)
{
#ifdef MALLOCPROTECT
    if ( mprotectInit() ) exit(0);
#endif
    return;
}

/*
        #] iniTools :
        #[ Malloc :

        Malloc routine with built in error checking.
        This saves lots of messages.
*/
#ifdef MALLOCDEBUG
char *dummymessage = "Malloc";
INILOCK(MallocLock);
```

```c
#endif

VOID *Malloc(LONG size)
{
    VOID *mem;
#ifdef MALLOCDEBUG
    char *t, *u;
    int i;
    LOCK(MallocLock);
/*  MLOCK(ErrorMessageLock); */
    if ( size == 0 ) {
        MesPrint("Asking for 0 bytes in Malloc");
    }
#endif
    if ( ( size & 7 ) != 0 ) { size = size - ( size&7 ) + 8; }
#ifdef MALLOCDEBUG
    size += 2*BANNER;
#endif
    mem = (VOID *)M_alloc(size);
    if ( mem == 0 ) {
#ifndef MALLOCDEBUG
        MLOCK(ErrorMessageLock);
#endif
        Error0("No memory!");
#ifndef MALLOCDEBUG
        MUNLOCK(ErrorMessageLock);
#else
/*      MUNLOCK(ErrorMessageLock); */
#endif
#ifdef MALLOCDEBUG
        UNLOCK(MallocLock);
#endif
        Terminate(-1);
    }
#ifdef MALLOCDEBUG
    mallocsizes[nummalloclist] = size;
    mallocstrings[nummalloclist] = dummymessage;
    malloclist[nummalloclist++] = mem;
    if ( filelist ) MesPrint("Mem0 at 0x%x, %l bytes",mem,size);
    {
        int i = nummalloclist-1;
        while ( --i >= 0 ) {
            if ( (char *)mem < (((char *)malloclist[i]) + mallocsizes[i])
            && (char *)(malloclist[i]) < ((char *)mem + size) ) {
                if ( filelist ) MesPrint("This memory overlaps with the block at 0x%x"
                    ,malloclist[i]);
            }
        }
    }
    t = (char *)mem;
    u = t + size;
    for ( i = 0; i < (int)BANNER; i++ ) { *t++ = FILLVALUE; *--u = FILLVALUE; }
    mem = (void *)t;
    {
        int j = nummalloclist-1, i;
        while ( --j >= 0 ) {
            t = (char *)(malloclist[j]);
            u = t + mallocsizes[j];
            for ( i = 0; i < (int)BANNER; i++ ) {
                u--;
                if ( *t != FILLVALUE || *u != FILLVALUE ) {
                    MesPrint("Writing outside memory for %s",malloclist[i]);
/*                  MUNLOCK(ErrorMessageLock); */
                    UNLOCK(MallocLock);
                    Terminate(-1);
                }
                t--;
            }
        }
    }
/*  MUNLOCK(ErrorMessageLock); */
    UNLOCK(MallocLock);
#endif
    return(mem);
}

/*
        #] Malloc :
        #[ Malloc1 :

        Malloc with more detailed error message.
        Gives the user some idea of what is happening.
*/

VOID *Malloc1(LONG size, const char *messageifwrong)
{
    VOID *mem;
```

```
#ifdef MALLOCDEBUG
    char *t, *u;
    int i;
    LOCK(MallocLock);
/*  MLOCK(ErrorMessageLock); */
    if ( size == 0 ) {
        MesPrint("%wAsking for 0 bytes in Malloc1");
    }
#endif
#ifdef WITHSTATS
    nummallocs++;
#endif
    if ( ( size & 7 ) != 0 ) { size = size - ( size&7 ) + 8; }
#ifdef MALLOCDEBUG
    size += 2*BANNER;
#endif
    mem = (VOID *)M_alloc(size);
    if ( mem == 0 ) {
#ifndef MALLOCDEBUG
        MLOCK(ErrorMessageLock);
#endif
        Error1("No memory while allocating ",(UBYTE *)messageifwrong);
#ifndef MALLOCDEBUG
        MUNLOCK(ErrorMessageLock);
#else
/*      MUNLOCK(ErrorMessageLock); */
#endif
#ifdef MALLOCDEBUG
        UNLOCK(MallocLock);
#endif
        Terminate(-1);
    }
#ifdef MALLOCDEBUG
    mallocsizes[nummalloclist] = size;
    mallocstrings[nummalloclist] = (char *)messageifwrong;
    malloclist[nummalloclist++] = mem;
    if ( AC.MemDebugFlag && filelist ) MesPrint("%wMem1 at 0x%x: %l bytes.
%s",mem,size,messageifwrong);
    {
        int i = nummalloclist-1;
        while ( --i >= 0 ) {
            if ( (char *)mem < (((char *)malloclist[i]) + mallocsizes[i])
            && (char *)(malloclist[i]) < ((char *)mem + size) ) {
                if ( filelist ) MesPrint("This memory overlaps with the block at 0x%x"
                    ,malloclist[i]);
            }
        }
    }

#ifdef MALLOCDEBUGOUTPUT
    printf ("Malloc1: %s, allocated %li bytes at %.8lx\n",messageifwrong,size,(unsigned long)mem);
    fflush (stdout);
#endif

    t = (char *)mem;
    u = t + size;
    for ( i = 0; i < (int)BANNER; i++ ) { *t++ = FILLVALUE; *--u = FILLVALUE; }
    mem = (void *)t;
    M_check();
/*  MUNLOCK(ErrorMessageLock); */
    UNLOCK(MallocLock);
#endif
/*
    if ( size > 500000000L ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Malloc1: %s, allocated %l bytes\n",messageifwrong,size);
        MUNLOCK(ErrorMessageLock);
    }
*/
    return(mem);
}

/*
        #] Malloc1 :
        #[ M_free :
*/

void M_free(VOID *x, const char *where)
{
#ifdef MALLOCDEBUG
    char *t = (char *)x;
    int i, j, k;
    LONG size = 0;
    x = (void *)(((char *)x)-BANNER);
/*  MLOCK(ErrorMessageLock); */
    if ( AC.MemDebugFlag ) MesPrint("%wFreeing 0x%x: %s",x,where);
    LOCK(MallocLock);
```

```
    for ( i = nummalloclist-1; i >= 0; i-- ) {
        if ( x == malloclist[i] ) {
            size = mallocsizes[i];
            for ( j = i+1; j < nummalloclist; j++ ) {
                malloclist[j-1] = malloclist[j];
                mallocsizes[j-1] = mallocsizes[j];
                mallocstrings[j-1] = mallocstrings[j];
            }
            nummalloclist--;
            break;
        }
    }
    if ( i < 0 ) {
        unsigned int xx = ((ULONG)x);
        printf("Error returning non-allocated address: 0x%x from %s\n"
            ,xx,where);
/*      MUNLOCK(ErrorMessageLock); */
        UNLOCK(MallocLock);
        exit(-1);
    }
    else {
        for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
            if ( *--t != FILLVALUE ) j++;
        }
        if ( j ) {
            LONG *tt = (LONG *)x;
            MesPrint("%w!!!!! Banner has been written in !!!!!: %x %x %x %x",
            tt[0],tt[1],tt[2],tt[3]);
        }
        t += size;
        for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
            if ( *--t != FILLVALUE ) j++;
        }
        if ( j ) {
            LONG *tt = (LONG *)x;
            MesPrint("%w!!!!! Tail has been written in !!!!!: %x %x %x %x",
            tt[0],tt[1],tt[2],tt[3]);
        }
        M_check();
/*      MUNLOCK(ErrorMessageLock); */
        UNLOCK(MallocLock);
    }
#else
    DUMMYUSE(where);
#endif
#ifdef WITHSTATS
    numfrees++;
#endif
    if ( x ) {
#ifdef MALLOCDEBUGOUTPUT
        printf ("M_free: %s, memory freed at %.8lx\n",where,(unsigned long)x);
        fflush(stdout);
#endif

#ifdef MALLOCPROTECT
        mprotectFree((void *)x);
#else
        free(x);
#endif
    }
}

/*
        #] M_free :
        #[ M_check :
*/

#ifdef MALLOCDEBUG

void M_check1() { MesPrint("Checking Malloc"); M_check(); }

void M_check()
{
    int i,j,k,error = 0;
    char *t;
    LONG *tt;
    for ( i = 0; i < nummalloclist; i++ ) {
        t = (char *)(malloclist[i]);
        for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
            if ( *t++ != FILLVALUE ) j++;
        }
        if ( j ) {
            tt = (LONG *)(malloclist[i]);
            MesPrint("%w!!!!! Banner %d (%s) has been written in !!!!!: %x %x %x %x",
            i,mallocstrings[i],tt[0],tt[1],tt[2],tt[3]);
            tt[0] = tt[1] = tt[2] = tt[3] = 0;
            error = 1;
```

```
        }
        t = (char *)(malloclist[i]) + mallocsizes[i];
        for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
            if ( *--t != FILLVALUE ) j++;
        }
        if ( j ) {
            tt = (LONG *)t;
            MesPrint("%w!!!!! Tail %d (%s) has been written in !!!!!: %x %x %x %x",
            i,mallocstrings[i],tt[0],tt[1],tt[2],tt[3]);
            tt[0] = tt[1] = tt[2] = tt[3] = 0;
            error = 1;
        }
        if ( ( mallocstrings[i][0] == ' ' ) || ( mallocstrings[i][0] == '#' ) ) {
            MesPrint("%w!!!!! Funny mallocstring");
            error = 1;
        }
    }
    if ( error ) {
        M_print();
/*      MUNLOCK(ErrorMessageLock); */
        UNLOCK(MallocLock);
        Terminate(-1);
    }
}

void M_print()
{
    int i;
    MesPrint("We have the following memory allocations left:");
    for ( i = 0; i < nummalloclist; i++ ) {
        MesPrint("0x%x: %l bytes. number %d: '%s'",malloclist[i],mallocsizes[i],i,mallocstrings[i]);
    }
}

#else

void M_check1() {}
void M_print() {}

#endif

/*
        #] M_check :
        #[ TermMalloc :
*/
#define TERMMEMSTARTNUM 16
#define TERMEXTRAWORDS 10

VOID TermMallocAddMemory(PHEAD0)
{
    WORD *newbufs;
    int i, extra;
    if ( AT.TermMemMax == 0 ) extra = TERMMEMSTARTNUM;
    else                      extra = AT.TermMemMax;
    if ( AT.TermMemHeap ) M_free(AT.TermMemHeap,"TermMalloc");
    newbufs = (WORD *)Malloc1(extra*(AM.MaxTer+TERMEXTRAWORDS*sizeof(WORD)),"TermMalloc");
    AT.TermMemHeap = (WORD **)Malloc1((extra+AT.TermMemMax)*sizeof(WORD *),"TermMalloc");
    for ( i = 0; i < extra; i++ ) {
        AT.TermMemHeap[i] = newbufs + i*(AM.MaxTer/sizeof(WORD)+TERMEXTRAWORDS);
    }
#ifdef TERMMALLOCDEBUG
    DebugHeap2 = (WORD **)Malloc1((extra+AT.TermMemMax)*sizeof(WORD *),"TermMalloc");
    for ( i = 0; i < AT.TermMemMax; i++ ) { DebugHeap2[i] = DebugHeap1[i]; }
    for ( i = 0; i < extra; i++ ) {
        DebugHeap2[i+AT.TermMemMax] = newbufs + i*(AM.MaxTer/sizeof(WORD)+TERMEXTRAWORDS);
    }
    if ( DebugHeap1 ) M_free(DebugHeap1,"TermMalloc");
    DebugHeap1 = DebugHeap2;
#endif
    AT.TermMemTop = extra;
    AT.TermMemMax += extra;
#ifdef TERMMALLOCDEBUG
    MesPrint("AT.TermMemMax is now %l",AT.TermMemMax);
#endif
}

#ifndef MEMORYMACROS

WORD *TermMalloc2(PHEAD char *text)
{
    if ( AT.TermMemTop <= 0 ) TermMallocAddMemory(BHEAD0);

#ifdef TERMMALLOCDEBUG
    MesPrint("TermMalloc: %s, %d",text,(AT.TermMemMax-AT.TermMemTop));
#endif

#ifdef MALLOCDEBUGOUTPUT
```

```
    MesPrint("TermMalloc: %s, %l/%l (%x)",text,AT.TermMemTop,AT.TermMemMax,AT.TermMemHeap[AT.TermMemTop-1]);
#endif

    DUMMYUSE(text);
    return(AT.TermMemHeap[--AT.TermMemTop]);
}

VOID TermFree2(PHEAD WORD *TermMem, char *text)
{
#ifdef TERMMALLOCDEBUG

    int i;

    for ( i = 0; i < AT.TermMemMax; i++ ) {
        if ( TermMem == DebugHeap1[i] ) break;
    }
    if ( i >= AT.TermMemMax ) {
        MesPrint(" ERROR: TermFree called with an address not given by TermMalloc.");
        Terminate(-1);
    }
#endif
    DUMMYUSE(text);
    AT.TermMemHeap[AT.TermMemTop++] = TermMem;

#ifdef TERMMALLOCDEBUG
    MesPrint("TermFree: %s, %d",text,(AT.TermMemMax-AT.TermMemTop));
#endif
#ifdef MALLOCDEBUGOUTPUT
    MesPrint("TermFree: %s, %l/%l (%x)",text,AT.TermMemTop,AT.TermMemMax,TermMem);
#endif
}

#endif

/*
        #] TermMalloc :
        #[ NumberMalloc :
*/
#define NUMBERMEMSTARTNUM 16
#define NUMBEREXTRAWORDS 10L

#ifdef TERMMALLOCDEBUG
UWORD **DebugHeap3, **DebugHeap4;
#endif

VOID NumberMallocAddMemory(PHEAD0)
{
    UWORD *newbufs;
    WORD extra;
    int i;
    if ( AT.NumberMemMax == 0 ) extra = NUMBERMEMSTARTNUM;
    else                        extra = AT.NumberMemMax;
    if ( AT.NumberMemHeap ) M_free(AT.NumberMemHeap,"NumberMalloc");
    newbufs = (UWORD *)Malloc1(extra*(AM.MaxTal+NUMBEREXTRAWORDS)*sizeof(UWORD),"NumberMalloc");
    AT.NumberMemHeap = (UWORD **)Malloc1((extra+AT.NumberMemMax)*sizeof(UWORD *),"NumberMalloc");
    for ( i = 0; i < extra; i++ ) {
        AT.NumberMemHeap[i] = newbufs + i*(LONG)(AM.MaxTal+NUMBEREXTRAWORDS);
    }
#ifdef TERMMALLOCDEBUG
    DebugHeap4 = (UWORD **)Malloc1((extra+AT.NumberMemMax)*sizeof(WORD *),"NumberMalloc");
    for ( i = 0; i < AT.NumberMemMax; i++ ) { DebugHeap4[i] = DebugHeap3[i]; }
    for ( i = 0; i < extra; i++ ) {
        DebugHeap4[i+AT.NumberMemMax] = newbufs + i*(LONG)(AM.MaxTal+NUMBEREXTRAWORDS);
    }
    if ( DebugHeap3 ) M_free(DebugHeap3,"NumberMalloc");
    DebugHeap3 = DebugHeap4;
#endif
    AT.NumberMemTop = extra;
    AT.NumberMemMax += extra;
/*
MesPrint("AT.NumberMemMax is now %l",AT.NumberMemMax);
*/
}

#ifndef MEMORYMACROS

UWORD *NumberMalloc2(PHEAD char *text)
{
    if ( AT.NumberMemTop <= 0 ) NumberMallocAddMemory(BHEAD text);

#ifdef MALLOCDEBUGOUTPUT
    if ( (AT.NumberMemMax-AT.NumberMemTop) > 10 )
    MesPrint("NumberMalloc: %s, %l/%l (%x)",text,AT.NumberMemTop,AT.NumberMemMax,AT.NumberMemHeap[AT.NumberMemTop-1]);
#endif

```

```
    DUMMYUSE(text);
    return(AT.NumberMemHeap[--AT.NumberMemTop]);
}

VOID NumberFree2(PHEAD UWORD *NumberMem, char *text)
{
#ifdef TERMMALLOCDEBUG
    int i;
    for ( i = 0; i < AT.NumberMemMax; i++ ) {
        if ( NumberMem == DebugHeap3[i] ) break;
    }
    if ( i >= AT.NumberMemMax ) {
        MesPrint(" ERROR: NumberFree called with an address not given by NumberMalloc.");
        Terminate(-1);
    }
#endif
    DUMMYUSE(text);
    AT.NumberMemHeap[AT.NumberMemTop++] = NumberMem;

#ifdef MALLOCDEBUGOUTPUT
    if ( (AT.NumberMemMax-AT.NumberMemTop) > 10 )
    MesPrint("NumberFree: %s, %l/%l (%x)",text,AT.NumberMemTop,AT.NumberMemMax,NumberMem);
#endif
}

#endif

/*
        #] NumberMalloc :
        #[ CacheNumberMalloc :

    Similar to NumberMalloc
 */

VOID CacheNumberMallocAddMemory(PHEAD0)
{
    UWORD *newbufs;
    WORD extra;
    int i;
    if ( AT.CacheNumberMemMax == 0 ) extra = NUMBERMEMSTARTNUM;
    else                             extra = AT.CacheNumberMemMax;
    if ( AT.CacheNumberMemHeap ) M_free(AT.CacheNumberMemHeap,"NumberMalloc");
    newbufs = (UWORD *)Malloc1(extra*(AM.MaxTal+NUMBEREXTRAWORDS)*sizeof(UWORD),"CacheNumberMalloc");
    AT.CacheNumberMemHeap = (UWORD **)Malloc1((extra+AT.NumberMemMax)*sizeof(UWORD
*),"CacheNumberMalloc");
    for ( i = 0; i < extra; i++ ) {
        AT.CacheNumberMemHeap[i] = newbufs + i*(LONG)(AM.MaxTal+NUMBEREXTRAWORDS);
    }
    AT.CacheNumberMemTop = extra;
    AT.CacheNumberMemMax += extra;
}

#ifndef MEMORYMACROS

UWORD *CacheNumberMalloc2(PHEAD char *text)
{
    if ( AT.CacheNumberMemTop <= 0 ) CacheNumberMallocAddMemory(BHEAD0);

#ifdef MALLOCDEBUGOUTPUT
    MesPrint("NumberMalloc: %s, %l/%l
(%x)",text,AT.NumberMemTop,AT.NumberMemMax,AT.NumberMemHeap[AT.NumberMemTop-1]);
#endif

    DUMMYUSE(text);
    return(AT.CacheNumberMemHeap[--AT.CacheNumberMemTop]);
}

VOID CacheNumberFree2(PHEAD UWORD *NumberMem, char *text)
{
    DUMMYUSE(text);
    AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = NumberMem;

#ifdef MALLOCDEBUGOUTPUT
    MesPrint("NumberFree: %s, %l/%l (%x)",text,AT.NumberMemTop,AT.NumberMemMax,NumberMem);
#endif
}

#endif

/*
        #] CacheNumberMalloc :
        #[ FromList :

    Returns the next object in a list.
    If the list has been exhausted we double it (like a realloc)
    If the list has not been initialized yet we start with 10 elements.
 */
```

```c

VOID *FromList(LIST *L)
{
    void *newlist;
    int i, *old, *newL;
    if ( L->num >= L->maxnum || L->lijst == 0 ) {
        if ( L->maxnum == 0 ) L->maxnum = 12;
        else if ( L->lijst ) L->maxnum *= 2;
        newlist = Malloc1(L->maxnum * L->size,L->message);
        if ( L->lijst ) {
            i = ( L->num * L->size ) / sizeof(int);
            old = (int *)L->lijst; newL = (int *)newlist;
            while ( --i >= 0 ) *newL++ = *old++;
            if ( L->lijst ) M_free(L->lijst,"L->lijst FromList");
        }
        L->lijst = newlist;
    }
    return( ((char *)(L->lijst)) + L->size * (L->num)++ );
}

/*
        #] FromList :
        #[ From0List :

        Same as FromList, but we zero excess variables.
*/

VOID *From0List(LIST *L)
{
    void *newlist;
    int i, *old, *newL;
    if ( L->num >= L->maxnum || L->lijst == 0 ) {
        if ( L->maxnum == 0 ) L->maxnum = 12;
        else if ( L->lijst ) L->maxnum *= 2;
        newlist = Malloc1(L->maxnum * L->size,L->message);
        i = ( L->num * L->size ) / sizeof(int);
        old = (int *)(L->lijst); newL = (int *)newlist;
        while ( --i >= 0 ) *newL++ = *old++;
        i = ( L->maxnum - L->num ) / sizeof(int);
        while ( --i >= 0 ) *newL++ = 0;
        if ( L->lijst ) M_free(L->lijst,"L->lijst From0List");
        L->lijst = newlist;
    }
    return( ((char *)(L->lijst)) + L->size * (L->num)++ );
}

/*
        #] From0List :
        #[ FromVarList :

    Returns the next object in a list of variables.
    If the list has been exhausted we double it (like a realloc)
    If the list has not been initialized yet we start with 10 elements.
    We allow at most MAXVARIABLES elements!
*/

VOID *FromVarList(LIST *L)
{
    void *newlist;
    int i, *old, *newL;
    if ( L->num >= L->maxnum || L->lijst == 0 ) {
        if ( L->maxnum == 0 ) L->maxnum = 12;
        else if ( L->lijst ) {
            L->maxnum *= 2;
            if ( L == &(AP.DollarList) ) {
                if ( L->maxnum > MAXDOLLARVARIABLES ) L->maxnum = MAXDOLLARVARIABLES;
                if ( L->num >= MAXDOLLARVARIABLES ) {
                    MesPrint("!!!More than %l objects in list of $-variables",
                        MAXDOLLARVARIABLES);
                    Terminate(-1);
                }
            }
            else {
                if ( L->maxnum > MAXVARIABLES ) L->maxnum = MAXVARIABLES;
                if ( L->num >= MAXVARIABLES ) {
                    MesPrint("!!!More than %l objects in list of variables",
                        MAXVARIABLES);
                    Terminate(-1);
                }
            }
        }
        newlist = Malloc1(L->maxnum * L->size,L->message);
        if ( L->lijst ) {
            i = ( L->num * L->size ) / sizeof(int);
            old = (int *)(L->lijst); newL = (int *)newlist;
            while ( --i >= 0 ) *newL++ = *old++;
            if ( L->lijst ) M_free(L->lijst,"L->lijst from VarList");
```

```
        }
        L->lijst = newlist;
    }
    return( ((char *)(L->lijst)) + L->size * ((L->num)++) );
}

/*
        #] FromVarList :
        #[ DoubleList :
*/

int DoubleList(VOID ***lijst, int *oldsize, int objectsize, char *nameoftype)
{
    VOID **newlist;
    LONG i, newsize, fullsize;
    VOID **to, **from;
    static LONG maxlistsize = (LONG)(MAXPOSITIVE);
    if ( *lijst == 0 ) {
        if ( *oldsize > 0 ) newsize = *oldsize;
        else newsize = 100;
    }
    else newsize = *oldsize * 2;
    if ( newsize > maxlistsize ) {
        if ( *oldsize == maxlistsize ) {
            MesPrint("No memory for extra space in %s",nameoftype);
            return(-1);
        }
        newsize = maxlistsize;
    }
    fullsize = ( newsize * objectsize + sizeof(VOID *)-1 ) & (-sizeof(VOID *));
    newlist = (VOID **)Malloc1(fullsize,nameoftype);
    if ( *lijst ) { /* Now some punning. DANGEROUS CODE in principle */
        to = newlist; from = *lijst; i = (*oldsize * objectsize)/sizeof(VOID *);
/*
#ifdef MALLOCDEBUG
if ( filelist ) MesPrint("    oldsize: %l, objectsize: %d, fullsize: %l"
        ,*oldsize,objectsize,fullsize);
#endif
*/
        while ( --i >= 0 ) *to++ = *from++;
    }
    if ( *lijst ) M_free(*lijst,"DoubleLList");
    *lijst = newlist;
    *oldsize = newsize;
    return(0);
/*
    int error;
    LONG lsize = *oldsize;

    maxlistsize = (LONG)(MAXPOSITIVE);
    error = DoubleLList(lijst,&lsize,objectsize,nameoftype);
    *oldsize = lsize;
    maxlistsize = (LONG)(MAXLONG);

    return(error);
*/
}

/*
        #] DoubleList :
        #[ DoubleLList :
*/

int DoubleLList(VOID ***lijst, LONG *oldsize, int objectsize, char *nameoftype)
{
    VOID **newlist;
    LONG i, newsize, fullsize;
    VOID **to, **from;
    static LONG maxlistsize = (LONG)(MAXLONG);
    if ( *lijst == 0 ) {
        if ( *oldsize > 0 ) newsize = *oldsize;
        else newsize = 100;
    }
    else newsize = *oldsize * 2;
    if ( newsize > maxlistsize ) {
        if ( *oldsize == maxlistsize ) {
            MesPrint("No memory for extra space in %s",nameoftype);
            return(-1);
        }
        newsize = maxlistsize;
    }
    fullsize = ( newsize * objectsize + sizeof(VOID *)-1 ) & (-sizeof(VOID *));
    newlist = (VOID **)Malloc1(fullsize,nameoftype);
    if ( *lijst ) { /* Now some punning. DANGEROUS CODE in principle */
        to = newlist; from = *lijst; i = (*oldsize * objectsize)/sizeof(VOID *);
/*
#ifdef MALLOCDEBUG
```

```
if ( filelist ) MesPrint("    oldsize: %l, objectsize: %d, fullsize: %l"
        ,*oldsize,objectsize,fullsize);
#endif
*/
        while ( --i >= 0 ) *to++ = *from++;
    }
    if ( *lijst ) M_free(*lijst,"DoubleLList");
    *lijst = newlist;
    *oldsize = newsize;
    return(0);
}

/*
        #] DoubleLList :
        #[ DoubleBuffer :
*/

#define DODOUBLE(x) { x *s, *t, *u; if ( *start ) { \
    oldsize = *(x **)stop - *(x **)start; newsize = 2*oldsize; \
    t = u = (x *)Malloc1(newsize*sizeof(x),text); s = *(x **)start; \
    for ( i = 0; i < oldsize; i++ ) *t++ = *s++; M_free(*start,"double"); } \
    else { newsize = 100; u = (x *)Malloc1(newsize*sizeof(x),text); } \
    *start = (void *)u; *stop = (void *)(u+newsize); }

void DoubleBuffer(void **start, void **stop, int size, char *text)
{
    LONG oldsize, newsize, i;
    if ( size == sizeof(char) ) DODOUBLE(char)
    else if ( size == sizeof(short) ) DODOUBLE(short)
    else if ( size == sizeof(int) ) DODOUBLE(int)
    else if ( size == sizeof(LONG) ) DODOUBLE(LONG)
    else if ( size % sizeof(int) == 0 ) DODOUBLE(int)
    else {
        MesPrint("---Cannot handle doubling buffers of size %d",size);
        Terminate(-1);
    }
}

/*
        #] DoubleBuffer :
        #[ ExpandBuffer :
*/

#define DOEXPAND(x) { x *newbuffer, *t, *m;                             \
    t = newbuffer = (x *)Malloc1((newsize+2)*type,"ExpandBuffer");      \
    if ( *buffer ) { m = (x *)*buffer; i = *oldsize;                    \
        while ( --i >= 0 ) *t++ = *m++; M_free(*buffer,"ExpandBuffer"); \
    } *buffer = newbuffer; *oldsize = newsize; }

void ExpandBuffer(void **buffer, LONG *oldsize, int type)
{
    LONG newsize, i;
    if ( *oldsize <= 0 ) { newsize = 100; }
    else newsize = 2*(*oldsize);
    if ( type == sizeof(char) ) DOEXPAND(char)
    else if ( type == sizeof(short) ) DOEXPAND(short)
    else if ( type == sizeof(int) ) DOEXPAND(int)
    else if ( type == sizeof(LONG) ) DOEXPAND(LONG)
    else if ( type == sizeof(POSITION) ) DOEXPAND(POSITION)
    else {
        MesPrint("---Cannot handle expanding buffers with objects of size %d",type);
        Terminate(-1);
    }
}

/*
        #] ExpandBuffer :
        #[ iexp :

        Raises the long integer y to the power p.
        Returnvalue is long, regardless of overflow.
*/

LONG iexp(LONG x, int p)
{
    int sign;
    LONG y;
    if ( x == 0 ) return(0);
    if ( p == 0 ) return(1);
    if ( x < 0 ) { sign = -1; x = -x; }
    else sign = 1;
    if ( sign < 0 && ( p & 1 ) == 0 ) sign = 1;
    if ( x == 1 ) return(sign);
    if ( p < 0 ) return(0);
    y = 1;
    while ( p ) {
        if ( ( p & 1 ) != 0 ) y *= x;
```

```
        p »= 1;
        x = x*x;
    }
    if ( sign < 0 ) y = -y;
    return(y);
}

/*
        #] iexp :
        #[ ToGeneral :

        Convert a fast argument to a general argument
        Input in r, output in m.
        If par == 0 we need the argument header also.
*/

void ToGeneral(WORD *r, WORD *m, WORD par)
{
    WORD *mm = m, j, k;
    if ( par ) m++;
    else { m[1] = 0; m += ARGHEAD + 1; }
    j = -*r++;
    k = 3;
/*      JV: Bugfix 1-feb-2016. Old code assumed FUNHEAD to be 2 */
    if ( j >= FUNCTION ) { *m++ = j; *m++ = FUNHEAD; FILLFUN(m) }
    else {
        switch ( j ) {
            case SYMBOL: *m++ = j; *m++ = 4; *m++ = *r++; *m++ = 1; break;
            case SNUMBER:
                if ( *r > 0 ) { *m++ =  *r; *m++ = 1; *m++ =  3; }
                else if ( *r == 0 ) { m--; }
                else          { *m++ = -*r; *m++ = 1; *m++ = -3; }
                goto MakeSize;
            case MINVECTOR: k = -k;
            case INDEX:
            case VECTOR: *m++ = INDEX; *m++ = 3; *m++ = *r++; break;
        }
    }
    *m++ = 1; *m++ = 1; *m++ = k;
MakeSize:
    *mm = m-mm;
    if ( !par ) mm[ARGHEAD] = *mm-ARGHEAD;
}

/*
        #] ToGeneral :
        #[ ToFast :

        Checks whether an argument can be converted to fast notation
        If this can be done it does it.
        Important: m should be allowed to be equal to r!
        Return value is 1 if conversion took place.
        If there was conversion the answer is in m.
        If there was no conversion m hasn't been touched.
*/

int ToFast(WORD *r, WORD *m)
{
    WORD i;
    if ( *r == ARGHEAD ) { *m++ = -SNUMBER; *m++ = 0; return(1); }
    if ( *r != r[ARGHEAD]+ARGHEAD ) return(0);  /* > 1 term */
    r += ARGHEAD;
    if ( *r == 4 ) {
        if ( r[2] != 1 || r[1] <= 0 ) return(0);
        *m++ = -SNUMBER; *m = ( r[3] < 0 ) ? -r[1] : r[1]; return(1);
    }
    i = *r - 1;
    if ( r[i-1] != 1 || r[i-2] != 1 ) return(0);
    if ( r[i] != 3 ) {
        if ( r[i] == -3 && r[2] == *r-4 && r[2] == 3 && r[1] == INDEX
        && r[3] < MINSPEC ) {}
        else return(0);
    }
    else if ( r[2] != *r - 4 ) return(0);
    r++;
    if ( *r >= FUNCTION ) {
        if ( r[1] <= FUNHEAD ) { *m++ = -*r; return(1); }
    }
    else if ( *r == SYMBOL ) {
        if ( r[1] == 4 && r[3] == 1 )
            { *m++ = -SYMBOL; *m++ = r[2]; return(1); }
    }
    else if ( *r == INDEX ) {
        if ( r[1] == 3 ) {
            if ( r[2] >= MINSPEC ) {
                if ( r[2] >= 0 && r[2] < AM.OffsetIndex ) *m++ = -SNUMBER;
                else *m++ = -INDEX;
```

```
            }
            else {
                if ( r[5] == -3 ) *m++ = -MINVECTOR;
                else *m++ = -VECTOR;
            }
            *m++ = r[2];
            return(1);
        }
    }
    return(0);
}

/*
        #] ToFast :
        #[ ToPolyFunGeneral :

    Routine forces a polyratfun into general notation if needed.
    If no action was needed, the return value is zero.
    A positive return value indicates how many arguments were converted.
    The new term overwrite the old.
*/

WORD ToPolyFunGeneral(PHEAD WORD *term)
{
    WORD *t = term+1, *tt, *to, *to1, *termout, *tstop, *tnext;
    WORD numarg, i, change = 0;
    tstop = term + *term; tstop -= ABS(tstop[-1]);
    termout = to = AT.WorkPointer;
    to++;
    while ( t < tstop ) { /* go through the subterms */
        if ( *t == AR.PolyFun ) {
            tt = t+FUNHEAD; tnext = t + t[1];
            numarg = 0;
            while ( tt < tnext ) { numarg++; NEXTARG(tt); }
            if ( numarg == 2 ) { /* this needs attention */
                tt = t + FUNHEAD;
                to1 = to;
                i = FUNHEAD; NCOPY(to,t,i);
                while ( tt < tnext ) { /* Do the arguments */
                    if ( *tt > 0 ) {
                        i = *tt; NCOPY(to,tt,i);
                    }
                    else if ( *tt == -SYMBOL ) {
                        to1[1] += 6+ARGHEAD; to1[2] |= MUSTCLEANPRF; change++;
                        *to++ = 8+ARGHEAD; *to++ = 0; FILLARG(to);
                        *to++ = 8; *to++ = SYMBOL; *to++ = 4; *to++ = tt[1];
                        *to++ = 1; *to++ = 1; *to++ = 1; *to++ = 3;
                        tt += 2;
                    }
                    else if ( *tt == -SNUMBER ) {
                        if ( tt[1] > 0 ) {
                            to1[1] += 2+ARGHEAD; to1[2] |= MUSTCLEANPRF; change++;
                            *to++ = 4+ARGHEAD; *to++ = 0; FILLARG(to);
                            *to++ = 4; *to++ = tt[1]; *to++ = 1; *to++ = 3;
                            tt += 2;
                        }
                        else if ( tt[1] < 0 ) {
                            to1[1] += 2+ARGHEAD; to1[2] |= MUSTCLEANPRF; change++;
                            *to++ = 4+ARGHEAD; *to++ = 0; FILLARG(to);
                            *to++ = 4; *to++ = -tt[1]; *to++ = 1; *to++ = -3;
                            tt += 2;
                        }
                        else {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Internal error: Zero in PolyRatFun");
                            MUNLOCK(ErrorMessageLock);
                            Terminate(-1);
                        }
                    }
                }
                t = tnext;
                continue;
            }
        }
        i = t[1]; NCOPY(to,t,i)
    }
    if ( change ) {
        tt = term + *term;
        while ( t < tt ) *to++ = *t++;
        *termout = to - termout;
        t = term; i = *termout; tt = termout;
        NCOPY(t,tt,i)
        AT.WorkPointer = term + *term;
    }
    return(change);
}
```

```
/*
        #] ToPolyFunGeneral :
        #[ IsLikeVector :

        Routine determines whether a function argument is like a vector.
        Returnvalue: 1: is vector or index
                     0: is not vector or index
                    -1: may be an index
*/

int IsLikeVector(WORD *arg)
{
    WORD *sstop, *t, *tstop;
    if ( *arg < 0 ) {
        if ( *arg == -VECTOR || *arg == -INDEX ) return(1);
        if ( *arg == -SNUMBER && arg[1] >= 0 && arg[1] < AM.OffsetIndex )
            return(-1);
        return(0);
    }
    sstop = arg + *arg; arg += ARGHEAD;
    while ( arg < sstop ) {
        t = arg + *arg;
        tstop = t - ABS(t[-1]);
        arg++;
        while ( arg < tstop ) {
            if ( *arg == INDEX ) return(1);
            arg += arg[1];
        }
        arg = t;
    }
    return(0);
}

/*
        #] IsLikeVector :
        #[ AreArgsEqual :
*/

int AreArgsEqual(WORD *arg1, WORD *arg2)
{
    int i;
    if ( *arg2 != *arg1 ) return(0);
    if ( *arg1 > 0 ) {
        i = *arg1;
        while ( --i > 0 ) { if ( arg1[i] != arg2[i] ) return(0); }
        return(1);
    }
    else if ( *arg1 <= -FUNCTION ) return(1);
    else if ( arg1[1] == arg2[1] ) return(1);
    return(0);
}

/*
        #] AreArgsEqual :
        #[ CompareArgs :
*/

int CompareArgs(WORD *arg1, WORD *arg2)
{
    int i1,i2;
    if ( *arg1 > 0 ) {
        if ( *arg2 < 0 ) return(-1);
        i1 = *arg1-ARGHEAD; arg1 += ARGHEAD;
        i2 = *arg2-ARGHEAD; arg2 += ARGHEAD;
        while ( i1 > 0 && i2 > 0 ) {
            if ( *arg1 != *arg2 ) return((int)(*arg1)-(int)(*arg2));
            i1--; i2--; arg1++; arg2++;
        }
        return(i1-i2);
    }
    else if ( *arg2 > 0 ) return(1);
    else {
        if ( *arg1 != *arg2 ) {
            if ( *arg1 < *arg2 ) return(-1);
            else return(1);
        }
        if ( *arg1 <= -FUNCTION ) return(0);
        return((int)(arg1[1])-(int)(arg2[1]));
    }
}

/*
        #] CompareArgs :
        #[ CompArg :

    returns 1 if arg1 comes first, -1 if arg2 comes first, 0 if equal
*/
```

```c

int CompArg(WORD *s1, WORD *s2)
{
    GETIDENTITY
    WORD *st1, *st2, x[7];
    int k;
    if ( *s1 < 0 ) {
        if ( *s2 < 0 ) {
            if ( *s1 <= -FUNCTION && *s2 <= -FUNCTION ) {
                if ( *s1 > *s2 ) return(-1);
                if ( *s1 < *s2 ) return(1);
                return(0);
            }
            if ( *s1 > *s2 ) return(1);
            if ( *s1 < *s2 ) return(-1);
            if ( *s1 <= -FUNCTION ) return(0);
            s1++; s2++;
            if ( *s1 > *s2 ) return(1);
            if ( *s1 < *s2 ) return(-1);
            return(0);
        }
        x[1] = AT.comsym[3];
        x[2] = AT.comnum[1];
        x[3] = AT.comnum[3];
        x[4] = AT.comind[3];
        x[5] = AT.comind[6];
        x[6] = AT.comfun[1];
        if ( *s1 == -SYMBOL ) {
            AT.comsym[3] = s1[1];
            st1 = AT.comsym+8; s1 = AT.comsym;
        }
        else if ( *s1 == -SNUMBER ) {
            if ( s1[1] < 0 ) {
                AT.comnum[1] = -s1[1]; AT.comnum[3] = -3;
            }
            else {
                AT.comnum[1] = s1[1]; AT.comnum[3] = 3;
            }
            st1 = AT.comnum+4;
            s1 = AT.comnum;
        }
        else if ( *s1 == -INDEX || *s1 == -VECTOR ) {
            AT.comind[3] = s1[1]; AT.comind[6] = 3;
            st1 = AT.comind+7; s1 = AT.comind;
        }
        else if ( *s1 == -MINVECTOR ) {
            AT.comind[3] = s1[1]; AT.comind[6] = -3;
            st1 = AT.comind+7; s1 = AT.comind;
        }
        else if ( *s1 <= -FUNCTION ) {
            AT.comfun[1] = -*s1;
            st1 = AT.comfun+FUNHEAD+4; s1 = AT.comfun;
        }
/*
            Symmetrize during compilation of id statement when properorder
            needs this one. Code added 10-nov-2001
*/
        else if ( *s1 == -ARGWILD ) {
             return(-1);
        }
        else { goto argerror; }
        st2 = s2 + *s2; s2 += ARGHEAD;
        goto docompare;
    }
    else if ( *s2 < 0 ) {
        x[1] = AT.comsym[3];
        x[2] = AT.comnum[1];
        x[3] = AT.comnum[3];
        x[4] = AT.comind[3];
        x[5] = AT.comind[6];
        x[6] = AT.comfun[1];
        if ( *s2 == -SYMBOL ) {
            AT.comsym[3] = s2[1];
            st2 = AT.comsym+8; s2 = AT.comsym;
        }
        else if ( *s2 == -SNUMBER ) {
            if ( s2[1] < 0 ) {
                AT.comnum[1] = -s2[1]; AT.comnum[3] = -3;
                st2 = AT.comnum+4;
            }
            else if ( s2[1] == 0 ) {
                st2 = AT.comnum+4; s2 = st2;
            }
            else {
                AT.comnum[1] = s2[1]; AT.comnum[3] = 3;
                st2 = AT.comnum+4;
            }
```

```
            s2 = AT.comnum;
        }
        else if ( *s2 == -INDEX || *s2 == -VECTOR ) {
            AT.comind[3] = s2[1]; AT.comind[6] = 3;
            st2 = AT.comind+7; s2 = AT.comind;
        }
        else if ( *s2 == -MINVECTOR ) {
            AT.comind[3] = s2[1]; AT.comind[6] = -3;
            st2 = AT.comind+7; s2 = AT.comind;
        }
        else if ( *s2 <= -FUNCTION ) {
            AT.comfun[1] = -*s2;
            st2 = AT.comfun+FUNHEAD+4; s2 = AT.comfun;
        }
/*
            Symmetrize during compilation of id statement when properorder
            needs this one. Code added 10-nov-2001
*/
        else if ( *s2 == -ARGWILD ) {
             return(1);
        }
        else { goto argerror; }
        st1 = s1 + *s1; s1 += ARGHEAD;
        goto docompare;
    }
    else {
        x[1] = AT.comsym[3];
        x[2] = AT.comnum[1];
        x[3] = AT.comnum[3];
        x[4] = AT.comind[3];
        x[5] = AT.comind[6];
        x[6] = AT.comfun[1];
        st1 = s1 + *s1; st2 = s2 + *s2;
        s1 += ARGHEAD; s2 += ARGHEAD;
docompare:
        while ( s1 < st1 && s2 < st2 ) {
            if ( ( k = CompareTerms(BHEAD s1,s2,(WORD)2) ) != 0 ) {
                AT.comsym[3] = x[1];
                AT.comnum[1] = x[2];
                AT.comnum[3] = x[3];
                AT.comind[3] = x[4];
                AT.comind[6] = x[5];
                AT.comfun[1] = x[6];
                return(-k);
            }
            s1 += *s1; s2 += *s2;
        }
        AT.comsym[3] = x[1];
        AT.comnum[1] = x[2];
        AT.comnum[3] = x[3];
        AT.comind[3] = x[4];
        AT.comind[6] = x[5];
        AT.comfun[1] = x[6];
        if ( s1 < st1 ) return(1);
        if ( s2 < st2 ) return(-1);
    }
    return(0);

argerror:
    MesPrint("Illegal type of short function argument in Normalize");
    Terminate(-1); return(0);
}

/*
        #] CompArg :
        #[ TimeWallClock :
*/

#include <sys/timeb.h>

LONG TimeWallClock(WORD par)
{
    /*
     * NOTE: this function is not thread-safe. Operations on tp are not atomic.
     */
    struct timeb tp;
    ftime(&tp);
    if ( par ) {
        return(((LONG)(tp.time)-AM.OldSecTime)*100 +
            ((LONG)(tp.millitm)-AM.OldMilliTime)/10);
    }
    else {
        AM.OldSecTime   = (LONG)(tp.time);
        AM.OldMilliTime = (LONG)(tp.millitm);
        return(0L);
    }
}
```

```

/*
        #] TimeWallClock :
        #[ TimeChildren :
*/

LONG TimeChildren(WORD par)
{
    if ( par ) return(Timer(1)-AM.OldChildTime);
    AM.OldChildTime = Timer(1);
    return(0L);
}

/*
        #] TimeChildren :
        #[ TimeCPU :
*/

LONG TimeCPU(WORD par)
{
    GETIDENTITY
    if ( par ) return(Timer(0)-AR.OldTime);
    AR.OldTime = Timer(0);
    return(0L);
}

/*
        #] TimeCPU :
        #[ Timer :
*/
#if defined(WINDOWS)

LONG Timer(int par)
{
#ifndef WITHPTHREADS
    static int initialized = 0;
    static HANDLE hProcess;
    FILETIME ftCreate, ftExit, ftKernel, ftUser;
    DUMMYUSE(par);

    if ( !initialized ) {
        hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, GetCurrentProcessId());
    }
    if ( GetProcessTimes(hProcess, &ftCreate, &ftExit, &ftKernel, &ftUser) ) {
        PFILETIME pftKernel = &ftKernel;  /* to avoid strict-aliasing rule warnings */
        PFILETIME pftUser   = &ftUser;
        __int64 t = *(__int64 *)pftKernel + *(__int64 *)pftUser;  /* in 100 nsec. */
        return (LONG)(t / 10000);  /* in msec. */
    }
    return 0;
#else
    LONG lResult = 0;
    HANDLE hThread;
    FILETIME ftCreate, ftExit, ftKernel, ftUser;
    DUMMYUSE(par);

    hThread = OpenThread(THREAD_QUERY_INFORMATION, FALSE, GetCurrentThreadId());
    if ( hThread ) {
        if ( GetThreadTimes(hThread, &ftCreate, &ftExit, &ftKernel, &ftUser) ) {
            PFILETIME pftKernel = &ftKernel;  /* to avoid strict-aliasing rule warnings */
            PFILETIME pftUser   = &ftUser;
            __int64 t = *(__int64 *)pftKernel + *(__int64 *)pftUser;  /* in 100 nsec. */
            lResult = (LONG)(t / 10000);  /* in msec. */
        }
        CloseHandle(hThread);
    }
    return lResult;
#endif
}

#elif defined(UNIX)
#include <sys/time.h>
#include <sys/resource.h>
#ifdef WITHPOSIXCLOCK
#include <time.h>
/*
    And include -lrt in the link statement (on blade02)
*/
#endif

LONG Timer(int par)
{
#ifdef WITHPOSIXCLOCK
/*
    Only to be used in combination with WITHPTHREADS
    This clock seems to be supported by the standard.
    The getrusage clock returns according to the standard only the combined
```

```
    time of the whole process. But in older versions of Linux LinuxThreads
    is used which gives a separate id to each thread and individual timings.
    In NPTL we get, according to the standard, one combined timing.
    To get individual timings we need to use
        clock_gettime(CLOCK_THREAD_CPUTIME_ID, &timing)
    with timing of the time
    struct timespec {
        time_t tv_sec;            Seconds.
        long   tv_nsec;           Nanoseconds.
    };

*/
    struct timespec t;
    if ( par == 0 ) {
        if ( clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t) ) {
            MesPrint("Error in getting timing information");
        }
        return (LONG)t.tv_sec * 1000 + (LONG)t.tv_nsec / 1000000;
    }
    return(0);
#else
    struct rusage rusage;
    if ( par == 1 ) {
        getrusage(RUSAGE_CHILDREN,&rusage);
        return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
    else {
        getrusage(RUSAGE_SELF,&rusage);
        return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
#endif
}

#elif defined(SUN)
#define _TIME_T_
#include <sys/time.h>
#include <sys/resource.h>

LONG Timer(int par)
{
    struct rusage rusage;
    if ( par == 1 ) {
        getrusage(RUSAGE_CHILDREN,&rusage);
        return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
    else {
        getrusage(RUSAGE_SELF,&rusage);
        return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
}

#elif defined(RS6K)
#include <sys/time.h>
#include <sys/resource.h>

LONG Timer(int par)
{
    struct rusage rusage;
    if ( par == 1 ) {
        getrusage(RUSAGE_CHILDREN,&rusage);
        return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
    else {
        getrusage(RUSAGE_SELF,&rusage);
        return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
}

#elif defined(ANSI)
LONG Timer(int par)
{
#ifdef ALPHA
/*  clock_t t,tikken = clock();                                  */
/*  MesPrint("ALPHA-clock = %l",(LONG)tikken);                   */
/*  t = tikken % CLOCKS_PER_SEC;                                 */
/*  tikken /= CLOCKS_PER_SEC;                                    */
/*  tikken *= 1000;                                              */
/*  tikken += (t*1000)/CLOCKS_PER_SEC;                           */
/*  return((LONG)tikken);                                        */
/* #define _TIME_T_                                              */
#include <sys/time.h>
```

```
#include <sys/resource.h>
    struct rusage rusage;
    if ( par == 1 ) {
        getrusage(RUSAGE_CHILDREN,&rusage);
        return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
    else {
        getrusage(RUSAGE_SELF,&rusage);
        return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
#else
#ifdef DEC_STATION
    clock_t tikken = clock();
    return((LONG)tikken/1000);
#else
    clock_t t, tikken = clock();
    t = tikken % CLK_TCK;
    tikken /= CLK_TCK;
    tikken *= 1000;
    tikken += (t*1000)/CLK_TCK;
    return(tikken);
#endif
#endif
}
#elif defined(VMS)

#include <time.h>
void times(tbuffer_t *buffer);

LONG
Timer(int par)
{
    tbuffer_t buffer;
    if ( par == 1 ) { return(0); }
    else {
        times(&buffer);
        return(buffer.proc_user_time * 10);
    }
}

#elif defined(mBSD)

#ifdef MICROTIME
/*
    There is only a CP time clock in microseconds here
    This can cause problems with AO.wrap around
*/
#else
#ifdef mBSD2
#include <sys/types.h>
#include <sys/times.h>
#include <time.h>
LONG pretime = 0;
#else
#define _TIME_T_
#include <sys/time.h>
#include <sys/resource.h>
#endif
#endif

LONG Timer(int par)
{
#ifdef MICROTIME
    LONG t;
    if ( par == 1 ) { return(0); }
    t = clock();
    if ( ( AO.wrapnum & 1 ) != 0 ) t ^= 0x80000000;
    if ( t < 0 ) {
        t ^= 0x80000000;
        warpnum++;
        AO.wrap += 2147584;
    }
    return(AO.wrap+(t/1000));
#else
#ifdef mBSD2
    struct tms buffer;
    LONG ret;
    ULONG a1, a2, a3, a4;
    if ( par == 1 ) { return(0); }
    times(&buffer);
    a1 = (ULONG)buffer.tms_utime;
    a2 = a1 » 16;
    a3 = a1 & 0xFFFFL;
    a3 *= 1000;
    a2 = 1000*a2 + (a3 » 16);
```

```
    a3 &= 0xFFFFL;
    a4 = a2/CLK_TCK;
    a2 %= CLK_TCK;
    a3 += a2 « 16;
    ret = (LONG)((a4 « 16) + a3 / CLK_TCK);
/*  ret = ((LONG)buffer.tms_utime * 1000)/CLK_TCK; */
    return(ret);
#else
#ifdef REALTIME
    struct timeval tp;
    struct timezone tzp;
    if ( par == 1 ) { return(0); }
    gettimeofday(&tp,&tzp); */
    return(tp.tv_sec*1000+tp.tv_usec/1000);
#else
    struct rusage rusage;
    if ( par == 1 ) {
        getrusage(RUSAGE_CHILDREN,&rusage);
        return((rusage.ru_utime.tv_sec+rusage.ru_stime.tv_sec)*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
    else {
        getrusage(RUSAGE_SELF,&rusage);
        return((rusage.ru_utime.tv_sec+rusage.ru_stime.tv_sec)*1000
              +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
    }
#endif
#endif
#endif
}

#endif

/*
        #] Timer :
        #[ Crash :

        Routine for debugging purposes
*/

int Crash()
{
    int retval;
#ifdef DEBUGGING
    int *zero = 0;
    retval = *zero;
#else
    retval = 0;
#endif
    return(retval);
}

/*
        #] Crash :
        #[ TestTerm :
*/

int TestTerm(WORD *term)
{
    int errorcode = 0, coeffsize;
    WORD *t, *tt, *tstop, *endterm, *targ, *targstop, *funstop, *argterm;
    endterm = term + *term;
    coeffsize = ABS(endterm[-1]);
    if ( coeffsize >= *term ) {
        MLOCK(ErrorMessageLock);
        MesPrint("TestTerm: Internal inconsistency in term. Coefficient too big.");
        MUNLOCK(ErrorMessageLock);
        errorcode = 1;
        goto finish;
    }
    if ( ( coeffsize < 3 ) || ( ( coeffsize & 1 ) != 1 ) ) {
        MLOCK(ErrorMessageLock);
        MesPrint("TestTerm: Internal inconsistency in term. Wrong size coefficient.");
        MUNLOCK(ErrorMessageLock);
        errorcode = 2;
        goto finish;
    }
    t = term+1;
    tstop = endterm - coeffsize;
    while ( t < tstop ) {
        switch ( *t ) {
            case SYMBOL:
            case DOTPRODUCT:
            case INDEX:
            case VECTOR:
            case DELTA:
            case HAAKJE:
```

```
                break;
            case SNUMBER:
            case LNUMBER:
                MLOCK(ErrorMessageLock);
                MesPrint("TestTerm: Internal inconsistency in term. L or S number");
                MUNLOCK(ErrorMessageLock);
                errorcode = 3;
                goto finish;
                break;
            case EXPRESSION:
            case SUBEXPRESSION:
            case DOLLAREXPRESSION:
/*
                MLOCK(ErrorMessageLock);
                MesPrint("TestTerm: Internal inconsistency in term. Expression survives.");
                MUNLOCK(ErrorMessageLock);
                errorcode = 4;
                goto finish;
*/
                break;
            case SETSET:
            case MINVECTOR:
            case SETEXP:
            case ARGFIELD:
                MLOCK(ErrorMessageLock);
                MesPrint("TestTerm: Internal inconsistency in term. Illegal subterm.");
                MUNLOCK(ErrorMessageLock);
                errorcode = 5;
                goto finish;
                break;
            case ARGWILD:
                break;
            default:
                if ( *t <= 0 ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("TestTerm: Internal inconsistency in term. Illegal subterm number.");
                    MUNLOCK(ErrorMessageLock);
                    errorcode = 6;
                    goto finish;
                }
/*
                This is a regular function.
*/
                if ( *t-FUNCTION >= NumFunctions ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("TestTerm: Internal inconsistency in term. Illegal function number");
                    MUNLOCK(ErrorMessageLock);
                    errorcode = 7;
                    goto finish;
                }
                funstop = t + t[1];
                if ( funstop > tstop ) goto subtermsize;
                if ( t[2] != 0 ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("TestTerm: Internal inconsistency in term. Dirty flag nonzero.");
                    MUNLOCK(ErrorMessageLock);
                    errorcode = 8;
                    goto finish;
                }
                targ = t + FUNHEAD;
                if ( targ > funstop ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("TestTerm: Internal inconsistency in term. Illegal function size.");
                    MUNLOCK(ErrorMessageLock);
                    errorcode = 9;
                    goto finish;
                }
                if ( functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
                }
                else {
                  while ( targ < funstop ) {
                    if ( *targ < 0 ) {
                        if ( *targ <= -(FUNCTION+NumFunctions) ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("TestTerm: Internal inconsistency in term. Illegal function number in argument.");
                            MUNLOCK(ErrorMessageLock);
                            errorcode = 10;
                            goto finish;
                        }
                        if ( *targ <= -FUNCTION ) { targ++; }
                        else {
                            if ( ( *targ != -SYMBOL ) && ( *targ != -VECTOR )
                            && ( *targ != -MINVECTOR )
                            && ( *targ != -SNUMBER )
                            && ( *targ != -ARGWILD )
                            && ( *targ != -INDEX ) ) {
```

```
                                MLOCK(ErrorMessageLock);
                                MesPrint("TestTerm: Internal inconsistency in term. Illegal object in argument.");
                                MUNLOCK(ErrorMessageLock);
                                errorcode = 11;
                                goto finish;
                            }
                            targ += 2;
                        }
                    }
                    else if ( ( *targ < ARGHEAD ) || ( targ+*targ > funstop ) ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("TestTerm: Internal inconsistency in term. Illegal size of argument.");
                        MUNLOCK(ErrorMessageLock);
                        errorcode = 12;
                        goto finish;
                    }
                    else if ( targ[1] != 0 ) {
                        MLOCK(ErrorMessageLock);
                        MesPrint("TestTerm: Internal inconsistency in term. Dirty flag in argument.");
                        MUNLOCK(ErrorMessageLock);
                        errorcode = 13;
                        goto finish;
                    }
                    else {
                        targstop = targ + *targ;
                        argterm = targ + ARGHEAD;
                        while ( argterm < targstop ) {
                            if ( ( *argterm < 4 ) || ( argterm + *argterm > targstop ) ) {
                                MLOCK(ErrorMessageLock);
                                MesPrint("TestTerm: Internal inconsistency in term. Illegal termsize in argument.");
                                MUNLOCK(ErrorMessageLock);
                                errorcode = 14;
                                goto finish;
                            }
                            if ( TestTerm(argterm) != 0 ) {
                                MLOCK(ErrorMessageLock);
                                MesPrint("TestTerm: Internal inconsistency in term. Called from TestTerm.");
                                MUNLOCK(ErrorMessageLock);
                                errorcode = 15;
                                goto finish;
                            }
                            argterm += *argterm;
                        }
                        targ = targstop;
                    }
                }
            }
            break;
        }
        tt = t + t[1];
        if ( tt > tstop ) {
subtermsize:
            MLOCK(ErrorMessageLock);
            MesPrint("TestTerm: Internal inconsistency in term. Illegal subterm size.");
            MUNLOCK(ErrorMessageLock);
            errorcode = 100;
            goto finish;
        }
        t = tt;
    }
    return(errorcode);
finish:
    return(errorcode);
}

/*
        #] TestTerm :
    #] Mixed :
*/
```

4.120 transform.c File Reference

`#include "form3.h"`
Include dependency graph for transform.c:

Functions

- int CoTransform (UBYTE ∗in)
- WORD RunTransform (PHEAD WORD ∗term, WORD ∗params)
- WORD RunEncode (PHEAD WORD ∗fun, WORD ∗args, WORD ∗info)
- WORD RunDecode (PHEAD WORD ∗fun, WORD ∗args, WORD ∗info)
- WORD RunReplace (PHEAD WORD ∗fun, WORD ∗args, WORD ∗info)
- WORD RunImplode (WORD ∗fun, WORD ∗args)
- WORD RunExplode (PHEAD WORD ∗fun, WORD ∗args)
- WORD RunPermute (PHEAD WORD ∗fun, WORD ∗args, WORD ∗info)
- WORD RunReverse (PHEAD WORD ∗fun, WORD ∗args)
- WORD RunDedup (PHEAD WORD ∗fun, WORD ∗args)
- WORD RunCycle (PHEAD WORD ∗fun, WORD ∗args, WORD ∗info)
- WORD RunAddArg (PHEAD WORD ∗fun, WORD ∗args)
- WORD RunMulArg (PHEAD WORD ∗fun, WORD ∗args)
- WORD RunIsLyndon (PHEAD WORD ∗fun, WORD ∗args, int par)
- WORD RunToLyndon (PHEAD WORD ∗fun, WORD ∗args, int par)
- WORD RunDropArg (PHEAD WORD ∗fun, WORD ∗args)
- WORD RunSelectArg (PHEAD WORD ∗fun, WORD ∗args)
- int TestArgNum (int n, int totarg, WORD ∗args)
- WORD PutArgInScratch (WORD ∗arg, UWORD ∗scrat)
- UBYTE ∗ ReadRange (UBYTE ∗s, WORD ∗out, int par)
- int FindRange (PHEAD WORD ∗args, WORD ∗arg1, WORD ∗arg2, WORD totarg)

4.120.1 Detailed Description

Routines that deal with the transform statement and its varieties.

Definition in file transform.c.

4.120.2 Function Documentation

4.120.2.1 CoTransform()

```
int CoTransform (
            UBYTE * in )
```

Definition at line 87 of file transform.c.

4.120.2.2 RunTransform()

```
WORD RunTransform (
            PHEAD WORD * term,
            WORD * params )
```

Definition at line 689 of file transform.c.

4.120.2.3 RunEncode()

```
WORD RunEncode (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 922 of file transform.c.

4.120.2.4 RunDecode()

```
WORD RunDecode (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 1109 of file transform.c.

4.120.2.5 RunReplace()

```
WORD RunReplace (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 1279 of file transform.c.

4.120.2.6 RunImplode()

```
WORD RunImplode (
            WORD * fun,
            WORD * args )
```

Definition at line 1757 of file transform.c.

4.120.2.7 RunExplode()

```
WORD RunExplode (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 1958 of file transform.c.

4.120.2.8 RunPermute()

```
WORD RunPermute (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 2072 of file transform.c.

4.120.2.9 RunReverse()

```
WORD RunReverse (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 2299 of file transform.c.

4.120.2.10 RunDedup()

```
WORD RunDedup (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 2386 of file transform.c.

4.120.2.11 RunCycle()

```
WORD RunCycle (
            PHEAD WORD * fun,
            WORD * args,
            WORD * info )
```

Definition at line 2475 of file transform.c.

4.120.2.12 RunAddArg()

```
WORD RunAddArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 2627 of file transform.c.

4.120.2.13 RunMulArg()

```
WORD RunMulArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 2710 of file transform.c.

4.120.2.14 RunIsLyndon()

```
WORD RunIsLyndon (
            PHEAD WORD * fun,
            WORD * args,
            int par )
```

Definition at line 2849 of file transform.c.

4.120.2.15 RunToLyndon()

```
WORD RunToLyndon (
            PHEAD WORD * fun,
            WORD * args,
            int par )
```

Definition at line 2927 of file transform.c.

4.120.2.16 RunDropArg()

```
WORD RunDropArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 3039 of file transform.c.

4.120.2.17 RunSelectArg()

```
WORD RunSelectArg (
            PHEAD WORD * fun,
            WORD * args )
```

Definition at line 3065 of file transform.c.

4.120.2.18 TestArgNum()

```
int TestArgNum (
            int n,
            int totarg,
            WORD * args )
```

Definition at line 3103 of file transform.c.

4.120.2.19 PutArgInScratch()

```
WORD PutArgInScratch (
            WORD * arg,
            UWORD * scrat )
```

Definition at line 3172 of file transform.c.

4.120.2.20 ReadRange()

```
UBYTE * ReadRange (
            UBYTE * s,
            WORD * out,
            int par )
```

Definition at line 3208 of file transform.c.

4.120.2.21 FindRange()

```
int FindRange (
            PHEAD WORD * args,
            WORD * arg1,
            WORD * arg2,
            WORD totarg )
```

Definition at line 3368 of file transform.c.

4.121 transform.c

Go to the documentation of this file.

```
00001
00005 /* #[ License : */
00006 /*
00007  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00008  *   When using this file you are requested to refer to the publication
00009  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  *   This is considered a matter of courtesy as the development was paid
00011  *   for by FOM the Dutch physics granting agency and we would like to
00012  *   be able to track its scientific use to convince FOM of its value
00013  *   for the community.
00014  *
00015  *   This file is part of FORM.
00016  *
00017  *   FORM is free software: you can redistribute it and/or modify it under the
00018  *   terms of the GNU General Public License as published by the Free Software
00019  *   Foundation, either version 3 of the License, or (at your option) any later
00020  *   version.
00021  *
00022  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00025  *   details.
00026  *
00027  *   You should have received a copy of the GNU General Public License along
00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030 /* #] License : */
00031 /*
00032     #[ Includes : transform.c
00033 */
00034
00035 #include "form3.h"
00036
00037 /*
00038     #] Includes :
00039     #[ Transform :
00040         #[ Intro :
00041
00042         Here are the routines for the transform statement. This is a
00043         group of transformations on function arguments or groups of
00044         function arguments. The purpose of this command is that it
00045         avoids repetitive pattern matching.
00046         Syntax:
00047             Transform,SetOfFunctions,OneOrMoreTransformations;
00048         Each transformation is given by
00049             Replace(argfirst,arglast)=(,,)
00050             Encode(argfirst,arglast):base=#
```

```
            Decode(argfirst,arglast):base=#
            Implode(argfirst,arglast)
            Explode(argfirst,arglast)
            Permute(cycle)(cycle)(cycle)...(cycle)
            Reverse(argfirst,arglast)
            Dedup(argfirst,arglast)
            Cycle(argfirst,arglast)=+/-num
            IsLyndon(argfirst,arglast)=(yes,no)
            ToLyndon(argfirst,arglast)=(yes,no)
        In replace the extra information is
            a replace_() without the name of the replace_ function.
            This can be as in (0,1,1,0) or (xarg_,1-xarg_) to indicate
            a symbolic argument or (x,y,y,x) to exchange x and y, etc.
        In Encode and Decode argfirst is the most significant 'word' and
        arglast is the least significant 'word'.
        Note that we need to introduce the generic symbolic arguments xarg_,
        parg_, iarg_ and farg_.
        Examples:
            Transform,{H,E}
                    ,Replace(1:'WEIGHT')=(0,1,1,0)
                    ,Encode(1:'WEIGHT')=base(2);
            Transform,{H,E}
                    ,Decode(1:'WEIGHT')=base(3)
                    ,Replace(1:'WEIGHT')=(2,-1,1,0,0,1);
        Others that can be added:
            symmetrize?

        6-may-2016: Changed MAXPOSITIVE2 into MAXPOSITIVE4. This makes room
                    for the use of dollar variables as arguments.

        #] Intro :
        #[ CoTransform :
*/

static WORD tranarray[10] = { SUBEXPRESSION, SUBEXPSIZE, 0, 1, 0, 0, 0, 0, 0, 0 };

int CoTransform(UBYTE *in)
{
    GETIDENTITY
    UBYTE *s = in, c, *ss, *Tempbuf;
    WORD number, type, num, i, *work = AT.WorkPointer+2, *wp, range[2], one = 1;
    WORD numdol, *wstart;
    int error = 0, irhs;
    LONG x;
    while ( *in == ',' ) in++;
    num = 0; wp = work + 1;
/*
    #[ Sets :

    First the set specification(s). No sets means all functions (dangerous!)
*/
    for(;;) {
        if ( *in == '{' ) {
            s = in+1;
            SKIPBRA2(in)
            number = DoTempSet(s,in);
            in++;
            if ( *in != ',' ) {
                c = in[1]; in[1] = 0;
                MesPrint("& %s: A set in a transform statement should be followed by a comma",s);
                in[1] = c; in++;
                if ( error == 0 ) error = 1;
            }
        }
        else if ( *in == '[' || FG.cTable[*in] == 0 ) {
            s = in;
            in = SkipAName(in);
            if ( *in != ',' ) break;
            c = *in; *in = 0;
            type = GetName(AC.varnames,s,&number,NOAUTO);
            if ( type == CFUNCTION ) { number += MAXVARIABLES + FUNCTION; }
            else if ( type != CSET ) {
                MesPrint("& %s: A transform statement starts with sets of functions",s);
                if ( error == 0 ) error = 1;
            }
            *in++ = c;
        }
        else {
            MesPrint("&Illegal syntax in Transform statement",s);
            if ( error == 0 ) error = 1;
            return(error);
        }
        if ( number >= 0 ) {
          if ( number < MAXVARIABLES ) {
/*
            Check that this is a set of functions
*/
```

```
            if ( Sets[number].type != CFUNCTION ) {
                MesPrint("&A set in a transform statement should be a set of functions");
                if ( error == 0 ) error = 1;
            }
          }
        }
        else if ( error == 0 ) error = 1;
/*
        Now write the number to the right place
*/
        *wp++ = number;
        num++;
        while ( *in == ',' ) in++;
    }
    *work = wp - work;
    work = wp; wp++;
/*
    #] Sets :

    Now we should loop over the various transformations
*/
    while ( *s ) {
        in = s;
        if ( FG.cTable[*in] != 0 ) {
            MesPrint("&Illegal character in Transform statement");
            if ( error == 0 ) error = 1;
            return(error);
        }
        in = SkipAName(in);
        if ( *in == '>' || *in == '<' ) in++;
        ss = in;
        c = *ss; *ss = 0;
        if ( c != '(' ) {
            MesPrint("&Illegal syntax in specifying a transformation inside a Transform statement");
            if ( error == 0 ) error = 1;
            return(error);
        }
/*
        #[ replace :
*/
        if ( StrICmp(s,(UBYTE *)"replace") == 0 ) {
/*
                Subkeys: (,,) as in replace_(,,)
            The idea here is to read the subkeys as the argument
            of a replace_ function.
            We put the whole together as in the multiply statement (which
            could just be a replace_(....)) and compile it.
            Then we expand the tree with Generator and check the complete
            expression for legality.
*/
            type = REPLACEARG;
doreplace:
            *ss = c;
            if ( ( in = ReadRange(in,range,0) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            in++;
/*
            We have replace(#,#)=(...), and we want dum_(...)  (DUMFUN)
            to send to the compiler. The pointer is after the '=';
*/
            s = in;
            if ( *s != '(' ) {
                MesPrint("&");
                if ( error == 0 ) error = 1;
                return(error);
            }
            SKIPBRA3(in);
            if ( *in != ')' ) {
                MesPrint("&");
                if ( error == 0 ) error = 1;
                return(error);
            }
            in++;
            if ( *in != ',' && *in != '\0' ) {
                MesPrint("&");
                if ( error == 0 ) error = 1;
                return(error);
            }
            i = in - s;
            ss = Tempbuf = (UBYTE *)Malloc1(i+5,"CoTransform/replace");
            *ss++ = 'd'; *ss++ = 'u'; *ss++ = 'm'; *ss++ = '_';
            NCOPY(ss,s,i)
            *ss++ = 0;
            AC.ProtoType = tranarray;
            tranarray[4] = AC.cbufnum;
```

```
            irhs = CompileAlgebra(Tempbuf,RHSIDE,AC.ProtoType);
            M_free(Tempbuf,"CoTransform/replace");
            if ( irhs < 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            tranarray[2] = irhs;
/*
            The result of the compilation goes through Generator during
            execution, because that takes care of $-variables.
            This is why we could not use replace_ and had to use dum_.
*/
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *wp++ = SUBEXPSIZE+4;
            for ( i = 0; i < SUBEXPSIZE; i++ ) *wp++ = tranarray[i];
            *wp++ = 1;
            *wp++ = 1;
            *wp++ = 3;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] replace :
        #[ encode/decode :
*/
        else if ( StrICmp(s,(UBYTE *)"decode" ) == 0 ) {
            type = DECODEARG;
            goto doencode;
        }
        else if ( StrICmp(s,(UBYTE *)"encode" ) == 0 ) {
            type = ENCODEARG;
doencode:   *ss = c;
            if ( ( in = ReadRange(in,range,2) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            in++;
            s = in; while ( FG.cTable[*in] == 0 ) in++;
            c = *in; *in = 0;
/*
                Subkeys: base=# or base=$var
*/
            if ( StrICmp(s,(UBYTE *)"base") == 0 ) {
                *in = c;
                if ( *in != '=' ) {
                    MesPrint("&Illegal base specification in encode/decode transformation");
                    if ( error == 0 ) error = 1;
                    return(error);
                }
                in++;
                if ( *in == '$' ) {
                    in++; ss = in;
                    in = SkipAName(in);
                    c = *in; *in = 0;
                    if ( GetName(AC.dollarnames,ss,&numdol,NOAUTO) != CDOLLAR ) {
                        MesPrint("&%s is undefined",ss-1);
                        numdol = AddDollar(ss,DOLINDEX,&one,1);
                        return(1);
                    }
                    *in = c;
                    x = -numdol;
                }
                else {
                    x = 0;
                    while ( FG.cTable[*in] == 1 ) {
                        x = 10*x + *in++ - '0';
                        if ( x > MAXPOSITIVE4 ) {
illsize:                    MesPrint("&Illegal value for base in encode/decode transformation");
                            if ( error == 0 ) error = 1;
                            return(error);
                        }
                    }
                    if ( x <= 1 ) goto illsize;
                }
                if ( *in != ',' && *in != '\0' ) {
                    MesPrint("&Illegal termination of transformation");
                    if ( error == 0 ) error = 1;
                    return(error);
                }
            }
            else {
                MesPrint("&Illegal option in encode/decode transformation");
                if ( error == 0 ) error = 1;
```

```
                return(error);
            }
/*
            Now we can put the whole statement together
            We have the set(s) in work up to wp and the range in range.
            The base is in x and the type tells whether it is encode or decode.
*/
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *wp++ = 4;
            *wp++ = BASECODE;
            *wp++ = (WORD)x;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] encode/decode :
        #[ implode :
*/
        else if ( StrICmp(s,(UBYTE *)"implode") == 0
                || StrICmp(s,(UBYTE *)"tosumnotation") == 0 ) {
/*
                Subkeys: ?
*/
            type = IMPLODEARG;
            *ss = c;
            if ( ( in = ReadRange(in,range,1) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] implode :
        #[ explode :
*/
        else if ( StrICmp(s,(UBYTE *)"explode") == 0
                || StrICmp(s,(UBYTE *)"tointegralnotation") == 0 ) {
/*
                Subkeys: ?
*/
            type = EXPLODEARG;
            *ss = c;
            if ( ( in = ReadRange(in,range,1) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] explode :
        #[ permute :
*/
        else if ( StrICmp(s,(UBYTE *)"permute") == 0 ) {
            type = PERMUTEARG;
            *ss = c;
            *wp++ = ARGRANGE;
            *wp++ = 1;
            *wp++ = MAXPOSITIVE4;
            *wp++ = type;
/*
            Now a sequence of cycles
*/
            do {
              wstart = wp; wp++;
              do {
                in++;
                if ( *in == '$' ) {
                    WORD number; UBYTE *t;
                    in++; t = in;
                    while ( FG.cTable[*in] < 2 ) in++;
                    c = *in; *in = 0;
```

```
                    if ( ( number = GetDollar(t) ) < 0 ) {
                        MesPrint("&Undefined variable $%s",t);
                        if ( !error ) error = 1;
                        number = AddDollar(t,0,0,0);
                    }
                    *in = c;
                    *wp++ = -number-1;
                }
                else {
                  x = 0;
                  while ( FG.cTable[*in] == 1 ) {
                    x = 10*x + *in++ - '0';
                    if ( x > MAXPOSITIVE4 ) {
                        MesPrint("&value in permute transformation too large");
                        if ( error == 0 ) error = 1;
                        return(error);
                    }
                  }
                  if ( x == 0 ) {
                    MesPrint("&value 0 in permute transformation not allowed");
                    if ( error == 0 ) error = 1;
                    return(error);
                  }
                  *wp++ = (WORD)x-1;
                }
              } while ( *in == ',' );
              if ( *in != ')' ) {
                MesPrint("&Illegal syntax in permute transformation");
                if ( error == 0 ) error = 1;
                return(error);
              }
              in++;
              if ( *in != ',' && *in != '(' && *in != '\0' ) {
                MesPrint("&Illegal ending in permute transformation");
                if ( error == 0 ) error = 1;
                return(error);
              }
              *wstart = wp-wstart;
              if ( *wstart == 1 ) wstart--;
            } while ( *in == '(' );
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] permute :
        #[ reverse :
*/
        else if ( StrICmp(s,(UBYTE *)"reverse") == 0 ) {
            type = REVERSEARG;
            *ss = c;
            if ( ( in = ReadRange(in,range,1) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] reverse :
        #[ dedup :
*/
        else if ( StrICmp(s,(UBYTE *)"dedup") == 0 ) {
            type = DEDUPARG;
            *ss = c;
            if ( ( in = ReadRange(in,range,1) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] dedup :
        #[ cycle :
*/
        else if ( StrICmp(s,(UBYTE *)"cycle") == 0 ) {
```

```
            type = CYCLEARG;
            *ss = c;
            if ( ( in = ReadRange(in,range,0) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
/*
            Now a sequence of cycles
*/
            in++;
            if ( *in == '+' ) {
            }
            else if ( *in == '-' ) {
                one = -1;
            }
            else {
                MesPrint("&Cycle in a Transform statement should be followed by =+/-number/$");
                if ( error == 0 ) error = 1;
                return(error);
            }
            in++; x = 0;
            if ( *in == '$' ) {
                UBYTE *si = in;
                in++; si = in;
                while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
                c = *in; *in = 0;
                if ( ( x = GetDollar(si) ) < 0 ) {
                    MesPrint("&Undefined $-variable in transform,cycle statement.");
                    error = 1;
                }
                *in = c;
                if ( one < 0 ) x += MAXPOSITIVE4;
                x += MAXPOSITIVE2;
                *wp++ = x;
            }
            else {
              while ( FG.cTable[*in] == 1 ) {
                x = 10*x + *in++ - '0';
                if ( x > MAXPOSITIVE4 ) {
                    MesPrint("&Number in cycle in a Transform statement too big");
                    if ( error == 0 ) error = 1;
                    return(error);
                }
              }
              *wp++ = x*one;
            }
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] cycle :
        #[ islyndon/tolyndon :
*/
        else if ( StrICmp(s,(UBYTE *)"islyndon" ) == 0 ) {
            type = ISLYNDON;
            goto doreplace;
        }
        else if ( StrICmp(s,(UBYTE *)"islyndon<" ) == 0 ) {
            type = ISLYNDON;
            goto doreplace;
        }
        else if ( StrICmp(s,(UBYTE *)"islyndon+" ) == 0 ) {
            type = ISLYNDON;
            goto doreplace;
        }
        else if ( StrICmp(s,(UBYTE *)"islyndon>" ) == 0 ) {
            type = ISLYNDONR;
            goto doreplace;
        }
        else if ( StrICmp(s,(UBYTE *)"islyndon-" ) == 0 ) {
            type = ISLYNDONR;
            goto doreplace;
        }
        else if ( StrICmp(s,(UBYTE *)"tolyndon" ) == 0 ) {
            type = TOLYNDON;
            goto doreplace;
        }
        else if ( StrICmp(s,(UBYTE *)"tolyndon<" ) == 0 ) {
            type = TOLYNDON;
            goto doreplace;
        }
        else if ( StrICmp(s,(UBYTE *)"tolyndon+" ) == 0 ) {
```

```
            type = TOLYNDON;
            goto doreplace;
        }
        else if ( StrICmp(s,(UBYTE *)"tolyndon>" ) == 0 ) {
            type = TOLYNDONR;
            goto doreplace;
        }
        else if ( StrICmp(s,(UBYTE *)"tolyndon-" ) == 0 ) {
            type = TOLYNDONR;
            goto doreplace;
        }
/*
        #] islyndon/tolyndon :
        #[ addarg :
*/
        else if ( StrICmp(s,(UBYTE *)"addargs" ) == 0 ) {
            type = ADDARG;
            *ss = c;
            if ( ( in = ReadRange(in,range,1) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] addarg :
        #[ mularg :
*/
        else if ( ( StrICmp(s,(UBYTE *)"mulargs" ) == 0 )
                || ( StrICmp(s,(UBYTE *)"multiplyargs" ) == 0 ) ) {
            type = MULTIPLYARG;
            *ss = c;
            if ( ( in = ReadRange(in,range,1) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] mularg :
        #[ droparg :
*/
        else if ( StrICmp(s,(UBYTE *)"dropargs" ) == 0 ) {
            type = DROPARG;
            *ss = c;
            if ( ( in = ReadRange(in,range,1) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
        }
/*
        #] droparg :
        #[ selectarg :
*/
        else if ( StrICmp(s,(UBYTE *)"selectargs" ) == 0 ) {
            type = SELECTARG;
            *ss = c;
            if ( ( in = ReadRange(in,range,1) ) == 0 ) {
                if ( error == 0 ) error = 1;
                return(error);
            }
            *wp++ = ARGRANGE;
            *wp++ = range[0];
            *wp++ = range[1];
            *wp++ = type;
            *work = wp-work;
            work = wp; *wp++ = 0;
            s = in;
```

```
        }
/*
        #] selectarg :
*/
        else {
            MesPrint("&Unknown transformation inside a Transform statement: %s",s);
            *ss = c;
            if ( error == 0 ) error = 1;
            return(error);
        }
        while ( *s == ',') s++;
    }
    AT.WorkPointer[0] = TYPETRANSFORM;
    AT.WorkPointer[1] = i = wp - AT.WorkPointer;
    AddNtoL(i,AT.WorkPointer);
    return(error);
}

/*
        #] CoTransform :
        #[ RunTransform :

        Executes the transform statement.
        This routine hunts down the functions and sends them to the various
        action routines.
        params: size,#set1,...,#setn, transformations

*/

WORD RunTransform(PHEAD WORD *term, WORD *params)
{
    WORD *t, *tstop, *w, *m, *out, *in, *tt, retval;
    WORD *fun, *args, *info, *infoend, *onetransform, *funs, *endfun;
    WORD *thearg = 0, *iterm, *newterm, *nt, *oldwork = AT.WorkPointer;
    int i;
    out = tstop = term + *term;
    tstop -= ABS(tstop[-1]);
    in = term;
    t = term + 1;
    while ( t < tstop ) {
        endfun = onetransform = params + *params;
        funs = params + 1;
        if ( *t < FUNCTION ) {}
        else if ( funs == endfun ) {  /* we do all functions */
hit:;
            while ( in < t ) *out++ = *in++;
            tt = t + t[1]; fun = out;
            while ( in < tt ) *out++ = *in++;
            do {
                args = onetransform + 1;
                info = args; while ( *info <= MAXRANGEINDICATOR ) {
                    if ( *info == ALLARGS ) info++;
                    else if ( *info == NUMARG ) info += 2;
                    else if ( *info == ARGRANGE ) info += 3;
                    else if ( *info == MAKEARGS ) info += 3;
                }
                switch ( *info ) {
                    case REPLACEARG:
                        if ( RunReplace(BHEAD fun,args,info) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case ENCODEARG:
                        if ( RunEncode(BHEAD fun,args,info) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case DECODEARG:
                        if ( RunDecode(BHEAD fun,args,info) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case IMPLODEARG:
                        if ( RunImplode(fun,args) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case EXPLODEARG:
                        if ( RunExplode(BHEAD fun,args) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case PERMUTEARG:
                        if ( RunPermute(BHEAD fun,args,info) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case REVERSEARG:
                        if ( RunReverse(BHEAD fun,args) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case DEDUPARG:
                        if ( RunDedup(BHEAD fun,args) ) goto abo;
```

```
                        out = fun + fun[1];
                        break;
                    case CYCLEARG:
                        if ( RunCycle(BHEAD fun,args,info) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case ADDARG:
                        if ( RunAddArg(BHEAD fun,args) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case MULTIPLYARG:
                        if ( RunMulArg(BHEAD fun,args) ) goto abo;
                        out = fun + fun[1];
                        break;
                    case ISLYNDON:
                        if ( ( retval = RunIsLyndon(BHEAD fun,args,1) ) < -1 ) goto abo;
                        goto returnvalues;
                        break;
                    case ISLYNDONR:
                        if ( ( retval = RunIsLyndon(BHEAD fun,args,-1) ) < -1 ) goto abo;
                        goto returnvalues;
                        break;
                    case TOLYNDON:
                        if ( ( retval = RunToLyndon(BHEAD fun,args,1) ) < -1 ) goto abo;
                        goto returnvalues;
                        break;
                    case TOLYNDONR:
                        if ( ( retval = RunToLyndon(BHEAD fun,args,-1) ) < -1 ) goto abo;
returnvalues:;
                        out = fun + fun[1];
                        if ( retval == -1 ) break;
/*
                        Work out the yes/no stuff
*/
                        AT.WorkPointer += 2*AM.MaxTer;
                        if ( AT.WorkPointer > AT.WorkTop ) {
                            MLOCK(ErrorMessageLock);
                            MesWork();
                            MUNLOCK(ErrorMessageLock);
                            return(-1);
                        }
                        iterm = AT.WorkPointer;
                        info++;
                        for ( i = 0; i < *info; i++ ) iterm[i] = info[i];
                        AT.WorkPointer = iterm + *iterm;
                        AR.Eside = LHSIDEX;
                        NewSort(BHEAD0);
                        if ( Generator(BHEAD iterm,AR.Cnumlhs) ) {
                            LowerSortLevel();
                            AT.WorkPointer = oldwork;
                            return(-1);
                        }
                        newterm = AT.WorkPointer;
                        if ( EndSort(BHEAD newterm,0) < 0 ) {}
                        if ( ( *newterm && *(newterm+*newterm) != 0 ) || *newterm == 0 ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("&yes/no information in islyndon/tolyndon does not evaluate into
    a single term");
                            MUNLOCK(ErrorMessageLock);
                            return(-1);
                        }
                        AR.Eside = RHSIDE;
                        i = *newterm; tt = iterm; nt = newterm;
                        NCOPY(tt,nt,i);
                        AT.WorkPointer = iterm + *iterm;
                        info = iterm + 1;
                        infoend = info+info[1];
                        info += FUNHEAD;

                        if ( retval == 0 ) {
/*
                            Need second argument (=no)
*/
                            if ( info >= infoend ) {
abortlyndon:;
                                MLOCK(ErrorMessageLock);
                                MesPrint("There should be a yes and a no argument in
    islyndon/tolyndon");
                                MUNLOCK(ErrorMessageLock);
                                Terminate(-1);
                            }
                            NEXTARG(info)
                            if ( info >= infoend ) goto abortlyndon;
                            thearg = info;
                        }
                        else if ( retval == 1 ) {
/*
```

```
                                Need first argument (=yes)
*/
                                if ( info >= infoend ) goto abortlyndon;
                                thearg = info;
                                NEXTARG(info)
                                if ( info >= infoend ) goto abortlyndon;
                            }
                            NEXTARG(info)
                            if ( info < infoend ) goto abortlyndon;
/*
                            The argument in thearg needs to be copied
                            We did not pull it through generator to guarantee
                            that it is a single argument.
                            The easiest way is to let the routine Normalize
                            do the job and put everything in an exponent function
                            with the power one.
*/
                            if ( *thearg == -SNUMBER && thearg[1] == 0 ) {
                                *term = 0; return(0);
                            }
                            if ( *thearg == -SNUMBER && thearg[1] == 1 ) { }
                            else {
                                fun = out;
                                *out++ = EXPONENT; out++; *out++ = 1; FILLFUN3(out);
                                COPY1ARG(out,thearg);
                                *out++ = -SNUMBER; *out++ = 1;
                                fun[1] = out-fun;
                            }
                            break;
                        case DROPARG:
                            if ( RunDropArg(BHEAD fun,args) ) goto abo;
                            out = fun + fun[1];
                            break;
                        case SELECTARG:
                            if ( RunSelectArg(BHEAD fun,args) ) goto abo;
                            out = fun + fun[1];
                            break;
                        default:
                            MLOCK(ErrorMessageLock);
                            MesPrint("Irregular code in execution of transform statement");
                            MUNLOCK(ErrorMessageLock);
                            Terminate(-1);
                    }
                    onetransform += *onetransform;
                } while ( *onetransform );
            }
            else {
              while ( funs < endfun ) {  /* sum over sets */
                if ( *funs > MAXVARIABLES ) {
                    if ( *t == *funs-MAXVARIABLES ) goto hit;
                }
                else {
                  w = SetElements + Sets[*funs].first;
                  m = SetElements + Sets[*funs].last;
                  while ( w < m ) {  /* sum over set elements */
                    if ( *w == *t ) goto hit;
                    w++;
                  }
                }
                funs++;
              }
            }
            t += t[1];
        }
        tt = term + *term; while ( in < tt ) *out++ = *in++;
        *tt = i = out - tt;
/*
        Now copy the whole thing back
*/
        NCOPY(term,tt,i)
        return(0);
abo:
        MLOCK(ErrorMessageLock);
        MesCall("RunTransform");
        MUNLOCK(ErrorMessageLock);
        return(-1);
}

/*
            #] RunTransform :
            #[ RunEncode :

            The info is given by
                ENCODEARG,size,BASECODE,num
            and possibly more codes to follow.
            Only one range is allowed and for now, it should be fully numerical
            If the range is in reverse order, we need to either revert it
```

```
        first or work with an array of pointers.
*/

WORD RunEncode(PHEAD WORD *fun, WORD *args, WORD *info)
{
    WORD base, *f, *funstop, *fun1, *t, size1, size2, size3, *arg;
    int num, num1, num2, n, i, i1, i2;
    UWORD *scrat1, *scrat2, *scrat3;
    WORD *tt, *tstop, totarg, arg1, arg2;
    if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunEncode");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( tt < tstop ) { totarg++; NEXTARG(tt); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
    if ( arg1 > totarg || arg2 > totarg ) return(0);

    if ( info[2] == BASECODE ) {
        base = info[3];
        if ( base <= 0 ) { /* is a dollar variable */
            i1 = -base;
            base = DolToNumber(BHEAD i1);
            if ( AN.ErrorInDollar || base < 2 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("$%s does not have a number value > 1 in base/encode/transform statement in module %l",
                    DOLLARNAME(Dollars,i1),AC.CModule);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
        }
/*
        Compute number of pointers needed and make sure there is space
*/
        if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; }
        else { num1 = arg1; num2 = arg2; }
        num = num2-num1+1;
        WantAddPointers(num);
/*
        Collect the pointers in pWorkSpace
*/
        n = 1; funstop = fun+fun[1]; f = fun+FUNHEAD;
        while ( n < num1 ) {
            if ( f >= funstop ) return(0);
            NEXTARG(f);
            n++;
        }
        fun1 = f; i = 0;
        while ( n <= num2 ) {
            if ( f >= funstop ) return(0);
            if ( *f != -SNUMBER ) {
                if ( *f < 0 ) return(0);
                t = f + *f - 1;
                i1 = ABS(*t);
                if ( (*f-i1) != (ARGHEAD+1) ) return(0); /* Not numerical */
                i1 = (i1-1)/2 - 1;
                t--;
                while ( i1 > 0 ) {
                    if ( *t != 0 ) return(0); /* Not an integer */
                    t--; i1--;
                }
            }
            AT.pWorkSpace[AT.pWorkPointer+i] = f;
            i++;
            NEXTARG(f);
            n++;
        }
/*
        f points now to after the arguments; fun1 at the first.
        Now check whether we need to revert the order
*/
        if ( arg1 > arg2 ) {
            i1 = 0; i2 = i-1;
            while ( i1 < i2 ) {
                t = AT.pWorkSpace[AT.pWorkPointer+i1];
                AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
                AT.pWorkSpace[AT.pWorkPointer+i2] = t;
                i1++; i2--;
            }
        }
/*
        Now we can put the thing together.
        x = arg1;
```

```
        x = base*x+arg2
        x = base*x+arg3  etc.
        We need three scratch arrays for long integers
                (see NumberMalloc in tools.c).
*/
        scrat1 = NumberMalloc("RunEncode");
        scrat2 = NumberMalloc("RunEncode");
        scrat3 = NumberMalloc("RunEncode");
        arg = AT.pWorkSpace[AT.pWorkPointer];
        size1 = PutArgInScratch(arg,scrat1);
        i--;
        while ( i > 0 ) {
            if ( MulLong(scrat1,size1,(UWORD *)(&base),1,scrat2,&size2) ) {
                NumberFree(scrat3,"RunEncode");
                NumberFree(scrat2,"RunEncode");
                NumberFree(scrat1,"RunEncode");
                goto CalledFrom;
            }
            NEXTARG(arg);
            size3 = PutArgInScratch(arg,scrat3);
            if ( AddLong(scrat2,size2,scrat3,size3,scrat1,&size1) ) {
                NumberFree(scrat3,"RunEncode");
                NumberFree(scrat2,"RunEncode");
                NumberFree(scrat1,"RunEncode");
                goto CalledFrom;
            }
            i--;
        }
/*
        Now put the output in place. There are two cases, one being much
        faster than the other. Hence we program both.
        Fast: it fits inside the old location.
        Slow: it does not.
        The total space is f-fun1
*/
        if ( size1 == 0 ) { /* Fits! */
            *fun1++ = -SNUMBER; *fun1++ = 0;
            while ( f < funstop ) *fun1++ = *f++;
            fun[1] = funstop-fun;
        }
        else if ( size1 == 1 && scrat1[0] <= MAXPOSITIVE ) { /* Fits! */
            *fun1++ = -SNUMBER; *fun1++ = scrat1[0];
            while ( f < funstop ) *fun1++ = *f++;
            fun[1] = fun1-fun;
        }
        else if ( size1 == -1 && scrat1[0] <= MAXPOSITIVE+1 ) { /* Fits! */
            *fun1++ = -SNUMBER;
            if ( scrat1[0] < MAXPOSITIVE ) *fun1++ = scrat1[0];
            else *fun1++ = (WORD)(MAXPOSITIVE+1);
            while ( f < funstop ) *fun1++ = *f++;
            fun[1] = fun1-fun;
        }
        else if ( ABS(size1)*2+2+ARGHEAD <= f-fun1 ) { /* Fits! */
            if ( size1 < 0 ) { size2 = size1*2-1; size1 = -size1; size3 = -size2; }
            else { size2 = 2*size1+1; size3 = size2; }
            *fun1++ = size3+ARGHEAD+1;
            *fun1++ = 0; FILLARG(fun1);
            *fun1++ = size3+1;
            for ( i = 0; i < size1; i++ ) *fun1++ = scrat1[i];
            *fun1++ = 1;
            for ( i = 1; i < size1; i++ ) *fun1++ = 0;
            *fun1++ = size2;
            while ( f < funstop ) *fun1++ = *f++;
            fun[1] = fun1-fun;
        }
        else {  /* Does not fit */
            t = funstop;
            if ( size1 < 0 ) { size2 = size1*2-1; size1 = -size1; size3 = -size2; }
            else { size2 = 2*size1+1; size3 = size2; }
            *t++ = size3+ARGHEAD+1;
            *t++ = 0; FILLARG(t);
            *t++ = size3+1;
            for ( i = 0; i < size1; i++ ) *t++ = scrat1[i];
            *t++ = 1;
            for ( i = 1; i < size1; i++ ) *t++ = 0;
            *t++ = size2;
            while ( f < funstop ) *t++ = *f++;
            f = funstop;
            while ( f < t ) *fun1++ = *f++;
            fun[1] = fun1-fun;
        }
        NumberFree(scrat3,"RunEncode");
        NumberFree(scrat2,"RunEncode");
        NumberFree(scrat1,"RunEncode");
    }
    else {
        MLOCK(ErrorMessageLock);
```

```
        MesPrint("Unimplemented type of encoding encountered in RunEncode");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    return(0);
CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("RunEncode");
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
        #] RunEncode :
        #[ RunDecode :
*/

WORD RunDecode(PHEAD WORD *fun, WORD *args, WORD *info)
{
    WORD base, num, num1, num2, n, *f, *funstop, *fun1, size1, size2, size3, *t;
    WORD i1, i2, i, sig;
    UWORD *scrat1, *scrat2, *scrat3;
    WORD *tt, *tstop, totarg, arg1, arg2;
    if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunDecode");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( tt < tstop ) { totarg++; NEXTARG(tt); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
    if ( arg1 > totarg && arg2 > totarg ) return(0);
    if ( info[2] == BASECODE ) {
        base = info[3];
        if ( base <= 0 ) { /* is a dollar variable */
            i1 = -base;
            base = DolToNumber(BHEAD i1);
            if ( AN.ErrorInDollar || base < 2 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("$%s does not have a number value > 1 in base/decode/transform statement in module %l",
                    DOLLARNAME(Dollars,i1),AC.CModule);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
            }
        }
/*
        Compute number of output arguments needed
*/
        if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; }
        else { num1 = arg1; num2 = arg2; }
        num = num2-num1+1;
        if ( num <= 1 ) return(0);
/*
        Find argument num1
*/
        funstop = fun + fun[1];
        f = fun + FUNHEAD; n = 1;
        while ( f < funstop ) {
            if ( n == num1 ) break;
            NEXTARG(f); n++;
        }
        if ( f >= funstop ) return(0);  /* not enough arguments */
/*
        Check that f is integer
*/
        if ( *f == -SNUMBER ) {}
        else if ( *f < 0 ) return(0);
        else {
            t = f + *f - 1;
            i1 = ABS(*t);
            if ( (*f-i1) != (ARGHEAD+1) ) return(0); /* Not numerical */
            i1 = (i1-1)/2 - 1;
            t--;
            while ( i1 > 0 ) {
                if ( *t != 0 ) return(0); /* Not an integer */
                t--; i1--;
            }
        }
        fun1 = f;
/*
        The argument that should be decoded is in fun1
        We have to copy it to scratch
*/
        scrat1 = NumberMalloc("RunEncode");
```

```
        scrat2 = NumberMalloc("RunEncode");
        scrat3 = NumberMalloc("RunEncode");
        size1 = PutArgInScratch(fun1,scrat1);
        if ( size1 < 0 ) { sig = -1; size1 = -size1; }
        else sig = 1;
/*
        We can check first whether this number can be decoded
*/
        scrat2[0] = base; size2 = 1;
        if ( RaisPow(BHEAD scrat2,&size2,num) ) {
            NumberFree(scrat3,"RunEncode");
            NumberFree(scrat2,"RunEncode");
            NumberFree(scrat1,"RunEncode");
            goto CalledFrom;
        }
        if ( BigLong(scrat1,size1,scrat2,size2) >= 0 ) { /* Number too big */
            NumberFree(scrat3,"RunEncode");
            NumberFree(scrat2,"RunEncode");
            NumberFree(scrat1,"RunEncode");
            return(0);
        }
/*
        We need num*2 spaces
*/
        if ( *fun1 > num*2 ) {  /* shrink space */
            t = fun1 + 2*num; f = fun1 + *fun1;
            while ( f < funstop ) *t++ = *f++;
            fun[1] = t - fun;
        }
        else if ( *fun1 < num*2 ) { /* case includes -SNUMBER */
            if ( *fun1 < 0 ) { /* expand space from -SNUMBER */
                fun[1] += (num-1)*2;
                t = funstop + (num-1)*2;
            }
            else { /* expand space from general argument */
                fun[1] += 2*num - *fun1;
                t = funstop +2*num - *fun1;
            }
            f = funstop;
            while ( f > fun1 ) *--t = *--f;
        }
/*
        Now there is space for num -SNUMBER arguments filled from the top.
*/
        for ( i = num-1; i >= 0; i-- ) {
            DivLong(scrat1,size1,(UWORD *)(&base),1,scrat2,&size2,scrat3,&size3);
            fun1[2*i]   = -SNUMBER;
            if ( size3 == 0 ) fun1[2*i+1] = 0;
            else fun1[2*i+1] = (WORD)(scrat3[0])*sig;
            for ( i1 = 0; i1 < size2; i1++ ) scrat1[i1] = scrat2[i1];
            size1 = size2;
        }
        if ( size2 != 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("RunDecode: number to be decoded is too big");
            MUNLOCK(ErrorMessageLock);
            NumberFree(scrat3,"RunEncode");
            NumberFree(scrat2,"RunEncode");
            NumberFree(scrat1,"RunEncode");
            goto CalledFrom;
        }
/*
        Now check whether we should change the order of the arguments
*/
        if ( arg1 > arg2 ) {
            i1 = 1; i2 = 2*num-1;
            while ( i2 > i1 ) {
                i = fun1[i1]; fun1[i1] = fun1[i2]; fun1[i2] = i;
                i1 += 2; i2 -= 2;
            }
        }
        NumberFree(scrat3,"RunEncode");
        NumberFree(scrat2,"RunEncode");
        NumberFree(scrat1,"RunEncode");
    }
    else {
        MLOCK(ErrorMessageLock);
        MesPrint("Unimplemented type of encoding encountered in RunDecode");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    return(0);
CalledFrom:
    MLOCK(ErrorMessageLock);
    MesCall("RunDecode");
    MUNLOCK(ErrorMessageLock);
    return(-1);
```

```
}

/*
        #] RunDecode :
        #[ RunReplace :

        Gets the function, passes the arguments and looks whether they
        need to be treated. If so, the exact treatment is found in info.
        The info is given as if it is a function of type REPLACEMENT but
        its name is REPLACEARG (which is NOT a function).
        It is performed on the arguments.
        The output is at first written after fun and in the end overwrites fun.
*/

WORD RunReplace(PHEAD WORD *fun, WORD *args, WORD *info)
{
    int n = 0, i, dirty = 0, totarg, nfix, nwild, ngeneral;
    WORD *t, *tt, *u, *tstop, *info1, *infoend, *oldwork = AT.WorkPointer;
    WORD *term, *newterm, *nt, *term1, *term2;
    WORD wild[4], mask, *term3, *term4, *oldmask = AT.WildMask;
    WORD n1, n2, doanyway;
    info++;
    t = fun; tstop = fun + fun[1]; u = tstop;
    for ( i = 0; i < FUNHEAD; i++ ) *u++ = *t++;
    tt = t;
    if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
        totarg = 0;
        while ( tt < tstop ) { totarg++; NEXTARG(tt); }
    }
    else {
        totarg = tstop - tt;
    }
/*
    Now get the info through Generator to bring it to standard form.
    info points at a single term that should be sent to Generator.

    We want to put the information in the WorkSpace but fun etc lies there
    already. This means that we have to move the WorkPointer quite high up.
*/
    AT.WorkPointer += 2*AM.MaxTer;
    if ( AT.WorkPointer > AT.WorkTop ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    term = AT.WorkPointer;
    for ( i = 0; i < *info; i++ ) term[i] = info[i];
    AT.WorkPointer = term + *term;
    AR.Eside = LHSIDEX;
    NewSort(BHEAD0);
    if ( Generator(BHEAD term,AR.Cnumlhs) ) {
        LowerSortLevel();
        AT.WorkPointer = oldwork;
        return(-1);
    }
    newterm = AT.WorkPointer;
    if ( EndSort(BHEAD newterm,0) < 0 ) {}
    if ( ( *newterm && *(newterm+*newterm) != 0 ) || *newterm == 0 ) {
        MLOCK(ErrorMessageLock);
        MesPrint("&information in replace transformation does not evaluate into a single term");
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    AR.Eside = RHSIDE;
    i = *newterm; tt = term; nt = newterm;
    NCOPY(tt,nt,i);
    AT.WorkPointer = term + *term;
    info = term + 1;

    term1 = term + *term;
    term2 = term1+1;
    *term2++ = REPLACEMENT;
    term2++; FILLFUN(term2)
/*
    First we count the different types of objects
*/
    infoend = info + info[1];
    info1 = info + FUNHEAD;
    nfix = nwild = ngeneral = 0;
    while ( info1 < infoend ) {
        if ( *info1 == -SNUMBER ) {
            nfix++;
            info1 += 2; NEXTARG(info1)
        }
        else if ( *info1 <= -FUNCTION ) {
            if ( *info1 == -WILDARGFUN ) {
```

```
                nwild++;
                info1++; NEXTARG(info1)
            }
            else {
                *term2++ = *info1++; COPY1ARG(term2,info1)
                ngeneral++;
            }
        }
        else if ( *info1 == -INDEX ) {
            if ( info1[1] == WILDARGINDEX + AM.OffsetIndex ) {
                nwild++;
                info1 += 2; NEXTARG(info1)
            }
            else {
                *term2++ = *info1++; *term2++ = *info1++; COPY1ARG(term2,info1)
                ngeneral++;
            }
        }
        else if ( *info1 == -SYMBOL ) {
            if ( info1[1] == WILDARGSYMBOL ) {
                nwild++;
                info1 += 2; NEXTARG(info1)
            }
            else {
                *term2++ = *info1++; *term2++ = *info1++; COPY1ARG(term2,info1)
                ngeneral++;
            }
        }
        else if ( *info1 == -MINVECTOR || *info1 == -VECTOR ) {
            if ( info1[1] == WILDARGVECTOR + AM.OffsetVector ) {
                nwild++;
                info1 += 2; NEXTARG(info1)
            }
            else {
                *term2++ = *info1++; *term2++ = *info1++; COPY1ARG(term2,info1)
                ngeneral++;
            }
        }
        else {
            MLOCK(ErrorMessageLock);
            MesPrint("&irregular code found in replace transformation (RunReplace)");
            MUNLOCK(ErrorMessageLock);
            Terminate(-1);
        }
    }
    AT.WorkPointer = term2;
    *term1 = term2 - term1;
    term1[2] = *term1 - 1;
/*
    And now stepping through the arguments
*/
    while ( t < tstop ) {
        n++;    /* The number of the argument. Now check whether we need it */
        if ( TestArgNum(n,totarg,args) == 0 ) {
            if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
                if ( *t <= -FUNCTION ) { *u++ = *t++; }
                else if ( *t < 0 ) { *u++ = *t++; *u++ = *t++; }
                else { i = *t; NCOPY(u,t,i) }
            }
            else *u++ = *t++;
            continue;
        }
/*
        Here we have in info effectively a replace_ function, but with
        additionally the possibility of integer arguments. We treat those first
        and for the rest we have to do some pattern matching.
        Note that the compilation routine should check that there is an
        even number of arguments in the replace function.

        First we go for number -> something
*/
        doanyway = 0;
        if ( nfix > 0 ) {
          if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
            if ( *t == -SNUMBER ) {
              info1 = info + FUNHEAD;
              while ( info1 < infoend ) {
                if ( *info1 == -SNUMBER ) {
                    if ( info1[1] == t[1] ) {
                      if ( info1[2] == -SNUMBER ) {
                        *u++ = -SNUMBER; *u++ = info1[3];
                        info1 += 4;
                      }
                      else {
                        info1 += 2;
                        if ( info1[0] <= -FUNCTION ) i = 1;
                        else if ( info1[0] < 0 ) i = 2;
```

```
                        else i = *info1;
                        NCOPY(u,info1,i)
                      }
                      t += 2; goto nextt;
                    }
                    info1 += 2;
                    NEXTARG(info1);
                }
                else {
                    NEXTARG(info1);
                    NEXTARG(info1);
                }
              }
/*
              Here we had no match in the style of 1->2. It could however
              be that xarg_ does something
*/
              doanyway = 1; n2 = t[1];
            }
          }
          else {  /* Tensor */
            if ( *t < AM.OffsetIndex && *t >= 0 ) {
              info1 = info + FUNHEAD;
              while ( info1 < infoend ) {
                if ( ( *info1 == -SNUMBER ) && ( info1[1] == *t )
                 && ( ( ( info1[2] == -SNUMBER ) && ( info1[3] >= 0 )
                 && ( info1[3] < AM.OffsetIndex ) )
                 || ( info1[2] == -INDEX || info1[2] == -VECTOR
                 || info1[2] == -MINVECTOR ) ) ) {
                    *u++ = info1[3];
                    info1 += 4;
                    t++; goto nextt;
                }
                else {
                    NEXTARG(info1);
                    NEXTARG(info1);
                }
              }
            }
          }
        }
        else if ( *t == -SNUMBER ) {
          doanyway = 1; n2 = t[1];
        }
/*
        First we try to catch those elements that have an exact match
        in the traditional replace_ part.
        This means that *t should be less than zero and match an entry
        in the replace_ function that we prepared.
*/
        if ( ngeneral > 0 ) {
          if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
            if ( *t < 0 ) {
                term3 = term1 + *term1;
                term4 = term1 + FUNHEAD;
                while ( term4 < term3 ) {
                    if ( *term4 == *t && ( *t <= -FUNCTION ||
                    ( t[1] == term4[1] ) ) ) break;
                    NEXTARG(term4)
                }
                if ( term4 < term3 ) goto dothisnow;
            }
          }
          else {
            term3 = term1 + *term1;
            term4 = term1 + FUNHEAD;
            while ( term4 < term3 ) {
                if ( ( term4[1] == *t ) &&
                     ( ( *term4 == -INDEX || *term4 == -VECTOR ||
                      ( *term4 == -SYMBOL && term4[1] < AM.OffsetIndex
                         && term4[1] >= 0 ) ) ) ) break;
                NEXTARG(term4)
            }
            if ( term4 < term3 ) goto dothisnow;
          }
        }
/*
        First we eliminate the fixed arguments and make a 'new info'
        If there is anything left we can continue.
        Now we look for whole argument wildcards (arg_, parg_, iarg_ or farg_)
*/
        if ( nwild > 0 ) {
/*
            If we have f(a)*replace_(xarg_,b(xarg_)) this gives f(b(a))
            In testing the wildcard we have CheckWild do the work.
            This means that we have to set op the special variables
            (AT.WildMask,AN.WildValue,AN.NumWild)
```

```

*/
            wild[1] = 4;
            info1 = info + FUNHEAD;
            while ( info1 < infoend ) {
                if ( *info1 == -SYMBOL && info1[1] == WILDARGSYMBOL
                && ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) ) {
                    wild[0] = SYMTOSUB;
                    wild[2] = WILDARGSYMBOL;
                    wild[3] = 0;
                    AN.WildValue = wild;
                    AT.WildMask = &mask;
                    mask = 0;
                    AN.NumWild = 1;
                    if ( *t == -SYMBOL || ( *t > 0 && CheckWild(BHEAD WILDARGSYMBOL,SYMTOSUB,1,t) == 0 )
                    || doanyway ) {
/*
                        We put the part in replace in a function and make
                        a replace_(xarg_,(t argument)).
*/
                        n1 = SYMBOL; n2 = WILDARGSYMBOL;
                        info1 += 2;
getthisone:;
                        term3 = term2+1;
                        if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
                            *term3++ = DUMFUN; term3++; FILLFUN(term3)
                            COPY1ARG(term3,info1)
                        }
                        else {
                            *term3++ = fun[0]; term3++; FILLFUN(term3)
                            *term3++ = *info1;
                        }
                        term2[2] = term3 - term2 - 1;
                        tt = term3;
                        *term3++ = REPLACEMENT;
                        term3++; FILLFUN(term3)
                        *term3++ = -n1;
                        if ( n1 < FUNCTION ) *term3++ = n2;
                        if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
                            term4 = t;
                            COPY1ARG(term3,term4)
                        }
                        else {
                            *term3++ = *t;
                        }
                        tt[1] = term3 - tt;
                        *term3++ = 1; *term3++ = 1; *term3++ = 3;
                        *term2 = term3 - term2;

                        AT.WorkPointer = term3;
                        NewSort(BHEAD0);
                        if ( Generator(BHEAD term2,AR.Cnumlhs) ) {
                            LowerSortLevel();
                            AT.WorkPointer = oldwork;
                            AT.WildMask = oldmask;
                            return(-1);
                        }
                        term4 = AT.WorkPointer;
                        if ( EndSort(BHEAD term4,0) < 0 ) {}
                        if ( ( *term4 && *(term4+*term4) != 0 ) || *term4 == 0 ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("&information in replace transformation does not evaluate into a single term");
                            MUNLOCK(ErrorMessageLock);
                            return(-1);
                        }
/*
                        Now we can copy the new function argument to the output u
*/
                        i = term4[2]-FUNHEAD;
                        term3 = term4+FUNHEAD+1;
                        NCOPY(u,term3,i)
                        if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
                            NEXTARG(t)
                        }
                        else t++;
                        AT.WorkPointer = term2;

                        goto nextt;
                    }
                    info1 += 2; NEXTARG(info1)
                }
                else if ( ( *info1 == -INDEX )
                    && ( info[1] == WILDARGINDEX + AM.OffsetIndex ) ) {
                    wild[0] = INDTOSUB;
                    wild[2] = WILDARGINDEX+AM.OffsetIndex;
```

```
                    wild[3] = 0;
                    AN.WildValue = wild;
                    AT.WildMask = &mask;
                    mask = 0;
                    AN.NumWild = 1;
                    if ( ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION )
                    || ( *t == -INDEX || ( *t > 0 && CheckWild(BHEAD WILDARGINDEX,INDTOSUB,1,t) == 0 )
    ) ) {
/*
                        We put the part in replace in a function and make
                        a replace_(xarg_,(t argument)).
*/
                        n1 = INDEX; n2 = WILDARGINDEX+AM.OffsetIndex;
                        info1 += 2;
                        goto getthisone;
                    }
                    info1 += 2; NEXTARG(info1)
                }
                else if ( ( *info1 == -VECTOR )
                    && ( info1[1] == WILDARGVECTOR + AM.OffsetVector ) ) {
                    wild[0] = VECTOSUB;
                    wild[2] = WILDARGVECTOR+AM.OffsetVector;
                    wild[3] = 0;
                    AN.WildValue = wild;
                    AT.WildMask = &mask;
                    mask = 0;
                    AN.NumWild = 1;
                    if ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION ) {
                        if ( *t < MINSPEC ) {
                            n1 = VECTOR; n2 = WILDARGVECTOR+AM.OffsetVector;
                            info1 += 2;
                            goto getthisone;
                        }
                    }
                    else if ( *t == -VECTOR || *t == -MINVECTOR ||
                    ( *t > 0 && CheckWild(BHEAD WILDARGVECTOR,VECTOSUB,1,t) == 0 ) ) {
/*
                        We put the part in replace in a function and make
                        a replace_(xarg_,(t argument)).
*/
                        n1 = VECTOR; n2 = WILDARGVECTOR+AM.OffsetVector;
                        info1 += 2;
                        goto getthisone;
                    }
                    info1 += 2; NEXTARG(info1)
                }
                else if ( *info1 == -WILDARGFUN ) {
                    wild[0] = FUNTOFUN;
                    wild[2] = WILDARGFUN;
                    wild[3] = 0;
                    AN.WildValue = wild;
                    AT.WildMask = &mask;
                    mask = 0;
                    AN.NumWild = 1;
                    if ( *t <= -FUNCTION || ( *t > 0 && CheckWild(BHEAD WILDARGFUN,FUNTOFUN,1,t) == 0
    ) ) {
/*
                        We put the part in replace in a function and make
                        a replace_(xarg_,(t argument)).
*/
                        n2 = n1 = -WILDARGFUN; /* n2 is to keep the compiler quiet */
                        info1++;
                        goto getthisone;
                    }
                    info1++; NEXTARG(info1)
                }
                else {
                    NEXTARG(info1) NEXTARG(info1)
                }
            }
        }
        if ( ngeneral > 0 ) {
/*
            They are all in a replace_ function.
            Compose the whole thing into a term with replace_()*dum_(arg)
            which will be given to Generator.
            If we have f(a(x))*replace_(x,b) this gives f(a(b))
*/
dothisnow:;
            term3 = term2; term4 = term1; i = *term1;
            NCOPY(term3,term4,i)
            term4 = term3;
            if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
                *term3++ = DUMFUN; term3++; FILLFUN(term3);
                tt = t;
                COPY1ARG(term3,tt)
            }
```

```
            else {
                *term3++ = fun[0]; term3++; FILLFUN(term3); *term3++ = *t;
            }
            term4[1] = term3-term4;
            *term3++ = 1; *term3++ = 1; *term3++ = 3;
            *term2 = term3-term2;
            AT.WorkPointer = term3;
            NewSort(BHEAD0);
            if ( Generator(BHEAD term2,AR.Cnumlhs) ) {
                LowerSortLevel();
                AT.WorkPointer = oldwork;
                AT.WildMask = oldmask;
                return(-1);
            }
            term4 = AT.WorkPointer;
            if ( EndSort(BHEAD term4,0) < 0 ) {}
            if ( ( *term4 && *(term4+*term4) != 0 ) || *term4 == 0 ) {
                MLOCK(ErrorMessageLock);
                MesPrint("&information in replace transformation does not evaluate into a single term");
                MUNLOCK(ErrorMessageLock);
                return(-1);
            }
/*
            Now we can copy the new function argument to the output u
*/
            i = term4[2]-FUNHEAD;
            term3 = term4+FUNHEAD+1;
            NCOPY(u,term3,i)
            NEXTARG(t)
            AT.WorkPointer = term2;

            goto nextt;
        }

/*
        No catch. Copy the argument and continue.
*/
        if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
            if ( *t <= -FUNCTION ) { *u++ = *t++; }
            else if ( *t < 0 ) { *u++ = *t++; *u++ = *t++; }
            else { i = *t; NCOPY(u,t,i) }
        }
        else {
            *u++ = *t++;
        }
nextt:;
    }
    i = u - tstop; tstop[1] = i; tstop[2] = dirty;
    t = fun; u = tstop; NCOPY(t,u,i)
    AT.WorkPointer = oldwork;
    AT.WildMask = oldmask;
    return(0);
}

/*
        #] RunReplace :
        #[ RunImplode :

        Note that we restrict ourselves to short integers and/or single symbols
*/

WORD RunImplode(WORD *fun, WORD *args)
{
    GETIDENTITY
    WORD *tt, *tstop, totarg, arg1, arg2, num1, num2, i, i1, n;
    WORD *f, *t, *ttt, *t4, *ff, *fff;
    WORD moveup, numzero, outspace;
    if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunImplode");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( tt < tstop ) { totarg++; NEXTARG(tt); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
/*
    Get the proper range in forward direction and the number of arguments
*/
    if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; }
    else { num1 = arg1; num2 = arg2; }
    if ( num1 > totarg || num2 > totarg ) return(0);
/*
    We need, for the most general case 4 spots for each:
            x,pow,coef,sign
```

```
    Hence we put these in the workspace above the term after tstop
*/
    n = 1; f = fun+FUNHEAD;
    while ( n < num1 ) {
        if ( f >= tstop ) return(0);
        NEXTARG(f);
        n++;
    }
    ff = f;
/*
    We are now at the first argument to be done
    Go through the terms and test their validity.
    If one of them doesn't conform to the rules we don't do anything.
    The terms to be done are put in special notation after the function.
    Notation: numsymbol, power, |coef|, sign
    If numsymbol is negative there is no symbol.
    We do it this way because otherwise stepping backwards (as in range=(4,1))
    would be very difficult.
*/
    tt = tstop; i = 0;
    while ( n <= num2 ) {
        if ( f >= tstop ) return(0);
        if ( *f == -SNUMBER ) { *tt++ = -1; *tt++ = 0;
            if ( f[1] < 0 ) { *tt++ = -f[1]; *tt++ = -1; }
            else { *tt++ = f[1]; *tt++ = 1; }
            f += 2;
        }
        else if ( *f == -SYMBOL ) { *tt++ = f[1]; *tt++ = 1; *tt++ = 1; *tt++ = 1; f += 2; }
        else if ( *f < 0 ) return(0);
        else {
            if ( *f != ( f[ARGHEAD]+ARGHEAD ) ) return(0); /* Not a single term */
            t = f + *f - 1;
            i1 = ABS(*t);
            if ( ( i1 > 3 ) || ( t[-1] != 1 ) ) return(0); /* Not an integer or too big */
            if ( (UWORD)(t[-2]) > MAXPOSITIVE4 ) return(0); /* number too big */
            if ( f[ARGHEAD] == i1+1 ) { /* numerical which is fine */
                *tt++ = -1; *tt++ = 0; *tt++ = t[-2];
                if ( *t < 0 ) { *tt++ = -1; }
                else { *tt++ = 1; }
            }
            else if ( ( f[ARGHEAD+1] != SYMBOL )
                 || ( f[ARGHEAD+2] != 4 )
                 || ( ( f+ARGHEAD+1+f[ARGHEAD+2] ) < ( t-i1 ) ) ) return(0);
                    /* not a single symbol with a coefficient */
            else {
                *tt++ = f[ARGHEAD+3];
                *tt++ = f[ARGHEAD+4];
                *tt++ = t[-2];
                if ( *t < 0 ) { *tt++ = -1; }
                else { *tt++ = 1; }
            }
            f += *f;
        }
        i++; n++;
    }
    fff = f;
/*
    At this point we can do the implosion.
    Requirement: no coefficient shall take more than one word.
    (a stricter requirement may be needed to keep the explosion contained)
*/
    if ( arg1 > arg2 ) {
/*
        Work backward.
*/
        t = tt - 4; numzero = 0;
        while ( t >= tstop ) {
            if ( t[2] == 0 ) numzero++;
            else {
                if ( numzero > 0 ) {
                    t[2] += numzero;
                    t4 = t+4;
                    ttt = t4 + 4*numzero;
                    while ( ttt < tt ) *t4++ = *ttt++;
                    tt -= 4*numzero;
                    numzero = 0;
                }
            }
            t -= 4;
        }
    }
    else {
        t = tstop;
        numzero = 0; ttt = t;
        while ( t < tt ) {
            if ( t[2] == 0 ) numzero++;
            else {
```

```
                if ( numzero > 0 ) {
                    t[2] += numzero;
                    t4 = t;
                    while ( t4 < tt ) *ttt++ = *t4++;
                    tt -= 4*numzero;
                    t -= 4*numzero;
                    ttt = t + 4;
                    numzero = 0;
                }
                else {
                    ttt = t + 4;
                }
            }
            t += 4;
        }
/*
        We may have numzero > 0 at the end. We leave them.
        Output space is currently from tstop to tt
*/
    }
/*
    Now we compute the real output space needed
*/
    t = tstop; outspace = 0;
    while ( t < tt ) {
        if ( t[0] == -1 ) {
            if ( t[2] > MAXPOSITIVE4 ) { return(0); /* Number too big */ }
            outspace += 2;
        }
        else if ( t[1] == 1 && t[2] == 1 && t[3] == 1 ) { outspace += 2; }
        else { outspace += 8 + ARGHEAD; }
        t += 4;
    }
    if ( outspace < (fff-ff) ) {
        t = tstop;
        while ( t < tt ) {
            if ( t[0] == -1 ) { *ff++ = -SNUMBER; *ff++ = t[2]*t[3]; }
            else if ( t[1] == 1 && t[2] == 1 && t[3] == 1 ) {
                *ff++ = -SYMBOL; *ff++ = t[0];
            }
            else {
                *ff++ = 8+ARGHEAD; *ff++ = 0; FILLARG(ff);
                *ff++ = 8; *ff++ = SYMBOL; *ff++ = 4; *ff++ = t[0]; *ff++ = t[1];
                *ff++ = t[2]; *ff++ = 1; *ff++ = t[3] > 0 ? 3: -3;
            }
            t += 4;
        }
        while ( fff < tstop ) *ff++ = *fff++;
        fun[1] = ff - fun;
    }
    else if ( outspace > (fff-ff) ) {
/*
        Move the answer up by the required amount.
        Move the tail to its new location
        Move in things as for outspace == (fff-ff)
*/
        moveup = outspace-(fff-ff);
        ttt = tt + moveup;
        t = tt;
        while ( t > fff ) *--ttt = *--t;
        tt += moveup; tstop += moveup;
        fff += moveup;
        fun[1] += moveup;
        goto moveinto;
    }
    else {
moveinto:
        t = tstop;
        while ( t < tt ) {
            if ( t[0] == -1 ) { *ff++ = -SNUMBER; *ff++ = t[2]*t[3]; }
            else if ( t[1] == 1 && t[2] == 1 && t[3] == 1 ) {
                *ff++ = -SYMBOL; *ff++ = t[0];
            }
            else {
                *ff++ = 8+ARGHEAD; *ff++ = 0; FILLARG(ff);
                *ff++ = 8; *ff++ = SYMBOL; *ff++ = 4; *ff++ = t[0]; *ff++ = t[1];
                *ff++ = t[2]; *ff++ = 1; *ff++ = t[3] > 0 ? 3: -3;
            }
            t += 4;
        }
    }
    return(0);
}

/*
        #] RunImplode :
        #[ RunExplode :
```

```
*/

WORD RunExplode(PHEAD WORD *fun, WORD *args)
{
    WORD arg1, arg2, num1, num2, *tt, *tstop, totarg, *tonew, *newfun;
    WORD *ff, *f;
    int reverse = 0, iarg, i, numzero;
    if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunExplode");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( tt < tstop ) { totarg++; NEXTARG(tt); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
/*
    Get the proper range in forward direction and the number of arguments
*/
    if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; reverse = 1; }
    else { num1 = arg1; num2 = arg2; }
    if ( num1 > totarg || num2 > totarg ) return(0);
    if ( tstop + AM.MaxTer > AT.WorkTop ) goto OverWork;
/*
    We will make the new function after the old one in the workspace
    Find the first argument
*/
    tonew = newfun = tstop;
    ff = fun + FUNHEAD; iarg = 0;
    while ( ff < tstop ) {
        iarg++;
        if ( iarg == num1 ) {
            i = ff - fun; f = fun;
            NCOPY(tonew,f,i)
            break;
        }
        NEXTARG(ff)
    }
/*
    We have reached the first argument to be done
*/
    while ( iarg <= num2 ) {
        if ( *ff == -SYMBOL || ( *ff == -SNUMBER && ff[1] == 0 ) )
            { *tonew++ = *ff++; *tonew++ = *ff++; }
        else if ( *ff == -SNUMBER ) {
            numzero = ABS(ff[1])-1;
            if ( reverse ) {
                *tonew++ = -SNUMBER; *tonew++ = ff[1] < 0 ? -1: 1;
                while ( numzero > 0 ) {
                    *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
                }
            }
            else {
                while ( numzero > 0 ) {
                    *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
                }
                *tonew++ = -SNUMBER; *tonew++ = ff[1] < 0 ? -1: 1;
            }
            ff += 2;
        }
        else if ( *ff < 0 ) { return(0); }
        else {
            if ( *ff != ARGHEAD+8 || ff[ARGHEAD] != 8
                 || ff[ARGHEAD+1] != SYMBOL || ABS(ff[ARGHEAD+7]) != 3
                 || ff[ARGHEAD+6] != 1 ) return(0);
            numzero = ff[ARGHEAD+5];
            if ( numzero >= MAXPOSITIVE4 ) return(0);
            numzero--;
            if ( reverse ) {
                if ( ff[ARGHEAD+7] > 0 ) { *tonew++ = -SNUMBER; *tonew++ = 1; }
                else {
                    *tonew++ = ARGHEAD+8; *tonew++ = 0; FILLARG(tonew)
                    *tonew++ = 8; *tonew++ = SYMBOL; *tonew++ = ff[ARGHEAD+3];
                    *tonew++ = ff[ARGHEAD+4]; *tonew++ = 1; *tonew++ = 1;
                    *tonew++ = -3;
                }
                while ( numzero > 0 ) {
                    *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
                }
            }
            else {
                while ( numzero > 0 ) {
                    *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
                }
                *tonew++ = ARGHEAD+8; *tonew++ = 0; FILLARG(tonew)
                *tonew++ = 8; *tonew++ = SYMBOL; *tonew++ = 4;
```

```
                *tonew++ = ff[ARGHEAD+3]; *tonew++ = ff[ARGHEAD+4];
                *tonew++ = 1; *tonew++ = 1;
                if ( ff[ARGHEAD+7] > 0 ) *tonew++ = 3;
                else                     *tonew++ = -3;
            }
            ff += *ff;
        }
        if ( tonew > AT.WorkTop ) goto OverWork;
        iarg++;
    }
/*
    Copy the tail, settle the size and copy the whole thing back.
*/
    while ( ff < tstop ) *tonew++ = *ff++;
    i = newfun[1] = tonew-newfun;
    NCOPY(fun,newfun,i)
    return(0);
OverWork:;
    MLOCK(ErrorMessageLock);
    MesWork();
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
        #] RunExplode :
        #[ RunPermute :
*/

WORD RunPermute(PHEAD WORD *fun, WORD *args, WORD *info)
{
    WORD *tt, totarg, *tstop, arg1, arg2, n, num, i, *f, *f1, *f2, *infostop;
    WORD *in, *iw, withdollar;
    DOLLARS d;
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunPermute");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
      tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
      while ( tt < tstop ) { totarg++; NEXTARG(tt); }
      arg1 = 1; arg2 = totarg;
/*
      We need to:
        1: get pointers to the arguments
        2: permute the pointers
        3: copy the arguments to safe territory in the new order
        4: copy this new order back in situ.
*/
      num = arg2-arg1+1;
      WantAddPointers(num); /* Guarantees the presence of enough pointers */
      f = fun+FUNHEAD; n = 1; i = 0;
      while ( n < arg1 ) { n++; NEXTARG(f) }
      f1 = f;
      while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
/*
      Now the permutations
*/
      info++;
      while ( *info ) {
        infostop = info + *info;
        info++;
        if ( *info > totarg ) return(0);
/*
        Now we have a look whether there are dollar variables to be expanded
        We also sift out all values that are out of range.
*/
        withdollar = 0;  in = info;
        while ( in < infostop ) {
            if ( *in < 0 ) { /* Dollar variable -(number+1) */
                d = Dollars - *in - 1;
#ifdef WITHPTHREADS
                {
                    int nummodopt, dtype = -1, numdollar = -*in-1;
                    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
                        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                            if ( numdollar == ModOptdollars[nummodopt].number ) break;
                        }
                        if ( nummodopt < NumModOptdollars ) {
                            dtype = ModOptdollars[nummodopt].type;
                            if ( dtype == MODLOCAL ) {
                                d = ModOptdollars[nummodopt].dstruct+AT.identity;
                            }
                            else {
                                LOCK(d->pthreadslockread);
```

```
                            }
                        }
                    }
                }
#endif
                if ( ( d->type == DOLNUMBER || d->type == DOLTERMS )
                 && d->where[0] == 4 && d->where[4] == 0 ) {
                    if ( d->where[3] < 0 || d->where[2] != 1 || d->where[1] > totarg ) return(0);
                }
                else if ( d->type == DOLWILDARGS ) {
                    iw = d->where+1;
                    while ( *iw ) {
                        if ( *iw == -SNUMBER ) {
                            if ( iw[1] <= 0 || iw[1] > totarg ) return(0);
                        }
                        else goto IllType;
                        iw += 2;
                    }
                }
                else {
IllType:
                    MLOCK(ErrorMessageLock);
                    MesPrint("Illegal type of $-variable in RunPermute");
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                withdollar++;
            }
            else if ( *in > totarg ) return(0);
            in++;
        }
        if ( withdollar ) { /* We need some space for a copy */
            WORD *incopy, *tocopy;
            incopy = TermMalloc("RunPermute");
            tocopy = incopy+1; in = info;
            while ( in < infostop ) {
                if ( *in < 0 ) {
                    d = Dollars - *in - 1;
#ifdef WITHPTHREADS
                {
                    int nummodopt, dtype = -1, numdollar = -*in-1;
                    if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
                        for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                            if ( numdollar == ModOptdollars[nummodopt].number ) break;
                        }
                        if ( nummodopt < NumModOptdollars ) {
                            dtype = ModOptdollars[nummodopt].type;
                            if ( dtype == MODLOCAL ) {
                                d = ModOptdollars[nummodopt].dstruct+AT.identity;
                            }
                            else {
                                LOCK(d->pthreadslockread);
                            }
                        }
                    }
                }
#endif
                    if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
                        *tocopy++ = d->where[1] - 1;
                    }
                    else if ( d->type == DOLWILDARGS ) {
                        iw = d->where+1;
                        while ( *iw ) {
                            *tocopy++ = iw[1] - 1;
                            iw += 2;
                        }
                    }
                    in++;
                }
                else *tocopy++ = *in++;
            }
            *tocopy = 0;
            *incopy = tocopy - incopy;
            in = incopy+1;
            tt = AT.pWorkSpace[AT.pWorkPointer+*in];
            in++;
            while ( in < tocopy ) {
                if ( *in > totarg ) return(0);
                AT.pWorkSpace[AT.pWorkPointer+in[-1]] = AT.pWorkSpace[AT.pWorkPointer+*in];
                in++;
            }
            AT.pWorkSpace[AT.pWorkPointer+in[-1]] = tt;
            TermFree(incopy,"RunPermute");
            info = infostop;
        }
        else {
            tt = AT.pWorkSpace[AT.pWorkPointer+*info];
```

```
            info++;
            while ( info < infostop ) {
                if ( *info > totarg ) return(0);
                AT.pWorkSpace[AT.pWorkPointer+info[-1]] = AT.pWorkSpace[AT.pWorkPointer+*info];
                info++;
            }
            AT.pWorkSpace[AT.pWorkPointer+info[-1]] = tt;
        }
      }
/*
      info++;
      while ( *info ) {
        infostop = info + *info;
        info++;
        if ( *info > totarg ) return(0);
        tt = AT.pWorkSpace[AT.pWorkPointer+*info];
        info++;
        while ( info < infostop ) {
            if ( *info > totarg ) return(0);
            AT.pWorkSpace[AT.pWorkPointer+info[-1]] = AT.pWorkSpace[AT.pWorkPointer+*info];
            info++;
        }
        AT.pWorkSpace[AT.pWorkPointer+info[-1]] = tt;
      }
*/
/*
      And the final cleanup
*/
      if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
      f2 = tstop;
      for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
      i = f2 - tstop;
      NCOPY(f1,tstop,i)
    }
    else {  /* tensors */
      tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop-tt;
      arg1 = 1; arg2 = totarg;
      num = arg2-arg1+1;
      WantAddPointers(num); /* Guarantees the presence of enough pointers */
      f = fun+FUNHEAD; n = 1; i = 0;
      while ( n < arg1 ) { n++; f++; }
      f1 = f;
      while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; f++; }
/*
      Now the permutations
*/
      info++;
      while ( *info ) {
        infostop = info + *info;
        info++;
        if ( *info > totarg ) return(0);
        tt = AT.pWorkSpace[AT.pWorkPointer+*info];
        info++;
        while ( info < infostop ) {
            if ( *info > totarg ) return(0);
            AT.pWorkSpace[AT.pWorkPointer+info[-1]] = AT.pWorkSpace[AT.pWorkPointer+*info];
            info++;
        }
        AT.pWorkSpace[AT.pWorkPointer+info[-1]] = tt;
      }
/*
      And the final cleanup
*/
      if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
      f2 = tstop;
      for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; *f2++= *f++; }
      i = f2 - tstop;
      NCOPY(f1,tstop,i)
    }
    return(0);
OverWork:;
    MLOCK(ErrorMessageLock);
    MesWork();
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
        #] RunPermute :
        #[ RunReverse :
*/

WORD RunReverse(PHEAD WORD *fun, WORD *args)
{
    WORD *tt, totarg, *tstop, arg1, arg2, n, num, i, *f, *f1, *f2, i1, i2;
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
```

```
        MesPrint("Illegal range encountered in RunReverse");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
      tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
      while ( tt < tstop ) { totarg++; NEXTARG(tt); }
      if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
/*
      We need to:
        1: get pointers to the arguments
        2: reverse the order of the pointers
        3: copy the arguments to safe territory in the new order
        4: copy this new order back in situ.
*/
      if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
      if ( arg2 > totarg ) return(0);

      num = arg2-arg1+1;
      WantAddPointers(num); /* Guarantees the presence of enough pointers */
      f = fun+FUNHEAD; n = 1; i = 0;
      while ( n < arg1 ) { n++; NEXTARG(f) }
      f1 = f;
      while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
      i1 = i-1; i2 = 0;
      while ( i1 > i2 ) {
        tt = AT.pWorkSpace[AT.pWorkPointer+i1];
        AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
        AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
        i1--; i2++;
      }
      if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
      f2 = tstop;
      for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
      i = f2 - tstop;
      NCOPY(f1,tstop,i)
    }
    else {  /* Tensors */
      tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop - tt;
      if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
/*
      We need to:
        1: get pointers to the arguments
        2: reverse the order of the pointers
        3: copy the arguments to safe territory in the new order
        4: copy this new order back in situ.
*/
      if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
      if ( arg2 > totarg ) return(0);

      num = arg2-arg1+1;
      WantAddPointers(num); /* Guarantees the presence of enough pointers */
      f = fun+FUNHEAD; n = 1; i = 0;
      while ( n < arg1 ) { n++; f++; }
      f1 = f;
      while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; f++; }
      i1 = i-1; i2 = 0;
      while ( i1 > i2 ) {
        tt = AT.pWorkSpace[AT.pWorkPointer+i1];
        AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
        AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
        i1--; i2++;
      }
      if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
      f2 = tstop;
      for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; *f2++ = *f++; }
      i = f2 - tstop;
      NCOPY(f1,tstop,i)
    }
    return(0);
OverWork:;
    MLOCK(ErrorMessageLock);
    MesWork();
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
        #] RunReverse :
        #[ RunDedup :
*/

WORD RunDedup(PHEAD WORD *fun, WORD *args)
{
    WORD *tt, totarg, *tstop, arg1, arg2, n, i, j,k, *f, *f1, *f2, *fd, *fstart;
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
```

```
        MesPrint("Illegal range encountered in RunDedup");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
      tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
      while ( tt < tstop ) { totarg++; NEXTARG(tt); }
      if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);

      if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
      if ( arg2 > totarg ) return(0);

      f = fun+FUNHEAD; n = 1;
      while ( n < arg1 ) { n++; NEXTARG(f) }
      f1 = f; // fast forward to first element in range
      i = 0; // new argument count
      fstart = f1;

      for (; n <= arg2; n++ ) {
        f2 = fstart;
        for ( j = 0; j < i; j++ ) { // check all previous terms
            fd = f2;
            NEXTARG(fd)
            for ( k = 0; k < fd-f2; k++ ) // byte comparison of args
                if ( f2[k] != f[k] ) break;

            if ( k == fd-f2 ) break; // duplicate arg
            f2 = fd;
        }

        if ( j == i ) {
            // unique factor, copy in situ
            COPY1ARG(f1,f)
            i++;
        } else {
            NEXTARG(f)
        }
      }

      // move the terms from after the range
      for (j = n; j <= totarg; j++) {
        COPY1ARG(f1,f)
      }

      fun[1] = f1 - fun; // resize function
    }
    else {  /* Tensors */
      tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop - tt;
      if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);

      if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
      if ( arg2 > totarg ) return(0);

      f = fun+FUNHEAD;
      i = arg1; // new argument count
      n = i;

      for (; n <= arg2; n++ ) {
        for ( j = arg1; j < i; j++ ) { // check all previous terms
            if ( f[n-1] == f[j-1] ) break; // duplicate arg
        }

        if ( j == i ) {
            // unique factor, copy in situ
            f[i-1] = f[n-1];
            i++;
        }
      }

      // move the terms from after the range
      for (j = n; j <= totarg; j++, i++) {
        f[i-1] = f[j-1];
      }

      fun[1] = f + i - 1 - fun; // resize function
    }
    return(0);
}

/*
        #] RunDedup :
        #[ RunCycle :
*/

WORD RunCycle(PHEAD WORD *fun, WORD *args, WORD *info)
{
    WORD *tt, totarg, *tstop, arg1, arg2, n, num, i, j, *f, *f1, *f2, x, ncyc, cc;
```

```
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunCycle");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    ncyc = info[1];
    if ( ncyc >= MAXPOSITIVE2 ) { /* $ variable */
        ncyc -= MAXPOSITIVE2;
        if ( ncyc >= MAXPOSITIVE4 ) {
            ncyc -= MAXPOSITIVE4; /* -$ */
            cc = -1;
        }
        else cc = 1;
        ncyc = DolToNumber(BHEAD ncyc);
        if ( AN.ErrorInDollar ) {
            MesPrint(" Error in Dollar variable in transform,cycle()=$");
            return(-1);
        }
        if ( ncyc >= MAXPOSITIVE4 || ncyc <= -MAXPOSITIVE4 ) {
            MesPrint(" Illegal value from Dollar variable in transform,cycle()=$");
            return(-1);
        }
        ncyc *= cc;
    }
    if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
      tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
      while ( tt < tstop ) { totarg++; NEXTARG(tt); }
      if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
      if ( arg1 > arg2 ) { n = arg1; arg1 = arg2; arg2 = n; }
      if ( arg2 > totarg ) return(0);
/*
      We need to:
        1: get pointers to the arguments
        2: cycle the pointers
        3: copy the arguments to safe territory in the new order
        4: copy this new order back in situ.
*/
      num = arg2-arg1+1;
      WantAddPointers(num); /* Guarantees the presence of enough pointers */
      f = fun+FUNHEAD; n = 1; i = 0;
      while ( n < arg1 ) { n++; NEXTARG(f) }
      f1 = f;
      while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
/*
      Now the cycle(s). First minimize the number of cycles.
*/
      x = ncyc;
      if ( x >= i ) {
        x %= i;
        if ( x > i/2 ) x -= i;
      }
      else if ( x <= -i ) {
        x = -((-x) % i);
        if ( x <= -i/2 ) x += i;
      }
      while ( x ) {
        if ( x > 0 ) {
            tt = AT.pWorkSpace[AT.pWorkPointer+i-1];
            for ( j = i-1; j > 0; j-- )
                AT.pWorkSpace[AT.pWorkPointer+j] = AT.pWorkSpace[AT.pWorkPointer+j-1];
            AT.pWorkSpace[AT.pWorkPointer] = tt;
            x--;
        }
        else {
            tt = AT.pWorkSpace[AT.pWorkPointer];
            for ( j = 1; j < i; j++ )
                AT.pWorkSpace[AT.pWorkPointer+j-1] = AT.pWorkSpace[AT.pWorkPointer+j];
            AT.pWorkSpace[AT.pWorkPointer+j-1] = tt;
            x++;
        }
      }
/*
      And the final cleanup
*/
      if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
      f2 = tstop;
      for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
      i = f2 - tstop;
      NCOPY(f1,tstop,i)
    }
    else {  /* Tensors */
      tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop - tt;
      if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
      if ( arg1 > arg2 ) { n = arg1; arg1 = arg2; arg2 = n; }
      if ( arg2 > totarg ) return(0);
/*
```

```
      We need to:
        1: get pointers to the arguments
        2: cycle the pointers
        3: copy the arguments to safe territory in the new order
        4: copy this new order back in situ.
*/
      num = arg2-arg1+1;
      WantAddPointers(num); /* Guarantees the presence of enough pointers */
      f = fun+FUNHEAD; n = 1; i = 0;
      while ( n < arg1 ) { n++; f++; }
      f1 = f;
      while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; f++; }
/*
      Now the cycle(s). First minimize the number of cycles.
*/
      x = ncyc;
      if ( x >= i ) {
        x %= i;
        if ( x > i/2 ) x -= i;
      }
      else if ( x <= -i ) {
        x = -((-x) % i);
        if ( x <= -i/2 ) x += i;
      }
      while ( x ) {
        if ( x > 0 ) {
            tt = AT.pWorkSpace[AT.pWorkPointer+i-1];
            for ( j = i-1; j > 0; j-- )
                AT.pWorkSpace[AT.pWorkPointer+j] = AT.pWorkSpace[AT.pWorkPointer+j-1];
            AT.pWorkSpace[AT.pWorkPointer] = tt;
            x--;
        }
        else {
            tt = AT.pWorkSpace[AT.pWorkPointer];
            for ( j = 1; j < i; j++ )
                AT.pWorkSpace[AT.pWorkPointer+j-1] = AT.pWorkSpace[AT.pWorkPointer+j];
            AT.pWorkSpace[AT.pWorkPointer+j-1] = tt;
            x++;
        }
      }
/*
      And the final cleanup
*/
      if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
      f2 = tstop;
      for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; *f2++ = *f++; }
      i = f2 - tstop;
      NCOPY(f1,tstop,i)
    }
    return(0);
OverWork:;
    MLOCK(ErrorMessageLock);
    MesWork();
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
        #] RunCycle :
        #[ RunAddArg :
*/

WORD RunAddArg(PHEAD WORD *fun, WORD *args)
{
    WORD *tt, totarg, *tstop, arg1, arg2, n, num, *f, *f1, *f2;
    WORD scribble[10+ARGHEAD];
    LONG space;
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunAddArg");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    if ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal attempt to add arguments of a tensor in AddArg");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( tt < tstop ) { totarg++; NEXTARG(tt); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
/*
    We need to:
        1: establish that we actually need to add something
        2: start a sort
        3: if needed, convert arguments to long arguments
```

```
        4: send (terms in) argument to StoreTerm
        5: EndSort and copy the result back into the function
    Note that the function is in the workspace, above the term and no
    relevant information is trailing it.
*/
    if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
    if ( arg2 > totarg ) return(0);
    num = arg2-arg1+1;
    if ( num == 1 ) return(0);
    f = fun+FUNHEAD; n = 1;
    while ( n < arg1 ) { n++; NEXTARG(f) }
    f1 = f;
    NewSort(BHEAD0);
    while ( n <= arg2 ) {
        if ( *f > 0 ) {
            f2 = f + *f; f += ARGHEAD;
            while ( f < f2 ) { StoreTerm(BHEAD f); f += *f; }
        }
        else if ( *f == -SNUMBER && f[1] == 0 ) {
            f+= 2;
        }
        else {
            ToGeneral(f,scribble,1);
            StoreTerm(BHEAD scribble);
            NEXTARG(f);
        }
        n++;
    }
    if ( EndSort(BHEAD tstop+ARGHEAD,0) ) return(-1);
    num = 0;
    f2 = tstop+ARGHEAD;
    while ( *f2 ) { f2 += *f2; num++; }
    *tstop = f2-tstop;
    for ( n = 1; n < ARGHEAD; n++ ) tstop[n] = 0;
    if ( num == 1 && ToFast(tstop,tstop) == 1 ) {
        f2 = tstop; NEXTARG(f2);
    }
/*
    Copy the trailing arguments after the new argument, then copy the whole back.
*/
    while ( f < tstop ) *f2++ = *f++;
    while ( f < f2 ) *f1++ = *f++;
    space = f1 - fun;
    if ( (space+8)*sizeof(WORD) > (UWORD)AM.MaxTer ) {
        MLOCK(ErrorMessageLock);
        MesWork();
        MUNLOCK(ErrorMessageLock);
        return(-1);
    }
    fun[1] = (WORD)space;
    return(0);
}

/*
        #] RunAddArg :
        #[ RunMulArg :
*/

WORD RunMulArg(PHEAD WORD *fun, WORD *args)
{
    WORD *t, totarg, *tstop, arg1, arg2, n, *f, nb, *m, i, *w;
    WORD *scratch, argbuf[20], argsize, *where, *newterm;
    LONG oldcpointer_pos;
    CBUF *C = cbuf + AT.ebufnum;
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunMulArg");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    if ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal attempt to multiply arguments of a tensor in MulArg");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    t = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( t < tstop ) { totarg++; NEXTARG(t); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
    if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
    if ( arg1 > totarg ) return(0);
    if ( arg2 < 1 ) return(0);
    if ( arg1 < 1 ) arg1 = 1;
    if ( arg2 > totarg ) arg2 = totarg;
    if ( arg1 == arg2 ) return(0);
/*
    Now we move the arguments to a compiler buffer
```

```
    Then we create a term in the workspace that is the product of
    subexpression pointers to the objects in the compiler buffer.
    Next we let Generator work out that term.
    Finally we pick up the results from EndSort and put it in the function.
*/
    f = fun+FUNHEAD; n = 1;
    while ( n < arg1 ) { n++; NEXTARG(f) }
    t = f;
    if ( fun >= AT.WorkSpace && fun < AT.WorkTop ) {
        if ( AT.WorkPointer < fun+fun[1] ) AT.WorkPointer = fun+fun[1];
    }
    scratch = AT.WorkPointer;
    w = scratch+1;
    oldcpointer_pos = C->Pointer-C->Buffer;
    nb = C->numrhs;
    while ( n <= arg2 ) {
        if ( *t > 0 ) {
            argsize = *t - ARGHEAD; where = t + ARGHEAD; t += *t;
        }
        else if ( *t <= -FUNCTION ) {
            argbuf[0] = FUNHEAD+4; argbuf[1] = -*t++; argbuf[2] = FUNHEAD;
            for ( i = 2; i < FUNHEAD; i++ ) argbuf[i+1] = 0;
            argbuf[FUNHEAD+1] = 1;
            argbuf[FUNHEAD+2] = 1;
            argbuf[FUNHEAD+3] = 3;
            argsize = argbuf[0];
            where = argbuf;
        }
        else if ( *t == -SYMBOL ) {
            argbuf[0] = 8; argbuf[1] = SYMBOL; argbuf[2] = 4;
            argbuf[3] = t[1]; argbuf[4] = 1;
            argbuf[5] = 1; argbuf[6] = 1; argbuf[7] = 3;
            argsize = 8; t += 2;
            where = argbuf;
        }
        else if ( *t == -VECTOR || *t == -MINVECTOR ) {
            argbuf[0] = 7; argbuf[1] = INDEX; argbuf[2] = 3;
            argbuf[3] = t[1];
            argbuf[4] = 1; argbuf[5] = 1;
            if ( *t == -MINVECTOR ) argbuf[6] = -3;
            else argbuf[6] = 3;
            argsize = 7; t += 2;
            where = argbuf;
        }
        else if ( *t == -INDEX ) {
            argbuf[0] = 7; argbuf[1] = INDEX; argbuf[2] = 3;
            argbuf[3] = t[1];
            argbuf[4] = 1; argbuf[5] = 1; argbuf[6] = 3;
            argsize = 7; t += 2;
            where = argbuf;
        }
        else if ( *t == -SNUMBER ) {
            if ( t[1] < 0 ) {
                argbuf[0] = 4; argbuf[1] = -t[1]; argbuf[2] = 1; argbuf[3] = -3;
            }
            else {
                argbuf[0] = 4; argbuf[1] = t[1]; argbuf[2] = 1; argbuf[3] = 3;
            }
            argsize = 4; t += 2;
            where = argbuf;
        }
        else {
            /* unreachable */
            return(1);
        }
/*
        Now add the argbuf to AT.ebufnum
*/
        m = AddRHS(AT.ebufnum,1);
        while ( (m + argsize + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,17);
        for ( i = 0; i < argsize; i++ ) m[i] = where[i];
        m[i] = 0;
        C->Pointer = m + i + 1;
        n++;
        *w++ = SUBEXPRESSION; *w++ = SUBEXPSIZE; *w++ = C->numrhs; *w++ = 1;
        *w++ = AT.ebufnum; FILLSUB(w);
    }
    *w++ = 1; *w++ = 1; *w++ = 3;
    *scratch = w-scratch;
    AT.WorkPointer = w;
    NewSort(BHEAD0);
    Generator(BHEAD scratch,AR.Cnumlhs);
    newterm = AT.WorkPointer;
    EndSort(BHEAD newterm+ARGHEAD,0);
    C->Pointer = C->Buffer+oldcpointer_pos;
    C->numrhs = nb;
    w = newterm+ARGHEAD; while ( *w ) w += *w;
```

```
    *newterm = w-newterm; newterm[1] = 0;
    if ( ToFast(newterm,newterm) ) {
        if ( *newterm <= -FUNCTION ) w = newterm+1;
        else w = newterm+2;
    }
    while ( t < tstop ) *w++ = *t++;
    i = w - newterm;
    t = newterm; NCOPY(f,t,i);
    fun[1] = f-fun;
    AT.WorkPointer = scratch;
    if ( AT.WorkPointer > AT.WorkSpace && AT.WorkPointer < f ) AT.WorkPointer = f;
    return(0);
}

/*
        #] RunMulArg :
        #[ RunIsLyndon :

        Determines whether the range constitutes a Lyndon word.
        The two cases of ordering are distinguised by the order of
        the numbers of the arguments in the range.
*/

WORD RunIsLyndon(PHEAD WORD *fun, WORD *args, int par)
{
    WORD *tt, totarg, *tstop, arg1, arg2, arg, num, *f, n, i;
/*  WORD *f1; */
    WORD sign, i1, i2, retval;
    if ( fun[0] <= GAMMASEVEN && fun[0] >= GAMMA ) return(0);
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunIsLyndon");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( tt < tstop ) { totarg++; NEXTARG(tt); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
    if ( arg1 > totarg || arg2 > totarg ) return(-1);
/*
    Now make a list of the relevant arguments.
*/
    if ( arg1 == arg2 ) return(1);
    if ( arg2 < arg1 ) {    /* greater, rather than smaller */
        arg = arg1; arg1 = arg2; arg2 = arg; sign = 1;
    }
    else sign = 0;

    num = arg2-arg1+1;
    WantAddPointers(num);   /* Guarantees the presence of enough pointers */
    f = fun+FUNHEAD; n = 1; i = 0;
    while ( n < arg1 ) { n++; NEXTARG(f) }
/*  f1 = f; */
    while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
/*
    If sign == 1 we should alter the order of the pointers first
*/
    if ( sign ) {
        i1 = i-1; i2 = 0;
        while ( i1 > i2 ) {
            tt = AT.pWorkSpace[AT.pWorkPointer+i1];
            AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
            AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
            i1--; i2++;
        }
    }
/*
    The argument range is from f1 to f and the num pointers to the arguments
    are in AT.pWorkSpace[AT.pWorkPointer] to AT.pWorkSpace[AT.pWorkPointer+num-1]
*/
    for ( i1 = 1; i1 < num; i1++ ) {
        retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+i1],
                               AT.pWorkSpace[AT.pWorkPointer]);
        if ( retval > 0 ) continue;
        if ( retval < 0 ) return(0);
        for ( i2 = 1; i2 < num; i2++ ) {
            retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+(i1+i2)%num],
                               AT.pWorkSpace[AT.pWorkPointer+i2]);
            if ( retval < 0 ) return(0);
            if ( retval > 0 ) goto nextii1;
        }
/*
        If we come here the sequence is not unique.
*/
        return(0);
nextii1:;
    }
```

```
    return(1);
}

/*
        #] RunIsLyndon :
        #[ RunToLyndon :

        Determines whether the range constitutes a Lyndon word.
        If not, we rotate it to a Lyndon word. If this is not possible
        we return the noLyndon condition.
        The two cases of ordering are distinguised by the order of
        the numbers of the arguments in the range.
*/

WORD RunToLyndon(PHEAD WORD *fun, WORD *args, int par)
{
    WORD *tt, totarg, *tstop, arg1, arg2, arg, num, *f, *f1, *f2, n, i;
    WORD sign, i1, i2, retval, unique;
    if ( fun[0] <= GAMMASEVEN && fun[0] >= GAMMA ) return(0);
    if ( *args != ARGRANGE ) {
        MLOCK(ErrorMessageLock);
        MesPrint("Illegal range encountered in RunToLyndon");
        MUNLOCK(ErrorMessageLock);
        Terminate(-1);
    }
    tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( tt < tstop ) { totarg++; NEXTARG(tt); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
    if ( arg1 > totarg || arg2 > totarg ) return(-1);
/*
    Now make a list of the relevant arguments.
*/
    if ( arg1 == arg2 ) return(1);
    if ( arg2 < arg1 ) {    /* greater, rather than smaller */
        arg = arg1; arg1 = arg2; arg2 = arg; sign = 1;
    }
    else sign = 0;

    num = arg2-arg1+1;
    WantAddPointers((2*num));   /* Guarantees the presence of enough pointers */
    f = fun+FUNHEAD; n = 1; i = 0;
    while ( n < arg1 ) { n++; NEXTARG(f) }
    f1 = f;
    while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
/*
    If sign == 1 we should alter the order of the pointers first
*/
    if ( sign ) {
        i1 = i-1; i2 = 0;
        while ( i1 > i2 ) {
            tt = AT.pWorkSpace[AT.pWorkPointer+i1];
            AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
            AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
            i1--; i2++;
        }
    }
/*
    The argument range is from f1 to f and the num pointers to the arguments
    are in AT.pWorkSpace[AT.pWorkPointer] to AT.pWorkSpace[AT.pWorkPointer+num-1]
*/
    unique = 1;
    for ( i1 = 1; i1 < num; i1++ ) {
        retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+i1],
                               AT.pWorkSpace[AT.pWorkPointer]);
        if ( retval > 0 ) continue;
        if ( retval < 0 ) {
Rotate:;
/*
            Rotate so that i1 becomes the zero element. Then start again.
*/
            for ( i2 = 0; i2 < num; i2++ ) {
                AT.pWorkSpace[AT.pWorkPointer+num+i2] =
                    AT.pWorkSpace[AT.pWorkPointer+(i1+i2)%num];
            }
            for ( i2 = 0; i2 < num; i2++ ) {
                AT.pWorkSpace[AT.pWorkPointer+i2] =
                    AT.pWorkSpace[AT.pWorkPointer+i2+num];
            }
            i1 = 0;
            goto nexti1;
        }
        for ( i2 = 1; i2 < num; i2++ ) {
            retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+(i1+i2)%num],
                               AT.pWorkSpace[AT.pWorkPointer+i2]);
            if ( retval < 0 ) goto Rotate;
            if ( retval > 0 ) goto nexti1;
        }
```

```
/*
        If we come here the sequence is not unique.
*/
        unique = 0;
nexti1:;
    }
    if ( sign ) {
        i1 = i-1; i2 = 0;
        while ( i1 > i2 ) {
            tt = AT.pWorkSpace[AT.pWorkPointer+i1];
            AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
            AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
            i1--; i2++;
        }
    }
/*
    Now rewrite the arguments into the proper order
*/
    if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
    f2 = tstop;
    for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
    i = f2 - tstop;
    NCOPY(f1,tstop,i)
/*
    The return value indicates whether we have a Lyndon word
*/
    return(unique);
OverWork:;
    MLOCK(ErrorMessageLock);
    MesWork();
    MUNLOCK(ErrorMessageLock);
    return(-2);
}

/*
        #] RunToLyndon :
        #[ RunDropArg :
*/

WORD RunDropArg(PHEAD WORD *fun, WORD *args)
{
    WORD *t, *tstop, *f, totarg, arg1, arg2, n;

    t = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( t < tstop ) { totarg++; NEXTARG(t); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
    if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
    if ( arg1 > totarg ) return(0);
    if ( arg2 < 1 ) return(0);
    if ( arg1 < 1 ) arg1 = 1;
    if ( arg2 > totarg ) arg2 = totarg;
    f = fun+FUNHEAD; n = 1;
    while ( n < arg1 ) { n++; NEXTARG(f) }
    t = f;
    while ( n <= arg2 ) { n++; NEXTARG(t) }
    while ( t < tstop ) *f++ = *t++;
    fun[1] = f-fun;
    return(0);
}

/*
        #] RunDropArg :
        #[ RunSelectArg :
*/

WORD RunSelectArg(PHEAD WORD *fun, WORD *args)
{
    WORD *t, *tstop, *f, *tt, totarg, arg1, arg2, n;

    t = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
    while ( t < tstop ) { totarg++; NEXTARG(t); }
    if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
    if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
    if ( arg1 > totarg ) return(0);
    if ( arg2 < 1 ) return(0);
    if ( arg1 < 1 ) arg1 = 1;
    if ( arg2 > totarg ) arg2 = totarg;
    f = fun+FUNHEAD; n = 1; t = f;
    while ( n < arg1 ) { n++; NEXTARG(t) }
    while ( n <= arg2 ) {
        tt = t; NEXTARG(tt)
        while ( t < tt ) *f++ = *t++;
        n++;
    }
    fun[1] = f-fun;
    return(0);
}
```

```c

/*
        #] RunSelectArg :
        #[ TestArgNum :

        Looks whether argument n is contained in any of the ranges
        specified in args. Args contains objects of the types
            ALLARGS
            NUMARG,num
            ARGRANGE,num1,num2
        The object MAKEARGS,num1,num2 is skipped
        Any other object terminates the range specifications.

        Currently only ARGRANGE is used (10-may-2016)
*/

int TestArgNum(int n, int totarg, WORD *args)
{
    GETIDENTITY
    WORD x1, x2;
    for(;;) {
        switch ( *args ) {
            case ALLARGS:
                return(1);
            case NUMARG:
                if ( n == args[1] ) return(1);
                if ( args[1] >= MAXPOSITIVE4 ) {
                    x1 = args[1]-MAXPOSITIVE4;
                    if ( totarg-x1 == n ) return(1);
                }
                args += 2;
                break;
            case ARGRANGE:
                if ( args[1] >= MAXPOSITIVE2 ) {
                    x1 = args[1] - MAXPOSITIVE2;
                    if ( x1 > MAXPOSITIVE4 ) {
                        x1 = x1 - MAXPOSITIVE4;
                        x1 = DolToNumber(BHEAD x1);
                        x1 = totarg - x1;
                    }
                    else {
                        x1 = DolToNumber(BHEAD x1);
                    }
                }
                else if ( args[1] >= MAXPOSITIVE4 ) {
                    x1 = totarg-(args[1]-MAXPOSITIVE4);
                }
                else x1 = args[1];
                if ( args[2] >= MAXPOSITIVE2 ) {
                    x2 = args[2] - MAXPOSITIVE2;
                    if ( x2 > MAXPOSITIVE4 ) {
                        x2 = x2 - MAXPOSITIVE4;
                        x2 = DolToNumber(BHEAD x2);
                        x2 = totarg - x2;
                    }
                    else {
                        x2 = DolToNumber(BHEAD x2);
                    }
                }
                else if ( args[2] >= MAXPOSITIVE4 ) {
                    x2 = totarg-(args[2]-MAXPOSITIVE4);
                }
                else x2 = args[2];
                if ( x1 >= x2 ) {
                    if ( n >= x2 && n <= x1 ) return(1);
                }
                else {
                    if ( n >= x1 && n <= x2 ) return(1);
                }
                args += 3;
                break;
            case MAKEARGS:
                args += 3;
                break;
            default:
                return(0);
        }
    }
}

/*
        #] TestArgNum :
        #[ PutArgInScratch :
*/

WORD PutArgInScratch(WORD *arg,UWORD *scrat)
{
```

```
    WORD size, *t, i;
    if ( *arg == -SNUMBER ) {
        scrat[0] = ABS(arg[1]);
        if ( arg[1] < 0 ) size = -1;
        else              size =  1;
    }
    else {
        t = arg+*arg-1;
        if ( *t < 0 ) { i = ((-*t)-1)/2; size = -i; }
        else          { i = (  *t -1)/2; size =  i; }
        t = arg+ARGHEAD+1;
        NCOPY(scrat,t,i);
    }
    return(size);
}

/*
        #] PutArgInScratch :
        #[ ReadRange :

        Comes in at the bracket and leaves at the = sign
        Ranges can be:
            #1,#2  with # numbers. If the second is smaller than the
                    first we work it backwards.
            first,#2 or #2,first
            #1,last  or last,#1
            first,last or last,first
        First is represented by 1. Last is represented by MAXPOSITIVE4.

        par = 0: we need the = after.
        par = 1: we need a , or '\0' after.
        par = 2: we need a :
*/

UBYTE *ReadRange(UBYTE *s, WORD *out, int par)
{
    UBYTE *in = s, *ss, c;
    LONG x1, x2;

    SKIPBRA3(in)
    if ( par == 0 && in[1] != '=' ) {
        MesPrint("&A range in this type of transform statement should be followed by an = sign");
        return(0);
    }
    else if ( par == 1 && in[1] != ',' && in[1] != '\0' ) {
        MesPrint("&A range in this type of transform statement should be followed by a comma or end-of-statement");
        return(0);
    }
    else if ( par == 2 && in[1] != ':' ) {
        MesPrint("&A range in this type of transform statement should be followed by a :");
        return(0);
    }
    s++;
    if ( FG.cTable[*s] == 0 ) {
        ss = s; while ( FG.cTable[*s] == 0 ) s++;
        c = *s; *s = 0;
        if ( StrICmp(ss,(UBYTE *)"first") == 0 ) {
            *s = c;
            x1 = 1;
        }
        else if ( StrICmp(ss,(UBYTE *)"last") == 0 ) {
            *s = c;
            if ( c == '-' ) {
                s++;
                if ( *s == '$' ) {
                    s++; ss = s;
                    while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
                    c = *s; *s = 0;
                    if ( ( x1 = GetDollar(ss) ) < 0 ) goto Error;
                    *s = c;
                    x1 += MAXPOSITIVE2;
                }
                else {
                    x1 = 0;
                    while ( *s >= '0' && *s <= '9' ) {
                        x1 = 10*x1 + *s++ - '0';
                        if ( x1 >= MAXPOSITIVE4 ) {
                            MesPrint("&Fixed range indicator bigger than %l",(LONG)MAXPOSITIVE4);
                            return(0);
                        }
                    }
                }
                x1 += MAXPOSITIVE4;
            }
            else x1 = MAXPOSITIVE4;
        }
```

```
        else {
            MesPrint("&Illegal keyword inside range specification");
            return(0);
        }
    }
    else if ( FG.cTable[*s] == 1 ) {
        x1 = 0;
        while ( *s >= '0' && *s <= '9' ) {
            x1 = x1*10 + *s++ - '0';
            if ( x1 >= MAXPOSITIVE4 ) {
                MesPrint("&Fixed range indicator bigger than %l",(LONG)MAXPOSITIVE4);
                return(0);
            }
        }
    }
    else if ( *s == '$' ) {
        s++; ss = s;
        while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
        c = *s; *s = 0;
        if ( ( x1 = GetDollar(ss) ) < 0 ) goto Error;
        *s = c;
        x1 += MAXPOSITIVE2;
    }
    else {
        MesPrint("&Illegal character in range specification");
        return(0);
    }
    if ( *s != ',' ) {
        MesPrint("&A range is two indicators, separated by a comma or blank");
        return(0);
    }
    s++;
    if ( FG.cTable[*s] == 0 ) {
        ss = s; while ( FG.cTable[*s] == 0 ) s++;
        c = *s; *s = 0;
        if ( StrICmp(ss,(UBYTE *)"first") == 0 ) {
            *s = c;
            x2 = 1;
        }
        else if ( StrICmp(ss,(UBYTE *)"last") == 0 ) {
            *s = c;
            if ( c == '-' ) {
                s++;
                if ( *s == '$' ) {
                    s++; ss = s;
                    while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
                    c = *s; *s = 0;
                    if ( ( x2 = GetDollar(ss) ) < 0 ) goto Error;
                    *s = c;
                    x2 += MAXPOSITIVE2;
                }
                else {
                    x2 = 0;
                    while ( *s >= '0' && *s <= '9' ) {
                        x2 = 10*x2 + *s++ - '0';
                        if ( x2 >= MAXPOSITIVE4 ) {
                            MesPrint("&Fixed range indicator bigger than %l",(LONG)MAXPOSITIVE4);
                            return(0);
                        }
                    }
                }
                x2 += MAXPOSITIVE4;
            }
            else x2 = MAXPOSITIVE4;
        }
        else {
            MesPrint("&Illegal keyword inside range specification");
            return(0);
        }
    }
    else if ( FG.cTable[*s] == 1 ) {
        x2 = 0;
        while ( *s >= '0' && *s <= '9' ) {
            x2 = x2*10 + *s++ - '0';
            if ( x2 >= MAXPOSITIVE4 ) {
                MesPrint("&Fixed range indicator bigger than %l",(LONG)MAXPOSITIVE4);
                return(0);
            }
        }
    }
    else if ( *s == '$' ) {
        s++; ss = s;
        while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
        c = *s; *s = 0;
        if ( ( x2 = GetDollar(ss) ) < 0 ) goto Error;
        *s = c;
        x2 += MAXPOSITIVE2;
```

```
    }
    else {
        MesPrint("&Illegal character in range specification");
        return(0);
    }
    if ( s < in ) {
        MesPrint("&A range is two indicators, separated by a comma or blank between parentheses");
        return(0);
    }
    out[0] = x1; out[1] = x2;
    return(in+1);
Error:
    MesPrint("&Undefined variable $%s in range",ss);
    return(0);
}

/*
        #] ReadRange :
        #[ FindRange :
*/

int FindRange(PHEAD WORD *args, WORD *arg1, WORD *arg2, WORD totarg)
{
    WORD n[2], fromlast, i;
    for ( i = 0; i < 2; i++ ) {
        n[i] = args[i+1];
        fromlast = 0;
        if ( n[i] >= MAXPOSITIVE2 ) { /* This is a dollar variable */
            n[i] -= MAXPOSITIVE2;
            if ( n[i] >= MAXPOSITIVE4 ) {
                fromlast = 1;
                n[i] -= MAXPOSITIVE4; /* Now we have the number of the dollar variable   */
            }
            n[i] = DolToNumber(BHEAD n[i]);
            if ( AN.ErrorInDollar ) goto Error;
            if ( fromlast ) n[i] = totarg-n[i];
        }
        else if ( n[i] >= MAXPOSITIVE4 ) { n[i] = totarg-(n[i]-MAXPOSITIVE4); }
        if ( n[i] <= 0 ) goto Error;
    }
    *arg1 = n[0];
    *arg2 = n[1];
    return(0);
Error:
    MLOCK(ErrorMessageLock);
    MesPrint("Illegal $ value in range while executing transform statement.");
    MUNLOCK(ErrorMessageLock);
    return(-1);
}

/*
        #] FindRange :
    #] Transform :
*/
```

4.122 unix.h File Reference

Macros

- #define LINEFEED '\n'
- #define CARRIAGERETURN 0x0D
- #define WITHPIPE
- #define WITHSYSTEM
- #define WITHEXTERNALCHANNEL
- #define TRAPSIGNALS
- #define P_term(code) exit((int)(code<0?-code:code))
- #define SEPARATOR '/'
- #define ALTSEPARATOR '/'
- #define PATHSEPARATOR ':'
- #define WITH_ENV

4.122.1 Detailed Description

Settings for Unix-like systems.

Definition in file unix.h.

4.122.2 Macro Definition Documentation

4.122.2.1 LINEFEED

```
#define LINEFEED '\n'
```

Definition at line 33 of file unix.h.

4.122.2.2 CARRIAGERETURN

```
#define CARRIAGERETURN 0x0D
```

Definition at line 34 of file unix.h.

4.122.2.3 WITHPIPE

```
#define WITHPIPE
```

Definition at line 36 of file unix.h.

4.122.2.4 WITHSYSTEM

```
#define WITHSYSTEM
```

Definition at line 37 of file unix.h.

4.122.2.5 WITHEXTERNALCHANNEL

```
#define WITHEXTERNALCHANNEL
```

Definition at line 46 of file unix.h.

4.122.2.6 TRAPSIGNALS

```
#define TRAPSIGNALS
```

Definition at line 49 of file unix.h.

4.122.2.7 P_term

```
#define P_term(
              code ) exit((int)(code<0?-code:code))
```

Definition at line 52 of file unix.h.

4.122.2.8 SEPARATOR

```
#define SEPARATOR '/'
```

Definition at line 54 of file unix.h.

4.122.2.9 ALTSEPARATOR

```
#define ALTSEPARATOR '/'
```

Definition at line 55 of file unix.h.

4.122.2.10 PATHSEPARATOR

```
#define PATHSEPARATOR ':'
```

Definition at line 56 of file unix.h.

4.122.2.11 WITH_ENV

```
#define WITH_ENV
```

Definition at line 57 of file unix.h.

4.123 unix.h

Go to the documentation of this file.

```
00001
00006 /* #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00026  *   details.
```

```
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

#define LINEFEED '\n'
#define CARRIAGERETURN 0x0D

#define WITHPIPE
#define WITHSYSTEM

/*[13jul2005 mt]:*/
/*With SAFESIGNAL defined, write() and read() syscalls are wrapped by
the errno checkup*/
/*#define SAFESIGNAL*/
/*:[13jul2005 mt]*/

/*[29apr2004 mt]:*/
#define WITHEXTERNALCHANNEL
/*
*/
#define TRAPSIGNALS
/*:[29apr2004 mt]*/

#define P_term(code)    exit((int)(code<0?-code:code))

#define SEPARATOR '/'
#define ALTSEPARATOR '/'
#define PATHSEPARATOR ':'
#define WITH_ENV
```

4.124 unixfile.c File Reference

`#include "form3.h"`

Include dependency graph for unixfile.c:

4.124.1 Detailed Description

The interface to a fast variety of file routines in the unix system.

Definition in file unixfile.c.

4.125 unixfile.c

Go to the documentation of this file.

```
/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
```

```
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes :

    File with direct interface to the UNIX functions.
    This makes a big difference in speed!
*/
#include "form3.h"


/*[13jul2005 mt]:*/
#ifdef SAFESIGNAL
/*[15oct2004 mt]:*/
/*To access errno variable and EINTR constant:*/
#include <errno.h>
/*:[15oct2004 mt]*/
#endif
/*:[13jul2005 mt]*/

#ifdef UFILES

FILES TheStdout = { 1 };
FILES *Ustdout = &TheStdout;

#ifdef GPP
extern "C" open();
extern "C" close();
extern "C" read();
extern "C" write();
extern "C" lseek();
#endif

/*
    #] Includes :
    #[ Uopen :
*/

FILES *Uopen(char *filename, char *mode)
{
    FILES *f = (FILES *)Malloc1(sizeof(FILES),"Uopen");
    int flags = 0, rights = 0644;
    while ( *mode ) {
        if      ( *mode == 'r' ) { flags |= O_RDONLY; }
        else if ( *mode == 'w' ) { flags |= O_CREAT | O_TRUNC; }
        else if ( *mode == 'a' ) { flags |= O_APPEND; }
        else if ( *mode == 'b' ) { }
        else if ( *mode == '+' ) { flags |= O_RDWR; }
        mode++;
    }
    f->descriptor = open(filename,flags,rights);
    if ( f->descriptor >= 0 ) return(f);
    if ( ( flags & O_APPEND ) != 0 ) {
        flags |= O_CREAT;
        f->descriptor = open(filename,flags,rights);
        if ( f->descriptor >= 0 ) return(f);
    }
    M_free(f,"Uopen");
    return(0);
}

/*
    #] Uopen :
    #[ Uclose :
*/

```

```c
int Uclose(FILES *f)
{
    int retval;
    if ( f ) {
        retval = close(f->descriptor);
        M_free(f,"Uclose");
        return(retval);
    }
    return(EOF);
}

/*
    #] Uclose :
    #[ Uread :
*/


size_t Uread(char *ptr, size_t size, size_t nobj, FILES *f)
{
/*[13jul2005 mt]:*/
#ifdef SAFESIGNAL

/*[15oct2004 mt]:*/
/*Operation read() can be interrupted by a signal. Note, this is rather unlikely,
so we do not save size*nobj for future attempts*/
size_t ret;
    /*If the syscall is interrupted by a signal before it
        succeeded in getting any progress, it must be repeated:*/
    while( ( (ret=read(f->descriptor,ptr,size*nobj))<1)&&(errno == EINTR) );
    return(ret);
#else
#ifdef DEEPDEBUG
    {
        POSITION pos;
        SETBASEPOSITION(pos,lseek(f->descriptor,0L,SEEK_CUR));
        MesPrint("handle %d: reading %ld bytes from position %p\n",f->descriptor,size*nobj,pos);
    }
#endif

    return(read(f->descriptor,ptr,size*nobj));

#endif
/*:[15oct2004 mt]*/
/*:[13jul2005 mt]*/
}

/*
    #] Uread :
    #[ Uwrite :
*/

size_t Uwrite(char *ptr, size_t size, size_t nobj, FILES *f)
{
/*[13jul2005 mt]:*/
#ifdef SAFESIGNAL
/*[15oct2004 mt]:*/
/*Operation write() can be interrupted by a signal. */
size_t ret;
size_t thesize=size*nobj;
    /*If the syscall is interrupted by a signal before it
        succeeded in getting any progress, it must be repeated:*/
    while( ( (ret=write(f->descriptor,ptr,thesize))<1 )&&(errno == EINTR) );

    return(ret);
#else
#ifdef DEEPDEBUG
    {
        POSITION pos;
        SETBASEPOSITION(pos,lseek(f->descriptor,0L,SEEK_CUR));
        MesPrint("handle %d: writing %ld bytes to position %p\n",f->descriptor,size*nobj,pos);
    }
#endif
    return(write(f->descriptor,ptr,size*nobj));

/*:[15oct2004 mt]*/
#endif
/*:[13jul2005 mt]*/
}

/*
    #] Uwrite :
    #[ Useek :
*/

int Useek(FILES *f, off_t offset, int origin)
{
    if ( f && ( lseek(f->descriptor,offset,origin) >= 0 ) ) return(0);
```

```
    return(-1);
}

/*
    #] Useek :
    #[ Utell :
*/

off_t Utell(FILES *f)
{
    if ( f ) return((off_t)lseek(f->descriptor,0L,SEEK_CUR));
    else return(-1);
}

/*
    #] Utell :
    #[ Uflush :
*/

void Uflush(FILES *f) { DUMMYUSE(f); }

/*
    #] Uflush :
    #[ Ugetpos :
*/

int Ugetpos(FILES *f, fpos_t *ptr)
{
    DUMMYUSE(f); DUMMYUSE(ptr);
    return(-1);
}

/*
    #] Ugetpos :
    #[ Usetpos :
*/

int Usetpos(FILES *f,fpos_t *ptr)
{
    DUMMYUSE(f); DUMMYUSE(ptr);
    return(-1);
}

/*
    #] Usetpos :
    #[ Usetbuf :
*/

void Usetbuf(FILES *f, char *ptr) { DUMMYUSE(f); DUMMYUSE(ptr); }

/*
    #] Usetbuf :
*/
#endif
```

4.126 variable.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define chartype FG.cTable
- #define Procedures ((PROCEDURE *)(AP.ProcList.lijst))
- #define NumProcedures AP.ProcList.num
- #define MaxProcedures AP.ProcList.maxnum
- #define DoLoops ((DOLOOP *)(AP.LoopList.lijst))
- #define NumDoLoops AP.LoopList.num

- #define MaxDoLoops AP.LoopList.maxnum
- #define PreVar ((PREVAR ∗)(AP.PreVarList.lijst))
- #define NumPre AP.PreVarList.num
- #define MaxNumPre AP.PreVarList.maxnum
- #define SetElements ((WORD ∗)(AC.SetElementList.lijst))
- #define Sets ((SETS)(AC.SetList.lijst))
- #define functions ((FUNCTIONS)(AC.FunctionList.lijst))
- #define indices ((INDICES)(AC.IndexList.lijst))
- #define symbols ((SYMBOLS)(AC.SymbolList.lijst))
- #define vectors ((VECTORS)(AC.VectorList.lijst))
- #define tablebases ((DBASE ∗)(AC.TableBaseList.lijst))
- #define NumFunctions AC.FunctionList.num
- #define NumIndices AC.IndexList.num
- #define NumSymbols AC.SymbolList.num
- #define NumVectors AC.VectorList.num
- #define NumSets AC.SetList.num
- #define NumSetElements AC.SetElementList.num
- #define NumTableBases AC.TableBaseList.num
- #define GlobalFunctions AC.FunctionList.numglobal
- #define GlobalIndices AC.IndexList.numglobal
- #define GlobalSymbols AC.SymbolList.numglobal
- #define GlobalVectors AC.VectorList.numglobal
- #define GlobalSets AC.SetList.numglobal
- #define GlobalSetElements AC.SetElementList.numglobal
- #define cbuf ((CBUF ∗)(AC.cbufList.lijst))
- #define channels ((CHANNEL ∗)(AC.ChannelList.lijst))
- #define NumOutputChannels AC.ChannelList.num
- #define Dollars ((DOLLARS)(AP.DollarList.lijst))
- #define NumDollars AP.DollarList.num
- #define Dubious ((DUBIOUSV)(AC.DubiousList.lijst))
- #define NumDubious AC.DubiousList.num
- #define Expressions ((EXPRESSIONS)(AC.ExpressionList.lijst))
- #define NumExpressions AC.ExpressionList.num
- #define autofunctions ((FUNCTIONS)(AC.AutoFunctionList.lijst))
- #define autoindices ((INDICES)(AC.AutoIndexList.lijst))
- #define autosymbols ((SYMBOLS)(AC.AutoSymbolList.lijst))
- #define autovectors ((VECTORS)(AC.AutoVectorList.lijst))
- #define xsymbol (cbuf[AM.sbufnum].rhs)
- #define numxsymbol (cbuf[AM.sbufnum].numrhs)
- #define PotModdollars ((WORD ∗)(AC.PotModDolList.lijst))
- #define NumPotModdollars AC.PotModDolList.num
- #define ModOptdollars ((MODOPTDOLLAR ∗)(AC.ModOptDolList.lijst))
- #define NumModOptdollars AC.ModOptDolList.num
- #define BUG A.bug;
- #define AC A.C
- #define AM A.M
- #define AN A.N
- #define AO A.O
- #define AP A.P
- #define AR A.R
- #define AS A.S
- #define AT A.T
- #define AX A.X

Variables

- WRITEBUFTOEXTCHANNEL **writeBufToExtChannel**
- GETCFROMEXTCHANNEL **getcFromExtChannel**
- SETTERMINATORFOREXTERNALCHANNEL **setTerminatorForExternalChannel**
- SETKILLMODEFOREXTERNALCHANNEL **setKillModeForExternalChannel**
- WRITEFILE WriteFile
- ALLGLOBALS A
- FIXEDGLOBALS FG
- FIXEDSET fixedsets [ ]
- char * setupfilename

4.126.1 Detailed Description

Contains a number of defines to make the coding easier. Especially the defines for the use of the lists are very nice. And of course the AC for A.C and AT for either A.T of B->T are indispensible to keep FORM and TFORM in one set of sources.

Definition in file variable.h.

4.126.2 Macro Definition Documentation

4.126.2.1 chartype

```
#define chartype FG.cTable
```

Definition at line 77 of file variable.h.

4.126.2.2 Procedures

```
#define Procedures ((PROCEDURE *)(AP.ProcList.lijst))
```

Definition at line 79 of file variable.h.

4.126.2.3 NumProcedures

```
#define NumProcedures AP.ProcList.num
```

Definition at line 80 of file variable.h.

4.126.2.4 MaxProcedures

```
#define MaxProcedures AP.ProcList.maxnum
```

Definition at line 81 of file variable.h.

4.126.2.5 DoLoops

```
#define DoLoops ((DOLOOP *)(AP.LoopList.lijst))
```

Definition at line 82 of file variable.h.

4.126.2.6 NumDoLoops

```
#define NumDoLoops AP.LoopList.num
```

Definition at line 83 of file variable.h.

4.126.2.7 MaxDoLoops

```
#define MaxDoLoops AP.LoopList.maxnum
```

Definition at line 84 of file variable.h.

4.126.2.8 PreVar

```
#define PreVar ((PREVAR *)(AP.PreVarList.lijst))
```

Definition at line 85 of file variable.h.

4.126.2.9 NumPre

```
#define NumPre AP.PreVarList.num
```

Definition at line 86 of file variable.h.

4.126.2.10 MaxNumPre

```
#define MaxNumPre AP.PreVarList.maxnum
```

Definition at line 87 of file variable.h.

4.126.2.11 SetElements

```
#define SetElements ((WORD *)(AC.SetElementList.lijst))
```

Definition at line 88 of file variable.h.

4.126.2.12 Sets

```
#define Sets ((SETS)(AC.SetList.lijst))
```

Definition at line 89 of file variable.h.

4.126.2.13 functions

```
#define functions ((FUNCTIONS)(AC.FunctionList.lijst))
```

Definition at line 90 of file variable.h.

4.126.2.14 indices

```
#define indices ((INDICES)(AC.IndexList.lijst))
```

Definition at line 91 of file variable.h.

4.126.2.15 symbols

```
#define symbols ((SYMBOLS)(AC.SymbolList.lijst))
```

Definition at line 92 of file variable.h.

4.126.2.16 vectors

```
#define vectors ((VECTORS)(AC.VectorList.lijst))
```

Definition at line 93 of file variable.h.

4.126.2.17 tablebases

```
#define tablebases ((DBASE *)(AC.TableBaseList.lijst))
```

Definition at line 94 of file variable.h.

4.126.2.18 NumFunctions

```
#define NumFunctions AC.FunctionList.num
```

Definition at line 95 of file variable.h.

4.126.2.19 NumIndices

```
#define NumIndices AC.IndexList.num
```

Definition at line 96 of file variable.h.

4.126.2.20 NumSymbols

```
#define NumSymbols AC.SymbolList.num
```

Definition at line 97 of file variable.h.

4.126.2.21 NumVectors

```
#define NumVectors AC.VectorList.num
```

Definition at line 98 of file variable.h.

4.126.2.22 NumSets

```
#define NumSets AC.SetList.num
```

Definition at line 99 of file variable.h.

4.126.2.23 NumSetElements

```
#define NumSetElements AC.SetElementList.num
```

Definition at line 100 of file variable.h.

4.126.2.24 NumTableBases

```
#define NumTableBases AC.TableBaseList.num
```

Definition at line 101 of file variable.h.

4.126.2.25 GlobalFunctions

```
#define GlobalFunctions AC.FunctionList.numglobal
```

Definition at line 102 of file variable.h.

4.126.2.26 GlobalIndices

```
#define GlobalIndices AC.IndexList.numglobal
```

Definition at line 103 of file variable.h.

4.126.2.27 GlobalSymbols

```
#define GlobalSymbols AC.SymbolList.numglobal
```

Definition at line 104 of file variable.h.

4.126.2.28 GlobalVectors

```
#define GlobalVectors AC.VectorList.numglobal
```

Definition at line 105 of file variable.h.

4.126.2.29 GlobalSets

```
#define GlobalSets AC.SetList.numglobal
```

Definition at line 106 of file variable.h.

4.126.2.30 GlobalSetElements

```
#define GlobalSetElements AC.SetElementList.numglobal
```

Definition at line 107 of file variable.h.

4.126.2.31 cbuf

```
#define cbuf ((CBUF *)(AC.cbufList.lijst))
```

Definition at line 108 of file variable.h.

4.126.2.32 channels

```
#define channels ((CHANNEL *)(AC.ChannelList.lijst))
```

Definition at line 109 of file variable.h.

4.126.2.33 NumOutputChannels

```
#define NumOutputChannels AC.ChannelList.num
```

Definition at line 110 of file variable.h.

4.126.2.34 Dollars

```
#define Dollars ((DOLLARS)(AP.DollarList.lijst))
```

Definition at line 111 of file variable.h.

4.126.2.35 NumDollars

```
#define NumDollars AP.DollarList.num
```

Definition at line 112 of file variable.h.

4.126.2.36 Dubious

```
#define Dubious ((DUBIOUSV)(AC.DubiousList.lijst))
```

Definition at line 113 of file variable.h.

4.126.2.37 NumDubious

```
#define NumDubious AC.DubiousList.num
```

Definition at line 114 of file variable.h.

4.126.2.38 Expressions

```
#define Expressions ((EXPRESSIONS)(AC.ExpressionList.lijst))
```

Definition at line 115 of file variable.h.

4.126.2.39 NumExpressions

```
#define NumExpressions AC.ExpressionList.num
```

Definition at line 116 of file variable.h.

4.126.2.40 autofunctions

```
#define autofunctions ((FUNCTIONS)(AC.AutoFunctionList.lijst))
```

Definition at line 117 of file variable.h.

4.126.2.41 autoindices

```
#define autoindices ((INDICES)(AC.AutoIndexList.lijst))
```

Definition at line 118 of file variable.h.

4.126.2.42 autosymbols

```
#define autosymbols ((SYMBOLS)(AC.AutoSymbolList.lijst))
```

Definition at line 119 of file variable.h.

4.126.2.43 autovectors

```
#define autovectors ((VECTORS)(AC.AutoVectorList.lijst))
```

Definition at line 120 of file variable.h.

4.126.2.44 xsymbol

```
#define xsymbol (cbuf[AM.sbufnum].rhs)
```

Definition at line 121 of file variable.h.

4.126.2.45 numxsymbol

```
#define numxsymbol (cbuf[AM.sbufnum].numrhs)
```

Definition at line 122 of file variable.h.

4.126.2.46 PotModdollars

```
#define PotModdollars ((WORD *)(AC.PotModDolList.lijst))
```

Definition at line 124 of file variable.h.

4.126.2.47 NumPotModdollars

```
#define NumPotModdollars AC.PotModDolList.num
```

Definition at line 125 of file variable.h.

4.126.2.48 ModOptdollars

```
#define ModOptdollars ((MODOPTDOLLAR *)(AC.ModOptDolList.lijst))
```

Definition at line 126 of file variable.h.

4.126.2.49 NumModOptdollars

```
#define NumModOptdollars AC.ModOptDolList.num
```

Definition at line 127 of file variable.h.

4.126.2.50 BUG

```
#define BUG A.bug;
```

Definition at line 129 of file variable.h.

4.126.2.51 AC

```
#define AC A.C
```

Definition at line 145 of file variable.h.

4.126.2.52 AM

```
#define AM A.M
```

Definition at line 146 of file variable.h.

4.126.2.53 AN

```
#define AN A.N
```

Definition at line 147 of file variable.h.

4.126.2.54 AO

```
#define AO A.O
```

Definition at line 148 of file variable.h.

4.126.2.55 AP

```
#define AP A.P
```

Definition at line 149 of file variable.h.

4.126.2.56 AR

```
#define AR A.R
```

Definition at line 150 of file variable.h.

4.126.2.57 AS

```
#define AS A.S
```

Definition at line 151 of file variable.h.

4.126.2.58 AT

```
#define AT A.T
```

Definition at line 152 of file variable.h.

4.126.2.59 AX

```
#define AX A.X
```

Definition at line 153 of file variable.h.

4.126.3 Variable Documentation

4.126.3.1 WriteFile

```
WRITEFILE WriteFile [extern]
```

Definition at line 1285 of file tools.c.

4.126.3.2 A

```
ALLGLOBALS A [extern]
```

Definition at line 131 of file inivar.h.

4.126.3.3 FG

```
FIXEDGLOBALS FG [extern]
```

Definition at line 34 of file inivar.h.

4.126.3.4 fixedsets

```
FIXEDSET fixedsets[] [extern]
```

Definition at line 236 of file inivar.h.

4.126.3.5 setupfilename

```
char* setupfilename [extern]
```

Definition at line 254 of file inivar.h.

4.127 variable.h

Go to the documentation of this file.

```
#ifndef __VARIABLE__

#define __VARIABLE__

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */

/*See the file extcmd.c*/

#ifdef REMOVEDBY_MT
extern int (*writeBufToExtChannel)(char *buf, size_t n);
extern int (*getcFromExtChannel)();
extern int (*setTerminatorForExternalChannel)(char *newterminator);
#endif
extern WRITEBUFTOEXTCHANNEL writeBufToExtChannel;
extern GETCFROMEXTCHANNEL getcFromExtChannel;
extern SETTERMINATORFOREXTERNALCHANNEL setTerminatorForExternalChannel;
extern SETKILLMODEFOREXTERNALCHANNEL setKillModeForExternalChannel;

/*
extern LONG (*WriteFile)(int handle, UBYTE *buffer, LONG number);
*/
extern WRITEFILE WriteFile;
/*:[17nov2005 mt]*/

extern ALLGLOBALS A;
#ifdef WITHPTHREADS
extern ALLPRIVATES **AB;
#endif

extern FIXEDGLOBALS FG;
extern FIXEDSET fixedsets[];

extern char *setupfilename;

EXTERNLOCK(ErrorMessageLock)
EXTERNLOCK(FileReadLock)
EXTERNLOCK(dummylock)

#ifdef VMS
#include <stdio.h>
extern FILE **FileStructs;
#endif

#define chartype FG.cTable

#define Procedures ((PROCEDURE *)(AP.ProcList.lijst))
#define NumProcedures AP.ProcList.num
#define MaxProcedures AP.ProcList.maxnum
#define DoLoops ((DOLOOP *)(AP.LoopList.lijst))
#define NumDoLoops AP.LoopList.num
#define MaxDoLoops AP.LoopList.maxnum
#define PreVar ((PREVAR *)(AP.PreVarList.lijst))
#define NumPre AP.PreVarList.num
#define MaxNumPre AP.PreVarList.maxnum
#define SetElements ((WORD *)(AC.SetElementList.lijst))
#define Sets ((SETS)(AC.SetList.lijst))
#define functions ((FUNCTIONS)(AC.FunctionList.lijst))
```

```
#define indices ((INDICES)(AC.IndexList.lijst))
#define symbols ((SYMBOLS)(AC.SymbolList.lijst))
#define vectors ((VECTORS)(AC.VectorList.lijst))
#define tablebases ((DBASE *)(AC.TableBaseList.lijst))
#define NumFunctions AC.FunctionList.num
#define NumIndices AC.IndexList.num
#define NumSymbols AC.SymbolList.num
#define NumVectors AC.VectorList.num
#define NumSets AC.SetList.num
#define NumSetElements AC.SetElementList.num
#define NumTableBases AC.TableBaseList.num
#define GlobalFunctions AC.FunctionList.numglobal
#define GlobalIndices AC.IndexList.numglobal
#define GlobalSymbols AC.SymbolList.numglobal
#define GlobalVectors AC.VectorList.numglobal
#define GlobalSets AC.SetList.numglobal
#define GlobalSetElements AC.SetElementList.numglobal
#define cbuf ((CBUF *)(AC.cbufList.lijst))
#define channels ((CHANNEL *)(AC.ChannelList.lijst))
#define NumOutputChannels AC.ChannelList.num
#define Dollars ((DOLLARS)(AP.DollarList.lijst))
#define NumDollars AP.DollarList.num
#define Dubious ((DUBIOUSV)(AC.DubiousList.lijst))
#define NumDubious AC.DubiousList.num
#define Expressions ((EXPRESSIONS)(AC.ExpressionList.lijst))
#define NumExpressions AC.ExpressionList.num
#define autofunctions ((FUNCTIONS)(AC.AutoFunctionList.lijst))
#define autoindices ((INDICES)(AC.AutoIndexList.lijst))
#define autosymbols ((SYMBOLS)(AC.AutoSymbolList.lijst))
#define autovectors ((VECTORS)(AC.AutoVectorList.lijst))
#define xsymbol (cbuf[AM.sbufnum].rhs)
#define numxsymbol (cbuf[AM.sbufnum].numrhs)

#define PotModdollars ((WORD *)(AC.PotModDolList.lijst))
#define NumPotModdollars AC.PotModDolList.num
#define ModOptdollars ((MODOPTDOLLAR *)(AC.ModOptDolList.lijst))
#define NumModOptdollars AC.ModOptDolList.num

#define BUG A.bug;

#ifdef WITHPTHREADS
#define AC A.C
#define AM A.M
#define AO A.O
#define AP A.P
#define AS A.S
#define AX A.X
#define AN B->N
#define AR B->R
#define AT B->T
#define AN0 B0->N
#define AR0 B0->R
#define AT0 B0->T
#else
#define AC A.C
#define AM A.M
#define AN A.N
#define AO A.O
#define AP A.P
#define AR A.R
#define AS A.S
#define AT A.T
#define AX A.X
#endif

#endif
```

4.128 vector.h File Reference

```
#include <stddef.h>
#include <string.h>
#include "declare.h"
```

Include dependency graph for vector.h:

This graph shows which files directly or indirectly include this file:

Macros

- #define VectorStruct(T)
- #define Vector(T, X) VectorStruct(T) X = { NULL, 0, 0 }
- #define DeclareVector(T, X) VectorStruct(T) X
- #define VectorInit(X)
- #define VectorFree(X)
- #define VectorPtr(X) ((X).ptr)
- #define VectorFront(X) ((X).ptr[0])
- #define VectorBack(X) ((X).ptr[(X).size - 1])
- #define VectorSize(X) ((X).size)
- #define VectorCapacity(X) ((X).capacity)
- #define VectorEmpty(X) ((X).size == 0)
- #define VectorClear(X) do { (X).size = 0; } while (0)
- #define VectorReserve(X, newcapacity)
- #define VectorPushBack(X, x)
- #define VectorPushBacks(X, src, n)
- #define VectorPopBack(X) do { (X).size --; } while (0)
- #define VectorInsert(X, index, x)
- #define VectorInserts(X, index, src, n)
- #define VectorErase(X, index)
- #define VectorErases(X, index, n)

4.128.1 Detailed Description

An implementation of dynamic array.

Example:
```
size_t i;
Vector(int, vec);
VectorPushBack(vec, 1);
VectorPushBack(vec, 2);
VectorPushBack(vec, 3);
for ( i = 0; i < VectorSize(vec); i++ )
  printf("%d\n", VectorPtr(vec)[i]);
VectorFree(vec);
```

Definition in file vector.h.

4.128.2 Macro Definition Documentation

4.128.2.1 VectorStruct

```
#define VectorStruct(
                T )
```

Value:
```
    struct { \
        T *ptr; \
        size_t size; \
        size_t capacity; \
    }
```

A struct for vector objects.

Parameters

T	the type of elements.

Definition at line 65 of file vector.h.

4.128.2.2 Vector

```
#define Vector(
                T,
                X )  VectorStruct(T) X = { NULL, 0, 0 }
```

Defines and initialises a vector X of the type T. The user must call VectorFree(X) after the use of X.

Parameters

T	the type of elements.
X	the name of vector object

Definition at line 84 of file vector.h.

4.128.2.3 DeclareVector

```
#define DeclareVector(
            T,
            X )  VectorStruct(T) X
```

Declares a vector X of the type T. The user must call VectorInit(X) before the use of X.

Parameters

T	the type of elements.
X	the name of vector object

Definition at line 99 of file vector.h.

4.128.2.4 VectorInit

```
#define VectorInit(
            X )
```

Value:

```
do { \
    (X).ptr = NULL; \
    (X).size = 0; \
    (X).capacity = 0; \
} while (0)
```

Initialises a vector X of the type T. The user must call VectorFree(X) after the use of X.

Parameters

X	the vector object.

Definition at line 113 of file vector.h.

4.128.2.5 VectorFree

```
#define VectorFree(
            X )
```

Value:

```
do { \
    M_free((X).ptr, "VectorFree:" #X); \
    (X).ptr = NULL; \
    (X).size = 0; \
    (X).capacity = 0; \
} while (0)
```

Frees the buffer allocated by the vector X.

Parameters

X	the vector object.

Definition at line 130 of file vector.h.

4.128.2.6 VectorPtr

```
#define VectorPtr(
            X )  ((X).ptr)
```

Returns the pointer to the buffer for the vector X. NULL when VectorCapacity(X) == 0.

Parameters

X	the vector object.

Returns

the pointer to the allocated buffer for the vector.

Definition at line 150 of file vector.h.

4.128.2.7 VectorFront

```
#define VectorFront(
            X )  ((X).ptr[0])
```

Returns the first element of the vector X. Undefined when VectorSize(X) == 0.

Parameters

X	the vector object.

Returns

the first element of the vector.

Definition at line 165 of file vector.h.

4.128.2.8 VectorBack

```
#define VectorBack(
            X )  ((X).ptr[(X).size - 1])
```

Returns the last element of the vector X. Undefined when VectorSize(X) == 0.

Parameters

X	the vector object.

Returns

the last element of the vector.

Definition at line 180 of file vector.h.

4.128.2.9 VectorSize

```
#define VectorSize(
            X )  ((X).size)
```

Returns the size of the vector X.

Parameters

X	the vector object.

Returns

the size of the vetor.

Definition at line 194 of file vector.h.

4.128.2.10 VectorCapacity

```
#define VectorCapacity(
            X )  ((X).capacity)
```

Returns the capacity (the number of the already allocated elements) of the vector X.

Parameters

X	the vector object.

Returns

the capacity of the vetor.

Definition at line 208 of file vector.h.

4.128.2.11 VectorEmpty

```
#define VectorEmpty(
            X )  ((X).size == 0)
```

Returns true the size of the vector X is zero.

Parameters

X	the vector object.

Returns

true if the vector has no elements, false otherwise.

Definition at line 222 of file vector.h.

4.128.2.12 VectorClear

```
#define VectorClear(
            X )  do { (X).size = 0; } while (0)
```

Sets the size of the vector X to zero.

Parameters

X	the vector object.

Definition at line 235 of file vector.h.

4.128.2.13 VectorReserve

```
#define VectorReserve(
            X,
            newcapacity )
```

Value:

```
do { \
    size_t v_tmp_newcapacity_ = (newcapacity); \
    if ( (X).capacity < v_tmp_newcapacity_ ) { \
        void *v_tmp_newptr_; \
        v_tmp_newcapacity_ = (v_tmp_newcapacity_ * 3) / 2; \
        if ( v_tmp_newcapacity_ < 4 ) v_tmp_newcapacity_ = 4; \
        v_tmp_newptr_ = Malloc1(sizeof((X).ptr[0]) * v_tmp_newcapacity_, "VectorReserve:" #X); \
        if ( (X).ptr != NULL ) { \
            memcpy(v_tmp_newptr_, (X).ptr, (X).size * sizeof((X).ptr[0])); \
            M_free((X).ptr, "VectorReserve:" #X); \
        } \
        (X).ptr = v_tmp_newptr_; \
        (X).capacity = v_tmp_newcapacity_; \
    } \
} while (0)
```

Requires that the capacity of the vector X is equal to or lager than newcapacity.

Parameters

X	the vector object.
newcapacity	the capacity to be reserved.

Definition at line 249 of file vector.h.

4.128.2.14 VectorPushBack

```
#define VectorPushBack(
            X,
            x )
```

Value:

```
do { \
    VectorReserve((X), (X).size + 1); \
    (X).ptr[(X).size++] = (x); \
} while (0)
```

Adds an element x at the end of the vector X.

Parameters

X	the vector object.
x	the element to be added.

Definition at line 277 of file vector.h.

4.128.2.15 VectorPushBacks

```
#define VectorPushBacks(
            X,
            src,
            n )
```

Value:

```
do { \
    size_t v_tmp_n_ = (n); \
    VectorReserve((X), (X).size + v_tmp_n_); \
    memcpy((X).ptr + (X).size, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
    (X).size += v_tmp_n_; \
} while (0)
```

Adds an n elements of src at the end of the vector X.

Parameters

X	the vector object.
src	the starting address of the buffer storing elements to be added.
n	the number of elements to be added.

Definition at line 295 of file vector.h.

4.128.2.16 VectorPopBack

```
#define VectorPopBack(
            X )  do { (X).size --; } while (0)
```

Removes the last element of the vector X. VectorSize(X) must be > 0.

Parameters

X	the vector object.

Definition at line 314 of file vector.h.

4.128.2.17 VectorInsert

```
#define VectorInsert(
            X,
            index,
            x )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index); \
    VectorReserve((X), (X).size + 1); \
    memmove((X).ptr + v_tmp_index_ + 1, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
  sizeof((X).ptr[0])); \
    (X).ptr[v_tmp_index_] = (x); \
    (X).size++; \
} while (0)
```

Inserts an element x at the specified index of the vector X. The index must be 0 <= index < VectorSize(X).

Parameters

X	the vector object.
index	the position at which the element will be inserted.
x	the element to be inserted.

Definition at line 330 of file vector.h.

4.128.2.18 VectorInserts

```
#define VectorInserts(
            X,
            index,
            src,
            n )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
    VectorReserve((X), (X).size + v_tmp_n_); \
    memmove((X).ptr + v_tmp_index_ + v_tmp_n_, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
  sizeof((X).ptr[0])); \
    memcpy((X).ptr + v_tmp_index_, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
    (X).size += v_tmp_n_; \
} while (0)
```

Inserts an n elements of src at the specified index of the vector X. The index must be 0 <= index < VectorSize(X).

Parameters

X	the vector object.
index	the position at which the elements will be inserted.
src	the starting address of the buffer storing elements to be inserted.
n	the number of elements to be inserted.

Definition at line 353 of file vector.h.

4.128.2.19 VectorErase

```
#define VectorErase(
            X,
            index )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index); \
    memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + 1, ((X).size - v_tmp_index_ - 1) *
  sizeof((X).ptr[0])); \
    (X).size--; \
} while (0)
```

Removes an element at the specified index of the vector X. The index must be 0 <= index < VectorSize(X).

Parameters

X	the vector object.
index	the position of the element to be removed.

Definition at line 374 of file vector.h.

4.128.2.20 VectorErases

```
#define VectorErases(
            X,
            index,
            n )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
    memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + v_tmp_n_, ((X).size - v_tmp_index_ - 1) *
  sizeof((X).ptr[0])); \
    (X).size -= v_tmp_n_; \
} while (0)
```

Removes an n elements at the specified index of the vector X. The index must be 0 <= index < VectorSize(X) - n + 1.

Parameters

X	the vector object.
index	the starting position of the elements to be removed.
n	the number of elements to be removed.

Definition at line 394 of file vector.h.

4.129 vector.h

Go to the documentation of this file.

```
#ifndef VECTOR_H_
#define VECTOR_H_

/* #[ License : */
/*
 *   Copyright (C) 1984-2017 J.A.M. Vermaseren
 *   When using this file you are requested to refer to the publication
 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
 *   This is considered a matter of courtesy as the development was paid
 *   for by FOM the Dutch physics granting agency and we would like to
 *   be able to track its scientific use to convince FOM of its value
 *   for the community.
 *
 *   This file is part of FORM.
 *
 *   FORM is free software: you can redistribute it and/or modify it under the
 *   terms of the GNU General Public License as published by the Free Software
 *   Foundation, either version 3 of the License, or (at your option) any later
 *   version.
 *
 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 *   details.
 *
 *   You should have received a copy of the GNU General Public License along
 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
 */
/* #] License : */
/*
    #[ Includes :
*/

#include <stddef.h>
#include <string.h>
#include "declare.h"

/*
    #] Includes :
    #[ Vector :
        #[ VectorStruct :
*/

#define VectorStruct(T) \
    struct { \
        T *ptr; \
        size_t size; \
        size_t capacity; \
    }

/*
        #] VectorStruct :
        #[ Vector :
*/

#define Vector(T, X) \
    VectorStruct(T) X = { NULL, 0, 0 }

/*
        #] Vector :
        #[ DeclareVector :
*/

#define DeclareVector(T, X) \
    VectorStruct(T) X

/*
        #] DeclareVector :
        #[ VectorInit :
*/

#define VectorInit(X) \
    do { \
        (X).ptr = NULL; \
        (X).size = 0; \
        (X).capacity = 0; \
    } while (0)

/*
        #] VectorInit :
        #[ VectorFree :
*/

#define VectorFree(X) \
    do { \
        M_free((X).ptr, "VectorFree:" #X); \
        (X).ptr = NULL; \
```

```
        (X).size = 0; \
        (X).capacity = 0; \
    } while (0)

/*
        #] VectorFree :
        #[ VectorPtr :
*/

#define VectorPtr(X) \
    ((X).ptr)

/*
        #] VectorPtr :
        #[ VectorFront :
*/

#define VectorFront(X) \
    ((X).ptr[0])

/*
        #] VectorFront :
        #[ VectorBack :
*/

#define VectorBack(X) \
    ((X).ptr[(X).size - 1])

/*
        #] VectorBack :
        #[ VectorSize :
*/

#define VectorSize(X) \
    ((X).size)

/*
        #] VectorSize :
        #[ VectorCapacity :
*/

#define VectorCapacity(X) \
    ((X).capacity)

/*
        #] VectorCapacity :
        #[ VectorEmpty :
*/

#define VectorEmpty(X) \
    ((X).size == 0)

/*
        #] VectorEmpty :
        #[ VectorClear :
*/

#define VectorClear(X) \
    do { (X).size = 0; } while (0)

/*
        #] VectorClear :
        #[ VectorReserve :
*/

#define VectorReserve(X, newcapacity) \
    do { \
        size_t v_tmp_newcapacity_ = (newcapacity); \
        if ( (X).capacity < v_tmp_newcapacity_ ) { \
            void *v_tmp_newptr_; \
            v_tmp_newcapacity_ = (v_tmp_newcapacity_ * 3) / 2; \
            if ( v_tmp_newcapacity_ < 4 ) v_tmp_newcapacity_ = 4; \
            v_tmp_newptr_ = Malloc1(sizeof((X).ptr[0]) * v_tmp_newcapacity_, "VectorReserve:" #X); \
            if ( (X).ptr != NULL ) { \
                memcpy(v_tmp_newptr_, (X).ptr, (X).size * sizeof((X).ptr[0])); \
                M_free((X).ptr, "VectorReserve:" #X); \
            } \
            (X).ptr = v_tmp_newptr_; \
            (X).capacity = v_tmp_newcapacity_; \
        } \
    } while (0)

/*
        #] VectorReserve :
        #[ VectorPushBack :
*/

```

```
#define VectorPushBack(X, x) \
    do { \
        VectorReserve((X), (X).size + 1); \
        (X).ptr[(X).size++] = (x); \
    } while (0)

/*
        #] VectorPushBack :
        #[ VectorPushBacks :
*/

#define VectorPushBacks(X, src, n) \
    do { \
        size_t v_tmp_n_ = (n); \
        VectorReserve((X), (X).size + v_tmp_n_); \
        memcpy((X).ptr + (X).size, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
        (X).size += v_tmp_n_; \
    } while (0)

/*
        #] VectorPushBacks :
        #[ VectorPopBack :
*/

#define VectorPopBack(X) \
    do { (X).size --; } while (0)

/*
        #] VectorPopBack :
        #[ VectorInsert :
*/

#define VectorInsert(X, index, x) \
    do { \
        size_t v_tmp_index_ = (index); \
        VectorReserve((X), (X).size + 1); \
        memmove((X).ptr + v_tmp_index_ + 1, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) * sizeof((X).ptr[0])); \
        (X).ptr[v_tmp_index_] = (x); \
        (X).size++; \
    } while (0)

/*
        #] VectorInsert :
        #[ VectorInserts :
*/

#define VectorInserts(X, index, src, n) \
    do { \
        size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
        VectorReserve((X), (X).size + v_tmp_n_); \
        memmove((X).ptr + v_tmp_index_ + v_tmp_n_, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) * sizeof((X).ptr[0])); \
        memcpy((X).ptr + v_tmp_index_, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
        (X).size += v_tmp_n_; \
    } while (0)

/*
        #] VectorInserts :
        #[ VectorErase :
*/

#define VectorErase(X, index) \
    do { \
        size_t v_tmp_index_ = (index); \
        memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + 1, ((X).size - v_tmp_index_ - 1) * sizeof((X).ptr[0])); \
        (X).size--; \
    } while (0)

/*
        #] VectorErase :
        #[ VectorErases :
*/

#define VectorErases(X, index, n) \
    do { \
        size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
        memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + v_tmp_n_, ((X).size - v_tmp_index_ - 1) * sizeof((X).ptr[0])); \
        (X).size -= v_tmp_n_; \
    } while (0)

/*
        #] VectorErases :
    #] Vector :
*/
```

```
00405 #endif  /* VECTOR_H_ */
```

4.130 wildcard.c File Reference

`#include "form3.h"`
Include dependency graph for wildcard.c:

Macros

- #define DEBUG(x)

Functions

- WORD WildFill (PHEAD WORD ∗to, WORD ∗from, WORD ∗sub)
- WORD ResolveSet (PHEAD WORD ∗from, WORD ∗to, WORD ∗subs)
- VOID ClearWild (PHEAD0)
- WORD AddWild (PHEAD WORD oldnumber, WORD type, WORD newnumber)
- WORD CheckWild (PHEAD WORD oldnumber, WORD type, WORD newnumber, WORD ∗newval)
- int DenToFunction (WORD ∗term, WORD numfun)

4.130.1 Detailed Description

Contains the functions that deal with the wildcards. During the pattern matching there are two steps: 1: check that a wildcard substitution is correct (if there was already an assignment for this variable, it is the same; it is part of the proper set; it is the proper type of variables, etc.) 2: make the assignment In addition we have to be able to clear assignments. During execution we have to make the actual replacements (WildFill)

Definition in file wildcard.c.

4.130.2 Macro Definition Documentation

4.130.2.1 DEBUG

```
#define DEBUG(
              x )
```

Definition at line 44 of file wildcard.c.

4.130.3 Function Documentation

4.130.3.1 WildFill()

```
WORD WildFill (
            PHEAD WORD * to,
            WORD * from,
            WORD * sub )
```

Definition at line 65 of file wildcard.c.

4.130.3.2 ResolveSet()

```
WORD ResolveSet (
            PHEAD WORD * from,
            WORD * to,
            WORD * subs )
```

Definition at line 1333 of file wildcard.c.

4.130.3.3 ClearWild()

```
VOID ClearWild (
            PHEAD0  )
```

Definition at line 1490 of file wildcard.c.

4.130.3.4 AddWild()

```
WORD AddWild (
            PHEAD WORD oldnumber,
            WORD type,
            WORD newnumber )
```

Definition at line 1516 of file wildcard.c.

4.130.3.5 CheckWild()

```
WORD CheckWild (
            PHEAD WORD oldnumber,
            WORD type,
            WORD newnumber,
            WORD * newval )
```

Definition at line 1761 of file wildcard.c.

4.130.3.6 DenToFunction()

```
int DenToFunction (
            WORD * term,
            WORD numfun )
```

Definition at line 2472 of file wildcard.c.

4.131 wildcard.c

Go to the documentation of this file.

```
00012 /* #[ License : */
00013 /*
00014  *   Copyright (C) 1984-2017 J.A.M. Vermaseren
00015  *   When using this file you are requested to refer to the publication
00016  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00017  *   This is considered a matter of courtesy as the development was paid
00018  *   for by FOM the Dutch physics granting agency and we would like to
00019  *   be able to track its scientific use to convince FOM of its value
00020  *   for the community.
00021  *
00022  *   This file is part of FORM.
00023  *
00024  *   FORM is free software: you can redistribute it and/or modify it under the
00025  *   terms of the GNU General Public License as published by the Free Software
00026  *   Foundation, either version 3 of the License, or (at your option) any later
00027  *   version.
00028  *
00029  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00030  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00031  *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00032  *   details.
00033  *
00034  *   You should have received a copy of the GNU General Public License along
00035  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00036  */
00037 /* #] License : */
00038 /*
00039     #[ Includes : wildcard.c
00040 */
00041
00042 #include "form3.h"
00043
00044 #define DEBUG(x)
00045
00046 /*
00047 #define DEBUG(x) x
00048
00049     #] Includes :
00050     #[ Wildcards :
00051         #[ WildFill :           WORD WildFill(to,from,sub)
00052
00053         Takes the term in from and puts it into to while
00054         making wildcard substitutions.
00055         The return value is the number of words put in to.
00056         The length as the first word of from is not copied.
00057
00058         There are two possible algorithms:
00059         1:  For each element in `from': scan sub.
00060         2:  For each wildcard in sub replace elements in term.
00061         The original algorithm used 1:
00062
00063 */
00064
00065 WORD WildFill(PHEAD WORD *to, WORD *from, WORD *sub)
00066 {
00067     GETBIDENTITY
00068     WORD i, j, *s, *t, *m, len, dflag, odirt, adirt;
00069     WORD *r, *u, *v, *w, *z, *zst, *zz, *subs, *accu, na, dirty = 0, *tstop;
00070     WORD *temp = 0, *uu, *oldcpointer, sgn;
00071     WORD subcount, setflag, *setlist = 0, si;
00072     accu = oldcpointer = AR.CompressPointer;
00073     t = sub;
00074     t += sub[1];
00075     s = sub + SUBEXPSIZE;
00076     i = 0;
```

```
    while ( s < t && *s != FROMBRAC ) {
        i++; s += s[1];
    }
    if ( !i ) {         /* No wildcards -> done quickly */
        j = i = *from;
        NCOPY(to,from,i);
        if ( dirty ) AN.WildDirt = dirty;
        return(j);
    }
    sgn = 0;
    subs = sub + SUBEXPSIZE;
    t = from;
    GETSTOP(t,r);
    t++;
    m = to + 1;
    if ( t < r ) do {
        uu = u = t + t[1];
        setflag = 0;
ReSwitch:
        switch ( *t ) {
            case SYMBOL:
/*
            #[ SYMBOLS :
*/
                z = accu;
                *m++ = *t++;
                *m++ = *t++;
                v = m;
                while ( t < u ) {
                    *m = *t;
                    for ( si = 0; si < setflag; si += 2 ) {
                        if ( t == temp + setlist[si] ) goto sspow;
                    }
                    s = subs;
                    for ( j = 0; j < i; j++ ) {
                        if ( *t == s[2] ) {
                            if ( *s == SYMTOSYM ) {
                                *m = s[3]; dirty = 1;
                                break;
                            }
                            else if ( *s == SYMTONUM ) {
                                dirty = 1;
                                zst = z;
                                *z++ = SNUMBER;
                                *z++ = 4;
                                *z++ = s[3];
                                w = z;
                                *z++ = *++t;
                                if ( ABS(*t) >= 2*MAXPOWER) {
DoPow:                          s = subs;
                                for ( j = 0; j < i; j++ ) {
                                    if ( ( *s == SYMTONUM ) &&
                                    ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
                                        dirty = 1;
                                        *w = s[3];
                                        if ( *t < 0 ) *w = -*w;
                                        break;
                                    }
                                    if ( ( *s == SYMTOSYM ) &&
                                    ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
                                        dirty = 1;
                                        zz = z;
                                        while ( --zz >= zst ) {
                                            zz[1+FUNHEAD+ARGHEAD] = *zz;
                                        }
                                        w += 1+FUNHEAD+ARGHEAD;
                                        *zst = EXPONENT;
                                        zst[2] = DIRTYFLAG;
                                        zst[FUNHEAD+ARGHEAD] = WORDDIF(z,zst)+4;
                                        zst[1+FUNHEAD] = 1;
                                        zst[FUNHEAD] = WORDDIF(z,zst)+4+ARGHEAD;
                                        z += FUNHEAD+ARGHEAD+1;
                                        *w = 1; /* exponent -> 1 */
                                        *z++ = 1;
                                        *z++ = 1;
                                        *z++ = 3;
                                        if ( *t > 0 ) {
                                            *z++ = -SYMBOL;
                                            *z++ = s[3];
                                        }
                                        else {
                                            *z++ = ARGHEAD+8;
                                            *z++ = 1;
                                            *z++ = 8;
                                            *z++ = SYMBOL;
                                            *z++ = 4;
                                            *z++ = s[3];
```

```
                                        *z++ = 1;
                                        *z++ = 1;
                                        *z++ = 1;
                                        *z++ = -3;
                                    }
                                    zst[1] = WORDDIF(z,zst);
                                    break;
                                }
                                if ( *s == SYMTOSUB &&
                                ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
MakeExp:                            dirty = 1;
                                    zz = z;
                                    while ( --zz >= zst ) {
                                        zz[1+FUNHEAD+ARGHEAD] = *zz;
                                    }
                                    w += 1+FUNHEAD+ARGHEAD;
                                    *zst = EXPONENT;
                                    zst[2] = DIRTYFLAG;
                                    zst[FUNHEAD+ARGHEAD] = WORDDIF(z,zst)+4;
                                    zst[1+FUNHEAD] = 1;
                                    zst[FUNHEAD] = WORDDIF(z,zst)+4+ARGHEAD;
                                    z += FUNHEAD+ARGHEAD+1;
                                    *w = 1; /* exponent -> 1 */
                                    *z++ = 1;
                                    *z++ = 1;
                                    *z++ = 3;
                                    *z++ = 4+SUBEXPSIZE+ARGHEAD;
                                    *z++ = 1;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = SUBEXPRESSION;
                                    *z++ = SUBEXPSIZE;
                                    *z++ = s[3];
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = 1;
                                    *z++ = 1;
                                    *z++ = *t > 0 ? 3: -3;
                                    zst[1] = WORDDIF(z,zst);
                                    break;
                                }
                                s += s[1];
                            }
                            }
                            if ( !*w ) z = w - 3;
                            t++;
                            goto Seven;
                        }
                        else if ( *s == SYMTOSUB ) {
                            dirty = 1;
                            zst = z;
                            *z++ = SUBEXPRESSION;
                            *z++ = SUBEXPSIZE;
                            *z++ = s[3];
                            w = z;
                            *z++ = *++t;
                            *z++ = AT.ebufnum;
                            FILLSUB(z)
                            goto DoPow;
                        }
                    }
                    s += s[1];
                }
sspow:          s = subs;
                *++m = *++t;
                for ( si = 0; si < setflag; si += 2 ) {
                    if ( t == temp + setlist[si] ) {
                        t++; m++;
                        goto Seven;
                    }
                }
                for ( j = 0; j < i; j++ ) {
                    if ( ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
                        if ( *s == SYMTONUM ) {
                            dirty = 1;
                            *m = s[3];
                            if ( *t < 0 ) *m = -*m;
                            break;
                        }
                        else if ( *s == SYMTOSYM ) {
                            dirty = 1;
                            *z++ = EXPONENT;
                            if ( *t < 0 ) *z++ = FUNHEAD+ARGHEAD+10;
                            else *z++ = 4+FUNHEAD;
                            *z++ = 0;
                            FILLFUN3(z)
```

```
                                *z++ = -SYMBOL;
                                *z++ = m[-1];
                                if ( *t < 0 ) {
                                    *z++ = ARGHEAD+8;
                                    *z++ = 0;
                                    *z++ = 8;
                                    *z++ = SYMBOL;
                                    *z++ = 4;
                                    *z++ = s[3];
                                    *z++ = 1;
                                    *z++ = 1;
                                    *z++ = 1;
                                    *z = -3;
                                }
                                else {
                                    *z++ = -SYMBOL;
                                    *z++ = s[3];
                                }
                                m -= 2;
                                break;
                            }
                            else if ( *s == SYMTOSUB ) {
                                zst = z;
                                *z++ = SYMBOL;
                                *z++ = 4;
                                *z++ = *--m;
                                w = z;
                                *z++ = *t;
                                goto MakeExp;
                            }
                        }
                        s += s[1];
                    }
                    t++;
                    if ( *m ) m++;
                    else m--;
Seven:;
                }
                j = WORDDIF(m,v);
                if ( !j ) m -= 2;
                else v[-1] = j + 2;
                s = accu;
                while ( s < z ) *m++ = *s++;
                break;
/*
            #] SYMBOLS :
*/
            case DOTPRODUCT:
/*
            #[ DOTPRODUCTS :
*/
                *m++ = *t++;
                *m++ = *t++;
                v = m;
                z = accu;
                while ( t < u ) {
                    *m = *t;
                    subcount = 0;
                    for ( si = 0; si < setflag; si += 2 ) {
                        if ( t == temp + setlist[si] ) goto ss2;
                    }
                    s = subs;
                    for ( j = 0; j < i; j++ ) {
                        if ( *t == s[2] ) {
                            if ( *s == VECTOVEC ) {
                                *m = s[3]; dirty = 1; break;
                            }
                            if ( *s == VECTOMIN ) {
                                *m = s[3]; dirty = 1; sgn += t[2]; break;
                            }
                            if ( *s == VECTOSUB ) {
                                *m = s[3]; dirty = 1; subcount = 1; break;
                            }
                        }
                        s += s[1];
                    }
ss2:
                    *++m = *++t;
                    s = subs;
                    for ( si = 0; si < setflag; si += 2 ) {
                        if ( t == temp + setlist[si] ) goto ss3;
                    }
                    for ( j = 0; j < i; j++ ) {
                        if ( *t == s[2] ) {
                            if ( *s == VECTOVEC ) {
                                *m = s[3]; dirty = 1; break;
                            }
```

```
                            if ( *s == VECTOMIN ) {
                                *m = s[3]; dirty = 1; sgn += t[1]; break;
                            }
                            if ( *s == VECTOSUB ) {
                                *m = s[3]; dirty = 1; subcount += 2; break;
                            }
                        }
                        s += s[1];
                    }
ss3:                *++m = *++t;
                    if ( ( ABS(*t) - 2*MAXPOWER ) < 0 ) goto RegPow;
                    s = subs;
                    for ( j = 0; j < i; j++ ) {
                        if ( ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
                            if ( *s == SYMTONUM ) {
                                *m = s[3];
                                if ( *t < 0 ) *m = -*m;
                                dirty = 1;
                                break;
                            }
                            if ( *s <= SYMTOSUB ) {
/*
                Here we put together a power function with the proper
                arguments. Note that a p?.q? resolves to a single power.
*/
                                m -= 2;
                                *z++ = EXPONENT;
                                w = z;
                                if ( subcount == 0 ) {
                                    *z++ = 17+FUNHEAD+2*ARGHEAD;
                                    *z++ = DIRTYFLAG;
                                    FILLFUN3(z)
                                    *z++ = 9+ARGHEAD;
                                    *z++ = 0;
                                    FILLARG(z)
                                    *z++ = 9;
                                    *z++ = DOTPRODUCT;
                                    *z++ = 5;
                                    *z++ = *m;
                                    *z++ = m[1];
                                    *z++ = 1;
                                    *z++ = 1;
                                    *z++ = 1;
                                    *z++ = 3;
                                    if ( *s == SYMTOSYM ) {
                                        *z++ = 8+ARGHEAD;
                                        *z++ = 0;
                                        FILLARG(z)
                                        *z++ = 8;
                                        *z++ = SYMBOL;
                                        *z++ = 4;
                                        *z++ = s[3];
                                        *z++ = 1;
                                    }
                                    else {
                                        *z++ = 4+SUBEXPSIZE+ARGHEAD;
                                        *z++ = 1;
                                        FILLARG(z)
                                        *z++ = 4+SUBEXPSIZE;
                                        *z++ = SUBEXPRESSION;
                                        *z++ = SUBEXPSIZE;
                                        *z++ = s[3];
                                        *z++ = 1;
                                        *z++ = AT.ebufnum;
                                        FILLSUB(z)
                                    }
                                    *z++ = 1; *z++ = 1;
                                    *z++ = ( s[2] > 0 ) ? 3: -3;
                                }
                                else if ( subcount == 3 ) {
                                    *z++ = 20+2*SUBEXPSIZE+FUNHEAD+2*ARGHEAD;
                                    *z++ = DIRTYFLAG;
                                    FILLFUN3(z)
                                    *z++ = 12+2*SUBEXPSIZE+ARGHEAD;
                                    *z++ = 1;
                                    *z++ = 12+2*SUBEXPSIZE;
                                    *z++ = SUBEXPRESSION;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = *m + 1;
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = INDTOIND;
                                    *z++ = 4;
                                    *z++ = FUNNYVEC;
                                    *z++ = ++AR.CurDum;

```

```
                                    *z++ = SUBEXPRESSION;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = m[1] + 1;
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = INDTOIND;
                                    *z++ = 4;
                                    *z++ = FUNNYVEC;
                                    *z++ = AR.CurDum;
                                    *z++ = 1; *z++ = 1; *z++ = 3;
                                }
                                else {
                                    if ( subcount == 2 ) {
                                        j = *m; *m = m[1]; m[1] = j;
                                    }
                                    *z++ = 16+SUBEXPSIZE+FUNHEAD+2*ARGHEAD;
                                    *z++ = DIRTYFLAG;
                                    FILLFUN3(z)
                                    *z++ = 8+SUBEXPSIZE+ARGHEAD;
                                    *z++ = 1;
                                    *z++ = 8+SUBEXPSIZE;
                                    *z++ = SUBEXPRESSION;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = *m + 1;
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = INDTOIND;
                                    *z++ = 4;
                                    *z++ = FUNNYVEC;
                                    *z++ = m[1];
                                    *z++ = 1; *z++ = 1; *z++ = 3;
                                }
                                if ( *s == SYMTOSYM ) {
                                    if ( s[2] > 0 ) {
                                        *z++ = -SYMBOL;
                                        *z++ = s[3];
                                        t++;
                                        *w = z-w+1;
                                        goto NextDot;
                                    }
                                    *z++ = 8+ARGHEAD;
                                    *z++ = 0;
                                    *z++ = 8;
                                    *z++ = SYMBOL;
                                    *z++ = 4;
                                    *z++ = s[3];
                                    *z++ = 1;
                                }
                                else {
                                    *z++ = 4+SUBEXPSIZE+ARGHEAD;
                                    *z++ = 1;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = SUBEXPRESSION;
                                    *z++ = SUBEXPSIZE;
                                    *z++ = s[3];
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                }
                                *z++ = 1; *z++ = 1;
                                *z++ = ( s[2] > 0 ) ? 3: -3;
                                t++;
                                *w = z-w+1;
                                goto NextDot;
                            }
                        }
                        s += s[1];
                    }
RegPow:             if ( *m ) m++;
                    else { m -= 2; subcount = 0; }
                    t++;
                    if ( subcount ) {
                        m -= 3;
                        if ( subcount == 3 ) {
                            if ( m[2] < 0 ) {
                                j = (-m[2]) * (2*SUBEXPSIZE+8);
                                *z++ = DENOMINATOR;
                                *z++ = j + 8 + FUNHEAD + ARGHEAD;
                                *z++ = DIRTYFLAG;
                                FILLFUN3(z)
                                *z++ = j + 8 + ARGHEAD;
                                *z++ = 1;
                                *z++ = j + 8;
                                while ( m[2] < 0 ) {
                                    (m[2])++;
```

```
                                    *z++ = SUBEXPRESSION;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = *m + 1;
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = INDTOIND;
                                    *z++ = 4;
                                    *z++ = FUNNYVEC;
                                    *z++ = ++AR.CurDum;
                                    *z++ = SUBEXPRESSION;
                                    *z++ = 8+SUBEXPSIZE;
                                    *z++ = m[1] + 1;
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = INDTOIND;
                                    *z++ = 4;
                                    *z++ = FUNNYVEC;
                                    *z++ = AR.CurDum;
                                    *z++ = SYMTOSYM;    /* Needed to avoid */
                                    *z++ = 4;           /* problems with   */
                                    *z++ = 1000;        /* conversion to   */
                                    *z++ = 1000;        /* square of subexp*/
                                }
                                *z++ = 1; *z++ = 1; *z++ = 3;
                            }
                            else {
                                while ( m[2] > 0 ) {
                                    (m[2])--;
                                    *z++ = SUBEXPRESSION;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = *m + 1;
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = INDTOIND;
                                    *z++ = 4;
                                    *z++ = FUNNYVEC;
                                    *z++ = ++AR.CurDum;
                                    *z++ = SUBEXPRESSION;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = m[1] + 1;
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = INDTOIND;
                                    *z++ = 4;
                                    *z++ = FUNNYVEC;
                                    *z++ = AR.CurDum;
                                }
                            }
                        }
                        else {
                            if ( subcount == 2 ) {
                                j = *m; *m = m[1]; m[1] = j;
                            }
                            if ( m[2] < 0 ) {
                                *z++ = DENOMINATOR;
                                *z++ = 8+SUBEXPSIZE+FUNHEAD+ARGHEAD;
                                *z++ = DIRTYFLAG;
                                FILLFUN3(z)
                                *z++ = 8+SUBEXPSIZE+ARGHEAD;
                                *z++ = 1;
                                *z++ = 8+SUBEXPSIZE;
                            }
                            *z++ = SUBEXPRESSION;
                            *z++ = 4+SUBEXPSIZE;
                            *z++ = *m + 1;
                            *z++ = ABS(m[2]);
                            *z++ = AT.ebufnum;
                            FILLSUB(z)
                            *z++ = INDTOIND;
                            *z++ = 4;
                            *z++ = FUNNYVEC;
                            *z++ = m[1];
                            if ( m[2] < 0 ) {
                                *z++ = 1; *z++ = 1; *z++ = 3;
                            }
                        }
                    }
NextDot:;
                }
                if ( m <= v ) m = v - 2;
                else v[-1] = WORDDIF(m,v) + 2;
                if ( z > accu ) {
                    j = WORDDIF(z,accu);
```

```
                    z = accu;
                    NCOPY(m,z,j);
                }
                break;
/*
            #] DOTPRODUCTS :
*/
            case SETSET:
/*
            #[ SETS :
*/
                temp = accu + (((AR.ComprTop - accu)»1)&(-2));
                if ( ResolveSet(BHEAD t,temp,sub) ) {
                    Terminate(-1);
                }
                setlist = t + 2 + t[3];
                setflag = t[1] - 2 - t[3];  /* Number of elements * 2 */
                t = temp; u = t + t[1];
                goto ReSwitch;
/*
            #] SETS :
*/
            case VECTOR:
/*
            #[ VECTORS :
*/
                *m++ = *t++;
                *m++ = *t++;
                v = m;
                z = accu;
                while ( t < u ) {
                    *m = *t;
                    for ( si = 0; si < setflag; si += 2 ) {
                        if ( t == temp + setlist[si] ) goto ss4;
                    }
                    s = subs;
                    for ( j = 0; j < i; j++ ) {
                        if ( *t == s[2] ) {
                            if ( *s == INDTOIND || *s == VECTOVEC ) {
                                *m = s[3]; dirty = 1; break;
                            }
                            if ( *s == VECTOMIN ) {
                                *m = s[3]; dirty = 1; sgn++; break;
                            }
                            else if ( *s == VECTOSUB ) {
                                *z++ = SUBEXPRESSION;
                                *z++ = 4+SUBEXPSIZE;
                                *z++ = s[3]+1;
                                *z++ = 1;
                                *z++ = AT.ebufnum;
                                FILLSUB(z)
                                *z++ = VECTOVEC;
                                *z++ = 4;
                                *z++ = FUNNYVEC;
                                *z++ = *++t;
                                m--;
                                s = subs;
                                for ( j = 0; j < i; j++ ) {
                                    if ( z[-1] == s[2] ) {
                                        if ( *s == INDTOIND || *s == VECTOVEC ) {
                                            z[-1] = s[3];
                                            break;
                                        }
                                        if ( *s == INDTOSUB || *s == VECTOSUB ) {
                                            z[-1] = ++AR.CurDum;
                                            *z++ = SUBEXPRESSION;
                                            *z++ = 4+SUBEXPSIZE;
                                            *z++ = s[3]+1;
                                            *z++ = 1;
                                            *z++ = AT.ebufnum;
                                            FILLSUB(z)
                                            if ( *s == INDTOSUB ) *z++ = INDTOIND;
                                            else *z++ = VECTOSUB;
                                            *z++ = 4;
                                            *z++ = FUNNYVEC;
                                            *z++ = AR.CurDum;
                                            break;
                                        }
                                    }
                                    s += s[1];
                                }
                                dirty = 1;
                                break;
                            }
                            else if ( *s == INDTOSUB ) {
                                *z++ = SUBEXPRESSION;
                                *z++ = 4+SUBEXPSIZE;
```

```
                                *z++ = s[3]+1;
                                *z++ = 1;
                                *z++ = AT.ebufnum;
                                FILLSUB(z)
                                *z++ = INDTOIND;
                                *z++ = 4;
                                *z++ = FUNNYVEC;
                                m -= 2;
                                *z++ = m[1];
                                dirty = 1;
                                t++;
                                break;
                            }
                        }
                        s += s[1];
                    }
ss4:                m++; t++;
                }
                if ( m <= v ) m = v-2;
                else v[-1] = WORDDIF(m,v)+2;
                if ( z > accu ) {
                    j = WORDDIF(z,accu); z = accu;
                    NCOPY(m,z,j);
                }
                break;
/*
            #] VECTORS :
*/
            case INDEX:
/*
            #[ INDEX :
*/
                *m++ = *t++;
                *m++ = *t++;
                v = m;
                z = accu;
                while ( t < u ) {
                    *m = *t;
                    for ( si = 0; si < setflag; si += 2 ) {
                        if ( t == temp + setlist[si] ) goto ss5;
                    }
                    s = subs;
                    for ( j = 0; j < i; j++ ) {
                        if ( *t == s[2] ) {
                            if ( *s == INDTOIND || *s == VECTOVEC )
                                { *m = s[3]; dirty = 1; break; }
                            if ( *s == VECTOMIN )
                                { *m = s[3]; dirty = 1; sgn++; break; }
                            else if ( *s == VECTOSUB || *s == INDTOSUB ) {
                                *z++ = SUBEXPRESSION;
                                *z++ = SUBEXPSIZE;
                                *z++ = s[3];
                                *z++ = 1;
                                *z++ = AT.ebufnum;
                                FILLSUB(z)
                                m--;
                                dirty = 1;
                                break;
                            }
                        }
                        s += s[1];
                    }
ss5:                m++; t++;
                }
                if ( m <= v ) m = v-2;
                else v[-1] = WORDDIF(m,v)+2;
                if ( z > accu ) {
                    j = WORDDIF(z,accu); z = accu;
                    NCOPY(m,z,j);
                }
                break;
/*
            #] INDEX :
*/
            case DELTA:
            case LEVICIVITA:
            case GAMMA:
/*
            #[ SPECIALS :
*/
                v = m;
                *m++ = *t++;
                *m++ = *t++;
#if FUNHEAD > 2
                if ( t[-2] != DELTA ) *m++ = *t++;
#endif
Tensors:
```

```
                COPYFUN3(m,t)
                z = accu;
                while ( t < u ) {
                    *m = *t;
                    for ( si = 0; si < setflag; si += 2 ) {
                        if ( t == temp + setlist[si] ) goto ss6;
                    }
                    s = subs;
                    if ( *m == FUNNYWILD ) {
                        CBUF *C = cbuf+AT.ebufnum;
                        t++;
                        for ( j = 0; j < i; j++ ) {
                            if ( *s == ARGTOARG && *t == s[2] ) {
                                v[2] |= DIRTYFLAG;
                                if ( s[3] < 0 ) { /* empty */
                                    t++; break;
                                }
                                w = C->rhs[s[3]];
DEBUG(MesPrint("Thread %w(a): s[3] = %d, w=(%d,%d,%d,%d)",s[3],w[0],w[1],w[2],w[3]);)
                                j = *w++;
                                if ( j > 0 ) {
                                    NCOPY(m,w,j);
                                }
                                else {
                                    while ( *w ) {
                                        if ( *w == -INDEX || *w == -VECTOR
                                        || *w == -MINVECTOR
                                        || ( *w == -SNUMBER && w[1] >= 0
                                        && w[1] < AM.OffsetIndex ) ) {
                                            if ( *w == -MINVECTOR ) sgn++;
                                            w++;
                                            *m++ = *w++;
                                        }
                                        else {
                                            MLOCK(ErrorMessageLock);
DEBUG(MesPrint("Thread %w(aa): *w = %d",*w);)
                                            MesPrint("Illegal substitution of argument field in tensor");
                                            MUNLOCK(ErrorMessageLock);
                                            SETERROR(-1)
                                        }
                                    }
                                }
                                t++;
                                break;
                            }
                            s += s[1];
                        }
                    }
                    else {
                        for ( j = 0; j < i; j++ ) {
                            if ( *t == s[2] ) {
                                if ( *s == INDTOIND || *s == VECTOVEC )
                                    { *m = s[3]; dirty = 1; break; }
                                if ( *s == VECTOMIN )
                                    { *m = s[3]; dirty = 1; sgn++; break; }
                                else if ( *s == VECTOSUB || *s == INDTOSUB ) {
                                    *m = ++AR.CurDum;
                                    *z++ = SUBEXPRESSION;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = s[3]+1;
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = INDTOIND;
                                    *z++ = 4;
                                    *z++ = FUNNYVEC;
                                    *z++ = AR.CurDum;
                                    dirty = 1;
                                    break;
                                }
                            }
                            s += s[1];
                        }
                        if ( j < i && *v != DELTA ) v[2] |= DIRTYFLAG;
ss6:                    m++; t++;
                    }
                }
                v[1] = WORDDIF(m,v);
                if ( z > accu ) {
                    j = WORDDIF(z,accu); z = accu;
                    NCOPY(m,z,j);
                }
                break;
/*
            #] SPECIALS :
*/
```

```
            case SUBEXPRESSION:
/*
            #[ SUBEXPRESSION :
*/
                dirty = 1;
                tstop = t + t[1];
                *m++ = *t++;
                *m++ = *t++;
                *m++ = *t++;
                *m++ = *t++;
                if ( t[-1] >= 2*MAXPOWER || t[-1] <= -2*MAXPOWER ) {
                    s = subs;
                    for ( j = 0; j < i; j++ ) {
                        if ( *s == SYMTONUM &&
                                ( ABS(t[-1]) - 2*MAXPOWER ) == s[2] ) {
                            m[-1] = s[3];
                            if ( t[-1] < 0 ) m[-1] = -m[-1];
                            break;
                        }
                        s += s[1];
                    }
                }
                *m++ = *t++;
                COPYSUB(m,t)
                while ( t < tstop ) {
                    for ( si = 0; si < setflag; si += 2 ) {
                        if ( t == temp + setlist[si] - 2 ) goto ss7;
                    }
                    s = subs;
                    for ( j = 0; j < i; j++ ) {
                        if ( s[2] == t[2] ) {
                            if ( ( *s <= SYMTOSUB && *t <= SYMTOSUB )
                            || ( *s == *t && *s < FROMBRAC )
                            || ( *s == VECTOVEC && ( *t == VECTOSUB || *t == VECTOMIN ) )
                            || ( *s == VECTOSUB && ( *t == VECTOVEC || *t == VECTOMIN ) )
                            || ( *s == VECTOMIN && ( *t == VECTOSUB || *t == VECTOVEC ) )
                            || ( *s == INDTOIND && *t == INDTOSUB )
                            || ( *s == INDTOSUB && *t == INDTOIND ) ) {
                                WORD *vv = m;
/*                              *t = *s;  Wrong!!! Overwrites compiler buffer  */
                                j = t[1];
                                NCOPY(m,t,j);
                                vv[0] = s[0];
                                vv[3] = s[3];
                                goto sr7;
                            }
                        }
                        s += s[1];
                    }
ss7:                    j = t[1];
                    NCOPY(m,t,j);
sr7:;
                }
                break;
/*
            #] SUBEXPRESSION :
*/
            case EXPRESSION:
/*
            #[ EXPRESSION :
*/
                dirty = 1;
                tstop = t + t[1];
                v = m;
                *m++ = *t++;
                *m++ = *t++;
                *m++ = *t++;
                *m++ = *t++;
                s = subs;
                for ( j = 0; j < i; j++ ) {
                    if ( ( ABS(t[-1]) - 2*MAXPOWER ) == s[2] ) {
                        if ( *s == SYMTONUM ) {
                            m[-1] = s[3];
                            if ( t[-1] < 0 ) m[-1] = -m[-1];
                            break;
                        }
                        else if ( *s <= SYMTOSUB ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Wildcard power of expression should be a number");
                            MUNLOCK(ErrorMessageLock);
                            SETERROR(-1)
                        }
                    }
                    s += s[1];
                }
                *m++ = *t++;
                COPYSUB(m,t)
```

```
                while ( t < tstop && *t != WILDCARDS ) {
                    j = t[1];
                    NCOPY(m,t,j);
                }
                if ( t < tstop && *t == WILDCARDS ) {
                    *m++ = *t;
                    s = sub;
                    j = s[1];
                    *m++ = j+2;
                    NCOPY(m,s,j);
                    t += t[1];
                }
                if ( t < tstop && *t == FROMBRAC ) {
                    w = m;
                    *m++ = *t;
                    *m++ = t[1];
                    if ( WildFill(BHEAD m,t+2,sub) < 0 ) {
                        MLOCK(ErrorMessageLock);
                        MesCall("WildFill");
                        MUNLOCK(ErrorMessageLock);
                        SETERROR(-1)
                    }
                    m += *m;
                    w[1] = m - w;
                    t += t[1];
                }
                while ( t < tstop ) {
                    j = t[1];
                    NCOPY(m,t,j);
                }
                v[1] = m-v;
                break;
/*
            #] EXPRESSION :
*/
            default:
/*
            #[ FUNCTIONS :
*/
                if ( *t >= FUNCTION ) {
                    dflag = 0;
                    na = 0;
                    *m = *t;
                    for ( si = 0; si < setflag; si += 2 ) {
                        if ( t == temp + setlist[si] ) {
                            dflag = DIRTYFLAG;  goto ss8;
                        }
                    }
                    s = subs;
                    for ( j = 0; j < i; j++ ) {
                        if ( *s == FUNTOFUN && *t == s[2] )
                            { *m = s[3]; dirty = 1; dflag = DIRTYFLAG; break; }
                        s += s[1];
                    }
ss8:                v = m;
                    if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
                    >= TENSORFUNCTION ) {
                        if ( *m < FUNCTION || functions[*m-FUNCTION].spec
                        < TENSORFUNCTION ) {
                            MLOCK(ErrorMessageLock);
                            MesPrint("Illegal wildcarding of regular function to tensorfunction");
                            MUNLOCK(ErrorMessageLock);
                            SETERROR(-1)
                        }
                        m++; t++;
                        *m++ = *t++;
                        *m++ = *t++ | dflag;
                        goto Tensors;
                    }
                    m++; t++;
                    *m++ = *t++;
                    *m++ = *t++ | dflag;
                    COPYFUN3(m,t)
                    z = accu;
                    while ( t < u ) {       /* do an argument */
                        if ( *t < 0 ) {
/*
            #[ Simple arguments :
*/
                            CBUF *C = cbuf+AT.ebufnum;
                            for ( si = 0; si < setflag; si += 2 ) {
                                if ( *t <= -FUNCTION ) {
                                    if ( t == temp + setlist[si] ) {
                                        v[2] |= DIRTYFLAG; goto ss10; }
                                }
                                else {
                                    if ( t == temp + setlist[si]-1 ) {
```

```
                                        v[2] |= DIRTYFLAG; goto ss9; }
                                }
                            }
                            if ( *t == -ARGWILD ) {
                                s = subs;
                                for ( j = 0; j < i; j++ ) {
                                    if ( *s == ARGTOARG && s[2] == t[1] ) break;
                                    s += s[1];
                                }
                                v[2] |= DIRTYFLAG;
                                w = C->rhs[s[3]];
DEBUG(MesPrint("Thread %w(b): s[3] = %d, w=(%d,%d,%d,%d)",s[3],w[0],w[1],w[2],w[3]);)
                                if ( *w == 0 ) {
                                    w++;
                                    while ( *w ) {
                                        if ( *w > 0 ) j = *w;
                                        else if ( *w <= -FUNCTION ) j = 1;
                                        else j = 2;
                                        NCOPY(m,w,j);
                                    }
                                }
                                else {
                                    j = *w++;
                                    while ( --j >= 0 ) {
                                        if ( *w < MINSPEC ) *m++ = -VECTOR;
                                        else if ( *w >= 0 && *w < AM.OffsetIndex )
                                            *m++ = -SNUMBER;
                                        else *m++ = -INDEX;
                                        *m++ = *w++;
                                    }
                                }
                                t += 2;
                                dirty = 1;
                                if ( ( *v == NUMARGSFUN || *v == NUMTERMSFUN )
                                && t >= u && m == v + FUNHEAD ) {
                                    m = v;
                                    *m++ = SNUMBER; *m++ = 3; *m++ = 0;
                                    break;
                                }
                            }
                            else if ( *t <= -FUNCTION ) {
                                *m = *t;
                                s = subs;
                                for ( j = 0; j < i; j++ ) {
                                    if ( -*t == s[2] ) {
                                        if ( *s == FUNTOFUN )
                                            { *m = -s[3]; dirty = 1; v[2] |= DIRTYFLAG; break; }
                                    }
                                    s += s[1];
                                }
                                m++; t++;
                            }
                            else if ( *t == -SYMBOL ) {
                                *m++ = *t++;
                                *m = *t;
                                s = subs;
                                for ( j = 0; j < i; j++ ) {
                                    if ( *t == s[2] && *s <= SYMTOSUB ) {
                                        dirty = 1; v[2] |= DIRTYFLAG;
                                        if ( AR.PolyFunType == 2 && v[0] == AR.PolyFun )
                                            v[2] |= MUSTCLEANPRF;
                                        if ( *s == SYMTOSYM ) *m = s[3];
                                        else if ( *s == SYMTONUM ) {
                                            m[-1] = -SNUMBER;
                                            *m = s[3];
                                        }
                                        else if ( *s == SYMTOSUB ) {
ToSub:                                      m--;
                                            w = C->rhs[s[3]];
DEBUG(MesPrint("Thread %w(c): s[3] = %d, w=(%d,%d,%d,%d)",s[3],w[0],w[1],w[2],w[3]);)
                                            s = m;
                                            m += 2;
                                            while ( *w ) {
                                                j = *w;
                                                NCOPY(m,w,j);
                                            }
                                            *s = WORDDIF(m,s);
                                            s[1] = 0;
                                            *m = 0;
                                            if ( t[-1] == -MINVECTOR ) {
                                                w = s+2;
                                                while ( *w ) {
                                                    w += *w;
                                                    w[-1] = -w[-1];
                                                }
                                            }
                                            if ( ToFast(s,s) ) {
```

```
                                        if ( *s <= -FUNCTION ) m = s;
                                        else m = s + 1;
                                    }
                                    else m--;
                                }
                                break;
                            }
                            s += s[1];
                        }
                        m++; t++;
                    }
                    else if ( *t == -INDEX ) {
                        *m++ = *t++;
                        *m = *t;
                        s = subs;
                        for ( j = 0; j < i; j++ ) {
                            if ( *t == s[2] ) {
                                if ( *s == INDTOIND || *s == VECTOVEC ) {
                                    *m = s[3];
                                    if ( *m < MINSPEC ) m[-1] = -VECTOR;
                                    else if ( *m >= 0 && *m < AM.OffsetIndex )
                                        m[-1] = -SNUMBER;
                                    else m[-1] = -INDEX;
                                }
                                else if ( *s == VECTOSUB || *s == INDTOSUB ) {
                                    m[-1] = -INDEX;
                                    *m = ++AR.CurDum;
                                    *z++ = SUBEXPRESSION;
                                    *z++ = 4+SUBEXPSIZE;
                                    *z++ = s[3]+1;
                                    *z++ = 1;
                                    *z++ = AT.ebufnum;
                                    FILLSUB(z)
                                    *z++ = INDTOIND;
                                    *z++ = 4;
                                    *z++ = FUNNYVEC;
                                    *z++ = AR.CurDum;
                                }
                                v[2] |= DIRTYFLAG; dirty = 1;
                                break;
                            }
                            s += s[1];
                        }
                        m++; t++;
                    }
                    else if ( *t == -VECTOR || *t == -MINVECTOR ) {
                        *m++ = *t++;
                        *m = *t;
                        s = subs;
                        for ( j = 0; j < i; j++ ) {
                            if ( *t == s[2] ) {
                                if ( *s == VECTOVEC ) *m = s[3];
                                else if ( *s == VECTOMIN ) {
                                    *m = s[3];
                                    if ( t[-1] == -VECTOR )
                                        m[-1] = -MINVECTOR;
                                    else
                                        m[-1] = -VECTOR;
                                }
                                else if ( *s == VECTOSUB ) goto ToSub;
                                dirty = 1; v[2] |= DIRTYFLAG;
                                break;
                            }
                            s += s[1];
                        }
                        m++; t++;
                    }
                    else if ( *t == -SNUMBER ) {
                        *m++ = *t++;
                        *m = *t;
                        s = subs;
                        for ( j = 0; j < i; j++ ) {
                            if ( *t == s[2] && *s >= NUMTONUM && *s <= NUMTOSUB ) {
                                dirty = 1; v[2] |= DIRTYFLAG;
                                if ( *s == NUMTONUM ) *m = s[3];
                                else if ( *s == NUMTOSYM ) {
                                    m[-1] = -SYMBOL;
                                    *m = s[3];
                                }
                                else if ( *s == NUMTOIND ) {
                                    m[-1] = -INDEX;
                                    *m = s[3];
                                }
                                else if ( *s == NUMTOSUB ) goto ToSub;
                                break;
                            }
                            s += s[1];
```

```
                                }
                                m++; t++;
                            }
                            else {
ss9:                            *m++ = *t++;
ss10:                           *m++ = *t++;
                            }
                            na = WORDDIF(z,accu);
/*
            #] Simple arguments :
*/
                        }
                        else {
                            w = m;
                            zz = t;
                            NEXTARG(zz)
                            odirt = AN.WildDirt; AN.WildDirt = 0;
                            AR.CompressPointer = accu + na;
                            for ( j = 0; j < ARGHEAD; j++ ) *m++ = *t++;
                            j = 0;
                            adirt = 0;
                            while ( t < zz ) {  /* do a term */
                                if ( ( len = WildFill(BHEAD m,t,sub) ) < 0 ) {
                                    MLOCK(ErrorMessageLock);
                                    MesCall("WildFill");
                                    MUNLOCK(ErrorMessageLock);
                                    SETERROR(-1)
                                }
                                if ( AN.WildDirt ) {
                                    adirt = AN.WildDirt;
                                    AN.WildDirt = 0;
                                }
                                m += len;
                                t += *t;
                            }
                            *w = WORDDIF(m,w);  /* Fill parameter length */
                            if ( adirt ) {
                                dirty = w[1] = 1; v[2] |= DIRTYFLAG;
                                if ( AR.PolyFunType == 2 && v[0] == AR.PolyFun )
                                    v[2] |= MUSTCLEANPRF;
                                AN.WildDirt = adirt;
                            }
                            else {
                                AN.WildDirt = odirt;
                            }
                            if ( ToFast(w,w) ) {
                                if ( *w <= -FUNCTION ) {
                                    if ( *w == NUMARGSFUN || *w == NUMTERMSFUN ) {
                                        *w = -SNUMBER; w[1] = 0; m = w + 2;
                                    }
                                    else m = w+1;
                                }
                                else m = w+2;
                            }
                            AR.CompressPointer = oldcpointer;
                        }
                    }
                    v[1] = WORDDIF(m,v);        /* Fill function length */
                    s = accu;
                    NCOPY(m,s,na);
/*
                    Now some code to speed up a few special cases
*/
                    if ( v[0] == EXPONENT ) {
                        if ( v[1] == FUNHEAD+4 && v[FUNHEAD] == -SYMBOL &&
                        v[FUNHEAD+2] == -SNUMBER && v[FUNHEAD+3] < MAXPOWER
                        && v[FUNHEAD+3] > -MAXPOWER ) {
                            v[0] = SYMBOL;
                            v[1] = 4;
                            v[2] = v[FUNHEAD+1];
                            v[3] = v[FUNHEAD+3];
                            m = v+4;
                        }
                        else if ( v[1] == FUNHEAD+ARGHEAD+11
                         && v[FUNHEAD] == ARGHEAD+9
                         && v[FUNHEAD+ARGHEAD] == 9
                         && v[FUNHEAD+ARGHEAD+1] == DOTPRODUCT
                         && v[FUNHEAD+ARGHEAD+8] == 3
                         && v[FUNHEAD+ARGHEAD+7] == 1
                         && v[FUNHEAD+ARGHEAD+6] == 1
                         && v[FUNHEAD+ARGHEAD+5] == 1
                         && v[FUNHEAD+ARGHEAD+9] == -SNUMBER
                         && v[FUNHEAD+ARGHEAD+10] < MAXPOWER
                         && v[FUNHEAD+ARGHEAD+10] > -MAXPOWER ) {
                            v[0] = DOTPRODUCT;
                            v[1] = 5;
                            v[2] = v[FUNHEAD+ARGHEAD+3];
```

```
                            v[3] = v[FUNHEAD+ARGHEAD+4];
                            v[4] = v[FUNHEAD+ARGHEAD+10];
                            m = v+5;
                        }
                    }
                }
                else { while ( t < u ) *m++ = *t++; }
/*
            #] FUNCTIONS :
*/
        }
        t = uu;
    } while ( t < r );
    t = from;           /* Copy coefficient */
    t += *t;
    if ( r < t ) do { *m++ = *r++; } while ( r < t );
    if ( ( sgn & 1 ) != 0 ) m[-1] = -m[-1];
    *to = WORDDIF(m,to);
    if ( dirty ) AN.WildDirt = dirty;
    return(*to);
}

/*
        #] WildFill :
        #[ ResolveSet :         WORD ResolveSet(from,to,subs)

        The set syntax is:
        SET,length,subterm,where,whichmember[,where,whichmember]

        setlength is 2*n+1 with n the number of set substitutions.
        length = setlength + subtermlength + 2

        At `where' is the number of the set and `whichmember' is the
        number of the element. This is still a symbol/dollar and we
        have to find the substitution in the wildcards.
        The output is the subterm in which the setelements have been
        substituted. This is ready for further wildcard substitutions.
*/

WORD ResolveSet(PHEAD WORD *from, WORD *to, WORD *subs)
{
    GETBIDENTITY
    WORD *m, *s, *w, j, i, ii, i3, flag, num;
    DOLLARS d = 0;
#ifdef WITHPTHREADS
    int nummodopt, dtype = -1;
#endif
    m = to;                     /* pointer in output */
    s = from + 2;
    w = s + s[1];
    while ( s < w ) *m++ = *s++;
    j = (from[1] - WORDDIF(w,from) ) » 1;
    m = subs + subs[1];
    subs += SUBEXPSIZE;
    s = subs;
    i = 0;
    while ( s < m ) { i++; s += s[1]; }
    m = to;
    if ( *m >= FUNCTION && functions[*m-FUNCTION].spec
    >= TENSORFUNCTION ) flag = 0;
    else flag = 1;
    while ( --j >= 0 ) {
        if ( w[1] >= 0 ) {
            s = subs;
            for ( ii = 0; ii < i; ii++ ) {
                if ( *s == SYMTONUM && s[2] == w[1] ) { num = s[3]; goto GotOne; }
                s += s[1];
            }
            MLOCK(ErrorMessageLock);
            MesPrint(" Unresolved setelement during substitution");
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        else {  /* Dollar ! */
            d = Dollars - w[1];
#ifdef WITHPTHREADS
            if ( AS.MultiThreaded ) {
                for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
                    if ( -w[1] == ModOptdollars[nummodopt].number ) break;
                }
                if ( nummodopt < NumModOptdollars ) {
                    dtype = ModOptdollars[nummodopt].type;
                    if ( dtype == MODLOCAL ) {
                        d = ModOptdollars[nummodopt].dstruct+AT.identity;
                    }
                    else {
                        LOCK(d->pthreadslockread);
```

```
                    }
                }
            }
#endif
            if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
                if ( d->where[0] == 4 && d->where[3] == 3 && d->where[2] == 1
                && d->where[1] > 0 && d->where[4] == 0 ) {
                    num = d->where[1]; goto GotOne;
                }
            }
            else if ( d->type == DOLINDEX ) {
                if ( d->index > 0 && d->index < AM.OffsetIndex ) {
                    num = d->index; goto GotOne;
                }
            }
            else if ( d->type == DOLARGUMENT ) {
                if ( d->where[0] == -SNUMBER && d->where[1] > 0 ) {
                    num = d->where[1]; goto GotOne;
                }
            }
            else if ( d->type == DOLWILDARGS ) {
                if ( d->where[0] == 1 &&
                d->where[1] > 0 && d->where[1] < AM.OffsetIndex ) {
                    num = d->where[1]; goto GotOne;
                }
                if ( d->where[0] == 0 && d->where[1] < 0 && d->where[3] == 0 ) {
                    if ( ( d->where[1] == -SNUMBER && d->where[2] > 0 )
                    || ( d->where[1] == -INDEX && d->where[2] > 0
                    && d->where[2] < AM.OffsetIndex ) ) {
                        num = d->where[2]; goto GotOne;
                    }
                }
            }
#ifdef WITHPTHREADS
            if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
            MLOCK(ErrorMessageLock);
            MesPrint("Unusable type of variable $%s in set substitution",
                AC.dollarnames->namebuffer+d->name);
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
GotOne:;
#ifdef WITHPTHREADS
        if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslockread); }
#endif
        ii = m[*w];
        if ( ii >= 2*MAXPOWER ) i3 = ii - 2*MAXPOWER;
        else if ( ii <= -2*MAXPOWER ) i3 = -ii - 2*MAXPOWER;
        else i3 = ( ii >= 0 ) ? ii: -ii - 1;

        if ( num > ( Sets[i3].last - Sets[i3].first ) || num <= 0 ) {
            MLOCK(ErrorMessageLock);
            MesPrint("Array bound check during set substitution");
            MesPrint("    value is %d",num);
            MUNLOCK(ErrorMessageLock);
            return(-1);
        }
        m[*w] = (SetElements+Sets[i3].first)[num-1];
        if ( Sets[i3].type == CSYMBOL && m[*w] > MAXPOWER ) {
            if ( ii >= 2*MAXPOWER ) m[*w] -= 2*MAXPOWER;
            else if ( ii <= -2*MAXPOWER ) m[*w] = -(m[*w] - 2*MAXPOWER);
            else {
                m[*w] -= MAXPOWER;
                if ( m[*w] < MAXPOWER ) m[*w] -= 2*MAXPOWER;
                if ( flag ) MakeDirty(m,m+*w,1);
            }
        }
        else if ( Sets[i3].type == CSYMBOL ) {
            if ( ii >= 2*MAXPOWER ) m[*w] += 2*MAXPOWER;
            else if ( ii <= -2*MAXPOWER ) m[*w] = -m[*w] - 2*MAXPOWER;
            else if ( ii < 0 ) m[*w] = - m[*w];
        }
        else if ( ii < 0 ) m[*w] = - m[*w];
        w += 2;
    }
    m = to;
    if ( *m >= FUNCTION && functions[*m-FUNCTION].spec
    >= TENSORFUNCTION ) {
        w = from + 2 + from[3];
        if ( *w == 0 ) {    /* We had function -> tensor */
            m = from + 2 + FUNHEAD; s = to + FUNHEAD;
            while ( m < w ) {
                if ( *m == -INDEX || *m == -VECTOR ) {}
                else if ( *m == -ARGWILD ) { *s++ = FUNNYWILD; }
                else {
                    MLOCK(ErrorMessageLock);
```

```
                    MesPrint("Illegal argument in tensor after set substitution");
                    MUNLOCK(ErrorMessageLock);
                    SETERROR(-1)
                }
                *s++ = m[1];
                m += 2;
            }
            to[1] = WORDDIF(s,to);
        }
    }
    return(0);
}

/*
        #] ResolveSet :
        #[ ClearWild :          VOID ClearWild()

    Clears the current wildcard settings and makes them ready for
    CheckWild and AddWild.

*/

VOID ClearWild(PHEAD0)
{
    GETBIDENTITY
    WORD n, nn, *w;
    n = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;     /* Number of wildcards */
    AN.NumWild = nn = n;
    if ( n > 0 ) {
        w = AT.WildMask;
        do { *w++ = 0; } while ( --n > 0 );
        w = AN.WildValue;
        do {
            if ( *w == SYMTONUM ) *w = SYMTOSYM;
            w += w[1];
        } while ( --nn > 0 );
    }
}

/*
        #] ClearWild :
        #[ AddWild :            WORD AddWild(oldnumber,type,newnumber)

        Adds a wildcard assignment.
        Extra parameter in AN.argaddress;

*/

WORD AddWild(PHEAD WORD oldnumber, WORD type, WORD newnumber)
{
    GETBIDENTITY
    WORD *w, *m, n, k, i = -1;
    CBUF *C = cbuf+AT.ebufnum;
DEBUG(WORD *mm;)
    AN.WildReserve = 0;
    m = AT.WildMask;
    w = AN.WildValue;
    n = AN.NumWild;
    if ( n <= 0 ) { return(-1); }
    if ( type <= SYMTOSUB ) {
        do {
            if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
                if ( n > 1 && w[4] == SETTONUM ) i = w[7];
                *w = type;
                if ( *m != 2 ) *m = 1;
                if ( type != SYMTOSUB ) {
                    if ( type == SYMTONUM ) AN.MaskPointer = m;
                    w[3] = newnumber;
                    goto FlipOn;
                }
                m = AddRHS(AT.ebufnum,1);
                w[3] = C->numrhs;
                w = AN.argaddress;
DEBUG(mm = m;)
                n = *w - ARGHEAD;
                w += ARGHEAD;
                while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,4);
                while ( --n >= 0 ) *m++ = *w++;
                *m++ = 0;
                C->rhs[C->numrhs+1] = m;
DEBUG(MesPrint("Thread %w(d): m=(%d,%d,%d,%d)(%d)",mm[0],mm[1],mm[2],mm[3],C->numrhs);)
                C->Pointer = m;
                goto FlipOn;
            }
            m++; w += w[1];
        } while ( --n > 0 );
    }
```

```
    else if ( type == ARGTOARG ) {
        do {
            if ( w[2] == oldnumber && *w == ARGTOARG ) {
                *m = 1;
                m = AddRHS(AT.ebufnum,1);
                w[3] = C->numrhs;
                w = AN.argaddress;
DEBUG(mm=m;)
                if ( ( newnumber & EATTENSOR ) != 0 ) {
                    n = newnumber & ~EATTENSOR;
                    *m++ = n;
                    w = AN.argaddress;
                }
                else {
                    while ( --newnumber >= 0 ) { NEXTARG(w) }
                    n = WORDDIF(w,AN.argaddress);
                    w = AN.argaddress;
                    *m++ = 0;
                }
                while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,5);
DEBUG(if ( mm != m-1 ) MesPrint("Thread %w(e): Alarm!"); mm = m-1;)
                while ( --n >= 0 ) *m++ = *w++;
                *m++ = 0;
                C->rhs[C->numrhs+1] = m;
                C->Pointer = m;
DEBUG(MesPrint("Thread %w(e): w=(%d,%d,%d,%d)(%d)",mm[0],mm[1],mm[2],mm[3],C->numrhs);)
                return(0);
            }
            m++; w += w[1];
        } while ( --n > 0 );
    }
    else if ( type == ARLTOARL ) {
        do {
            if ( w[2] == oldnumber && *w == ARGTOARG ) {
                WORD **a;
                *m = 1;
                m = AddRHS(AT.ebufnum,1);
                w[3] = C->numrhs;
DEBUG(mm=m;)
                a = (WORD **)(AN.argaddress); n = 0; k = newnumber;
                while ( --newnumber >= 0 ) {
                    w = *a++;
                    if ( *w > 0 ) n += *w;
                    else if ( *w <= -FUNCTION ) n++;
                    else n += 2;
                }
                *m++ = 0;
                while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,6);
DEBUG(if ( mm != m-1 ) MesPrint("Thread %w(f): Alarm!"); mm = m-1;)
                a = (WORD **)(AN.argaddress);
                while ( --k >= 0 ) {
                    w = *a++;
                    if ( *w > 0 ) { n = *w; NCOPY(m,w,n); }
                    else if ( *w <= -FUNCTION ) *m++ = *w++;
                    else { *m++ = *w++; *m++ = *w++; }
                }
                *m++ = 0;
                C->rhs[C->numrhs+1] = m;
DEBUG(MesPrint("Thread %w(f): w=(%d,%d,%d,%d)(%d)",mm[0],mm[1],mm[2],mm[3],C->numrhs);)
                C->Pointer = m;
                return(0);
            }
            m++; w += w[1];
        } while ( --n > 0 );
    }
    else if ( type == VECTOSUB || type == INDTOSUB ) {
        WORD *ss, *sstop, *tt, *ttstop, j, *v1, *v2 = 0;
        do {
            if ( w[2] == oldnumber && ( *w == type ||
            ( type == VECTOSUB && ( *w == VECTOVEC || *w == VECTOMIN ) )
            || ( type == INDTOSUB && *w == INDTOIND ) ) ) {
                if ( n > 1 && w[4] == SETTONUM ) i = w[7];
                *w = type;
                *m = 1;
                m = AddRHS(AT.ebufnum,1);
                w[3] = C->numrhs;
                w = AN.argaddress;
                n = *w - ARGHEAD;
                w += ARGHEAD;
                while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,7);
                while ( --n >= 0 ) *m++ = *w++;
                *m++ = 0;
                C->rhs[C->numrhs+1] = m;
                C->Pointer = m;
                m = AddRHS(AT.ebufnum,1);
                w = AN.argaddress;
                n = *w - ARGHEAD;
```

```
                w += ARGHEAD;
                while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,8);
                sstop = w + n;
                while ( w < sstop ) {           /* Run over terms */
                    tt = w + *w; ttstop = tt - ABS(tt[-1]);
                    ss = m; m++; w++;
                    while ( w < ttstop ) {      /* Subterms */
                        if ( *w != INDEX ) {
                            j = w[1];
                            NCOPY(m,w,j);
                        }
                        else {
                            v1 = m;
                            *m++ = *w++;
                            *m++ = j = *w++;
                            j -= 2;
                            while ( --j >= 0 ) {
                                if ( *w >= MINSPEC ) *m++ = *w++;
                                else v2 = w++;
                            }
                            j = WORDDIF(m,v1);
                            if ( j != v1[1] ) {
                                if ( j <= 2 ) m -= 2;
                                else v1[1] = j;
                                *m++ = VECTOR;
                                *m++ = 4;
                                *m++ = *v2;
                                *m++ = FUNNYVEC;
                            }
                        }
                    }
                    while ( w < tt ) *m++ = *w++;
                    *ss = WORDDIF(m,ss);
                }
                *m++ = 0;
                C->rhs[C->numrhs+1] = m;
                C->Pointer = m;
                if ( m > C->Top ) {
                    MLOCK(ErrorMessageLock);
                    MesPrint("Internal problems with extra compiler buffer");
                    MUNLOCK(ErrorMessageLock);
                    Terminate(-1);
                }
                goto FlipOn;
            }
            m++; w += w[1];
        } while ( --n > 0 );
    }
    else {
        do {
            if ( w[2] == oldnumber && ( *w == type || ( type == VECTOVEC
            && ( *w == VECTOMIN || *w == VECTOSUB ) ) || ( type == VECTOMIN
            && ( *w == VECTOVEC || *w == VECTOSUB ) )
            || ( type == INDTOIND && *w == INDTOSUB ) ) ) {
                if ( n > 1 && w[4] == SETTONUM ) i = w[7];
                *w = type;
                w[3] = newnumber;
                *m = 1;
                goto FlipOn;
            }
            m++; w += w[1];
        } while ( --n > 0 );
    }
    MLOCK(ErrorMessageLock);
    MesPrint("Bug in AddWild.");
    MUNLOCK(ErrorMessageLock);
    return(-1);
FlipOn:
    if ( i >= 0 ) {
        m = AT.WildMask;
        w = AN.WildValue;
        n = AN.NumWild;
        while ( --n >= 0 ) {
            if ( w[2] == i && *w == SYMTONUM ) {
                *m = 2;
                return(0);
            }
            m++; w += w[1];
        }
        MLOCK(ErrorMessageLock);
        MesPrint(" Bug in AddWild with passing set[i]");
        MUNLOCK(ErrorMessageLock);
/*
        For the moment we want to crash here. That is easier with debugging.
*/
#ifdef WITHPTHREADS
        { WORD *s = 0;
```

```
            *s++ = 1;
        }
#endif
        Terminate(-1);
    }
    return(0);
}

/*
        #] AddWild :
        #[ CheckWild :          WORD CheckWild(oldnumber,type,newnumber,newval)

        Tests whether a wildcard assignment is allowed.
        A return value of zero means that it is allowed (nihil obstat).
        If the variable has been assigned already its existing
        assignment is returned in AN.oldvalue and AN.oldtype, which are
        global variables.

        Note the special problem with name?set[i]. Here we have to pass
        an extra assignment. This cannot be done via globals as we
        call CheckWild sometimes twice before calling AddWild.
        Trick: Check the assignment of the number and if OK put it
        in place, but don't alter the used flag (if needed).
        Then AddWild can alter the used flag but the value is there.
        As long as this trick is `hanging' we turn on the flag:
        `AN.WildReserve' which is either turned off by AddWild or by
        a failing call to CheckWild.

        With ARGTOARG the tensors give the number of arguments
        or-ed with EATTENSOR which is at least 8192.
*/

WORD CheckWild(PHEAD WORD oldnumber, WORD type, WORD newnumber, WORD *newval)
{
    GETBIDENTITY
    WORD *w, *m, *s, n, old2, inset;
    WORD n2, oldval, dirty, i, j, notflag = 0, retblock = 0;
    CBUF *C = cbuf+AT.ebufnum;
    m = AT.WildMask;
    w = AN.WildValue;
    n = AN.NumWild;
    if ( n <= 0 ) { AN.oldtype = -1; AN.WildReserve = 0; return(-1); }
    switch ( type ) {
        case SYMTONUM :
            *newval = newnumber;
            do {
                if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
                    old2 = *w;
                    if ( !*m ) goto TestSet;
                    AN.MaskPointer = m;
                    if ( *w == SYMTONUM  && w[3] == newnumber ) {
                        return(0);
                    }
                    AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
                }
                m++; w += w[1];
            } while ( --n > 0 );
            break;
        case SYMTOSYM :
            *newval = newnumber;
            do {
                if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
                    old2 = *w;
                    if ( *w == SYMTOSYM ) {
                        if ( !*m ) goto TestSet;
                        if ( newnumber >= 0 && (w+4) < AN.WildStop
                        && ( w[4] == FROMSET || w[4] == SETTONUM )
                        && w[7] >= 0 ) goto TestSet;
                        if ( w[3] == newnumber ) return(0);
                    }
                    else {
                        if ( !*m ) goto TestSet;
                    }
                    goto NoM;
                }
                m++; w += w[1];
            } while ( --n > 0 );
            break;
        case SYMTOSUB :
/*
            Now newval contains the pointer to the argument.
*/
            {
/*
                Search for vector or index nature. If so: reject.
*/
                WORD *ss, *sstop, *tt, *ttstop;
```

```
                ss = newval;
                sstop = ss + *ss;
                ss += ARGHEAD;
                while ( ss < sstop ) {
                    tt = ss + *ss;
                    ttstop = tt - ABS(tt[-1]);
                    ss++;
                    while ( ss < ttstop ) {
                        if ( *ss == INDEX ) goto NoMatch;
                        ss += ss[1];
                    }
                    ss = tt;
                }
            }
            do {
                if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
                    old2 = *w;
                    if ( *w == SYMTONUM || *w == SYMTOSYM ) {
                        if ( !*m ) {
                            s = w + w[1];
                            if ( s >= AN.WildStop || *s != SETTONUM )
                                    goto TestSet;
                        }
                    }
                    else if ( *w == SYMTOSUB ) {
                        if ( !*m ) {
                            s = w + w[1];
                            if ( s >= AN.WildStop || *s != SETTONUM )
                                    goto TestSet;
                        }
                        n = *newval - 2;
                        newval += 2;
                        m = C->rhs[w[3]];
                        if ( (C->rhs[w[3]+1] - m - 1) == n ) {
                            while ( n > 0 ) {
                                if ( *m != *newval ) {
                                    m++; newval++; break;
                                }
                                m++; newval++;
                                n--;
                            }
                            if ( n <= 0 ) return(0);
                        }
                    }
                    AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
                }
                m++; w += w[1];
            } while ( --n > 0 );
            break;
        case ARGTOARG :
            do {
                if ( w[2] == oldnumber && *w == ARGTOARG ) {
                    if ( !*m ) return(0);       /* nihil obstat */
                    m = C->rhs[w[3]];
                    if ( ( newnumber & EATTENSOR ) != 0 ) {
                        n = newnumber & ~EATTENSOR;
                        if ( *m != 0 ) {
                            if ( n == *m ) {
                                m++;
                                while ( --n >= 0 ) {
                                    if ( *m != *newval ) {
                                        m++; newval++; break;
                                    }
                                    m++; newval++;
                                }
                                if ( n < 0 ) return(0);
                            }
                        }
                        else {
                            m++;
                            while ( --n >= 0 ) {
                                if ( *newval != m[1] || ( *m != -INDEX
                                && *m != -VECTOR && *m != -SNUMBER ) ) break;
                                m += 2;
                                newval++;
                            }
                            if ( n < 0 && *m == 0 ) return(0);
                        }
                    }
                    else {
                        i = newnumber;
                        if ( *m != 0 ) {    /* Tensor field */
                            if ( *m == i ) {
                                m++;
                                while ( --i >= 0 ) {
                                    if ( *m != newval[1]
                                    || ( *newval != -VECTOR
```

```
                                    && *newval != -INDEX
                                    && *newval != -SNUMBER ) ) break;
                                newval += 2;
                                m++;
                            }
                            if ( i < 0 ) return(0);
                        }
                    }
                    else {
                        m++;
                        s = newval;
                        while ( --i >= 0 ) { NEXTARG(s) }
                        n = WORDDIF(s,newval);
                        while ( --n >= 0 ) {
                            if ( *m != *newval ) {
                                m++; newval++; break;
                            }
                            m++; newval++;
                        }
                        if ( n < 0 && *m == 0 ) return(0);
                    }
                }
                AN.oldtype = *w; AN.oldvalue = w[3]; goto NoMatch;
            }
            m++; w += w[1];
        } while ( --n > 0 );
        break;
    case ARLTOARL :
        do {
            if ( w[2] == oldnumber && *w == ARGTOARG ) {
                WORD **a;
                if ( !*m ) return(0);       /* nihil obstat */
                m = C->rhs[w[3]];
                i = newnumber;
                a = (WORD **)newval;
                if ( *m != 0 ) {    /* Tensor field */
                    if ( *m == i ) {
                        m++;
                        while ( --i >= 0 ) {
                            s = *a++;
                            if ( *m != s[1]
                            || ( *s != -VECTOR
                              && *s != -INDEX
                              && *s != -SNUMBER ) ) break;
                            m++;
                        }
                        if ( i < 0 ) return(0);
                    }
                }
                else {
                    m++;
                    while ( --i >= 0 ) {
                        s = *a++;
                        if ( *s > 0 ) {
                            n = *s;
                            while ( --n >= 0 ) {
                                if ( *s != *m ) {
                                    s++; m++; break;
                                }
                                s++; m++;
                            }
                            if ( n >= 0 ) break;
                        }
                        else if ( *s <= -FUNCTION ) {
                            if ( *s != *m ) {
                                s++; m++; break;
                            }
                            s++; m++;
                        }
                        else {
                            if ( *s != *m ) {
                                s++; m++; break;
                            }
                            s++; m++;
                            if ( *s != *m ) {
                                s++; m++; break;
                            }
                            s++; m++;
                        }
                    }
                    if ( i < 0 && *m == 0 ) return(0);
                }
                AN.oldtype = *w; AN.oldvalue = w[3]; goto NoMatch;
            }
            m++; w += w[1];
        } while ( --n > 0 );
        break;
```

```
        case VECTOSUB :
        case INDTOSUB :
/*
            Now newval contains the pointer to the argument(s).
*/
            {
/*
                Search for vector or index nature. If not so: reject.
*/
                WORD *ss, *sstop, *tt, *ttstop, count, jt;
                ss = newval;
                sstop = ss + *ss;
                ss += ARGHEAD;
                while ( ss < sstop ) {
                    tt = ss + *ss;
                    ttstop = tt - ABS(tt[-1]);
                    ss++;
                    count = 0;
                    while ( ss < ttstop ) {
                        if ( *ss == INDEX ) {
                            jt = ss[1] - 2; ss += 2;
                            while ( --jt >= 0 ) {
                                if ( *ss < MINSPEC ) count++;
                                ss++;
                            }
                        }
                        else ss += ss[1];
                    }
                    if ( count != 1 ) goto NoMatch;
                    ss = tt;
                }
            }
            do {
                if ( w[2] == oldnumber ) {
                    old2 = *w;
                    if ( ( type == VECTOSUB && ( *w == VECTOVEC || *w == VECTOMIN ) )
                    || ( type == INDTOSUB && *w == INDTOIND ) ) {
                        if ( !*m ) goto TestSet;
                        AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
                    }
                    else if ( *w == type ) {
                        if ( !*m ) goto TestSet;
                        if ( type != INDTOIND && type != INDTOSUB ) {   /* Prevent double index */
                            n = *newval - 2;
                            newval += 2;
                            m = C->rhs[w[3]];
                            if ( (C->rhs[w[3]+1] - m - 1) == n ) {
                                while ( n > 0 ) {
                                    if ( *m != *newval ) {
                                        m++; newval++; break;
                                    }
                                    m++; newval++;
                                    n--;
                                }
                                if ( n <= 0 ) return(0);
                            }
                        }
                        AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
                    }
                }
                m++; w += w[1];
            } while ( --n > 0 );
            break;
        default :
            *newval = newnumber;
            do {
                if ( w[2] == oldnumber ) {
                    if ( *w == type ) {
                        old2 = *w;
                        if ( !*m ) goto TestSet;
                        if ( newnumber >= 0 && (w+4) < AN.WildStop &&
                            ( w[4] == FROMSET || w[4] == SETTONUM )
                            && w[7] >= 0 ) goto TestSet;
                        if ( newnumber < 0 && *w == VECTOVEC
                            && (w+4) < AN.WildStop && ( w[4] == FROMSET
                            || w[4] == SETTONUM ) && w[7] >= 0 ) goto TestSet;
/*
                        The next statement kills multiple indices -> vector
*/
                        if ( *w == INDTOIND && w[3] < 0 ) goto NoMatch;
                        if ( w[3] == newnumber ) {
                            if ( *w != FUNTOFUN || newnumber < FUNCTION
                            || functions[newnumber-FUNCTION].spec ==
                            functions[oldnumber-FUNCTION].spec )
                                return(0);
                        }
                        AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
```

```
                    }
                    else if ( ( type == VECTOVEC &&
                            ( *w == VECTOSUB || *w == VECTOMIN ) )
                           || ( type == INDTOIND && *w == INDTOSUB ) ) {
                        if ( *m ) goto NoMatch;
                        old2 = *w;
                        goto TestSet;
                    }
                    else if ( type == VECTOMIN &&
                            ( *w == VECTOSUB || *w == VECTOVEC ) ) {
                        if ( *m ) goto NoMatch;
                        old2 = *w;
                        goto TestSet;
                    }
                }
                m++; w += w[1];
                if ( n > 1 && ( *w == FROMSET
                || *w == SETTONUM ) ) { n--; m++; w += w[1]; }
            } while ( --n > 0 );
            break;
    }
    AN.oldtype = -1;
    AN.oldvalue = -1;
    AN.WildReserve = 0;
    MLOCK(ErrorMessageLock);
    MesPrint("Inconsistency in Wildcard prototype.");
    MUNLOCK(ErrorMessageLock);
    return(-1);
NoMatch:
    AN.WildReserve = 0;
    return(1+retblock);
/*
    Here we test the compatibility with a set specification.
*/
TestSet:
    dirty = *m;
    oldval = w[3];
    w += w[1];
    if ( w < AN.WildStop && ( *w == FROMSET || *w == SETTONUM ) ) {
        WORD k;
        s = w;
        j = w[2]; n2 = w[3];
        if ( j > WILDOFFSET ) {
            j -= 2*WILDOFFSET;
            notflag = 1;
/*
        ???????
*/
            AN.oldtype = -1;
            AN.oldvalue = -1;
        }
        if ( j < AM.NumFixedSets ) {    /* special set */
            retblock = 1;
            switch ( j ) {
                case POS_:
                    if ( type != SYMTONUM ||
                    newnumber <= 0 ) goto NoMnot;
                    break;
                case POS0_:
                    if ( type != SYMTONUM ||
                    newnumber < 0 ) goto NoMnot;
                    break;
                case NEG_:
                    if ( type != SYMTONUM ||
                    newnumber >= 0 ) goto NoMnot;
                    break;
                case NEG0_:
                    if ( type != SYMTONUM ||
                    newnumber > 0 ) goto NoMnot;
                    break;
                case EVEN_:
                    if ( type != SYMTONUM ||
                    ( newnumber & 1 ) != 0 ) goto NoMnot;
                    break;
                case ODD_:
                    if ( type != SYMTONUM ||
                    ( newnumber & 1 ) == 0 ) goto NoMnot;
                    break;
                case Z_:
                    if ( type != SYMTONUM ) goto NoMnot;
                    break;
                case SYMBOL_:
                    if ( type != SYMTOSYM ) goto NoMnot;
                    break;
                case FIXED_:
                    if ( type != INDTOIND ||
                    newnumber >= AM.OffsetIndex ||
```

```
                    newnumber < 0 ) goto NoMnot;
                    break;
                case INDEX_:
                    if ( type != INDTOIND ||
                    newnumber < 0 ) goto NoMnot;
                    break;
                case Q_:
                    if ( type == SYMTONUM ) break;
                    if ( type == SYMTOSUB ) {
                        WORD *ss, *sstop;
                        ss = newval;
                        sstop = ss + *ss;
                        ss += ARGHEAD;
                        if ( ss >= sstop ) break;
                        if ( ss + *ss < sstop ) goto NoMnot;
                        if ( ABS(sstop[-1]) == ss[0]-1 ) break;
                    }
                    goto NoMnot;
                case DUMMYINDEX_:
                    if ( type != INDTOIND ||
                    newnumber < AM.IndDum || newnumber >= AM.IndDum+MAXDUMMIES ) goto NoMnot;
                    break;
                case VECTOR_:
                    if ( type != VECTOVEC ) goto NoMnot;
                    break;
                default:
                    goto NoMnot;
            }
Mnot:
            if ( notflag ) goto NoM;
            return(0);
NoMnot:
            if ( !notflag ) goto NoM;
            return(0);
        }
        else if ( Sets[j].type == CRANGE ) {
            if ( ( type == SYMTONUM )
            || ( type == INDTOIND && ( newnumber > 0
             && newnumber <= AM.OffsetIndex ) ) ) {
                if ( Sets[j].first < MAXPOWER ) {
                    if ( newnumber >= Sets[j].first ) goto NoMnot;
                }
                else if ( Sets[j].first < 3*MAXPOWER ) {
                    if ( newnumber+2*MAXPOWER > Sets[j].first ) goto NoMnot;
                }
                if ( Sets[j].last > -MAXPOWER ) {
                    if ( newnumber <= Sets[j].last ) goto NoMnot;
                }
                else if ( Sets[j].last > -3*MAXPOWER ) {
                    if ( newnumber-2*MAXPOWER < Sets[j].last ) goto NoMnot;
                }
                goto Mnot;
            }
            goto NoMnot;
        }
        w = SetElements + Sets[j].first;
        m = SetElements + Sets[j].last;
        i = 1;
        if ( Sets[j].type == -1 || Sets[j].type == CNUMBER ) { do {
            if ( notflag ) {
                switch ( type ) {
                    case SYMTOSYM:
                        if ( Sets[j].type == CNUMBER ) {}
                        else {
                            if ( *w == newnumber ) goto NoMatch;
                        }
                        break;
                    case SYMTONUM:
                    case INDTOIND:
                        if ( *w == newnumber ) goto NoMatch;
                        break;
                    default:
                        break;
                }
            }
            else if ( type != SYMTONUM && type != INDTOIND
            && type != SYMTOSYM ) goto NoMatch;
            else if ( type == SYMTOSYM && Sets[j].type == CNUMBER ) goto NoMatch;
            else if ( *w == newnumber ) {
                if ( *s == SETTONUM ) {
                    if ( n2 == oldnumber && type
                    <= SYMTOSUB ) goto NoMatch;
                    m = AT.WildMask;
                    w = AN.WildValue;
                    n = AN.NumWild;
                    while ( --n >= 0 ) {
                        if ( w[2] == n2 && *w <= SYMTOSUB ) {
```

```
                            if ( !*m ) {
                                *w = SYMTONUM;
                                w[3] = i;
                                AN.WildReserve = 1;
                                return(0);
                            }
                            if ( *w != SYMTONUM )
                                goto NoMatch;
                            if ( w[3] == i ) return(0);
                            i = w[3];
                            j = (SetElements + Sets[j].first)[i];
                            if ( j == n2 ) return(0);
                            goto NoMatch;
                        }
                        m++; w += w[1];
                    }
                }
                else if ( n2 >= 0 ) {
                    *newval = *(w - Sets[j].first + Sets[n2].first);
                    if ( *newval > MAXPOWER ) *newval -= 2*MAXPOWER;
                    if ( dirty && *newval != oldval ) {
                        *newval = oldval; goto NoMatch;
                    }
                }
                return(0);
            }
            i++;
        } while ( ++w < m ); }
        else { do {
            inset = *w;
            if ( notflag ) {
            switch ( type ) {
                case SYMTONUM:
                case SYMTOSYM:
                    if ( ( type == SYMTOSYM && *w == newnumber )
                    || ( type == SYMTONUM && *w-2*MAXPOWER == newnumber ) ) {
                        goto NoMatch;
                    }
                case SYMTOSUB:
                    if ( *w < 0 ) {
                        WORD *mm = AT.WildMask, *mmm, *part;
                        WORD *ww = AN.WildValue;
                        WORD nn = AN.NumWild;
                        k = -*w;
                        while ( --nn >= 0 ) {
                            if ( *mm && ww[2] == k && ww[0] == type ) {
                                if ( type != SYMTOSUB ) {
                                    if ( ww[3] == newnumber ) goto NoMatch;
                                }
                                else {
                                    mmm = C->rhs[ww[3]];
                                    nn = *newval-2;
                                    part = newval+2;
                                    if ( (C->rhs[ww[3]+1]-mmm-1) == nn ) {
                                        while ( --nn >= 0 ) {
                                            if ( *mmm != *part ) {
                                                mmm++; part++; break;
                                            }
                                            mmm++; part++;
                                        }
                                        if ( nn < 0 ) goto NoMatch;
                                    }
                                }
                                break;
                            }
                            mm++; ww += ww[1];
                        }
                    }
                    break;
                case VECTOMIN:
                    if ( type == VECTOMIN ) {
                        if ( inset >= AM.OffsetVector ) { i++; continue; }
                        inset += WILDMASK;
                    }
                case VECTOVEC:
                    if ( inset == newnumber ) goto NoMatch;
                case VECTOSUB:
                    if ( inset - WILDOFFSET >= AM.OffsetVector ) {
                        WORD *mm = AT.WildMask, *mmm, *part;
                        WORD *ww = AN.WildValue;
                        WORD nn = AN.NumWild;
                        k = inset - WILDOFFSET;
                        while ( --nn >= 0 ) {
                            if ( *mm && ww[2] == k && ww[0] == type ) {
                                if ( type == VECTOVEC ) {
                                    if ( ww[3] == newnumber ) goto NoMatch;
                                }
```

```
                                else {
                                    mmm = C->rhs[ww[3]];
                                    nn = *newval-2;
                                    part = newval+2;
                                    if ( (C->rhs[ww[3]+1]-mmm-1) == nn ) {
                                        while ( --nn >= 0 ) {
                                            if ( *mmm != *part ) {
                                                mmm++; part++; break;
                                            }
                                            mmm++; part++;
                                        }
                                        if ( nn < 0 ) goto NoMatch;
                                    }
                                }
                                break;
                            }
                            mm++; ww += ww[1];
                        }
                    }
                    break;
                case INDTOIND:
                    if ( *w == newnumber ) goto NoMatch;
                case INDTOSUB:
                    if ( *w - (WORD)WILDMASK >= AM.OffsetIndex ) {
                        WORD *mm = AT.WildMask, *mmm, *part;
                        WORD *ww = AN.WildValue;
                        WORD nn = AN.NumWild;
                        k = *w - WILDMASK;
                        while ( --nn >= 0 ) {
                            if ( *mm && ww[2] == k && ww[0] == type ) {
                                if ( type == INDTOIND ) {
                                    if ( ww[3] == newnumber ) goto NoMatch;
                                }
                                else {
                                    mmm = C->rhs[ww[3]];
                                    nn = *newval-2;
                                    part = newval+2;
                                    if ( (C->rhs[ww[3]+1]-mmm-1) == nn ) {
                                        while ( --nn >= 0 ) {
                                            if ( *mmm != *part ) {
                                                mmm++; part++; break;
                                            }
                                            mmm++; part++;
                                        }
                                        if ( nn < 0 ) goto NoMatch;
                                    }
                                }
                                break;
                            }
                            mm++; ww += ww[1];
                        }
                    }
                    break;
                case FUNTOFUN:
                    if ( *w == newnumber ) goto NoMatch;
                    if ( ( type == FUNTOFUN &&
                    ( k = *w - WILDMASK ) > FUNCTION ) ) {
                        WORD *mm = AT.WildMask;
                        WORD *ww = AN.WildValue;
                        WORD nn = AN.NumWild;
                        while ( --nn >= 0 ) {
                            if ( *mm && ww[2] == k && ww[0] == type ) {
                                if ( ww[3] == newnumber ) goto NoMatch;
                                break;
                            }
                            mm++; ww += ww[1];
                        }
                    }
                default:
                    break;
            }
            }
            else {
              if ( type == VECTOMIN ) {
                if ( inset >= AM.OffsetVector ) { i++; continue; }
                inset += WILDMASK;
              }
              if ( ( inset == newnumber && type != SYMTONUM ) ||
              ( type == SYMTONUM && inset-2*MAXPOWER == newnumber ) ) {
                if ( *s == SETTONUM ) {
                    if ( n2 == oldnumber && type
                    <= SYMTOSUB ) goto NoMatch;
                    m = AT.WildMask;
                    w = AN.WildValue;
                    n = AN.NumWild;
                    while ( --n >= 0 ) {
                        if ( w[2] == n2 && *w <= SYMTOSUB ) {
```

```
                            if ( !*m ) {
                                *w = SYMTONUM;
                                w[3] = i;
                                AN.WildReserve = 1;
                                return(0);
                            }
                            if ( *w != SYMTONUM )
                                goto NoMatch;
                            if ( w[3] == i ) return(0);
                            i = w[3];
                            j = (SetElements + Sets[j].first)[i];
                            if ( j == n2 ) return(0);
                            goto NoMatch;
                        }
                        m++; w += w[1];
                    }
                }
                else if ( n2 >= 0 ) {
                    *newval = *(w - Sets[j].first + Sets[n2].first);
                    if ( *newval > MAXPOWER ) *newval -= 2*MAXPOWER;
                    if ( dirty && *newval != oldval ) {
                        *newval = oldval; goto NoMatch;
                    }
                }
                return(0);
            }
        }
        i++;
    } while ( ++w < m ); }
    if ( notflag ) return(0);
    AN.oldtype = old2; AN.oldvalue = oldval; goto NoMatch;
  }
  else { return(0); }

NoM:
    AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
}

/*
        #] CheckWild :
    #] Wildcards :
    #[ DenToFunction :

    Renames the denominator function into a function with the given number.
    For the syntax see   Denominators,function;
*/

int DenToFunction(WORD *term, WORD numfun)
{
    int action = 0;
    WORD *t, *tstop, *tnext, *arg, *argstop, *targ;
    t = term+1;
    tstop = term + *term; tstop -= ABS(tstop[-1]);
    while ( t < tstop ) {
        if ( *t == DENOMINATOR ) {
            *t = numfun; t[2] |= DIRTYFLAG; action = 1;
        }
        tnext = t + t[1];
        if ( *t >= FUNCTION && functions[*t-FUNCTION].spec == 0 ) {
            arg = t + FUNHEAD;
            while ( arg < tnext ) {
                if ( *arg > 0 ) {
                    targ = arg + ARGHEAD; argstop = arg + *arg;
                    while ( targ < argstop ) {
                        if ( DenToFunction(targ,numfun) ) {
                            arg[1] |= DIRTYFLAG; t[2] |= DIRTYFLAG; action = 1;
                        }
                        targ += *targ;
                    }
                    arg = argstop;
                }
                else if ( *arg <= -FUNCTION ) arg++;
                else arg += 2;
            }
        }
        t = tnext;
    }
    return(action);
}

/*
    #] DenToFunction :
*/
```

Index

_CHECK
parallel.c, 1420
__CHECK
parallel.c, 1420

A
inivar.h, 1135
variable.h, 2412
a
MODNUM, 129
a3
TrAcEs, 267
a4
TrAcEs, 267
ABS
declare.h, 574
ABSFUNCTION
ftypes.h, 1015
AC
variable.h, 2410
accu
TrAcEn, 262
TrAcEs, 264
AccumGCD
declare.h, 597
reken.c, 1791
accup
TrAcEn, 262
TrAcEs, 264
ACOSFUNCTION
ftypes.h, 1019
ACOSHFUNCTION
ftypes.h, 1020
active
FiLe, 75
PF_BUFFER, 188
activenames
C_const, 23
ad
TrAcEs, 267
add
COST, 51
Add2Com
declare.h, 587
Add2ComStrings
compcomm.c, 402
declare.h, 690
ADD2POS
declare.h, 585
Add3Com
declare.h, 588
Add4Com
declare.h, 588
Add5Com
declare.h, 588
ADDARG
ftypes.h, 1064
AddArgs
declare.h, 597
sort.c, 1955
AddArrayIndex
declare.h, 596
sch.c, 1889
AddCoef
declare.h, 598
sort.c, 1953
AddComString
compcomm.c, 402
declare.h, 717
AddDictionary
declare.h, 767
dict.c, 792
AddDollar
declare.h, 658
names.c, 1266
AddDubious
declare.h, 658
names.c, 1266
AddExpression
declare.h, 657
names.c, 1267
AddFunction
declare.h, 659
names.c, 1263
AddIndex
declare.h, 659
names.c, 1262
AddLHS
comtool.c, 539
declare.h, 686
AddLineFeed
declare.h, 575
AddLong
declare.h, 598
reken.c, 1791
AddName
declare.h, 655
names.c, 1259
AddNtoC

comtool.c, 541
declare.h, 688
AddNtoL
comtool.c, 540
declare.h, 687
AddObject
declare.h, 742
minos.c, 1172
AddPLon
declare.h, 598
reken.c, 1791
AddPoly
declare.h, 599
sort.c, 1954
ADDPOS
declare.h, 584
AddPotModdollar
declare.h, 728
dollar.c, 816
AddRat
declare.h, 600
reken.c, 1790
AddRHS
comtool.c, 539
declare.h, 687
AddSet
declare.h, 660
names.c, 1265
AddSymbol
declare.h, 657
names.c, 1262
AddTableName
declare.h, 743
minos.c, 1172
AddToCB
declare.h, 579
AddToCom
compcomm.c, 402
declare.h, 690
AddToDictionary
declare.h, 766
dict.c, 792
AddToDollarBuffer
declare.h, 723
dollar.c, 809
AddToIndex
declare.h, 741
minos.c, 1172
AddToLine
declare.h, 600
sch.c, 1887
AddToPreTypes
declare.h, 686
pre.c, 1582
AddToScratch
declare.h, 770
store.c, 2052
AddToString
declare.h, 679
tools.c, 2299
AddVector
declare.h, 658
names.c, 1263
AddWild
declare.h, 600
wildcard.c, 2428
AddWildcardName
declare.h, 685
names.c, 1261
AdjustRenumScratch
declare.h, 618
reshuf.c, 1845
aebufnum
T_const, 246
afterwards
StreaM, 235
alfatable1
compiler.c, 502
alignof
form3.h, 950
ALLARGS
ftypes.h, 1061
allbufnum
T_const, 246
ALLDIRTY
ftypes.h, 1037
ALLFUNPOWERS
ftypes.h, 1057
ALLGLOBALS
structs.h, 2125
AllGlobals, 5
C, 6
M, 6
N, 6
O, 7
P, 7
R, 6
S, 6
T, 7
X, 7
AllocFileHandle
declare.h, 731
setfile.c, 1926
AllocSetups
setfile.c, 1925
AllocSort
declare.h, 668
setfile.c, 1925
AllocSortFileName
declare.h, 668
setfile.c, 1926
AllowDelay
P_const, 178
allsign
TrAcEn, 263
TrAcEs, 268

ALLVARIABLES
ftypes.h, 997
ALSODOLLARS
ftypes.h, 1060
ALSOFUNARGS
ftypes.h, 1060
ALSOPOWERS
ftypes.h, 1060
ALSOREVERSE
ftypes.h, 1008
AltCollectFun
C_const, 42
ALTSEPARATOR
fwin.h, 1110
unix.h, 2398
AM
variable.h, 2410
AN
variable.h, 2410
ANDCOND
ftypes.h, 1054
ANNOUNCE
checkpoint.c, 327
ANTISYMMETRIC
ftypes.h, 1049
ANYTYPE
ftypes.h, 998
AO
variable.h, 2411
AP
variable.h, 2411
Apply
declare.h, 610
tables.c, 2183
ApplyExec
declare.h, 610
tables.c, 2183
ApplyReset
declare.h, 610
tables.c, 2184
AR
variable.h, 2411
AreArgsEqual
declare.h, 666
tools.c, 2305
argaddress
N_const, 135
ARGBUFFER, 7
buffer, 8
dollar, 8
dummy, 8
size, 8
type, 8
ArgDotproductMerge
argument.c, 278
declare.h, 693
ArgFactorize
argument.c, 277
declare.h, 690
ARGFIELD
ftypes.h, 1011
ARGHEAD
ftypes.h, 1037
arglevel
C_const, 40
arglist
N_const, 140
arglistsize
N_const, 143
argnames
pReVaR, 195
ARGRANGE
ftypes.h, 1062
args
PaRtI, 183
argsize
NeStInG, 154
argstack
C_const, 29
argsumcheck
C_const, 39
ArgSymbolMerge
argument.c, 278
declare.h, 693
argtail
TaBlEs, 257
ARGTOARG
ftypes.h, 1035
argument.c, 275, 281
ArgDotproductMerge, 278
ArgFactorize, 277
ArgSymbolMerge, 278
ArgumentExplode, 276
ArgumentImplode, 276
CleanupArgCache, 277
execarg, 276
execterm, 276
FindArg, 277
InsertArg, 277
MakeInteger, 279
MakeMod, 279
NEWORDER, 276
SortWeights, 280
TakeArgContent, 278
ArgumentExplode
argument.c, 276
declare.h, 738
ArgumentImplode
argument.c, 276
declare.h, 738
ARGWILD
ftypes.h, 1010
ARLTOARL
ftypes.h, 1035
arraysize
POLYMOD, 193

AS
variable.h, 2411
ASINFUNCTION
ftypes.h, 1019
ASINHFUNCTION
ftypes.h, 1020
AssignDollar
declare.h, 722
dollar.c, 809
AT
variable.h, 2411
ATAN2FUNCTION
ftypes.h, 1019
ATANFUNCTION
ftypes.h, 1019
ATANHFUNCTION
ftypes.h, 1020
ATENDOFMODULE
ftypes.h, 996
atstartup
M_const, 126
AutoDeclareFlag
C_const, 31
AutoFunctionList
C_const, 22
autofunctions
variable.h, 2409
AutoIndexList
C_const, 22
autoindices
variable.h, 2409
AUTONAMES
ftypes.h, 1061
autonames
C_const, 22
AutoSymbolList
C_const, 22
autosymbols
variable.h, 2409
AutoVectorList
C_const, 22
autovectors
variable.h, 2409
AWORDMASK
form3.h, 949
AX
variable.h, 2411

BACKINOUT
declare.h, 579
balance
NaMeNode, 150
Balancing
S_const, 210
BASECODE
ftypes.h, 1064
BASEPOSITION
declare.h, 586
Bernoulli
declare.h, 615
reken.c, 1798
BERNOULLIFUNCTION
ftypes.h, 1016
bernoullis
T_const, 243
BHEAD
ftypes.h, 987
BHEAD0
ftypes.h, 987
BigLong
declare.h, 600
reken.c, 1792
BinomGen
declare.h, 600
ratio.c, 1739
BINOMIAL
ftypes.h, 1014
bit_0
bit_field, 9
bit_1
bit_field, 9
bit_2
bit_field, 9
bit_3
bit_field, 9
bit_4
bit_field, 9
bit_5
bit_field, 10
bit_6
bit_field, 10
bit_7
bit_field, 10
bit_field, 9
bit_0, 9
bit_1, 9
bit_2, 9
bit_3, 9
bit_4, 9
bit_5, 10
bit_6, 10
bit_7, 10
blnce
tree, 270
BlockSpaces
O_const, 162
boomlijst
CbUf, 48
TaBlEs, 257
BottomLevel
C_const, 33
bounds
TaBlEs, 259
BrackBuf
T_const, 241
bracket
BrAcKeTiNdEx, 11

O_const, 159
bracketbuffer
BrAcKeTiNfO, 12
bracketbuffersize
BrAcKeTiNfO, 13
BRACKETCURRENTEXPR
execute.c, 862
BracketFactors
M_const, 127
bracketfill
BrAcKeTiNfO, 13
BrAcKeTiNdEx, 10
bracket, 11
next, 11
start, 11
termsinbracket, 11
bracketindexflag
C_const, 36
T_const, 248
BrAcKeTiNfO, 12
bracketbuffer, 12
bracketbuffersize, 13
bracketfill, 13
indexbuffer, 12
indexbuffersize, 13
indexfill, 13
SortType, 13
bracketinfo
ExPrEsSiOn, 68
T_const, 245
BracketNormalize
C_const, 36
declare.h, 628
normal.c, 1307
BracketOn
R_const, 203
BRACKETOTHEREXPR
execute.c, 862
buff
PF_BUFFER, 187
Buffer
CbUf, 47
buffer
ARGBUFFER, 8
INSIDEINFO, 98
longMultiStruct, 103
PRELOAD, 194
StOrEcAcHe, 228
StreaM, 233
bufferedInd
O_const, 165
BufferForOutput
inivar.h, 1136
buffernum
SuBbUf, 236
bufferposition
StreaM, 234
buffers
TaBlEs, 257
buffersfill
TaBlEs, 260
BufferSize
CbUf, 49
buffersize
StreaM, 234
bufferssize
TaBlEs, 260
bufIPstruct, 14
e, 14
i, 14
bufnum
TaBlEs, 260
bufpos
longMultiStruct, 103
BUG
variable.h, 2410
bugtool.c, 321
ExprStatus, 321
BUILTINDOLLARS
ftypes.h, 1024
BUILTININDICES
ftypes.h, 1024
BUILTINSYMBOLS
ftypes.h, 1023
BUILTINVECTORS
ftypes.h, 1024
bzero
form3.h, 954

C
AllGlobals, 6
C_const, 15
activenames, 23
AltCollectFun, 42
arglevel, 40
argstack, 29
argsumcheck, 39
AutoDeclareFlag, 31
AutoFunctionList, 22
AutoIndexList, 22
autonames, 22
AutoSymbolList, 22
AutoVectorList, 22
BottomLevel, 33
bracketindexflag, 36
BracketNormalize, 36
cbufList, 22
cbufnum, 30
ChannelList, 20
CheckpointFlag, 39
CheckpointInterval, 30
CheckpointRunAfter, 28
CheckpointRunBefore, 28
CheckpointStamp, 30
cmod, 24
CModule, 30
Cnumpows, 42

CodesFlag, 34
CollectFun, 42
CollectPercentage, 44
ComDefer, 42
Commercial, 45
CommuteInSet, 29
CompileLevel, 33
compiletype, 31
CurrentStream, 23
debugFlags, 45
DidClean, 43
DirtPow, 41
dollarnames, 19
doloopplevel, 39
doloopnest, 28
doloopstack, 28
doloopstacksize, 39
DubiousList, 20
DumNum, 41
dumnumflag, 36
endoftokens, 27
ExpressionList, 20
exprfillwarning, 37
exprnames, 19
extrasym, 28
extrasymbols, 45
ffbufnum, 38
FinalStats, 35
firstconstindex, 31
firstctypemessage, 33
FixIndices, 26
Fortran90Kind, 27
FunctionList, 20
Functions, 23
funpowers, 34
halfmod, 24
HideLevel, 43
iBuffer, 25
iBufferSize, 29
idoption, 33
IfCount, 25
IfHeap, 25
IfLevel, 43
IfStack, 25
IfSumCheck, 28
IndexList, 20
Indices, 23
inexprlevel, 40
inexprstack, 29
inexprsumcheck, 40
InnerTest, 45
inparallelflag, 36
insidefirst, 31
insidelevel, 40
insidestack, 29
insidesumcheck, 40
iPointer, 25
IsFortran90, 38
iStop, 26
LabelNames, 26
Labels, 26
lDefDim, 32
lDefDim4, 32
lhdollarflag, 37
LineLength, 43
LogHandle, 43
lPolyFun, 44
lPolyFunExp, 44
lPolyFunInv, 44
lPolyFunPow, 44
lPolyFunType, 44
lPolyFunVar, 44
lSortType, 35
lUniTrace, 40
lUnitTrace, 42
MaxIf, 32
MaxLabels, 32
MaxNumStreams, 33
maxtermlevel, 36
MemDebugFlag, 38
minsidefirst, 31
modinverses, 27
modmode, 41
ModOptDolList, 21
modpowers, 24
mparallelflag, 36
mProcessBucketSize, 30
MultiBracketBuf, 28
MultiBracketLevels, 38
MustTestTable, 41
NamesFlag, 34
ncmod, 41
nhalfmod, 41
NoCompress, 37
NoShowInput, 31
npowmod, 41
NumLabels, 32
NumStreams, 33
NumWildcardNames, 32
NwildC, 42
OldFactArgFlag, 38
OldGCDflag, 38
OldParallelStats, 35
origin, 39
OutNumberType, 43
OutputMode, 42
OutputSpaces, 42
outsidefun, 34
parallelflag, 36
partodoflag, 37
PolyRatFunChanged, 45
PotModDolList, 21
powmod, 24
ProcessBucketSize, 30
ProcessStats, 36
properorderflag, 37

ProtoType, 25
RepLevel, 40
RepSumCheck, 40
separators, 19
SetElementList, 21
SetList, 21
SetupFlag, 34
ShortStats, 31
ShortStatsMax, 45
SizeCommuteInSet, 39
SortType, 35
StatsFlag, 34
StoreFileSize, 19
StoreHandle, 43
Streams, 23
SymbolList, 21
Symbols, 23
SymChangeFlag, 44
TableBaseList, 22
tablecheck, 33
tablefilling, 37
termlevel, 41
termsortstack, 24
termstack, 24
termsumcheck, 26
TestValue, 29
ThreadBalancing, 35
ThreadBucketSize, 30
ThreadsFlag, 35
ThreadSortFileSynch, 35
ThreadStats, 35
ToBeInFactors, 45
tokenarglevel, 27
tokens, 27
TokensWriteFlag, 33
topolynomialflag, 38
toptokens, 27
TransEname, 29
UnsureDollarMode, 34
varnames, 20
vectorlikeLHS, 39
VectorList, 21
Vectors, 23
vetofilling, 37
vetotablebasefill, 37
WarnFlag, 34
WhileLevel, 43
WildC, 25
WildcardBufferSize, 32
WildcardNames, 26
wildflag, 32
WTimeStatsFlag, 38
cache_buffer
checkpoint.c, 330
cache_fill
checkpoint.c, 330
CACHE_SIZE
checkpoint.c, 324
CACHED_SNAPSHOT
checkpoint.c, 324
CacheNumberFree
declare.h, 590
CacheNumberMalloc
declare.h, 589
CacheNumberMallocAddMemory
declare.h, 753
tools.c, 2303
CacheNumberMemHeap
T_const, 244
CacheNumberMemMax
T_const, 248
CacheNumberMemTop
T_const, 248
CanCommu
CbUf, 48
CARRIAGERETURN
fwin.h, 1109
unix.h, 2397
CatchDollar
declare.h, 722
dollar.c, 809
CBUF
structs.h, 2123
CbUf, 46
boomlijst, 48
Buffer, 47
BufferSize, 49
CanCommu, 48
dimension, 48
lhs, 47
maxlhs, 49
maxrhs, 49
MaxTreeSize, 50
mnumlhs, 49
mnumrhs, 49
numdum, 48
numlhs, 49
numrhs, 49
NumTerms, 48
numtree, 50
Pointer, 47
rhs, 47
rootnum, 50
Top, 47
cbuf
variable.h, 2408
cBuffer
sOrT, 219
cBufferSize
sOrT, 223
cbufList
C_const, 22
cbufnum
C_const, 30
cComChar
P_const, 180

CDELETE
ftypes.h, 998
CDELTA
ftypes.h, 999
CDOLLAR
ftypes.h, 999
CDOTPRODUCT
ftypes.h, 999
CDOUBLEDOT
ftypes.h, 1000
CDUBIOUS
ftypes.h, 999
CEXPRESSION
ftypes.h, 999
CFD
minos.c, 1168
CFUN
ftypes.h, 1071
CFUNCTION
ftypes.h, 998
ChainIn
declare.h, 738
function.c, 1086
ChainOut
declare.h, 738
function.c, 1086
CHANNEL
structs.h, 2123
ChAnNeL, 50
handle, 51
name, 51
ChannelList
C_const, 20
channels
variable.h, 2408
characters
DICTIONARY, 56
CharOut
declare.h, 669
pre.c, 1571
chartype
variable.h, 2404
CHECK
parallel.c, 1420
check_memory
poly, 191
checkpoint.c, 322, 331
ANNOUNCE, 327
cache_buffer, 330
cache_fill, 330
CACHE_SIZE, 324
CACHED_SNAPSHOT, 324
CheckRecoveryFile, 328
DeleteRecoveryFile, 328
DoCheckpoint, 330
DoRecovery, 329
flush_cache, 329
fwrite_cached, 328
InitRecovery, 328
R_COPY_B, 325
R_COPY_LIST, 326
R_COPY_NAMETREE, 326
R_COPY_S, 325
R_FREE, 324
R_FREE_NAMETREE, 324
R_FREE_STREAM, 324
R_SET, 325
RecoveryFilename, 328
S_FLUSH_B, 325
S_WRITE_B, 325
S_WRITE_DOLLAR, 327
S_WRITE_LIST, 326
S_WRITE_NAMETREE, 327
S_WRITE_S, 326
CheckpointFlag
C_const, 39
CheckpointInterval
C_const, 30
CheckpointRunAfter
C_const, 28
CheckpointRunBefore
C_const, 28
CheckpointStamp
C_const, 30
CHECKPOLY
token.c, 2263
CheckPower
O_const, 161
CheckRecoveryFile
checkpoint.c, 328
CheckTableDeclarations
declare.h, 609
tables.c, 2181
CheckWild
declare.h, 601
wildcard.c, 2428
CHISHOLM
ftypes.h, 1008
Chisholm
declare.h, 601
opera.c, 1387
CINDEX
ftypes.h, 998
CleanDollarFactors
declare.h, 726
dollar.c, 814
CleanExpr
declare.h, 601
execute.c, 862
CLEANFLAG
ftypes.h, 1036
CleanUp
declare.h, 601
startup.c, 2026
CleanupArgCache
argument.c, 277

declare.h, 693
CleanUpSort
declare.h, 731
sort.c, 1961
CleanupTerm
declare.h, 734
execute.c, 864
CLEAR
ftypes.h, 1054
ClearBracketIndex
declare.h, 736
index.c, 1126
clearcbuf
comtool.c, 538
declare.h, 730
ClearMacro
declare.h, 671
pre.c, 1575
CLEARMODULE
ftypes.h, 990
clearnamefill
NaMeTree, 153
clearnodefill
NaMeTree, 153
ClearPushback
pre.c, 1571
ClearSpectators
declare.h, 770
spectator.c, 2014
ClearTableTree
declare.h, 729
tables.c, 2179
ClearTree
comtool.c, 542
declare.h, 722
ClearWild
declare.h, 601
wildcard.c, 2428
ClearWildcardNames
names.c, 1261
closeAllExternalChannels
extcmd.c, 897
CloseChannel
declare.h, 664
tools.c, 2298
closeExternalChannel
declare.h, 745
extcmd.c, 896
CloseFile
declare.h, 661
tools.c, 2295
CloseStream
declare.h, 654
tools.c, 2293
cmod
C_const, 24
N_const, 140
CMODE
ftypes.h, 1006
CModule
C_const, 30
CNUMBER
ftypes.h, 999
Cnumlhs
R_const, 202
Cnumpows
C_const, 42
CoAntiBracket
compcomm.c, 409
declare.h, 695
CoAntiPutInside
compcomm.c, 416
declare.h, 697
CoAntiSymmetrize
compcomm.c, 405
declare.h, 695
CoApply
declare.h, 698
tables.c, 2183
CoArgExplode
compcomm.c, 413
declare.h, 695
CoArgImplode
compcomm.c, 413
declare.h, 695
CoArgToExtraSymbol
compcomm.c, 414
declare.h, 760
CoArgument
compcomm.c, 403
declare.h, 695
CoAssign
comexpr.c, 371
declare.h, 720
CoAuto
declare.h, 715
names.c, 1266
CoBracket
compcomm.c, 409
declare.h, 696
CoCFunction
declare.h, 697
names.c, 1264
CoChainin
compcomm.c, 407
declare.h, 714
CoChainout
compcomm.c, 407
declare.h, 714
CoChisholm
compcomm.c, 407
declare.h, 714
CoClearTable
compcomm.c, 413
declare.h, 714
CoCollect

compcomm.c, 397
declare.h, 697
CoCommuteInSet
declare.h, 659
names.c, 1263
CoCompress
compcomm.c, 398
declare.h, 697
CoContract
compcomm.c, 403
declare.h, 698
CoCopySpectator
declare.h, 769
spectator.c, 2014
CoCreateSpectator
declare.h, 769
spectator.c, 2013
CoCTable
declare.h, 713
names.c, 1265
CoCTensor
declare.h, 697
names.c, 1264
CoCycleSymmetrize
compcomm.c, 405
declare.h, 698
code
OPTIMIZERESULT, 172
CoDeallocateTable
comexpr.c, 371
declare.h, 726
CodeFactors
compiler.c, 501
declare.h, 721
CodeGenerator
compiler.c, 501
declare.h, 721
CoDelete
compcomm.c, 399
declare.h, 698
CoDenominators
compcomm.c, 413
declare.h, 698
CodesFlag
C_const, 34
codesize
OPTIMIZERESULT, 172
CodeToLine
declare.h, 595
sch.c, 1888
CoDimension
declare.h, 698
names.c, 1263
CoDiscard
compcomm.c, 402
declare.h, 699
CoDisorder
comexpr.c, 369
declare.h, 699
CoDo
compcomm.c, 414
declare.h, 760
CoDrop
compcomm.c, 401
declare.h, 699
CoDropCoefficient
compcomm.c, 413
declare.h, 699
CoDropSymbols
compcomm.c, 413
declare.h, 699
COEFFI
ftypes.h, 1052
COEFFICIENTSONLY
ftypes.h, 1060
COEFFSYMBOL
ftypes.h, 1023
coefs
POLYMOD, 192
CoElse
compcomm.c, 410
declare.h, 699
CoElseIf
compcomm.c, 410
declare.h, 699
CoEmptySpectator
declare.h, 769
spectator.c, 2014
CoEndArgument
compcomm.c, 403
declare.h, 700
CoEndDo
compcomm.c, 415
declare.h, 760
CoEndIf
compcomm.c, 410
declare.h, 700
CoEndInExpression
compcomm.c, 408
declare.h, 696
CoEndInside
compcomm.c, 404
declare.h, 700
CoEndRepeat
compcomm.c, 409
declare.h, 700
CoEndTerm
compcomm.c, 411
declare.h, 700
CoEndWhile
compcomm.c, 410
declare.h, 700
CoExit
compcomm.c, 407
declare.h, 700
CoExtraSymbols

compcomm.c, 414
declare.h, 760
CoFactArg
compcomm.c, 404
declare.h, 701
CoFactDollar
compcomm.c, 415
declare.h, 701
CoFactorize
compcomm.c, 415
declare.h, 701
CoFill
comexpr.c, 370
declare.h, 702
CoFillExpression
comexpr.c, 370
declare.h, 702
CoFindLoop
compcomm.c, 411
declare.h, 724
CoFixIndex
compcomm.c, 399
declare.h, 702
CoFlags
compcomm.c, 398
declare.h, 707
CoFormat
compcomm.c, 399
declare.h, 702
CoFromPolynomial
compcomm.c, 414
declare.h, 759
CoFunction
declare.h, 659
names.c, 1263
CoFunPowers
compcomm.c, 411
declare.h, 724
CoGlobal
comexpr.c, 368
declare.h, 702
CoGlobalFactorized
comexpr.c, 368
declare.h, 702
CoGoTo
compcomm.c, 403
declare.h, 702
CoHide
compcomm.c, 401
declare.h, 710
CoId
comexpr.c, 368
declare.h, 703
CoIdExpression
comexpr.c, 370
declare.h, 720
CoIdNew
comexpr.c, 369
declare.h, 703
CoIdOld
comexpr.c, 368
declare.h, 703
CoIf
compcomm.c, 410
declare.h, 703
CoIfMatch
comexpr.c, 369
declare.h, 703
CoIfNoMatch
comexpr.c, 369
declare.h, 703
CoIndex
declare.h, 703
names.c, 1262
CoInExpression
compcomm.c, 408
declare.h, 696
CoInParallel
compcomm.c, 407
declare.h, 696
CoInside
compcomm.c, 403
declare.h, 695
CoInsideFirst
compcomm.c, 398
declare.h, 704
CoIntoHide
compcomm.c, 401
declare.h, 711
CoKeep
compcomm.c, 399
declare.h, 704
CoLabel
compcomm.c, 403
declare.h, 704
CollectFun
C_const, 42
CollectOverFlag
S_const, 210
CollectPercentage
C_const, 44
CoLoad
declare.h, 704
store.c, 2052
CoLocal
comexpr.c, 368
declare.h, 704
CoLocalFactorized
comexpr.c, 368
declare.h, 704
CoMakeInteger
compcomm.c, 404
declare.h, 707
CoMany
comexpr.c, 369
declare.h, 704

combilast
SHvariables, 215
ComChar
P_const, 179
ComDefer
C_const, 42
CoMerge
compcomm.c, 412
declare.h, 705
CoMetric
compcomm.c, 399
declare.h, 705
comexpr.c, 367, 371
CoAssign, 371
CoDeallocateTable, 371
CoDisorder, 369
CoFill, 370
CoFillExpression, 370
CoGlobal, 368
CoGlobalFactorized, 368
CoId, 368
CoIdExpression, 370
CoIdNew, 369
CoIdOld, 368
CoIfMatch, 369
CoIfNoMatch, 369
CoLocal, 368
CoLocalFactorized, 368
CoMany, 369
CoMulti, 369
CoMultiply, 370
CoOnce, 369
CoOnly, 370
CoPrintTable, 370
CoSelect, 370
DoExpr, 368
comfun
T_const, 249
COMFUNPOWERS
ftypes.h, 1057
comind
T_const, 249
commentchar
setfile.c, 1927
Commercial
C_const, 45
COMMERCIALSIZE
fsizes.h, 966
Commute
declare.h, 602
normal.c, 1305
commute
FUN_INFO, 84
FuNcTiOn, 86
CommuteInSet
C_const, 29
comnum
T_const, 249
CoModOption
declare.h, 705
module.c, 1198
CoModuleOption
declare.h, 705
module.c, 1198
CoModulus
compcomm.c, 408
declare.h, 705
CompactifyTree
declare.h, 657
names.c, 1260
COMPARE
declare.h, 594
Compare1
declare.h, 603
sort.c, 1955
CompareArgs
declare.h, 666
tools.c, 2305
CompareFunctions
declare.h, 602
normal.c, 1305
CompareHSymbols
declare.h, 746
sort.c, 1957
CompareRoutine
R_const, 201
CompareSymbols
declare.h, 746
sort.c, 1956
CompareTerms
declare.h, 593
CompArg
declare.h, 602
tools.c, 2306
CompCoef
declare.h, 602
reken.c, 1797
compcomm.c, 394, 417
Add2ComStrings, 402
AddComString, 402
AddToCom, 402
CoAntiBracket, 409
CoAntiPutInside, 416
CoAntiSymmetrize, 405
CoArgExplode, 413
CoArgImplode, 413
CoArgToExtraSymbol, 414
CoArgument, 403
CoBracket, 409
CoChainin, 407
CoChainout, 407
CoChisholm, 407
CoClearTable, 413
CoCollect, 397
CoCompress, 398
CoContract, 403

CoCycleSymmetrize, 405
CoDelete, 399
CoDenominators, 413
CoDiscard, 402
CoDo, 414
CoDrop, 401
CoDropCoefficient, 413
CoDropSymbols, 413
CoElse, 410
CoElseIf, 410
CoEndArgument, 403
CoEndDo, 415
CoEndIf, 410
CoEndInExpression, 408
CoEndInside, 404
CoEndRepeat, 409
CoEndTerm, 411
CoEndWhile, 410
CoExit, 407
CoExtraSymbols, 414
CoFactArg, 404
CoFactDollar, 415
CoFactorize, 415
CoFindLoop, 411
CoFixIndex, 399
CoFlags, 398
CoFormat, 399
CoFromPolynomial, 414
CoFunPowers, 411
CoGoTo, 403
CoHide, 401
CoIf, 410
CoInExpression, 408
CoInParallel, 407
CoInside, 403
CoInsideFirst, 398
CoIntoHide, 401
CoKeep, 399
CoLabel, 403
CoMakeInteger, 404
CoMerge, 412
CoMetric, 399
CoModulus, 408
CoMultiBracket, 409
CoNFactorize, 415
CoNoDrop, 401
CoNoHide, 401
CoNormalize, 404
CoNoSkip, 401
CoNotInParallel, 408
CoNoUnHide, 402
CoNPrint, 400
CoNUnFactorize, 415
CoNWrite, 405
CoOff, 398
CoOn, 398
CoOptimizeOption, 416
CoPolyFun, 412
CoPolyRatFun, 412
CoPopHide, 400
CoPrint, 399
CoPrintB, 400
CoProcessBucket, 412
CoProperCount, 398
CoPushHide, 400
CoPutInside, 416
CoRatio, 406
CoRCycleSymmetrize, 405
CoRedefine, 406
CoRenumber, 406
CoRepeat, 409
CoReplaceLoop, 411
CoSetExitFlag, 408
CoSkip, 401
CoSort, 411
CoSplitArg, 404
CoSplitFirstArg, 404
CoSplitLastArg, 404
CoStuffle, 412
CoSum, 406
CoSymmetrize, 405
CoTerm, 411
CoThreadBucket, 412
CoToPolynomial, 413
CoToTensor, 406
CoToVector, 406
CoTrace4, 406
CoTraceN, 407
CoTryReplace, 408
CoUnFactorize, 415
CoUnHide, 402
CoUnitTrace, 411
CountComp, 409
CoWhile, 410
CoWrite, 405
DoArgPlode, 412
DoArgument, 403
DoBrackets, 409
DoChain, 407
DoFactorize, 415
DoFindLoop, 410
DoInParallel, 408
DoPrint, 399
DoPutInside, 416
DoSymmetrize, 405
GetDoParam, 414
GetIfDollarFactor, 414
SetExpr, 400
SetExprCases, 400
setonoff, 397
CompGroup
declare.h, 603
function.c, 1085
CompileAlgebra
compiler.c, 500
declare.h, 717

CompileLevel
C_const, 33
compiler.c, 497, 502
alfatable1, 502
CodeFactors, 501
CodeGenerator, 501
CompileAlgebra, 500
CompileStatement, 500
CompileSubExpressions, 501
CompleteTerm, 501
findcommand, 499
GenerateFactors, 501
inictable, 499
insubexpbuffers, 502
IsIdStatement, 500
IsRHS, 500
OPTION0, 499
OPTION1, 499
OPTION2, 499
ParenthesesTest, 499
REDUCESUBEXPBUFFERS, 499
SkipAName, 500
subexpbuffers, 502
TestTables, 500
topsubexpbuffers, 502
COMPILERBUFFER
fsizes.h, 967
CompileStatement
compiler.c, 500
declare.h, 682
CompileSubExpressions
compiler.c, 501
declare.h, 721
compiletype
C_const, 31
COMPINC
fsizes.h, 970
CompleteTerm
compiler.c, 501
declare.h, 721
complex
FuNcTiOn, 86
SyMbOl, 237
VeCtOr, 272
ComposeTableNames
declare.h, 743
minos.c, 1171
ComPress
declare.h, 684
sort.c, 1957
compress.c, 528
COMPRESSBUFFER
fsizes.h, 967
CompressBuffer
R_const, 200
compression
ExPrEsSiOn, 70
CompressPointer
R_const, 200
CompressSize
InDeXeNtRy, 94
M_const, 110
compressSize
N_const, 144
compressSpace
N_const, 140
ComprTop
R_const, 200
COMPTREE
structs.h, 2121
comsym
T_const, 249
comtool.c, 537, 543
AddLHS, 539
AddNtoC, 541
AddNtoL, 540
AddRHS, 539
clearcbuf, 538
ClearTree, 542
DoubleCbuffer, 539
FindTree, 542
finishcbuf, 538
inicbufs, 538
IniFbuffer, 542
InsTree, 541
numcommute, 542
RedoTree, 542
comtool.h, 550, 551
CoMulti
comexpr.c, 369
declare.h, 705
CoMultiBracket
compcomm.c, 409
declare.h, 697
CoMultiply
comexpr.c, 370
declare.h, 706
CoNFactorize
compcomm.c, 415
declare.h, 701
CoNFunction
declare.h, 706
names.c, 1264
CONJUGATION
ftypes.h, 1016
CoNoDrop
compcomm.c, 401
declare.h, 706
CoNoHide
compcomm.c, 401
declare.h, 711
CoNormalize
compcomm.c, 404
declare.h, 707
CoNoSkip
compcomm.c, 401

declare.h, 706
CoNotInParallel
compcomm.c, 408
declare.h, 696
CoNoUnHide
compcomm.c, 402
declare.h, 711
CoNPrint
compcomm.c, 400
declare.h, 706
CoNTable
declare.h, 713
names.c, 1265
CoNTensor
declare.h, 706
names.c, 1264
ContentMerge
declare.h, 734
execute.c, 864
contents
DoLoOp, 63
CONTENTTERM
ftypes.h, 1021
CONTRACT
ftypes.h, 1033
CoNUnFactorize
compcomm.c, 415
declare.h, 701
ConvertArgument
declare.h, 765
ratio.c, 1744
convertblock
declare.h, 739
minos.c, 1169
ConvertFromPoly
declare.h, 758
notation.c, 1369
convertiniinfo
declare.h, 740
minos.c, 1170
convertnamesblock
declare.h, 740
minos.c, 1169
ConvertToPoly
declare.h, 758
notation.c, 1369
ConWord
declare.h, 667
tools.c, 2299
CoNWrite
compcomm.c, 405
declare.h, 706
CoOff
compcomm.c, 398
declare.h, 707
CoOn
compcomm.c, 398
declare.h, 707
CoOnce
comexpr.c, 369
declare.h, 707
CoOnly
comexpr.c, 370
declare.h, 707
CoOptimizeOption
compcomm.c, 416
declare.h, 708
CoPolyFun
compcomm.c, 412
declare.h, 708
CoPolyRatFun
compcomm.c, 412
declare.h, 708
CoPopHide
compcomm.c, 400
declare.h, 710
CoPrint
compcomm.c, 399
declare.h, 708
CoPrintB
compcomm.c, 400
declare.h, 708
CoPrintTable
comexpr.c, 370
declare.h, 726
CoProcessBucket
compcomm.c, 412
declare.h, 710
CoProperCount
compcomm.c, 398
declare.h, 708
CoPushHide
compcomm.c, 400
declare.h, 710
CoPutInside
compcomm.c, 416
declare.h, 697
COPY1ARG
declare.h, 582
COPYARG
declare.h, 580
CopyArg
declare.h, 580
CopyExpression
declare.h, 744
store.c, 2056
CopyFile
declare.h, 661
tools.c, 2295
COPYFUN
declare.h, 580
COPYFUN3
declare.h, 581
COPYLONG
reken.c, 1789
COPYSUB

declare.h, 581
CopyTree
declare.h, 657
names.c, 1261
CoRatio
compcomm.c, 406
declare.h, 709
CoRCycleSymmetrize
compcomm.c, 405
declare.h, 709
CoRedefine
compcomm.c, 406
declare.h, 709
CoRemoveSpectator
declare.h, 769
spectator.c, 2013
CoRenumber
compcomm.c, 406
declare.h, 709
CoRepeat
compcomm.c, 409
declare.h, 709
CoReplaceLoop
compcomm.c, 411
declare.h, 724
CoSave
declare.h, 709
store.c, 2052
CoSelect
comexpr.c, 370
declare.h, 709
CoSet
declare.h, 710
names.c, 1265
CoSetExitFlag
compcomm.c, 408
declare.h, 710
COSFUNCTION
ftypes.h, 1018
COSHFUNCTION
ftypes.h, 1019
CoSkip
compcomm.c, 401
declare.h, 710
CoSort
compcomm.c, 411
declare.h, 711
CoSplitArg
compcomm.c, 404
declare.h, 711
CoSplitFirstArg
compcomm.c, 404
declare.h, 711
CoSplitLastArg
compcomm.c, 404
declare.h, 712
COST, 51
add, 51
div, 52
mul, 51
pow, 52
CoStuffle
compcomm.c, 412
declare.h, 705
CoSum
compcomm.c, 406
declare.h, 712
CoSymbol
declare.h, 712
names.c, 1262
CoSymmetrize
compcomm.c, 405
declare.h, 712
CoTable
declare.h, 712
names.c, 1264
CoTableBase
declare.h, 698
tables.c, 2180
CoTBaddto
declare.h, 715
tables.c, 2181
CoTBaudit
declare.h, 715
tables.c, 2182
CoTBcleanup
declare.h, 715
tables.c, 2182
CoTBcreate
declare.h, 716
tables.c, 2181
CoTBenter
declare.h, 716
tables.c, 2181
CoTBhelp
declare.h, 716
tables.c, 2183
CoTBload
declare.h, 716
tables.c, 2182
CoTBoff
declare.h, 716
tables.c, 2182
CoTBon
declare.h, 716
tables.c, 2182
CoTBopen
declare.h, 716
tables.c, 2181
CoTBreplace
declare.h, 717
tables.c, 2182
CoTBuse
declare.h, 717
tables.c, 2183
CoTerm

compcomm.c, 411
declare.h, 712
CoTestUse
declare.h, 717
tables.c, 2181
CoThreadBucket
compcomm.c, 412
declare.h, 717
CoToPolynomial
compcomm.c, 413
declare.h, 759
CoToSpectator
declare.h, 769
spectator.c, 2013
CoToTensor
compcomm.c, 406
declare.h, 713
CoToVector
compcomm.c, 406
declare.h, 713
CoTrace4
compcomm.c, 406
declare.h, 713
CoTraceN
compcomm.c, 407
declare.h, 713
CoTransform
declare.h, 714
transform.c, 2353
CoTryReplace
compcomm.c, 408
declare.h, 714
COULDCOMMUTE
ftypes.h, 1068
CoUnFactorize
compcomm.c, 415
declare.h, 701
CoUnHide
compcomm.c, 402
declare.h, 711
CoUnitTrace
compcomm.c, 411
declare.h, 708
CountComp
compcomm.c, 409
declare.h, 694
CountDo
declare.h, 604
reshuf.c, 1846
counter
ExPrEsSiOn, 69
CountFun
declare.h, 604
reshuf.c, 1846
COUNTFUNCTION
ftypes.h, 1016
countfunnum
M_const, 125
CountTerms1
declare.h, 739
execute.c, 865
CoVector
declare.h, 715
names.c, 1263
CoWhile
compcomm.c, 410
declare.h, 715
CoWrite
compcomm.c, 405
declare.h, 715
cprocedureExtension
P_const, 176
CRANGE
ftypes.h, 1000
Crash
tools.c, 2307
CreateExpression
declare.h, 764
ratio.c, 1742
CreateFile
declare.h, 661
tools.c, 2295
CreateHandle
tools.c, 2295
CreateLogFile
declare.h, 661
tools.c, 2295
CreateStream
declare.h, 689
tools.c, 2294
CSET
ftypes.h, 999
CSUBEXP
ftypes.h, 999
CSYMBOL
ftypes.h, 998
cTable
FixedGlobals, 81
CTD
minos.c, 1168
cTerm
N_const, 135
CurBufWrt
O_const, 160
CurDictFunWithArgs
O_const, 163
CurDictInDollars
O_const, 164
CurDictNotInFunctions
O_const, 164
CurDictNumbers
O_const, 163
CurDictNumberWarning
O_const, 163
CurDictSpecials
O_const, 163

CurDictVariables
O_const, 163
curdirp
setfile.c, 1927
CurDum
R_const, 203
CurExpr
R_const, 205
current
TaBlEbAsE, 253
CURRENTBRACKET
execute.c, 862
CurrentDictionary
O_const, 163
currentExternalChannel
X_const, 274
currentPrompt
X_const, 273
CurrentStream
C_const, 23
currentTerm
N_const, 140
cursortdirp
setfile.c, 1927
CVECTOR
ftypes.h, 998
CVECTOR1
ftypes.h, 1000
cycle
DiStRiBuTe, 59
PeRmUtE, 185
PeRmUtEp, 186
CYCLE1
declare.h, 579
CYCLEARG
ftypes.h, 1063
CYCLESYMMETRIC
ftypes.h, 1049

DAEMON
pre.c, 1570
daemonize
X_const, 274
date
objects, 167
dbase, 52
fullname, 54
handle, 54
iblocks, 54
info, 53
mode, 53
name, 54
nblocks, 54
tablenamefill, 53
tablenames, 54
tablenamessize, 53
topnumber, 53
DBGOUT
parallel.c, 1420
DBGOUT_NINTERMS
parallel.c, 1421
dbufnum
M_const, 114
DeAllocFileHandle
declare.h, 731
setfile.c, 1926
DEBUG
wildcard.c, 2427
DebugFlag
P_const, 179
debugFlags
C_const, 45
debugflags
OPTIMIZE, 171
DECLARATION
ftypes.h, 996
declare.h, 552, 771
ABS, 574
AccumGCD, 597
Add2Com, 587
Add2ComStrings, 690
ADD2POS, 585
Add3Com, 588
Add4Com, 588
Add5Com, 588
AddArgs, 597
AddArrayIndex, 596
AddCoef, 598
AddComString, 717
AddDictionary, 767
AddDollar, 658
AddDubious, 658
AddExpression, 657
AddFunction, 659
AddIndex, 659
AddLHS, 686
AddLineFeed, 575
AddLong, 598
AddName, 655
AddNtoC, 688
AddNtoL, 687
AddObject, 742
AddPLon, 598
AddPoly, 599
ADDPOS, 584
AddPotModdollar, 728
AddRat, 600
AddRHS, 687
AddSet, 660
AddSymbol, 657
AddTableName, 743
AddToCB, 579
AddToCom, 690
AddToDictionary, 766
AddToDollarBuffer, 723
AddToIndex, 741
AddToLine, 600

AddToPreTypes, 686
AddToScratch, 770
AddToString, 679
AddVector, 658
AddWild, 600
AddWildcardName, 685
AdjustRenumScratch, 618
AllocFileHandle, 731
AllocSort, 668
AllocSortFileName, 668
Apply, 610
ApplyExec, 610
ApplyReset, 610
AreArgsEqual, 666
ArgDotproductMerge, 693
ArgFactorize, 690
ArgSymbolMerge, 693
ArgumentExplode, 738
ArgumentImplode, 738
AssignDollar, 722
BACKINOUT, 579
BASEPOSITION, 586
Bernoulli, 615
BigLong, 600
BinomGen, 600
BracketNormalize, 628
CacheNumberFree, 590
CacheNumberMalloc, 589
CacheNumberMallocAddMemory, 753
CatchDollar, 722
ChainIn, 738
ChainOut, 738
CharOut, 669
CheckTableDeclarations, 609
CheckWild, 601
Chisholm, 601
CleanDollarFactors, 726
CleanExpr, 601
CleanUp, 601
CleanupArgCache, 693
CleanUpSort, 731
CleanupTerm, 734
ClearBracketIndex, 736
clearcbuf, 730
ClearMacro, 671
ClearSpectators, 770
ClearTableTree, 729
ClearTree, 722
ClearWild, 601
CloseChannel, 664
closeExternalChannel, 745
CloseFile, 661
CloseStream, 654
CoAntiBracket, 695
CoAntiPutInside, 697
CoAntiSymmetrize, 695
CoApply, 698
CoArgExplode, 695
CoArgImplode, 695
CoArgToExtraSymbol, 760
CoArgument, 695
CoAssign, 720
CoAuto, 715
CoBracket, 696
CoCFunction, 697
CoChainin, 714
CoChainout, 714
CoChisholm, 714
CoClearTable, 714
CoCollect, 697
CoCommuteInSet, 659
CoCompress, 697
CoContract, 698
CoCopySpectator, 769
CoCreateSpectator, 769
CoCTable, 713
CoCTensor, 697
CoCycleSymmetrize, 698
CoDeallocateTable, 726
CodeFactors, 721
CodeGenerator, 721
CoDelete, 698
CoDenominators, 698
CodeToLine, 595
CoDimension, 698
CoDiscard, 699
CoDisorder, 699
CoDo, 760
CoDrop, 699
CoDropCoefficient, 699
CoDropSymbols, 699
CoElse, 699
CoElseIf, 699
CoEmptySpectator, 769
CoEndArgument, 700
CoEndDo, 760
CoEndIf, 700
CoEndInExpression, 696
CoEndInside, 700
CoEndRepeat, 700
CoEndTerm, 700
CoEndWhile, 700
CoExit, 700
CoExtraSymbols, 760
CoFactArg, 701
CoFactDollar, 701
CoFactorize, 701
CoFill, 702
CoFillExpression, 702
CoFindLoop, 724
CoFixIndex, 702
CoFlags, 707
CoFormat, 702
CoFromPolynomial, 759
CoFunction, 659
CoFunPowers, 724

CoGlobal, 702
CoGlobalFactorized, 702
CoGoTo, 702
CoHide, 710
CoId, 703
CoIdExpression, 720
CoIdNew, 703
CoIdOld, 703
CoIf, 703
CoIfMatch, 703
CoIfNoMatch, 703
CoIndex, 703
CoInExpression, 696
CoInParallel, 696
CoInside, 695
CoInsideFirst, 704
CoIntoHide, 711
CoKeep, 704
CoLabel, 704
CoLoad, 704
CoLocal, 704
CoLocalFactorized, 704
CoMakeInteger, 707
CoMany, 704
CoMerge, 705
CoMetric, 705
Commute, 602
CoModOption, 705
CoModuleOption, 705
CoModulus, 705
CompactifyTree, 657
COMPARE, 594
Compare1, 603
CompareArgs, 666
CompareFunctions, 602
CompareHSymbols, 746
CompareSymbols, 746
CompareTerms, 593
CompArg, 602
CompCoef, 602
CompGroup, 603
CompileAlgebra, 717
CompileStatement, 682
CompileSubExpressions, 721
CompleteTerm, 721
ComposeTableNames, 743
ComPress, 684
CoMulti, 705
CoMultiBracket, 697
CoMultiply, 706
CoNFactorize, 701
CoNFunction, 706
CoNoDrop, 706
CoNoHide, 711
CoNormalize, 707
CoNoSkip, 706
CoNotInParallel, 696
CoNoUnHide, 711
CoNPrint, 706
CoNTable, 713
CoNTensor, 706
ContentMerge, 734
CoNUnFactorize, 701
ConvertArgument, 765
convertblock, 739
ConvertFromPoly, 758
convertiniinfo, 740
convertnamesblock, 740
ConvertToPoly, 758
ConWord, 667
CoNWrite, 706
CoOff, 707
CoOn, 707
CoOnce, 707
CoOnly, 707
CoOptimizeOption, 708
CoPolyFun, 708
CoPolyRatFun, 708
CoPopHide, 710
CoPrint, 708
CoPrintB, 708
CoPrintTable, 726
CoProcessBucket, 710
CoProperCount, 708
CoPushHide, 710
CoPutInside, 697
COPY1ARG, 582
COPYARG, 580
CopyArg, 580
CopyExpression, 744
CopyFile, 661
COPYFUN, 580
COPYFUN3, 581
COPYSUB, 581
CopyTree, 657
CoRatio, 709
CoRCycleSymmetrize, 709
CoRedefine, 709
CoRemoveSpectator, 769
CoRenumber, 709
CoRepeat, 709
CoReplaceLoop, 724
CoSave, 709
CoSelect, 709
CoSet, 710
CoSetExitFlag, 710
CoSkip, 710
CoSort, 711
CoSplitArg, 711
CoSplitFirstArg, 711
CoSplitLastArg, 712
CoStuffle, 705
CoSum, 712
CoSymbol, 712
CoSymmetrize, 712
CoTable, 712

CoTableBase, 698
CoTBaddto, 715
CoTBaudit, 715
CoTBcleanup, 715
CoTBcreate, 716
CoTBenter, 716
CoTBhelp, 716
CoTBload, 716
CoTBoff, 716
CoTBon, 716
CoTBopen, 716
CoTBreplace, 717
CoTBuse, 717
CoTerm, 712
CoTestUse, 717
CoThreadBucket, 717
CoToPolynomial, 759
CoToSpectator, 769
CoToTensor, 713
CoToVector, 713
CoTrace4, 713
CoTraceN, 713
CoTransform, 714
CoTryReplace, 714
CoUnFactorize, 701
CoUnHide, 711
CoUnitTrace, 708
CountComp, 694
CountDo, 604
CountFun, 604
CountTerms1, 739
CoVector, 715
CoWhile, 715
CoWrite, 715
CreateExpression, 764
CreateFile, 661
CreateLogFile, 661
CreateStream, 689
CYCLE1, 579
DeAllocFileHandle, 731
Deferred, 605
defineChannel, 745
DeleteObject, 742
DeleteStore, 605
DenToFunction, 739
DetCommu, 602
DetCurDum, 606
DetVars, 606
DictFromBytes, 769
DictToBytes, 769
DIFBASE, 585
DIFPOS, 585
DimensionExpression, 604
DimensionSubterm, 604
DimensionTerm, 604
Distribute, 606
DIVfunction, 764
DivLong, 606
DIVPOS, 585
DivRat, 606
Divvy, 607
DO_UFFLE, 595
DoArgPlode, 695
DoArgument, 690
DoBrackets, 694
DoBreakDo, 677
DoCall, 677
DoChain, 714
DoCheckpoint, 752
DoClearOptimize, 673
DoCommentChar, 678
DoDebug, 677
DoDefine, 670
DoDelta, 607
DoDelta3, 607
DoDimension, 659
DoDistrib, 608
DoDo, 677
DoElements, 660
DoElse, 676
DoElseif, 676
DoEnddo, 677
DoEndif, 676
DoEndInside, 678
DoEndprocedure, 677
DoesCommu, 602
DoExecute, 651
DoExpr, 720
DoExternal, 672
DoFactDollar, 673
DoFactorize, 701
DoFindLoop, 724
DoFromExternal, 672
DoIf, 676
DoIfdef, 676
DoIfStatement, 611
DoInclude, 671
DoInParallel, 696
DoinParallel, 737
DoInside, 677
DollarFactorize, 726
DOLLARNAME, 583
DollarRaiseLow, 658
DolToFunction, 725
DolToIndex, 726
DolToLong, 726
DolToNumber, 725
DolToSymbol, 726
DolToTensor, 725
DolToTerms, 732
DolToVector, 725
DoMessage, 674
DoModDollar, 737
DoModLocal, 737
DoModMax, 737
DoModMin, 737

DoModSum, 737
DoNoParallel, 736
DonotinParallel, 738
DoOnePow, 611
DoOptimize, 673
DoParallel, 736
DoPartitions, 607
DoPermutations, 608
DoPipe, 676
DoPolyfun, 682
DoPolyratfun, 682
DoPrcExtension, 679
DoPreAdd, 765
DoPreAddSeparator, 744
DoPreAppend, 674
DoPreAppendPath, 771
DoPreAssign, 674
DoPreBreak, 674
DoPreCase, 675
DoPreClose, 678
DoPreCloseDictionary, 766
DoPreCreate, 674
DoPreDefault, 675
DoPreEndSwitch, 675
DoPreExchange, 675
DoPreOpenDictionary, 766
DoPreOut, 674
DoPrePrependPath, 771
DoPrePrintTimes, 678
DoPreRemove, 678
DoPreReset, 679
DoPreRmSeparator, 745
DoPreShow, 675
DoPreSwitch, 675
DoPreUseDictionary, 766
DoPreWrite, 678
DoPrint, 689
DoProcedure, 678
DoProcessBucket, 737
DoPrompt, 672
DoPutInside, 694
DoRecovery, 751
DoRedefine, 670
DoReverseInclude, 672
DoRevert, 612
DoRmExternal, 673
DoSetExternal, 672
DoSetExternalAttr, 673
DoSetRandom, 673
DoShtuffle, 593
DoShuffle, 608
DoSkipExtraSymbols, 674
DoStuffle, 608
DoSumF1, 612
DoSumF2, 612
DoSymmetrize, 690
DoSystem, 675
DoTable, 712
DoTableExpansion, 607
DoTail, 652
DoTempSet, 660
DoTerminate, 676
DoTheta, 612
DoToExternal, 672
DoubleBuffer, 665
DoubleCbuffer, 686
DoubleList, 665
DoubleLList, 665
DoUndefine, 671
DropCoefficient, 628
DropSymbols, 628
DUMMYUSE, 584
DumpNode, 656
DumpTree, 656
EmptyTable, 713
EndOfToken, 681
EndSort, 613
EntVar, 614
EpfCon, 614
EpfFind, 614
EpfGen, 615
EqualArg, 615
Error0, 653
Error1, 653
Error2, 653
EvalDoLoopArg, 732
EvalPreIf, 681
ExchangeDollars, 733
ExchangeExpressions, 733
EXCHINOUT, 579
execarg, 650
ExecInside, 696
ExecModule, 681
execterm, 650
ExistsObject, 742
ExpandBuffer, 665
ExpandRat, 765
ExpandTripleDots, 684
EXPRNAME, 583
ExprStatus, 753
EXTERNLOCK, 591
EXTERNRWLOCK, 592
ExtraSymbol, 628
ExtraSymFun, 761
Factorial, 615
FactorIn, 615
FactorInExpr, 616
FILLARG, 580
FILLEXPR, 581
FILLFUN, 580
FILLFUN3, 581
FILLSUB, 581
FindAll, 616
FindArg, 692
FindBracket, 736
findcommand, 664

FindDictionary, 767
FindExtraSymbol, 768
FindFunction, 768
FindFunWithArgs, 768
FindIndex, 768
FindInIndex, 596
FindInKeyWord, 670
FindKeyWord, 670
FindLocalSubterm, 759
FindLus, 724
FindMulti, 616
FindOnce, 616
FindOnly, 616
FindRange, 756
FindRest, 616
FindrNumber, 617
FindSubexpression, 759
FindSubterm, 758
FindSymbol, 768
FindTableNumber, 743
FindTableTree, 730
FindTB, 609
FindTree, 722
FindVar, 610
FindVector, 768
finishcbuf, 730
FinishShuffle, 608
FinishStuffle, 609
FINISHUFFLE, 595
FiniShuffle, 593
FiniTerm, 617
FlipTable, 738
FLUSHCONSOLE, 578
FlushFile, 663
FlushOut, 617
FlushSpectators, 770
FORM_INLINE, 589
FreeNameTree, 657
FreeTableBase, 743
From0List, 664
FromList, 664
FromPolyRatFun, 762
FromVarList, 665
FullRenumber, 723
FullSymmetrize, 641
FunLevel, 618
FunMatchCy, 624
FunMatchSy, 624
GCDclean, 762
GCDfunction, 761
GCDfunction3, 761
GcdLong, 618
GCDterms, 762
GenerateFactors, 721
Generator, 619
GetAppendChannel, 663
GetAutoName, 656
GETBIDENTITY, 593
GetBinom, 620
GETCFROMEXTCHANNEL, 594
GetChannel, 663
GetChar, 669
GETCOEF, 574
GetContent, 734
GetDbase, 741
GetDollar, 682
GetDollarNumber, 673
GetDolNum, 728
GetDoParam, 760
GetFirstBracket, 734
GetFirstTerm, 734
GetFromSpectator, 770
GetFromStore, 620
GetFromStream, 653
GETIDENTITY, 593
GetIfDollarFactor, 760
GetIfDollarNum, 610
GetLabel, 720
GetLastExprName, 655
GetLong, 620
GetLongModInverses, 635
GetModInverses, 634
GetMoreFromMem, 621
GetMoreTerms, 620
GetName, 655
GetNode, 655
GetOName, 656
GetOneTerm, 621
GetPosFile, 663
GetPreVar, 667
GetSetupPar, 668
GETSTOP, 574
GetStreamPosition, 686
GetTable, 621
GetTableName, 743
GETTERM, 594
GetTerm, 621
GetVar, 656
GetWildcardName, 685
Globalize, 685
Glue, 621
HighWarning, 651
HowMany, 739
iexp, 665
INCLENG, 574
Include, 672
InFunction, 621
inicbufs, 664
IniFbuffer, 761
IniLine, 622
INILOCK, 591
IniModule, 669
INIRWLOCK, 592
IniSpecialModule, 669
iniTools, 753
initPresetExternalChannels, 745

iniwranf, 747
InsertArg, 692
InsertTerm, 622
InsideDollar, 731
InsTableTree, 730
InsTree, 721
InvPoly, 765
iranf, 747
ISEQUALPOS, 586
ISEQUALPOSINC, 584
IsExponentSign, 767
ISGEPOS, 587
ISGEPOSINC, 584
IsIdStatement, 718
ISLESSPOS, 586
IsLikeVector, 666
ISMINPOS, 586
IsMultipleOf, 735
IsMultiplySign, 767
ISNEGPOS, 587
ISNOTEQUALPOS, 586
ISNOTZEROPOS, 587
ISPOSPOS, 587
IsRHS, 718
IsSetMember, 735
ISZEROPOS, 587
LcmLong, 618
LinkTree, 657
LoadInputFile, 668
LoadInstruction, 669
LoadStatement, 670
LocalConvertToPoly, 758
LocateBase, 638
LocateFile, 654
LOCK, 591
LongCopy, 683
LongLongCopy, 683
LongToLine, 623
LookInStream, 654
Lus, 724
M_alloc, 593
M_free, 685
MakeDirty, 623
MakeDollarInteger, 727
MakeDollarMod, 727
MakeDubious, 656
MakeInteger, 691
MakeLongRational, 729
MakeMod, 691
MakeRational, 729
Malloc, 652
Malloc1, 652
MarkDirty, 623
MarkPolyRatFunDirty, 590
MatchArgument, 625
MatchCy, 624
MatchE, 624
MatchFunction, 625
MatchIsPossible, 725
MaX, 573
MEMORYMACROS, 589
MergeDotproductLists, 764
MergePatches, 625
MergeSymbolLists, 764
MesCall, 682
MesCerr, 626
MesComp, 626
MesNesting, 590
MessPreNesting, 686
MiN, 573
MLOCK, 592
ModuleInstruction, 669
Modulus, 626
MoveDummies, 626
MULfunc, 764
MulLong, 627
Mully, 627
MULPOS, 585
MulRat, 627
MultDo, 627
MultiplyWithTerm, 763
MUNLOCK, 593
NameConflict, 660
NCOPY, 576
NCOPYB, 577
NCOPYI, 576
NCOPYI32, 577
NeedNumber, 577
NestingChecksum, 590
NewDbase, 742
NewSort, 627
NEXTARG, 581
NextFileIndex, 596
NextPrime, 747
Normalize, 628
NormalModulus, 634
NormPolyTerm, 758
NOTSTARTPOS, 586
NumberFree, 590
NumberMalloc, 589
NumberMallocAddMemory, 753
numcommute, 723
NumCopy, 682
NumToStr, 680
OpenAddFile, 661
OpenBracketIndex, 736
OpenDbase, 741
openExternalChannel, 745
OpenFile, 660
OpenStream, 654
Pack, 629
ParenthesesTest, 718
ParseNumber, 576
ParseSign, 576
ParseSignedNumber, 576
PasteFile, 629

PasteTerm, 596
Permute, 630
PermuteP, 630
PolyDiv, 762
PolyFunClean, 624
PolyFunDirty, 624
PolyFunMul, 630
PolyRatFunSpecial, 731
POPPREASSIGNLEVEL, 591
PopPreVars, 669
PositionStream, 654
pParseObject, 680
PreCmp, 680
PreEq, 680
PreEval, 679
PreIfDollarEval, 734
PreIfEval, 680
PreLoad, 681
PrepPoly, 631
PreRandom, 747
PreSkip, 681
PREV, 583
PrintExtraSymbol, 759
PrintSeq, 684
PrintSubTerm, 683
PrintSubtermList, 759
PrintTerm, 683
PrintTermC, 683
PrintWords, 684
ProcessOption, 655
Product, 631
PrtLong, 631
PruneExtraSymbols, 761
PUSHPREASSIGNLEVEL, 591
PutArgInScratch, 755
PutBracket, 632
PutBracketInIndex, 736
PutExtraSymbols, 763
PutIn, 632
PutInside, 629
PutInSpectator, 770
PutInStore, 632
PutInVflags, 642
PutOut, 632
PutPreVar, 652
PutTableNames, 743
PUTZERO, 585
Quotient, 633
RaisPow, 633
RaisPowCached, 634
RaisPowMod, 634
RatioFind, 635
RatioGen, 635
RatToLine, 635
ReadFile, 662
ReadFromScratch, 770
ReadFromStream, 653
ReadijObject, 741
ReadIndex, 740
ReadIniInfo, 741
ReadObject, 741
ReadPolyRatFun, 761
ReadPosFile, 662
ReadRange, 756
ReadSaveExpression, 748
ReadSaveIndex, 748
ReadSaveTerm32, 749
ReadSaveVariables, 750
ReadSnum, 636
REDLENG, 574
RedoTableTree, 730
RedoTree, 722
Remain10, 636
Remain4, 636
RemoveDictionary, 766
ReNumber, 635
ReOpenFile, 661
ReplaceDollar, 658
ReserveTempFiles, 683
ResetVariables, 685
ResolveSet, 636
ReverseStatements, 654
ReWorkT, 610
RunAddArg, 756
RunCycle, 756
RunDecode, 755
RunDedup, 757
RunDropArg, 757
RunEncode, 754
RunExplode, 755
RunImplode, 755
RunIsLyndon, 757
RunMulArg, 757
RunPermute, 756
RunReplace, 755
RunReverse, 756
RunSelectArg, 757
RunToLyndon, 757
RunTransform, 754
RWLOCKR, 592
RWLOCKW, 592
ScanFunctions, 636
SeekFile, 662
SeekScratch, 636
selectExternalChannel, 745
set_del, 744
set_in, 744
set_set, 744
set_sub, 744
SETBASELENGTH, 584
SETBASEPOSITION, 584
SetDictionaryOptions, 766
SetEndHScratch, 637
SetEndScratch, 637
SETERROR, 583
SetExpr, 689

SetExprCases, 732
SETKILLMODEFOREXTERNALCHANNEL, 594
SetMods, 651
setonoff, 689
SetPosFile, 663
SetScratch, 651
SetSpecialMode, 681
SETSTARTPOS, 586
SETTERMINATORFOREXTERNALCHANNEL, 594
Sflush, 637
SGN, 574
ShrinkDictionary, 768
Shuffle, 608
simp1token, 718
simp2token, 719
simp3atoken, 719
simp3btoken, 719
simp4token, 719
simp5token, 719
simp6token, 720
Simplify, 637
simpwtoken, 719
SkipAName, 720
SKIPBLANKS, 577
SKIPBRA1, 578
SKIPBRA2, 578
SKIPBRA3, 578
SKIPBRA4, 578
SKIPBRA5, 579
SkipField, 666
SortTheList, 725
SortWeights, 694
SortWild, 637
SpareTable, 651
SpecialCleanup, 650
SplitMerge, 638
StageSort, 685
StartVariables, 595
StoreTerm, 639
str_dup, 739
StrCmp, 666
StrCopy, 597
strDup1, 652
StrHICmp, 667
StrICmp, 666
StrICont, 667
StrLen, 667
StudyPattern, 725
StuffAdd, 575
Stuffle, 609
StuffRootAdd, 609
SubPLon, 640
SubsInAll, 733
Substitute, 640
SymbolNormalize, 746
SymFind, 640
SymGen, 640
Symmetrize, 641
SynchFile, 663
TABLESIZE, 582
TakeArgContent, 690
TakeContent, 763
TakeDollarContent, 727
TakeExtraSymbols, 763
TakeIDfunction, 733
TakeLongRoot, 729
TakeModulus, 641
TakeNormalModulus, 641
TakeRatRoot, 729
TakeSymbolContent, 762
TalToLine, 641
TELLFILE, 595
TellFile, 662
TenVec, 642
TenVecFind, 642
TermAssign, 723
TermFree, 590
Terminate, 655
TermMalloc, 589
TermMallocAddMemory, 753
TermRenumber, 642
TermsInBracket, 739
TermsInDollar, 735
TermsInExpression, 735
TestArgNum, 755
TestDoLoop, 765
TestEndDoLoop, 765
TestFunFlag, 746
TestMatch, 643
TestName, 660
TestPartitions, 607
TestSelect, 733
TestSub, 643
TestTerm, 754
TestUse, 609
TheDefine, 670
TheUndefine, 671
TimeChildren, 645
TimeCPU, 645
TimeWallClock, 645
ToFast, 668
ToGeneral, 667
tokenize, 718
TOKENTOLINE, 575
TokenToLine, 646
TOLONG, 587
ToPolyFunGeneral, 668
ToStorage, 646
ToToken, 682
Trace4, 646
Trace4Gen, 646
Trace4no, 646
TraceFind, 647
TraceN, 647
TraceNgen, 647

TraceNno, 647
Traces, 647
TransferBuffer, 733
TransformRational, 767
TranslateExpression, 735
TreatPolyRatFun, 748
Trick, 647
TruncateFile, 663
TryDo, 648
TryRecover, 575
TwoExprCompare, 735
UngetChar, 575
UngetFromStream, 575
UNLOCK, 592
UnPack, 648
UNRWLOCK, 592
UnSetDictionary, 766
UnSetMods, 651
VARNAME, 583
VarStore, 648
WantAddLongs, 588
WantAddPointers, 588
WantAddPositions, 589
Warning, 651
WCOPY, 577
WildDollars, 723
WildFill, 648
WORDDIF, 582
wranf, 747
WriteArgument, 649
WRITEBUFTOEXTCHANNEL, 594
WriteDictionary, 767
WriteDollarFactorToBuffer, 723
WriteDollarToBuffer, 722
WriteExpression, 649
WRITEFILE, 594
WriteFileToFile, 662
WriteIndex, 740
WriteIndexBlock, 740
WriteIniInfo, 741
WriteInnerTerm, 649
WriteNamesBlock, 740
WriteObject, 742
WriteOne, 648
WriteStats, 649
WriteStoreHeader, 751
WriteString, 679
WriteSubTerm, 650
WriteTerm, 650
writeToChannel, 746
WriteTokens, 718
WriteUnfinString, 679
WrtPower, 597
wsizeof, 583
ZEROARG, 580
ZeroFillRange, 582
DeclareVector
vector.h, 2416
DECODEARG
ftypes.h, 1062
DEDUPARG
ftypes.h, 1064
DEFAULTPROCESSBUCKETSIZE
fsizes.h, 970
defaulttempfilename
startup.c, 2027
DEFAULTTHREADBUCKETSIZE
fsizes.h, 971
DEFAULTTHREADLOADBALANCING
fsizes.h, 971
DEFAULTTHREADS
fsizes.h, 971
DeferFlag
R_const, 203
Deferred
declare.h, 605
proces.c, 1674
deferskipped
N_const, 141
defineChannel
declare.h, 745
pre.c, 1583
defined
TaBIEs, 258
DEFINEVALUE
setfile.c, 1924
DEFINITION
ftypes.h, 996
DefPosition
R_const, 200
DelayPrevar
P_const, 178
DeleteObject
declare.h, 742
minos.c, 1173
DeleteRecoveryFile
checkpoint.c, 328
DeleteStore
declare.h, 605
store.c, 2052
DELTA
ftypes.h, 1011
DELTA2
ftypes.h, 1015
DELTA3
ftypes.h, 1013
DELTAP
ftypes.h, 1015
DENOMINATOR
ftypes.h, 1011
DENOMINATORSYMBOL
ftypes.h, 1023
denomnum
M_const, 124
DenToFunction
declare.h, 739

wildcard.c, 2428
description
fixedset, 82
DetCommu
declare.h, 602
normal.c, 1306
DetCurDum
declare.h, 606
reshuf.c, 1845
DetVars
declare.h, 606
store.c, 2053
dict.c, 789, 794
AddDictionary, 792
AddToDictionary, 792
DictFromBytes, 793
DictToBytes, 793
DoPreAdd, 793
DoPreCloseDictionary, 793
DoPreOpenDictionary, 793
DoPreUseDictionary, 793
FindDictionary, 791
FindExtraSymbol, 791
FindFunction, 791
FindFunWithArgs, 791
FindIndex, 791
FindSymbol, 791
FindVector, 791
IsExponentSign, 790
IsMultiplySign, 790
RemoveDictionary, 792
SetDictionaryOptions, 792
ShrinkDictionary, 793
TransformRational, 790
UnSetDictionary, 792
UseDictionary, 792
DICT_ALLNUMBERS
ftypes.h, 1068
DICT_DOFUNWITHARGS
ftypes.h, 1069
DICT_DOSPECIALS
ftypes.h, 1069
DICT_DOVARIABLES
ftypes.h, 1068
DICT_FUNCTION
ftypes.h, 1070
DICT_FUNCTION_WITH_ARGUMENTS
ftypes.h, 1070
DICT_INDEX
ftypes.h, 1070
DICT_INDOLLARS
ftypes.h, 1069
DICT_INTEGERNUMBER
ftypes.h, 1069
DICT_INTEGERONLY
ftypes.h, 1068
DICT_NOFUNWITHARGS
ftypes.h, 1069
DICT_NONUMBERS
ftypes.h, 1068
DICT_NOSPECIALS
ftypes.h, 1069
DICT_NOTINDOLLARS
ftypes.h, 1069
DICT_NOVARIABLES
ftypes.h, 1068
DICT_RANGE
ftypes.h, 1070
DICT_RATIONALNUMBER
ftypes.h, 1069
DICT_RATIONALONLY
ftypes.h, 1068
DICT_SPECIALCHARACTER
ftypes.h, 1070
DICT_SYMBOL
ftypes.h, 1070
DICT_VECTOR
ftypes.h, 1070
DictFromBytes
declare.h, 769
dict.c, 793
Dictionaries
O_const, 161
DICTIONARY, 55
characters, 56
elements, 55
funwith, 56
gnumelements, 56
name, 55
numbers, 56
numelements, 56
ranges, 56
sizeelements, 55
variables, 56
DICTIONARY_ELEMENT, 57
lhs, 57
rhs, 57
size, 57
type, 57
DictToBytes
declare.h, 769
dict.c, 793
DidClean
C_const, 43
DIFBASE
declare.h, 585
DIFPOS
declare.h, 585
dimension
CbUf, 48
fixedset, 82
FuNcTiOn, 88
InDeX, 90
SeTs, 212
SyMbOl, 238
VeCtOr, 272

DimensionExpression
declare.h, 604
reshuf.c, 1846
DimensionSubterm
declare.h, 604
reshuf.c, 1846
DIMENSIONSYMBOL
ftypes.h, 1023
DimensionTerm
declare.h, 604
reshuf.c, 1846
DirtPow
C_const, 41
DIRTYFLAG
ftypes.h, 1036
minos.h, 1193
dirtyflag
FiLeDaTa, 77
DIRTYSYMFLAG
ftypes.h, 1037
DisOrderFlag
N_const, 145
DISTRIBUTE
structs.h, 2124
DiStRiBuTe, 58
cycle, 59
n, 59
n1, 59
n2, 59
obj1, 58
obj2, 58
out, 58
sign, 59
Distribute
declare.h, 606
symmetr.c, 2150
DISTRIBUTION
ftypes.h, 1013
div
COST, 52
DIVFUNCTION
ftypes.h, 1017
DIVfunction
declare.h, 764
ratio.c, 1743
DivLong
declare.h, 606
reken.c, 1792
DIVPOS
declare.h, 585
DivRat
declare.h, 606
reken.c, 1790
divrem
ratio.c, 1744
Divvy
declare.h, 607
reken.c, 1790
DO_UFFLE
declare.h, 595
do_uffle
SHvariables, 215
DOALL
ftypes.h, 1067
DoArgPlode
compcomm.c, 412
declare.h, 695
DoArgument
compcomm.c, 403
declare.h, 690
DoBrackets
compcomm.c, 409
declare.h, 694
DoBreakDo
declare.h, 677
pre.c, 1576
DoCall
declare.h, 677
pre.c, 1576
DoChain
compcomm.c, 407
declare.h, 714
DoCheckpoint
checkpoint.c, 330
declare.h, 752
DoClearOptimize
declare.h, 673
pre.c, 1584
DoCommentChar
declare.h, 678
pre.c, 1574
DoDebug
declare.h, 677
pre.c, 1576
DoDefine
declare.h, 670
pre.c, 1574
DoDelta
declare.h, 607
normal.c, 1306
DoDelta3
declare.h, 607
reshuf.c, 1847
DoDimension
declare.h, 659
names.c, 1262
DoDistrib
declare.h, 608
reshuf.c, 1847
DoDo
declare.h, 677
pre.c, 1576
DODOUBLE
tools.c, 2292
DoElements
declare.h, 660

names.c, 1265
DoElse
declare.h, 676
pre.c, 1576
DoElseif
declare.h, 676
pre.c, 1576
DoEnddo
declare.h, 677
pre.c, 1577
DoEndif
declare.h, 676
pre.c, 1577
DoEndInside
declare.h, 678
pre.c, 1577
DoEndprocedure
declare.h, 677
pre.c, 1577
DoesCommu
declare.h, 602
normal.c, 1306
DOESNOTCOMMUTE
ftypes.h, 1068
DoExecStatement
module.c, 1200
DoExecute
declare.h, 651
execute.c, 863
DOEXPAND
tools.c, 2292
DoExpr
comexpr.c, 368
declare.h, 720
DoExternal
declare.h, 672
pre.c, 1582
DoFactDollar
declare.h, 673
pre.c, 1584
DoFactorize
compcomm.c, 415
declare.h, 701
DoFindLoop
compcomm.c, 410
declare.h, 724
DoFromExternal
declare.h, 672
pre.c, 1583
DoIf
declare.h, 676
pre.c, 1577
DoIfdef
declare.h, 676
pre.c, 1577
DoIfStatement
declare.h, 611
if.c, 1112
DoInclude
declare.h, 671
pre.c, 1575
DoInParallel
compcomm.c, 408
declare.h, 696
DoinParallel
declare.h, 737
module.c, 1200
DoInside
declare.h, 677
pre.c, 1577
DOLARGUMENT
ftypes.h, 1056
DOLINDEX
ftypes.h, 1056
dollar
ARGBUFFER, 8
dollar.c, 807, 817
AddPotModdollar, 816
AddToDollarBuffer, 809
AssignDollar, 809
CatchDollar, 809
CleanDollarFactors, 814
DollarFactorize, 813
DollarRaiseLow, 812
DolToFunction, 810
DolToIndex, 810
DolToLong, 811
DolToNumber, 810
DolToSymbol, 810
DolToTensor, 810
DolToTerms, 810
DolToVector, 810
EvalDoLoopArg, 812
ExchangeDollars, 811
ExecInside, 811
GetDolNum, 816
InsideDollar, 811
IsMultipleOf, 812
IsSetMember, 812
MakeDollarInteger, 814
MakeDollarMod, 815
PreIfDollarEval, 811
STEP2, 808
TakeDollarContent, 814
TermAssign, 809
TermsInDollar, 811
TestDoLoop, 813
TestEndDoLoop, 813
TranslateExpression, 812
TwoExprCompare, 812
WildDollars, 809
WriteDollarFactorToBuffer, 809
WriteDollarToBuffer, 809
dollar_buf, 59
DOLLAREXPR2
ftypes.h, 1010

DOLLAREXPRESSION
ftypes.h, 1010
DollarFactorize
declare.h, 726
dollar.c, 813
DOLLARFLAG
ftypes.h, 1051
DollarInOutBuffer
O_const, 162
DollarList
P_const, 174
DOLLARNAME
declare.h, 583
dollarname
DoLoOp, 63
DOLLARNAMES
ftypes.h, 1061
dollarnames
C_const, 19
DollarOutBuffer
O_const, 159
DollarOutSizeBuffer
O_const, 162
DollarRaiseLow
declare.h, 658
dollar.c, 812
DoLlArS, 60
factors, 60
index, 61
name, 61
nfactors, 61
node, 61
numdummies, 61
size, 60
type, 61
where, 60
zero, 61
Dollars
variable.h, 2408
DOLLARSTREAM
ftypes.h, 988
dollarzero
M_const, 126
DOLNUMBER
ftypes.h, 1056
DOLOOP
structs.h, 2123
DoLoOp, 62
contents, 63
dollarname, 63
errorsinloop, 64
firstdollar, 64
firstloopcall, 64
firstnum, 64
incdollar, 65
incnum, 64
lastdollar, 65
lastnum, 64
name, 63
NoShowInput, 64
NumPreTypes, 65
p, 63
PreIfLevel, 65
PreSwitchLevel, 65
startlinenumber, 63
type, 64
vars, 63
dolooplevel
C_const, 39
doloopnest
C_const, 28
DoLoops
variable.h, 2404
doloopstack
C_const, 28
doloopstacksize
C_const, 39
DOLSUBTERM
ftypes.h, 1056
DOLTERMS
ftypes.h, 1056
DolToFunction
declare.h, 725
dollar.c, 810
DolToIndex
declare.h, 726
dollar.c, 810
DolToLong
declare.h, 726
dollar.c, 811
DolToNumber
declare.h, 725
dollar.c, 810
DolToSymbol
declare.h, 726
dollar.c, 810
DolToTensor
declare.h, 725
dollar.c, 810
DolToTerms
declare.h, 732
dollar.c, 810
DolToVector
declare.h, 725
dollar.c, 810
DOLUNDEFINED
ftypes.h, 1056
DOLWILDARGS
ftypes.h, 1056
DOLZERO
ftypes.h, 1056
DoMessage
declare.h, 674
pre.c, 1578
DoModDollar
declare.h, 737

module.c, 1200
DoModLocal
declare.h, 737
module.c, 1200
DoModMax
declare.h, 737
module.c, 1199
DoModMin
declare.h, 737
module.c, 1199
DoModSum
declare.h, 737
module.c, 1199
DoNoParallel
declare.h, 736
module.c, 1199
DonotinParallel
declare.h, 738
module.c, 1200
DoOnePow
declare.h, 611
proces.c, 1673
DoOptimize
declare.h, 673
pre.c, 1584
DoParallel
declare.h, 736
module.c, 1199
DoPartitions
declare.h, 607
reshuf.c, 1847
DoPermutations
declare.h, 608
reshuf.c, 1848
DoPipe
declare.h, 676
pre.c, 1578
DoPipeStatement
module.c, 1200
DoPolyfun
declare.h, 682
module.c, 1199
DoPolyratfun
declare.h, 682
module.c, 1199
DoPrcExtension
declare.h, 679
pre.c, 1578
DoPreAdd
declare.h, 765
dict.c, 793
DoPreAddSeparator
declare.h, 744
pre.c, 1582
DoPreAppend
declare.h, 674
pre.c, 1578
DoPreAppendPath
declare.h, 771
pre.c, 1585
DoPreAssign
declare.h, 674
pre.c, 1574
DoPreBreak
declare.h, 674
pre.c, 1579
DoPreCase
declare.h, 675
pre.c, 1579
DoPreClose
declare.h, 678
pre.c, 1579
DoPreCloseDictionary
declare.h, 766
dict.c, 793
DoPreCreate
declare.h, 674
pre.c, 1578
DoPreDefault
declare.h, 675
pre.c, 1579
DoPreEndSwitch
declare.h, 675
pre.c, 1580
DoPreExchange
declare.h, 675
pre.c, 1575
DoPreOpenDictionary
declare.h, 766
dict.c, 793
DoPreOut
declare.h, 674
pre.c, 1578
DoPrePrependPath
declare.h, 771
pre.c, 1585
DoPrePrintTimes
declare.h, 678
pre.c, 1578
DoPreRemove
declare.h, 678
pre.c, 1579
DoPreReset
declare.h, 679
pre.c, 1585
DoPreRmSeparator
declare.h, 745
pre.c, 1582
DoPreShow
declare.h, 675
pre.c, 1580
DoPreSwitch
declare.h, 675
pre.c, 1580
DoPreUseDictionary
declare.h, 766

dict.c, 793
DoPreWrite
declare.h, 678
pre.c, 1579
DoPrint
compcomm.c, 399
declare.h, 689
DoProcedure
declare.h, 678
pre.c, 1579
DoProcessBucket
declare.h, 737
module.c, 1200
DoPrompt
declare.h, 672
pre.c, 1583
DoPutInside
compcomm.c, 416
declare.h, 694
DoRecovery
checkpoint.c, 329
declare.h, 751
DoRedefine
declare.h, 670
pre.c, 1574
DoReverseInclude
declare.h, 672
pre.c, 1575
DoRevert
declare.h, 612
normal.c, 1306
DoRmExternal
declare.h, 673
pre.c, 1583
DoSetExternal
declare.h, 672
pre.c, 1583
DoSetExternalAttr
declare.h, 673
pre.c, 1583
DoSetRandom
declare.h, 673
pre.c, 1584
DoSetups
setfile.c, 1925
DoShtuffle
declare.h, 593
DoShuffle
declare.h, 608
reshuf.c, 1848
DoSkipExtraSymbols
declare.h, 674
pre.c, 1585
DoStuffle
declare.h, 608
reshuf.c, 1848
DoSumF1
declare.h, 612
ratio.c, 1739
DoSumF2
declare.h, 612
ratio.c, 1740
DoSymmetrize
compcomm.c, 405
declare.h, 690
DoSystem
declare.h, 675
pre.c, 1580
DoTable
declare.h, 712
names.c, 1264
DoTableExpansion
declare.h, 607
tables.c, 2180
DoTail
declare.h, 652
startup.c, 2025
dotchar
setfile.c, 1927
DoTempSet
declare.h, 660
names.c, 1266
DoTerminate
declare.h, 676
pre.c, 1576
DoTheta
declare.h, 612
normal.c, 1306
DoToExternal
declare.h, 672
pre.c, 1583
DOTPBIT
ftypes.h, 1055
DOTPRODUCT
ftypes.h, 1009
DoubleBuffer
declare.h, 665
tools.c, 2304
DoubleCbuffer
comtool.c, 539
declare.h, 686
DoubleFlag
O_const, 165
DOUBLEFORTRANMODE
ftypes.h, 1007
DoubleIfBuffers
if.c, 1112
DoubleList
declare.h, 665
tools.c, 2304
DoubleLList
declare.h, 665
tools.c, 2304
DOUBLEPRECISIONFLAG
ftypes.h, 1007
DoUndefine

declare.h, 671
pre.c, 1575
DROP
ftypes.h, 1038
DROPARG
ftypes.h, 1064
DropCoefficient
declare.h, 628
normal.c, 1307
DROPGEXPRESSION
ftypes.h, 1039
DROPHGEXPRESSION
ftypes.h, 1040
DROPHLEXPRESSION
ftypes.h, 1040
DROPLEXPRESSION
ftypes.h, 1038
DROPPEDEXPRESSION
ftypes.h, 1039
DROPSPECTATOREXPRESSION
ftypes.h, 1041
DropSymbols
declare.h, 628
normal.c, 1307
DuBiOuS, 65
dummy, 66
name, 66
node, 66
Dubious
variable.h, 2408
DubiousList
C_const, 20
DumFound
N_const, 136
DUMFUN
ftypes.h, 1013
DumFunPlace
N_const, 136
DumInd
M_const, 120
DUMMY
ftypes.h, 1054
dummy
ARGBUFFER, 8
DuBiOuS, 66
DUMMYBUFFER
ftypes.h, 1061
DUMMYFLAG
ftypes.h, 1046
DUMMYFUN
ftypes.h, 1013
DUMMYINDEX_
ftypes.h, 1044
dummyrenumlist
N_const, 137
dummysubexp
T_const, 249
DUMMYTEN
ftypes.h, 1013
DUMMYUSE
declare.h, 584
DumNum
C_const, 41
dumnumflag
C_const, 36
DUMPINTERMS
ftypes.h, 993
DumPlace
N_const, 136
DumpNode
declare.h, 656
names.c, 1260
DUMPOUTTERMS
ftypes.h, 993
DUMPTOCOMPILER
ftypes.h, 993
DUMPTOPARALLEL
ftypes.h, 993
DUMPTOSORT
ftypes.h, 993
DumpTree
declare.h, 656
names.c, 1260

e
bufIPstruct, 14
e3
TrAcEs, 265
e4
TrAcEs, 265
eat
P_const, 178
EATTENSOR
ftypes.h, 1047
ebufnum
T_const, 246
eers
TrAcEs, 265
element
objects, 168
ELEMENTLOADED
ftypes.h, 1059
elements
DICTIONARY, 55
ELEMENTSIZE
minos.h, 1193
ELEMENTUSED
ftypes.h, 1059
empty
FiLeInDeX, 79
EMPTYINDEX
ftypes.h, 1046
EMPTYININDEX
structs.h, 2120
emptystring
startup.c, 2027
EmptyTable

declare.h, 713
names.c, 1265
ENCODEARG
ftypes.h, 1062
END
ftypes.h, 1054
endianness
STOREHEADER, 230
ENDMODULE
ftypes.h, 990
EndNest
N_const, 134
ENDOFINPUT
ftypes.h, 988
ENDOFSTREAM
ftypes.h, 988
EndOfToken
declare.h, 681
tools.c, 2300
endoftokens
C_const, 27
EndSort
declare.h, 613
sort.c, 1950
entriesinindex
iniinfo, 96
EntVar
declare.h, 614
names.c, 1260
EpfCon
declare.h, 614
opera.c, 1385
EpfFind
declare.h, 614
opera.c, 1385
EpfGen
declare.h, 615
opera.c, 1386
eqnum
StreaM, 235
EQUAL
ftypes.h, 1053
EqualArg
declare.h, 615
reshuf.c, 1847
Error0
declare.h, 653
message.c, 1153
Error1
declare.h, 653
message.c, 1153
Error2
declare.h, 653
message.c, 1154
ErrorBlock
O_const, 166
ErrorInDollar
N_const, 142
ERROROUT
ftypes.h, 994
errorsinloop
DoLoOp, 64
Eside
R_const, 205
EvalDoLoopArg
declare.h, 732
dollar.c, 812
EvalPreIf
declare.h, 681
pre.c, 1581
EVEN_
ftypes.h, 1043
EXCH
normal.c, 1305
ExchangeDollars
declare.h, 733
dollar.c, 811
ExchangeExpressions
declare.h, 733
execute.c, 864
EXCHINOUT
declare.h, 579
EXCHN
normal.c, 1305
execarg
argument.c, 276
declare.h, 650
ExecInside
declare.h, 696
dollar.c, 811
ExecMode
S_const, 210
ExecModule
declare.h, 681
module.c, 1198
ExecStore
module.c, 1198
execterm
argument.c, 276
declare.h, 650
execute.c, 861, 866
BRACKETCURRENTEXPR, 862
BRACKETOTHEREXPR, 862
CleanExpr, 862
CleanupTerm, 864
ContentMerge, 864
CountTerms1, 865
CURRENTBRACKET, 862
DoExecute, 863
ExchangeExpressions, 864
GetContent, 864
GetFirstBracket, 864
GetFirstTerm, 864
MakeGlobal, 862
NOBRACKETACTIVE, 862
PopVariables, 862

PutBracket, 863
PutInVflags, 863
SetMods, 863
SpecialCleanup, 863
TermsInBracket, 865
TermsInExpression, 865
TestDrop, 863
UnSetMods, 863
UpdatePositions, 865
EXECUTINGIF
ftypes.h, 992
EXECUTINGPRESWITCH
ftypes.h, 992
ExistsObject
declare.h, 742
minos.c, 1173
exitflag
M_const, 126
ExpandBuffer
declare.h, 665
tools.c, 2304
ExpandRat
declare.h, 765
ratio.c, 1744
ExpandTripleDots
declare.h, 684
pre.c, 1573
expchanged
R_const, 205
ExpectedSign
N_const, 147
expflags
R_const, 205
EXPLODEARG
ftypes.h, 1063
expnum
M_const, 124
EXPONENT
ftypes.h, 1011
exprbufsize
ParallelVars, 182
EXPRESSION
ftypes.h, 1009
ExPrEsSiOn, 67
bracketinfo, 68
compression, 70
counter, 69
hidelevel, 69
inmem, 69
name, 69
namesize, 70
newbracketinfo, 68
node, 70
numdummies, 70
numfactors, 70
onfile, 68
printflag, 69
prototype, 68
renum, 68
renumlists, 69
replace, 70
size, 68
sizeprototype, 71
status, 70
vflags, 69
whichbuffer, 70
expression
FiLeInDeX, 79
ExpressionList
C_const, 20
EXPRESSIONNUMBER
ftypes.h, 1066
EXPRESSIONONLY
ftypes.h, 998
Expressions
variable.h, 2409
exprfillwarning
C_const, 37
EXPRHEAD
ftypes.h, 1037
EXPRNAME
declare.h, 583
EXPRNAMES
ftypes.h, 1061
exprnames
C_const, 19
exprnr
OPTIMIZERESULT, 172
exprnumber
SpecTatoR, 226
EXPRSOUT
ftypes.h, 994
ExprStat
FixedGlobals, 80
ExprStatus
bugtool.c, 321
declare.h, 753
exprtodo
ParallelVars, 182
EXPTOEXP
ftypes.h, 1035
extcmd.c, 896, 897
closeAllExternalChannels, 897
closeExternalChannel, 896
getCurrentExternalChannel, 897
initPresetExternalChannels, 896
openExternalChannel, 896
selectExternalChannel, 897
EXTERNALCHANNELOUT
ftypes.h, 994
EXTERNALCHANNELSTREAM
ftypes.h, 988
EXTERNLOCK
declare.h, 591
EXTERNRWLOCK
declare.h, 592

EXTEUCLIDEAN
ftypes.h, 1021
EXTRAPARAMETER
ftypes.h, 1055
extrasym
C_const, 28
EXTRASYMBOL
ftypes.h, 1065
ExtraSymbol
declare.h, 628
normal.c, 1305
extrasymbols
C_const, 45
EXTRASYMFUN
ftypes.h, 1020
ExtraSymFun
declare.h, 761
notation.c, 1370

f
sOrT, 221
FaCdOlLaR, 71
size, 71
type, 71
value, 72
where, 71
facnum
M_const, 124
FACTARGFLAG
ftypes.h, 1065
factor
factorized_poly, 72
TrAcEn, 263
TrAcEs, 268
factor.c, 916, 917
FactorIn, 917
FactorInExpr, 917
FACTORIAL
ftypes.h, 1014
Factorial
declare.h, 615
reken.c, 1798
factorials
T_const, 243
FACTORIN
ftypes.h, 1018
FactorIn
declare.h, 615
factor.c, 917
FactorInExpr
declare.h, 616
factor.c, 917
factorized_poly, 72
factor, 72
power, 72
FactorMode
O_const, 166
FactorNum
O_const, 166
factors
DoLIArS, 60
FACTORSYMBOL
ftypes.h, 1023
FALSE_EXPR
pre.c, 1570
FASTFUNCTION
ftypes.h, 1043
FATAL_SIG_ERROR
portsignals.h, 1563
FBUFFERSIZE
fsizes.h, 972
fbuffersize
M_const, 119
fbufnum
T_const, 246
ff
sOrT, 221
ffbufnum
C_const, 38
FG
inivar.h, 1135
variable.h, 2412
fh
SpecTatoR, 226
FiLe, 73
active, 75
filesize, 74
handle, 75
inbuffer, 75
name, 74
numblocks, 75
PObuffer, 74
POfill, 74
POfull, 74
POposition, 74
POsize, 75
POstop, 74
file
sOrT, 218
FiLeDaTa, 76
dirtyflag, 77
Fill, 77
Handle, 77
Index, 77
Position, 77
FILEHANDLE
structs.h, 2122
FILEINDEX
structs.h, 2121
FiLeInDeX, 78
empty, 79
expression, 79
next, 79
number, 79
filelist
tools.c, 2308
filelistsize

tools.c, 2308
filenum
N_const, 144
FileOnlyFlag
M_const, 113
fileposition
StreaM, 232
FILES
form3.h, 952
filesize
FiLe, 74
FILESTREAM
ftypes.h, 987
Fill
FiLeDaTa, 77
fill
PF_BUFFER, 187
FILLARG
declare.h, 580
FILLEXPR
declare.h, 581
FILLFUN
declare.h, 580
FILLFUN3
declare.h, 581
fillpoint
TaBlEbAsE, 253
FILLSUB
declare.h, 581
FILLVALUE
tools.c, 2291
FinalStats
C_const, 35
finalstep
TrAcEs, 268
FindAll
declare.h, 616
pattern.c, 1529
FindArg
argument.c, 277
declare.h, 692
FindBracket
declare.h, 736
index.c, 1125
findcommand
compiler.c, 499
declare.h, 664
FindDictionary
declare.h, 767
dict.c, 791
FindExtraSymbol
declare.h, 768
dict.c, 791
FindFunction
declare.h, 768
dict.c, 791
FindFunWithArgs
declare.h, 768
dict.c, 791
FindIndex
declare.h, 768
dict.c, 791
FindInIndex
declare.h, 596
store.c, 2055
FindInKeyWord
declare.h, 670
pre.c, 1573
FindKeyWord
declare.h, 670
pre.c, 1573
FindLocalSubterm
declare.h, 759
notation.c, 1370
FINDLOOP
ftypes.h, 1057
FindLus
declare.h, 724
lus.c, 1140
FindMulti
declare.h, 616
findpat.c, 928
FindOnce
declare.h, 616
findpat.c, 928
FindOnly
declare.h, 616
findpat.c, 928
findpat.c, 928, 929
FindMulti, 928
FindOnce, 928
FindOnly, 928
FindRest, 929
findPattern
N_const, 137
FindRange
declare.h, 756
transform.c, 2357
FindRest
declare.h, 616
findpat.c, 929
FindrNumber
declare.h, 617
store.c, 2055
FindSubexpression
declare.h, 759
notation.c, 1370
FindSubterm
declare.h, 758
notation.c, 1370
FindSymbol
declare.h, 768
dict.c, 791
FindTableNumber
declare.h, 743
minos.c, 1172

FindTableTree
declare.h, 730
tables.c, 2180
FindTB
declare.h, 609
tables.c, 2181
findTerm
N_const, 137
FindTree
comtool.c, 542
declare.h, 722
FindVar
declare.h, 610
if.c, 1112
FindVector
declare.h, 768
dict.c, 791
FiniLine
sch.c, 1887
FINISH
ftypes.h, 1057
finishcbuf
comtool.c, 538
declare.h, 730
FinishShuffle
declare.h, 608
reshuf.c, 1848
FinishStuffle
declare.h, 609
reshuf.c, 1849
finishuf
SHvariables, 215
FINISHUFFLE
declare.h, 595
FiniShuffle
declare.h, 593
FiniTerm
declare.h, 617
proces.c, 1671
first
SeTs, 211
FIRSTBRACKET
ftypes.h, 1017
firstconstindex
C_const, 31
firstctypemessage
C_const, 33
firstdollar
DoLoOp, 64
firstindexblock
iniinfo, 96
firstloopcall
DoLoOp, 64
FIRSTMODULE
ftypes.h, 989
firstnameblock
iniinfo, 97
firstnum
DoLoOp, 64
FIRSTTERM
ftypes.h, 1021
FIRSTUSERFUNCTION
ftypes.h, 1022
FIRSTUSERSYMBOL
ftypes.h, 1024
FIXED_
ftypes.h, 1044
FIXEDGLOBALS
structs.h, 2125
FixedGlobals, 79
cTable, 81
ExprStat, 80
fname, 81
fname2, 81
fname2base, 81
fnamebase, 81
FunNam, 80
OperaFind, 80
Operation, 80
s_one, 81
swmes, 81
VarType, 80
fixedset, 82
description, 82
dimension, 82
name, 82
type, 82
fixedsets
inivar.h, 1135
variable.h, 2412
FixIndices
C_const, 26
flags
FuNcTiOn, 87
InDeX, 91
indexblock, 92
KEYWORD, 100
SETUPPARAMETERS, 213
SpecTatoR, 226
SyMbOl, 238
TaBlEs, 256
VeCtOr, 272
FlipLONG
O_const, 160
FlipPOINTER
O_const, 160
FlipPOS
O_const, 160
FlipTable
declare.h, 738
tables.c, 2180
FlipWORD
O_const, 160
FLOATMODE
ftypes.h, 1047
flush_cache

checkpoint.c, 329
FLUSHCONSOLE
declare.h, 578
FlushFile
declare.h, 663
tools.c, 2297
FlushOut
declare.h, 617
sort.c, 1953
FlushSpectators
declare.h, 770
spectator.c, 2014
fname
FixedGlobals, 81
fname2
FixedGlobals, 81
fname2base
FixedGlobals, 81
fnamebase
FixedGlobals, 81
FoldName
StreaM, 233
ForFindOnly
N_const, 137
form3.h, 945, 955
alignof, 950
AWORDMASK, 949
bzero, 954
FILES, 952
FULLMAX, 949
GetPID, 954
inline, 948
MAJORVERSION, 947
MAX_OPEN_FILES, 954
MAXLONG, 949
MAXPOSITIVE, 949
MAXPOSITIVE2, 950
MAXPOSITIVE4, 950
MINORVERSION, 947
MLONG, 955
NDEBUG, 948
PADDUMMY, 950
PADINT, 951
PADLONG, 951
PADPOINTER, 950
PADPOSITION, 950
PADWORD, 951
PRODUCTIONDATE, 947
RLONG, 955
SBYTE, 954
SPECMASK, 949
STATIC_ASSERT, 948
STATIC_ASSERT__1, 948
STATIC_ASSERT__2, 948
STATIC_ASSERT__3, 948
TOPBITONLY, 948
TOPLONGBITONLY, 949
UBYTE, 955
Uclose, 952
Uflush, 952
Ugetpos, 953
UINT, 955
Uopen, 952
Uread, 952
Useek, 953
Usetbuf, 953
Usetpos, 953
Ustdout, 954
Usync, 953
Utell, 953
Utruncate, 954
Uwrite, 952
VOID, 954
WILDMASK, 949
WITHSORTBOTS, 951
WORDMASK, 949
FORM_INLINE
declare.h, 589
FORMNAME
startup.c, 2024
FortDotChar
O_const, 166
FortFirst
O_const, 165
Fortran90Kind
C_const, 27
FortranCont
M_const, 123
FORTRANCONTINUATIONLINES
fsizes.h, 967
FORTRANMODE
ftypes.h, 1006
FoStage4
R_const, 200
fPatches
sOrT, 220
fPatchesStop
sOrT, 221
fPatchN
sOrT, 224
fPatchN1
sOrT, 223
FreeNameTree
declare.h, 657
names.c, 1261
FreeTableBase
declare.h, 743
minos.c, 1171
from
PF_BUFFER, 188
From0List
declare.h, 664
tools.c, 2303
FROMBRAC
ftypes.h, 1035
FROMDISK

minos.h, 1193
FROMFUNCTION
ftypes.h, 1061
fromindex
T_const, 251
FromList
declare.h, 664
tools.c, 2303
FROMMODULEOPTION
ftypes.h, 1065
FROMPOINTINSTRUCTION
ftypes.h, 1065
FromPolyRatFun
declare.h, 762
ratio.c, 1742
FROMSET
ftypes.h, 1035
FromVarList
declare.h, 665
tools.c, 2303
Frozen
N_const, 134
Fscr
R_const, 199
fsizes.h, 961, 973
COMMERCIALSIZE, 966
COMPILERBUFFER, 967
COMPINC, 970
COMPRESSBUFFER, 967
DEFAULTPROCESSBUCKETSIZE, 970
DEFAULTTHREADBUCKETSIZE, 971
DEFAULTTHREADLOADBALANCING, 971
DEFAULTTHREADS, 971
FBUFFERSIZE, 972
FORTRANCONTINUATIONLINES, 967
GZIPDEFAULT, 970
HALFMAX, 964
INDENTSPACE, 972
INITNAMESIZE, 965
INITNODESIZE, 965
LARGEBUFFER, 968
MAXBRACKETBUFFERSIZE, 970
MAXCOMBUFRHS, 971
MAXDOLLARVARIABLES, 963
MAXDUMMIES, 964
MAXENAME, 964
MAXFILESTREAMSIZE, 964
MAXFLAGS, 966
MAXFLEVELS, 969
MAXFPATCHES, 968
MAXIF, 966
MAXINNAMETREE, 963
MAXLABELS, 966
MAXLEVELS, 967
MAXLHS, 967
MAXLINELENGTH, 972
MAXLOOPS, 966
MAXMATCH, 966
MAXMULTIBRACKETLEVELS, 972
MAXNEST, 965
MAXNUMBERSIZE, 965
MAXNUMSIZE, 970
MAXPARLEVEL, 965
MAXPATCHES, 968
MAXPOWER, 963
MAXPRENAMESIZE, 963
MAXREPEAT, 965
MAXSAVEFUNCTION, 964
MAXSUBEXPRESSIONS, 964
MAXTABLECOMBUF, 971
MAXTER, 964
MAXVARIABLES, 963
MAXWILDC, 967
MULTIINDENTSPACE, 972
NORMSIZE, 965
NPAIR1, 972
NPAIR2, 972
NUMFACS, 966
NUMFIXED, 965
NUMSTORECACHES, 971
NUMTABLEENTRIES, 967
SCRATCHSIZE, 969
SFHSIZE, 970
SHMWINSIZE, 970
SIZEFACS, 966
SIZESTORECACHE, 972
SLARGEBUFFER, 969
SMALLBUFFER, 967
SMALLOVERFLOW, 968
SMAXFPATCHES, 969
SMAXPATCHES, 969
SORTIOSIZE, 968
SPECTATORSIZE, 969
SSMALLBUFFER, 968
SSMALLOVERFLOW, 968
SSORTIOSIZE, 969
STERMSSMALL, 969
TABLEEXTENSION, 970
TERMSSMALL, 968
THREADSCRATCHOUTSIZE, 971
THREADSCRATCHSIZE, 971
WILDOFFSET, 963
WORKBUFFER, 964
ftypes.h, 975, 1072
ABSFUNCTION, 1015
ACOSFUNCTION, 1019
ACOSHFUNCTION, 1020
ADDARG, 1064
ALLARGS, 1061
ALLDIRTY, 1037
ALLFUNPOWERS, 1057
ALLVARIABLES, 997
ALSODOLLARS, 1060
ALSOFUNARGS, 1060
ALSOPOWERS, 1060
ALSOREVERSE, 1008

ANDCOND, 1054
ANTISYMMETRIC, 1049
ANYTYPE, 998
ARGFIELD, 1011
ARGHEAD, 1037
ARGRANGE, 1062
ARGTOARG, 1035
ARGWILD, 1010
ARLTOARL, 1035
ASINFUNCTION, 1019
ASINHFUNCTION, 1020
ATAN2FUNCTION, 1019
ATANFUNCTION, 1019
ATANHFUNCTION, 1020
ATENDOFMODULE, 996
AUTONAMES, 1061
BASECODE, 1064
BERNOULLIFUNCTION, 1016
BHEAD, 987
BHEAD0, 987
BINOMIAL, 1014
BUILTINDOLLARS, 1024
BUILTININDICES, 1024
BUILTINSYMBOLS, 1023
BUILTINVECTORS, 1024
CDELETE, 998
CDELTA, 999
CDOLLAR, 999
CDOTPRODUCT, 999
CDOUBLEDOT, 1000
CDUBIOUS, 999
CEXPRESSION, 999
CFUN, 1071
CFUNCTION, 998
CHISHOLM, 1008
CINDEX, 998
CLEANFLAG, 1036
CLEAR, 1054
CLEARMODULE, 990
CMODE, 1006
CNUMBER, 999
COEFFI, 1052
COEFFICIENTSONLY, 1060
COEFFSYMBOL, 1023
COMFUNPOWERS, 1057
CONJUGATION, 1016
CONTENTTERM, 1021
CONTRACT, 1033
COSFUNCTION, 1018
COSHFUNCTION, 1019
COULDCOMMUTE, 1068
COUNTFUNCTION, 1016
CRANGE, 1000
CSET, 999
CSUBEXP, 999
CSYMBOL, 998
CVECTOR, 998
CVECTOR1, 1000
CYCLEARG, 1063
CYCLESYMMETRIC, 1049
DECLARATION, 996
DECODEARG, 1062
DEDUPARG, 1064
DEFINITION, 996
DELTA, 1011
DELTA2, 1015
DELTA3, 1013
DELTAP, 1015
DENOMINATOR, 1011
DENOMINATORSYMBOL, 1023
DICT_ALLNUMBERS, 1068
DICT_DOFUNWITHARGS, 1069
DICT_DOSPECIALS, 1069
DICT_DOVARIABLES, 1068
DICT_FUNCTION, 1070
DICT_FUNCTION_WITH_ARGUMENTS, 1070
DICT_INDEX, 1070
DICT_INDOLLARS, 1069
DICT_INTEGERNUMBER, 1069
DICT_INTEGERONLY, 1068
DICT_NOFUNWITHARGS, 1069
DICT_NONUMBERS, 1068
DICT_NOSPECIALS, 1069
DICT_NOTINDOLLARS, 1069
DICT_NOVARIABLES, 1068
DICT_RANGE, 1070
DICT_RATIONALNUMBER, 1069
DICT_RATIONALONLY, 1068
DICT_SPECIALCHARACTER, 1070
DICT_SYMBOL, 1070
DICT_VECTOR, 1070
DIMENSIONSYMBOL, 1023
DIRTYFLAG, 1036
DIRTYSYMFLAG, 1037
DISTRIBUTION, 1013
DIVFUNCTION, 1017
DOALL, 1067
DOESNOTCOMMUTE, 1068
DOLARGUMENT, 1056
DOLINDEX, 1056
DOLLAREXPR2, 1010
DOLLAREXPRESSION, 1010
DOLLARFLAG, 1051
DOLLARNAMES, 1061
DOLLARSTREAM, 988
DOLNUMBER, 1056
DOLSUBTERM, 1056
DOLTERMS, 1056
DOLUNDEFINED, 1056
DOLWILDARGS, 1056
DOLZERO, 1056
DOTPBIT, 1055
DOTPRODUCT, 1009
DOUBLEFORTRANMODE, 1007
DOUBLEPRECISIONFLAG, 1007
DROP, 1038

DROPARG, 1064
DROPGEXPRESSION, 1039
DROPHGEXPRESSION, 1040
DROPHLEXPRESSION, 1040
DROPLEXPRESSION, 1038
DROPPEDEXPRESSION, 1039
DROPSPECTATOREXPRESSION, 1041
DUMFUN, 1013
DUMMY, 1054
DUMMYBUFFER, 1061
DUMMYFLAG, 1046
DUMMYFUN, 1013
DUMMYINDEX_, 1044
DUMMYTEN, 1013
DUMPINTERMS, 993
DUMPOUTTERMS, 993
DUMPTOCOMPILER, 993
DUMPTOPARALLEL, 993
DUMPTOSORT, 993
EATTENSOR, 1047
ELEMENTLOADED, 1059
ELEMENTUSED, 1059
EMPTYINDEX, 1046
ENCODEARG, 1062
END, 1054
ENDMODULE, 990
ENDOFINPUT, 988
ENDOFSTREAM, 988
EQUAL, 1053
ERROROUT, 994
EVEN_, 1043
EXECUTINGIF, 992
EXECUTINGPRESWITCH, 992
EXPLODEARG, 1063
EXPONENT, 1011
EXPRESSION, 1009
EXPRESSIONNUMBER, 1066
EXPRESSIONONLY, 998
EXPRHEAD, 1037
EXPRNAMES, 1061
EXPRSOUT, 994
EXPTOEXP, 1035
EXTERNALCHANNELOUT, 994
EXTERNALCHANNELSTREAM, 988
EXTEUCLIDEAN, 1021
EXTRAPARAMETER, 1055
EXTRASYMBOL, 1065
EXTRASYMFUN, 1020
FACTARGFLAG, 1065
FACTORIAL, 1014
FACTORIN, 1018
FACTORSYMBOL, 1023
FASTFUNCTION, 1043
FILESTREAM, 987
FINDLOOP, 1057
FINISH, 1057
FIRSTBRACKET, 1017
FIRSTMODULE, 989
FIRSTTERM, 1021
FIRSTUSERFUNCTION, 1022
FIRSTUSERSYMBOL, 1024
FIXED_, 1044
FLOATMODE, 1047
FORTRANMODE, 1006
FROMBRAC, 1035
FROMFUNCTION, 1061
FROMMODULEOPTION, 1065
FROMPOINTINSTRUCTION, 1065
FROMSET, 1035
FUNBIT, 1055
FUNCTION, 1010
FUNCTIONONLY, 997
FUNCTIONSORT, 1046
FUNHEAD, 1037
FUNNYDOLLAR, 1046
FUNNYVEC, 1045
FUNNYWILD, 1045
FUNTOFUN, 1035
GAMMA, 1012
GAMMA1, 1045
GAMMA5, 1045
GAMMA6, 1045
GAMMA7, 1045
GAMMAFIVE, 1012
GAMMAFUNCTION, 1043
GAMMAI, 1012
GAMMASEVEN, 1012
GAMMASIX, 1012
GCDFUNCTION, 1017
GENCOMMUTE, 1055
GENERALFUNCTION, 1042
GENNONCOMMUTE, 1055
GFIVE, 1051
GIDENT, 1051
GLOBAL, 1054
GLOBALEXPRESSION, 1039
GLOBALMODULE, 989
GLOBALSPECTATORFLAG, 1071
GMINUS, 1051
GPLUS, 1051
GREATER, 1053
GREATEREQUAL, 1053
HAAKJE, 1011
HAAKJE0, 1010
HEADERFILE, 989
HIDDENGEXPRESSION, 1039
HIDDENLEXPRESSION, 1039
HIDE, 1038
HIDEGEXPRESSION, 1040
HIDELEXPRESSION, 1040
IDFUNCTION, 1021
IDHEAD, 1051
IFDOLLAR, 1052
IFDOLLAREXTRA, 1052
IFEXPRESSION, 1052
IFISFACTORIZED, 1053

IFOCCURS, 1053
IMPLODEARG, 1062
INCEXPRESSION, 1039
INDEX, 1009
INDEX_, 1044
INDEXONLY, 997
INDTOIND, 1034
INDTOSUB, 1034
INPUTOUT, 994
INTFUNCTION, 1015
INTOHIDE, 1038
INTOHIDEGEXPRESSION, 1040
INTOHIDELEXPRESSION, 1040
INUSE, 1067
INVERSEFACTORIAL, 1014
INVERSEFUNCTION, 1021
INVERSETABLE, 1060
ISCOMPRESSED, 1047
ISFACTORIZED, 1048
ISFORTRAN90, 1008
ISINRHS, 1047
ISLYNDON, 1063
ISLYNDONR, 1063
ISNOTFORTRAN90, 1008
ISUNMODIFIED, 1047
ISYMBOL, 1022
ISZERO, 1047
KEEPZERO, 1048
LBRACE, 1002
LESS, 1053
LESSEQUAL, 1053
LEVICIVITA, 1013
LHSIDE, 1005
LHSIDEX, 1005
LI2FUNCTION, 1020
LINFUNCTION, 1020
LISTEDLOOP, 995
LNFUNCTION, 1018
LNUMBER, 1011
LOADDOLLAR, 1036
LOCALEXPRESSION, 1038
LONGNUMBER, 1052
LOOKINGFORELSE, 992
LOOKINGFORENDIF, 992
LPARENTHESIS, 1001
LRPARENTHESES, 1004
MAINSORT, 1046
MAKEARGS, 1062
MAKERATIONAL, 1021
MAPLEMODE, 1006
MATCH, 1052
MATCHFUNCTION, 1016
MATHEMATICAMODE, 1006
MAXBUILTINFUNCTION, 1022
MAXFUNCTION, 1015
MAXPOWEROF, 1017
MAXRANGEINDICATOR, 1062
MINFUNCTION, 1014
MINPOWEROF, 1017
MINSPEC, 1046
MINVECTOR, 1010
MIXED, 996
MOD2FUNCTION, 1014
MODFUNCTION, 1014
MODLOCAL, 1059
MODMAX, 1059
MODMIN, 1059
MODNONE, 1059
MODSUM, 1059
MODULEINSTACK, 992
MULFUNCTION, 1022
MULTIPLEOF, 1052
MULTIPLYARG, 1064
MUSTCLEANPRF, 1037
NAMENOTFOUND, 1055
NEG0_, 1043
NEG_, 1043
NEWFACTARG, 1065
NEWLYDEFINEDEXPRESSION, 1042
NEWSTATEMENT, 992
NMIN4SHIFT, 1009
NOAUTO, 997
NODOUBLEMASK, 1007
NOFUNPOWERS, 1057
NOINDEX, 1045
NOINVERSES, 1060
NOLYNDON, 1064
NOPARALLEL_CONVPOLY, 990
NOPARALLEL_DOLLAR, 990
NOPARALLEL_NPROC, 991
NOPARALLEL_RHS, 990
NOPARALLEL_SPECTATOR, 990
NOPARALLEL_TBLDOLLAR, 991
NOPARALLEL_USER, 991
NOQUADMASK, 1007
NORMALFORMAT, 1007
NORMALIZEFLAG, 1051
NOSPACEFORMAT, 1007
NOTEQUAL, 1053
NOTRICK, 1008
NOUNPACK, 1061
NUMARG, 1062
NUMARGSFUN, 1014
NUMERATORSYMBOL, 1023
NUMERICALLOOP, 994
NUMFACTORS, 1021
NUMSPECSETS, 1047
NUMTERMSFUN, 1017
NUMTOIND, 1036
NUMTONUM, 1036
NUMTOSUB, 1036
NUMTOSYM, 1036
O_BACKWARD, 1067
O_CSE, 1066
O_CSEGREEDY, 1066
O_FORWARD, 1067

O_FORWARDANDBACKWARD, 1067
O_FORWARDORBACKWARD, 1067
O_GREEDY, 1066
O_MCTS, 1066
O_NONE, 1066
O_OCCURRENCE, 1066
O_SIMULATED_ANNEALING, 1066
ODD_, 1044
OLDFACTARG, 1065
OLDSTATEMENT, 992
ONEEXPRESSION, 995
ONLYFUNCTIONS, 1067
OPTHEAD, 1067
ORCOND, 1054
PARALLELFLAG, 991
PARTEST, 997
PARTITIONS, 1022
PATTERNFUNCTION, 1016
PERMUTATIONS, 1022
PERMUTEARG, 1063
PFORTRANMODE, 1006
PHEAD, 986
PHEAD0, 986
PIPESTREAM, 987
PISYMBOL, 1022
POLYADD, 1058
POLYDIV, 1058
POLYFUN, 990
POLYGCD, 1058
POLYINTFAC, 1058
POLYMUL, 1058
POLYNORM, 1058
POLYREM, 1058
POLYSUB, 1058
POS0_, 1043
POS_, 1043
POSITIVEONLY, 1060
POSNEG, 1059
PRECALCSTREAM, 988
PRELOWERAFTER, 991
PRENOACTION, 991
PREPROONLY, 993
PRERAISEAFTER, 991
PREREADSTREAM, 987
PREREADSTREAM2, 988
PREREADSTREAM3, 988
PRETYPEDO, 995
PRETYPEIF, 995
PRETYPEINSIDE, 995
PRETYPENONE, 995
PRETYPEPROCEDURE, 995
PRETYPESWITCH, 995
PREVARSTREAM, 987
PRIMENUMBER, 1021
PRINTALL, 1042
PRINTCONTENT, 1041
PRINTCONTENTS, 1041
PRINTLFILE, 1042
PRINTOFF, 1041
PRINTON, 1041
PRINTONEFUNCTION, 1042
PRINTONETERM, 1042
PROCEDUREFILE, 989
PROPERORDERFLAG, 1057
PUTFIRST, 1022
PVFUNV, 1071
PVFUNWP, 1071
Q_, 1044
QUADRUPLEFORTRANMODE, 1007
QUADRUPLEPRECISIONFLAG, 1007
RANDOMFUNCTION, 1020
RANPERM, 1020
RATIO, 1033
RATIONALMODE, 1047
RBRACE, 1002
RCYCLESYMMETRIC, 1049
READSPECTATORFLAG, 1070
REDEFINEDEXPRESSION, 1042
REDUCEMODE, 1006
REGULAR, 1057
REGULAREXPRESSION, 1042
REGULARSYMBOL, 1065
REMFUNCTION, 1017
REPLACEARG, 1062
REPLACELOOP, 1057
REPLACEMENT, 1013
REVERSEARG, 1063
REVERSEFILESTREAM, 988
REVERSEFUNCTION, 1013
REVERSEORDER, 1049
RHSIDE, 1006
ROOTFUNCTION, 1016
RPARENTHESIS, 1002
SEARCHINGPRECASE, 993
SEARCHINGPREENDSWITCH, 993
SELECTARG, 1064
SEPARATESYMBOL, 1023
SETBIT, 1055
SETEXP, 1010
SETFUNCTION, 1012
SETNUMBER, 1036
SETONLY, 998
SETSET, 1010
SETTONUM, 1035
SETUPFILE, 989
SIGFUNCTION, 1015
SIGNFUN, 1014
SINFUNCTION, 1018
SINHFUNCTION, 1019
SKIP, 1038
SKIPGEXPRESSION, 1039
SKIPLEXPRESSION, 1038
SKIPUNHIDEGEXPRESSION, 1041
SKIPUNHIDELEXPRESSION, 1041
SNUMBER, 1011
SORT, 1054

SORTANTIPOWER, 1009
SORTHIGHFIRST, 1008
SORTLOWFIRST, 1008
SORTMODULE, 989
SORTPOWERFIRST, 1008
SPECIFICATION, 996
SPECTATOREXPRESSION, 1041
SQRTFUNCTION, 1018
STATEMENT, 996
STATSOUT, 994
STORE, 1054
STOREDEXPRESSION, 1039
STOREMODULE, 990
SUBAFTER, 1050
SUBAFTERNOT, 1051
SUBALL, 1050
SUBDISORDER, 1050
SUBEXPR, 1052
SUBEXPRESSION, 1009
SUBEXPSIZE, 1037
SUBMANY, 1050
SUBMASK, 1050
SUBMULTI, 1049
SUBONCE, 1049
SUBONLY, 1050
SUBROUTINEFILE, 989
SUBSELECT, 1050
SUBSORT, 1046
SUBVECTOR, 1050
SUMF1, 1012
SUMF2, 1012
SUMMEDIND, 1045
SUMNUM1, 1033
SUMNUM2, 1033
SYMBOL, 1009
SYMBOL_, 1044
SYMBOLONLY, 997
SYMMETRIC, 1049
SYMMETRIZE, 1033
SYMTONUM, 1034
SYMTOSUB, 1034
SYMTOSYM, 1034
TABLEBASEFILE, 989
TABLEFUNCTION, 1016
TABLESTUB, 1018
TAKETRACE, 1033
TANFUNCTION, 1019
TANHFUNCTION, 1019
TCOMMA, 1002
TCONJUGATE, 1004
TDELTA, 1001
TDIVIDE, 1003
TDOLLAR, 1001
TDOT, 1002
TDOTPRODUCT, 1001
TDUBIOUS, 1001
TEMPTY, 1004
TENDOFIT, 1003
TENSORFUNCTION, 1043
TENVEC, 1033
TERMFUNCTION, 1016
TERMSINBRACKET, 1018
TERMSINEXPR, 1017
TEXPRESSION, 1001
TFUN, 1071
TFUN1, 1071
TFUNCLOSE, 1003
TFUNCTION, 1000
TFUNOPEN, 1002
TGENINDEX, 1004
THETA, 1015
THETA2, 1015
THREADSDEBUG, 994
TINDEX, 1000
TMINUS, 1003
TMPPOLYFUN, 1011
TMULTIPLY, 1003
TNOT, 1003
TNUMBER, 1001
TNUMBER1, 1004
TOBEFACTORED, 1048
TOBEUNFACTORED, 1048
TOLYNDON, 1063
TOLYNDONR, 1063
TOOUTPUT, 996
TOPOLYNOMIALFLAG, 1065
TPLUS, 1003
TPOWER, 1003
TPOWER1, 1004
TSET, 1000
TSETCLOSE, 1004
TSETDOL, 1005
TSETNUM, 1005
TSETOPEN, 1004
TSGAMMA, 1005
TSUBEXP, 1001
TSYMBOL, 1000
TVECTOR, 1000
TWILDARG, 1002
TWILDCARD, 1002
TYPEADJUSTBOUNDS, 1029
TYPEAPPLY, 1030
TYPEAPPLYRESET, 1030
TYPEARG, 1026
TYPEARGEXPLODE, 1031
TYPEARGHEADSIZE, 1037
TYPEARGIMPLODE, 1031
TYPEARGTOEXTRASYMBOL, 1032
TYPEASSIGN, 1028
TYPECHAININ, 1030
TYPECHAINOUT, 1030
TYPECHISHOLM, 1026
TYPECOUNT, 1025
TYPEDENOMINATORS, 1031
TYPEDETCURDUM, 1029
TYPEDISCARD, 1025

TYPEDOLOOP, 1032
TYPEDROPCOEFFICIENT, 1031
TYPEDROPSYMBOLS, 1032
TYPEELIF, 1026
TYPEELSE, 1026
TYPEENDDOLOOP, 1032
TYPEENDIF, 1026
TYPEENDREPEAT, 1025
TYPEEXIT, 1027
TYPEEXPRESSION, 1024
TYPEFACTARG, 1028
TYPEFACTARG2, 1028
TYPEFACTOR, 1031
TYPEFINDLOOP, 1029
TYPEFPRINT, 1027
TYPEFROMPOLYNOMIAL, 1032
TYPEGOTO, 1025
TYPEIDNEW, 1024
TYPEIDOLD, 1025
TYPEIF, 1026
TYPEINEXPRESSION, 1029
TYPEINSIDE, 1029
TYPEISFUN, 1005
TYPEISMYSTERY, 1005
TYPEISSUB, 1005
TYPEMERGE, 1030
TYPEMULT, 1025
TYPENORM, 1027
TYPENORM2, 1027
TYPENORM3, 1027
TYPENORM4, 1031
TYPEOPERATION, 1025
TYPEPRINT, 1027
TYPEPUTINSIDE, 1032
TYPEREDEFPRE, 1027
TYPERENUMBER, 1028
TYPEREPEAT, 1025
TYPEREVERSE, 1026
TYPESETEXIT, 1027
TYPESORT, 1029
TYPESPLITARG, 1028
TYPESPLITARG2, 1028
TYPESPLITFIRSTARG, 1030
TYPESPLITLASTARG, 1030
TYPESTUFFLE, 1031
TYPESUM, 1026
TYPESUMFIX, 1028
TYPETERM, 1029
TYPETESTUSE, 1030
TYPETOPOLYNOMIAL, 1032
TYPETOSPECTATOR, 1032
TYPETRANSFORM, 1031
TYPETRY, 1028
TYPEUNRAVEL, 1029
UNHIDE, 1038
UNHIDEGEXPRESSION, 1040
UNHIDELEXPRESSION, 1040
UNPACK, 1060
USEDFLAG, 1046
VARNAMES, 1061
VARTYPECOMPLEX, 1048
VARTYPEIMAGINARY, 1048
VARTYPEMINUS, 1049
VARTYPENONE, 1048
VARTYPEROOTOFUNITY, 1048
VECTBIT, 1055
VECTOMIN, 1034
VECTOR, 1009
VECTOR_, 1044
VECTORONLY, 997
VECTOSUB, 1034
VECTOVEC, 1034
VOID, 996
VORTRANMODE, 1006
WILDARGFUN, 1018
WILDARGINDEX, 1024
WILDARGSYMBOL, 1023
WILDARGVECTOR, 1024
WILDCARDS, 1035
WILDDUMMY, 1033
WITHAUTO, 997
WITHERROR, 987
WITHOUTERROR, 987
WITHOUTSEMICOLON, 992
WITHSEMICOLON, 991
WRITEOUT, 994
YESLYNDON, 1064
Z_, 1044
full
PF_BUFFER, 187
FullCleanUp
module.c, 1198
FULLMAX
form3.h, 949
fullname
dbase, 54
FullProto
N_const, 134
FullRenumber
declare.h, 723
reshuf.c, 1845
FullSymmetrize
declare.h, 641
function.c, 1085
FUN_INFO, 83
commute, 84
location, 83
numargs, 83
numfunnies, 83
numwildcards, 84
symmet, 84
tensor, 84
FunArg
T_const, 250
funargs
N_const, 137

FUNBIT
ftypes.h, 1055
func
KEYWORD, 100
ReNuMbEr, 207
FUNCTION
ftypes.h, 1010
FuNcTiOn, 85
commute, 86
complex, 86
dimension, 88
flags, 87
maxnumargs, 88
minnumargs, 88
name, 86
namesize, 87
node, 87
number, 86
spec, 87
symmetric, 87
symminfo, 86
tabl, 86
function.c, 1083, 1087
ChainIn, 1086
ChainOut, 1086
CompGroup, 1085
FullSymmetrize, 1085
MakeDirty, 1084
MarkDirty, 1084
MatchFunction, 1086
PolyFunClean, 1084
PolyFunDirty, 1084
ScanFunctions, 1086
SymFind, 1085
SymGen, 1085
Symmetrize, 1085
FunctionList
C_const, 20
FUNCTIONONLY
ftypes.h, 997
FUNCTIONS
structs.h, 2122
Functions
C_const, 23
functions
variable.h, 2405
FUNCTIONSORT
ftypes.h, 1046
FUNHEAD
ftypes.h, 1037
funinds
N_const, 138
FunInfo
N_const, 139
funisize
N_const, 143
FunLevel
declare.h, 618
reshuf.c, 1845
funlocs
N_const, 137
FunMatchCy
declare.h, 624
symmetr.c, 2150
FunMatchSy
declare.h, 624
symmetr.c, 2151
FunNam
FixedGlobals, 80
funnum
ReNuMbEr, 208
FUNNYDOLLAR
ftypes.h, 1046
FUNNYVEC
ftypes.h, 1045
FUNNYWILD
ftypes.h, 1045
funpowers
C_const, 34
FunScratch
N_const, 139
funsize
NeStInG, 154
FunSorts
N_const, 139
FUNTOFUN
ftypes.h, 1035
funwith
DICTIONARY, 56
fval
OPTIMIZE, 169
fwin.h, 1109, 1111
ALTSEPARATOR, 1110
CARRIAGERETURN, 1109
LINEFEED, 1109
P_term, 1110
PATHSEPARATOR, 1110
SEPARATOR, 1110
WITH_ENV, 1110
WITHRETURN, 1110
WITHSYSTEM, 1110
fwrite_cached
checkpoint.c, 328

gamm
TrAcEs, 267
GAMMA
ftypes.h, 1012
GAMMA1
ftypes.h, 1045
GAMMA5
ftypes.h, 1045
gamma5
TrAcEs, 268
GAMMA6
ftypes.h, 1045
GAMMA7

ftypes.h, 1045
GAMMAFIVE
ftypes.h, 1012
GAMMAFUNCTION
ftypes.h, 1043
GAMMAI
ftypes.h, 1012
GAMMASEVEN
ftypes.h, 1012
GAMMASIX
ftypes.h, 1012
GarbHand
sort.c, 1958
gcd_heuristic_failed, 88
GCDclean
declare.h, 762
ratio.c, 1743
GCDFUNCTION
ftypes.h, 1017
GCDfunction
declare.h, 761
ratio.c, 1740
GCDfunction3
declare.h, 761
ratio.c, 1740
GcdLong
declare.h, 618
reken.c, 1795
GCDMAX
reken.c, 1788
GCDterms
declare.h, 762
ratio.c, 1742
gcmod
M_const, 108
gcNumDollars
M_const, 118
gCnumpows
M_const, 122
gCodesFlag
M_const, 115
gDefDim
M_const, 114
gDefDim4
M_const, 114
GENCOMMUTE
ftypes.h, 1055
GENERALFUNCTION
ftypes.h, 1042
GenerateFactors
compiler.c, 501
declare.h, 721
Generator
declare.h, 619
proces.c, 1671
GENNONCOMMUTE
ftypes.h, 1055
GenSpace
sOrT, 222
GenTerms
sOrT, 222
GetAppendChannel
declare.h, 663
tools.c, 2298
GetAutoName
declare.h, 656
names.c, 1259
GETBIDENTITY
declare.h, 593
GetBinom
declare.h, 620
reken.c, 1796
GETCFROMEXTCHANNEL
declare.h, 594
GetChannel
declare.h, 663
tools.c, 2297
GetChar
declare.h, 669
pre.c, 1571
GETCOEF
declare.h, 574
GetContent
declare.h, 734
execute.c, 864
getCurrentExternalChannel
extcmd.c, 897
GetDbase
declare.h, 741
minos.c, 1171
GetDollar
declare.h, 682
names.c, 1260
GetDollarNumber
declare.h, 673
pre.c, 1584
GetDolNum
declare.h, 728
dollar.c, 816
GetDoParam
compcomm.c, 414
declare.h, 760
GetFile
R_const, 203
GetFirstBracket
declare.h, 734
execute.c, 864
GetFirstTerm
declare.h, 734
execute.c, 864
GetFromSpectator
declare.h, 770
spectator.c, 2014
GetFromStore
declare.h, 620
store.c, 2053

GetFromStream
declare.h, 653
tools.c, 2293
GETIDENTITY
declare.h, 593
GetIfDollarFactor
compcomm.c, 414
declare.h, 760
GetIfDollarNum
declare.h, 610
if.c, 1112
GetInput
pre.c, 1571
GetLabel
declare.h, 720
names.c, 1267
GetLastExprName
declare.h, 655
names.c, 1259
GetLong
declare.h, 620
reken.c, 1795
GetLongModInverses
declare.h, 635
reken.c, 1794
GetModInverses
declare.h, 634
reken.c, 1794
GetMoreFromMem
declare.h, 621
store.c, 2053
GetMoreTerms
declare.h, 620
store.c, 2053
GetName
declare.h, 655
names.c, 1259
GetNode
declare.h, 655
names.c, 1259
GetOName
declare.h, 656
names.c, 1259
GetOneFile
R_const, 204
GetOneTerm
declare.h, 621
store.c, 2053
GetPID
form3.h, 954
GetPosFile
declare.h, 663
tools.c, 2297
GetPreVar
declare.h, 667
pre.c, 1571
GetRunningTime
startup.c, 2027
GetSetupPar
declare.h, 668
setfile.c, 1925
GETSTOP
declare.h, 574
GetStreamPosition
declare.h, 686
tools.c, 2294
GetTable
declare.h, 621
store.c, 2055
GetTableName
declare.h, 743
minos.c, 1172
GETTERM
declare.h, 594
GetTerm
declare.h, 621
store.c, 2052
GetVar
declare.h, 656
names.c, 1260
GetWildcardName
declare.h, 685
names.c, 1262
gextrasym
M_const, 109
gextrasymbols
M_const, 127
gFinalStats
M_const, 116
GFIVE
ftypes.h, 1051
gFortran90Kind
M_const, 109
gfunpowers
M_const, 115
ggextrasym
M_const, 109
ggextrasymbols
M_const, 127
ggFinalStats
M_const, 116
ggIndentSpace
M_const, 126
ggNoSpacesInNumbers
M_const, 118
ggnumextrasym
M_const, 119
ggOldFactArgFlag
M_const, 119
ggOldGCDflag
M_const, 120
ggOldParallelStats
M_const, 118
ggProcessStats
M_const, 117
ggShortStatsMax

M_const, 127
ggStatsFlag
M_const, 123
ggThreadBalancing
M_const, 117
ggThreadBucketSize
M_const, 112
ggThreadsFlag
M_const, 117
ggThreadSortFileSynch
M_const, 117
ggThreadStats
M_const, 116
ggWTimeStatsFlag
M_const, 120
GIDENT
ftypes.h, 1051
gIndentSpace
M_const, 126
ginsidefirst
M_const, 113
gIsFortran90
M_const, 119
gLineLength
M_const, 123
GLOBAL
ftypes.h, 1054
GLOBALEXPRESSION
ftypes.h, 1039
GlobalFunctions
variable.h, 2407
GlobalIndices
variable.h, 2407
Globalize
declare.h, 685
names.c, 1267
GLOBALMODULE
ftypes.h, 989
globalnamefill
NaMeTree, 153
globalnodefill
NaMeTree, 153
GlobalSetElements
variable.h, 2408
GlobalSets
variable.h, 2407
GLOBALSPECTATORFLAG
ftypes.h, 1071
GlobalSymbols
variable.h, 2407
GlobalVectors
variable.h, 2407
Glue
declare.h, 621
ratio.c, 1740
GMINUS
ftypes.h, 1051
gmodmode
M_const, 121
gNamesFlag
M_const, 115
gncmod
M_const, 121
gNoSpacesInNumbers
M_const, 118
gnpowmod
M_const, 121
GNUC_PREREQ
parallel.h, 1495
gNumDictionaries
O_const, 164
gnumelements
DICTIONARY, 56
gnumextrasym
M_const, 119
gNumPre
P_const, 179
gOldFactArgFlag
M_const, 119
gOldGCDflag
M_const, 120
gOldParallelStats
M_const, 118
gOutNumberType
M_const, 121
gOutputMode
M_const, 121
gOutputSpaces
M_const, 121
gparallelflag
M_const, 116
GPLUS
ftypes.h, 1051
gPolyFun
M_const, 125
gPolyFunExp
M_const, 125
gPolyFunInv
M_const, 125
gPolyFunPow
M_const, 125
gPolyFunType
M_const, 125
gPolyFunVar
M_const, 125
gpowmod
M_const, 108
gProcessBucketSize
M_const, 111
gProcessStats
M_const, 117
gproperorderflag
M_const, 115
GREATER
ftypes.h, 1053
GREATEREQUAL

ftypes.h, 1053
greedymaxperc
OPTIMIZE, 171
greedyminnum
OPTIMIZE, 170
greedytimelimit
OPTIMIZE, 170
gShortStatsMax
M_const, 126
gSizeCommuteInSet
M_const, 116
gSortType
M_const, 115
gStatsFlag
M_const, 115
gThreadBalancing
M_const, 117
gThreadBucketSize
M_const, 112
gThreadsFlag
M_const, 117
gThreadSortFileSynch
M_const, 117
gThreadStats
M_const, 116
gTokensWriteFlag
M_const, 115
gUniTrace
M_const, 122
gUnitTrace
M_const, 121
gWTimeStatsFlag
M_const, 120
gzipCompress
R_const, 202
GZIPDEFAULT
fsizes.h, 970

HAAKJE
ftypes.h, 1011
HAAKJE0
ftypes.h, 1010
HALFMAX
fsizes.h, 964
halfmod
C_const, 24
Handle
FiLeDaTa, 77
handle
ChAnNeL, 51
dbase, 54
FiLe, 75
StreaM, 234
HANDLERS, 88
newhandle, 89
newlogonly, 89
oldhandle, 89
oldlogonly, 89
oldprinttype, 89
oldsilent, 89
havesortdir
M_const, 127
HEADERFILE
ftypes.h, 989
headermark
STOREHEADER, 229
headnode
NaMeTree, 153
helptb
tables.c, 2184
hi
VaRrEnUm, 271
HIDDENGEXPRESSION
ftypes.h, 1039
HIDDENLEXPRESSION
ftypes.h, 1039
HIDE
ftypes.h, 1038
hidefile
R_const, 200
HIDEGEXPRESSION
ftypes.h, 1040
HideLevel
C_const, 43
hidelevel
ExPrEsSiOn, 69
HIDELEXPRESSION
ftypes.h, 1040
HideSize
M_const, 110
highfirst
setfile.c, 1927
HighWarning
declare.h, 651
message.c, 1154
HoldFlag
M_const, 123
horner
OPTIMIZE, 169
hornerdirection
OPTIMIZE, 170
HowMany
declare.h, 739
if.c, 1112
hparallelflag
M_const, 116
hProcessBucketSize
M_const, 111

i
bufIPstruct, 14
iblocks
dbase, 54
iBufError
P_const, 177
iBuffer
C_const, 25
iBufferSize

C_const, 29
idallflag
T_const, 247
idallmaxnum
T_const, 247
idallnum
T_const, 247
IDFUNCTION
ftypes.h, 1021
idfunctionflag
N_const, 147
IDHEAD
ftypes.h, 1051
idoption
C_const, 33
iexp
declare.h, 665
tools.c, 2304
if.c, 1111, 1113
DoIfStatement, 1112
DoubleIfBuffers, 1112
FindVar, 1112
GetIfDollarNum, 1112
HowMany, 1112
IfCount
C_const, 25
IFDOLLAR
ftypes.h, 1052
IFDOLLAREXTRA
ftypes.h, 1052
IFEXPRESSION
ftypes.h, 1052
IfHeap
C_const, 25
IFISFACTORIZED
ftypes.h, 1053
IfLevel
C_const, 43
IFOCCURS
ftypes.h, 1053
IfStack
C_const, 25
IfSumCheck
C_const, 28
IMPLODEARG
ftypes.h, 1062
INANDOUT
minos.h, 1193
inbuffer
FiLe, 75
StreaM, 234
IncDir
M_const, 108
incdollar
DoLoOp, 65
INCEXPRESSION
ftypes.h, 1039
INCLENG
declare.h, 574
Include
declare.h, 672
pre.c, 1575
incnum
DoLoOp, 64
incoef
SHvariables, 214
IndDum
M_const, 120
N_const, 147
INDENTSPACE
fsizes.h, 972
IndentSpace
O_const, 165
INDEX
ftypes.h, 1009
InDeX, 90
dimension, 90
flags, 91
name, 90
namesize, 91
nmin4, 91
node, 91
number, 90
type, 90
Index
FiLeDaTa, 77
index
DoLIArS, 61
PF_BUFFER, 188
index.c, 1125, 1126
ClearBracketIndex, 1126
FindBracket, 1125
OpenBracketIndex, 1126
PutBracketInIndex, 1125
PutInside, 1126
INDEX_
ftypes.h, 1044
indexblock, 91
flags, 92
objects, 92
position, 92
previousblock, 92
indexbuffer
BrAcKeTiNfO, 12
indexbuffersize
BrAcKeTiNfO, 13
INDEXENTRY
structs.h, 2121
InDeXeNtRy, 93
CompressSize, 94
length, 94
name, 95
nfunctions, 95
nindices, 94
nsymbols, 94
nvectors, 95

position, 94
size, 95
variables, 94
indexfill
BrAcKeTiNfO, 13
IndexList
C_const, 20
INDEXONLY
ftypes.h, 997
indi
ReNuMbEr, 207
Indices
C_const, 23
indices
parallel.h, 1495
variable.h, 2406
indnum
ReNuMbEr, 208
INDTOIND
ftypes.h, 1034
INDTOSUB
ftypes.h, 1034
inexprlevel
C_const, 40
inexprstack
C_const, 29
inexprsumcheck
C_const, 40
InFbrack
O_const, 165
infile
R_const, 200
INFILEINDEX
structs.h, 2120
info
dbase, 53
InFunction
declare.h, 621
proces.c, 1668
InHiBuf
R_const, 201
inicbufs
comtool.c, 538
declare.h, 664
inictable
compiler.c, 499
IniFbuffer
comtool.c, 542
declare.h, 761
iniinfo, 96
entriesinindex, 96
firstindexblock, 96
firstnameblock, 97
lastindexblock, 96
lastnameblock, 97
numberofindexblocks, 96
numberofnamesblocks, 97
numberoftables, 96
IniLine
declare.h, 622
sch.c, 1887
INILOCK
declare.h, 591
IniModule
declare.h, 669
pre.c, 1572
InInBuf
R_const, 201
INIRWLOCK
declare.h, 592
IniSpecialModule
declare.h, 669
pre.c, 1572
INITNAMESIZE
fsizes.h, 965
INITNODESIZE
fsizes.h, 965
iniTools
declare.h, 753
tools.c, 2302
initPresetExternalChannels
declare.h, 745
extcmd.c, 896
InitRecovery
checkpoint.c, 328
inivar.h, 1135, 1136
A, 1135
BufferForOutput, 1136
FG, 1135
fixedsets, 1135
setupfilename, 1136
IniVars
startup.c, 2026
iniwranf
declare.h, 747
reken.c, 1798
inline
form3.h, 948
inlist
TrAcEn, 262
TrAcEs, 264
inmem
ExPrEsSiOn, 69
InnerTest
C_const, 45
inNum
sOrT, 224
InNumMem
T_const, 246
InOutBuf
P_const, 176
inparallelflag
C_const, 36
inPatches
sOrT, 220
inprimelist

T_const, 251
InputFileName
M_const, 108
INPUTONLY
minos.h, 1194
INPUTOUT
ftypes.h, 994
inscbuf
INSIDEINFO, 99
inscheme
O_const, 161
InScratch
N_const, 141
InScratch1
N_const, 141
InsertArg
argument.c, 277
declare.h, 692
InsertTerm
declare.h, 622
proces.c, 1669
inside
P_const, 175
InsideDollar
declare.h, 731
dollar.c, 811
insidefirst
C_const, 31
INSIDEINFO, 97
buffer, 98
inscbuf, 99
numdollars, 98
oldcbuf, 98
oldcnumlhs, 99
oldcompiletype, 98
oldnumpotmoddollars, 98
oldparallelflag, 98
oldrbuf, 99
size, 98
insidelevel
C_const, 40
insidestack
C_const, 29
insidesumcheck
C_const, 40
INSLENGTH
sort.c, 1949
InsTableTree
declare.h, 730
tables.c, 2179
InsTree
comtool.c, 541
declare.h, 721
insubexpbuffers
compiler.c, 502
Interact
M_const, 113
INTFUNCTION
ftypes.h, 1015
INTOHIDE
ftypes.h, 1038
INTOHIDEGEXPRESSION
ftypes.h, 1040
INTOHIDELEXPRESSION
ftypes.h, 1040
intracestack
N_const, 143
INUSE
ftypes.h, 1067
INVERSEFACTORIAL
ftypes.h, 1014
INVERSEFUNCTION
ftypes.h, 1021
INVERSETABLE
ftypes.h, 1060
invfacnum
M_const, 124
InvPoly
declare.h, 765
ratio.c, 1744
iPatches
sOrT, 221
iPointer
C_const, 25
iranf
declare.h, 747
reken.c, 1799
IsBracket
O_const, 165
ISCOMPRESSED
ftypes.h, 1047
ISEQUALPOS
declare.h, 586
ISEQUALPOSINC
declare.h, 584
IsExponentSign
declare.h, 767
dict.c, 790
ISFACTORIZED
ftypes.h, 1048
ISFORTRAN90
ftypes.h, 1008
IsFortran90
C_const, 38
ISGEPOS
declare.h, 587
ISGEPOSINC
declare.h, 584
IsIdStatement
compiler.c, 500
declare.h, 718
ISINRHS
ftypes.h, 1047
ISLESSPOS
declare.h, 586
IsLikeVector

declare.h, 666
tools.c, 2305
ISLYNDON
ftypes.h, 1063
ISLYNDONR
ftypes.h, 1063
ISMINPOS
declare.h, 586
IsMultipleOf
declare.h, 735
dollar.c, 812
IsMultiplySign
declare.h, 767
dict.c, 790
ISNEGPOS
declare.h, 587
isnextchar
StreaM, 235
ISNOTEQUALPOS
declare.h, 586
ISNOTFORTRAN90
ftypes.h, 1008
ISNOTZEROPOS
declare.h, 587
ISPOSPOS
declare.h, 587
IsRHS
compiler.c, 500
declare.h, 718
IsSetMember
declare.h, 735
dollar.c, 812
iStop
C_const, 26
ISUNMODIFIED
ftypes.h, 1047
ISYMBOL
ftypes.h, 1022
ISZERO
ftypes.h, 1047
ISZEROPOS
declare.h, 587
ival
OPTIMIZE, 169

j3
TrAcEs, 265
j4
TrAcEs, 265

KEEPZERO
ftypes.h, 1048
KeptInHold
R_const, 203
KEYWORD, 99
flags, 100
func, 100
name, 100
type, 100
KILL
pre.c, 1569
KILLALL
pre.c, 1569
killSignal
X_const, 274
killWholeGroup
X_const, 274
kstep
TrAcEs, 266
ktoi
sOrT, 220

LabelNames
C_const, 26
Labels
C_const, 26
LARGEBUFFER
fsizes.h, 968
LargeSize
sOrT, 221
last
SeTs, 211
lastdollar
DoLoOp, 65
lastindexblock
iniinfo, 96
lastLen
longMultiStruct, 103
lastnameblock
iniinfo, 97
lastnum
DoLoOp, 64
LBRACE
ftypes.h, 1002
lBuffer
sOrT, 218
lc3
TrAcEs, 267
lc4
TrAcEs, 267
LcmLong
declare.h, 618
reken.c, 1796
lDefDim
C_const, 32
lDefDim4
C_const, 32
LeaveNegative
T_const, 248
left
NaMeNode, 149
NoDe, 155
tree, 269
length
InDeXeNtRy, 94
lenLONG
STOREHEADER, 229
lenPOINTER

STOREHEADER, 229
lenPOS
STOREHEADER, 229
lenWORD
STOREHEADER, 229
LESS
ftypes.h, 1053
LESSEQUAL
ftypes.h, 1053
level
R_const, 205
SHvariables, 215
TrAcEn, 262
TrAcEs, 268
LEVICIVITA
ftypes.h, 1013
lFill
sOrT, 218
lhdollarerror
P_const, 178
lhdollarflag
C_const, 37
lhs
CbUf, 47
DICTIONARY_ELEMENT, 57
LHSIDE
ftypes.h, 1005
LHSIDEX
ftypes.h, 1005
LI2FUNCTION
ftypes.h, 1020
lijst
LIST, 101
LINEFEED
fwin.h, 1109
unix.h, 2397
LineLength
C_const, 43
linenumber
StreaM, 232
LINFUNCTION
ftypes.h, 1020
LinkTree
declare.h, 657
names.c, 1261
LIST, 100
lijst, 101
maxnum, 101
message, 101
num, 101
numclear, 102
numglobal, 101
numtemp, 102
size, 101
LISTEDLOOP
ftypes.h, 995
listinprint
N_const, 139
ListPoly
T_const, 245
ListSymbols
T_const, 245
ListSymbolsSize
T_const, 248
lloser
NoDe, 155
LNFUNCTION
ftypes.h, 1018
LNUMBER
ftypes.h, 1011
lo
VaRrEnUm, 271
LOADDOLLAR
ftypes.h, 1036
LoadInputFile
declare.h, 668
tools.c, 2293
LoadInstruction
declare.h, 669
pre.c, 1573
loadmode
PROCEDURE, 197
LoadStatement
declare.h, 670
pre.c, 1573
LocalConvertToPoly
declare.h, 758
notation.c, 1369
LOCALEXPRESSION
ftypes.h, 1038
LocateBase
declare.h, 638
minos.c, 1170
LocateFile
declare.h, 654
tools.c, 2293
location
FUN_INFO, 83
LOCK
declare.h, 591
locwildvalue
T_const, 250
log
ParallelVars, 182
LogFileName
M_const, 108
LogHandle
C_const, 43
LogType
M_const, 122
LongCopy
declare.h, 683
tools.c, 2301
LongLongCopy
declare.h, 683
tools.c, 2301

longMultiStruct, 102
buffer, 103
bufpos, 103
lastLen, 103
next, 103
nPacks, 103
packpos, 103
LONGNUMBER
ftypes.h, 1052
LongToLine
declare.h, 623
sch.c, 1888
LOOKINGFORELSE
ftypes.h, 992
LOOKINGFORENDIF
ftypes.h, 992
LookInStream
declare.h, 654
tools.c, 2293
LoopList
P_const, 175
LowerSortLevel
sort.c, 1961
lowfirst
setfile.c, 1927
LPARENTHESIS
ftypes.h, 1001
lPatch
sOrT, 223
lPolyFun
C_const, 44
lPolyFunExp
C_const, 44
lPolyFunInv
C_const, 44
lPolyFunPow
C_const, 44
lPolyFunType
C_const, 44
lPolyFunVar
C_const, 44
LRPARENTHESES
ftypes.h, 1004
lSortType
C_const, 35
lsrc
NoDe, 156
lTop
sOrT, 218
lUniTrace
C_const, 40
lUnitTrace
C_const, 42
Lus
declare.h, 724
lus.c, 1140
lus.c, 1139, 1141
FindLus, 1140
Lus, 1140
SortTheList, 1140
lWorkPointer
T_const, 245
lWorkSize
R_const, 201
lWorkSpace
T_const, 242

M
AllGlobals, 6
m
MODNUM, 129
M_alloc
declare.h, 593
M_check1
tools.c, 2302
M_const, 104
atstartup, 126
BracketFactors, 127
CompressSize, 110
countfunnum, 125
dbufnum, 114
denomnum, 124
dollarzero, 126
DumInd, 120
exitflag, 126
expnum, 124
facnum, 124
fbuffersize, 119
FileOnlyFlag, 113
FortranCont, 123
gcmod, 108
gcNumDollars, 118
gCnumpows, 122
gCodesFlag, 115
gDefDim, 114
gDefDim4, 114
gextrasym, 109
gextrasymbols, 127
gFinalStats, 116
gFortran90Kind, 109
gfunpowers, 115
ggextrasym, 109
ggextrasymbols, 127
ggFinalStats, 116
ggIndentSpace, 126
ggNoSpacesInNumbers, 118
ggnumextrasym, 119
ggOldFactArgFlag, 119
ggOldGCDflag, 120
ggOldParallelStats, 118
ggProcessStats, 117
ggShortStatsMax, 127
ggStatsFlag, 123
ggThreadBalancing, 117
ggThreadBucketSize, 112
ggThreadsFlag, 117
ggThreadSortFileSynch, 117

ggThreadStats, 116
ggWTimeStatsFlag, 120
gIndentSpace, 126
ginsidefirst, 113
gIsFortran90, 119
gLineLength, 123
gmodmode, 121
gNamesFlag, 115
gncmod, 121
gNoSpacesInNumbers, 118
gnpowmod, 121
gnumextrasym, 119
gOldFactArgFlag, 119
gOldGCDflag, 120
gOldParallelStats, 118
gOutNumberType, 121
gOutputMode, 121
gOutputSpaces, 121
gparallelflag, 116
gPolyFun, 125
gPolyFunExp, 125
gPolyFunInv, 125
gPolyFunPow, 125
gPolyFunType, 125
gPolyFunVar, 125
gpowmod, 108
gProcessBucketSize, 111
gProcessStats, 117
gproperorderflag, 115
gShortStatsMax, 126
gSizeCommuteInSet, 116
gSortType, 115
gStatsFlag, 115
gThreadBalancing, 117
gThreadBucketSize, 112
gThreadsFlag, 117
gThreadSortFileSynch, 117
gThreadStats, 116
gTokensWriteFlag, 115
gUniTrace, 122
gUnitTrace, 121
gWTimeStatsFlag, 120
havesortdir, 127
HideSize, 110
HoldFlag, 123
hparallelflag, 116
hProcessBucketSize, 111
IncDir, 108
IndDum, 120
InputFileName, 108
Interact, 113
invfacnum, 124
LogFileName, 108
LogType, 122
matchfunnum, 125
MaxBracketBufferSize, 111
maxFlevels, 118
MaxParLevel, 113
MaxStreamSize, 110
MaxTal, 120
MaxTer, 110
MaxWildcards, 122
mTraceDum, 122
MultiRun, 118
NumFixedFunctions, 114
NumFixedSets, 114
NumSpectatorFiles, 119
NumStoreCaches, 126
OffsetIndex, 122
OffsetVector, 122
OldChildTime, 112
OldMilliTime, 112
oldnumextrasymbols, 109
OldOrderFlag, 124
OldSecTime, 112
onerhs, 127
Ordering, 123
OutBuffer, 109
OutBufSize, 113
Path, 109
PrintTotalSize, 119
qError, 123
rbufnum, 114
RepMax, 122
resetTimeOnClear, 118
S0, 107
sbufnum, 114
ScratSize, 110
SetupDir, 109
SetupFile, 109
shmWinSize, 111
silent, 123
SIOsize, 110
SizeForSpectatorFiles, 120
SizeStoreCache, 110
SkipClears, 115
SLargeSize, 111
SMaxFpatches, 113
SMaxPatches, 113
SpectatorFiles, 110
SpectatorSize, 112
SSmallEsize, 111
SSmallSize, 111
StdOut, 113
STermsInSmall, 111
sumnum, 124
sumpnum, 124
SumTime, 112
TempDir, 108
TempSortDir, 108
termfunnum, 124
totalnumberofthreads, 116
tracebackflag, 123
WilInd, 121
WorkSize, 112
zbufnum, 114

zeropos, 107
zerorhs, 127
M_free
declare.h, 685
tools.c, 2302
M_print
tools.c, 2303
main
startup.c, 2026
MAINSORT
ftypes.h, 1046
MAJORVERSION
form3.h, 947
MAKEARGS
ftypes.h, 1062
MakeDate
tools.c, 2301
MakeDirty
declare.h, 623
function.c, 1084
MakeDollarInteger
declare.h, 727
dollar.c, 814
MakeDollarMod
declare.h, 727
dollar.c, 815
MakeDubious
declare.h, 656
names.c, 1266
MakeGlobal
execute.c, 862
MakeInteger
argument.c, 279
declare.h, 691
MakeInverses
reken.c, 1793
MakeLongRational
declare.h, 729
reken.c, 1796
MakeMod
argument.c, 279
declare.h, 691
MakeModTable
reken.c, 1798
MakeNameTree
names.c, 1261
MAKERATIONAL
ftypes.h, 1021
MakeRational
declare.h, 729
reken.c, 1796
MakeSetupAllocs
setfile.c, 1926
Malloc
declare.h, 652
tools.c, 2302
Malloc1
declare.h, 652
tools.c, 2302
mallocprotect.h, 1148
MAPLEMODE
ftypes.h, 1006
MarkDirty
declare.h, 623
function.c, 1084
MarkPolyRatFunDirty
declare.h, 590
MaskPointer
N_const, 137
MASTER
parallel.h, 1493
MATCH
ftypes.h, 1052
MatchArgument
declare.h, 625
symmetr.c, 2151
MatchCy
declare.h, 624
symmetr.c, 2150
MatchE
declare.h, 624
symmetr.c, 2150
MATCHFUNCTION
ftypes.h, 1016
MatchFunction
declare.h, 625
function.c, 1086
matchfunnum
M_const, 125
MatchIsPossible
declare.h, 725
smart.c, 1942
MATHEMATICAMODE
ftypes.h, 1006
MaX
declare.h, 573
MAX_OPEN_FILES
form3.h, 954
MAXBASES
minos.h, 1192
MaxBracket
R_const, 203
MAXBRACKETBUFFERSIZE
fsizes.h, 970
MaxBracketBufferSize
M_const, 111
MAXBUILTINFUNCTION
ftypes.h, 1022
MAXCOMBUFRHS
fsizes.h, 971
MAXDOLLARVARIABLES
fsizes.h, 963
MaxDoLoops
variable.h, 2405
MaxDum
R_const, 205

MAXDUMMIES
fsizes.h, 964
MAXENAME
fsizes.h, 964
MaxExprSize
S_const, 209
MAXFILESTREAMSIZE
fsizes.h, 964
MAXFLAGS
fsizes.h, 966
MAXFLEVELS
fsizes.h, 969
maxFlevels
M_const, 118
MAXFPATCHES
fsizes.h, 968
MaxFpatches
sOrT, 223
MAXFUNCTION
ftypes.h, 1015
MaxFunSorts
N_const, 143
maxi
MiNmAx, 128
MAXIF
fsizes.h, 966
MaxIf
C_const, 32
MAXINDEXSIZE
minos.h, 1193
MAXINNAMETREE
fsizes.h, 963
MAXLABELS
fsizes.h, 966
MaxLabels
C_const, 32
MAXLEVELS
fsizes.h, 967
MAXLHS
fsizes.h, 967
maxlhs
CbUf, 49
MAXLINELENGTH
fsizes.h, 972
MAXLONG
form3.h, 949
MAXLOOPS
fsizes.h, 966
MAXMATCH
fsizes.h, 966
MAXMULTIBRACKETLEVELS
fsizes.h, 972
MAXNEST
fsizes.h, 965
maxnum
LIST, 101
maxnumargs
FuNcTiOn, 88
MAXNUMBEROFNONCOMTERMS
normal.c, 1305
MAXNUMBERSIZE
fsizes.h, 965
MaxNumPre
variable.h, 2405
MAXNUMSIZE
fsizes.h, 970
MaxNumStreams
C_const, 33
MAXPARLEVEL
fsizes.h, 965
MaxParLevel
M_const, 113
MAXPATCHES
fsizes.h, 968
MaxPatches
sOrT, 223
MAXPOSITIVE
form3.h, 949
MAXPOSITIVE2
form3.h, 950
MAXPOSITIVE4
form3.h, 950
MAXPOWER
fsizes.h, 963
maxpower
STOREHEADER, 231
SyMbOl, 237
MAXPOWEROF
ftypes.h, 1017
MaxPreAssignLevel
P_const, 179
MaxPreIfLevel
P_const, 178
MAXPRENAMESIZE
fsizes.h, 963
MaxPreTypes
P_const, 177
MaxProcedures
variable.h, 2404
MAXRANGEINDICATOR
ftypes.h, 1062
MaxRenumScratch
N_const, 144
MAXREPEAT
fsizes.h, 965
maxrhs
CbUf, 49
MAXSAVEFUNCTION
fsizes.h, 964
MaxStreamSize
M_const, 110
MAXSUBEXPRESSIONS
fsizes.h, 964
MAXTABLECOMBUF
fsizes.h, 971
MaxTal

M_const, 120
MAXTER
fsizes.h, 964
MaxTer
M_const, 110
maxtermlevel
C_const, 36
maxtermsize
sOrT, 224
MaxTreeSize
CbUf, 50
TaBlEs, 260
maxvar
OPTIMIZERESULT, 173
MAXVARIABLES
fsizes.h, 963
MAXWILDC
fsizes.h, 967
MaxWildcards
M_const, 122
mBer
T_const, 251
MCLEANFLAG
minos.h, 1193
mctsdecaymode
OPTIMIZE, 171
mctsnumexpand
OPTIMIZE, 170
mctsnumkeep
OPTIMIZE, 170
mctsnumrepeat
OPTIMIZE, 170
mctstimelimit
OPTIMIZE, 170
mdefined
TaBlEs, 258
mdel
TrAcEs, 266
mdum
TrAcEs, 266
me
ParallelVars, 181
MemDebugFlag
C_const, 38
MEMORYMACROS
declare.h, 589
mepf
TrAcEs, 265
MergeDotproductLists
declare.h, 764
ratio.c, 1742
MergePatches
declare.h, 625
sort.c, 1959
MergeSymbolLists
declare.h, 764
ratio.c, 1741
MesCall
declare.h, 682
message.c, 1154
MesCerr
declare.h, 626
message.c, 1154
MesComp
declare.h, 626
message.c, 1155
MesNesting
declare.h, 590
MesPrint
message.c, 1154
message
LIST, 101
message.c, 1153, 1156
Error0, 1153
Error1, 1153
Error2, 1154
HighWarning, 1154
MesCall, 1154
MesCerr, 1154
MesComp, 1155
MesPrint, 1154
MesWork, 1154
PrintSeq, 1155
PrintSubTerm, 1155
PrintTerm, 1155
PrintTermC, 1155
PrintWords, 1155
Warning, 1154
MessPreNesting
declare.h, 686
pre.c, 1582
MesWork
message.c, 1154
method
OPTIMIZE, 170
mfac
T_const, 246
MiN
declare.h, 573
MINFUNCTION
ftypes.h, 1014
mini
MiNmAx, 128
MiNmAx, 128
maxi, 128
mini, 128
size, 128
minnumargs
FuNcTiOn, 88
MINORVERSION
form3.h, 947
minos.c, 1167, 1174
AddObject, 1172
AddTableName, 1172
AddToIndex, 1172
CFD, 1168

ComposeTableNames, 1171
convertblock, 1169
convertiniinfo, 1170
convertnamesblock, 1169
CTD, 1168
DeleteObject, 1173
ExistsObject, 1173
FindTableNumber, 1172
FreeTableBase, 1171
GetDbase, 1171
GetTableName, 1172
LocateBase, 1170
minosread, 1169
minoswrite, 1169
NewDbase, 1171
OpenDbase, 1171
PutTableNames, 1172
ReadijObject, 1173
ReadIndex, 1170
ReadIniInfo, 1171
ReadObject, 1173
str_dup, 1169
withoutflush, 1174
WriteIndex, 1170
WriteIndexBlock, 1170
WriteIniInfo, 1171
WriteNamesBlock, 1170
WriteObject, 1173
minos.h, 1191, 1195
DIRTYFLAG, 1193
ELEMENTSIZE, 1193
FROMDISK, 1193
INANDOUT, 1193
INPUTONLY, 1194
MAXBASES, 1192
MAXINDEXSIZE, 1193
MCLEANFLAG, 1193
minosread, 1194
minoswrite, 1194
NAMETABLESIZE, 1193
NOCOMPRESS, 1194
NUMOBJECTS, 1192
OUTPUTONLY, 1194
TODISK, 1193
withoutflush, 1195
minosread
minos.c, 1169
minos.h, 1194
minoswrite
minos.c, 1169
minos.h, 1194
minpower
SyMbOl, 237
MINPOWEROF
ftypes.h, 1017
minsidefirst
C_const, 31
MINSPEC
ftypes.h, 1046
minvar
OPTIMIZERESULT, 172
MinVecArg
T_const, 250
MINVECTOR
ftypes.h, 1010
MIXED
ftypes.h, 996
mkSlaveInfile
ParallelVars, 182
MLOCK
declare.h, 592
MLONG
form3.h, 955
mm
TaBlEs, 256
mnumlhs
CbUf, 49
mnumrhs
CbUf, 49
MOD2FUNCTION
ftypes.h, 1014
mode
dbase, 53
TaBlEs, 261
MODFUNCTION
ftypes.h, 1014
modinverses
C_const, 27
MODLOCAL
ftypes.h, 1059
MODMAX
ftypes.h, 1059
MODMIN
ftypes.h, 1059
modmode
C_const, 41
modn
poly, 192
MODNONE
ftypes.h, 1059
MODNUM, 129
a, 129
m, 129
na, 129
nm, 129
modnum
POLYMOD, 193
MoDoPtDoLlArS, 130
number, 130
type, 130
ModOptdollars
variable.h, 2410
ModOptDolList
C_const, 21
modp
poly, 191

modpowers
C_const, 24
MODSUM
ftypes.h, 1059
module.c, 1197, 1201
CoModOption, 1198
CoModuleOption, 1198
DoExecStatement, 1200
DoinParallel, 1200
DoModDollar, 1200
DoModLocal, 1200
DoModMax, 1199
DoModMin, 1199
DoModSum, 1199
DoNoParallel, 1199
DonotinParallel, 1200
DoParallel, 1199
DoPipeStatement, 1200
DoPolyfun, 1199
DoPolyratfun, 1199
DoProcessBucket, 1200
ExecModule, 1198
ExecStore, 1198
FullCleanUp, 1198
ModuleInstruction, 1198
SetSpecialMode, 1198
MODULEINSTACK
ftypes.h, 992
ModuleInstruction
declare.h, 669
module.c, 1198
Modulus
declare.h, 626
reken.c, 1797
monomial_larger, 130
monomial_larger, 131
operator(), 131
MoveDummies
declare.h, 626
reshuf.c, 1845
mparallelflag
C_const, 36
mpi.c, 1210, 1226
MPI_ERRCODE_CHECK, 1211
PF_Bcast, 1215
PF_Broadcast, 1221
PF_IRecvRbuf, 1214
PF_ISendSbuf, 1213
PF_LibInit, 1212
PF_LibTerminate, 1213
PF_LongMultiBroadcast, 1225
PF_LongMultiPackImpl, 1223
PF_LongMultiUnpackImpl, 1224
PF_LongSinglePack, 1221
PF_LongSingleReceive, 1223
PF_LongSingleSend, 1222
PF_LongSingleUnpack, 1222
PF_maxDollarChunkSize, 1226
PF_Pack, 1218
PF_PACKSIZE, 1211
PF_PackString, 1219
PF_PrepareLongMultiPack, 1223
PF_PrepareLongSinglePack, 1221
PF_PreparePack, 1217
PF_PrintPackBuf, 1217
PF_Probe, 1213
PF_RawProbe, 1217
PF_RawRecv, 1216
PF_RawSend, 1216
PF_RealTime, 1212
PF_Receive, 1220
PF_RecvWbuf, 1214
PF_Send, 1220
PF_Unpack, 1218
PF_UnpackString, 1219
PF_WaitRbuf, 1215
MPI_Barrier
mpidbg.h, 1248
MPI_Bcast
mpidbg.h, 1248
MPI_Bsend
mpidbg.h, 1245
MPI_Cancel
mpidbg.h, 1248
MPI_ERRCODE_CHECK
mpi.c, 1211
MPI_Ibsend
mpidbg.h, 1246
MPI_Iprobe
mpidbg.h, 1248
MPI_Irecv
mpidbg.h, 1246
MPI_Irsend
mpidbg.h, 1246
MPI_Isend
mpidbg.h, 1246
MPI_Issend
mpidbg.h, 1246
MPI_Probe
mpidbg.h, 1248
MPI_Recv
mpidbg.h, 1245
MPI_Rsend
mpidbg.h, 1246
MPI_Send
mpidbg.h, 1245
MPI_Ssend
mpidbg.h, 1245
MPI_Test
mpidbg.h, 1247
MPI_Test_cancelled
mpidbg.h, 1248
MPI_Testall
mpidbg.h, 1247
MPI_Testany
mpidbg.h, 1247

MPI_Testsome
mpidbg.h, 1247
MPI_Wait
mpidbg.h, 1246
MPI_Waitall
mpidbg.h, 1247
MPI_Waitany
mpidbg.h, 1247
MPI_Waitsome
mpidbg.h, 1247
mpidbg.h, 1244, 1249
MPI_Barrier, 1248
MPI_Bcast, 1248
MPI_Bsend, 1245
MPI_Cancel, 1248
MPI_Ibsend, 1246
MPI_Iprobe, 1248
MPI_Irecv, 1246
MPI_Irsend, 1246
MPI_Isend, 1246
MPI_Issend, 1246
MPI_Probe, 1248
MPI_Recv, 1245
MPI_Rsend, 1246
MPI_Send, 1245
MPI_Ssend, 1245
MPI_Test, 1247
MPI_Test_cancelled, 1248
MPI_Testall, 1247
MPI_Testany, 1247
MPI_Testsome, 1247
MPI_Wait, 1246
MPI_Waitall, 1247
MPI_Waitany, 1247
MPI_Waitsome, 1247
MPIDBG_EXTARG, 1245
MPIDBG_Out, 1245
MPIDBG_RANK, 1245
MPIDBG_EXTARG
mpidbg.h, 1245
MPIDBG_Out
mpidbg.h, 1245
MPIDBG_RANK
mpidbg.h, 1245
mpower
O_const, 164
mProcessBucketSize
C_const, 30
mTraceDum
M_const, 122
mul
COST, 51
MULfunc
declare.h, 764
ratio.c, 1743
MULFUNCTION
ftypes.h, 1022
MulLong
declare.h, 627
reken.c, 1792
Mully
declare.h, 627
reken.c, 1789
mulpat
T_const, 250
MULPOS
declare.h, 585
MulRat
declare.h, 627
reken.c, 1790
MultDo
declare.h, 627
reshuf.c, 1846
MultiBracketBuf
C_const, 28
MultiBracketLevels
C_const, 38
MULTIINDENTSPACE
fsizes.h, 972
MULTIPLEOF
ftypes.h, 1052
MULTIPLYARG
ftypes.h, 1064
MultiplyToLine
sch.c, 1888
MultiplyWithTerm
declare.h, 763
ratio.c, 1741
MultiRun
M_const, 118
MultiThreaded
S_const, 210
MUNLOCK
declare.h, 593
MUSTCLEANPRF
ftypes.h, 1037
MustTestTable
C_const, 41
mytime.h, 1257

N
AllGlobals, 6
n
DiStRiBuTe, 59
PeRmUtE, 185
PeRmUtEp, 186
n1
DiStRiBuTe, 59
n2
DiStRiBuTe, 59
N_const, 131
argaddress, 135
arglist, 140
arglistsize, 143
cmod, 140
compressSize, 144
compressSpace, 140

cTerm, 135
currentTerm, 140
deferskipped, 141
DisOrderFlag, 145
DumFound, 136
DumFunPlace, 136
dummyrenumlist, 137
DumPlace, 136
EndNest, 134
ErrorInDollar, 142
ExpectedSign, 147
filenum, 144
findPattern, 137
findTerm, 137
ForFindOnly, 137
Frozen, 134
FullProto, 134
funargs, 137
funinds, 138
FunInfo, 139
funisize, 143
funlocs, 137
FunScratch, 139
FunSorts, 139
idfunctionflag, 147
IndDum, 147
InScratch, 141
InScratch1, 141
intracestack, 143
listinprint, 139
MaskPointer, 137
MaxFunSorts, 143
MaxRenumScratch, 144
nargs, 142
ncmod, 147
ninterms, 141
nogroundlevel, 144
NoScrat2, 138
numfargs, 142
numflocs, 142
NumFound, 145
numfuninfo, 143
NumFunSorts, 143
numlistinprint, 146
numoffuns, 142
NumTotWildArgs, 141
numtracesctack, 143
NumWild, 145
OldPosIn, 134
OldPosOut, 134
oldtype, 144
oldvalue, 145
patstop, 135
patternbuffer, 138
patternbuffersize, 146
PoinScratch, 138
poly_num_vars, 147
poly_vars, 140
poly_vars_type, 147
PolyFunTodo, 146
PolyNormFlag, 146
polysortflag, 144
RenumScratch, 139
RepFunList, 135
RepFunNum, 145
ReplaceScrat, 138
RepPoint, 135
RSsize, 143
selecttermundo, 138
SHcombi, 140
SHcombisize, 141
SHvar, 140
SignCheck, 147
sizeselecttermundo, 146
SoScratC, 139
SplitScratch, 139
SplitScratch1, 139
SplitScratchSize, 141
SplitScratchSize1, 141
subsubveto, 144
TeInFun, 145
terfirstcomm, 136
termbuffer, 138
terstart, 136
terstop, 135
TeSuOut, 146
theposition, 134
tlistbuf, 140
tlistsize, 144
tohunt, 142
tracestack, 138
tryterm, 148
UsedFunction, 137
UsedIndex, 136
UsedOtherFind, 142
UsedSymbol, 136
UsedVector, 136
UseFindOnly, 142
WildArgs, 146
WildDirt, 145
WildEat, 146
WildReserve, 145
WildStop, 135
WildValue, 135
na
MODNUM, 129
name
ChAnNeL, 51
dbase, 54
DICTIONARY, 55
DoLlArS, 61
DoLoOp, 63
DuBiOuS, 66
ExPrEsSiOn, 69
FiLe, 74
fixedset, 82

FuNcTiOn, 86
InDeX, 90
InDeXeNtRy, 95
KEYWORD, 100
NaMeNode, 149
pReVaR, 195
PROCEDURE, 197
SeTs, 211
SpecTatoR, 226
StreaM, 233
SyMbOl, 237
TaBlEbAsE, 253
VeCtOr, 272
nameblock, 148
names, 148
position, 148
previousblock, 148
namebuffer
NaMeTree, 151
NameConflict
declare.h, 660
names.c, 1267
namefill
NaMeTree, 152
NAMENODE
structs.h, 2121
NaMeNode, 149
balance, 150
left, 149
name, 149
number, 150
parent, 149
right, 150
type, 150
namenode
NaMeTree, 151
NAMENOTFOUND
ftypes.h, 1055
nameofexpr
OPTIMIZERESULT, 172
names
nameblock, 148
names.c, 1257, 1268
AddDollar, 1266
AddDubious, 1266
AddExpression, 1267
AddFunction, 1263
AddIndex, 1262
AddName, 1259
AddSet, 1265
AddSymbol, 1262
AddVector, 1263
AddWildcardName, 1261
ClearWildcardNames, 1261
CoAuto, 1266
CoCFunction, 1264
CoCommuteInSet, 1263
CoCTable, 1265
CoCTensor, 1264
CoDimension, 1263
CoFunction, 1263
CoIndex, 1262
CompactifyTree, 1260
CoNFunction, 1264
CoNTable, 1265
CoNTensor, 1264
CopyTree, 1261
CoSet, 1265
CoSymbol, 1262
CoTable, 1264
CoVector, 1263
DoDimension, 1262
DoElements, 1265
DoTable, 1264
DoTempSet, 1266
DumpNode, 1260
DumpTree, 1260
EmptyTable, 1265
EntVar, 1260
FreeNameTree, 1261
GetAutoName, 1259
GetDollar, 1260
GetLabel, 1267
GetLastExprName, 1259
GetName, 1259
GetNode, 1259
GetOName, 1259
GetVar, 1260
GetWildcardName, 1262
Globalize, 1267
LinkTree, 1261
MakeDubious, 1266
MakeNameTree, 1261
NameConflict, 1267
RemoveDollars, 1267
ReplaceDollar, 1266
ResetVariables, 1267
TestName, 1268
NamesFlag
C_const, 34
namesize
ExPrEsSiOn, 70
FuNcTiOn, 87
InDeX, 91
NaMeTree, 152
SeTs, 212
SyMbOl, 238
VeCtOr, 272
NAMETABLESIZE
minos.h, 1193
NAMETREE
structs.h, 2121
NaMeTree, 151
clearnamefill, 153
clearnodefill, 153
globalnamefill, 153

globalnodefill, 153
headnode, 153
namebuffer, 151
namefill, 152
namenode, 151
namesize, 152
nodefill, 152
nodesize, 152
oldnamefill, 152
oldnodefill, 152
nargs
N_const, 142
PaRtI, 183
pReVaR, 195
nBer
T_const, 251
nblocks
dbase, 54
ncmod
C_const, 41
N_const, 147
NCOPY
declare.h, 576
NCOPYB
declare.h, 577
NCOPYI
declare.h, 576
NCOPYI32
declare.h, 577
NDEBUG
form3.h, 948
NeedNumber
declare.h, 577
NEG0_
ftypes.h, 1043
NEG_
ftypes.h, 1043
Nest
T_const, 241
NESTING
structs.h, 2124
NeStInG, 154
argsize, 154
funsize, 154
termsize, 154
NestingChecksum
declare.h, 590
NestPoin
T_const, 241
NestStop
T_const, 241
newbracketinfo
ExPrEsSiOn, 68
NEWCODE
reshuf.c, 1845
NewDbase
declare.h, 742
minos.c, 1171
NEWFACTARG
ftypes.h, 1065
newhandle
HANDLERS, 89
newlogonly
HANDLERS, 89
NEWLYDEFINEDEXPRESSION
ftypes.h, 1042
newmaxtermsize
sOrT, 224
NEWORDER
argument.c, 276
NewSort
declare.h, 627
sort.c, 1950
NEWSPLITMERGE
sort.c, 1949
NEWSTATEMENT
ftypes.h, 992
NEWTRICK
reken.c, 1788
next
BrAcKeTiNdEx, 11
FiLeInDeX, 79
longMultiStruct, 103
StOrEcAcHe, 228
NEXTARG
declare.h, 581
nextchar
StreaM, 235
NextFileIndex
declare.h, 596
store.c, 2054
NextPrime
declare.h, 747
reken.c, 1798
nfac
T_const, 251
nfactors
DoLlArS, 61
nfun
PaRtI, 184
nfunctions
InDeXeNtRy, 95
nhalfmod
C_const, 41
nincoef
SHvariables, 215
nindices
InDeXeNtRy, 94
ninterms
N_const, 141
sOrT, 222
nm
MODNUM, 129
nmin4
InDeX, 91
NMIN4SHIFT

ftypes.h, 1009
NOAUTO
ftypes.h, 997
NOBRACKETACTIVE
execute.c, 862
NOCOMPRESS
minos.h, 1194
NoCompress
C_const, 37
R_const, 202
NODE
parallel.c, 1421
NoDe, 155
left, 155
lloser, 155
lsrc, 156
rght, 155
rloser, 155
rsrc, 156
node
DoLlArS, 61
DuBiOuS, 66
ExPrEsSiOn, 70
FuNcTiOn, 87
InDeX, 91
SeTs, 212
SyMbOl, 238
VeCtOr, 272
nodefill
NaMeTree, 152
nodesize
NaMeTree, 152
NODOUBLEMASK
ftypes.h, 1007
NOFUNPOWERS
ftypes.h, 1057
nogroundlevel
N_const, 144
NOINDEX
ftypes.h, 1045
NOINVERSES
ftypes.h, 1060
NOLYNDON
ftypes.h, 1064
NOPARALLEL_CONVPOLY
ftypes.h, 990
NOPARALLEL_DOLLAR
ftypes.h, 990
NOPARALLEL_NPROC
ftypes.h, 991
NOPARALLEL_RHS
ftypes.h, 990
NOPARALLEL_SPECTATOR
ftypes.h, 990
NOPARALLEL_TBLDOLLAR
ftypes.h, 991
NOPARALLEL_USER
ftypes.h, 991
NOQUADMASK
ftypes.h, 1007
normal.c, 1304, 1308
BracketNormalize, 1307
Commute, 1305
CompareFunctions, 1305
DetCommu, 1306
DoDelta, 1306
DoesCommu, 1306
DoRevert, 1306
DoTheta, 1306
DropCoefficient, 1307
DropSymbols, 1307
EXCH, 1305
EXCHN, 1305
ExtraSymbol, 1305
MAXNUMBEROFNONCOMTERMS, 1305
Normalize, 1305
SymbolNormalize, 1307
TestFunFlag, 1307
TreatPolyRatFun, 1306
NORMALFORMAT
ftypes.h, 1007
Normalize
declare.h, 628
normal.c, 1305
NORMALIZEFLAG
ftypes.h, 1051
NormalModulus
declare.h, 634
reken.c, 1793
NormPolyTerm
declare.h, 758
notation.c, 1369
NORMSIZE
fsizes.h, 965
NoScrat2
N_const, 138
NOSHELL
pre.c, 1570
NoShowInput
C_const, 31
DoLoOp, 64
NOSPACEFORMAT
ftypes.h, 1007
NoSpacesInNumbers
O_const, 162
notation.c, 1368, 1371
ConvertFromPoly, 1369
ConvertToPoly, 1369
ExtraSymFun, 1370
FindLocalSubterm, 1370
FindSubexpression, 1370
FindSubterm, 1370
LocalConvertToPoly, 1369
NormPolyTerm, 1369
PrintExtraSymbol, 1370
PrintSubtermList, 1370

PruneExtraSymbols, 1371
NOTEQUAL
ftypes.h, 1053
NOTRICK
ftypes.h, 1008
NOTSTARTPOS
declare.h, 586
NOUNPACK
ftypes.h, 1061
nPacks
longMultiStruct, 103
NPAIR1
fsizes.h, 972
NPAIR2
fsizes.h, 972
npowmod
C_const, 41
NSIG
portsignals.h, 1563
nsymbols
InDeXeNtRy, 94
nt3
TrAcEs, 265
nt4
TrAcEs, 265
num
LIST, 101
TrAcEn, 262
TrAcEs, 268
NUMARG
ftypes.h, 1062
numargs
FUN_INFO, 83
PaRtI, 184
NUMARGSFUN
ftypes.h, 1014
number
FiLeInDeX, 79
FuNcTiOn, 86
InDeX, 90
MoDoPtDoLlArS, 130
NaMeNode, 150
SyMbOl, 237
VeCtOr, 272
NUMBEREXTRAWORDS
tools.c, 2292
NumberFree
declare.h, 590
NumberMalloc
declare.h, 589
NumberMallocAddMemory
declare.h, 753
tools.c, 2303
NumberMemHeap
T_const, 244
NumberMemMax
T_const, 247
NUMBERMEMSTARTNUM
tools.c, 2292
NumberMemTop
T_const, 247
numberofindexblocks
iniinfo, 96
numberofnamesblocks
iniinfo, 97
numberoftables
iniinfo, 96
numbers
DICTIONARY, 56
numblocks
FiLe, 75
numbufs
PF_BUFFER, 188
numclear
LIST, 102
numcommute
comtool.c, 542
declare.h, 723
NumCopy
declare.h, 682
tools.c, 2300
NumDictionaries
O_const, 163
NumDollars
variable.h, 2408
numdollars
INSIDEINFO, 98
NumDoLoops
variable.h, 2405
NumDubious
variable.h, 2408
numdum
CbUf, 48
numdummies
DoLlArS, 61
ExPrEsSiOn, 70
TaBlEs, 261
numelements
DICTIONARY, 56
NUMERATORSYMBOL
ftypes.h, 1023
NUMERICALLOOP
ftypes.h, 994
NUMERICALVALUE
setfile.c, 1924
NumExpressions
variable.h, 2409
NUMFACS
fsizes.h, 966
NUMFACTORS
ftypes.h, 1021
numfactors
ExPrEsSiOn, 70
numfargs
N_const, 142
NUMFIXED

fsizes.h, 965
NumFixedFunctions
M_const, 114
NumFixedSets
M_const, 114
numflocs
N_const, 142
NumFound
N_const, 145
NumFunctions
variable.h, 2406
numfuninfo
N_const, 143
numfunnies
FUN_INFO, 83
NumFunSorts
N_const, 143
numglobal
LIST, 101
NumInBrack
O_const, 162
numind
TaBlEs, 258
NumIndices
variable.h, 2406
numinfilelist
tools.c, 2308
NumLabels
C_const, 32
numlhs
CbUf, 49
numlistinprint
N_const, 146
NumListSymbols
T_const, 248
NumMem
T_const, 245
NumModOptdollars
variable.h, 2410
NUMOBJECTS
minos.h, 1192
numoffuns
N_const, 142
NumOldNumFactors
S_const, 210
NumOldOnFile
S_const, 210
NumOutputChannels
variable.h, 2408
numpart
PaRtI, 184
numpoly
T_const, 248
NumPotModdollars
variable.h, 2410
NumPre
variable.h, 2405
NumPreSwitchStrings
P_const, 177
NumPreTypes
DoLoOp, 65
P_const, 178
NumProcedures
variable.h, 2404
numrbufs
ParallelVars, 182
numrhs
CbUf, 49
numsbufs
ParallelVars, 182
NumSetElements
variable.h, 2407
NumSets
variable.h, 2407
NUMSPECSETS
ftypes.h, 1047
NumSpectatorFiles
M_const, 119
NUMSTORECACHES
fsizes.h, 971
NumStoreCaches
M_const, 126
NumStreams
C_const, 33
numsym
POLYMOD, 192
NumSymbols
variable.h, 2406
NumTableBases
variable.h, 2407
NUMTABLEENTRIES
fsizes.h, 967
numtables
TaBlEbAsE, 253
numtasks
ParallelVars, 181
numtemp
LIST, 102
NumTerms
CbUf, 48
NUMTERMSFUN
ftypes.h, 1017
NUMTOIND
ftypes.h, 1036
NUMTONUM
ftypes.h, 1036
NumToStr
declare.h, 680
tools.c, 2299
NUMTOSUB
ftypes.h, 1036
NUMTOSYM
ftypes.h, 1036
NumTotWildArgs
N_const, 141
numtracesctack

N_const, 143
numtree
CbUf, 50
TaBlEs, 259
NumVectors
variable.h, 2406
NumWild
N_const, 145
NumWildcardNames
C_const, 32
numwildcards
FUN_INFO, 84
numxsymbol
variable.h, 2409
nvectors
InDeXeNtRy, 95
NwildC
C_const, 42

O
AllGlobals, 7
O_BACKWARD
ftypes.h, 1067
O_const, 156
BlockSpaces, 162
bracket, 159
bufferedInd, 165
CheckPower, 161
CurBufWrt, 160
CurDictFunWithArgs, 163
CurDictInDollars, 164
CurDictNotInFunctions, 164
CurDictNumbers, 163
CurDictNumberWarning, 163
CurDictSpecials, 163
CurDictVariables, 163
CurrentDictionary, 163
Dictionaries, 161
DollarInOutBuffer, 162
DollarOutBuffer, 159
DollarOutSizeBuffer, 162
DoubleFlag, 165
ErrorBlock, 166
FactorMode, 166
FactorNum, 166
FlipLONG, 160
FlipPOINTER, 160
FlipPOS, 160
FlipWORD, 160
FortDotChar, 166
FortFirst, 165
gNumDictionaries, 164
IndentSpace, 165
InFbrack, 165
inscheme, 161
IsBracket, 165
mpower, 164
NoSpacesInNumbers, 162
NumDictionaries, 163
NumInBrack, 162
OptimizationLevel, 166
Optimize, 162
OptimizeResult, 158
OutFill, 159
OutInBuffer, 162
OutputLine, 158
OutSkip, 165
OutStop, 159
powerFlag, 164
PrintType, 165
RenumberVec, 161
ResizeData, 160
resizeFlag, 164
ResizeLONG, 161
ResizeNCWORD, 160
ResizePOINTER, 161
ResizePOS, 161
ResizeWORD, 160
SaveData, 158
SaveHeader, 158
schemenum, 164
SizeDictionaries, 163
tabstring, 159
tensorList, 161
termbuf, 159
transFlag, 164
wlen, 162
wpoin, 159
wpos, 159
O_CSE
ftypes.h, 1066
O_CSEGREEDY
ftypes.h, 1066
O_FORWARD
ftypes.h, 1067
O_FORWARDANDBACKWARD
ftypes.h, 1067
O_FORWARDORBACKWARD
ftypes.h, 1067
O_GREEDY
ftypes.h, 1066
O_MCTS
ftypes.h, 1066
O_NONE
ftypes.h, 1066
O_OCCURRENCE
ftypes.h, 1066
O_SIMULATED_ANNEALING
ftypes.h, 1066
obj1
DiStRiBuTe, 58
obj2
DiStRiBuTe, 58
objects, 167
date, 167
element, 168
indexblock, 92

PeRmUtE, 185
PeRmUtEp, 186
position, 167
size, 167
spare1, 168
spare2, 168
spare3, 168
tablenumber, 167
uncompressed, 167
ODD_
ftypes.h, 1044
OffsetIndex
M_const, 122
OffsetVector
M_const, 122
oldcbuf
INSIDEINFO, 98
OldChildTime
M_const, 112
oldcnumlhs
INSIDEINFO, 99
oldcompiletype
INSIDEINFO, 98
olddelay
StreaM, 235
OLDFACTARG
ftypes.h, 1065
OldFactArgFlag
C_const, 38
OldGCDflag
C_const, 38
oldhandle
HANDLERS, 89
oldlogonly
HANDLERS, 89
OldMilliTime
M_const, 112
oldnamefill
NaMeTree, 152
oldnodefill
NaMeTree, 152
oldnoshowinput
StreaM, 235
oldnumextrasymbols
M_const, 109
OldNumFactors
S_const, 210
oldnumpotmoddollars
INSIDEINFO, 98
OldOnFile
S_const, 209
OldOrderFlag
M_const, 124
oldparallelflag
INSIDEINFO, 98
OldParallelStats
C_const, 35
OldPosIn
N_const, 134
OldPosOut
N_const, 134
oldprinttype
HANDLERS, 89
oldrbuf
INSIDEINFO, 99
OldSecTime
M_const, 112
oldsilent
HANDLERS, 89
OLDSTATEMENT
ftypes.h, 992
OldTime
R_const, 201
oldtype
N_const, 144
oldvalue
N_const, 145
Oldvflags
S_const, 210
one_byte
structs.h, 2123
ONEEXPRESSION
ftypes.h, 995
onerhs
M_const, 127
onfile
ExPrEsSiOn, 68
ONLYFUNCTIONS
ftypes.h, 1067
ONOFFVALUE
setfile.c, 1924
OpenAddFile
declare.h, 661
tools.c, 2294
OpenBracketIndex
declare.h, 736
index.c, 1126
OpenDbase
declare.h, 741
minos.c, 1171
OpenDictionary
P_const, 179
openExternalChannel
declare.h, 745
extcmd.c, 896
OpenFile
declare.h, 660
tools.c, 2294
OpenInput
startup.c, 2025
OpenStream
declare.h, 654
tools.c, 2293
OpenTemp
store.c, 2050
opera.c, 1385, 1388

Chisholm, 1387
EpfCon, 1385
EpfFind, 1385
EpfGen, 1386
TenVec, 1388
TenVecFind, 1388
Trace4, 1386
Trace4Gen, 1386
Trace4no, 1386
TraceFind, 1387
TraceN, 1387
TraceNgen, 1387
TraceNno, 1387
Traces, 1387
Trick, 1386
OperaFind
FixedGlobals, 80
Operation
FixedGlobals, 80
operator()
monomial_larger, 131
operator[]
poly, 191
OPTHEAD
ftypes.h, 1067
OptimizationLevel
O_const, 166
OPTIMIZE, 169
debugflags, 171
fval, 169
greedymaxperc, 171
greedyminnum, 170
greedytimelimit, 170
horner, 169
hornerdirection, 170
ival, 169
mctsdecaymode, 171
mctsnumexpand, 170
mctsnumkeep, 170
mctsnumrepeat, 170
mctstimelimit, 170
method, 170
printstats, 171
salter, 171
schemeflags, 171
Optimize
O_const, 162
OPTIMIZERESULT, 172
code, 172
codesize, 172
exprnr, 172
maxvar, 173
minvar, 172
nameofexpr, 172
OptimizeResult
O_const, 158
optimtimes
T_const, 248
option
SHvariables, 215
OPTION0
compiler.c, 499
OPTION1
compiler.c, 499
OPTION2
compiler.c, 499
ORCOND
ftypes.h, 1054
Ordering
M_const, 123
origin
C_const, 39
out
DiStRiBuTe, 58
OutBuffer
M_const, 109
OutBufSize
M_const, 113
outfile
R_const, 200
OutFill
O_const, 159
outfun
SHvariables, 214
OutInBuffer
O_const, 162
OutNumberType
C_const, 43
OutputLine
O_const, 158
OutputMode
C_const, 42
outputmode
sOrT, 224
OUTPUTONLY
minos.h, 1194
OutputSpaces
C_const, 42
outsidefun
C_const, 34
OutSkip
O_const, 165
OutStop
O_const, 159
outterm
SHvariables, 214
outtohide
R_const, 202

P
AllGlobals, 7
p
DoLoOp, 63
PROCEDURE, 197
p1
PoSiTiOn, 193
P_const, 173

AllowDelay, 178
cComChar, 180
ComChar, 179
cprocedureExtension, 176
DebugFlag, 179
DelayPrevar, 178
DollarList, 174
eat, 178
gNumPre, 179
iBufError, 177
InOutBuf, 176
inside, 175
lhdollarerror, 178
LoopList, 175
MaxPreAssignLevel, 179
MaxPreIfLevel, 178
MaxPreTypes, 177
NumPreSwitchStrings, 177
NumPreTypes, 178
OpenDictionary, 179
PreAssignFlag, 177
PreAssignLevel, 179
PreAssignStack, 176
PreContinuation, 177
PreDebug, 179
preError, 179
preFill, 175
PreIfLevel, 178
PreIfStack, 176
PreInsideLevel, 178
PreOut, 177
PreproFlag, 177
preStart, 175
preStop, 175
PreSwitchLevel, 177
PreSwitchModes, 176
PreSwitchStrings, 175
PreTypes, 176
PreVarList, 174
procedureExtension, 175
ProcList, 175
pSize, 176
StopWatchZero, 176
P_term
fwin.h, 1110
unix.h, 2397
Pack
declare.h, 629
reken.c, 1789
PACK_LONG
parallel.c, 1419
packpos
longMultiStruct, 103
PADDUMMY
form3.h, 950
PADINT
form3.h, 951
PADLONG
form3.h, 951
PADPOINTER
form3.h, 950
PADPOSITION
form3.h, 950
PADWORD
form3.h, 951
parallel
ParallelVars, 182
parallel.c, 1417, 1442
_CHECK, 1420
__CHECK, 1420
CHECK, 1420
DBGOUT, 1420
DBGOUT_NINTERMS, 1421
NODE, 1421
PACK_LONG, 1419
PF, 1442
PF_BroadcastBuffer, 1431
PF_BroadcastCBuf, 1436
PF_BroadcastExpFlags, 1437
PF_BroadcastExpr, 1437
PF_BroadcastModifiedDollars, 1434
PF_BroadcastNumber, 1430
PF_BroadcastPreDollar, 1432
PF_BroadcastRedefinedPreVars, 1435
PF_BroadcastRHS, 1438
PF_BroadcastString, 1432
PF_CollectModifiedDollars, 1433
PF_Deferred, 1426
PF_EndSort, 1426
PF_FlushStdOutBuffer, 1441
PF_FreeErrorMessageBuffers, 1442
PF_GetSlaveTimes, 1430
PF_Init, 1428
PF_InParallelProcessor, 1438
PF_IRecvRbuf, 1423
PF_LibInit, 1422
PF_LibTerminate, 1422
PF_MLock, 1440
PF_MUnlock, 1440
PF_Probe, 1423
PF_Processor, 1427
PF_RawProbe, 1425
PF_RawRecv, 1425
PF_RawSend, 1424
PF_RealTime, 1421
PF_RecvFile, 1439
PF_RecvWbuf, 1423
PF_RestoreInsideInfo, 1436
PF_SendFile, 1439
PF_SNDFILEBUFSIZE, 1421
PF_STATS_SIZE, 1421
PF_StoreInsideInfo, 1436
PF_Terminate, 1429
PF_WaitRbuf, 1424
PF_WriteFileToFile, 1441
PRINTFBUF, 1419

recvBuffer, 1421
SWAP, 1419
UNPACK_LONG, 1419
parallel.h, 1491, 1523
GNUC_PREREQ, 1495
indices, 1495
MASTER, 1493
PF, 1523
PF_ANY_MSGTAG, 1496
PF_ANY_SOURCE, 1495
PF_Bcast, 1497
PF_Broadcast, 1501
PF_BroadcastBuffer, 1512
PF_BroadcastCBuf, 1516
PF_BroadcastExpFlags, 1518
PF_BroadcastExpr, 1519
PF_BroadcastModifiedDollars, 1515
PF_BroadcastNumber, 1511
PF_BroadcastPreDollar, 1513
PF_BroadcastRedefinedPreVars, 1516
PF_BroadcastRHS, 1519
PF_BroadcastString, 1512
PF_BUFFER_MSGTAG, 1494
PF_BYTE, 1496
PF_CollectModifiedDollars, 1514
PF_COMM, 1496
PF_DATA_MSGTAG, 1494
PF_Deferred, 1507
PF_DOLLAR_MSGTAG, 1493
PF_EMPTY_MSGTAG, 1494
PF_ENDBUFFER_MSGTAG, 1494
PF_EndSort, 1506
PF_ENDSORT_MSGTAG, 1493
PF_FlushStdOutBuffer, 1523
PF_GetSlaveTimes, 1511
PF_Init, 1509
PF_InParallelProcessor, 1520
PF_INT, 1496
PF_ISendSbuf, 1496
PF_LOG_MSGTAG, 1494
PF_LongMultiBroadcast, 1506
PF_LongMultiPack, 1496
PF_LongMultiPackImpl, 1504
PF_LongMultiUnpack, 1496
PF_LongMultiUnpackImpl, 1505
PF_LongSinglePack, 1502
PF_LongSingleReceive, 1503
PF_LongSingleSend, 1503
PF_LongSingleUnpack, 1502
PF_maxDollarChunkSize, 1523
PF_MISC_MSGTAG, 1495
PF_MLock, 1521
PF_MUnlock, 1522
PF_OPT_COLLECT_MSGTAG, 1495
PF_OPT_HORNER_MSGTAG, 1495
PF_OPT_MCTS_MSGTAG, 1495
PF_Pack, 1498
PF_PackString, 1499
PF_PrepareLongMultiPack, 1504
PF_PrepareLongSinglePack, 1502
PF_PreparePack, 1498
PF_Processor, 1508
PF_RawRecv, 1498
PF_RawSend, 1497
PF_READY_MSGTAG, 1494
PF_Receive, 1501
PF_RecvFile, 1521
PF_RESET, 1493
PF_RestoreInsideInfo, 1519
PF_Send, 1500
PF_SendFile, 1520
PF_STDOUT_MSGTAG, 1494
PF_StoreInsideInfo, 1518
PF_TERM_MSGTAG, 1493
PF_Terminate, 1510
PF_TIME, 1493
PF_Unpack, 1499
PF_UnpackString, 1500
PF_WriteFileToFile, 1522
PARALLELFLAG
ftypes.h, 991
parallelflag
C_const, 36
ParallelVars, 180
exprbufsize, 182
exprtodo, 182
log, 182
me, 181
mkSlaveInfile, 182
numrbufs, 182
numsbufs, 182
numtasks, 181
parallel, 182
rbufs, 181
rhsInParallel, 182
sbuf, 181
slavebuf, 181
parameter
SETUPPARAMETERS, 213
parent
NaMeNode, 149
tree, 269
ParenthesesTest
compiler.c, 499
declare.h, 718
ParseNumber
declare.h, 576
ParseSign
declare.h, 576
ParseSignedNumber
declare.h, 576
PARTEST
ftypes.h, 997
PARTI
structs.h, 2124
PaRtI, 183

args, 183
nargs, 183
nfun, 184
numargs, 184
numpart, 184
psize, 183
where, 184
PARTITIONS
ftypes.h, 1022
partitions
T_const, 245
partodoflag
C_const, 37
PasteFile
declare.h, 629
proces.c, 1670
PasteTerm
declare.h, 596
proces.c, 1670
Patches
sOrT, 220
Path
M_const, 109
PATHSEPARATOR
fwin.h, 1110
unix.h, 2398
PATHVALUE
setfile.c, 1924
patstop
N_const, 135
pattern
TaBlEs, 256
pattern.c, 1527, 1530
FindAll, 1529
PutInBuffers, 1528
SubsInAll, 1529
Substitute, 1529
TakeIDfunction, 1530
TestMatch, 1528
TestSelect, 1529
TransferBuffer, 1530
patternbuffer
N_const, 138
patternbuffersize
N_const, 146
PATTERNFUNCTION
ftypes.h, 1016
pBer
T_const, 244
pdel
TrAcEs, 266
pepf
TrAcEs, 266
PERM
structs.h, 2124
perm
TrAcEn, 262
TrAcEs, 264
PERMP
structs.h, 2124
PERMUTATIONS
ftypes.h, 1022
PeRmUtE, 184
cycle, 185
n, 185
objects, 185
sign, 185
Permute
declare.h, 630
symmetr.c, 2150
PERMUTEARG
ftypes.h, 1063
PeRmUtEp, 185
cycle, 186
n, 186
objects, 186
sign, 186
PermuteP
declare.h, 630
symmetr.c, 2150
PF
parallel.c, 1442
parallel.h, 1523
PF_ANY_MSGTAG
parallel.h, 1496
PF_ANY_SOURCE
parallel.h, 1495
PF_Bcast
mpi.c, 1215
parallel.h, 1497
PF_Broadcast
mpi.c, 1221
parallel.h, 1501
PF_BroadcastBuffer
parallel.c, 1431
parallel.h, 1512
PF_BroadcastCBuf
parallel.c, 1436
parallel.h, 1516
PF_BroadcastExpFlags
parallel.c, 1437
parallel.h, 1518
PF_BroadcastExpr
parallel.c, 1437
parallel.h, 1519
PF_BroadcastModifiedDollars
parallel.c, 1434
parallel.h, 1515
PF_BroadcastNumber
parallel.c, 1430
parallel.h, 1511
PF_BroadcastPreDollar
parallel.c, 1432
parallel.h, 1513
PF_BroadcastRedefinedPreVars
parallel.c, 1435

parallel.h, 1516
PF_BroadcastRHS
parallel.c, 1438
parallel.h, 1519
PF_BroadcastString
parallel.c, 1432
parallel.h, 1512
PF_BUFFER, 186
active, 188
buff, 187
fill, 187
from, 188
full, 187
index, 188
numbufs, 188
request, 188
retstat, 188
status, 187
stop, 187
tag, 188
type, 188
PF_BUFFER_MSGTAG
parallel.h, 1494
PF_BYTE
parallel.h, 1496
PF_CollectModifiedDollars
parallel.c, 1433
parallel.h, 1514
PF_COMM
parallel.h, 1496
PF_DATA_MSGTAG
parallel.h, 1494
PF_Deferred
parallel.c, 1426
parallel.h, 1507
PF_DOLLAR_MSGTAG
parallel.h, 1493
PF_EMPTY_MSGTAG
parallel.h, 1494
PF_ENDBUFFER_MSGTAG
parallel.h, 1494
PF_EndSort
parallel.c, 1426
parallel.h, 1506
PF_ENDSORT_MSGTAG
parallel.h, 1493
PF_FlushStdOutBuffer
parallel.c, 1441
parallel.h, 1523
PF_FreeErrorMessageBuffers
parallel.c, 1442
PF_GetSlaveTimes
parallel.c, 1430
parallel.h, 1511
PF_Init
parallel.c, 1428
parallel.h, 1509
PF_InParallelProcessor
parallel.c, 1438
parallel.h, 1520
PF_INT
parallel.h, 1496
PF_IRecvRbuf
mpi.c, 1214
parallel.c, 1423
PF_ISendSbuf
mpi.c, 1213
parallel.h, 1496
PF_LibInit
mpi.c, 1212
parallel.c, 1422
PF_LibTerminate
mpi.c, 1213
parallel.c, 1422
PF_LOG_MSGTAG
parallel.h, 1494
PF_LongMultiBroadcast
mpi.c, 1225
parallel.h, 1506
PF_LongMultiPack
parallel.h, 1496
PF_LongMultiPackImpl
mpi.c, 1223
parallel.h, 1504
PF_LongMultiUnpack
parallel.h, 1496
PF_LongMultiUnpackImpl
mpi.c, 1224
parallel.h, 1505
PF_LongSinglePack
mpi.c, 1221
parallel.h, 1502
PF_LongSingleReceive
mpi.c, 1223
parallel.h, 1503
PF_LongSingleSend
mpi.c, 1222
parallel.h, 1503
PF_LongSingleUnpack
mpi.c, 1222
parallel.h, 1502
PF_maxDollarChunkSize
mpi.c, 1226
parallel.h, 1523
PF_MISC_MSGTAG
parallel.h, 1495
PF_MLock
parallel.c, 1440
parallel.h, 1521
PF_MUnlock
parallel.c, 1440
parallel.h, 1522
PF_OPT_COLLECT_MSGTAG
parallel.h, 1495
PF_OPT_HORNER_MSGTAG
parallel.h, 1495

PF_OPT_MCTS_MSGTAG
parallel.h, 1495
PF_Pack
mpi.c, 1218
parallel.h, 1498
PF_PACKSIZE
mpi.c, 1211
PF_PackString
mpi.c, 1219
parallel.h, 1499
PF_PrepareLongMultiPack
mpi.c, 1223
parallel.h, 1504
PF_PrepareLongSinglePack
mpi.c, 1221
parallel.h, 1502
PF_PreparePack
mpi.c, 1217
parallel.h, 1498
PF_PrintPackBuf
mpi.c, 1217
PF_Probe
mpi.c, 1213
parallel.c, 1423
PF_Processor
parallel.c, 1427
parallel.h, 1508
PF_RawProbe
mpi.c, 1217
parallel.c, 1425
PF_RawRecv
mpi.c, 1216
parallel.c, 1425
parallel.h, 1498
PF_RawSend
mpi.c, 1216
parallel.c, 1424
parallel.h, 1497
PF_READY_MSGTAG
parallel.h, 1494
PF_RealTime
mpi.c, 1212
parallel.c, 1421
PF_Receive
mpi.c, 1220
parallel.h, 1501
PF_RecvFile
parallel.c, 1439
parallel.h, 1521
PF_RecvWbuf
mpi.c, 1214
parallel.c, 1423
PF_RESET
parallel.h, 1493
PF_RestoreInsideInfo
parallel.c, 1436
parallel.h, 1519
PF_Send
mpi.c, 1220
parallel.h, 1500
PF_SendFile
parallel.c, 1439
parallel.h, 1520
PF_SNDFILEBUFSIZE
parallel.c, 1421
PF_STATS_SIZE
parallel.c, 1421
PF_STDOUT_MSGTAG
parallel.h, 1494
PF_StoreInsideInfo
parallel.c, 1436
parallel.h, 1518
PF_TERM_MSGTAG
parallel.h, 1493
PF_Terminate
parallel.c, 1429
parallel.h, 1510
PF_TIME
parallel.h, 1493
PF_Unpack
mpi.c, 1218
parallel.h, 1499
PF_UnpackString
mpi.c, 1219
parallel.h, 1500
PF_WaitRbuf
mpi.c, 1215
parallel.c, 1424
PF_WriteFileToFile
parallel.c, 1441
parallel.h, 1522
pfac
T_const, 244
PFORTRANMODE
ftypes.h, 1006
PHEAD
ftypes.h, 986
PHEAD0
ftypes.h, 986
PIPESTREAM
ftypes.h, 987
PISYMBOL
ftypes.h, 1022
pname
StreaM, 234
PObuffer
FiLe, 74
POfill
FiLe, 74
POfull
FiLe, 74
poin2a
sOrT, 220
poina
sOrT, 220
PoinFill

sOrT, 219
PoinScratch
N_const, 138
Pointer
CbUf, 47
pointer
StreaM, 233
poly, 189
check_memory, 191
modn, 192
modp, 191
operator[], 191
size_of_terms, 191
terms, 191
termscopy, 191
poly.h, 1557
poly_num_vars
N_const, 147
poly_vars
N_const, 140
poly_vars_type
N_const, 147
PolyAct
T_const, 251
POLYADD
ftypes.h, 1058
POLYDIV
ftypes.h, 1058
PolyDiv
declare.h, 762
ratio.c, 1743
polyfact.h, 1560
PolyFlag
sOrT, 223
POLYFUN
ftypes.h, 990
PolyFun
R_const, 204
PolyFunClean
declare.h, 624
function.c, 1084
PolyFunDirty
declare.h, 624
function.c, 1084
PolyFunExp
R_const, 204
PolyFunInv
R_const, 204
PolyFunMul
declare.h, 630
proces.c, 1675
PolyFunPow
R_const, 204
PolyFunTodo
N_const, 146
PolyFunType
R_const, 204
PolyFunVar
R_const, 204
POLYGCD
ftypes.h, 1058
polygcd.h, 1561
POLYINTFAC
ftypes.h, 1058
POLYMOD, 192
arraysize, 193
coefs, 192
modnum, 193
numsym, 192
polysize, 193
POLYMUL
ftypes.h, 1058
POLYNORM
ftypes.h, 1058
PolyNormFlag
N_const, 146
PolyRatFunChanged
C_const, 45
PolyRatFunSpecial
declare.h, 731
sort.c, 1961
POLYREM
ftypes.h, 1058
polysize
POLYMOD, 193
polysortflag
N_const, 144
POLYSUB
ftypes.h, 1058
PolyWise
sOrT, 223
POposition
FiLe, 74
POPPREASSIGNLEVEL
declare.h, 591
PopPreVars
declare.h, 669
pre.c, 1572
PopVariables
execute.c, 862
portsignals.h, 1562, 1566
FATAL_SIG_ERROR, 1563
NSIG, 1563
SIGALRM, 1565
SIGEMT, 1564
SIGFPE, 1563
SIGHUP, 1565
SIGILL, 1564
SIGINT, 1565
SIGLOST, 1564
SIGPIPE, 1564
SIGPROF, 1565
SIGQUIT, 1565
SIGSEGV, 1563
SIGSYS, 1564
SIGTERM, 1564

SIGVTALRM, 1565
SIGXCPU, 1564
SIGXFSZ, 1564
POS0_
ftypes.h, 1043
POS_
ftypes.h, 1043
PoSiTiOn, 193
p1, 193
Position
FiLeDaTa, 77
position
indexblock, 92
InDeXeNtRy, 94
nameblock, 148
objects, 167
SpecTatoR, 226
StOrEcAcHe, 227
PositionStream
declare.h, 654
tools.c, 2294
POSITIVEONLY
ftypes.h, 1060
POsize
FiLe, 75
POSNEG
ftypes.h, 1059
POstop
FiLe, 74
posWorkPointer
T_const, 246
posWorkSize
R_const, 201
posWorkSpace
T_const, 242
PotModdollars
variable.h, 2410
PotModDolList
C_const, 21
pow
COST, 52
power
factorized_poly, 72
powerFlag
O_const, 164
powmod
C_const, 24
pParseObject
declare.h, 680
pre.c, 1581
pre.c, 1567, 1586
AddToPreTypes, 1582
CharOut, 1571
ClearMacro, 1575
ClearPushback, 1571
DAEMON, 1570
defineChannel, 1583
DoBreakDo, 1576
DoCall, 1576
DoClearOptimize, 1584
DoCommentChar, 1574
DoDebug, 1576
DoDefine, 1574
DoDo, 1576
DoElse, 1576
DoElseif, 1576
DoEnddo, 1577
DoEndif, 1577
DoEndInside, 1577
DoEndprocedure, 1577
DoExternal, 1582
DoFactDollar, 1584
DoFromExternal, 1583
DoIf, 1577
DoIfdef, 1577
DoInclude, 1575
DoInside, 1577
DoMessage, 1578
DoOptimize, 1584
DoPipe, 1578
DoPrcExtension, 1578
DoPreAddSeparator, 1582
DoPreAppend, 1578
DoPreAppendPath, 1585
DoPreAssign, 1574
DoPreBreak, 1579
DoPreCase, 1579
DoPreClose, 1579
DoPreCreate, 1578
DoPreDefault, 1579
DoPreEndSwitch, 1580
DoPreExchange, 1575
DoPreOut, 1578
DoPrePrependPath, 1585
DoPrePrintTimes, 1578
DoPreRemove, 1579
DoPreReset, 1585
DoPreRmSeparator, 1582
DoPreShow, 1580
DoPreSwitch, 1580
DoPreWrite, 1579
DoProcedure, 1579
DoPrompt, 1583
DoRedefine, 1574
DoReverseInclude, 1575
DoRmExternal, 1583
DoSetExternal, 1583
DoSetExternalAttr, 1583
DoSetRandom, 1584
DoSkipExtraSymbols, 1585
DoSystem, 1580
DoTerminate, 1576
DoToExternal, 1583
DoUndefine, 1575
EvalPreIf, 1581
ExpandTripleDots, 1573

FALSE_EXPR, 1570
FindInKeyWord, 1573
FindKeyWord, 1573
GetChar, 1571
GetDollarNumber, 1584
GetInput, 1571
GetPreVar, 1571
Include, 1575
IniModule, 1572
IniSpecialModule, 1572
KILL, 1569
KILLALL, 1569
LoadInstruction, 1573
LoadStatement, 1573
MessPreNesting, 1582
NOSHELL, 1570
PopPreVars, 1572
pParseObject, 1581
PreCalc, 1582
PreCmp, 1581
PreEq, 1581
PreEval, 1582
PreIfEval, 1581
PreLoad, 1580
PreProcessor, 1572
PreProInstruction, 1572
PreSkip, 1580
PutPreVar, 1571
SHELL, 1570
SKIPBUFSIZE, 1569
StartPrepro, 1581
STDERR, 1570
TERMINAL, 1570
TheDefine, 1573
TheUndefine, 1575
TRUE_EXPR, 1570
UnsetAllowDelay, 1571
writeToChannel, 1584
PreAssignFlag
P_const, 177
PreAssignLevel
P_const, 179
PreAssignStack
P_const, 176
PreCalc
pre.c, 1582
PRECALCSTREAM
ftypes.h, 988
PreCmp
declare.h, 680
pre.c, 1581
PreContinuation
P_const, 177
PreDebug
P_const, 179
PreEq
declare.h, 680
pre.c, 1581
preError
P_const, 179
PreEval
declare.h, 679
pre.c, 1582
preFill
P_const, 175
PreIfDollarEval
declare.h, 734
dollar.c, 811
PreIfEval
declare.h, 680
pre.c, 1581
PreIfLevel
DoLoOp, 65
P_const, 178
PreIfStack
P_const, 176
PreInsideLevel
P_const, 178
PRELOAD, 194
buffer, 194
size, 194
PreLoad
declare.h, 681
pre.c, 1580
PRELOWERAFTER
ftypes.h, 991
PRENOACTION
ftypes.h, 991
PreOut
P_const, 177
PrepPoly
declare.h, 631
proces.c, 1675
PreProcessor
pre.c, 1572
PreproFlag
P_const, 177
PreProInstruction
pre.c, 1572
PREPROONLY
ftypes.h, 993
PRERAISEAFTER
ftypes.h, 991
PreRandom
declare.h, 747
reken.c, 1799
PREREADSTREAM
ftypes.h, 987
PREREADSTREAM2
ftypes.h, 988
PREREADSTREAM3
ftypes.h, 988
PreSkip
declare.h, 681
pre.c, 1580
preStart

P_const, 175
preStop
P_const, 175
PreSwitchLevel
DoLoOp, 65
P_const, 177
PreSwitchModes
P_const, 176
PreSwitchStrings
P_const, 175
PRETYPEDO
ftypes.h, 995
PRETYPEIF
ftypes.h, 995
PRETYPEINSIDE
ftypes.h, 995
PRETYPENONE
ftypes.h, 995
PRETYPEPROCEDURE
ftypes.h, 995
PreTypes
P_const, 176
PRETYPESWITCH
ftypes.h, 995
PREV
declare.h, 583
PREVAR
structs.h, 2123
PreVar
variable.h, 2405
pReVaR, 194
argnames, 195
name, 195
nargs, 195
value, 195
wildarg, 195
PreVarList
P_const, 174
prevars
StreaM, 235
PREVARSTREAM
ftypes.h, 987
previous
StreaM, 234
previousblock
indexblock, 92
nameblock, 148
previousEfactor
T_const, 244
previousNoShowInput
StreaM, 235
prevline
StreaM, 233
primelist
T_const, 243
PRIMENUMBER
ftypes.h, 1021
PRINTALL
ftypes.h, 1042
PRINTCONTENT
ftypes.h, 1041
PRINTCONTENTS
ftypes.h, 1041
PrintExtraSymbol
declare.h, 759
notation.c, 1370
PRINTFBUF
parallel.c, 1419
printflag
ExPrEsSiOn, 69
PRINTLFILE
ftypes.h, 1042
PRINTOFF
ftypes.h, 1041
PRINTON
ftypes.h, 1041
PRINTONEFUNCTION
ftypes.h, 1042
PRINTONETERM
ftypes.h, 1042
PrintRunningTime
startup.c, 2027
printscratch
proces.c, 1676
PrintSeq
declare.h, 684
message.c, 1155
printstats
OPTIMIZE, 171
PrintSubTerm
declare.h, 683
message.c, 1155
PrintSubtermList
declare.h, 759
notation.c, 1370
PrintTerm
declare.h, 683
message.c, 1155
PrintTermC
declare.h, 683
message.c, 1155
PrintTime
sch.c, 1889
PrintTotalSize
M_const, 119
PrintType
O_const, 165
PrintWords
declare.h, 684
message.c, 1155
PROCEDURE, 196
loadmode, 197
name, 197
p, 197
procedureExtension
P_const, 175

procedureextension
setfile.c, 1927
PROCEDUREFILE
ftypes.h, 989
Procedures
variable.h, 2404
proces.c, 1665, 1677
Deferred, 1674
DoOnePow, 1673
FiniTerm, 1671
Generator, 1671
InFunction, 1668
InsertTerm, 1669
PasteFile, 1670
PasteTerm, 1670
PolyFunMul, 1675
PrepPoly, 1675
printscratch, 1676
Processor, 1666
TestSub, 1667
ProcessBucketSize
C_const, 30
ProcessOption
declare.h, 655
setfile.c, 1925
Processor
proces.c, 1666
ProcessStats
C_const, 36
ProcList
P_const, 175
Product
declare.h, 631
reken.c, 1794
PRODUCTIONDATE
form3.h, 947
proexp
T_const, 250
PROPERORDERFLAG
ftypes.h, 1057
properorderflag
C_const, 37
ProtoType
C_const, 25
prototype
ExPrEsSiOn, 68
TaBlEs, 256
prototypeSize
TaBlEs, 258
PrtLong
declare.h, 631
reken.c, 1795
PrtTerms
sch.c, 1889
PruneExtraSymbols
declare.h, 761
notation.c, 1371
pSize
P_const, 176
psize
PaRtI, 183
pStop
sOrT, 220
PUSHPREASSIGNLEVEL
declare.h, 591
PutArgInScratch
declare.h, 755
transform.c, 2356
PutBracket
declare.h, 632
execute.c, 863
PutBracketInIndex
declare.h, 736
index.c, 1125
PutExtraSymbols
declare.h, 763
ratio.c, 1740
PUTFIRST
ftypes.h, 1022
PutIn
declare.h, 632
sort.c, 1951
PutInBuffers
pattern.c, 1528
PutInside
declare.h, 629
index.c, 1126
putinsize
sOrT, 222
PutInSpectator
declare.h, 770
spectator.c, 2014
PutInStore
declare.h, 632
store.c, 2052
PutInVflags
declare.h, 642
execute.c, 863
PutOut
declare.h, 632
sort.c, 1952
PutPreVar
declare.h, 652
pre.c, 1571
PutTableNames
declare.h, 743
minos.c, 1172
PUTZERO
declare.h, 585
PVFUNV
ftypes.h, 1071
PVFUNWP
ftypes.h, 1071
pWorkPointer
T_const, 245
pWorkSize

R_const, 201
pWorkSpace
T_const, 242

Q_
ftypes.h, 1044
qError
M_const, 123
QUADRUPLEFORTRANMODE
ftypes.h, 1007
QUADRUPLEPRECISIONFLAG
ftypes.h, 1007
Quotient
declare.h, 633
reken.c, 1795

R
AllGlobals, 6
R_const, 197
BracketOn, 203
Cnumlhs, 202
CompareRoutine, 201
CompressBuffer, 200
CompressPointer, 200
ComprTop, 200
CurDum, 203
CurExpr, 205
DeferFlag, 203
DefPosition, 200
Eside, 205
expchanged, 205
expflags, 205
FoStage4, 200
Fscr, 199
GetFile, 203
GetOneFile, 204
gzipCompress, 202
hidefile, 200
infile, 200
InHiBuf, 201
InInBuf, 201
KeptInHold, 203
level, 205
lWorkSize, 201
MaxBracket, 203
MaxDum, 205
NoCompress, 202
OldTime, 201
outfile, 200
outtohide, 202
PolyFun, 204
PolyFunExp, 204
PolyFunInv, 204
PolyFunPow, 204
PolyFunType, 204
PolyFunVar, 204
posWorkSize, 201
pWorkSize, 201
ShortSortCount, 205
sLevel, 203
SortType, 205
Stage4Name, 204
StoreData, 199
TePos, 203
wranfcall, 202
wranfia, 201
wranfnpair1, 202
wranfnpair2, 202
wranfseed, 202
R_COPY_B
checkpoint.c, 325
R_COPY_LIST
checkpoint.c, 326
R_COPY_NAMETREE
checkpoint.c, 326
R_COPY_S
checkpoint.c, 325
R_FREE
checkpoint.c, 324
R_FREE_NAMETREE
checkpoint.c, 324
R_FREE_STREAM
checkpoint.c, 324
R_SET
checkpoint.c, 325
RaisPow
declare.h, 633
reken.c, 1792
RaisPowCached
declare.h, 634
reken.c, 1793
RaisPowMod
declare.h, 634
reken.c, 1793
RANDOMFUNCTION
ftypes.h, 1020
ranges
DICTIONARY, 56
RANPERM
ftypes.h, 1020
RATIO
ftypes.h, 1033
ratio.c, 1738, 1745
BinomGen, 1739
ConvertArgument, 1744
CreateExpression, 1742
DIVfunction, 1743
divrem, 1744
DoSumF1, 1739
DoSumF2, 1740
ExpandRat, 1744
FromPolyRatFun, 1742
GCDclean, 1743
GCDfunction, 1740
GCDfunction3, 1740
GCDterms, 1742
Glue, 1740

InvPoly, 1744
MergeDotproductLists, 1742
MergeSymbolLists, 1741
MULfunc, 1743
MultiplyWithTerm, 1741
PolyDiv, 1743
PutExtraSymbols, 1740
RatioFind, 1739
RatioGen, 1739
ReadPolyRatFun, 1742
TakeContent, 1741
TakeExtraSymbols, 1741
TakeSymbolContent, 1742
TheErrorMessage, 1744
RatioFind
declare.h, 635
ratio.c, 1739
RatioGen
declare.h, 635
ratio.c, 1739
RATIONALMODE
ftypes.h, 1047
RatToLine
declare.h, 635
sch.c, 1888
RBRACE
ftypes.h, 1002
rbufnum
M_const, 114
rbufs
ParallelVars, 181
RCYCLESYMMETRIC
ftypes.h, 1049
ReadFile
declare.h, 662
tools.c, 2296
ReadFromScratch
declare.h, 770
store.c, 2051
ReadFromStream
declare.h, 653
tools.c, 2293
ReadijObject
declare.h, 741
minos.c, 1173
ReadIndex
declare.h, 740
minos.c, 1170
ReadIniInfo
declare.h, 741
minos.c, 1171
ReadObject
declare.h, 741
minos.c, 1173
ReadPolyRatFun
declare.h, 761
ratio.c, 1742
readpos
SpecTatoR, 226
ReadPosFile
declare.h, 662
tools.c, 2296
ReadRange
declare.h, 756
transform.c, 2356
ReadSaveExpression
declare.h, 748
store.c, 2059
ReadSaveHeader
store.c, 2056
ReadSaveIndex
declare.h, 748
store.c, 2057
ReadSaveTerm32
declare.h, 749
store.c, 2058
ReadSaveVariables
declare.h, 750
store.c, 2057
ReadSnum
declare.h, 636
tools.c, 2300
READSPECTATORFLAG
ftypes.h, 1070
RecalcSetups
setfile.c, 1925
RecFlag
T_const, 251
RecoveryFilename
checkpoint.c, 328
recvBuffer
parallel.c, 1421
REDEFINEDEXPRESSION
ftypes.h, 1042
REDLENG
declare.h, 574
RedoTableTree
declare.h, 730
tables.c, 2180
RedoTree
comtool.c, 542
declare.h, 722
REDUCEMODE
ftypes.h, 1006
REDUCESUBEXPBUFFERS
compiler.c, 499
REGULAR
ftypes.h, 1057
REGULAREXPRESSION
ftypes.h, 1042
REGULARSYMBOL
ftypes.h, 1065
reken.c, 1787, 1799
AccumGCD, 1791
AddLong, 1791
AddPLon, 1791

AddRat, 1790
Bernoulli, 1798
BigLong, 1792
CompCoef, 1797
COPYLONG, 1789
DivLong, 1792
DivRat, 1790
Divvy, 1790
Factorial, 1798
GcdLong, 1795
GCDMAX, 1788
GetBinom, 1796
GetLong, 1795
GetLongModInverses, 1794
GetModInverses, 1794
iniwranf, 1798
iranf, 1799
LcmLong, 1796
MakeInverses, 1793
MakeLongRational, 1796
MakeModTable, 1798
MakeRational, 1796
Modulus, 1797
MulLong, 1792
Mully, 1789
MulRat, 1790
NEWTRICK, 1788
NextPrime, 1798
NormalModulus, 1793
Pack, 1789
PreRandom, 1799
Product, 1794
PrtLong, 1795
Quotient, 1795
RaisPow, 1792
RaisPowCached, 1793
RaisPowMod, 1793
Remain10, 1795
Remain4, 1795
Simplify, 1790
SubPLon, 1791
TakeLongRoot, 1796
TakeModulus, 1797
TakeNormalModulus, 1797
TakeRatRoot, 1791
UnPack, 1789
WARMUP, 1789
wranf, 1799
ReleaseTB
tables.c, 2184
Remain10
declare.h, 636
reken.c, 1795
Remain4
declare.h, 636
reken.c, 1795
REMFUNCTION
ftypes.h, 1017
RemoveDictionary
declare.h, 766
dict.c, 792
RemoveDollars
names.c, 1267
renum
ExPrEsSiOn, 68
RENUMBER
structs.h, 2121
ReNuMbEr, 206
func, 207
funnum, 208
indi, 207
indnum, 208
startposition, 207
symb, 207
symnum, 208
vecnum, 208
vect, 207
ReNumber
declare.h, 635
reshuf.c, 1845
RenumberVec
O_const, 161
renumlists
ExPrEsSiOn, 69
RenumScratch
N_const, 139
ReOpenFile
declare.h, 661
tools.c, 2295
RepCount
T_const, 243
RepFunList
N_const, 135
RepFunNum
N_const, 145
replace
ExPrEsSiOn, 70
REPLACEARG
ftypes.h, 1062
ReplaceDollar
declare.h, 658
names.c, 1266
REPLACELOOP
ftypes.h, 1057
REPLACEMENT
ftypes.h, 1013
ReplaceScrat
N_const, 138
RepLevel
C_const, 40
RepMax
M_const, 122
RepPoint
N_const, 135
RepSumCheck
C_const, 40

RepTop
T_const, 243
request
PF_BUFFER, 188
reserved
STOREHEADER, 231
StreaM, 236
TaBlEs, 258
ReserveTempFiles
declare.h, 683
startup.c, 2025
ResetScratch
store.c, 2051
resetTimeOnClear
M_const, 118
ResetVariables
declare.h, 685
names.c, 1267
reshuf.c, 1844, 1849
AdjustRenumScratch, 1845
CountDo, 1846
CountFun, 1846
DetCurDum, 1845
DimensionExpression, 1846
DimensionSubterm, 1846
DimensionTerm, 1846
DoDelta3, 1847
DoDistrib, 1847
DoPartitions, 1847
DoPermutations, 1848
DoShuffle, 1848
DoStuffle, 1848
EqualArg, 1847
FinishShuffle, 1848
FinishStuffle, 1849
FullRenumber, 1845
FunLevel, 1845
MoveDummies, 1845
MultDo, 1846
NEWCODE, 1845
ReNumber, 1845
Shuffle, 1848
Stuffle, 1848
StuffRootAdd, 1849
TestPartitions, 1847
TryDo, 1847
ResizeData
O_const, 160
resizeFlag
O_const, 164
ResizeLONG
O_const, 161
ResizeNCWORD
O_const, 160
ResizePOINTER
O_const, 161
ResizePOS
O_const, 161
ResizeWORD
O_const, 160
ResolveSet
declare.h, 636
wildcard.c, 2428
retstat
PF_BUFFER, 188
REVERSEARG
ftypes.h, 1063
REVERSEFILESTREAM
ftypes.h, 988
REVERSEFUNCTION
ftypes.h, 1013
REVERSEORDER
ftypes.h, 1049
ReverseStatements
declare.h, 654
tools.c, 2294
RevertScratch
store.c, 2051
revision
STOREHEADER, 231
ReWorkT
declare.h, 610
tables.c, 2183
rght
NoDe, 155
rhs
CbUf, 47
DICTIONARY_ELEMENT, 57
RHSIDE
ftypes.h, 1006
rhsInParallel
ParallelVars, 182
right
NaMeNode, 150
tree, 269
RLONG
form3.h, 955
rloser
NoDe, 155
ROOTFUNCTION
ftypes.h, 1016
rootnum
CbUf, 50
TaBlEs, 259
RPARENTHESIS
ftypes.h, 1002
rsrc
NoDe, 156
RSsize
N_const, 143
RunAddArg
declare.h, 756
transform.c, 2355
RunCycle
declare.h, 756
transform.c, 2355

RunDecode
declare.h, 755
transform.c, 2354
RunDedup
declare.h, 757
transform.c, 2355
RunDropArg
declare.h, 757
transform.c, 2356
RunEncode
declare.h, 754
transform.c, 2354
RunExplode
declare.h, 755
transform.c, 2354
RunImplode
declare.h, 755
transform.c, 2354
RunIsLyndon
declare.h, 757
transform.c, 2355
RunMulArg
declare.h, 757
transform.c, 2355
RunPermute
declare.h, 756
transform.c, 2354
RunReplace
declare.h, 755
transform.c, 2354
RunReverse
declare.h, 756
transform.c, 2355
RunSelectArg
declare.h, 757
transform.c, 2356
RunToLyndon
declare.h, 757
transform.c, 2356
RunTransform
declare.h, 754
transform.c, 2353
RWLOCKR
declare.h, 592
RWLOCKW
declare.h, 592

S
AllGlobals, 6
S0
M_const, 107
T_const, 241
S_const, 209
Balancing, 210
CollectOverFlag, 210
ExecMode, 210
MaxExprSize, 209
MultiThreaded, 210
NumOldNumFactors, 210
NumOldOnFile, 210
OldNumFactors, 210
OldOnFile, 209
Oldvflags, 210
S_FLUSH_B
checkpoint.c, 325
s_one
FixedGlobals, 81
S_WRITE_B
checkpoint.c, 325
S_WRITE_DOLLAR
checkpoint.c, 327
S_WRITE_LIST
checkpoint.c, 326
S_WRITE_NAMETREE
checkpoint.c, 327
S_WRITE_S
checkpoint.c, 326
saIter
OPTIMIZE, 171
SaveData
O_const, 158
SaveHeader
O_const, 158
SAVEREVISION
store.c, 2050
sBer
T_const, 245
sbuf
ParallelVars, 181
sBuffer
sOrT, 218
sbufnum
M_const, 114
SBYTE
form3.h, 954
ScanFunctions
declare.h, 636
function.c, 1086
sch.c, 1885, 1891
AddArrayIndex, 1889
AddToLine, 1887
CodeToLine, 1888
FiniLine, 1887
IniLine, 1887
LongToLine, 1888
MultiplyToLine, 1888
PrintTime, 1889
PrtTerms, 1889
RatToLine, 1888
StrCopy, 1887
TalToLine, 1888
TokenToLine, 1888
va_arg, 1887
va_dcl, 1886
va_end, 1886
va_list, 1887
va_start, 1886

WriteAll, 1890
WriteArgument, 1889
WriteDictionary, 1889
WriteExpression, 1890
WriteInnerTerm, 1890
WriteLists, 1889
WriteOne, 1890
WriteSubTerm, 1890
WriteTerm, 1890
WrtPower, 1889
schemeflags
OPTIMIZE, 171
schemenum
O_const, 164
SCRATCHSIZE
fsizes.h, 969
ScratSize
M_const, 110
SEARCHINGPRECASE
ftypes.h, 993
SEARCHINGPREENDSWITCH
ftypes.h, 993
SeekFile
declare.h, 662
tools.c, 2296
SeekScratch
declare.h, 636
store.c, 2050
SELECTARG
ftypes.h, 1064
selectExternalChannel
declare.h, 745
extcmd.c, 897
selecttermundo
N_const, 138
SEPARATESYMBOL
ftypes.h, 1023
SEPARATOR
fwin.h, 1110
unix.h, 2398
separators
C_const, 19
set_del
declare.h, 744
tools.c, 2301
set_in
declare.h, 744
tools.c, 2301
set_of_char
structs.h, 2123
set_set
declare.h, 744
tools.c, 2301
set_sub
declare.h, 744
tools.c, 2302
SETBASELENGTH
declare.h, 584
SETBASEPOSITION
declare.h, 584
SETBIT
ftypes.h, 1055
SETBUFSIZE
setfile.c, 1924
SetDictionaryOptions
declare.h, 766
dict.c, 792
SetElementList
C_const, 21
SetElements
variable.h, 2405
SetEndHScratch
declare.h, 637
store.c, 2051
SetEndScratch
declare.h, 637
store.c, 2051
SETERROR
declare.h, 583
SETEXP
ftypes.h, 1010
SetExpr
compcomm.c, 400
declare.h, 689
SetExprCases
compcomm.c, 400
declare.h, 732
setfile.c, 1923, 1928
AllocFileHandle, 1926
AllocSetups, 1925
AllocSort, 1925
AllocSortFileName, 1926
commentchar, 1927
curdirp, 1927
cursortdirp, 1927
DeAllocFileHandle, 1926
DEFINEVALUE, 1924
DoSetups, 1925
dotchar, 1927
GetSetupPar, 1925
highfirst, 1927
lowfirst, 1927
MakeSetupAllocs, 1926
NUMERICALVALUE, 1924
ONOFFVALUE, 1924
PATHVALUE, 1924
procedureextension, 1927
ProcessOption, 1925
RecalcSetups, 1925
SETBUFSIZE, 1924
setupparameters, 1928
STRINGVALUE, 1924
TryEnvironment, 1926
TryFileSetups, 1926
WriteSetup, 1925
SetFileIndex

store.c, 2054
SETFUNCTION
ftypes.h, 1012
SETKILLMODEFOREXTERNALCHANNEL
declare.h, 594
SetList
C_const, 21
SetMods
declare.h, 651
execute.c, 863
SETNUMBER
ftypes.h, 1036
SETONLY
ftypes.h, 998
setonoff
compcomm.c, 397
declare.h, 689
SetPosFile
declare.h, 663
tools.c, 2297
SeTs, 211
dimension, 212
first, 211
last, 211
name, 211
namesize, 212
node, 212
type, 211
Sets
variable.h, 2405
SetScratch
declare.h, 651
store.c, 2051
SETSET
ftypes.h, 1010
SetSpecialMode
declare.h, 681
module.c, 1198
SETSTARTPOS
declare.h, 586
SETTERMINATORFOREXTERNALCHANNEL
declare.h, 594
SETTONUM
ftypes.h, 1035
SetupDir
M_const, 109
SETUPFILE
ftypes.h, 989
SetupFile
M_const, 109
setupfilename
inivar.h, 1136
variable.h, 2412
SetupFlag
C_const, 34
SETUPPARAMETERS, 212
flags, 213
parameter, 213
type, 213
value, 213
setupparameters
setfile.c, 1928
sfact
T_const, 246
SFHSIZE
fsizes.h, 970
sFill
sOrT, 219
Sflush
declare.h, 637
sort.c, 1952
sFun
STOREHEADER, 230
SGN
declare.h, 574
sgn
TrAcEn, 262
TrAcEs, 266
sHalf
sOrT, 219
SHcombi
N_const, 140
SHcombisize
N_const, 141
SHELL
pre.c, 1570
shellname
X_const, 273
SHMWINSIZE
fsizes.h, 970
shmWinSize
M_const, 111
ShortSortCount
R_const, 205
ShortStats
C_const, 31
ShortStatsMax
C_const, 45
ShrinkDictionary
declare.h, 768
dict.c, 793
Shuffle
declare.h, 608
reshuf.c, 1848
SHvar
N_const, 140
SHvariables, 213
combilast, 215
do_uffle, 215
finishuf, 215
incoef, 214
level, 215
nincoef, 215
option, 215
outfun, 214
outterm, 214

stop1, 214
stop2, 214
ststop1, 214
ststop2, 214
thefunction, 215
SIGALRM
portsignals.h, 1565
SIGEMT
portsignals.h, 1564
SIGFPE
portsignals.h, 1563
SIGFUNCTION
ftypes.h, 1015
SIGHUP
portsignals.h, 1565
SIGILL
portsignals.h, 1564
SIGINT
portsignals.h, 1565
SIGLOST
portsignals.h, 1564
sign
DiStRiBuTe, 59
PeRmUtE, 185
PeRmUtEp, 186
sign1
TrAcEs, 267
sign2
TrAcEs, 267
SignCheck
N_const, 147
SIGNFUN
ftypes.h, 1014
SIGPIPE
portsignals.h, 1564
SIGPROF
portsignals.h, 1565
SIGQUIT
portsignals.h, 1565
SIGSEGV
portsignals.h, 1563
SIGSYS
portsignals.h, 1564
SIGTERM
portsignals.h, 1564
SIGVTALRM
portsignals.h, 1565
SIGXCPU
portsignals.h, 1564
SIGXFSZ
portsignals.h, 1564
silent
M_const, 123
simp1token
declare.h, 718
token.c, 2263
simp2token
declare.h, 719
token.c, 2264
simp3atoken
declare.h, 719
token.c, 2264
simp3btoken
declare.h, 719
token.c, 2264
simp4token
declare.h, 719
token.c, 2264
simp5token
declare.h, 719
token.c, 2264
simp6token
declare.h, 720
token.c, 2265
Simplify
declare.h, 637
reken.c, 1790
simpwtoken
declare.h, 719
token.c, 2264
sInd
STOREHEADER, 230
SINFUNCTION
ftypes.h, 1018
SINHFUNCTION
ftypes.h, 1019
SIOsize
M_const, 110
size
ARGBUFFER, 8
DICTIONARY_ELEMENT, 57
DoLIArS, 60
ExPrEsSiOn, 68
FaCdOlLaR, 71
InDeXeNtRy, 95
INSIDEINFO, 98
LIST, 101
MiNmAx, 128
objects, 167
PRELOAD, 194
TaBlEbAsEsUbInDeX, 254
size_of_terms
poly, 191
SizeCommuteInSet
C_const, 39
SizeDictionaries
O_const, 163
sizeelements
DICTIONARY, 55
SIZEFACS
fsizes.h, 966
SizeForSpectatorFiles
M_const, 120
SizeInFile
sOrT, 218
sizeprimelist

T_const, 251
sizeprototype
ExPrEsSiOn, 71
sizeselecttermundo
N_const, 146
SIZESTORECACHE
fsizes.h, 972
SizeStoreCache
M_const, 110
SKIP
ftypes.h, 1038
SkipAName
compiler.c, 500
declare.h, 720
SKIPBLANKS
declare.h, 577
SKIPBRA1
declare.h, 578
SKIPBRA2
declare.h, 578
SKIPBRA3
declare.h, 578
SKIPBRA4
declare.h, 578
SKIPBRA5
declare.h, 579
SKIPBUFSIZE
pre.c, 1569
SkipClears
M_const, 115
SkipField
declare.h, 666
tools.c, 2300
SKIPGEXPRESSION
ftypes.h, 1039
SKIPLEXPRESSION
ftypes.h, 1038
SKIPUNHIDEGEXPRESSION
ftypes.h, 1041
SKIPUNHIDELEXPRESSION
ftypes.h, 1041
SLARGEBUFFER
fsizes.h, 969
SLargeSize
M_const, 111
slavebuf
ParallelVars, 181
sLevel
R_const, 203
small_power
T_const, 243
small_power_maxn
T_const, 249
small_power_maxx
T_const, 249
small_power_n
T_const, 243
SMALLBUFFER
fsizes.h, 967
SmallEsize
sOrT, 221
SMALLOVERFLOW
fsizes.h, 968
SmallSize
sOrT, 221
smart.c, 1942, 1943
MatchIsPossible, 1942
StudyPattern, 1942
SMAXFPATCHES
fsizes.h, 969
SMaxFpatches
M_const, 113
SMAXPATCHES
fsizes.h, 969
SMaxPatches
M_const, 113
SNUMBER
ftypes.h, 1011
SORT
ftypes.h, 1054
sOrT, 216
cBuffer, 219
cBufferSize, 223
f, 221
ff, 221
file, 218
fPatches, 220
fPatchesStop, 221
fPatchN, 224
fPatchN1, 223
GenSpace, 222
GenTerms, 222
inNum, 224
inPatches, 220
iPatches, 221
ktoi, 220
LargeSize, 221
lBuffer, 218
lFill, 218
lPatch, 223
lTop, 218
MaxFpatches, 223
MaxPatches, 223
maxtermsize, 224
newmaxtermsize, 224
ninterms, 222
outputmode, 224
Patches, 220
poin2a, 220
poina, 220
PoinFill, 219
PolyFlag, 223
PolyWise, 223
pStop, 220
putinsize, 222
sBuffer, 218

sFill, 219
sHalf, 219
SizeInFile, 218
SmallEsize, 221
SmallSize, 221
SpaceLeft, 222
SplitScratch, 219
sPointer, 219
stage4, 224
stagelevel, 224
sTerms, 221
sTop, 219
sTop2, 219
Terms2InSmall, 222
TermsInSmall, 222
TermsLeft, 222
tree, 220
type, 223
used, 218
sort.c, 1948, 1962
AddArgs, 1955
AddCoef, 1953
AddPoly, 1954
CleanUpSort, 1961
Compare1, 1955
CompareHSymbols, 1957
CompareSymbols, 1956
ComPress, 1957
EndSort, 1950
FlushOut, 1953
GarbHand, 1958
INSLENGTH, 1949
LowerSortLevel, 1961
MergePatches, 1959
NewSort, 1950
NEWSPLITMERGE, 1949
PolyRatFunSpecial, 1961
PutIn, 1951
PutOut, 1952
Sflush, 1952
SortWild, 1960
SplitMerge, 1957
StageSort, 1960
StoreTerm, 1959
toterms, 1962
WriteStats, 1949
SORTANTIPOWER
ftypes.h, 1009
SORTHIGHFIRST
ftypes.h, 1008
SORTING
structs.h, 2124
SORTIOSIZE
fsizes.h, 968
SORTLOWFIRST
ftypes.h, 1008
SORTMODULE
ftypes.h, 989
SORTPOWERFIRST
ftypes.h, 1008
SortTheList
declare.h, 725
lus.c, 1140
SortType
BrAcKeTiNfO, 13
C_const, 35
R_const, 205
SortWeights
argument.c, 280
declare.h, 694
SortWild
declare.h, 637
sort.c, 1960
SoScratC
N_const, 139
SpaceLeft
sOrT, 222
spare
TaBlEs, 257
spare1
objects, 168
spare2
objects, 168
spare3
objects, 168
SpareTable
declare.h, 651
tables.c, 2180
sparse
TaBlEs, 259
spec
FuNcTiOn, 87
SpecialCleanup
declare.h, 650
execute.c, 863
SPECIFICATION
ftypes.h, 996
SPECMASK
form3.h, 949
SpecTatoR, 225
exprnumber, 226
fh, 226
flags, 226
name, 226
position, 226
readpos, 226
spectator.c, 2013, 2015
ClearSpectators, 2014
CoCopySpectator, 2014
CoCreateSpectator, 2013
CoEmptySpectator, 2014
CoRemoveSpectator, 2013
CoToSpectator, 2013
FlushSpectators, 2014
GetFromSpectator, 2014
PutInSpectator, 2014

SPECTATOREXPRESSION
ftypes.h, 1041
SpectatorFiles
M_const, 110
SPECTATORSIZE
fsizes.h, 969
SpectatorSize
M_const, 112
SplitMerge
declare.h, 638
sort.c, 1957
SplitScratch
N_const, 139
sOrT, 219
SplitScratch1
N_const, 139
SplitScratchSize
N_const, 141
SplitScratchSize1
N_const, 141
sPointer
sOrT, 219
SQRTFUNCTION
ftypes.h, 1018
SS
T_const, 241
SSMALLBUFFER
fsizes.h, 968
SSmallEsize
M_const, 111
SSMALLOVERFLOW
fsizes.h, 968
SSmallSize
M_const, 111
SSORTIOSIZE
fsizes.h, 969
sSym
STOREHEADER, 230
stage4
sOrT, 224
Stage4Name
R_const, 204
stagelevel
sOrT, 224
StageSort
declare.h, 685
sort.c, 1960
stap
TrAcEs, 266
start
BrAcKeTiNdEx, 11
VaRrEnUm, 271
StartFiles
tools.c, 2294
startlinenumber
DoLoOp, 63
StartMore
startup.c, 2026
startposition
ReNuMbEr, 207
StartPrepro
pre.c, 1581
startup.c, 2023, 2028
CleanUp, 2026
defaulttempfilename, 2027
DoTail, 2025
emptystring, 2027
FORMNAME, 2024
GetRunningTime, 2027
IniVars, 2026
main, 2026
OpenInput, 2025
PrintRunningTime, 2027
ReserveTempFiles, 2025
StartMore, 2026
StartVariables, 2025
STRINGIFY, 2024
STRINGIFY__, 2024
TAKEPATH, 2024
Terminate, 2026
VERSIONSTR, 2024
VERSIONSTR__, 2024
StartVariables
declare.h, 595
startup.c, 2025
STATEMENT
ftypes.h, 996
STATIC_ASSERT
form3.h, 948
STATIC_ASSERT__1
form3.h, 948
STATIC_ASSERT__2
form3.h, 948
STATIC_ASSERT__3
form3.h, 948
StatsFlag
C_const, 34
STATSOUT
ftypes.h, 994
status
ExPrEsSiOn, 70
PF_BUFFER, 187
STDERR
pre.c, 1570
stderrname
X_const, 273
StdOut
M_const, 113
step1
TrAcEs, 266
STEP2
dollar.c, 808
sTerms
sOrT, 221
STermsInSmall
M_const, 111

STERMSSMALL
fsizes.h, 969
sTop
sOrT, 219
stop
PF_BUFFER, 187
stop1
SHvariables, 214
sTop2
sOrT, 219
stop2
SHvariables, 214
StopWatchZero
P_const, 176
STORE
ftypes.h, 1054
store.c, 2049, 2060
AddToScratch, 2052
CoLoad, 2052
CopyExpression, 2056
CoSave, 2052
DeleteStore, 2052
DetVars, 2053
FindInIndex, 2055
FindrNumber, 2055
GetFromStore, 2053
GetMoreFromMem, 2053
GetMoreTerms, 2053
GetOneTerm, 2053
GetTable, 2055
GetTerm, 2052
NextFileIndex, 2054
OpenTemp, 2050
PutInStore, 2052
ReadFromScratch, 2051
ReadSaveExpression, 2059
ReadSaveHeader, 2056
ReadSaveIndex, 2057
ReadSaveTerm32, 2058
ReadSaveVariables, 2057
ResetScratch, 2051
RevertScratch, 2051
SAVEREVISION, 2050
SeekScratch, 2050
SetEndHScratch, 2051
SetEndScratch, 2051
SetFileIndex, 2054
SetScratch, 2051
TermRenumber, 2054
ToStorage, 2053
VarStore, 2054
WriteStoreHeader, 2056
STORECACHE
structs.h, 2124
StOrEcAcHe, 227
buffer, 228
next, 228
position, 227
toppos, 227
StoreCache
T_const, 242
StoreCacheAlloc
T_const, 242
StoreData
R_const, 199
STOREDEXPRESSION
ftypes.h, 1039
StoreFileSize
C_const, 19
StoreHandle
C_const, 43
STOREHEADER, 228
endianness, 230
headermark, 229
lenLONG, 229
lenPOINTER, 229
lenPOS, 229
lenWORD, 229
maxpower, 231
reserved, 231
revision, 231
sFun, 230
sInd, 230
sSym, 230
sVec, 230
wildoffset, 231
STOREMODULE
ftypes.h, 990
StoreTerm
declare.h, 639
sort.c, 1959
str_dup
declare.h, 739
minos.c, 1169
StrCmp
declare.h, 666
tools.c, 2298
StrCopy
declare.h, 597
sch.c, 1887
strDup1
declare.h, 652
tools.c, 2300
STREAM
structs.h, 2122
StreaM, 232
afterwards, 235
buffer, 233
bufferposition, 234
buffersize, 234
eqnum, 235
fileposition, 232
FoldName, 233
handle, 234
inbuffer, 234
isnextchar, 235

linenumber, 232
name, 233
nextchar, 235
olddelay, 235
oldnoshowinput, 235
pname, 234
pointer, 233
prevars, 235
previous, 234
previousNoShowInput, 235
prevline, 233
reserved, 236
top, 233
type, 234
Streams
C_const, 23
StrHICmp
declare.h, 667
tools.c, 2298
StrICmp
declare.h, 666
tools.c, 2298
StrICont
declare.h, 667
tools.c, 2298
strict
TaBlEs, 259
STRINGIFY
startup.c, 2024
STRINGIFY__
startup.c, 2024
STRINGVALUE
setfile.c, 1924
StrLen
declare.h, 667
tools.c, 2299
structs.h, 2117, 2126
ALLGLOBALS, 2125
CBUF, 2123
CHANNEL, 2123
COMPTREE, 2121
DISTRIBUTE, 2124
DOLOOP, 2123
EMPTYININDEX, 2120
FILEHANDLE, 2122
FILEINDEX, 2121
FIXEDGLOBALS, 2125
FUNCTIONS, 2122
INDEXENTRY, 2121
INFILEINDEX, 2120
NAMENODE, 2121
NAMETREE, 2121
NESTING, 2124
one_byte, 2123
PARTI, 2124
PERM, 2124
PERMP, 2124
PREVAR, 2123
RENUMBER, 2121
set_of_char, 2123
SORTING, 2124
STORECACHE, 2124
STREAM, 2122
TABLES, 2122
TRACEN, 2122
TRACES, 2122
VARRENUM, 2121
WCN, 2125
WCN2, 2125
ststop1
SHvariables, 214
ststop2
SHvariables, 214
StudyPattern
declare.h, 725
smart.c, 1942
StuffAdd
declare.h, 575
Stuffle
declare.h, 609
reshuf.c, 1848
StuffRootAdd
declare.h, 609
reshuf.c, 1849
SUBAFTER
ftypes.h, 1050
SUBAFTERNOT
ftypes.h, 1051
SUBALL
ftypes.h, 1050
SuBbUf, 236
buffernum, 236
subexpnum, 236
SUBDISORDER
ftypes.h, 1050
subexpbuffers
compiler.c, 502
subexpnum
SuBbUf, 236
SUBEXPR
ftypes.h, 1052
SUBEXPRESSION
ftypes.h, 1009
SUBEXPSIZE
ftypes.h, 1037
subindex
TaBlEbAsE, 253
SUBMANY
ftypes.h, 1050
SUBMASK
ftypes.h, 1050
SUBMULTI
ftypes.h, 1049
SUBONCE
ftypes.h, 1049
SUBONLY

ftypes.h, 1050
SubPLon
declare.h, 640
reken.c, 1791
SUBROUTINEFILE
ftypes.h, 989
SUBSELECT
ftypes.h, 1050
SubsInAll
declare.h, 733
pattern.c, 1529
SUBSORT
ftypes.h, 1046
Substitute
declare.h, 640
pattern.c, 1529
subsubveto
N_const, 144
SUBVECTOR
ftypes.h, 1050
SUMF1
ftypes.h, 1012
SUMF2
ftypes.h, 1012
SUMMEDIND
ftypes.h, 1045
sumnum
M_const, 124
SUMNUM1
ftypes.h, 1033
SUMNUM2
ftypes.h, 1033
sumpnum
M_const, 124
SumTime
M_const, 112
sVec
STOREHEADER, 230
SWAP
parallel.c, 1419
swmes
FixedGlobals, 81
symb
ReNuMbEr, 207
SYMBOL
ftypes.h, 1009
SyMbOl, 237
complex, 237
dimension, 238
flags, 238
maxpower, 237
minpower, 237
name, 237
namesize, 238
node, 238
number, 237
SYMBOL_
ftypes.h, 1044
SymbolList
C_const, 21
SymbolNormalize
declare.h, 746
normal.c, 1307
SYMBOLONLY
ftypes.h, 997
Symbols
C_const, 23
symbols
variable.h, 2406
SymChangeFlag
C_const, 44
SymFind
declare.h, 640
function.c, 1085
SymGen
declare.h, 640
function.c, 1085
symmet
FUN_INFO, 84
symmetr.c, 2149, 2151
Distribute, 2150
FunMatchCy, 2150
FunMatchSy, 2151
MatchArgument, 2151
MatchCy, 2150
MatchE, 2150
Permute, 2150
PermuteP, 2150
SYMMETRIC
ftypes.h, 1049
symmetric
FuNcTiOn, 87
SYMMETRIZE
ftypes.h, 1033
Symmetrize
declare.h, 641
function.c, 1085
symminfo
FuNcTiOn, 86
symnum
ReNuMbEr, 208
SYMTONUM
ftypes.h, 1034
SYMTOSUB
ftypes.h, 1034
SYMTOSYM
ftypes.h, 1034
SynchFile
declare.h, 663
tools.c, 2297

T
AllGlobals, 7
T_const, 239
aebufnum, 246
allbufnum, 246
bernoullis, 243

BrackBuf, 241
bracketindexflag, 248
bracketinfo, 245
CacheNumberMemHeap, 244
CacheNumberMemMax, 248
CacheNumberMemTop, 248
comfun, 249
comind, 249
comnum, 249
comsym, 249
dummysubexp, 249
ebufnum, 246
factorials, 243
fbufnum, 246
fromindex, 251
FunArg, 250
idallflag, 247
idallmaxnum, 247
idallnum, 247
InNumMem, 246
inprimelist, 251
LeaveNegative, 248
ListPoly, 245
ListSymbols, 245
ListSymbolsSize, 248
locwildvalue, 250
lWorkPointer, 245
lWorkSpace, 242
mBer, 251
mfac, 246
MinVecArg, 250
mulpat, 250
nBer, 251
Nest, 241
NestPoin, 241
NestStop, 241
nfac, 251
NumberMemHeap, 244
NumberMemMax, 247
NumberMemTop, 247
NumListSymbols, 248
NumMem, 245
numpoly, 248
optimtimes, 248
partitions, 245
pBer, 244
pfac, 244
PolyAct, 251
posWorkPointer, 246
posWorkSpace, 242
previousEfactor, 244
primelist, 243
proexp, 250
pWorkPointer, 245
pWorkSpace, 242
RecFlag, 251
RepCount, 243
RepTop, 243
S0, 241
sBer, 245
sfact, 246
sizeprimelist, 251
small_power, 243
small_power_maxn, 249
small_power_maxx, 249
small_power_n, 243
SS, 241
StoreCache, 242
StoreCacheAlloc, 242
TermMemHeap, 244
TermMemMax, 247
TermMemTop, 247
TMaddr, 244
TMbuff, 250
TMdolfac, 250
TMout, 250
TrimPower, 249
WildArgTaken, 243
WildcardBufferSize, 247
WildMask, 244
WorkPointer, 242
WorkSpace, 242
WorkTop, 242
tabl
FuNcTiOn, 86
TaBlEbAsE, 252
current, 253
fillpoint, 253
name, 253
numtables, 253
subindex, 253
tablenumbers, 253
TABLEBASEFILE
ftypes.h, 989
TableBaseList
C_const, 22
tablebases
variable.h, 2406
TaBlEbAsEsUbInDeX, 254
size, 254
where, 254
tablecheck
C_const, 33
TABLEEXTENSION
fsizes.h, 970
tablefilling
C_const, 37
TABLEFUNCTION
ftypes.h, 1016
tablenamefill
dbase, 53
tablenames
dbase, 54
tablenamessize
dbase, 53
tablenum

TaBlEs, 260
tablenumber
objects, 167
tablenumbers
TaBlEbAsE, 253
tablepointers
TaBlEs, 256
TableReset
tables.c, 2184
TABLES
structs.h, 2122
TaBlEs, 255
argtail, 257
boomlijst, 257
bounds, 259
buffers, 257
buffersfill, 260
bufferssize, 260
bufnum, 260
defined, 258
flags, 256
MaxTreeSize, 260
mdefined, 258
mm, 256
mode, 261
numdummies, 261
numind, 258
numtree, 259
pattern, 256
prototype, 256
prototypeSize, 258
reserved, 258
rootnum, 259
spare, 257
sparse, 259
strict, 259
tablenum, 260
tablepointers, 256
totind, 257
tables.c, 2178, 2185
Apply, 2183
ApplyExec, 2183
ApplyReset, 2184
CheckTableDeclarations, 2181
ClearTableTree, 2179
CoApply, 2183
CoTableBase, 2180
CoTBaddto, 2181
CoTBaudit, 2182
CoTBcleanup, 2182
CoTBcreate, 2181
CoTBenter, 2181
CoTBhelp, 2183
CoTBload, 2182
CoTBoff, 2182
CoTBon, 2182
CoTBopen, 2181
CoTBreplace, 2182
CoTBuse, 2183
CoTestUse, 2181
DoTableExpansion, 2180
FindTableTree, 2180
FindTB, 2181
FlipTable, 2180
helptb, 2184
InsTableTree, 2179
RedoTableTree, 2180
ReleaseTB, 2184
ReWorkT, 2183
SpareTable, 2180
TableReset, 2184
TestUse, 2182
TABLESIZE
declare.h, 582
TABLESTUB
ftypes.h, 1018
tabstring
O_const, 159
tag
PF_BUFFER, 188
TakeArgContent
argument.c, 278
declare.h, 690
TakeContent
declare.h, 763
ratio.c, 1741
TakeDollarContent
declare.h, 727
dollar.c, 814
TakeExtraSymbols
declare.h, 763
ratio.c, 1741
TakeIDfunction
declare.h, 733
pattern.c, 1530
TakeLongRoot
declare.h, 729
reken.c, 1796
TakeModulus
declare.h, 641
reken.c, 1797
TakeNormalModulus
declare.h, 641
reken.c, 1797
TAKEPATH
startup.c, 2024
TakeRatRoot
declare.h, 729
reken.c, 1791
TakeSymbolContent
declare.h, 762
ratio.c, 1742
TAKETRACE
ftypes.h, 1033
TalToLine
declare.h, 641

sch.c, 1888
TANFUNCTION
ftypes.h, 1019
TANHFUNCTION
ftypes.h, 1019
TCOMMA
ftypes.h, 1002
TCONJUGATE
ftypes.h, 1004
TDELTA
ftypes.h, 1001
TDIVIDE
ftypes.h, 1003
TDOLLAR
ftypes.h, 1001
TDOT
ftypes.h, 1002
TDOTPRODUCT
ftypes.h, 1001
TDUBIOUS
ftypes.h, 1001
TeInFun
N_const, 145
TELLFILE
declare.h, 595
tools.c, 2296
TellFile
declare.h, 662
tools.c, 2296
TempDir
M_const, 108
TempSortDir
M_const, 108
TEMPTY
ftypes.h, 1004
TENDOFIT
ftypes.h, 1003
tensor
FUN_INFO, 84
TENSORFUNCTION
ftypes.h, 1043
tensorList
O_const, 161
TENVEC
ftypes.h, 1033
TenVec
declare.h, 642
opera.c, 1388
TenVecFind
declare.h, 642
opera.c, 1388
TePos
R_const, 203
terfirstcomm
N_const, 136
TermAssign
declare.h, 723
dollar.c, 809
termbuf
O_const, 159
termbuffer
N_const, 138
TERMEXTRAWORDS
tools.c, 2291
TermFree
declare.h, 590
TERMFUNCTION
ftypes.h, 1016
termfunnum
M_const, 124
TERMINAL
pre.c, 1570
Terminate
declare.h, 655
startup.c, 2026
termlevel
C_const, 41
TermMalloc
declare.h, 589
TermMallocAddMemory
declare.h, 753
tools.c, 2303
TermMemHeap
T_const, 244
TermMemMax
T_const, 247
TERMMEMSTARTNUM
tools.c, 2291
TermMemTop
T_const, 247
termp
TrAcEn, 262
TrAcEs, 264
TermRenumber
declare.h, 642
store.c, 2054
terms
poly, 191
Terms2InSmall
sOrT, 222
termscopy
poly, 191
TERMSINBRACKET
ftypes.h, 1018
TermsInBracket
declare.h, 739
execute.c, 865
termsinbracket
BrAcKeTiNdEx, 11
TermsInDollar
declare.h, 735
dollar.c, 811
TERMSINEXPR
ftypes.h, 1017
TermsInExpression
declare.h, 735

execute.c, 865
TermsInSmall
sOrT, 222
termsize
NeStInG, 154
TermsLeft
sOrT, 222
termsortstack
C_const, 24
TERMSSMALL
fsizes.h, 968
termstack
C_const, 24
termsumcheck
C_const, 26
terstart
N_const, 136
terstop
N_const, 135
TestArgNum
declare.h, 755
transform.c, 2356
TestDoLoop
declare.h, 765
dollar.c, 813
TestDrop
execute.c, 863
TestEndDoLoop
declare.h, 765
dollar.c, 813
TestFunFlag
declare.h, 746
normal.c, 1307
TestMatch
declare.h, 643
pattern.c, 1528
TestName
declare.h, 660
names.c, 1268
TestPartitions
declare.h, 607
reshuf.c, 1847
TestSelect
declare.h, 733
pattern.c, 1529
TestSub
declare.h, 643
proces.c, 1667
TestTables
compiler.c, 500
TestTerm
declare.h, 754
tools.c, 2307
TestUse
declare.h, 609
tables.c, 2182
TestValue
C_const, 29
TeSuOut
N_const, 146
TEXPRESSION
ftypes.h, 1001
TFUN
ftypes.h, 1071
TFUN1
ftypes.h, 1071
TFUNCLOSE
ftypes.h, 1003
TFUNCTION
ftypes.h, 1000
TFUNOPEN
ftypes.h, 1002
TGENINDEX
ftypes.h, 1004
TheDefine
declare.h, 670
pre.c, 1573
TheErrorMessage
ratio.c, 1744
thefunction
SHvariables, 215
theposition
N_const, 134
THETA
ftypes.h, 1015
THETA2
ftypes.h, 1015
TheUndefine
declare.h, 671
pre.c, 1575
ThreadBalancing
C_const, 35
ThreadBucketSize
C_const, 30
threads.c, 2211
THREADSCRATCHOUTSIZE
fsizes.h, 971
THREADSCRATCHSIZE
fsizes.h, 971
THREADSDEBUG
ftypes.h, 994
ThreadsFlag
C_const, 35
ThreadSortFileSynch
C_const, 35
ThreadStats
C_const, 35
TimeChildren
declare.h, 645
tools.c, 2306
TimeCPU
declare.h, 645
tools.c, 2306
timeout
X_const, 274
TimeWallClock

declare.h, 645
tools.c, 2306
TINDEX
ftypes.h, 1000
tlistbuf
N_const, 140
tlistsize
N_const, 144
TMaddr
T_const, 244
TMbuff
T_const, 250
TMdolfac
T_const, 250
TMINUS
ftypes.h, 1003
TMout
T_const, 250
TMPPOLYFUN
ftypes.h, 1011
TMULTIPLY
ftypes.h, 1003
TNOT
ftypes.h, 1003
TNUMBER
ftypes.h, 1001
TNUMBER1
ftypes.h, 1004
TOBEFACTORED
ftypes.h, 1048
ToBeInFactors
C_const, 45
TOBEUNFACTORED
ftypes.h, 1048
TODISK
minos.h, 1193
ToFast
declare.h, 668
tools.c, 2305
ToGeneral
declare.h, 667
tools.c, 2305
tohunt
N_const, 142
token.c, 2262, 2265
CHECKPOLY, 2263
simp1token, 2263
simp2token, 2264
simp3atoken, 2264
simp3btoken, 2264
simp4token, 2264
simp5token, 2264
simp6token, 2265
simpwtoken, 2264
tokenize, 2263
ttypes, 2265
WriteTokens, 2263
tokenarglevel
C_const, 27
tokenize
declare.h, 718
token.c, 2263
tokens
C_const, 27
TokensWriteFlag
C_const, 33
TOKENTOLINE
declare.h, 575
TokenToLine
declare.h, 646
sch.c, 1888
TOLONG
declare.h, 587
TOLYNDON
ftypes.h, 1063
TOLYNDONR
ftypes.h, 1063
tools.c, 2289, 2308
AddToString, 2299
AreArgsEqual, 2305
CacheNumberMallocAddMemory, 2303
CloseChannel, 2298
CloseFile, 2295
CloseStream, 2293
CompareArgs, 2305
CompArg, 2306
ConWord, 2299
CopyFile, 2295
Crash, 2307
CreateFile, 2295
CreateHandle, 2295
CreateLogFile, 2295
CreateStream, 2294
DODOUBLE, 2292
DOEXPAND, 2292
DoubleBuffer, 2304
DoubleList, 2304
DoubleLList, 2304
EndOfToken, 2300
ExpandBuffer, 2304
filelist, 2308
filelistsize, 2308
FILLVALUE, 2291
FlushFile, 2297
From0List, 2303
FromList, 2303
FromVarList, 2303
GetAppendChannel, 2298
GetChannel, 2297
GetFromStream, 2293
GetPosFile, 2297
GetStreamPosition, 2294
iexp, 2304
iniTools, 2302
IsLikeVector, 2305
LoadInputFile, 2293

LocateFile, 2293
LongCopy, 2301
LongLongCopy, 2301
LookInStream, 2293
M_check1, 2302
M_free, 2302
M_print, 2303
MakeDate, 2301
Malloc, 2302
Malloc1, 2302
NUMBEREXTRAWORDS, 2292
NumberMallocAddMemory, 2303
NUMBERMEMSTARTNUM, 2292
NumCopy, 2300
numinfilelist, 2308
NumToStr, 2299
OpenAddFile, 2294
OpenFile, 2294
OpenStream, 2293
PositionStream, 2294
ReadFile, 2296
ReadFromStream, 2293
ReadPosFile, 2296
ReadSnum, 2300
ReOpenFile, 2295
ReverseStatements, 2294
SeekFile, 2296
set_del, 2301
set_in, 2301
set_set, 2301
set_sub, 2302
SetPosFile, 2297
SkipField, 2300
StartFiles, 2294
StrCmp, 2298
strDup1, 2300
StrHICmp, 2298
StrICmp, 2298
StrICont, 2298
StrLen, 2299
SynchFile, 2297
TELLFILE, 2296
TellFile, 2296
TERMEXTRAWORDS, 2291
TermMallocAddMemory, 2303
TERMMEMSTARTNUM, 2291
TestTerm, 2307
TimeChildren, 2306
TimeCPU, 2306
TimeWallClock, 2306
ToFast, 2305
ToGeneral, 2305
ToPolyFunGeneral, 2305
ToToken, 2300
TruncateFile, 2297
UpdateMaxSize, 2298
WriteFile, 2308
WriteFileToFile, 2296
WriteString, 2299
WriteUnfinString, 2299
TOOUTPUT
ftypes.h, 996
Top
CbUf, 47
top
StreaM, 233
TOPBITONLY
form3.h, 948
TOPLONGBITONLY
form3.h, 949
topnumber
dbase, 53
ToPolyFunGeneral
declare.h, 668
tools.c, 2305
TOPOLYNOMIALFLAG
ftypes.h, 1065
topolynomialflag
C_const, 38
toppos
StOrEcAcHe, 227
topsubexpbuffers
compiler.c, 502
toptokens
C_const, 27
ToStorage
declare.h, 646
store.c, 2053
totalnumberofthreads
M_const, 116
toterms
sort.c, 1962
totind
TaBlEs, 257
ToToken
declare.h, 682
tools.c, 2300
TPLUS
ftypes.h, 1003
TPOWER
ftypes.h, 1003
TPOWER1
ftypes.h, 1004
Trace4
declare.h, 646
opera.c, 1386
Trace4Gen
declare.h, 646
opera.c, 1386
Trace4no
declare.h, 646
opera.c, 1386
tracebackflag
M_const, 123
TraceFind
declare.h, 647

opera.c, 1387
TRACEN
structs.h, 2122
TrAcEn, 261
accu, 262
accup, 262
allsign, 263
factor, 263
inlist, 262
level, 262
num, 262
perm, 262
sgn, 262
termp, 262
TraceN
declare.h, 647
opera.c, 1387
TraceNgen
declare.h, 647
opera.c, 1387
TraceNno
declare.h, 647
opera.c, 1387
TRACES
structs.h, 2122
TrAcEs, 263
a3, 267
a4, 267
accu, 264
accup, 264
ad, 267
allsign, 268
e3, 265
e4, 265
eers, 265
factor, 268
finalstep, 268
gamm, 267
gamma5, 268
inlist, 264
j3, 265
j4, 265
kstep, 266
lc3, 267
lc4, 267
level, 268
mdel, 266
mdum, 266
mepf, 265
nt3, 265
nt4, 265
num, 268
pdel, 266
pepf, 266
perm, 264
sgn, 266
sign1, 267
sign2, 267
stap, 266
step1, 266
termp, 264
Traces
declare.h, 647
opera.c, 1387
tracestack
N_const, 138
TransEname
C_const, 29
TransferBuffer
declare.h, 733
pattern.c, 1530
transFlag
O_const, 164
transform.c, 2353, 2357
CoTransform, 2353
FindRange, 2357
PutArgInScratch, 2356
ReadRange, 2356
RunAddArg, 2355
RunCycle, 2355
RunDecode, 2354
RunDedup, 2355
RunDropArg, 2356
RunEncode, 2354
RunExplode, 2354
RunImplode, 2354
RunIsLyndon, 2355
RunMulArg, 2355
RunPermute, 2354
RunReplace, 2354
RunReverse, 2355
RunSelectArg, 2356
RunToLyndon, 2356
RunTransform, 2353
TestArgNum, 2356
TransformRational
declare.h, 767
dict.c, 790
TranslateExpression
declare.h, 735
dollar.c, 812
TRAPSIGNALS
unix.h, 2397
TreatPolyRatFun
declare.h, 748
normal.c, 1306
tree, 269
blnce, 270
left, 269
parent, 269
right, 269
sOrT, 220
usage, 270
value, 269
Trick
declare.h, 647

opera.c, 1386
TrimPower
T_const, 249
TRUE_EXPR
pre.c, 1570
TruncateFile
declare.h, 663
tools.c, 2297
TryDo
declare.h, 648
reshuf.c, 1847
TryEnvironment
setfile.c, 1926
TryFileSetups
setfile.c, 1926
TryRecover
declare.h, 575
tryterm
N_const, 148
TSET
ftypes.h, 1000
TSETCLOSE
ftypes.h, 1004
TSETDOL
ftypes.h, 1005
TSETNUM
ftypes.h, 1005
TSETOPEN
ftypes.h, 1004
TSGAMMA
ftypes.h, 1005
TSUBEXP
ftypes.h, 1001
TSYMBOL
ftypes.h, 1000
ttypes
token.c, 2265
TVECTOR
ftypes.h, 1000
TWILDARG
ftypes.h, 1002
TWILDCARD
ftypes.h, 1002
TwoExprCompare
declare.h, 735
dollar.c, 812
type
ARGBUFFER, 8
DICTIONARY_ELEMENT, 57
DoLlArS, 61
DoLoOp, 64
FaCdOlLaR, 71
fixedset, 82
InDeX, 90
KEYWORD, 100
MoDoPtDoLlArS, 130
NaMeNode, 150
PF_BUFFER, 188
SeTs, 211
SETUPPARAMETERS, 213
sOrT, 223
StreaM, 234
TYPEADJUSTBOUNDS
ftypes.h, 1029
TYPEAPPLY
ftypes.h, 1030
TYPEAPPLYRESET
ftypes.h, 1030
TYPEARG
ftypes.h, 1026
TYPEARGEXPLODE
ftypes.h, 1031
TYPEARGHEADSIZE
ftypes.h, 1037
TYPEARGIMPLODE
ftypes.h, 1031
TYPEARGTOEXTRASYMBOL
ftypes.h, 1032
TYPEASSIGN
ftypes.h, 1028
TYPECHAININ
ftypes.h, 1030
TYPECHAINOUT
ftypes.h, 1030
TYPECHISHOLM
ftypes.h, 1026
TYPECOUNT
ftypes.h, 1025
TYPEDENOMINATORS
ftypes.h, 1031
TYPEDETCURDUM
ftypes.h, 1029
TYPEDISCARD
ftypes.h, 1025
TYPEDOLOOP
ftypes.h, 1032
TYPEDROPCOEFFICIENT
ftypes.h, 1031
TYPEDROPSYMBOLS
ftypes.h, 1032
TYPEELIF
ftypes.h, 1026
TYPEELSE
ftypes.h, 1026
TYPEENDDOLOOP
ftypes.h, 1032
TYPEENDIF
ftypes.h, 1026
TYPEENDREPEAT
ftypes.h, 1025
TYPEEXIT
ftypes.h, 1027
TYPEEXPRESSION
ftypes.h, 1024
TYPEFACTARG
ftypes.h, 1028

TYPEFACTARG2
ftypes.h, 1028
TYPEFACTOR
ftypes.h, 1031
TYPEFINDLOOP
ftypes.h, 1029
TYPEFPRINT
ftypes.h, 1027
TYPEFROMPOLYNOMIAL
ftypes.h, 1032
TYPEGOTO
ftypes.h, 1025
TYPEIDNEW
ftypes.h, 1024
TYPEIDOLD
ftypes.h, 1025
TYPEIF
ftypes.h, 1026
TYPEINEXPRESSION
ftypes.h, 1029
TYPEINSIDE
ftypes.h, 1029
TYPEISFUN
ftypes.h, 1005
TYPEISMYSTERY
ftypes.h, 1005
TYPEISSUB
ftypes.h, 1005
TYPEMERGE
ftypes.h, 1030
TYPEMULT
ftypes.h, 1025
TYPENORM
ftypes.h, 1027
TYPENORM2
ftypes.h, 1027
TYPENORM3
ftypes.h, 1027
TYPENORM4
ftypes.h, 1031
TYPEOPERATION
ftypes.h, 1025
TYPEPRINT
ftypes.h, 1027
TYPEPUTINSIDE
ftypes.h, 1032
TYPEREDEFPRE
ftypes.h, 1027
TYPERENUMBER
ftypes.h, 1028
TYPEREPEAT
ftypes.h, 1025
TYPEREVERSE
ftypes.h, 1026
TYPESETEXIT
ftypes.h, 1027
TYPESORT
ftypes.h, 1029
TYPESPLITARG
ftypes.h, 1028
TYPESPLITARG2
ftypes.h, 1028
TYPESPLITFIRSTARG
ftypes.h, 1030
TYPESPLITLASTARG
ftypes.h, 1030
TYPESTUFFLE
ftypes.h, 1031
TYPESUM
ftypes.h, 1026
TYPESUMFIX
ftypes.h, 1028
TYPETERM
ftypes.h, 1029
TYPETESTUSE
ftypes.h, 1030
TYPETOPOLYNOMIAL
ftypes.h, 1032
TYPETOSPECTATOR
ftypes.h, 1032
TYPETRANSFORM
ftypes.h, 1031
TYPETRY
ftypes.h, 1028
TYPEUNRAVEL
ftypes.h, 1029

UBYTE
form3.h, 955
Uclose
form3.h, 952
Uflush
form3.h, 952
Ugetpos
form3.h, 953
UINT
form3.h, 955
uncompressed
objects, 167
UngetChar
declare.h, 575
UngetFromStream
declare.h, 575
UNHIDE
ftypes.h, 1038
UNHIDEGEXPRESSION
ftypes.h, 1040
UNHIDELEXPRESSION
ftypes.h, 1040
unix.h, 2396, 2398
ALTSEPARATOR, 2398
CARRIAGERETURN, 2397
LINEFEED, 2397
P_term, 2397
PATHSEPARATOR, 2398
SEPARATOR, 2398
TRAPSIGNALS, 2397

WITH_ENV, 2398
WITHEXTERNALCHANNEL, 2397
WITHPIPE, 2397
WITHSYSTEM, 2397
unixfile.c, 2399
UNLOCK
declare.h, 592
UNPACK
ftypes.h, 1060
UnPack
declare.h, 648
reken.c, 1789
UNPACK_LONG
parallel.c, 1419
UNRWLOCK
declare.h, 592
UnsetAllowDelay
pre.c, 1571
UnSetDictionary
declare.h, 766
dict.c, 792
UnSetMods
declare.h, 651
execute.c, 863
UnsureDollarMode
C_const, 34
Uopen
form3.h, 952
UpdateMaxSize
tools.c, 2298
UpdatePositions
execute.c, 865
Uread
form3.h, 952
usage
tree, 270
used
sOrT, 218
USEDFLAG
ftypes.h, 1046
UsedFunction
N_const, 137
UseDictionary
dict.c, 792
UsedIndex
N_const, 136
UsedOtherFind
N_const, 142
UsedSymbol
N_const, 136
UsedVector
N_const, 136
Useek
form3.h, 953
UseFindOnly
N_const, 142
Usetbuf
form3.h, 953
Usetpos
form3.h, 953
Ustdout
form3.h, 954
Usync
form3.h, 953
Utell
form3.h, 953
Utruncate
form3.h, 954
Uwrite
form3.h, 952

va_arg
sch.c, 1887
va_dcl
sch.c, 1886
va_end
sch.c, 1886
va_list
sch.c, 1887
va_start
sch.c, 1886
value
FaCdOlLaR, 72
pReVaR, 195
SETUPPARAMETERS, 213
tree, 269
variable.h, 2402, 2413
A, 2412
AC, 2410
AM, 2410
AN, 2410
AO, 2411
AP, 2411
AR, 2411
AS, 2411
AT, 2411
autofunctions, 2409
autoindices, 2409
autosymbols, 2409
autovectors, 2409
AX, 2411
BUG, 2410
cbuf, 2408
channels, 2408
chartype, 2404
Dollars, 2408
DoLoops, 2404
Dubious, 2408
Expressions, 2409
FG, 2412
fixedsets, 2412
functions, 2405
GlobalFunctions, 2407
GlobalIndices, 2407
GlobalSetElements, 2408
GlobalSets, 2407
GlobalSymbols, 2407

GlobalVectors, 2407
indices, 2406
MaxDoLoops, 2405
MaxNumPre, 2405
MaxProcedures, 2404
ModOptdollars, 2410
NumDollars, 2408
NumDoLoops, 2405
NumDubious, 2408
NumExpressions, 2409
NumFunctions, 2406
NumIndices, 2406
NumModOptdollars, 2410
NumOutputChannels, 2408
NumPotModdollars, 2410
NumPre, 2405
NumProcedures, 2404
NumSetElements, 2407
NumSets, 2407
NumSymbols, 2406
NumTableBases, 2407
NumVectors, 2406
numxsymbol, 2409
PotModdollars, 2410
PreVar, 2405
Procedures, 2404
SetElements, 2405
Sets, 2405
setupfilename, 2412
symbols, 2406
tablebases, 2406
vectors, 2406
WriteFile, 2412
xsymbol, 2409
variables
DICTIONARY, 56
InDeXeNtRy, 94
VARNAME
declare.h, 583
VARNAMES
ftypes.h, 1061
varnames
C_const, 20
VARRENUM
structs.h, 2121
VaRrEnUm, 270
hi, 271
lo, 271
start, 271
vars
DoLoOp, 63
VarStore
declare.h, 648
store.c, 2054
VarType
FixedGlobals, 80
VARTYPECOMPLEX
ftypes.h, 1048
VARTYPEIMAGINARY
ftypes.h, 1048
VARTYPEMINUS
ftypes.h, 1049
VARTYPENONE
ftypes.h, 1048
VARTYPEROOTOFUNITY
ftypes.h, 1048
vecnum
ReNuMbEr, 208
vect
ReNuMbEr, 207
VECTBIT
ftypes.h, 1055
VECTOMIN
ftypes.h, 1034
VECTOR
ftypes.h, 1009
VeCtOr, 271
complex, 272
dimension, 272
flags, 272
name, 272
namesize, 272
node, 272
number, 272
Vector
vector.h, 2416
vector.h, 2414, 2423
DeclareVector, 2416
Vector, 2416
VectorBack, 2418
VectorCapacity, 2419
VectorClear, 2420
VectorEmpty, 2419
VectorErase, 2423
VectorErases, 2423
VectorFree, 2417
VectorFront, 2418
VectorInit, 2417
VectorInsert, 2422
VectorInserts, 2422
VectorPopBack, 2421
VectorPtr, 2418
VectorPushBack, 2420
VectorPushBacks, 2421
VectorReserve, 2420
VectorSize, 2419
VectorStruct, 2416
VECTOR_
ftypes.h, 1044
VectorBack
vector.h, 2418
VectorCapacity
vector.h, 2419
VectorClear
vector.h, 2420
VectorEmpty

vector.h, 2419
VectorErase
vector.h, 2423
VectorErases
vector.h, 2423
VectorFree
vector.h, 2417
VectorFront
vector.h, 2418
VectorInit
vector.h, 2417
VectorInsert
vector.h, 2422
VectorInserts
vector.h, 2422
vectorlikeLHS
C_const, 39
VectorList
C_const, 21
VECTORONLY
ftypes.h, 997
VectorPopBack
vector.h, 2421
VectorPtr
vector.h, 2418
VectorPushBack
vector.h, 2420
VectorPushBacks
vector.h, 2421
VectorReserve
vector.h, 2420
Vectors
C_const, 23
vectors
variable.h, 2406
VectorSize
vector.h, 2419
VectorStruct
vector.h, 2416
VECTOSUB
ftypes.h, 1034
VECTOVEC
ftypes.h, 1034
VERSIONSTR
startup.c, 2024
VERSIONSTR__
startup.c, 2024
vetofilling
C_const, 37
vetotablebasefill
C_const, 37
vflags
ExPrEsSiOn, 69
VOID
form3.h, 954
ftypes.h, 996
VORTRANMODE
ftypes.h, 1006
WantAddLongs
declare.h, 588
WantAddPointers
declare.h, 588
WantAddPositions
declare.h, 589
WARMUP
reken.c, 1789
WarnFlag
C_const, 34
Warning
declare.h, 651
message.c, 1154
WCN
structs.h, 2125
WCN2
structs.h, 2125
WCOPY
declare.h, 577
where
DoLlArS, 60
FaCdOlLaR, 71
PaRtI, 184
TaBlEbAsEsUbInDeX, 254
whichbuffer
ExPrEsSiOn, 70
WhileLevel
C_const, 43
wildarg
pReVaR, 195
WILDARGFUN
ftypes.h, 1018
WILDARGINDEX
ftypes.h, 1024
WildArgs
N_const, 146
WILDARGSYMBOL
ftypes.h, 1023
WildArgTaken
T_const, 243
WILDARGVECTOR
ftypes.h, 1024
WildC
C_const, 25
wildcard.c, 2427, 2429
AddWild, 2428
CheckWild, 2428
ClearWild, 2428
DEBUG, 2427
DenToFunction, 2428
ResolveSet, 2428
WildFill, 2428
WildcardBufferSize
C_const, 32
T_const, 247
WildcardNames
C_const, 26
WILDCARDS

ftypes.h, 1035
WildDirt
N_const, 145
WildDollars
declare.h, 723
dollar.c, 809
WILDDUMMY
ftypes.h, 1033
WildEat
N_const, 146
WildFill
declare.h, 648
wildcard.c, 2428
wildflag
C_const, 32
WILDMASK
form3.h, 949
WildMask
T_const, 244
WILDOFFSET
fsizes.h, 963
wildoffset
STOREHEADER, 231
WildReserve
N_const, 145
WildStop
N_const, 135
WildValue
N_const, 135
WilInd
M_const, 121
WITH_ENV
fwin.h, 1110
unix.h, 2398
WITHAUTO
ftypes.h, 997
WITHERROR
ftypes.h, 987
WITHEXTERNALCHANNEL
unix.h, 2397
WITHOUTERROR
ftypes.h, 987
withoutflush
minos.c, 1174
minos.h, 1195
WITHOUTSEMICOLON
ftypes.h, 992
WITHPIPE
unix.h, 2397
WITHRETURN
fwin.h, 1110
WITHSEMICOLON
ftypes.h, 991
WITHSORTBOTS
form3.h, 951
WITHSYSTEM
fwin.h, 1110
unix.h, 2397
wlen
O_const, 162
WORDDIF
declare.h, 582
WORDMASK
form3.h, 949
WORKBUFFER
fsizes.h, 964
WorkPointer
T_const, 242
WorkSize
M_const, 112
WorkSpace
T_const, 242
WorkTop
T_const, 242
wpoin
O_const, 159
wpos
O_const, 159
wranf
declare.h, 747
reken.c, 1799
wranfcall
R_const, 202
wranfia
R_const, 201
wranfnpair1
R_const, 202
wranfnpair2
R_const, 202
wranfseed
R_const, 202
WriteAll
sch.c, 1890
WriteArgument
declare.h, 649
sch.c, 1889
WRITEBUFTOEXTCHANNEL
declare.h, 594
WriteDictionary
declare.h, 767
sch.c, 1889
WriteDollarFactorToBuffer
declare.h, 723
dollar.c, 809
WriteDollarToBuffer
declare.h, 722
dollar.c, 809
WriteExpression
declare.h, 649
sch.c, 1890
WRITEFILE
declare.h, 594
WriteFile
tools.c, 2308
variable.h, 2412
WriteFileToFile

declare.h, 662
tools.c, 2296
WriteIndex
declare.h, 740
minos.c, 1170
WriteIndexBlock
declare.h, 740
minos.c, 1170
WriteIniInfo
declare.h, 741
minos.c, 1171
WriteInnerTerm
declare.h, 649
sch.c, 1890
WriteLists
sch.c, 1889
WriteNamesBlock
declare.h, 740
minos.c, 1170
WriteObject
declare.h, 742
minos.c, 1173
WriteOne
declare.h, 648
sch.c, 1890
WRITEOUT
ftypes.h, 994
WriteSetup
setfile.c, 1925
WriteStats
declare.h, 649
sort.c, 1949
WriteStoreHeader
declare.h, 751
store.c, 2056
WriteString
declare.h, 679
tools.c, 2299
WriteSubTerm
declare.h, 650
sch.c, 1890
WriteTerm
declare.h, 650
sch.c, 1890
writeToChannel
declare.h, 746
pre.c, 1584
WriteTokens
declare.h, 718
token.c, 2263
WriteUnfinString
declare.h, 679
tools.c, 2299
WrtPower
declare.h, 597
sch.c, 1889
wsizeof
declare.h, 583
WTimeStatsFlag
C_const, 38

X
AllGlobals, 7
X_const, 273
currentExternalChannel, 274
currentPrompt, 273
daemonize, 274
killSignal, 274
killWholeGroup, 274
shellname, 273
stderrname, 273
timeout, 274
xsymbol
variable.h, 2409

YESLYNDON
ftypes.h, 1064

Z_
ftypes.h, 1044
zbufnum
M_const, 114
zero
DoLIArS, 61
ZEROARG
declare.h, 580
ZeroFillRange
declare.h, 582
zeropos
M_const, 107
zerorhs
M_const, 127