AVZ_Package: This notebook contains our procedures and functions created in process of preparation of the following books published in Belarus, Estonia, Ukraine and USA, namely:

1. *Aladjev V.Z., Vaganov V.A. Modular programming: Mathematica vs Maple, and vice versa.– USA, CA: Palo Alto, Fultus Corporation, 2011, ISBN 9781596822689, 418 p.*

2. *Aladjev V.Z., Bezrukavyi A.S., Haritonov V.N., Hodakov V.E. Programming: System Maple or Mathematica?– Ukraine: Kherson, Oldi–Plus Press, 2011, ISBN 9789662393460, 474 p.*
3. *Aladjev V.Z., Boiko V.K., Rovba E.A. Programming in system Mathematica and Maple: A Comparative Aspect.– Belarus: Grodno, Grodno State University, 2011, ISBN 9789855154816, 517 p.*
4. *Aladjev V.Z., Grinn D.S., Vaganov V.A. The extended functional means for system Mathematica.– Ukraine: Kherson: Oldi–Plus Press, 2012, ISBN 9789662393590, 404 p.*
5. *Aladjev V.Z., Grinn D.S. Extension of functional environment of system Mathematica.– Ukraine: Kherson: Oldi–Plus Press, 2012, ISBN 9789662393729, 552 p.*
6. *Aladjev V.Z., Grinn D.S., Vaganov V.A. The selected system problems in Mathematica software.– Ukraine: Kherson, Oldi–Plus Press, 2013, ISBN 9789662890129, 556 p.*
7. *Aladjev V.Z. Classical Cellular Automata: Mathematical Theory and Applications.– Germany: Saarbrucken, Scholar`s Press, 2014, ISBN 9783639713459, 517 p.*
8. *Aladjev V.Z., Vaganov V.A. Extension of the Mathematica system functionality.– USA: Seattle, CreateSpace, An Amazon.com Company, 2015, ISBN–13: 9781514237823, ISBN–10: 1514237822, 590 p.*
9. *Aladjev V.Z., Vaganov V.A. Toolbox for the Mathematica programmers.- USA: Seattle, CreateSpace, 2016, ISBN 9781532748837, 630 p.*

This file has been created, and its contents has been tested in the system Mathematica 8.0.0.0 – 10.4.0.0 on PC MicroLink 500 with Windows XP Professional (Version 5.1, Build 2600, Service Pack 3) and on PC Dell OptiPlex 3020 with Windows 7 Professional (Version 6.1.7601, Build 7601, Service Pack 1) during January 2013 – April 2014 and October 2014 – May 2016.
Help represented here is brief, the detailed enough description of software represented in this notebook along with the most typical examples of its usage can be found in the above books. The above books [2, 4–6] can be ordered on website
http://oldiplus.com/catalog/programuvannya-kompyuterni-nauki/; aladjev@yandex.ru; valadjev@yahoo.com

This notebook contains definitions of the following procedures and functions grouped according to their basic functional purpose (for review of the list of the tools of the corresponding group that are contained in the notebook, it is necessary to select the relevant cell (cells) and to expand it (them) by the commands chain of the GUI "Cell –> Convert To –> TextDisplay"), namely:

1. Software for work with string structures and symbols: Border, CharacterQ, Characters1, CorrectSubString, DelSubStr, DelSuffPref, ExprOfStr, ExprOfStr1, ExprQ, ExtrExpr, ExtrName, ExtrVarsOfStr, InsertN, IsMonotonic, IsPermutation, LeftFold, ListStrToStr, LongestCommonSubSequence, LongestCommonSubsequence1, LongestCommonSubsequence2, LongestCommonSubString, MaximalPalindromicSubstring, OverLap, PrefCond, PrefixQ, PrevNextVar, RedSymbStr, ReduceAdjacentStr, RightFold, SortQ, SortRevStr, SortString, Spos, StandStrForm, StrAllSymbNumQ, StrDelEnds, StrExprQ, StrFromStr, StringToList, StrOfSymb, StrToList, StringDependAllQ, StringDependQ, StringDependQ1, StringJoin1, StringLevels, StringMultiple, StringMultiple1, StringPosition1, StringReplace1, StringReplace2, StringReplace3, StringReplaceS, StringReplacePart1, StringSplit1, StringSplit2, StringStringQ, StringTake1, StringTake2, StringTake3, StringTrim1, StringTrim2, StrOfSymblQ, StrPartition, StrStr, StrSub, StrSymbParity, SubCfEntries, SubDelStr, SubsBstr, SubsDel, SubsPosSymb, SubsStr, SubsString, SubsString1, SubsStrLim, SubsStrLim1, SubStr, SubStrSymbolParity, SubStrSymbolParity1, SubStrToSymb, SuffixQ, SuffPref, SymbolQ, SymbolQ1, SymbolsOfString, ToString1, ToString2, ToString3, ToString4

2. Software for work with list structures: ArrayInd, AssignL, AssignToList, BinaryListQ, DelEl, ElemLevelsL, ElemLevelsN, ElemOnLevels, ElemsOnLevelList, ElemsList, FirstPositionList, FullNestListQ, GroupIdentMult, IntegerListQ, LevelsOfList, LevelsList, ListAppValue, ListAssign, ListAssignP, ListCount, ListExprHeadQ, ListLevels, ListListGroup, ListListQ, ListNumericQ, ListOp, ListPosition, ListRulesQ, ListTrim, ListsAssign, ListStringQ, ListStrList, ListSymbolQ, ListToSeq, ListToString, MapNestList, MaxLevel, MaxNestLevel, MemberQL, MinusList, MinusList1, MinusLists, MultEntryList, NestListQ, NestListQ1, NestQL, ParVar, PosIntListQ, PosListTest, PositionsListCond, PosSubList, ReplaceLevelList, ReplaceListCond, ReduceList, ReplaceSubLists, RestListList, RestListList1, SelectPos, SortLpos, SortNestList, SortNL, SortNL1, Split1, SplitList, SplitList1, SubLists, SubListsMin, SubsList, SymbolToList, Table1, ToList, TransListList

3. Software for work with sequence structures: SEQ, SeqDel, SeqIns, SeqQ, SeqToList, SeqToList1, SeqToString, SequenceQ, Sequences, SeqUnion, Sq

4. Software for work with functions and procedures of the types "Module" and " Block": $AobjNobj, $CallProc, $InBlockMod, $ProcName, $ProcType, $TestArgsTypes, $TypeProc, ActBFMuserQ, ActCsProcFunc, AllCalls, Args, Args1, ArgsBFM, ArgsBFM1, ArgsTypes, Arity, Arity1, ArityBFM, ArityBFM1, AritySystemFunction, AritySystemFunction1, AttrOpts, Avg, BFMSubsQ, BlockFuncModQ, BlockFuncModVars, BlockModQ, BlockQ, BlockQ1, BlockToModule, CallListable, CallsInProc, ClearAllAttributes, CompActPF, CompActPF1, CompileFuncQ, ContentObj, ContextsInModule, CsProcsFuncs, CsProcsFuncs1, Default1, Defaults, DefaultsM, DefaultsQ, DefaultValues1, DefaultValues2, DefFunc, DefFunc3, DefOpt, DefOpt1, DefOptimum, DelDuplLocals, DelDuplLocalsM, DelRestPF, DelRestPF1, DuplicateLocals, DuplicateLocalsQ, ExpArgs, ExpLocals, ExtensionHeading, ExtrCall, ExtrNames, ExtrProcFunc, FullUserTools, FullToolsCalls, FullToolsCallsM, FuncBlockModQ, FunCompose, FunctionQ, FuncToPure, Globals, Globals1, Globals2, GlobalToLocal, GlobalToLocalM, GotoLabel, HeadingQ, HeadingQ1, HeadingQ2, HeadingQ3, Headings, HeadingsPF, HeadName, HeadName1, HeadPF, HeadToCall, Locals, Locals1, Locals2, Locals3, LocalsGlobals, LocalsGlobals1, LocalsGlobalsM, MdP, ModToPureFunc, ModuleQ, ModuleQ1, ModuleQ2, MultipleArgsQ, NamesProc, NotSubsProcs, OptDefinition, OptimalLocals, OptimalLocalsM, OptimalLocalsN, OptRes, PartProc, ProcActCallsQ, ProcBMQ, ProcBody, ProcCalls, ProcCalls1, ProcContent, ProcFuncBlQ, ProcFuncBlQ1, ProcFuncCS, ProcFuncTypeQ, ProcQ, ProcQ1, ProcLocals, ProcsAct, ProcUniToMod, PureFuncQ, PureToFunc, ReplaceSubProcs, QBlockMod, QFunction, QFunction1, QmultiplePF, RealProcQ, RedundantLocals, RedundantLocalsM, RemovePF, ReplaceLocals, RemProcOnHead, RemProcOnHead1, RenameH, RenBlockFuncMod, RenBlockFuncMod1, ReplaceProc, ReduceArgs, ReplaceProcBody, ScanLikeProcs, SingleDefQ, ShortPureFuncQ, StandHead, StrNestedMod, StructProcFunc, SubProcs, SubProcs1, SubProcs2, SubProcs3, SubsProcQ, SubsProcs, SyntCorProcQ, SysFuncQ, SysFuncQ1, SysUserSoft, TestArgsCall, TestArgsLocals, TestArgsTypes, TestArgsTypes1, TestArgsTypes2, TestDefBFM, TestBFM, TestFactArgs, TestHeadingQ, TestProcCalls, ToDefOptPF, TrueCallQ, UprocQ, Uprocs, VarsInBlockMod

5. Software for work with packages and documents files of the types "m"," mx ", and " nb: $UserContext, AcNb, ActivateMeansFromCdfNb, ActUcontexts, AddMxFile, AllContexts, AllCurrentNb, Aobj, Aobj1, CALL, CALLmx, CALLmxH, CallSave, CdfNbToText, CNames, CodeObjMfile, ContCodeUsageM, ContentOfMfile, ContentOfMfile1, ContentOfMfile2, ContentsCdfNb, ContentsMx, ContentOfNbCdf, ContextActQ, ContextsCS, ContextInFile, ContextFromFile, ContextFromM, ContextFromM1, ContextMfile, ContextMfile1, ContextMXfile, ContextInMxFile, ContextFromMx, ContextNBfile, ContextQ, ContextForPackage, ContextRepMx, ContextToFileName1, Contexts1, ContextsInFiles, ContextInMfile, ContMxFile, ContMxFile1, ContMxFile2, ContMxW7, ContOfContext, ConvertMtoMx, ContUsageMfile, CopyFile1, CurrentNb, CurrentPackageQ, DefFromM, DefFromPackage, DefInPackage, DefWithContext, DeletePackage, DelOfPackage, DiffContexts, DumpSaveP, EvaluateCdfNbFile, ExcessVarsPack, ExtrDefFromM, ExtrFromNBfile, ExtrFromNBfile1, ExtrFromNBfile2, ExtrFromNBfile3, ExtrFromM, ExtrFromMfile, ExtrOfMfile, ExtrPackName, FindFileContext, FindFileContext1, FileCurrentNb, FullCalls, FullCalls1, HelpBasePac, HelpPrint, IsPackageQ, LoadMyPackage, LoadNameFromM, LoadPackage, MathPackages, NbFileEvaluate, MfileEvaluate, MfileLoad, MfilePackageQ, MfileToMx, ModLibraryPath, MxFileToMfile, MxPackNames, MxToTxt, MxToTxt1, MxToTxt2, MxToMpackage, NamesContext, NamesCS, NamesFromMx, NamesFromMx1, NamesFromMx2, NamesMPackage, NamesNbPackage, NamesNbPackage1, NbCallProc, NbName, Need, Npackage, ObjInCurrentNb, ObjInCurrentNb1, OptDefPackage, OptimizedDefPackage, OptimizedMfile, PackageFileQ, PackageMxCont, PackageQ, Packages, PackNames, PackNames1, PackNames2, PackReplaceQ, PackageUsages, QSaveGUI, RedMfile, RedMxFile, ReloadPackage, ReloadPackage1, RemoveContext, RemovePackage, RestoreDelPackage, SymbolsContext, SymbolsFromMx, SystemPackages, TempInPack, TestCdfNbFile, ToContextPath, TypeWinMx, UpdateContextPaths, UpdatePackages, UpdatePath, UsagesCdfNb, UserPackTempVars, VizContentsNB, VizContext

6. Software expanding the standard functions of the Mathematica system or its software as a whole: $Line1, ActBFM, ActRemObj, Affiliate, AtomicQ, Attributes1, AttributesH, AttributesQ, BitGet1, Bits, BitSet1, CallQ, CallsInMean, CallsInMeansPackage, CatN, Clear1, ClearContextVars, ClearOut, ClearValues, ComplexQ, ContextDef, ContextSymbol, ContextToSymbol, ContextToSymbol1, ContextToSymbol2, CountOptions, CurrentNames, D1, Decomp, Def, Def1, DefAttributesH, Definition1, Definition2, Definition3, Definition4, DefOnHead, DefOp, DeleteOptsAttr, DefinedActSymbols, DefToString, Df, Df1, Df2, Diff, Difff, Diff1, Diff2, DO, DumpSave1, DumpSave2, EquExprPatt, EquExprPatt1, ExprComp, ExpressionQ, ExprOnLevels, ExprPatternQ, ExprsInStrQ, FormalArgs, FreeQ1, FreeQ2, FullFormF, Gather1, Gather2, GatherStrLetters, GC, GenRules, GenRules1, GenRules2, GroupNames, GV, Head1, Head2, Head3, HowAct, Iff, IFk, IFk1, Ind, Index, IndexedQ, IndexQ, Indices, Int, Int1, Integral1, Integral2, Integrate1, Integrate2, Intersection1, IntersectStrings, LangHoldFuncQ, Levels, ListableC, ListableQ, ListStrQ, LocObj, Map1, Map10, Map11, Map12, Map13, Map14, Map15, Map16, Map17, Map18, Map19, Map20, Map2, Map3, Map4, Map5, Map6, Map7, Map8, Map9, MainContexts, MapInSitu, MapInSitu1, MapInSitu2, Mapp, Mapp1, MaxParts, MemberLN, MemberQ1, MemberQ2, MemberQ3, MemberQ4, MemberT, MessagesOut, MixCaseQ, MultipleContexts, Names1, NbCurrentQ, NbDocumentQ, NotebookSave1, Nvalue, Nvalue1, ObjType, Op, OP, OpenCurrentNb, PalindromeQ1, PartialSums, PatternQ, PosIntQ, ProtectedQ, PureDefinition, Range1, Range2, Range3, Range4, Range5, RemovedQ, RemoveNames, Rename, Rename1, Replace1, Replace2, Replace3, Replace4, ReplaceAll1, ReplaceAll2, ReplaceOut, RepStandFunc, RestoreCS, RevRules, Riffle1, RhsLhs, RhsLhs1, Rule1, RuleQ, SaveCurrentSession, SaveInMx, SetAttributes1, SelectContains, SelectStrings, SortLS, Subs, Subs1, Subs1Q, Subs2, Substitution, Substitution1, StringPat, StringCases1, StringCases2, StringFreeQ1, StringFreeQ2, StringFreeQ3, StringTrim1, StringTrim2, SymbolGreater, SymbolLess, SyntaxLength1, SystemQ, SysFunctionQ, SystemSymbols, SysContexts, TemporaryQ, ToContext, ToStringRule, ToStringRule1, Tuples1, TwoHandQ, TypeActObj, UnDef, UndefinedQ, UnDefVars, UnDefVars1, UnevaluatedQ, UniqueV, VarExch, VarExch1, VarsValues, WhatObj, WhatType, WhatValue, WhichN

7. Software for work with file system of the computer: Adrive, Adrive1, Attrib, Attrib1, Attribs, BootDrive, BootDrive1, CDir, ClearRecycler, ClearRecyclerBin, Close1, Close2, CloseAll, Closes, CopyDir, CopyFileToDir, DelAllAttribs, DelDirFile, DelDirFile1, DeleteFile1, DirEmptyQ, DirFilePaths, DirsFiles, DirFD, DirFull, DirName, DirQ, EmptyFileQ, ExtProgExe, FileDirStForm, FileExistsQ1, FileFormat1, FileFormat2, FileFormat3, FileOpenQ, FileOpenQ1, FileQ, FilesDistrDirs, FindFile1, FindFile2, FindFileObject, FindSubDir, FreeSpaceVol, InOutFiles, IsFile, IsFileOpen, LoadExtProg, LoadFile, MathematicaDF, Memory, Nobj, OpenFileQ, OpenFiles, PathToFileQ, ReadFullFile, ReadFullFile1, RenDirFile, RestoreDelFile, Save1, Save2, SearchDir, SearchFile, SearchFile1, SetDir, SetDir1, SetPathSeparator, StandPath, StreamFiles, StreamsU, TypeFilesD, Ver, VolDir

8. Some tools of a special purpose: $Version1, $Version2, ClearCS, CFsequences, CodeEncode, CodeEncode1, ComposeGTF, Cost, FunctionToRules, GtfMod2, GtfMod2Q, HS, HSD, MinNCF, NcfQ, Nconcat, NfToLtf, OpSys, OSplatform, PCOS, Predecessors, PredecessorsL, PredecessorsR, ReductRes, ReprodHSD, ReprodHSM, ReprodHSM1, ReprodHSwVi, ReprodXOR1, ReprodXOR11, Restart, SelfReprod, SelfReprod1, SubConf, TabLib, ToLTF, Try, Un, Unique1, UsageBase, Usages, Usages1, UsagesMNb, UserLib, XOR1

In[1]:=

The given tools are grouped according to their basic functional purpose; meanwhile, this grouping to a considerable degree is conditional, because many of these tools are multifunctional, however even such classification allows to orientate oneself better in such multitude of the tools. Along with usages, the more detailed information on tools of the notebook can be found in the above books [1-6, 8,9] containing description of each tool, its source code and the most typical examples of its usage.

In[2]:=
```
BeginPackage["AladjevProcedures`"]

UprocQ::usage = "The call UprocQ[x] returns the False if x is not a procedure; otherwise,
      two−element list of the format {True, {\"Module\"|\"Block\"|\"DynamicModule\"}} is returned."

SymbolQ::usage = "The call SymbolQ[x] returns the True if x is a symbol; otherwise, the False is returned."

SymbolQ1::usage = "The call SymbolQ1[x] returns the True if x is a single symbol; otherwise, the False is returned."

SymbolsOfString::usage =
  "The procedure call SymbolsOfString[x] returns the list of the symbols which enter an expression represented by a string x. While
      the call SymbolsOfString[x, 1] returns the list of the symbols different from numbers which enter an expression represented
      by a string x. At last, the procedure call SymbolsOfString[x, a] where a – an expression different from 1, returns the the
      list of the symbols different from numbers and system symbols that enter an expression represented by a string x."

PrevNextVar::usage =
  "The PrevNextVar procedure for a special processing of the symbols of the format <symbol><integer> which end with
      an integer. The procedure call PrevNextVar[x, t] on a symbol x of the mentioned format <symbol><integer>
      returns the symbol of the format <symbol><integer – t> while the call PrevNextVar[x, t, h] where h – an arbitrary
      expression returns symbol of the format <symbol><integer + t>. At condition `integer – t <= 0` or in case of the
      format x different from the mentioned the source symbol x is returned. The previous fragment represents source
      code of the procedure with examples of its usage. The given procedure represents as independent interest, and is
      essentially used for solution of the problem of local variables in case of definition of the user traditional functions."

IsFile::usage = "The call IsFile[x] returns the True if x is a really existing datafile; otherwise, the False is returned."

FileQ::usage = "The call FileQ[x] returns the True if x is a really existing datafile; otherwise, the False is returned."

Rule1::usage =
  "The Rule1 procedure redefines the system Rule function. The call Rule1[x, y] is equivalent to the call Rule[x, y], whereas
      the calls Rule1[a, b, c, d, ... ] and Rule1[{a, b, c, d, ...}] return the list of format {a –> b, c –> d, ...} in case
      of even quantity of elements a, b, c, d, ... ; in other cases the call of Rule1 is returned unevaluated."

RuleQ::usage = "The call RuleQ[x] returns the True if x is a rule; otherwise, the False is returned."

TemporaryQ::usage =
  "The function call TemporaryQ[x] returns True if a symbol x defines the temporary variable, and False otherwise."

ListRulesQ::usage = "The call ListRulesQ[x] returns the True if x is a list of rules; otherwise, the False is returned."

ListCount::usage =
  "The function ListCount is an extension of the standard function StringCount onto the lists. The call ListCount[L, {L1,
      L2,...,Ln}] counts the number of occurrences of any of the sublists Lj in a list L (j=1..n)."

StrOfSymblQ::usage =
  "The call StrOfSymblQ[x, A] returns the True if a string x contains only symbols of a list A; otherwise, the False is returned."

StrPartition::usage =
  "The procedure call StrPartition[x, Y] returns the list of substrings of a string x that are limited by the first symbol of the string x
      and symbol Y in string format or by their list. For example \"xx ... xYzz ... zYhh ... hYpp ... p\" –> {\"xx ... xY\", \"xx
      ... xYzz ... zY\", \"xx ... xYzz ... zYhh ... hY\"}. In case of impossibility of the above partition the empty list, i.e. {} is
      returned. In particular, the procedure is useful enough at processing of contexts, directories and full paths to datafiles."

CallQ::usage =
  "The call CallQ[x] accurate to the sign returns the True if x is an expression of the format W[args] where W – the name of
      a procedure or function, and args – the  tuple of actual arguments, and False otherwise."
```

SubsString::usage =
"The call SubsString[s, {a, b, c, d, ...}] returns the list of substrings of a string s that are limited by its substrings {a, b, c, d, ...}, whereas the call SubsString[s, {a, b, c, d, ...}, p] with third optional argument p – a pure function in the short format – returns the list of substrings of the string s which are limited by its substrings {a, b, c, d, ...} and meet a condition determined by the pure function p. While the call SubsString[s, {a, b, c, d, ...}, p] with the 3rd optional argument p – an arbitrary expression different from a pure function – returns the list of the substrings parted by the substrings {a, b, c, d, ...}, with deleting of prefixes and suffixes {a, b, c, d, ...}[[1]] and {a, b, c, d, ...}[[-1]] respectively. At absence in the string s of at least one of substrings {a, b, c, d, ...} the SubsString call returns the empty list, i.e. {}."

Intersection1::usage =
"The procedure Intersection1 is an extension of the standard Intersection function. The procedure call Intersection1[x, y, z, h, ..., J] returns the list of elements common to all lists of strings in the mode IgnoreCase -> J ∈ {True, False} of search of identical strings. The empty list defines the empty intersection of lists x, y, z, h, ..., whereas $Falled is returned if the call contains empty lists or any list contains elements different from strings."

IntersectStrings::usage =
"In addition to the Intersection1 procedure the IntersectStrings procedure can be useful enough that solves the question of finding of all contiguous intersections of symbols between strings of the given tuple. The procedure call IntersectStrings[x, y, ..., h, J] in mode IgnoreCase -> J ∈ {True, False} finds all contiguous substrings common to the strings x, y, ..., h. At that, the setting {True|False} for the last argument J is equivalent to the option IgnoreCase -> {True|False} that treats lowercase and uppercase letters as equivalent or not at search of common substrings. Furthermore, the call IntersectStrings[x, J] returns the list of all contiguous substrings of symbols in a string x."

CorrectSubString::usage =
"The procedure call CorrectSubString[x, n] returns the result of extracting from a string x of the first syntactically correct expression, beginning from the n–th position to the right. At absence of such expression or at n beyond the range of {1, StringLength[x]}, or at x== \"\" the procedure call returns $Failed."

ReduceAdjacentStr::usage =
"In a number of cases there is a need of reducing to the given number of the quantity of entries into a string of its adjacent substrings. This problem is solved successfully by the ReduceAdjacentStr procedure. The procedure call ReduceAdjacentStr[x, y, n] returns a string – result of reducing to the quantity n >= 0 of occurrences into a string x of its adjacent substrings y. If a string x not contain substrings y, then the call returns the initial string x. Whereas the procedure call ReduceAdjacentStr[x, y, n, h], where h – an arbitrary expression, returns the above result on condition that at search of substrings y in a string x the lowercase and uppercase letters are considered as equivalent."

ReduceArgs::usage =
"The procedure call ReduceArgs[x] returns nothing, providing removal of \"excess\" formal arguments of a block/function/module x in the current session, leaving only their first entries in the tuple of formal arguments. This procedure has a certain independent interest as the Mathematica system even at the time of a function call with \"excess\" formal arguments can't correctly identify this situation."

MultipleArgsQ::usage =
"The function call MultipleArgsQ[x] returns True if at least one object composing the definition of a block/function/module x has multiple occurrences of formal arguments, and False otherwise."

ExtensionHeading::usage =
"The procedure call ExtensionHeading[G, y1, y2, ...] provides expansion of the tuple of formal arguments of a function, module or block G by arguments {y1, y2, ...} to the right from the tuple, with returning Null, i.e. nothing, and activation in the current session of the updated definition of the object G. Expansion of the tuple of formal arguments is made for object of G only on variables from the list {y1, y2, ...} that aren't its formal arguments; otherwise any expansion isn't made. List {y1, y2, ...} for updating can be both symbols in string format, and names of arguments with tests ascribed to it for their admissibility."

SubsString1::usage =
"The procedure SubsString1 is an extension of the above procedure SubsString. The call SubsString1[s, {a, b, c, d, ...}, pf, t] returns the list of substrings of a string s parted by the substrings {a, b, c, d, ...}; in addition, if a testing pure function acts as argument pf, the returned list will contain only the substrings satisfying such function. Moreover, at t = 1 the returned substrings are bounded by extreme substrings of the list y, i.e. y[[1]] and y[[-1]] accordingly, whereas at t = 0 the substrings are returned without the bounding extreme substrings y[[1]] and y[[-1]] accordingly. At last, in the presence of the fifth optional argument r – an arbitrary expression – search of substrings in string s is made from right to left, what in a number of cases simplifies algorithms of search of the required substrings. At absence in the string s of at least one of substrings {a, b, c, d, ...} the SubsString1 call returns the empty list, i.e. {}."

StringToList::usage =
"The procedure call StringToList[x] returns the list of syntactically correct substrings of a string x that are parted by the comma. The given procedure is useful enough at programming of problems, connected in particular with processing of headings and local variables."

SubStrToSymb::usage =

"The procedure call SubStrToSymb[x, n, y, p] returns a substring of a string x bounded on the left (p = 1) by a position n and the first occurrence of a symbol y, and on the right (p = 0) by a position n and the first occurrence of a symbol y, i.e., at p = 0 and p = 1 the search of the symbol y is done right to left and left to right accordingly. Moreover, in a case of absence at search of a required symbol y the procedure call SubStrToSymb[x, n, y, p] returns $Failed, while in other especial cases the procedure call is returned unevaluated."

StrAllSymbNumQ::usage =
"The function call StrAllSymbNumQ[x] returns True if a string x contains only symbols and/or integers, and False otherwise."

ListStringQ::usage = "The call ListStringQ[x] returns True if x is a list of strings; otherwise, False is returned."

PosListTest::usage =
"The call PosListTest[L, p] returns the list of positions of a list L which satisfy a test determined by a pure function p."

FirstPositionsList::usage = "The function call FirstPositionsList[x] returns the list of positions of only first occurrences of elements in a list x, i.e. ignoring their multiplicities."

SortRevStr::usage =
"The call SortRevStr[x, y] returns the result of reversion of a string x, if y = Reverse, or result of symbol–by–symbol sorting of a string x in an increasing order, if y = Sort; whereas the call SortRevStr[x, y, z] with the third optional argument z = SymbolGreater returns the result of symbol–by–symbol sorting of a string x in a decreasing order."

UnDef::usage = "The call UnDef[x] returns True if a symbol x is not determined, and False otherwise. Whereas the call UnDef[x, y] returns through the second optional argument y the value Head1[x]."

RemovedQ::usage =
"The function call Remove[x1, ...] removes x1 symbols completely, so that their names are no longer recognized in the current session. Whereas the function call RemovedQ[x] returns True, if a x symbol has been removed from the current session, and False otherwise."

PrefCond::usage =
"The call PrefCond[x, y] returns the result of extraction from a string x of a prefix limited by the beginning of x and by the first occurrence in it of a substring y; otherwise, the call returns the empty string, i.e. \"\"."

Ind::usage = "The call Ind[x] returns a list of form {Symbol, {Expression}} if x has format Symbol[Expression], and a simplified expression x otherwise. A sequence can be as an Expression."

LangHoldFuncQ::usage =
"The call LangHoldFuncQ[x] returns True if x – a basic function of Math–language, and False otherwise. At that, under basic function is understood a system function with one of the attributes ascribed to it, namely: HoldFirst, HoldAll or HoldRest."

Clear1::usage =
"The procedure Clear1 can be considered as an useful generalization of standard functions Clear and ClearAll. The call Clear1[h, \"x1\", \"x2\", \"x3\", ..., \"xn\"] returns Null, i.e. nothing, for h=1, clearing symbols {x1, x2, x3, ..., xn} from the expressions assigned to them with preservation of all their attributes, whereas for h=2, clearing symbols {x1, x2, x3, ..., xn} from the expressions assigned to them and from all their attributes."

ClearCS::usage =
"The procedure ClearCS can be considered as an useful generalization of standard functions Clear, ClearAll and Remove. The call ClearCS[ClearAll] returns Null, i.e. nothing, clearing all values, definitions, attributes, messages and defaults associated with symbols that received values in the current session. Whereas the call ClearCS[Remove] returns Null, i.e. nothing, removing symbols that been received values in the current session completely, so that their names are no longer recognized by Mathematica."

DuplicateLocalsQ::usage =
"The call DuplicateLocalsQ[P] returns the True in the presence in definition of a procedure P of duplication of local variables, otherwise the False is returned. In addition, at return of True through the second optional argument the list, simple or of ListList–type, whose elements define names of duplicated local variables with their multiplicities of occurrences in the list of local variables is returned."

DuplicateLocals::usage =
"The procedure call DuplicateLocals[x] returns the simple or the nested list the first element of a list or sublists of which defines a name in string format of a multiple local variable of a block/module x whereas the second defines its multiplicity. In the absence of multiple local variables the empty list, i.e. {} is returned."

DelDuplLocals::usage =
"The procedure call DelDuplLocals[x] returns the name of a module/block x, reducing its local variables of the same name to 1 with activation of the updated definition x in the current session. Whereas the call DelDuplLocals[x, y] with the second optional argument y – an undefinite variable – through y returns the list of excess

local variables. At that, first of all only simple local variables (without initial values) are reduced. This procedure well supplements the previous DuplicateLocals procedure. Procedure provides processing of the objects having single definitions, however it is easily generalized to the objects of the same name."

DelDuplLocalsM::usage =
"Unlike the DelDuplLocals procedure the DelDuplLocalsM procedure supports the operating with the blocks and modules of the same name. The procedure call DelDuplLocalsM[x, y] is completely analogous to a procedure call DelDuplLocals[x, y], including admissible tuples of the actual arguments."

ReplaceLocals::usage =
"The ReplaceLocals procedure provides dynamic replacement of local variables of a block/module for a period of the current session. The procedure call ReplaceLocals[x, y] returns Null, i.e. nothing, providing replacement of local variables of a block/module x by new local variables of the same name, including their initial values which are defined by the second argument of y – a separate string of the format \"name\" or \"name=initial value\", or their list. Only those local variables of the object x are subjected to replacement whose names are in y."

OptimalLocals::usage =
"The procedure call OptimalLocals[x] returns name of a module/block x, providing optimization in the current session of its local variables in the context of eliminating duplication and redundancy of variables. In case of a x object of the same name the message \"Object <x> has multiple definitions\" is returned."

OptimalLocalsM::usage =
"The procedure OptimalLocalsM is an extension of the above OptimalLocals procedure onto case of objects (blocks, functions, modules) of the same name. The procedure call OptimalLocalsM[x] returns definition of an object x, providing an optimization in the current session of its local variables in the context of eliminating duplication and redundancy of variables. At that, the OptimalLocalsM procedure succesfully processes single objects too."

OptimalLocalsN::usage =
"The procedure OptimalLocalsN is an extension of the above OptimalLocals procedure onto case of the nested objects (blocks and modules). The procedure call OptimalLocalsN[x] returns definition of a block or module x, providing optimization in the current session of its local variables and of local variables of all subobjects (blocks and modules) composing it in the context of eliminating duplication and redundancy of local variables. At that, the OptimalLocalsN procedure succesfully processes simple blocks or modules (i.e. objects that are non–nested) too."

ProcUniToMod::usage =
"The procedure call ProcUniToMod[x] returns the definition of a block or module x which can be the object of the same name, with replacement of all blocks by modules along with optimization of local variables of all subobjects composing the object x."

ReplaceSubProcs::usage =
"The ReplaceSubProcs procedure is intended for replacement in an arbitrary procedure of subprocedures of the type {Block, Module}. The procedure call ReplaceSubProcs[x, y] returns the result of replacement in a procedure x of type {Block, Module} of subprocedures that is defined by the 2nd argument y that has the format {a1, b1} or {{a1, b1}, ..., {ap, bp}}, defining replacements of the aj subprocedures of the procedure x onto the bj procedures of the type {Block, Module}, the bj definitions should be previously evaluated in the current Mathematica session; if bj = \"\" (empty string) then the appropriate subprocedure aj is deleted (j = 1..n). At that, optimization of local variables of the main x procedure and all its subprocedures is carried out. The following fragment represents source code of the procedure with examples of its usage."

IsMonotonic::usage =
"The call IsMonotonic[x] returns the value True if x is monotonic, and the value False otherwise. A string x is monotonic if the sequence of characters it comprises is non−increasing or non−decreasing, when the characters are identified with their ASCII code points. The call IsMonotonic[x, t] in case of basic result True through 2−nd undefinite symbol t the order of the succession of symbols in string x {\"Increase\", \"Decrease\"} is returned. Whereas in other cases the procedure call is returned unevaluated. If x is the empty string, the call returns the value True."

IsPackageQ::usage =
"The IsPackageQ procedure is intended for testing of any mx–file regarding existence of the user's package in it along with upload of such package into the current session. The call of the IsPackageQ[x] procedure returns $Failed if the mx–file doesn't contain a package, True if the package which is in the mx–file x is loaded into the current session, and False otherwise. Moreover, the procedure call IsPackageQ[x, y] through the second optional argument y – an undefinite variable – returns the context associated with the package uploaded into the current session. In addition, is supposed that a datafile x is recognized by the testing function FileExistsQ, otherwise the procedure call is returned unevaluated."

Uprocs::usage =
"The call Uprocs[] returns the nested list of procedures activated in the current session; in addition, each procedure is identified by 3−element list whose the first element the procedure name, the second element – its heading, and the third element – the type of procedure (Block, Module). In the absence of such procedures the empty list is returned."

ActCsProcFunc::usage =
"The call ActCsProcFunc[] returns the nested 2−element list whose elements are two sublists of variable length.

The first sublist contains \"Procedure\" as the first element, hereas other elements define 2−element lists containing names of procedures activated in the current session along with their headings. While the second sublist contains \"Function\" as the first element whereas other elements define the 2−element lists containing names of functions activated in the current session along with their headings."

ActBFM::usage =
"The call ActBFM[] of the rather simple function returns the sorted list of names in string format of the user blocks, functions and modules, whose definitions have been evaluated in the current Mathematica session."

BlockToModule::usage =
"The call of procedure BlockToModule[x] returns Null, i.e. nothing, providing converting of a procedure of Block−type into the corresponding procedure of Module−type with saving of all attributes and options of the source procedure. Moreover, with an object x several definitions of modules, blocks and/or functions can be associated also, however the call BlockToModule[x] provides converting into module structures only the block components of the object x."

BlockFuncModQ::usage =
"The call BlockFuncModQ[x] returns True, if x – a symbol defining a block, a traditional function (with heading) or a module; otherwise, False is returned. Moreover, through the optional argument y – an indefinite variable the call BlockFuncModQ[x,y returns the type of the object x in the context of {\"Block\", \"Function\", \"Module\"} on condition that main result is True."

BlockModQ::usage =
"The call BlockModQ[x] returns True, if x – a symbol defining a block or a module; otherwise, False is returned. Moreover, through the optional argument y – an indefinite variable the call BlockModQ[x,y returns the type of the object x in the context of {\"Block\", \"Module\"} on condition that main result is True."

QBlockMod::usage = "The call QBlockMod[x] returns True, if x – a symbol defining a block or a module, including objects of the same name; otherwise, False is returned."

SingleDefQ::usage = "The call SingleDefQ[x] returns True if the actual argument x defines a name of a procedure, a block or a function having single definition; in other cases the function call returns False."

StructProcFunc::usage =
"The call StructProcFunc[x] returns the simple or nested list, whose elements depending on the type {\"Block\", \"Module\", \"Function\"} of the actual argument x possess the format {Type, Heading, Locals, Body} for {\"Block\", \"Module\"} and {Type, Heading, Body} for Function\"; in addition, under the function is understood such object x that QFunction[x] = True. Procedure reveals the general structural organization of objects of the above type."

StrNestedMod::usage =
"In some cases there is a problem of the structural analysis of the nested procedures of Module–type. The procedure call StrNestedMod[x] returns the nested list of names in string format of subprocedures of a procedure x; at that, each name is located at an appropriate level of nesting. Whereas the procedure call StrNestedMod[x, y] with the second optional argument y – an undefinite variable – through y returns the list of names of all subprocedures of the procedure x, in which the first element – a name x of the main procedure. If a procedure x doesn't contain subprocedures, the procedure call StrNestedMod[x, y] returns {\"x\"} whereas the second argument y retains own value."

BFMSubsQ::usage =
"The call BFMSubsQ[x] returns the list of format {True, Heading}, if the definition of a symbol x, defining a block or a module, including objects of the same name; otherwise, {False, Heading} is returned. In case of an object x of the above type of the same name, the call returns the nested list whose sublists have the above format. On an object x of type, different from {Block, Module}, the call returns False. In addition, the call BFMSubsQ[x, y] with the 2nd optional argument y – an indefinite variable – through y returns the list of format {Heading, N}, where N defines number of blocks, functions, and modules entering into a subobject with the heading Heading of the object of the same name x."

ProcLocals::usage =
"The call ProcLocals[x] with one argument returns the list of local variables of a procedure x in string−format, whereas the call ProcLocals[x, y] in addition through an undefined variable y returns number of the position in string−representation of definition of a procedure x with which its body begins."

ProcBody::usage =
"The call ProcBody[x] returns strictly speaking the body of a block, a function with heading or a module x in string−format. The procedure successfully processes also the objects of the same name x, returning the list of bodies of subobjects composing the object x."

PatternQ::usage = "The call PatternQ[x] returns True if x is a pattern, and False otherwise."

StringStringQ::usage = "The call StringStringQ[x] returns True if x is a string of type StringString, and False otherwise."

CurrentNb::usage = "The function call CurrentNb[] returns the name of the current document of the format {\"cdf\", \"nb\"}."

StringJoin1::usage =
"The StringJoin1 procedure is a modification of the standard function StringJoin. The call StringJoin1[x] returns the result of consecutive concatenation of elements−strings parted by the comma of a list x."

StringLevels::usage = "The call StringLevels[x] returns the level of nesting of a string x of StringString−type; in addition, the null level is defined for the usual strings."

StringDependAllQ::usage = "The call StringDependAllQ[s, p] returns True only if a string p is the substring of a string s, or every string from a list p is contained in the string s."

StringPat::usage = "The procedure call StringPat[x, y] returns the string expression formed by strings of a list x and objects {\"_\", \"__\", \"___\"}; the call returns x if x – a string."

StringCases1::usage = "The procedure call StringCases1[x, y, z] returns the list of the substrings in a string x that match a string expression, created by the call StringPat[x, y]."

StringCases2::usage =
"The procedure call StringCases2[x, y] returns a list of the disjoint substrings in a string x that match the string expression of the format Shortest[j1~~__~~j2 ~~__~~ ... ~~__~~jk], where y = {j1, j2, ..., jk} and y is different from {}."

StringFreeQ1::usage = "The function call StringFreeQ1[x, y, z] returns True if no substring in a string x matches a string expression, created by the call StringPat[x, y], and False otherwise."

StringFreeQ2::usage = "The function call StringFreeQ2[x, {a1, a2, a3, ...}] returns True if all substrings {a1, a2, a3, ...} are absent in a string x, and False otherwise."

SelectStrings::usage =
"The procedure call SelectStrings[x, y, J] in mode IgnoreCase –> J ∈ {True, False} finds all elements of a list x consisting from strings that are free from all substrings from a list y. At that, the setting {True|False} for the optional argument J is equivalent to the option IgnoreCase –> {True|False} that treats lowercase and uppercase letters as equivalent or not at search of the demanded elements of x. If argument had been omitted, it should be replaced by True."

SelectContains::usage =
"The SelectContains procedure is a certain extension of standard functions Select, Cases and StringCases in the case of a list of strings. The procedure call SelectContains[x, y, r, Ig] according to the value False or True of the r argument returns the list of elements−strings of a list x that contain elements−strings from a list y or don't contain occurrences from the list y respectively. At that, the setting {True (the default in the case of the omitted argument)|False} for the optional argument Ig is equivalent to the option IgnoreCase –> {True|False} that considers lowercase and uppercase letters as equivalent or not at search of substrings."

StringFreeQ3::usage =
"The StringFreeQ3 function extends the StringFreeQ function onto a case of the first argument different from a string. The function call StringFreeQ3[x, y] returns True if an expression x doesn't contain occurrences of a string y, and False otherwise."

Ver::usage = "The call Ver[] returns the string with type refinement of an operational platform."

$Version1::usage = "The $Version1 variable returns the number of the Mathematica version for Microsoft Windows."

$Version2::usage = "The $Version2 variable is an analog of the $Version1 variable and returns the number of the Mathematica version for Microsoft Windows."

PCOS::usage =
"Values of the global variables $System, $SystemID and $OperatingSystem define the strings describing the current operational platform. Meanwhile, in a number of cases the specification of the current operational platform represented by them can be insufficient, in that case it is possible to use the PCOS procedure, whose call PCOS[] returns the 2–element list, whose first element determines the name of the computer owner, whereas the second element – the type of an operating platform."

OSplatform::usage =
"The procedure call OSplatform[] in string format returns the 2–element list, whose the first element – the type of the current operating system, and the second element – main characteristics of this system."

Table1::usage =
"For simulation of the main operations with structures of the table organization, similar to the Maple system, in the environment of Mathematica the procedure Table1[L, x] can be used which considers a list L of the ListList–type, whose 2–element sublists {x, y} which correspond to {index, entry} of Maple–table respectively, as the table. As the second x–argument may be: (1) a list {a, b}, (2) a word {\"index\"|\"entry\"} or (3) an expression of other type. Result of the call Table1[L, x] is the list of ListList–type which is derivable from the initial list L as follows. Case (1): in the presence in the L of a sublist with the first element a it is replaced by a list {a, b}, otherwise it supplements L; if the argument x has view {a, Null}, then in the L the sublist with the first element a is removed. Case (2): the list of

{indices|entries} of the list L is returned; whereas in case (3) the procedure call returns entry for a X–index if such in the given table L really exists. On other tuples of the actual arguments the call Table1[x, y] returns $Failed."

TabLib::usage =
"On the basis of the table organization supported by the Table1 procedure, enough simply the user libraries of procedures and functions can be defined. On this basis as one of such approaches we will give the LibBase library example whose structural organization has the format of the ListList list and whose elements have length 2. Such library has the following basic view:

LibBase := {{Help, {\"O1::usage = \"Help on O1. \", ..., \"On::usage = \"Help on On. \"}}, {O1, PureDefinition[O1]}, {O2, PureDefinition[O2]},..., {On, PureDefinition[On]}}

The first 2–element sublist of LibBase library by the first element contains Help while the second element represents the list of usages in string–format on all objects, whose definitions are in LibBase library; in addition, their actual presence at the library isn't required. The rest elements of the library are 2–element sublists of the format {Oj, PureDefinition[Oj]}, where Oj – the name of j–object, and PureDefinition[Oj] – its definition in string–format. The main operations with the library organized thus are provided by the TabLib procedure. TabLib call[x, y] depending on the second argument y returns or the current contents of the library being in a mx–file x, or the names of the objects being in the library, or their definitions, namely:

TabLib[x, \"index\"] – returns the list of names of the objects whose definitions are in the library x, including the name Help of the HelpBase of the library;
TabLib[x, \"entry\"] – returns the list of definitions of objects which are contained in the library x, including the Help base of the library;
TabLib[x, {N, Df}] – returns the contents of library x after its extension onto a new definition of an object with name N if Df is different from Null; in addition, the old definition of N–object is updated;
TabLib[x, {N, Null}] – returns the contents of the library x as a result of removal from it of definition of object with a name N; in addition, the help (usage) for N remains;
TabLib[x, N, y, z, ...] – returns the result of the call N[y, z, ...] of a procedure/function from the library x; if the object N is absent in the library, $Failed is returned;
TabLib[x, N] – if N – the help (usage) concerning an object N, it supplements the help base of the library x with return of the updated contents of the library x.

In other cases the call is returned unevaluated or returns $Failed. More detailed information concerning the TabLib procedure can be found in our latest book [6]."

UsageBase::usage =
"On the basis of the list organization supported by the Mathematica system, enough simply is possible to define a help base for the user libraries. On this basis as one of such approaches we will give an example of the help BaseHelp base, whose structural organization has the list format with elements of the following view. Below, the basic type of such help base is presented, namely:

BaseHelp := {{\"O1::usage = \"Help on O1.\", ..., \"On::usage = \"Help on On.\"}

Thus, the help base represents the list of references (usage) in string–format concerning all Oj–objects, whose definitions are in the user library; moreover, as the initial help BaseHelp base intended for its filling by the necessary contents, the empty list can be defined, namely:

BaseHelp := {}

Base operations with the database of usages, organized thus, the UsageBase procedure provides. The call UsageBase[x, y] depending on the second argument y returns the current contents of the help base being in a mx–file x, or deletes its usages, either updates or activates them in the current session, namely:

UsageBase[x, \"?\"] – returns the contents of the BaseHelp database in the list format which is in a mx–file x;

UsageBase[x] – returns Null, i.e. nothing, activating in the current session the help database which has been set by the mx–file x;

UsageBase[x, y] – returns Null, i.e. nothing, deleting from the help database (which has been set by the mx–file x) the usage concerning an object with the name y if such exists;

UsageBase[x, y] – returns Null, i.e. nothing, supplementing the help database which has been set by the mx–file x, by the usage y of the above–mentioned format, or replaces in the help database the usage by an usage y if names of Oj–objects in both usages coincide.

More detailed information concerning the UsageBase procedure can be found in our latest book [6]."

Usages::usage =
"The Usages procedure provides maintaining of help database irrespective of library, what is rather convenient in a series of cases of the organization of the user software. For initial filling of a help database in the current session all usages known at present concerning means which are planned for inclusion into the user library are evaluated. Then, by the call Usages[x, y] is

provided the saving in a x–datafile of ASCII–format of all usages relating to program means whose names are defined by a list y. In addition, the saving of usages in a x–datafile is made in the append mode; if the specified datafile x is absent, the empty datafile x is created. While the call Usages[x, y, z, ...] when as arguments, since the second argument, the names {y, z, ...} of software are given, from help database the usages concerning these means are deleted. At last, the call Usages[x] activates all usages contained in help database x in the current session, making them available irrespectively from existence of a library of the means described by these usages. Moreover, the successful call of the Usages procedure returns the Null, i.e. nothing, otherwise $Failed is returned, in particular, in case of the call Usages[x] at the absent or empty datafile x."

Usages1::usage = "The call of the simple function Usages1[x] provides the output of all usages describing the means contained in the user package activated in the current session and associated with a context x."

UsagesMNb::usage =
"The call UsagesMNb[x] returns the list of references concerning software of the user package being in a datafile x of the format {\"m\", \"nb\"}. These references is returned in string–format. In addition, for a datafile x of the m–format the list of the references containing a prefix \"Name:: usage =\" is returned, whereas for a datafile x of the nb–format the list of references in string–format without such prefixes is returned. If for a package from a datafile x of the m–format its activation in the current session isn't required, then for a package from a datafile x of nb–format x such activation is required.
NB: The UsageMNb procedure correctly operates only on files of {m, nb}–type that have been created in Mathematica of versions 10.3.0.0 and below because of change of internal formats of these files, beginning with Mathematica of version 10.4.0.0."

CallSave::usage =
"The procedure call CallSave[x, y, z] returns the result of the call y[z] of a procedure/function y on a list z of factual arguments passed the y provided that object definition y with usage are located in a txt–file x that has been earlier created by the Save function. If an object with the given name y is absent in a datafile x, the procedure call returns $Failed. If a datafile x contains definitions of several procedures or functions of the same name y, the procedure call is executed relative to their definition whose formal arguments correspond to a list z of actual arguments."

QSaveGUI::usage =
"The function call QSaveGUI[x] returns True if a m–file x has the format corresponding to the format of a file created by means of commands chain of the GUI \"File –> Save As –> Mathematica Package (*.m)\", and False otherwise."

PackageUsages::usage =
"The question of documenting of the user package is an important enough its component; at that, absence in a package of usage for an object contained in it does such object as inaccessible at uploading the package into the current session. So, description of each object of the user package has to be supplied with the corresponding usage. At the same time it must be kept in mind that mechanism of documenting of the user libraries in the Maple system is much more developed, than similar mechanism of documenting of the user package in the Mathematica system. Thus, if the mechanism of formation of the user libraries in the Maple is simple enough, providing simple documenting of library means and providing access both to means of library, and to their references at the level of the system means, in the Mathematica system the similar mechanism is absent. Receiving of the usage concerning a x package tool is possible only by means of calls ?x or Information[x] provided that a package has been uploaded into the current session. Meanwhile, in case the package contains enough many means, for obtaining the usages concerning the demanded means it is necessary to be sure in their existence, first of all. The next PackageUsages procedure can be rather useful to these purposes. The procedure call PackageUsages[x] returns the path to a datafile in which the extension \".m\" of a datafile x is replaced on \"_Usages.txt\"; the received datafile contains usages of the user package formed standardly in the form of a nb–document (see above) with the subsequent its saving in a m–file x by means of chain of the commands \"File – >Save As\" of the GUI. The information on the specific package tool y has the format y = \"Help on y\". The received txt–file allows to look through easily its contents regarding search of necessary means of the user package."

ExcessVarsPack::usage =
"When uploading the package along with the main exports can be generated so–called excess symbols whose analysis represents a quite certain interest. The procedure call ExcessVarsPack[x] returns the nested four–element list of the names of the excess symbols that are generated by the uploading of the package with a context x into the current session. The procedure call with a context x which is absent in the list determined by the $Packages variable is returned unevaluated. The elements of the above nested list are determined as follows: (1) the list of names of the excess temporary definite symbols with context x of the package, (2) the list of names of the excess temporary undefinite symbols with context x of the package, (3) the list of names of the excess definite symbols with context x of the package, (4) the list of names of the excess undefinite symbols with context x of the package, finally (5) the list of other excess symbols of the package with context x."

RedMfile::usage =
"It is supposed that a datafile x of m–format structurally corresponds to the standard file with a package; an example of such datafile of m–format is given in the first shaded area of the previous fragment. The procedure call RedMfile[x,n,y] returns the full path to a m–file, whose FileBaseName has view FileBaseName[x] <> \"$\" which is a result of application to an initial m–file of an operation y concerning its object determined by a name n, namely:

\"delete\" – from a x datafile the usage and definition of object with a n name are removed, the initial datafile doesn't change; if such object in the datafile is absent, the full path to the initial datafile is returned;
\"add\" – usage and definition of object with a n name are added into a x file whereas the initial datafile doesn't change; if such object in the file already exists, the full path to the initial datafile x is returned;

\"replace\" – usage and definition of object with a n name are replaced in a x file while the initial
datafile doesn't change; if such object in a file is absent, $Failed is returned.

If an initial datafile x has structure, different from specified, the procedure call returns
$Failed; at that, successful performance of the operations \"add\" and \"replace\" requires preliminary
evaluation in the current session of the construction n::usage along with definition for object n."

RedMxFile::usage =
"The RedMxFile procedure provides automation of a modification of datafiles of mx–format which is considered above. The call
RedMxFile[x, y, r, f] returns the full path to a mx–datafile, whose FileBaseName has view FileBaseName[x] <> \"$\" that
is a result of application to an initial mx–file of an operation r concerning its object determined by a name y, namely:

\"delete\" – from a x datafile the usage and definition of object with a y name are removed, the
initial datafile doesn't change; if such object in the datafile is absent, the procedure call returns $Failed;
\"add\" – usage and definition of object with a y name are added into a x file whereas the initial datafile
doesn't change; if such object in the file already exists, the procedure call returns $Failed; the fourth
argument f defines mx–file containing the usage and definition of the supplemented object y;
\"replace\" – usage and definition of object with a y name are replaced in a x file while the initial datafile
doesn't change; if such object in a file is absent, the procedure call returns $Failed; the fourth argument
f defines a mx–file containing a package with context of the initial mx–file x along with usage and
definition of the added y object. At that, if an object y is undefined the procedure call returns $Failed.

Thus, return of the path to an updated datafile \"x$.mx\" serves as an indicator of success of the
RedMxFile procedure call. At that, successful performance of the operations \"add\" and \"replace\"
requires preliminary evaluation in the current session of the expressions of the following formats, namely:

Cont`Name[….]:= Definition of an object Name
Cont`Name::usage = \"Help on the object Name.\"

where Name – the name of an object and \"Cont`\" is the context ascribed to the
updated initial mx–file x, with the subsequent saving of the forenamed evaluated object Cont`Name
in a mx–file f. The unsuccessful procedure call returns $Failed or is returned unevaluated."

ContextForPackage::usage =
"The procedure call ContextForPackage[x, y] returns two–element list whose first element determines a new context y, assigned to
the user package from mx–file x, while the second element defines path to the file which contains the updated file
x. The updated file is saved in the same directory as the datafile x, but with the name FileBaseName[x] <> \"$.mx\".
At the same time, irrespective of whether the initial file x has been loaded into the current session, as a result of
the procedure call the file is removed from the current session, remaining in external memory without change. At
the same time, if the initial package from x file has been already uploaded up the procedure call, then the package
with the updated y context remains in the current session, otherwise it is removed from the current session."

AddMxFile::usage =
"The AddMxFile procedure illustrates the use of the ContextForPackage procedure for merging of mx–files with the user
packages with saving of the result in a mx–file with an ascribed context y. It is supposed that the merged mx–files
haven't been loaded in the current session and have been supplied with appropriate contexts. The procedure
call AddMxFile[x, y] returns the 2–element list whose the first element defines context y, whereas the second
element determines mx–file with the result of merging of mx–files which are defined by the list x. At last, the
call AddMxFile[x, y, z] with the third optional argument z – an arbitrary expression – additionally allows to save
in the current session the means with the y context, otherwise the package with y context is unloaded from the
current session. Meantime, the AddMxFile procedure has quite concrete appendices in practical programming."

MathPackages::usage =
"The call MathPackages[] returns the sorted list of names of packages of the current version of Mathematica system. Whereas
the call MathPackages[w] with optional argument w – an indefinite variable – additionally provides return
through it the 3–element list whose first element defines the current version of the Mathematica system, the
second element defines the type of the license, and the third element defines a deadline of action of the license."

SystemPackages::usage =
"The SystemPackages procedure is used for testing of the system packages (m–files) being in the catalog, defined by the variable
$InstallationDirectory. The call SystemPackages[] returns the list whose 2–element sublists have the format {Package,
its context} while the call SystemPackages[w] through optional argument w – an indefinite variable – additionally
returns the list of the system packages which aren't possessing contexts, i.e. are used for internal needs of the system."

ActUcontexts::usage =
"The ActUcontexts procedure provides obtaining of the list of contexts of the current session which are associated with
the user packages. The procedure call ActUcontexts[] for obtaining of the list uses an algorithm that is based
on the analysis of system datafiles of formats {\"m\", \"tr\"}, while the call ActUcontexts[x] where optional
argument x is arbitrary expression, is based on the search of system datafiles of the view \"StringTake[Context, {1,

–2}]<>{\"m\", \"tr\"}\". If the first algorithm is more universal, whereas the second significantly more high–speed."

SysContexts::usage = "The procedure call SysContexts[] returns a list of all system contexts at the current moment."

MainContexts::usage = "The function call MainContexts[] returns a
list of all main contexts at the current moment, supplementing the system Contexts function."

AllContexts::usage =
"The procedure call AllContexts[] returns the list of contexts contained in system packages of the current Mathematica version, whereas the call AllContexts[x] returns True, if x – a context of the above type, and False otherwise."

MultipleContexts::usage =
"The mechanism of contexts of the Mathematica system allows existence in the current session of symbols of the same name with various contexts. The procedure call MultipleContexts[x] returns the list of contexts attributed to a symbol x."

ContextsCS::usage = "The function call ContextsCS[] returns the list of contexts of the current Mathematica session."

SystemSymbols::usage = "The function call SystemSymbols[] returns all system symbols at the current moment."

ContextsInModule::usage = "The procedure call ContextsInModule[x] returns the nested list of the following format, namely:

{{\"Args\", ...}, {\"Locals\", ...}, {\"SubProcs\", ...}, {\"Context1'\", ...}, ..., {\"ContextN'\", ...}}

where the first sublist defines formal arguments of a module x, the second sublist defines its local variables, the third sublist defines names of subprocedures of the procedure x whereas other sublists, starting with the fourth, define variables of the module x with the contexts corresponding to them. All elements of sublists of the returned list have string format."

ContentObj::usage =
"The procedure call ContentObj[x] returns the list of the format {Contextj, j1, j2, ...}, where jk – the user means, used by a block/function/module x, and Contextj – a context corresponding to them. In the case of several contexts the nested list is returned whose sublists have the above–mentioned format. The call returns all means of the user packages on which a x object depends. In the absence of such means the procedure call returns the empty list, i.e. {}."

StrToList::usage =
"The StrToList procedure is intended for converting of strings of the structure \"{xxxxxxx... x}\" or \"xxxxxxx... x\" to the list of strings received from the strings of the mentioned format, parted by symbols of comma \",\" and/or \"=\". Examples from the above books quite visually illustrate the principle of performance of StrToList procedure along with format of results of converting of strings returned by it."

ReplaceLevelList::usage =
"In some problems of programming which use the list structures arises a quite urgent need of replacement of values of the lists which are located at the given nesting levels. Standard means of Mathematica system don't give such opportunity. In this regard the procedure has been created whose call ReplaceLevelList[x, n, y, z] returns result of replacement of an element y of a list x that is located at a nesting level n onto a value z. Lists which have identical length also can act as arguments y, n and z. At violation of this condition the procedure call returns $Failed. In case of absence of the fourth optional argument z the call ReplaceLevelList[x, n, y] returns result of removal of elements y which are located on nesting level n of a list x. The procedure uses our procedures GenRules, LevelsOfList, ToString4 and IntegerListQ of the AVZ_Package package."

ReplaceListCond::usage =
"In addition to standard functions of so–called Replace–group a rather simple ReplaceListCond procedure provides conditional replacement of elements of a list. The call ReplaceListCond[x, f, y] returns result of replacement of the elements of a x list which meet a condition defined by the testing Boolean f function onto an expression y. At that, the lists can be used as f and y; more precisely, w objects for which FunctionQ[w] = True or SysFuncQ[w] = True are admitted as an argument f."

PositionsListCond::usage =
"The procedure call PositionsListCond[x, f] returns list of positions of the elements of a x list which meet a condition determined by the testing Boolean f function or their list. At that, the list can be used as f; more precisely, w objects for which FunctionQ[w] = True or SysFuncQ[w] = True are admitted as an argument f."

LevelsList::usage =
"The procedure call LevelsList[x] returns the nesting level of a list x. For example, LevelsList[{{{{{a, b, c}, {{x, y, z}}, m, n}}}}] = 5 and LevelsList[{{a, b, c}, {{x, y, z}}, m, n}] = 1. The procedure is useful enough at processing of the nested lists."

EquExprPatt::usage = "The call EquExprPatt[x, p] returns True if an expression x corresponds to a pattern p, and False otherwise."

EquExprPatt1::usage =
"The call EquExprPatt1[x, y] returns True if an expression x structurally corresponds to an expression y, and False otherwise."

OpSys::usage = "The procedure call OpSys[] returns the type of used operational

platform. The procedure is useful in certain appendices above all of the system character."

Riffle1::usage = "The procedure call Riffle1[x] restructures a ListList–list x into the ListList–list as follows:
Riffle1[{{a1, b1, c1, ...}, {a2, b2, c2, ...}, ...] → {{a1, a2, a3, ...}, {b1, b2, b3, ...}, ...}."

ExprComp::usage =
"The call ExprComp[x] returns the set of all subexpressions composing an expression x, whereas the call ExprComp[x, z], where the second optional argument z is an indefinite variable, through z is additionally returned the enclosed list of subexpressions of the expression x by the levels, since the first level."

Cost::usage =
"An expression can be composed by using the arithmetic operators. Such expressions can be one of three types, namely: type `+`, type `*`, type `^` along with type `Indexed` and `Function`. That is, the expression a – b is of type `+` with operands a and –b. Similarly, a/b is of type `*` with operands a and b^(–1). Finally, a^b is of type `^` with operands a and b. Cost is used to compute an operation count for the numerical evaluation of the given expressions. The operation count is expressed as a polynomial in the names Plus, Times, Power, Indexed and Function with non–negative integer coefficients. The procedure of the same name provides calculation of this Cost indicator. The call Cost[x] returns the Cost indicator of the above format for an arbitrary algebraic expression x; in the absence for x operators the procedure call returns 0."

SubLists::usage = "The call SubLists[x] returns the list of all possible sublists of a nested list x, taking into account their nesting. If the list x is simple, the call SubLists[x] returns the empty list, i.e. {}."

ElemsList::usage =
"The successful procedure call ElemsList[x, y] returns the elements of a list x depending on list of their positions given by a list y. The list y format in the general case has the view {n1, ..., nt, {m1 ; ... ; mp}}, returning elements of a list x according to a standard relation x[[n1]] ... [[nt]][[m1 ; ... ; mp]]. At that, the argument y allows the following formats of the coding {n1, ..., nt}, {m1 ; ... ; mp}}, {}."

SubListsMin::usage =
"The SubListsMin procedure as a whole is useful for operations with lists. The call SubListsMin[L, x, y, t] returns the sublists of a list L that are limited by elements {x, y} and have the minimum length; at t = \"r\" selection is made from left to right, and at t = by \"l\" from right to left. Whereas the call SubListsMin[L, x, y, t, z] with optional 5th argument z – an arbitrary expression – returns sublists without the limiting elements {x, y}."

SubListsMin::usage =
"The call ElemsList[x, y] returns elements of a list x depending on a list y of their positions which have been set by the list y. Generally, the list y has format of the view {n1,...,nt, {m1;...;mp}}, returning elements of the list x according to the standard formula x[[n1]]...[[nt]][[m1;...;mp]]; otherwise $Failed is returned. The argument y, meanwhile, admits the following formats of the coding {n1,...,nt}, {{m1;...;mp}} and {}."

LevelsOfList::usage =
"The procedure call LevelsOfList[L] returns the list of levels of elements of a list Flatten[L] of an initial list L. At that, in case of L = {} the empty list is returned, i.e. {}; in case of a simple list L the single list of length Length[Flatten[L]], i.e. {1, 1, ..., 1} is returned. Level of elements of a simple list is equal 1. For example, LevelsOfList[{a, m, n, {{b}, {c}, {{m, {{{g}}}, n, {{{{{gs}}}}}}}, d}] –> {1, 1, 1, 3, 3, 4, 7, 4, 9, 1}."

ListTrim::usage =
"In contrast to the function StringTrim the procedure ListTrim[L, p] trims sublists of a list L that match p from the beginning and end. At that, an arbitrary expression or their list can be as a actual argument p."

Split1::usage =
"Unlike two standard functions Split and SplitBy, the call Split1[x, y] splits a list x into the sublists consisting of its elements, located between occurrences of an element or a list of elements y. If y don't belong to the list x, the initial list x is returned."

SplitList::usage =
"The call SplitList[L, x] returns the result of splitting of an initial list L onto sublists by an element or elements x; in addition, dividers x from the result are removed. If elements x don't belong to the list L, the procedure call returns the initial list L."

SplitList1::usage =
"The call SplitList1[L, y, z] returns the sublists of a list L that are limited by its sublists y and z; in addition, dividers y and z from the result are removed. If elements y,z don't belong to the list L, the procedure call returns the empty list, i.e. {}."

MultEntryList::usage =
"The call MultEntryList[x] returns the ListList–list; the first element of its sublists defines the element of a list x whereas the second element defines multiplicity of it in the list x irrespective of its nesting. If the list x is empty, the call SubLists[x] returns the empty list, i.e. {}."

ReloadPackage::usage =
"The call of procedure ReloadPackage[x] returns nothing, providing activation in the current session of all means

of a package located in a m−file x as though their definitions were calculated in input stream. If call ReloadPackage[x, y] contains the second optional y−argument − the list of names − the reload is made only for means of the package with names specified in y. The call ReloadPackage[x, y, t] additionally with third optional argument t, where t is an arbitrary expression, returns nothing also, providing the reload in the current session of all means of the package x, excluding only means of the package with names, specified in the list y."

ReloadPackage1::usage =
"The ReloadPackage1 procedure is a functionally equivalent modification of the ReloadPackage procedure. This modification is of interest from the point of view of approaches used by the procedure. The call of procedure ReloadPackage1[x] returns nothing, providing activation in the current session of all means of a package located in a m−file x as though their definitions were calculated in input stream. If call ReloadPackage1[x, y] contains the second optional y−argument − the list of names − the reload is made only for means of the package with names specified in y. The call ReloadPackage1[x, y, t] additionally with third optional argument t, where t is an arbitrary expression, returns nothing also, providing the reload in the current session of all means of the package x, excluding only means of the package with names, specified in the list y."

ConvertMtoMx::usage =
"In [4−6,8] the organization of the user package, simple and convenient for modifications and saved in a m−file by means of chain of the GUI commands \"File −> Save As −> Mathematica Package (∗.m)\" is presented. The \"AVZ_Package.m\" package was organized and modified exactly in such way [10]. The procedure call ConvertMtoMx[x, y] returns Null, providing converting of a m−file x created by the above method into the datafile with the same main name, but with \"mx\" extension; through the 2nd argument y − a symbol − the list of names in string format of objects which are contained in the m−file x with a package is returned. Whereas the procedure call ConvertMtoMx[x, y, z] with the 3rd optional argument z − an arbitrary expression − in addition unload a package contained in datafile x from the current session."

ContUsageMfile::usage =
"In [4−6,8] the organization of the user package, simple and convenient for modifications and saved in a m−file by means of chain of the GUI commands \"File −> Save As −> Mathematica Package (∗.m)\" is presented. The present package was organized and modified exactly in such way. The function call ContUsageMfile[x] returns the sorted list of names in string format of objects which are contained in the m−file x with a package. Whereas the function call ContUsageMfile[x, y] returns the usage in string format of an object which is contained in a m−file x with a package and has name y."

CodeObjMfile::usage =
"In [4−6,8] the organization of the user package, simple and convenient for modifications and saved in a m−file by means of chain of the GUI commands \"File −> Save As −> Mathematica Package (∗.m)\" is presented. The present package was organized and modified exactly in such way. The function call CodeObjMfile[x, y] returns the source code in string format of an object with name y that is contained in the m−file x with a package organized by the above manner."

ContCodeUsageM::usage =
"In [4−6,8] the organization of the user package, simple and convenient for modifications and saved in a m−file by means of chain of the GUI commands \"File −> Save As −> Mathematica Package (∗.m)\" is presented. The procedure call ContCodeUsageM[x] returns the sorted list of names in string format of objects which are contained in the m−file x with a package. The procedure call ContCodeUsageM[x, y] returns the usage in string format of an object which is contained in a m−file x with a package and has name y. Whereas the procedure call ContCodeUsageM[x, y, z] with the third optional argument z − an arbitrary expression − returns the source code in string format of an object which is contained in a m−file x with a package and has name y."

LoadMyPackage::usage =
"The call of procedure LoadMyPackage[x, y] returns Null, i.e. nothing, providing the loading into the current session of a package located in a mx−file x with a context y with subsequent re−evaluation of all definitions contained in the package, what provides optimal format of all such definitions, i.e. without context."

LoadPackage::usage =
"The call LoadPackage[x] returns Null, i.e. nothing, loading a package contained in a datafile x of the mx−format, into the current session of Mathematica with activation of all definitions contained in it in a mode similar to the mode of the Input−paragraph of Mathematica system."

ExtrOfMfile::usage =
"The call ExtrOfMfile[x, y] returns nothing, loading into the current session the definitions of only those means which are defined by argument y and are in the m−format datafile x with a package. In addition, in case of existence in the m−datafile of several means of the same name, the last from them is loaded into the current session only. Whereas the call ExtrOfMfile[x, y, z] with the third optional argument z − an indefinite variable − through z additionally returns the list of definitions of means from y being in the m−datafile x with package. In absence in the m−datafile x of means from y, the procedure call returns $Failed."

ExtrDefFromM::usage =
"To certain extent, to the ExtrOfMfile procedure the following procedure adjoins whose call ExtrDefFromM[x, y] in tabular form returns the usage and definition of a y tool contained in a x m−file with the user package. At the same time, in the absence of one of these components of a y tool the message \"Usage for y is absent\" or \"Definition for y is absent\" is returned accordingly; while in the absence of the both components $Failed is returned. At last, the procedure call ExtrDefFromM[x, y, z] with the third optional argument z − an arbitrary expression − additionally in the current session the usage and definition of y tool are evaluated. We will note that procedure doesn't demand

loading of x package into the current session, allowing in it selectively to activate the tools of the user package."

ExtrFromNBfile::usage =
"The successful call ExtrFromNBfile[x, y] returns an object definition in string format with a name y, given in string format, from a datafile x of the format {\"cdf\", \"nb\"}, at the same time activating this definition in the current session. Otherwise, the procedure call returns $Failed. In addition, the loading in the current session of the datafile x isn't required."

ExtrFromNBfile3::usage =
"The procedure call ExtrFromNBfile3[x, n] returns in string format the definition of an object located in a x file of format {\"cdf\", \"nb\"} that is defined by a name n in string format or by their list; if the names list is as an argument n, the list of their definitions is returned generally speaking. Whereas the procedure call ExtrFromNBfile3[x, n, y] with third optional y argument – an undefined variable – through it returns the list of elements of the kind {mnw, $Failed} that defines names mnw whose definitions are absent in the x file with the user package. It is assumed that the user package located in the x file is arranged in the above format, and tools definitions of the same name which are in it, are in different blocks of the format \"Begin[...] ... End[]\". In some cases the approach used in this procedure seems an useful enough."

CdfNbToText::usage =
"In some cases the approach used in the following procedure may be an useful enough means at programming of tools based on the internal format {\"cdf\", \"nb\"}. The procedure call CdfNbToText[x] returns full path to txt−file which contains in text format the content of x file of the format {\"cdf\", \"nb\"}. Whereas the call CdfNbToText[x, y] with second optional argument – an arbitrary expression – returns the 2−element list whose first element is the above path to the txt−file, and the second element – the content of this txt−file, including system characteristics of document contained in the x file."

ExtrFromM::usage =
"The procedure call ExtrFromM[x, y] returns nothing, evaluating in the current session a separate means or their list whose names are defined by argument y that are supplied with usages and whose definitions are in a datafile x of the \"m\" format. Whereas the procedure call ExtrFromM[x, y, z] where z – an undefinite symbol – through argument z returns the four−element list whose elements are names lists of y that – (1) have definitions and usages, (2) have usages without definitions, (3) have definitions without usages, (4) have neither definitions or usages. At that, uploading of the file x into the current session isn't required."

ExtrFromNBfile1::usage =
"The successful procedure call ExtrFromNBfile1[x, y] returns the definition of an object in the DisplayForm format with a name y given in string format from an unuploaded datafile x of format {\"cdf\", \"nb\"}, activating the object y in the current Mathematica session; otherwise, the call returns the empty list, i.e. {}, whereas the call ExtrFromNBfile1[x, y, t], where t – an undefinite variable through t additionally returns full path to cdf−file with definition of the object y. Qua of an useful property of this procedure is the circumstance that a datafile x not require of uploading into the current session."

ExtrFromNBfile2::usage =
"The procedure call ExtrFromNBfile2[x, n] returns the definition of an object with a name n given in string format from the unuploaded or uploaded user package located in a datafile x of format {\"cdf\", \"nb\"}. At the same time, definition of an object n saves in the \"n.cdf\" file of the the current working directory. At that, the datafile x not require of uploading into the current session. If the user package located in the file x of the format {\"cdf\", \"nb\"} has been loaded into the current session before ExtrFromNBfile2 procedure call, it remains active in the current session, otherwise the user package will be unloaded after the procedure call. The unsuccessful procedure call returns $Failed or is returned unevaluated."

EvaluateCdfNbFile::usage =
"The function call EvaluateCdfNbFile[w] returns nothing, calculating in the current session a file w of the format {\"cdf\", \"nb\"} without its visualization and uploading in the current session. While the call EvaluateCdfNbFile[w, y] with the second optional argument y – an arbitrary expression – does the above actions, visualizing additionally the datafile w."

ContentsCdfNb::usage =
"The procedure call ContentsCdfNb[x] returns the sorted list of objects names whose definitions are located in a file x of the format {\"cdf\", \"nb\"} without uploading of datafile x into the current session. The procedure allows to obtain the contents of the {\"cdf\", \"nb\"}−files without uploading their into the current session. In many applications, the given tool is quite useful in a program mode."

ContentsMx::usage =
"The procedure call ContentsMx[x] returns the list of means names contained in a mx−file x with a context; the x file is not necessarily loaded into the current session; if the x file has been already uploaded into the current session, then the file remains in it, otherwise the file is unloaded from it."

ContentOfNbCdf::usage =
"As it was already noted earlier, definitions of the user package without the usages corresponding to them, are ignored at uploading of the package into the current session. Thus, the means testing such situations are presented to us as rather important. So, the procedure call ContentOfNbCdf[w] returns the sorted list of objects names whose definitions are located in a datafile w of the format {\"cdf\", \"nb\"} without uploading of datafile w into the current session. Whereas the procedure call ContentOfNbCdf[w, y] where y – an undefinite symbol – additionally through y returns 3−element list whose the first element represents the sorted list of objects names whose definitions are located in the file w and have usages, the second

element represents the sorted list of objects names whose definitions are located in the file w and have not usages, the third element presents the sorted list of objects names which have not the corresponding definitions. The given procedure allows to verify the contents of the {\"cdf\", \"nb\"}–files without uploading their into the current Mathematica session."

UsagesCdfNb::usage =
"The procedure call UsagesCdfNb[x, y] provides usages output relative to the means defined by a separate symbol or their list y that are in a file x of the format {\"cdf\", \"nb\"} without its uploading into the current session. For a single means the result is returned in the format \"y::usage\" or \"y::No\" if the usage for a means with name N is absent in the datafile x."

ActivateMeansFromCdfNb::usage =
"The procedure call ActivateMeansFromCdfNb[x, y] returns nothing, evaluating in the current session a separate means or their list whose names are defined by argument y along with their usages whose definitions are in a datafile x of the format {\"cdf\", \"nb\"}. At that, uploading of the datafile x into the current session isn't required."

TestCdfNbFile::usage =
"It is known that definitions of means of the user packages kept in files of the format {\"cdf\", \"nb\"}, have to be supplied with usages of the corresponding format. The given requirement is obligatory for ensuring availability of these means at evaluation of such datafiles in the current session. In this connexion it is expedient to have the means for testing of supply of means of such {\"cdf\", \"nb\"}–files with the corresponding usages. The procedure call TestCdfNbFile[x] returns the sorted list of means of a file x of the format {\"cdf\", \"nb\"} which have not usages; otherwise the empty list, i.e. {} is returned. Whereas the procedure call TestCdfNbFile[x, y] where y – an undefinite variable – through y additionally returns the list of means names, whose definition are in the datafile x with the user package. It should be noted that testing of the above datafiles of the format {\"cdf\", \"nb\"} is carried out without their uploading into the current session."

ExtrFromMfile::usage =
"The next ExtrFromMfile procedure is specific complement of the ExtrFromNBfile procedure, providing extraction of definitions of functions and procedures along with their usages from an unuploaded package that is located in a datafile of m–format. The procedure call ExtrFromMfile[x, y] returns the definition of an object in the string format with a name or list of their names y given in string format from an unuploaded file x of m–format, at the same time activating these definitions and usages corresponding to them in the current session; otherwise, the call returns empty list, i.e. {}."

MfilePackageQ::usage =
"The call MfilePackageQ[x] returns True only in case a string x defines the real datafile of m–format that is the standard package."

PackageFileQ::usage = "The call PackageFileQ[x] returns True if the argument x defines a datafile of formats {\"cdf\", \"mx\", \"m\", \"nb\"} with a package, otherwise False is returned."

UserPackTempVars::usage =
"As a rule, rather large packages of the user contain in own structure the variables of several types which appear at their loading into the current session of the system. For definition of such variables the procedure can be used, whose call UserPackTempVars[x] returns the 3–element nested list where the first sublist determines the undefinite variables associated with the package determined by a context x, the second sublist defines the temporary variables associated with the package and having names of the format \"Name$\" whereas the third sublist defines symbols of the format \"Name$Integer\" which in the current session aren't distinguished as symbols."

TempInPack::usage =
"The function call TempInPack[x] returns the list of the temporary variables associated with the package defined by a context x. The TempInPack function, based on the TemporaryQ function is a simplified version of the UserPackTempVars procedure."

$UserContexts::usage =
"The global variable $UserContexts defining a list of contexts of the user packages loaded into the current session completes the given fragment. At that, the variable defines only contexts of the packages that generate in the current session the variables of two types presented above according to the UserPackTempVars procedure."

ContMxFile::usage =
"The call ContMxFile[x] returns the nested list, whose first element defines the context associated with a package that is contained in a mx–datafile x while the second element defines the list of names in string format of all objects of this package irrespective of existence of the references (usages) for them, i.e. both local, and global objects. Whereas the call ContMxFile[x, y] where the second argument y – an arbitrary expression – returns the nested list of analogous structure, but with the difference, that its second element defines the list of names of objects of this package that are provided with references (usages), i.e. only global objects. Withal, it should be noted that ContMxFile procedure presented in the previous fragment is intended for usage with the mx–files created on platform Windows XP/7 Professional, its use for other platforms can demand the appropriate adaptation. The reason of it consists in that the algorithm of the ContMxFile procedure is based on an analysis of structure of mx–files that depends on platform used at creation of such datafiles."

ContMxFile1::usage =
"The procedure ContMxFile1 is an useful enough modification of the ContMxFile procedure which also uses an analysis of structure of mx–files which depends on platform used at creation of such datafiles. The procedure call ContMxFile1[x] returns the nested list whose first element defines the context associated with the package contained

in a mx–datafile x while the second element determines the list of names in string format of all objects of this package irrespectively from existence for them of usages, i.e. local and global objects. Furthermore, similarly to the ContMxFile procedure the returned names determine objects whose definition returned by the call Definition contains the context. At that, is supposed that a mx–datafile x is recognized by the FileExistsQ function."

ContMxFile2::usage =
"Unlike the procedures ContMxFile and ContMxFile1, the ContMxFile2 procedure is based on another algorithm whose essence is as follows. First of all the existence in a mx–file x of a package is checked; at its absence $Failed is returned. Then upload in the current session of a package containing in the mx–file x is checked. At positive result the required result without unloading of a package x is returned, otherwise the required result with unloading of a package is returned. In both cases a call ContMxFile2[x] returns the 2–element list, whose first element determines a package context whereas the second – the list of names in string format of means, contained in the package. The procedure essentially uses the IsPackageQ procedure."

ContMxW7::usage =
"For the platform Windows 7 Professional the algorithm of the ContMxFile procedure is modified in the corresponding manner, taking into account the internal structure of the mx–datafiles created on the specified platform. This algorithm is realized by the procedure ContMxW7, whose call ContMxW7[x] returns the nested list whose first element defines the context connected with the package contained in a mx–file x whereas the second element defines the list of names in string format of all global objects of this package whose definitions contains a context ascribed to the package. Whereas on a mx–file without context the procedure call returns $Failed. At that, is supposed that a file x is recognized by the FileExistsQ function."

DiffContexts::usage =
"The objects of the same name have various headings therefore in certain cases arises a question of their more exact identification. The next procedure provides one of such approaches, trying to associate the components composing such objects with the contexts ascribed to them. At the heart of the procedure algorithm lies a principle of creation for separate components of an object of the same name of packages in m–files with the unique contexts ascribed to them. Then, having removed an object x of the same name from the current session, by means of uploading of these m–files into the current session we have opportunity of access to components of the object x of the same name through a construction of the \"Context'x\" format.
The procedure call DiffContexts[x] returns the nested list of ListList–type whose sublists by the first element define context while the second element define heading of a certain component of an object of the same name x in the format {{\"xn'\", \"cn'\"}, …, {\"x2'\", \"c2'\"}, {\"x1'\", \"c1'\"}} whose order is defined by order of the contexts in the list defined by the $Packages variable, where n – number of components of the object of the same name x. Moreover, the datafiles \"xj.m\" with the packages with components definitions composing the object of the same name x remain in the current directory of the session (j=1..n). At the same time the procedure call DiffContexts[x, y] with the 2nd argument y – an arbitrary expression – returns the above result, removing the intermediate m–files. Whereas on x objects different from objects of the same name the procedure call DiffContexts[x] returns the context of an object x."

MxPackNames::usage =
"After loading in the current session of the user package from a datafile of format {m, nb}, the most part of definitions of its means received by the standard Definition function, will include context links of the format \"Context'x'\", where x – the name of a means, and \"Context'\" – the context, ascribed to this package. The call MxPackNames[x] returns the list of names of means of nb–file y – analog of the datafile x – whose definitions after loading of this datafile y into the current session the system function Definition will return with context links of the above format."

NamesFromMx::usage =
"The call NamesFromMx[x] returns the list of names of the means whose definitions are in a datafile of the mx–format with a package. In case this package wasn't loaded into the current session, the call NamesFromMx leaves it unloaded."

PackReplaceQ::usage =
"The procedure call PackReplaceQ[x] returns True, if definitions contained in a m–file x with package replace the definitions activated in the current session, and False otherwise at uploading of the file x into current session. At that, the procedure call leaves the datafile x unloaded. Whereas the call PackReplaceQ[x, y] through optional argument y – an undefinite variable – returns the list of objects of the current session whose definitions will be replaced at uploading of the file x into current session."

NamesFromMx1::usage =
"The call NamesFromMx1[x] returns the list of names of the means whose definitions are in a datafile x of the mx–format with a package. Procedure doesn't demand the loading of datafile x into the current session. In addition, only those names of the means whose definitions received by the call of standard function Definition contain a context associated with the package are returned."

NamesFromMx2::usage =
"The call NamesFromMx2[x] returns the list of names of the means whose definitions are in a datafile x of the mx–format with a package. Procedure doesn't demand the loading of datafile x into the current session. In contrast to the NamesFromMx1 procedure, the NamesFromMx2 returns the list of names of the means irrespectively from context existence in their definitions returned by the Definition function."

$ProcName::usage =
"The procedural variable $ProcName which is used only in the body of a procedure activated in the current session, returns the list,

whose first element defines a name, whereas the second – the heading in string format of the procedure containing it. Moreover, for maintenance of the given opportunity in the list of local variables of the procedure containing the variable $ProcName, it is necessary to code expression of the kind $$NameProc$$ = \"Name_Procedure\"; otherwise, the call of procedure returns \"UndefinedName\" as a value for the variable $ProcName."

$InBlockMod::usage =
"The call of the procedural $InBlockMod variable in a procedure body of the type Block or Module in string–format returns an initial code of the procedure without its heading in a point of its call. When using the procedural $InBlockMod variable it must be kept in mind that it makes sense only in a procedures body of the type Block or Module, without returning anything, i.e. Null, in other expressions or in the Input–paragraph."

MathematicaDF::usage =
"The call MathematicaDF[] returns the list of the ListList–type whose 2–element members by the first element contain type of an element of Mathematica file system whereas by the second element the quantity of elements of this type. In addition, \"Dir\" defines the catalog, \"NoExtension\" – files without extension, whereas others – type of file extension (in particular, datafiles with name of the format \".xxxxxx\" also are understood as extension)."

Memory::usage =
"The successful call Memory[] provides return of the list, whose elements define the structural organization of a computer memory. While the call Memory[x] with optional argument x – an arbitrary expression – in addition deletes the program mem.exe from the catalog determined by system variable $InstallationDirectory."

DefWithContext::usage =
"The call DefWithContext[x] returns the 2–element nested list: its first element defines the list of names in string–format of means of a package loaded from a m–file x, whose definitions do not contain context identifiers whereas the second – the list of names in string–format of means whose definitions contain context identifiers."

ContextMXfile::usage =
"The call ContextMXfile[x] returns the context associated with the package contained in a datafile x of mx–format. Furthermore, the uploading of the package into the current session is not needed."

ContextInMxFile::usage =
"The procedure call ContextInMxFile[x] returns the context that is associated with a mx–file x. In the absence of a context the procedure call returns $Failed. Furthermore, the uploading of the datafile x into the current session is not needed."

ContextFromMx::usage =
"The procedure call ContextFromMx[x] returns the context ascribed to the user package contained in a mx–file, at a context absence $Failed is returned. The procedure operates on platform Windows XP Professional and Windows 7 Professional. Moreover, performance of the procedure is higher if it is applied to a mx–file created on the current platform."

ContextsInFiles::usage =
"The procedure ContextsInFiles provides evaluation of the context ascribed to a datafile of the format {\"m\", \"mx\", \"cdf\", \"nb\"}. The procedure call ContextsInFiles[w] returns the single context ascribed to a file w of the above formats. In the absence of a context the call returns $Failed. At that, it must be kept in mind that the context in datafiles of the specified format is sought relative to the key word \"BeginPackage\", which is typically used at the beginning of a Mathematica package. A return of the list of the format {\"Context1\", \"Context2\", …, \"Contextp\"} is equivalent to existence in the file of m–format of construction BeginPackage[\"context1'\", {\"context2'\", ..., \"contextp'\"}] where {\"context2'\", ..., \"contextp'\"} define uploadings of the appropriate files if their contexts aren't in the $Packages variable."

ContextInMfile::usage =
"The procedure ContextInMfile provides evaluation of all contexts contained in a datafile of the m–format. The procedure call ContextInMfile[x] returns the list of contexts contained in a m–file x. In the absence of context the call returns the empty list, i.e. {}. At that, it must be kept in mind that contexts in m–files are sought relative to the key words {\"BeginPackage\", \"Needs\", \"Get\", \"Package\"} which are used in m–files."

ContextFromMx1::usage =
"The procedure call ContextFromMx1[x] returns the context ascribed to the user package contained in a mx–file, at a context absence $Failed is returned. The procedure operates on platform Windows XP Professional and Windows 7 Professional. Moreover, performance of the procedure is higher if it is applied to a mx–file created on the current platform."

ContextFromFile::usage =
"The call ContextFromFile[x] returns the context associated with the user package contained in a datafile x of formats {cdf, m, mx, nb}, and $Failed otherwise. Furthermore, the loading of the package into the current session is not done and the file x remains closed."

ContextDef::usage = "The call ContextDef[x] returns the list of contexts associated with a symbol x. If x isn't associated with a context, the empty list is returned, i.e. {}."

ContentOfMfile::usage =

"The call ContentOfMfile[f] returns the list of names in string–format of all means, whose definitions are in a package (m–datafile) determined by argument f. In absence in the m–datafile of definitions of means in the standard package format the call returns the empty list, i.e. { }."

ContentOfMfile1::usage = "The function ContentOfMfile1 is functionally equivalent to the procedure ContentOfMfile, however has a rather other implementation."

ContentOfMfile2::usage = "The function ContentOfMfile2 is functionally equivalent to the procedure ContentOfMfile1, however has a rather other implementation."

ContextToFileName1::usage =
"The procedure ContextToFileName1 is useful enough addition to the standard function ContextToFileName. The procedure call ContextToFileName1[] returns the list of contexts associated with system m–files, whereas the procedure call ContextToFileName1[y] returns the full path to m–file associated with a context y."

SymbolsContext::usage =
"The function call SymbolsContext[x] returns the list of all symbols associated with a context x that is activated in the current session. In particular, the function is useful enough at receiving of names of all main objects contained in the package with the given context x."

VizContext::usage =
"The call VizContext[x] returns the notebook containing all definitions of the user package that is downloaded in the current session and associated with a context x. The notebook is returned in the separate window, providing comfortable scan of definitions with possibility of their subsequent saving in a datafile of formats {cdf, m, nb, ...}. In case of absence of the context x in the current session the procedure call returns $Failed."

VizContentsNB::usage =
"A quite simple VizContentsNB procedure provides the print of a datafile of the nb–format in a readable view that is quite convenient for expeditious viewing of contents, in particular, of a package containing in the nb–datafile without its loading into the current session. In addition, the call VizContentsNB[w] in the readable view outputs the contents of nb–datafile w onto the monitor, otherwise $Failed is returned."

ToContextPath::usage =
"The call ToContextPath[x] provides updating of contents of the current list determined by the system variable $ContextPath by means of adding to its end of all contexts from a m–file x containing simple or the nested package. The successful call ToContextPath[x] returns the updated value of $ContextPath variable."

DeleteOptsAttr::usage =
"The call DeleteOptsAttr[x] returns Null, i.e. nothing, deleting the options ascribed to a symbol x; whereas the call DeleteOptsAttr[x, y] returns Null also, i.e. nothing, deleting both the options and attributes ascribed to a symbol x; in addition, an arbitrary expression can be used as the second optional argument y."

DefFromPackage::usage =
"The call DefFromPackage[x] returns the 3–element list: its first element – the definition in string–format of a symbol x whose context is different from {\"Global`\", \"System`\"}, the second element – help (usage) concerning the symbol x, and the third element – the list of attributes of the symbol x. On symbols associated with the above contexts the procedure call returns the list of their attributes."

DefFromM::usage =
"The call DefFromM[x, y] returns the object definition with a name y which is in a datafile x of the m–format with a package, whereas the call DefFromM[x, y, z] where z – an arbitrary expression, in addition evaluates this definition in the current session, making object y available."

MxToTxt::usage =
"The procedure MxToTxt allows from 2 to 4 actual arguments. The procedure call MxToTxt[x, y] returns Null, i.e. nothing, saving both in a datafile y of txt–format, and in the current session all definitions of the user package being in a datafile x of mx–format. In addition, definitions of the file x remain in the optimal format (without the context associated with the package). If the call MxToTxt[x, y, z], since the 3rd argument, contains optional argument \"Del\", the package x isn't loaded into the current session, otherwise its definitions remain in the current session in the optimal format. If at call arguments of procedure, since the third, contain an undefinite variable, through it the list of all objects whose definitions are in the datafile x with the user package are returned."

MxToTxt1::usage =
"The procedure MxToTxt1 is a modification of the above procedure MxToTxt. The procedure MxToTxt1 allows from 2 to 4 actual arguments too. The procedure call MxToTxt1[x, y] returns Null, i.e. nothing, saving both in a datafile y of txt–format, and in the current session all definitions of the user package being in a datafile x of mx–format. In addition, definitions of the file x remain in the optimal format (without the context associated with the package). If the call MxToTxt1[x, y, z], since the 3rd argument, contains optional argument \"Del\", the package x isn't loaded into the current session, otherwise its definitions remain in the current session in the optimal format. If at call arguments of procedure, since the

third, contain an undefinite variable, through it the list of all objects whose definitions are in the datafile x with the user package are returned. The call Get[y] returns \"OK!\" activating in the current session all definitions from the datafile y."

MxToTxt2::usage =
"The MxToTxt2 procedure represents a quite useful modification of the procedures MxToTxt and MxToTxt1, allowing 2 factual arguments and saving in ASCII–datafile which is defined by the second argument, all definitions contained in a mx–file which is defined by the first argument. The call MxToTxt2[x, y] returns \"OK!\", saving in the datafile y all definitions from the mx–file x, that are parted by string \"OK!\". In addition, if the package from the mx–file x wasn't loaded into the current session, then and after the call MxToTxt2 the package will be inaccessible in the current session."

MxToMpackage::usage =
"The MxToMpackage procedure provides converting of the package that is in a mx–file into the package of the format presented here and in our books [4–6,8] which is rather convenient at creation of the user packages. The procedure call MxToMpackage[x] returns the path to the file FileBaseName[x] <> \".m\" which will contain the package of the above format contained in a mx–file x, while the procedure call MxToMpackage[x, y] returns the path to the file FileBaseName[y] <> \".m\"; at that, if a package from the mx–file x yet has been uploaded into the current session, then it remains in it, otherwise it is unloaded from the current session. At impossibility of such converting, the procedure call is returned unevaluated or returns $Failed."

MxFileToMfile::usage =
"The MxFileToMfile procedure provides converting of a package located in a datafile of the mx–format, into datafile of the m–format. The call MxFileToMfile[x, y] returns the path to a datafile y – result of converting of the mx–file x with the package into the datafile of the m–format. Moreover, the procedure call deletes packages x and y from the current session."

MfileToMx::usage =
"The MfileToMx procedure provides converting of a package located in a datafile of the m–format, into datafile of the mx–format. The call MfileToMx[x] returns the path to a datafile – result of converting of the m–datafile x with the package into the datafile of the mx–format whose name coincides with a name of the initial datafile x with replacement of its extension \"m\" on \"mx\". Moreover, the procedure call deletes package x from the current session if until the call MfileToMx the datafile wasn't loaded into the current session, otherwise not."

HeadToCall::usage =
"The call HeadToCall[h] in string format returns the call of a procedure/function on the basis of its heading on `pure` formal arguments (i.e. without the tests for an admissibility ascribed to them), where h – admissible heading of a procedure/function."

CallListable::usage =
"The call CallListable[x, y] returns the list of values Map[x, Flatten[{y}]], where x – a block, a function or a module from one formal argument, and y – the list or sequence of the factual arguments which can be and empty."

DefOnHead::usage =
"The call DefOnHead[h] returns the list whose the first element is the definition in string–format of a procedure/function with the heading h (or list of definitions for the objects of the same name), whereas the others – options (if they exist) and the list of attributes ascribed to the given procedure/function. In case of impossibility to evaluate the definition the call returns $Failed."

CompActPF::usage =
"The call CompActPF[x] returns the nested 2–element list whose the first element contains the list of all blocks, functions and modules, composing a block/function/modules x, whereas the second element contains the list of headings corresponding to them. In addition, into the list are included only means whose definition were activated in the current session. For calls entering into object x are added also and all their calls on the full depth of multiplicity."

CompActPF1::usage =
"The call CompActPF1[x] returns the nested list whose elements represent sublists of the following format, namely:
– the sublist with the first \"System\" element defines the calls of system functions in definition of a block, function or a module x;
– the sublist with the first \"Undefined\" element defines names of objects which aren't entered into the list of arguments and local variables of a block, function or a module x;
– the sublist of a format, different from above–stated, contains the user couples {block/function/module, its heading}, whose calls are available in definition of an object x."

FuncToPure::usage =
"The procedure FuncToPure provides converting of a function defined by the format G[x _, y _,...] := W(x, y,...), into pure function of any admissible format, namely: the call FuncToPure[x] returns the pure function being analog of a function x, of the third format (short format), whereas the call FuncToPure[x, p] where p – an arbitrary expression, returns the pure function of the two first formats."

PureToFunc::usage =
"The following procedure in a certain measure is inverse to the FuncToPure procedure, its call PureToFunc[x, y] where x – definition of a pure function, and y – an unevaluated symbol – returns Null, i.e. nothing, providing converting of definition of a pure function x into the evaluated definition of equivalent function with

a name y. In addition, on inadmissible actual arguments the procedure call is returned unevaluated."

ModToPureFunc::usage = "The call ModToPureFunc[x] provides the converting of a module or a block x into the corresponding pure function under the following conditions, namely: (1) the module/block x can't have local variables or all its local variables have the initial values; (2) the module/block x can't have active global variables, i.e. variables to which in object x assignments are made; (3) formal arguments of the returned function don't keep tests for admissibility of corresponding factual arguments; (4) the returned function inherits attributes and options of the object x. The successful call ModToPureFunc[x] returns the name of the returned pure function in the form ToString[Unique[x]], otherwise the call returns the enclosed list of the format {$Failed, {\"Locals\"|\"Globals\"}, {the list of variables string format}} whose the first element $Failed defines inadmissibility of the above converting, the second element – type of the variables whose were at the bottom of it, and the third element – the list of variables of this type in string format."

SyntCorProcQ::usage =
"The call of procedure SyntCorProcQ[x] returns True if definition of a procedure x, activated in the current session, is syntactically correct in the context, presented in our book [4]; otherwise, False is returned. If x – not a procedure the call is returned unevaluated."

RealProcQ::usage =
"A real procedure is understood as an object of type {Module, Block} which in the software environment of Mathematica is functionally equivalent to Module, i.e. to the procedure in its classical understanding. The call RealProcQ[x] returns True if the symbol x defines a Module or a Block which is equivalent to a Module, and False otherwise. At that, it is supposed that a certain block is equivalent to a module if all its local variables have initial values or some local variables have initial values while others obtain values by the operator \"=\" in the block body. From all our means solving the problem of testing of the procedural objects, the above RealProcQ procedure with the greatest possible reliability identifies the procedure in its classical understanding; in addition, the procedure can be of type {Module, Block}."

TestBFM::usage =
"The call TestBFM[x] in the format \"Module\", \"Block\", \"Function\" or \"DynamicModule\" returns the type of procedural or functional object x; whereas on argument x of other types the procedure call returns &Failed. Moreover, if x defines an object of the same name, the call TestBFM[x] returns the list of types of the subobjects composing it, that has one–to–one correspondence with the list of definitions returned by the call PureDefinition[x]."

TestArgsCall::usage =
"The call TestArgsCall[x, y] returns the definition or list of definitions of a block/function/module x on which the procedure call with tuple of factual arguments y is correct; i.e. their types corresponds to admissible types of formal arguments. Otherwise, the call returns the empty list, i.e. {}; on inadmissible argument x different from a lock/function/module the procedure call is returned unevaluated."

TestArgsLocals::usage =
"As noted in [9], the Mathematica system at evaluation of definitions of objects of types {Block, Function, Module} does not identify situations of duplication of formal arguments, local variables and/or their intersections as clearly illustrates the examples in [9]. These situations are tested by means of the procedure whose call TestArgsLocals[x] returns False if a block/function/module x has a duplication of formal arguments, local variables and/or their intersections, and True otherwise. Whereas the procedure call TestArgsLocals[x, y] through the second optional argument y – an undefinite symbol – in case of the main return False returns the 2–element list whose elements–sublists are in one–to–one correspondence. The first element defines the definitions, composing the x object, for which the above situations exist, whereas the second element defines the list of sublists of the format {t1, t2, t3}; (tj ∈ {True, False}; j=1÷3); its tj elements define the duplication of formal arguments, duplication of local variables, and/or their intersections {False/True} accordingly."

TestFactArgs::usage =
"The procedure call TestFactArgs[x, y] returns the list from True and False that defines who of the actual arguments determined by a sequence y will be admissible in the call x[y], where x – an object name with a heading (block, function, module). The procedure assumes equal number of the formal and actual arguments defined by a sequence y, along with existence for an object x of the fixed number of arguments; otherwise, the call TestFactArgs returns $Failed."

FormalArgs::usage = "The call FormalArgs[x] returns the list of formal arguments of a heading x irrespective of the definition assigned to it. On an inadmissible argument x the call returns $Failed."

SyntaxLength1::usage =
"The call of procedure SyntaxLength1[x] returns the maximal number of position in a string x such that ToExpression[StringTake[x, {1, p}]] is a syntactically correct expression, and 0 otherwise, whereas the call SyntaxLength1[x, y] through second optional argument y – an indefinite variable – additionally returns the list containing substrings of the string x which represent syntactically correct expressions."

ProcFuncCS::usage =
"The call of function ProcFuncCS[] returns the 3–element nested list, whose sublists define names in string–format of the user blocks, functions and modules accordingly whose definitions have been evaluated in the current session."

ProcActCallsQ::usage =
"The procedure tests the existence in the user block, function or module x of calls of active user software provided by standard

supplemental information (usage). The call ProcActCallsQ[x] returns True, if the definition of a block/function/module x contains the calls of such software. In addition, through the second optional argument y – an undefinite variable – the call ProcActCallsQ[x, y] returns the exhaustive list of software whose calls are contained in the definition of x."

CompileFuncQ::usage = "The call CompileFuncQ[x] returns True if x represents a Compile function, and False otherwise."

GC::usage = "The call GC[x] returns the unique decimal code of an arbitrary x–expression."

$AobjNobj::usage = "Global variable for certain package means, in particular, Aobj1."

ArgsTypes::usage =
"Procedure ArgsTypes serves for testing formal arguments of a block/function/module, including pure functions and Compile functions, which are active in the current session of Mathematica. The call of procedure ArgsTypes[x] returns the nested list, whose 2–element sublists in string–format define names of formal arguments and their allowable types (and in a more comprehensive sense tests their for admissibility and initial values by default) accordingly. In case of absence for an argument of type it is defined as \"Arbitrary\" whereas one actual argument initiates return of the simple list of the mentioned format; on inadmissible formal argument the call is returned unevaluated. In addition, the Args procedure processes the situation \" the objects of the same name with various headings \", returning the nested list of formal arguments concerning of all subobjects of an object x in the order determined by the function Definition."

FilesDistrDirs::usage =
"The FilesDistrDirs procedure in a certain measure bears structural character for a directory which has been defined by its factual argument. The call FilesDistrDirs[x] returns the nested list, whose elements are sublists of the following format {dir_p, f1, f2, ..., fn}, where dir_p – the catalog x and all its subdirectories of any nesting level, whereas f1, f2, ..., fn – names of the datafiles located exactly in this catalog."

QmultiplePF::usage =
"The procedure call QmultiplePF[x] returns True, if x – an object of the same name (block, function, module), and False otherwise. While the procedure call QmultiplePF[x, y] with the 2nd optional argument y – an indefinite variable – returns through y the list of definitions of all subobjects with a name x."

FindFile1::usage =
"The FindFile1 procedure serves as useful extension of the standard FindFile function, providing search of a datafile within file system of the computer. The procedure call FindFile1[x] returns a full path to the found datafile x, or the list of full paths (if datafile x is located in different directories of file system of the computer), otherwise the call returns the empty list, i.e. {}. While the call FindFile1[x, y] with the second optional argument y – full path to a directory – returns a full path to the found datafile x, or the list of full paths located in the directory y and its subdirectories."

FindFile2::usage =
"The FindFile2 function serves as an useful extension of the standard FindFile function, providing search of a y datafile within a x directory. The function call FindFile2[x, y] returns the list of full paths to the found datafile y, otherwise the call returns the empty list, i.e. {}. Search is done in the directory x and all its subdirectories."

DeleteFile1::usage =
"Removal of a datafile in the current session is made by means of the standard DeleteFile function whose call DeleteFile[{x, y, z,...}] returns Null, i.e. nothing in case of successful removal of the given datafile or their list, and $Failed otherwise. At that, in the list of datafiles only those are deleted that have no Protected–attribute. Moreover, this operation doesn't save the deleted datafiles in the system Recycle Bin directory, that in certain cases is extremely undesirable, first of all, in the light of possibility of their subsequent restoration. The fact that the system function DeleteFile is based on the Dos command Del that according to specifics of this operating system immediately deletes a datafile from file system of the computer without its preservation, that significantly differs from similar operation of the Windows system that by default saves the deleted datafile in the special Recycle Bin directory.
For elimination of similar shortcoming the DeleteFile1 procedure has been offered, whose source code with examples of application are represented by the fragment below. The successful procedure call DeleteFile1[x] returns 0, deleting datafiles given by an argument x with saving them in the Recycle Bin directory of the Windows system. Meanwhile, the datafiles removed by means of procedure call DeleteFile1[x] are saved in Recycle Bin directory, however they are invisible to viewing by the system means, for example, by means of Ms Explorer, complicating cleaning of the given system directory. Whereas the procedure call DeleteFile1[x, t] with the 2nd optional argument t – an undefinite variable – thru it in addition returns the list of datafiles which for one reason or another were not removed. At that, in the system Recycle Bin directory a copy only of the last deleted datafile always turns out. This procedure is oriented on Windows XP and Windows 7, however it can be spread to other operational platforms."

ClearRecycler::usage =
"The datafiles x removed by means of the call DeleteFile1[x] remain in the directory Recycle Bin, but they are invisible to viewing by system means, for example, Ms Explorer, complicating cleaning of this system directory. For removal from the directory Recycle Bin of the datafiles saved by the DeleteFile1 procedure, the procedure, whose successful call ClearRecycler[] returns 0 is used, deleting the specified datafiles from the system directory Recycle Bin with preservation in it of other datafiles removed by means of Windows or its appendices. At last, the Dick Cleanup command in Windows XP in some cases completely doesn't clear the system Recycler directory from datafiles but it successfully does the call

ClearRecycler[\"ALL\"] with returning 0 and providing removal of all datafiles from the system Recycler directory."

ClearRecyclerBin::usage =
"The successful procedure call ClearRecyclerBin[] returns Null, i.e. nothing, and provides removal from the system Recycle Bin directory of directories and datafiles that are caused by the DeleteFile1 procedure. While the procedure call ClearRecyclerBin[x], where x – an arbitrary expression – also returns Null, i.e. nothing, and provides removal from the system Recycle Bin directory of all directories and datafiles whatever the cause of their appearance in the given directory. At that, the procedure call on the empty Recycler directory returns $Failed."

WhatValue::usage =
"The call WhatValue[x] returns value ascribed to a variable x; on an undefined variable x the list of format {\"Undefined\", x} is returned while on a system variable x the list of format {\"System\", x}, and on a local variable x the list of format {\"Local\", x} is returned."

CountOptions::usage =
"The call CountOptions[] returns the nested list whose elements are both lists, and separate options. The list contains the name of a group of options as the first element, whereas the second – number of options in this group. Whereas the call CountOptions[p] in addition through argument p – an undefined variable – returns the total of the preset options/suboptions of the package."

FreeSpaceVol::usage =
"The procedure call FreeSpaceVol[x] depending on type of an actual argument x which should define the logical name in string format of a device, returns simple or the nested list; elements of its sublists determine a device name, a volume of free memory on the volume of direct access, and the unit of its measurement respectively. In the case of absence or inactivity of the device x the procedure call returns the message \"Device is not ready\"."

VolDir::usage =
"The procedure call VolDir[x] returns the nested 2–element list, whose first element determines the volume occupied by a directory x in bytes whereas the second element determines the size of free space on a hard disk with the given directory."

DirsFiles::usage =
"The procedure call DirsFiles[x] returns the nested 2–element list, whose first element defines the list of directories contained in a directory x, including x, and the second element defines the list of all datafiles contained in the given directory."

TypeActObj::usage =
"The call TypeActObj[] returns the nested list whose sublists contain names of objects activated in the current session (in Input–mode) in string format and as the first element – their type, recognized by the package or defined by us, in particular, {\"Procedure\", \"Function\"}."

TypeWinMx::usage =
"In view of distinctions of the mx–files created on different platforms there is a natural expediency of creation of the means testing any mx–file regarding a platform in which it was created in virtue of the DumpSave function. The following TypeWinMx procedure is one of such means. The procedure call TypeWinMx[x] in string format returns the type of operating platform on which a mx–file x has been created; at that, correct result is returned for case of Windows platform, while on other platforms $Failed is returned. This is conditioned by lack of a possibility to carry out debugging on other platforms."

PureDefinition::usage =
"The call PureDefinition[x] returns the definition in string format or their list of a procedure/function x without options and attributes, ascribed to it whereas the call PureDefinition[x, t] with the second optional argument t – an undefined variable – returns through it the list of options and attributes, ascribed to x, including defaults for its formal arguments if they exist. In case of inadmissible argument x the call of procedure is returned unevaluated."

OpenFiles::usage =
"call OpenFiles[] returns the 2–element nested list, whose the first sublist with the first \"read\" element contains full paths to the datafiles opened on reading whereas the second sublist with the first \"write\" element contains full paths to the files opened on writing in the current session. In the absence of such datafiles the procedure call returns the empty list, i.e. {}. Whereas the call OpenFiles[x] with one actual argument x – a datafile classifier – returns result of the above format relative to the open datafile x irrespective of a format of coding of its qualifier. If x defines a closed or nonexistent datafile then the procedure call returns the empty list, i.e. {}."

StreamFiles::usage =
"call StreamFiles[] returns the nested list from two sublists, the first sublist with the first \"in\" element contains full paths/names of the files opened on the reading while the second sublist with the first \"out\" element contains full paths/names of the datafiles opened on the recording. Whereas in the absence of the open datafiles the procedure call StreamFiles[] returns \"AllFilesClosed\"."

PathToFileQ::usage =
"The call PathToFileQ[x] returns True if x defines a potentially allowable full path to a directory or a datafile, and False otherwise."

ProcQ::usage = "The call ProcQ[x] returns the True if x is a procedure and the False otherwise."

$TestArgsTypes::usage = "The global variable $TestArgsType defined by the procedures TestArgsTypes and TestArgsTypes1."

TestArgsTypes::usage =
"Like the Maple system, the Mathematica system doesn't give a possibility to test inadmissibility of all actual arguments in a block/function/module in a point of its call, interrupting its call already on the first inadmissible actual argument. Meanwhile, in view of importance of definition of all inadmissible actual arguments only for one pass, the TestArgsTypes procedure solving this important enough problem has been created. Call of the above procedure TestArgsTypes[x, x[...]] processes a procedure x call in way that returns result of a procedure call x[...] in case of absence of inadmissible actual arguments and equal number of the factual and formal arguments in a point of procedure call x; otherwise $Failed is returned. At that through the global variable $TestArgsTypes the nested list is returned, whose two−element sublists define the set of inadmissible actual arguments, namely: the first element of a sublist defines number of inadmissible actual argument while the second element – its value. At discrepancy of number of formal arguments to number of actual arguments through $TestArgsTypes the appropriate diagnostic message is returned, namely: \"Quantities of formal and factual arguments are different \". Meanwhile, for simplification of the testing algorithm realized by the above procedure it is supposed that formal arguments of a certain procedure x are typified by the pattern \"_\" or by construction \"Argument_/; Test\". Moreover, it is supposed that the unevaluated procedure call x is caused by discrepancy of types of the actual arguments to the formal arguments or by discrepancy of their quantities only. So, the question of testing of the actual arguments is considered at the level of the heading of a block/function/module only for a case when their number is fixed. If a procedure/function allows optional arguments, their typifying assumes correct usage of any expressions as the actual values, i.e. the type of the format \"x_\" is supposed. In this regard at necessity, their testing should be made in the body of a procedure/function as it is illustrated by useful enough examples. So, at difficult enough algorithms of check of the received actual arguments onto admissibility it is recommended to program them in the body of blocks/modules what is more appropriate as a whole."

TestArgsTypes1::usage =
"Meanwhile, as an expansion of the TestArgsTypes procedure the possibility of testing of the actual arguments onto admissibility on condition of existence in headings of formal arguments of types {\"x__\",\"x___\"} can be considered.The receptions,used in the TestArgsTypes1 procedure which is one useful modification of the above TestArgsTypes procedure is a rather perspective prerequisite for further expansion of functionality of these means.A result of call TestArgsTypes1[x,x[...]] is similar to the call TestArgsTypes[x,x[...]] only with difference that values of inadmissible actual arguments are given in string format.Meanwhile,it must be kept in mind that use by procedures TestArgsTypes and TestArgsTypes1 of the global variable $TestArgsTypes through which information on the inadmissible actual arguments received by a tested block/procedure at its calls is returned,should be defined in the user's package that contains definitions of these procedures,i.e.to be predetermined,otherwise diagnostic information isn't returned thru it."

TestArgsTypes2::usage =
"In some cases the TestArgsTypes2 procedure which is a modification of the procedures TestArgsTypes and TestArgsTypes1 is a rather useful means; the call TestArgsTypes2[P, y], where P – a block, function with the heading, or module, and y – a nonempty sequence of the actual arguments passed to the P, returns the list of the format {True, P[y]} if all arguments y are admissible; the call returns the nested list whose elements are 2−element sublists whose first element defines an actual argument whereas the second element defines its admissibility {True, False}; at last, in case of discrepancy of quantities of formal and actual arguments the next message is returned: \"Quantities of formal and factual arguments are different\"."

ProcQ1::usage =
"The ProcQ1 procedure generalizes the ProcQ procedure, first of all, in case of the objects of the same name. The previous fragment represents source code of the ProcQ1 procedure with examples of its most typical application. The call ProcQ1[x] returns True if the symbol x defines a procedural object of the type {Block, Module, DynamicModule} with unique definition along with an object consisting of their any combinations with different headings (the objects of the same name). Moreover, in case of a separate object or an object x of the same name True is returned only when all its components is procedural objects in the sense stated above, i.e. they have a type {Block, DynamicModule, Module}. Meanwhile, the procedure call ProcQ1[x, y] with the 2nd optional argument y – an undefinite variable – thru it returns simple or the nested list of the following format

{{a1, a2, a3,…, ap}, {b1, b2, b3,…, bp}}

where aj∈{True, False} whereas bj∈{\"Block\", \"DynamicModule\", \"Function\", \"Module\"}; at that, between elements of the above sublists exists one−to−one correspondence while pairs {aj, bj} (j=1..p) correspond to subobjects of the object x according to their order as a result of the call Definition[x]."

ProcBMQ::usage =
"The call ProcBMQ[x] returns True if a block or a module x is a real procedure in the traditional sense, and False otherwise. In addition, the given procedure is oriented only onto single objects of the mentioned type, i.e. whose definitions are unique. The call ProcBMQ[x] with one argument returns True, if a block or a module x – a real procedure in the above context, and False otherwise; the procedure call ProcBMQ[x, y] with the second optional argument y – an undefinite variable – returns thru it the list of local variables of the block x in string format which have no initial values or for which in a body of the block x the assignments of values weren't made. We will note, the ProcBMQ procedure is oriented only on one−defined objects whose definitions are unique while the message \"Object <x> has multiple definitions\" is returned on objects x of the same name."

RedundantLocals::usage =
"The call RedundantLocals[W] returns the list of redundant local variables of a block or a module W, i.e. local variables which have not obtained initial values or values in the body of W, or are not names of traditional functions (with headings) or of blocks/modules whose definitions are in the object W. Meanwhile, and the local variables used as argument at a call of one or another function in a body of the object W can be in such list. Let's note, the RedundantLocals procedure is focused only on the single objects W whose definitions are unique whereas otherwise, the message \"Object <W> has multiple definitions\" is returned. At the same time the RedundantLocals procedure quite successfully processes the objects containing in the body the definitions of typical functions, modules and blocks."

RedundantLocalsM::usage =
"The RedundantLocalsM procedure expands the RedundantLocals procedure onto the objects of the same name. The call RedundantLocalsM[W] on a single object W of the type block/module is similar to the call RedundantLocals[W] whereas on a traditional function W \"Function\" is returned; on an object W of the same name of type блок/модуль/функция the list of applications of RedundantLocals to its subobjects of type block/module and \"Function\" on traditional functions–subobjects is returned."

ProcFuncTypeQ::usage =
"Math–language identifies procedures and functions not by their names, but by headings, admitting not only the procedures with the same name with different headings, but also their combinations with functions. In this connection it is very expedient to define some testing procedure that determines belonging of an object x to a group {Block, CompiledFunction, Function, Module, PureFunction, ShortPureFunction}. As one of similar approaches it is possible to offer procedure, whose call ProcFuncTypeQ[x] returns the list of format {True, {t1, t2, ... ,tp}} if a simple object x or subobjects of an object x of the same name whose name x is coded in string format have the types tj from the set {CompiledFunction, PureFunction, ShortPureFunction, Block, Function, Module}, otherwise the list of format {False, x, \"Expression\"} or {False, x, \"System \"} is returned. In the case of an object x of the same name a sublist of types {t1, t2, ... ,tp} (j=1..p) of subobjects composing x is returned; whereas \"System\" and \"Expression\" determines a system function x and an expression x respectively."

SubsProcs::usage =
"The procedure call SubsProcs[x] returns generally the nested list of definitions in string format of all subobjects of the type {Block, Module} whose definitions are in the body of an object x of type {Block, Module}. At that, the first sublist defines subobjects of Module–type, the second sublist defines subobjects of Block–type. In the presence of only one sublist the simple list is returned while in the presence of the 1–element simple list its element is returned. At lack of subobjects of the above type the call SubsProcs[x] returns the empty list, i.e. {} while on an object x, different from a block or module, the call SubsProcs[x] is returned unevaluated."

SubStrSymbolParity::usage =
"The call of procedure SubStrSymbolParity[x, y, z, d] with four arguments returns the list of substrings of a string x, limited by one–symbolical strings {y, z} (y ≠ z); in addition, search of such substrings is made from left to right for d = 0 whereas for d=1 the search in the string x is made from right to left. Furthermore, the call of procedure SubStrSymbolParity[x, y, z, d, t] with fifth optional argument – a positive integer t > 0 – provides search in a substring of x which is limited by a position t and the end of the string for d = 0, and by the beginning of the string and t for d = 1. At receiving of inadmissible actual arguments the call of procedure is returned unevaluated, whereas in case of impossibility of extraction of the required substrings the call of procedure returns $Failed."

SubStrSymbolParity1::usage =
"Meanwhile, in many cases it is quite possible to use the simpler and reactive version of the above procedure SubStrSymbolParity, namely the procedure, whose call SubStrSymbolParity1[x,y,z] with 3 actual arguments returns the list of substrings of a string x limited by the one–symbols strings {y, z} (y ≠ z); in addition, search of such substrings is done from left to right end of the string x. In case of lack of such substrings the call returns the empty list, i.e. { }."

StrSymbParity::usage =
"The procedure StrSymbParity is a very useful modification of the SubStrSymbolParity1 procedure; its call StrSymbParity[S, S1, x, y] returns the list, whose elements – the substrings of a string S which have format S1W and on condition of parity of the minimum number of occurrences of symbols x, y (x ≠ y) into substring W. In a case of lack of such substrings or identity of symbols x and y the call returns the empty list, i.e. { }."

RedSymbStr::usage =
"The call RedSymbStr[x, y, z] returns the result of replacement of all substrings consisting of a symbol y, in a string x onto a symbol or a string z. In case of absence of occurrences of y in string x, the call of procedure returns the string x without change."

SubCfEntries::usage =
"The procedure call SubCfEntries[x, n, y] generally returns the nested list of ListList–type, whose elements – 2–element sublists whose first elements define the multiplicity while the second elements define the substrings of n length of a string x which have the specified multiplicity > 1 of their occurrences into x string. At that, with the default argument y = False, overlapping substrings are not treated as separate whereas the setting y = True, the SubCfEntries counts substrings that overlap as separate."

ReduceList::usage =
"The call ReduceList[L,x,z,t] returns the result of reducing of elements of a list L that are determined by a separate element x or their

list to a multiplicity determined by a separate element z or their list. If elements of x don't belong to the list L, the procedure call returns the initial list L. At that, if Length[z] < Length[x] a list z is padded on the right by 1 to the list length x. In addition, the fourth argument t defines direction of reducing in the list L (on the left at t = 1 and on the right at t = 2)."

ListPosition::usage =
"The simple enough procedure ListPosition expands the standard function Position onto the list as its second actual argument. The call ListPosition[x, y], where x – a simple list and y – a list of arbitrary expressions, returns the nested list whose elements define lists of positions of elements of the list y in the list x."

PosSubList::usage =
"The procedure call PosSubList[x, y], where x – a simple list and y – a list of arbitrary expressions, returns the nested list whose elements define positions of a tuple of elements given by a list y."

ListableQ::usage = "The call of function ListableQ[x] returns True if an object x has Listable–attribute, and False otherwise."

ShortPureFuncQ::usage =
"The call ShortPureFuncQ[x] returns True if x determines a pure function in short format, and False otherwise."

ActBFMuserQ::usage =
"A quite natural interest represents the question of existence of the user procedures and functions activated in the current session. The solution of the given question can be received by means of the procedure whose procedure call ActBFMuserQ[] returns True if such objects in the current session exist, and False otherwise; meanwhile, the call ActBFMuserQ[x] through optional argument x – an undefinite variable – returns the 2–element nested list, whose the first element contains name of the user object in string format whereas the second element defines the list of its types in string format respectively."

GroupIdentMult::usage =
"The procedure GroupIdentMult is intended for the grouping of elements of a list according to their multiplicity. The call GroupIdentMult[x] returns the nested list of the following format, namely:
{{{n1}, {x1, x2,..., xa}}, {{n2}, {y1, y2,..., yb}},..., {{nk}, {z1, z2,..., zc}}}
where {xi, yj, zp} – elements of a list x and nt – multiplicities corresponding to them {j=1..a; j=1..b; p=1..c; t=1..k}. The call GroupIdentMult[{}] returns the empty list, i.e. {}."

ClearValues::usage =
"The call ClearValues[x] returns the empty list, by removing from the current session all variables having values from the list x; whereas the call ClearValues[x, y] with the second optional argument y – an arbitrary expression – returns the empty list too, however such variables are only cleared of values and attributes without removal from the current session."

ClearContextVars::usage =
"Unlike the ClearValues function, the function call ClearContextVars[x] clears all values, definitions, attributes, messages, and defaults associated with symbols having a context x while the function call ClearContextVars[x, y] removes the above symbols completely, so that their names are no longer recognized in the current session where y – an arbitrary expression. At that, in both cases the call returns the list of the cleared\removed symbols. If the used context x not belong to the list of main contexts of the current session the function call is returned unevaluated."

VarsValues::usage = "The call VarsValues[x] returns the list of variables in string–format that have values from a list x."

ListableC::usage = "The call ListableC[h] returns the list of attributes of h–object, providing the setting of Listable–attribute for h–object (the user's procedure/function or standard function)."

ListAssignP::usage =
"The call ListAssignP[x, n, y] returns the updated value of the list x which is based on result of assignment of a single value, or the list of values to n–th elements of the list x as which can act one position of the list or their list. Moreover, in case lists n and y have different lengths, their minimum value is chosen."

Levels::usage =
"The call Levels[x, h] returns the list of all subexpressions for an expression x on its all possible levels, whereas through second argument h – an undefined variable – the maximal number of levels of the expression x is returned."

ListNumericQ::usage =
"The call of function ListNumericQ[x] returns True if x is a numeric list, including all its sublists of any nesting level, and False otherwise. In addition, instead of the list x the list N[x] is considered."

IntegerListQ::usage = "The call of function IntegerListQ[x] returns True if x is a integer list, including all its sublists of any nesting level, and False otherwise."

ElemLevelsL::usage =
"The call ElemLevelsL[x] returns the nested list whose elements are 2–element lists whose first element defines the nesting level whereas the second one – the list of elements of this level with type, different from List."

ElemLevelsN::usage =
"The call ElemLevelsN[x] returns the nested list whose elements are 2−element lists whose first element defines the nesting level whereas the second one − amount of elements of this level with type, different from List."

ElemOnLevels::usage =
"Generally, the call ElemOnLevels[x] returns the nested list whose elements are the sublists whose the first elements are levels of a nested list x while the others – elements of this level. In case of lack of elements at w–level the sublist takes on form {w}; the call ElemOnLevels[x] on a simple list x returns {0, x}, i.e. an arbitrary simple list has nesting level 0."

ElemsOnLevelList::usage =
"The procedure call ElemsOnLevelList[L] returns the nested list whose elements – the nested 2–element lists, whose first elements are nesting levels of a list L whereas the second elements – lists of elements at these nesting levels. In case of the empty list L, i.e. {}, {} is returned."

ListSymbolQ::usage = "The call of function ListSymbolQ[x] returns True if x is a symbolic list,including all its sublists of any nesting level,and False otherwise."

SymbolGreater::usage =
"The call of function SymbolLess[x, y] returns True if a symbolic expression x is greater than a symbolic expression y; in addition, the comparison is done on the basis of their character codes."

SymbolLess::usage =
"The call of function SymbolLess[x, y] returns True if a symbolic expression x is less than a symbolic expression y; in addition, the comparison is done on the basis of their character codes."

ListExprHeadQ::usage = "The call of function ListExprHeadQ[x, h] returns True if x is a list containing only elements t for which the following relation Head[t]==h takes place, and False otherwise."

RemoveNames::usage =
"The call of procedure RemoveNames[] provides the removal from the current session of names, whose types are distinct from procedures and functions whose definitions have been evaluated in the current session; names are removed so, that they more will not be recognizable by the package. The call RemoveNames[] along with removal of the above−mentioned names from the current session returns the nested 2−element list, whose first element defines the list of names of procedures, whereas the second − the list of names of functions whose definitions have been evaluated in the current session of the package."

RemovePF::usage =
"Mathematica supposes presence of the same objects with various headings which identify objects, instead of their names. Standard function Definition, and also our procedures DefFunc, Deffunc3, PureDefinition etc. allow to receive by name of an object the definitions of all active objects in the current session with identical names, but with various headings. In view of told, there is quite a specific problem of removal from the current session not of all objects with a concrete name, but only the objects with concrete headings. The given problem is solved by means of procedure RemovePF, whose call RemovePF[x] returns Null, i.e. nothing, providing removal from the current session of objects with the headings determined by the factual argument x (a heading in string−format or their list). The call RemovePF[x] is returned unevaluated if x not defines headings."

RemovePackage::usage =
"The call of procedure RemovePackage[x] returns Null, i.e. nothing, providing removal from the current session of the package, defined by a context x, including all its exported symbols and accordingly updating the lists $Packages, $ContextPath and Contexts[]."

RemoveContext::usage =
"Sometimes, it is expedient to replace a context of means of the user package uploaded into the current session. This problem is solved by the procedure RemoveContext, whose successful call RemoveContext[w, x, y, z, ...] returns nothing, replacing in the current session w context ascribed to means {x, y, z, ...} of the user package by the context \"Global'\". In the absence of means {x, y, z, ...} with w context the procedure call returns $Failed. In particular, the procedure can be useful enough in case of replacing in the current session of the definitions of the uploaded user package, cancelling their contexts."

DeletePackage::usage =
"The procedure DeletePackage is a version of the RemovePackage procedure. The procedure call DeletePackage[x] returns Null, i.e. nothing, providing removal from the current session of the package, defined by a context x, including all its exported symbols and accordingly updating the lists $Packages, $ContextPath and Contexts[]. The difference of the DeletePackage from RemovePackage is that the symbols defined by the package, removes completely, so that their names are no longer recognized in the current session."

DelOfPackage::usage =
"Natural addition to the DeletePackage and RemovePackage procedures is the DelOfPackage procedure providing removal from the current session of the given means of a loaded package. Its call DelOfPackage[x, y] returns the list y of means names of a package given by its context x which have been removed from the current session. While the call DelOfPackage[x, y, z] with the third optional argument z – a mx–file – returns 2–element list whose the first element defines mx–file z, while the second element defines the list y of means of a package x that have been removed from the current session

and have been saved in mx–file z. At that, only means of y that are contained in a package x will be removed."

PackageQ::usage = "The call PackageQ[x] returns True if x is a package containing global symbols, and False otherwise."

Packages::usage =
"The call Packages[] returns the list of contexts of active actual packages, i.e. packages that define global symbols (exports)."

SaveCurrentSession::usage =
"The procedure call SaveCurrentSession[] saves a state of the Mathematica current session in the m–file \"SaveCS.m\" with returning of the name of a target datafile. While the call SaveCurrentSession[x] saves a state of the Mathematica current session in a m–file x with returning of the name of the target datafile x; at that, if a datafile x has not extension \"m\" then this extension is added to the x–string."

RestoreCS::usage =
"The procedure call RectoreCS[] restores the Mathematica current session that has been previously stored by means of the SaveCurrentSession procedure in datafile \"SaveCS.m\" with returning the Null, i.e. nothing. While the call RectoreCS[x] restores the Mathematica current session that has been previously stored by means of the procedure SaveCurrentSession in a m–datafile x with returning the Null, i.e. nothing. In absence of the above datafile the procedure call returns $Failed."

DumpSaveP::usage = "The call DumpSaveP[f, x] is equivalent to the call
DumpSave[f, x] if x defines a package containing global symbols; otherwise, the call returns $Failed."

DumpSave1::usage =
"The successful call DumpSave1[x, y] returns the nested list whose first element defines the full path to a receiving datafile x of mx−format (if necessary to the datafile is assigned the mx−extension) whereas the second element defines the list of objects and/or contexts from the list determined by argument y whose definitions were unloaded into the datafile x of mx−format. In case of absence of objects (symbols with definitions, and/or the contexts which are present at the list defined by the system variable $ContextPath) which are defined by argument y, the call DumpSave1 returns the $Failed."

DumpSave2::usage =
"At evaluation of definition of a symbol x in the current session it will associate with the context of \"Global'\" which remains at its unloading into a mx–file by the DumpSave function. Whereas in some cases there is a need of saving of symbols in the mx–files with other contexts. This problem is solved by the procedure, whose call DumpSave2[f, x, y] returns nothing, unloading into mx–file f the definition of a symbol or the list of symbols x that have the context \"Global'\", with a context y. Thus, in the current session the symbols x receive context y."

SaveInMx::usage =
"At evaluation of definition of a symbol x in the current session it will associate with the context of \"Global'\" which remains at its unloading into a mx–file by the DumpSave function. Whereas in some cases there is a need of saving of symbols in the mx–files with other contexts. This problem is solved by the procedure DumpSave2 whereas the procedure SaveInMx is more user–friendly whose call SaveInMx[x, y, z] returns nothing, unloading into mx–file x with context z the definition of a symbol or the list of symbols y whose definitions have been defined in the current session."

CurrentPackageQ::usage =
"The call CurrentPackage[w] returns True, if w is the context for a package active in the current session, and False otherwise."

PackageMxCont::usage =
"The procedure call PackageMxCont[x, y] thru the 2nd optional argument – an undefinite variable y – returns the nested list whose first element defines the context of x, while the second element determines the list of global symbols of the package that are contained in the mx–file x [33,48]. On mx–files without context or local/global symbols the procedure call PackageMxCont[x] returns $Failed or the empty list, i.e. {}, accordingly. The procedure PackageMxCont is oriented on platforms Windows XP Professional and Windows 7 Professional."

ClearOut::usage = "The call ClearOut[x] returns nothing, simultaneously
deleting Out−paragraphs with numbers determined by a positive integer or their list x."

ReplaceOut::usage =
"Successful call ReplaceOut[x, y] returns nothing, at the same time carrying out the replacement of contents of existing Out−paragraphs which are given by a positive integer or their list x, onto the new expressions defined by y−argument. The call presumes parity of number of the replaced Out−paragraphs to number of the expressions replacing their current values; otherwise, the call of procedure ReplaceOut[x, y] is returned unevaluated. Procedure ReplaceOut generalizes the above procedure ClearOut."

GroupNames::usage =
"Procedure GroupNames makes a grouping of the expressions given by argument L, according to their types determined by the function Head2; in addition, a single expression or their list is coded as argument L. The call GroupNames[L] returns the list or the nested list, whose elements are lists, whose first element is a type of an object according to the function Head2, whereas the others are expressions of the given type. A x name that is returned by the procedure call GroupNames[x] belongs to a group defined by the procedure call Head2[x]."

DelSuffPref::usage =
"The call DelSuffPref[x, y, n] provides return of the result of truncation of a string x by a substring y at the left (n=1), on the right (n=2) or from both ends (n=3); otherwise, the string x is returned without change."

SubsBstr::usage = "The call SubsBstr[S, x, y] returns the list of all non−overlapping substrings in a string S that are limited by symbols x and y; otherwise, the empty list, i.e. {} is returned."

SubsStrLim::usage =
"The call SubsStrLim[x, y, z] returns the list of substrings of a string x which are limited by symbols {y, z} provided that such symbols do not belong to the given substrings, excepting their ends."

SubsStrLim1::usage =
"The call SubsStrLim1[x, y, z] returns the list of substrings of a string x which are limited by symbols {y, z} provided that the given symbols or do not belong to the given substrings, excepting their ends, or together with their ends have the same number of occurrences of y and z."

SubsList::usage =
"The call SubsList[x, y, z] returns the list of all non−overlapping sublists in a list x that are limited by elements y and z; the lists can act as elements {y, z}. If any element {y, z} no belongs to x the empty list, i.e. {} is returned."

Definition1::usage =
"A lot of functions of Math−language suppose only the objects of types {Symbol, String, HoldPattern[Symbol]} as actual arguments, what in some cases is rather inconvenient at programming of problems of various purpose. In particular, function Definition evidently gives such example. With a view of expansion of the standard function onto the types which are distinct from above−mentioned ones, the procedure Definition1 is presented, whose the call Definition1[x] in string−format returns the definition of an object x, \"Null\" if x is not determined, otherwise $Failed is returned."

Definition2::usage =
"The call Definition2[x] returns the list whose the first element is the identifier \"System\" if a symbol x defines a system function, whereas the second element is the list of the attributes, attributed to the symbol x. In other cases of a defined symbol x the first element of the returned list is the sequence of all definitions in string−format for the symbol x whereas the second element of the returned list remains the same as it was mentioned earlier. On an undefined x the call Definition2[x] is returned unevaluated."

Definition3::usage =
"The call Definition3[x, y] returns the optimum definition of a procedure or a function x, whereas through the second argument y – an undefined variable − the type of x from the angle of {\"Procedure\", \"Function\", \"Procedure&Function\"} is returned if x is procedure or function, on system functions the procedure call returns the list {\"Function\", {Attributes}} whereas thru the second argument y the first argument is returned; at inadmissibility of the first argument x the call is returned unevaluated, i.e. Definition[x]."

Definition4::usage =
"The call Definition4[x] in a convenient format returns the definition of an object x whose name is coded in string format, namely: (1) on a system function x its attributes are returned, (2) on the user block, function or module the call returns the definition of object x in string format with the attributes, options and/or values by default for formal arguments ascribed to it (if such are available), (3) the call returns the definition of the object x in string format for assignments by operators {\":=\", \"=\"}, and (4) in other cases the call returns $Failed."

Def::usage =
"The calls of procedures Def[x] and Def1[x] return the pure definitions of x−object. Pure definition is a definition of object x without attributes (if such attributes exist), that are ascribed to it. Attributes remain without change. Functionally identical procedures have various implementations, intended for an illustration of different mechanisms of implementation."

Def1::usage =
"The calls of procedures Def[x] and Def1[x] return the optimized definitions of x−object. Optimized definition is a definition of object x without attributes (if such attributes exist), that are ascribed to it. Attributes remain without change. In event of an object x of the same name the procedure call Def1[x] returns the list of the optimized definitions of the object x in string format without the attributes ascribed to it. If x defines an unique name, the call returns the optimized definition of the object x in string format without the attributes ascribed to it. The name of an object x is given in string format; in addition, on unacceptable values of argument x $Failed is returned.Functionally identical procedures have various implementations, intended for an illustration of different mechanisms of implementation."

OptRes::usage =
"The call OptRes[x, y] returns a result y optimized for the subsequent processing that is returned by a function/procedure x."

CsProcsFuncs::usage =
"The call CsProcsFuncs[] returns the list of procedures and functions whose definition have been evaluated in the current session."

GlobalToLocal::usage =
"The call GlobalToLocal[x] provides the converting of definition of a procedure x into definition of procedure $$$x in which all global variables of initial procedure x are included in tuple of its local variables; the procedure call returns a name of the procedure activated in the current session which has no global variables. Resultant procedure is equivalent to the initial procedure provided that the generated values of global variables aren't as values of global variables for the current session. Whereas the call GlobalToLocal[x, y] with the second optional argument y – an indefinite variable – additionally through it returns the nested list, whose the first element defines the sublist of local variables, and the second element defines the sublist of global variables of a procedure x."

GlobalToLocalM::usage =
"The call GlobalToLocalM[x] returns Null, i.e. nothing, converting a block or a module x into object x of the same type and with the same attributes and options in which global variables (if such variables were) of the source object receive the status of the local variables. In addition, in case of the objects of the same name x the procedure provides correct converting."

LocalsGlobals::usage =
"The call LocalsGlobals[x] returns the 2−element nested list whose first element defines the list of local variables with initial values (if such values exist) in string format whereas the second element defines the list of global variables in string format of a procedure (block or module) x."

LocalsGlobals1::usage =
"The call LocalsGlobals1[x] returns the nested 3−element list whose first element defines the list of local variables in string format, the second one – local variables in string format with initial values (if such values exist), whereas the third element defines the list of global variables in string format of a block or module x. On argument x of type different from block/module the procedure call is returned unevaluated."

LocalsGlobalsM::usage =
"The procedure LocalsGlobalsM expands the procedure LocalsGlobals1 onto a case of the blocks/modules of the same name; the call LocalsGlobalsM[x] returns the list of the nested 3−element lists of the format similar to the format of results returned by the call LocalsGlobals1[x] whose elements are biunique with subobjects x, according to their order at the call PureDefinition[x]. On argument x of type different from block/module the procedure call is returned unevaluated."

CsProcsFuncs1::usage =
"The call CsProcsFuncs1[] returns the nested list whose elements are lists whose the first element defines a procedure or function whose definition have been evaluated in the current session, whereas the second element defines multiplicity of it, i.e. number of objects of the same name ."

Attributes1::usage =
"A lot of functions of Math−language suppose only the objects of types {Symbol, String, HoldPattern[Symbol]} as actual arguments, what in some cases is rather inconvenient at programming of problems of various purpose. In particular, function Attributes evidently gives such example. With a view of expansion of the standard function onto the types which are distinct from above−mentioned ones, the procedure Attributes1 is presented, whose the call Attributes1[x, y, z, ...] on objects different from admissible ones by the Attributes function, returns the empty list, i.e. { }, without print of error messages, what in a number of cases is more preferably from the point of view of processing of erroneous and special situations. While on admissible objects x, y, z, ... an the call Attributes1[x, y, z, ...] returns the list of the attributes, ascribed to objects x, y, z, ..."

AttributesQ::usage =
"The call AttributesQ[x] returns True, if x – the list of admissible attributes of the current version of Mathematica, and False otherwise. Moreover, the call AttributesQ[x, y] with the 2nd optional argument y – an undefined variable – returns through y the list of elements of the list x which aren't attributes."

ParVar::usage =
"In a whole series of cases it is necessary to execute assignments of expressions to variables whose number is beforehand not known and which is determined as a result of some calculations, for example, of cyclic character. The given problem is solved by simple enough procedure ParVar. The call ParVar[x, y] provides the assignment of elements of a list y to the list of variables generated on the basis of a symbol x with return of the list in string−format of these variables."

Save1::usage =
"The call of procedure Save1[x, y] saves in a file defined by the first actual argument x, definitions of the objects defined by the second actual argument y which can be by the name of an active object in the current session or by its heading in string−format, or by their combination in the form of the list. The successful call of the procedure returns Null; otherwise $Failed or unevaluated call is returned."

Save2::usage =
"The function call Save2[x, y] append to a datafile x the definitions of the means given by a name or their list y in the format convenient for the subsequent processing by a number of means, in particular, by the CallSave procedure. The essence of such formatting consists in completion of a datafile created by the Save1 function by the separator \"\\\\r \\\\n \\\\r \\\\n\". The successful function call returns Null, i.e. nothing. This function is a rather useful extension of the standard Save function [8]."

SysFuncQ::usage =
"The call SysFuncQ[F] returns True if a F−object is a standard function of the Mathematica system, and False otherwise."

SysUserSoft::usage =
"Generally, the call SysUserSoft[x] returns the enclosed 2−element list, whose first element contains 2−element sublists, whose the first element − a name in string format of system function, and the second element − its frequency, whereas the second element of the list also contains 2−element sublists, whose the first element − a name in string format of the user means (Block, Function, Module), and the second element − its frequency. In absence for x of means of the above types the call SysUserSoft[x] returns the empty list, i.e. { }. If the type of the actual argument x is different from (Block, Function, Module), the call SysUserSoft[x] is returned unevaluated."

ExprPatternQ::usage = "The call ExprPatternQ[x] returns True if an expression x contains occurences at least one of the patterns {\"_\", \"__\", \"___\"}, and False otherwise."

ProcFuncBlQ::usage =
"The call ProcFuncBlQ[x, y] returns True if x is a function, a procedure or a block, and False otherwise. Moreover, at return the True through argument y the type of x−object {\"DynamicModule\", \"Module\", \"Function\", \"PureFunction\", \"Block\"} is returned; otherwise, the second argument y remains indefinite. It should be noted that this procedure is correctly carried out only on objects of the specified type provided that they have the unique definitions, returning False otherwise."

ProcFuncBlQ1::usage =
"The ProcFuncBlQ1 procedure can appear as an useful enough extension for the testing of objects, its call ProcFuncBlQ1[x, y] returns True if x is a procedure, function or block, otherwise False is returned. Moreover, at return of True, through argument y − an undefinite variable − a x−object type is returned {\"DynamicModule\", \"Module\", \"Function\", \"PureFunction\", \"Block\"}, otherwise the second argument remains undefinite. It should be noted that the above procedure is correctly executed both on single objects, and on objects of the same name, otherwise False is returned. The following fragment represents source code of the ProcFuncBlQ1 procedure with the most typical examples of its usage."

BootDrive::usage =
"The call BootDrive[] returns the 2−element list whose first element defines the name of a device of initial loading of operational system whereas the second element defines the type of operational system. In the absence (damage of system or system is more senior than Windows XP) of file \"boot.ini\" of initialization of loading of an operational system, the call of procedure returns the $Failed. Procedure BootDrive successfully functions in Mathematica of releases 7 − 8 on platform Windows {2000|2003|NT|XP} with several operational environments too."

BootDrive1::usage =
"The BootDrive procedure is correct for Windows 2000|2003| NT|XP while since Windows 7, it is necessary to use the BootDrive1 function. The call BootDrive1[] returns the 3−element list, whose first element − homedrive, the second − the system catalog, the third − type of an operating system. Furthermore, this function can be used for an arbitrary operation system, supported by the Mathematica. Meantime the type of an operating system in some cases by the call GetEnvironment[] that is used by the BootDrive1 is returned incorrectly; in particular, the Windows 7 is recognized as the Windows_NT."

ExtrName::usage =
"The call ExtrName[x, n, p] returns the substring of a string x which is limited by a position n on the one hand and includes only symbols which are allowable in composition of names of objects. Whereas the third argument p defines the direction of extraction of the substring (p=1 − to the right and p=−1 − to the left from position p; in addition, the symbol in the position p is ignored)."

ExtrVarsOfStr::usage =
"The call ExtrVarsOfStr[S, t] returns at t = 2 the sorted list and at t = 1 the unsorted list of variables in string format that were successfully extracted from a string S; in the absence of such variables the empty list is returned, i.e. {}. Whereas the call ExtrVarsOfStr[S, t, y] with the 3nd optional argument y − an arbitrary expression − returns the list of variables contained in S, without reducing of their frequencies to 1. In addition, under a variable is understood an identifier to which a value can be appropriated. For correct use of the procedure ExtrVarsOfStr is supposed that an expression Expr defined by a string S, is in InputForm−format, i.e. S = ToString[InputForm[Expr]]."

BlockFuncModVars::usage =
"The call BlockFuncModVars[x] returns the enclosed 6−element list, whose the first element − the list of the system functions used by a block/module x, whose the first element \"System\", whereas others − names of system functions in string format; the second element − the list of the user means used by the block/module x, whose the first element \"Users\" whereas the others define names of means in string format; the third element − the list of formal arguments of the block/module; the fourth element − the list of local variables of the block/module x in string format; the fifth element − the list of active global variables of the block/module x in string format; at last, the sixth element defines the list of passive global variables of the block/module x in string format. Whereas on a function x the call BlockFuncModVars[x] returns the enclosed 4−element list, whose the first element − the list of the system functions used by a function x, whose the first element \"System\", whereas others − names of system functions in string format; the second element − the list of the user means used by the function x, whose the first element \"Users\" whereas the others define names of means in string format; the third element − the list of formal arguments of the function x, and the fourth element − the list of global variables of the function x in string format."

ExtrNames::usage =
"The call ExtrNames[x] returns the nested 3−element list whose first element defines the list of all local variables of a procedure x in string−format, the second − the list of local variables of the procedure x in string−format for which assignments of expressions by operators {\":=\", \"=\"} are made in body of the procedure x, while the third element defines the list of global variables for which assignments of expressions by operators by operators {\":=\", \"=\"} in body of the procedure x are made."

SysFuncQ1::usage = "This function is a modification of the procedure SysFuncQ. The call SysFuncQ1[F] returns True if a F−object is a standard function of the package Mathematica, and False otherwise."

UnevaluatedQ::usage =
"The call UnevaluatedQ[F, x] returns True if a call F[x] is returned unevaluated, otherwise False is returned; in addition, in case of erroneous call F[x] the value \"ErrorInNumArgs\" is returned."

ComplexQ::usage = "The call ComplexQ[x] returns the True if x is a complex number, and the False otherwise."

ListStrQ::usage = "The call ListStrQ[x] returns True if x is a list whose all elements are strings, and False otherwise."

MaxParts::usage = "The call MaxParts[x] returns the quantity of all parts of an expression x."

StringDependQ::usage =
"The call StringDependQ[x, y] returns True if a string x contains occurrences of a substring or all substrings, given by a list y, and False otherwise. Whereas the call StringDependQ[x, y, z] in the presence of the third optional argument – an undefined variable – through z additionally is returned the list of substrings not having occurrences in the string x."

StringDependQ1::usage =
"The procedure call StringDependQ1[x, y] returns True if a string x contains occurrences of a chain of the substrings determined by a list of strings y and in the order determined by their order in the list y, otherwise False is returned."

DO::usage = "The call DO[x, y, k] returns the list of values of cyclic evaluation of an expression x on variable k which possesses the values from a list Op[y]."

NestListQ::usage = "The call of function NestListQ[x] returns the True, if x − a nested list, and False otherwise. In addition, by the nested list is understood list whose all elements are lists."

NestListQ1::usage = "The call of function NestListQ1[x] returns the True, if x − a nested list, and False otherwise. In addition, by the nested list is understood list for which at least one element is list."

GotoLabel::usage =
"The call GotoLabel[P] allows to analyse the user block or module P, whose definition is active in the current session, for the purpose of formal correctness of use by it of Goto−functions and the labels Label corresponding to them. The call of the given procedure returns the nested 3−element list, whose first element represents the list of all Goto−functions used by the block/module P, the second − the list of all labels (without taking their multiplicity), and the third element − the list whose sublists define functions Goto with labels corresponding to them (in addition, the first elements of such sublists are the calls of Goto, whereas multiplicities of Goto−functions and Labels are saved)."

SubProcs::usage =
"The procedure call SubProcs[P] returns the 2−element nested list of ListList−type whose the first element is the list of headings of subprocedures composing a main procedure P of Module−type whereas the second element is the list of the generated names of all these subprocedures including a main procedure P which are activated in the current session P. Between the both lists one−to−one correspondence exists."

SubProcs1::usage =
"The procedure call SubProcs1[P] depending on existence of procedures P of the same name but with different headings returns the 2−element nested list of ListList−type whose the first element is the heading of procedure with name P whereas the second element is number of subprocedures composing a procedure P with the given heading. If a procedure P has unique heading then simple 2−element list is returned. In other cases $Failed is returned."

SubProcs2::usage =
"The call SubProcs2[y] depending on an unique procedure y or of the same name with various headings, returns simple or nested list. For the returned list or sublists the first element is the procedure y heading, while the others – definitions in string format of subprocedures of the Module type that enter into the y definition. In absence for y of subprocedures of the specified type or in the case of type of argument y, different from Module, the procedure call SubProcs2[y] returns $Failed. In case of the second optional argument z – an arbitrary expression – the call SubProcs3[y, z] returns the similar result with simultaneous activation of these subprocedures in the current session."

SubProcs3::usage =
"The call SubProcs3[y] differs from the call SubProcs2[y] by the following two moments, namely: (1) as argument y can act both the user block, function or module, and (2) the returned list contains the heading in string−format of an object y as the first element whereas other element are definitions of blocks, functions and modules contained

in definition of y. In case of the object y of the same name, the nested list is returned, its sublists have the format stated above. Thus, the call SubProcs3[y, z] with the second optional argument z – an arbitrary expression, returning the list stated above, at the same time does all objects of the above type, which are entered into the object y as active in the current session. If a function with heading acts as an object y, its heading is returned only; the similar result takes place and in case the object y doesn't contain subobjects of the above type whereas on object y of type different from the user block, function or module, the call SubProcs3 returns the value $Failed."

SubsProcQ::usage =
"The call SubsProcQ[x, y] returns True if y is a global active subobject of an object x of the above type, and False otherwise. But as the Math−objects of the given type differ not by names as that is accepted in the majority of programming systems, but by headings then through the 3rd optional argument − an undefinite variable − the procedure call returns the nested list whose sublists as first element contain headings with a name x while the second element contain the headings of subobjects corresponding to them with a name y. On the first 2 arguments {x,y} of the types, different from specified in a procedure heading, the procedure call SubsProcQ[x, y] returns False. In principle, on the basis of the above five means {SubProcs ÷ SubProcs3, SubsProcQ} it is possible to program a number of useful enough means of work with expressions of the types {Block, Module}."

NotSubsProcs::usage =
"The Mathematica system allows use of the nested procedures containing other procedures of various nesting in its body. Such organization of the nested procedures is supposed as correct when the name of each nested procedure has to be in the list of local variables of a procedure, which directly its contains. In this case by a call of the main procedure the access to all its subprocedures is inaccessible, for example, by means of the Definition function. For testing of correctness of the above type of any procedure of Module−type a procedure has been programmed , whose call NotSubsProcs[x, y] returns the list of names in string format of subprocedures of a procedure x that don't satisfy to the specified agreement and whose definitions are available in the current session after a procedure call x. Whereas through the second optional argument y – an undefinite variable – the list of names in string format of all subprocedures of the procedure x without their nesting is returned. As an indication of correctness of a nested procedure x is the return of the empty list, i.e. {} by means of call NotSubsProcs[x]. The NotSubsProcs procedure is intended for testing of the nested procedures of Module−type with only one heading, however it can be rather simply extended and onto procedures of the same name."

SubsStr::usage =
"The call of procedure SubsStr[x, y, h, t] returns result of replacement in a string x of occurrences of substrings, formed by means of concatenation (on the right, if t=1 or at the left, if t=0) of a substring y with strings from a list h, onto the strings from the list h accordingly. In case of impossibility of execution of replacement the initial string x is returned."

StrOfSymb::usage =
"The function call StrOfSymb[x, y] returns the string composed of symbols of a string x which satisfy a pure function y."

ProcContent::usage =
"In some cases the certain interest represents a question of definition of all means being used by the user block, function or module. Procedure ProcContent provides such analysis of an object x activated in the current session with correct heading for the purpose of use in its definition of the user internal means, and external means, supplied with usage information or without it. The call ProcContent[x] returns the nested 3−element list whose the first element defines the name of a block/function/module x, the second – the list of names of all external blocks, functions or the modules used by object x, and the third – the list of names of internal blocks, functions or the modules defined in the body of x."

CallsInMean::usage =
"The CallsInMean procedure provides the analysis of means of a package uploaded into the current session regarding existence in their definitions of calls of the means supplied with usages. The procedure call CallsInMean[x] returns the 2−element list, whose the first element is x – the symbol defining the means name of a package while the second element – the list of names of the means that enter in definition of the symbol x. On a symbol x, whose context is \"System`\" or \"Global`\", the procedure call returns $Failed."

CallsInMeansPackage::usage =
"The CallsInMeansPackage procedure that is based on the previous procedure provides the analysis of all means of the package given by a context x and uploaded into the current session, regarding existence in their definitions of calls of the means supplied with references. The procedure call CallsInMeansPackage[x] returns the 3−element list, whose the first element defines the list of names of the means which aren't containing such calls, the second element – the list of names of means containing such calls, and the third element – the list returned by the procedure call CallsInMean for the means having the maximum number of such calls occurrences. On a x context \"System`\" or \"Global`\", the procedure call returns $Failed."

HeadName::usage =
"The call HeadName[x] returns the name of a heading x in the string format provided that the heading is distinguished by procedure HeadingQ or HeadingQ1 as a syntactically correct heading, i.e. the call HeadingQ[x] or HeadingQ1[x] returns True; otherwise, the function call HeadName[x] will be returned unevaluated."

HeadName1::usage = "The procedure call HeadName1[x] returns the heading name of a correct x heading, and False otherwise."

BitSet1::usage =

"The call BitSet1[n, p] returns the result of setting into positions of binary representation of an integer n which are defined by the first elements of sublists of a nested list p values {0|1}; in case of non−nested list p replacement of value only in a unique position of number n is made."

BitGet1::usage =
"The call BitGet1[x, n, p] returns the list of bits in positions of binary representation of an integer n, that are defined by a list p; in case of an integer p the bit in a position p is returned. Whereas the call BitGet1[x, n, p] through a symbol x in addition returns number of bits in binary representation of an integer n."

CodeEncode::usage = "The call CodeEncode[x] returns the coded string x of Latin printable ASCII−symbols, and vice versa."

CodeEncode1::usage =
"The CodeEncode1 procedure is a rather useful extension of the CodeEncode procedure in the event of datafiles of ASCII format. The procedure call CodeEncode1[x] is equivalent to a call CodeEncode[x] if x is a string consisting of ASCII symbols. While the call CodeEncode1[x] in case of x argument defining a datafile consisting of ASCII symbols, returns nothing, updating in situ the initial uncoded datafile x onto the coded file x. At last, the procedure call CodeEncode1[x] in case of x argument defining a coded datafile consisting of ASCII symbols, returns contents of the decoded datafile x, at the same time updating in situ a coded file x onto the decoded datafile x."

StringPosition1::usage =
"The call StringPosition1[x, y] returns a nested list whose sublists define y in string−format along with the starting and ending positions at which y appears as a substring of a string x. The call StringPosition1[x, {y1, y2, ..., yp}] returns a sorted nested list whose sublists define each of yj (j=1..p) in string−format along with the starting and ending positions at which yj appear as a substrings of a string x. At absence of occurrences y in x the empty list is returned, i.e. {}."

ClearAllAttributes::usage =
"The call ClearAllAttributes[x, y, z, ...] cancels all attributes of symbols {x, y, z, ...}. If to symbols {x, y, z, ...} have been assigned values, their names are coded in the string format, i.e. {\"x\", \"y\", \"z\", ...}."

DelRestPF::usage =
"For temporary removal from the current session of the user blocks, functions and modules the procedure DelRestPF serves. The call DelRestPF[\"d\", x, y, z, ...] returns Null with deleting from the current session of the user blocks, functions and modules defined by list {x, y, z, ...} whereas the subsequent call DelRestPF[\"r\"] restores their availability in the current session or in other sessions with preservation of the options and attributes, ascribed to them, returning Null."

DelRestPF1::usage =
"This procedure is an useful extension of procedure DelRestPF. The call DelRestPF1[\"d\", f, x, y, z, ...] returns Null with deleting from the current session of the user blocks, functions and modules defined by list {x, y, z, ...} with their saving in a datafile f, whereas the subsequent call DelRestPF1[\"r\", f] restores their availability in the current session or in other sessions from the datafile f with preservation of the options and attributes, ascribed to them, returning Null."

ContOfContext::usage =
"The call ContOfContext[x] returns the nested 2−element list whose first element defines the sublist of all names in string−format of software of the user package defined by a context x whose definitions received by means of function Definition in the current session are returned with the context x included into them, whereas the second element defines the sublist of all names in string−format of means of the package defined by the context x whose definitions received by means of function Definition in the current session are returned without the context x. In the absence of the context x in the current session $Failed is returned."

SubsDel::usage =
"The call of procedure SubsDel[S, x, y, p] returns the result of deleting from a S−string of all substrings which are limited on the right (at the left) by substring x and at the left (on the right) by the first met symbol in string format of a list y; moreover, search of a y−symbols is made to the left (p = −1) or to the right (p = 1). In addition, the deleted substrings will contain substring x from one end along with the first symbol up to met of y from other end. Furthermore, if not symbols of the list y that were not found, then the rest in the string S is deleted."

HeadPF::usage =
"The following procedure serves as an useful enough means at manipulating with procedures and functions, its call HeadPF[x] returns heading in string format of a block, module or function with a name x activated in the current session, i.e. of function in its traditional understanding with heading. While on other values of argument x the call is returned unevaluated. Meanwhile, the problem of definition of headings is actual also in the case of the objects of the above type of the same name which have more than one heading. In this case the procedure call HeadPF[w] returns the list of headings in string format of the subobjects composing an object w as a whole."

Headings::usage =
"The Headings procedure – an useful enough expansion of the HeadPF procedure in the case of the blocks/functions/modules of the same name but with various headings. Generally the call Headings[x] returns the nested list whose elements are the sublists defining respectively headings of subobjects composing an object x; the first elements of such sublists defines the types of subobjects whereas others define the headings corresponding to them. On x

arguments different from the block/function/module, the procedure call Headings[x] is returned unevaluated."

HeadingsPF::usage =
"Generally the procedure call HeadingsPF[] returns the nested list, whose elements are the sublists defining respectively headings of functions, blocks and modules whose definitions have been evaluated in the current session; the first element of each such sublist defines an object type in the context of {\"Block\", \"Module\", \"Function\"} while the others define the headings corresponding to it. The procedure call returns the simple list if any of sublists doesn't contain headings; at that, if in the current session the evaluations of definitions of objects of the specified three types weren't made, the procedure call returns the empty list. At that, the procedure call with any arguments is returned unevaluated. But it must be kept in mind that performance of the procedure directly depends on stage of the current session when the HeadingsPF procedure has been called and how many definitions for the user means of the type {Function, Module, Block} were calculated in the current session."

HS::usage =
"The HS[Ltf, Cf, P] procedure has 3 formal arguments allowing to define the actual values for: Ltf – a local transition function which is given by the set of parallel substitutions; Cf – an initial finite configuration, and P – the demanded quantity of copies of Cf. The successful conclusion of the analysis prints the initial Cf studied to reproducibility, with number of its copies, and quantity of steps, required for this purpose while through global variable CFfin is returned a final configuration on which P copies was reached."

Predecessors::usage =
"The procedure call Predecessors[Ltf, Co, n] on the basis of a list Ltf that defines the local transition function of a 1−dimensional classical homogeneous structure, of the size n of its neighbourhood template and an initial configuration Co of a finite block of states of elementary automata of the structure returns the list of configurations−predecessors for the block configuration Co. At that, parallel substitutions (x1x2...xn −> x`1) that define the local transition function of the structure in the list Ltf are presented by strings of the form \"x1x2...xn x`1\". The procedure can identify existence for a structure of nonconstructability of NCF−type, printing the appropriate message."

PredecessorsL::usage =
"The procedure call PredecessorsL[Ltf, Co, n, Cf] on the basis of a list Ltf that defines the local transition function of a 1−dimensional classical homogeneous structure, of the size n of its neighbourhood template and an initial configuration Co of a finite block of states of elementary automata of the structure returns the number of configurations−predecessors for the block configuration Co, whereas through Cf−argument the list of all predecessors is returned. At that, parallel substitutions (x1x2...xn −> x`1) that define the local transition function of the structure in the list Ltf are presented by strings of the form \"x1x2...xn x`1\". The procedure can identify existence for a structure of nonconstructability of NCF−type."

PredecessorsR::usage =
"The procedure call PredecessorsR[Ltf, Co, n, Cf] on the basis of a list Ltf that defines the local transition function of a 1−dimensional classical homogeneous structure, of the size n of its neighbourhood template and an initial configuration Co of a finite block of states of elementary automata of the structure returns the number of configurations−predecessors for the block configuration Co, whereas through Cf−argument the list of all predecessors is returned. At that, parallel substitutions (x1x2...xn −> x`1) that define the local transition function of the structure in the list Ltf are presented by strings of the form \"x1x2...xn x`1\". Its difference from the procedure PredecessorsL is considered in the above books. The procedure can identify existence for a structure of nonconstructability of NCF−type."

NcfQ::usage = "The procedure call NcfQ[Ltf, S, t] returns True, if a finite block configuration S is the NCF in a classical structure 1–HS with local transition function, defined by the list of parallel substitutions Ltf, and in alphabet A={0,1,...,t} (t ≤ 9); otherwise the procedure call returns False."

MinNCF::usage =
"The procedure call MinNCF[Ltf, n] returns the minimal block with NCF for a classical binary structure 1–HS with the local transition function, defined by the list of parallel substitutions Ltf; in addition, n defines a predictable size of minimal NCF; furthermore, it is supposed that the structure possesses the NCF nonconstructability."

NfToLtf::usage =
"The procedure NfToLtf has an auxilary character; its call NfToLtf[p, n] (p – length of neighbourhood template, n – an integer < 2^2^m) returns the list of parallel substitutions defining a local transition function for binary 1–HS with a discriminating number n. Note, the procedure NfToLtf along with use for the problem considered here can be a rather useful in a number of other problems of experimental study of binary 1–HS."

ComposeGTF::usage =
"The procedure call ComposeGTF[g] returns the 2–element list, whose first element defines the neighbourhood template size and the second defines the discriminating number of GTF that is result of composition of two or more global transition functions, whose the local transition functions are determined by neighbourhood template sizes {n1,...,nk} and discriminating numbers {m1, ..., mk} respectively. In addition, in the call ComposeGTF[g] the pairs {nj,mj} (j=1..k) are used as argument g, i.e. ComposeGTF[n1, m1,...,nk, mk]; number of such pairs should be at least two. Meanwhile, if the procedure call contains odd number of actual arguments (as the last argument can be used an arbitrary expression) the list of parallel substitutions defining the LTF of the resulting GTF is returned. The procedure ComposeGTF processes the basic especial and erroneous situations."

GtfMod2::usage =
"The call GtfMod2[n] returns the discriminating number of a structure 1–HS with LTF σ(n)=Σj xj (mod 2); xj∈{0,1}; j=1..n."

CFsequences::usage =
"The procedure call CFsequences[Co, A, Ltf, n] prints the sequence of configurations generated by a 1–CA with alphabet of states A = {0,1, ..., p} (p = 1..9), local transition function Ltf from a finite configuration Co given in string format for n steps of the automaton. At the same time, a function of the kind F[x, y, z, ..., t] := x∗, and the list of substitutions of the kind \"xyz...t\" –> \"x∗\" {x, x∗, y, z, ..., t}∈A can act as the third argument Ltf. The procedure processes basic mistakes arisen at encoding an initial configuration Co, an alphabet A and/or a local transition function Ltf, returning $Failed with output of strings with the appropriate messages."

GtfMod2Q::usage =
"The procedure call GtfMod2Q[n, m] returns True if composition τ1(n)τ2(m) of two 1–dimensional GTF with LTF of the above form is a GTF τ(n+m–1) with LTF of the same form, and False otherwise."

FunctionToRules::usage =
"In some cases at computer research of 1–CA it is preferable to define their local transition functions not functionally in the form Ltf[x_, y_, z_, ...] := x∗, but in the form of the list of parallel substitutions of the format \"xyz ... \" –> \"x∗\". The given problem is solved by a simple enough function, whose call FunctionToRules[x, A] returns the list of parallel substitutions of the above format on the basis of a function x and alphabet A = {0, ..., n} (n = 1..9)."

SystemQ::usage = "The call SystemQ[S] returns True if an object with name S is system object, i.e. is determined by package Mathematica, and False otherwise."

SysFunctionQ::usage =
"The procedure call SysFunctionQ[x] returns True if x determines a function of the Mathematica language, and False otherwise."

StringMultiple::usage =
"The procedure call StringMultiple[s, p] with two arguments returns a string – result of p–fold concatenation of a string s. Whereas the call StringMultiple[s, p, h] on three arguments (3–rd an arbitrary expression) returns the result of p–fold concatenation of a string s parted by a string h. In case of receiving of an inadmissible tuple of actual arguments the call of procedure is returned unevaluated."

StringMultiple1::usage = "The procedure call StringMultiple1[s, p] returns the string – result of p–fold concatenation of a string s. The procedure is a simple variant of the StringMultiple procedure."

NbCallProc::usage =
"The call NbCallProc[x] returns Null, i.e. nothing, over–activating the definition of a procedure/function x in the current session, whose definition was in a nb–file loaded and was previously activated in the current session.It is necessary to note, that the call of procedure NbCallProc[x] over–activates all definitions of procedures/functions with an identical name x and with various headings in the current session."

ListStrList::usage =
"The call of procedure ListStrList[x] on a list x = {a, b, ...} returns a string s of the format \"ahbh...\", whereas x = ListStrList[s], where h = FromCharacterCode[2]. In case of absence in a s–string of the h–symbol the call ListStrList[s] returns the string s. So, the calls of procedure ListStrList operate by principle of the switch."

OP::usage = "The procedure call OP[x] returns the list of atomic elements composing an expression x."

Op::usage = "The procedure call Op[x] extracts operands from an expression x. The procedure is an analog of built–in function of Maple accurate within axiomatics of both packages."

UnDefVars::usage = "The call UnDefVars[x] returns the list of all independent variables in an arbitrary expression x."

UnDefVars1::usage =
"The call UnDefVars1[x] returns the list of all independent variables in string format in an arbitrary expression x."

Sequences::usage = "The function Sequences[x] generalizes the standard function Sequence[x], allowing a list {a, b, c, ...} as an actual argument x and returning Sequence[a, b, c, ...]."

Sq::usage = "The function Sq[x] is a modification of the above function Sequences[x]."

ProtectedQ::usage = "The call ProtectedQ[x] returns the True if x has attribute Protected, and the False otherwise."

ExprOfStr::usage =
"The call ExprOfStr[w, m, n, L] returns the result of extracting from a string w, limited by its m–th position and the end, of the first correct expression on condition that search is made to the left (n = –1)/to the right (n = 1) from the given position and a symbol following or in front of a found expression should belong to a

list L. The result of the procedure call is returned in string−format; at absence of a correct expression $Failed is returned, while the call on inadmissible actual arguments is returned unevaluated."

ExprOfStr1::usage =
"The call ExprOfStr1[x, n, p] returns a substring minimal on length of a string x in which the boundary element is a symbol in n−th position of the string x containing correct expression. In addition, search of such substring is made to the right from n−th position and prior to the end of the string x at p=1, and to the left from n−th position of the string x and prior to the beginning of the string x at p=−1. In case of absence of such substring the call of procedure ExprOfStr1[x, n, p] returns $Failed."

IndexedQ::usage = "The call IndexedQ[Quiet[x]] returns the True if x is an indexed expression, and the False otherwise."

IndexQ::usage =
"The call IndexQ[x] returns True if x given in string−format is an indexed expression; otherwise, False is returned. In addition, under the indexed is understood any expression in a reduced form which ends with the index bracket \"]]\"."

Index::usage = "The call Index[Quiet[x]] returns the index of an indexed expression x; otherwise the call is returned unevaluated."

Indices::usage = "The call Indices[x] returns the index part of an
indexed expression x, given in string−format; otherwise, the call is returned unevaluated."

ListStrToStr::usage =
"Procedure ListStrToStr represents indubitable interest in work with lists in the string format, more precisely, the call of procedure ListStrToStr[x] with single factual argument x where argument x has format {\"a\", \"b\", \"c\",...}, converts x into string of the format \"a, b, c, ... \"; if the procedure call uses any additional expression p as the second argument, the call returns a string of the format \"abcd...\"."

ExprQ::usage =
"The procedure ExprQ provides testing of a string construction W for presence in it of a correct expression; in addition, not a substring of W (substring as an expression), but a string W is completely considered.
Successful call of ExprQ[W] returns True; otherwise, False is returned."

Nvalue::usage =
"The procedure call Nvalue[w] returns the list of global names to which in the current session the value w has been assigned."

Nvalue1::usage =
"The procedure Nvalue1 is an extension of functionalities of earlier presented procedure Nvalue. The call of procedure Nvalue1[x] returns the list of names of variables in string−format for which in the current session a value x had been assigned."

ProcsAct::usage =
"The procedure call ProcsAct[] returns the nested 5−element list in which the sublists as the first element define types of objects in the context of \"Block\", \"Module\", \"DynamicModule\", \"Function\" and \"Others\", whereas other elements of sublists define names in string format of the user objects that have been activated in the current session and have the appropriate type. The user objects can be activated by means of immediate evaluations of their definitions or by mediated activation by the user packages containing them."

RemProcOnHead::usage =
"The successful call of the procedure RemProcOnHead[x] returns \"Done\", having removed from the current session a procedure/function or their list with heading or accordingly with list of the headings x that are given in the string format; at that, on inadmissible factual argument x the procedure call returns $Failed or is returned unevaluated.
At the same time, the remaining subobjects with the name of the processed object x save options and attributes, except in the case when the object x is removed completely. At that, it is necessary to do two remarks, namely: (1) the means of this fragment which are used as examples have only formally correct code, no more, and (2) a heading in the procedure call RemProcOnHead[x] is coded according to the format ToString1[x], as the following example illustrates:

In[2564]:= ToString1[F[x_/;SameQ[x,\"avz\"],y_]]
Out[2564]= \"F[x_ /; x === \"avz\", y_]\"."

RemProcOnHead1::usage =
"Procedure RemProcOnHead1 is an extension of the above procedure RemProcOnHead onto case of existence of attributes for an object with heading x. The call RemProcOnHead1[x] returns \"Done\", having removed from the current session a separate block/function/module with heading x or blocks/functions/modules with the headings determined by the list x, irrespective of existence of attributes at object with the name HeadName[x]. The procedure call on the heading of an inactive object returns \"Done\" also with output of the corresponding message. In addition, the remaining subobjects with name HeadName[x] keep options and attributes, excepting the case when the object with name HeadName[x] is removed completely."

LocObj::usage =
"The call LocObj[x] returns the three−element list, whose first element defines an object x, the second defines its type in the context of {\"Module\", \"SFunction\" (system function), \"Function\", \"Expression\"}, whereas the third – its initial location in the context of {\"Global`\" – the current session, \"System`\" –

library or kernel of Mathematica, and \"Context\" – a system or user package, which is associated with the Context and is loaded into the current session, and contains the definition of the object x}."

FileExistsQ1::usage =
"The call of procedure FileExistsQ1[x] with one actual argument returns the True if x defines a real datafile in system of directories of a computer, and False otherwise; whereas the call FileExistsQ1[x, y] in addition through 2−nd actual argument returns the list of full paths to the found datafile x if the basic result of the call is True."

RestoreDelPackage::usage =
"The RestoreDelPackage procedure restores the packages removed by Del command of Ms Dos system, and by our DeleteFile1 function from the Recycler directory. The successful call RestoreDelPackage[F, \"Context'\"], where – the first argument F defines a name of datafile of the format {\"cdf\", \"m\", \"mx\", \"nb\"} which is subject to restoration, whereas the second argument defines a context associated with the package, returns the list of full paths to the restored datafiles, at the same time with deleting from the Recycler directory of the restored datafiles with the demanded package."

RestoreDelFile::usage =
"The successful call RestoreDelFile[F, r], where the first argument F defines the name of a datafile or their list that are subject to restoration whereas the second argument determines the name of a target directory or full path to it for the restored datafiles returns the list of paths to the restored datafiles; at the same time, the deleting of the restored files from the directory Recycler Bin isn't done. In the absence of the requested files in the directory Recycler Bin the procedure call returns the empty list, i.e. {}. It should be noted that only nonempty datafiles are restored. If the second argument r defines a directory name in string format, but not the full path to it, a target directory r is created in the active directory of the current session."

ListOp::usage =
"The procedure call ListOp[x, y, z] returns the list whose elements are results of application of a procedure/function z to the corresponding elements of lists x and y; in addition, in case of various lengths of the given lists the described procedure is applied to both lists within the minimal length of lists, leaving other elements of the greater list without change."

ListsAssign::usage =
"The package does not support the assignments such as x = y in a case of different lengths of lists x and y whereas this operation is supported by the call ListsAssign[x, y], returning the renewed list x. In addition, the procedure processes the erroneous and especial situations conditioned by assignments x = y."

Avg::usage =
"Procedure Avg is internal, i.e. the procedure call Avg[] is meaningful only in the body of other block or module, returning the list of nesting {1|2}, whose elements define or local variables in string−format of the block/module external in relation to Avg, or 2−element lists whose first elements define local variables in string−format of external procedure whereas the second ones − their initial values in string format; in addition, lack of initial value is coded by the symbol \"None\". At absence for the external block/module of local variables the procedure call Avg[] returns the empty list, i.e. {}. The call Avg[] outside a block/module doesn't make special sense, returning the list of the above format for two local variables of the Avg procedure."

NestQL::usage = "The procedure call NestQL[L] returns the True if L is the nested list; otherwise, the False is returned."

Tuples1::usage = "The procedure call Tuples1[A, n] returns the list of all possible tuples in string format of length n, composed from symbols of an alphabet A."

SelectPos::usage =
"The SelectPos procedure provides the choice of elements from a list according to the given positions. The call SelectPos[x, y, z] returns the list of elements of a list x, whose positions are different from elements of a integer list y at z = 1, while at z = 2 the list of elements of the list x whose positions coincide with elements of the integer list y is returned."

ActRemObj::usage =
"The procedure call ActRemObj[x, y] depending on value {\"Act\", \" Rem\"} of the second factual argument deletes from the current session of the package an object defined by its name in string−format, or activates it in the current session or in other session accordingly. Successful removal of an object from the current session of the package returns the value \"Remove\" whereas its restoration in the current session of the package returns the value \"Activate\". If the file containing definition of a removed object, has not been found in the user home directory $HomeDirectory the procedure call ActRemObj is returned unevaluated. In addition, on inadmissible argument x the call ActRemObj[x, y] returns $Failed."

AtomicQ::usage = "The procedure call AtomicQ[x] yields True if x is an expression which cannot be divided into subexpressions, and yields False otherwise."

StringTake1::usage = "The procedure call StringTake1[x, y] returns the list of substrings of a string x which are partitioned by substrings y; any expression or their list can be as an argument y."

StringTake2::usage =
"The procedure StringTake2[x, y] is functionally equivalent to the procedure StringTake1 but it realizes other algorithm. At that, for StringTake2 the strings should not be included in the second argument."

StringTake3::usage = "The procedure call StringTake3[x, y] returns the list of substrings of a string x which are partitioned by substrings y; any expression or their list can be as an argument y."

Decomp::usage = "The call Decomp[Expr] returns the list of atomic components composing an expression Expr, including names of variables, functions, procedures, operations along with constants."

ProcCalls::usage =
"Objects of type {Block, Function, Module}, and in combination with objects of other types can be as objects of the same name; in whole series of cases at calculations with such objects especial or erroneous situations arise. For removal from the objects of the same name of subobjects of types, different from {Block, Function, Module} the quite simple procedure can be useful whose the call ProcCalls[g] returns Null, i.e. nothing, removing from the sublist of the object g of the same name the subobjects of types, different from {Block, Function, Module}. Argument g is coded in string format."

ProcCalls1::usage =
"The ProcCalls1 procedure is a rather interesting analog of the ProcCalls procedure. The given procedure is based on the TestDefBFM procedure; the call ProcCalls1[w] returns Null, i.e. nothing, removing from the sublist of an object w of the same name the subobjects of types, different from {Block, Function, Module}. At that, the argument w is coded in string format."

ScanLikeProcs::usage =
"In addition to the procedure ProcCalls with the purpose of definition of the same procedures of the current session of the package rather simple procedure is intended, whose call ScanLikeProcs[] returns the list of all same procedures which have been activated in the current session whereas as a result of call ScanLikeProcs[W] in addition through variable W the list of headings of similar procedures in string–format is returned."

TestDefBFM::usage =
"The procedure call TestDefBFM[x, y] returns True if a string x contains the definition of an object of type {Block, Function, Module} with a name y given in string format, and False otherwise."

Default1::usage =
"The call Default1[x, y, z] returns Null, i.e. nothing, providing setting of default values, defined by a list z, for arguments of a function/procedure x, whose positions are defined by a list y of PosIntList–type."

Defaults::usage =
"The procedure Defaults[G, y] for any subtuple of a tuple of formal arguments of a block, a function or a module G which is defined by a 2–element list y (the first element – number of position of an argument; the second element – an expression) provides the setting of an expression as value by default for a formal argument. For several values by default the list y has ListList–type, whose sublists have the specified format. Procedure successfully works with objects of the given type of the same name, processing only the first subobject from the list of the subobjects returned by the call Definition[G]. The successful call returns Null, i.e. nothing, whereas in special situations the call returns $Failed or is returned unevaluated."

DefaultsM::usage =
"The procedure DefaultsM extends the above procedure Defaults onto case of objects of the same name. The procedure DefaultsM[G, y] for any subtuple of a tuple of formal arguments of a block, a function or a module G which is defined by a 2–element list y (the first element – number of position of an argument; the second element – an expression) provides the setting of an expression as value by default for a formal argument. For several values by default the list y has ListList–type, whose sublists have the specified format. Procedure successfully works with objects of the given type of the same name, processing all subobjects from the list returned by the call Definition[G]. The successful call returns Null, i.e. nothing, whereas in special situations the call is returned unevaluated."

DefaultValues1::usage =
"The call DefaultValues1[x] returns the list of format {{N1} :> V1,..., {Np} :> Vp}, where Nj and Vj (j=1..p) – numbers of positions of formal arguments in a function/procedure x along with default values, ascribed to them, accordingly, irrespective of the method of their definition; in addition, the priority is taken into account (the setting of default values in heading of functions/procedures has the highest priority)."

DefaultValues2::usage =
"The following DefaultValues2 procedure to a certain degree extends the DefaultValues1 procedure, its call DefaultValues2[x, y, z], where y – the list of the positions of arguments in the heading of x and z – the list of default values corresponding to them, returns the list of format {{N1} –> V1, ..., {Np} –> Vp}, where Nj and Vj (j=1..p) – numbers of positions of formal arguments in a function/procedure x and default values, ascribed to them, accordingly, irrespective of the method of their definition; at that, the priority is taken into account (the settings of default values in heading of the functions/procedures have the highest priority). The procedure call does in the current session x as an active object of the above type with the new default values with saving of options and attributes of old x object."

MapInSitu::usage =
"The call MapInSitu[x, y] returns the result of evaluation Map[x, y]; in addition a factual argument y is updated in situ. The procedure demands to code argument y in string format. The procedure MapInSitu1[x, y] is a modification of the above procedure, updating y in situ also, however every time for argument y must be used the same identifier y."

MapInSitu1::usage = "The procedure MapInSitu1[x, y] is a modification of the above procedure MapInSitu, updating y in situ also, however every time for argument y must be used the same identifier y."

MapInSitu2::usage =
"The procedure MapInSitu2[x, y] is a modification of the above procedures MapInSitu and MapInSitu1, updating y in situ also, however instead of factual argument y is used an expression but not an identifier y to which this expression was ascribed. In addition, all identifiers for which in the current session the above expression was ascribed obtain value Map[x, y]; it constitutes the essence of updating in situ."

MemberQ1::usage =
"The procedure call MemberQ1[L, x, y] returns the True if x is an element of any level of nesting of a list L (provided that a non−nested list has level of nesting 0); otherwise, the False is returned. In case of return True through the third argument y the list of levels of the list L which contain occurrences of x−value is returned. The procedure MemberQ1 in the certain degree expands the standard function MemberQ onto the nested lists."

MemberQ2::usage =
"The procedure call MemberQ2[L, x, y] returns the True if x is an element of a list L; otherwise, the False is returned. In case of return the True through the third argument y the number of occurences of a x−expression into list L is returned. The procedure MemberQ2 in the certain degree expands the standard function MemberQ onto the number of occurrences of x−expressions into lists."

MemberQ3::usage =
"The call MemberQ3[x, y] returns True if all elements of the list y belong to a list x, however without taking into consideration the nesting, and False otherwise. Whereas the call MemberQ3[x, y, t] with the third optional argument – an arbitrary expression – returns True, if the list y is a sublist of the list x on any nesting level, and False otherwise."

MemberQ4::usage =
"The call MemberQ4[x, y] returns True if at least one element of a list y or an element y is a part of the list x, and False otherwise. Whereas the call MemberQ4[x, y, z] where z – optional positive integer – returns the True if at least z elements of a list y are a part of the list x, and False otherwise."

MemberT::usage = "The procedure call MemberT[L, x] returns the total number of occurences of an expression x into a list L."

MemberQL::usage =
"The call MemberQ[x,y] returns True if an expression y belongs to any nesting level of a list x, and False otherwise. Whereas the call with the third optional argument z – an undefinite variable – in addition through z returns the list of ListList–type, the first element of each its sublist defines a level of the list x whereas the second defines quantity of elements y on this level provided that the main output is True, otherwise z remains undefinite."

MinusList::usage = "The call of procedure MinusList[x, y] returns the result of subtraction of a list y from a list x which consists in removal from the list x of all occurrences of elements from the list y."

MinusLists::usage =
"The call MinusLists[x, y, 1] returns result of subtraction of a list y from a list x that consists in deletion in the list x of all occurrences of elements from the list y. Whereas the call MinusLists[x, y, 2] returns result of subtraction of a list y from a list x which consists in parity removal from the list x of entries of elements from the list y, i.e. the number of the elements deleted from the list x strictly correspond to their number in the list y."

HelpPrint::usage =
"The call of procedure HelpPrint[n] prints information in string−format concerning all means of the user package. The mechanism of use of the procedure is simple enough, namely: during work with Mathematica in mode of the document on a certain step, in particular, Out[n] into the current session is loaded a package which is located in a datafile of the format {\"cdf\", \"m\", \"mx\", \"nb\"} or is reloaded already loaded package, then the call HelpPrint[n+1] prints the above information, prenex by a context associated with the package."

HelpBasePac::usage =
"The function call HelpBasePac[w] prints the contents of help database of all means provided by usages, of a package with context w on condition of its availability in the current session. Whereas the call HelpBasePac[w, p], where p – an arbitrary expression, additionally prints the definitions of all such means of the package w."

FreeQ1::usage =
"The simple procedure FreeQ1 enough essentially extends the standard function FreeQ, providing the extended testing of occurrences in an expression of subexpressions. FreeQ1[x, y] returns True if an expression x does not contain subexpressions y; otherwise, False is returned."

FreeQ2::usage =
"The function FreeQ2 expands the standard function FreeQ onto the list as the second actual argument. The call FreeQ2[x, p] returns True if an expression x do not contain a subexpression p or subexpressions from a list p; otherwise, False is returned."

MinusList1::usage = "The call of procedure MinusList1[x, y] returns result of subtraction of the list y from the list x which consists in parity removal from the list x of all occurrences of elements from the list y."

ModLibraryPath::usage =
"The procedure call ModLibraryPath[x] extends the list, defined by the global variable $LibraryPath onto subdirectory x with the user dynamical library. The procedure call returns the Null, i.e. nothing, with saving of attributes of variable $LibraryPath."

ListToString::usage = "The procedure call ListToString[L] returns the result of converting into an integrated string of all elements of a list L, disregarding its nesting."

ExtProgExe::usage =
"The following procedure facilitates the solution of the problem of use of the external Mathematica programs or operational platform. The procedure call ExtProgExe[x, y, h] provides search in file system of the computer of a {exe| com} file with the program with its subsequent execution on parameters y of the command string. Both arguments x and y should be encoded in string format. Successful performance of the given procedure returns the full path to \"$TempFile$\" datafile of ASCII–format containing result of execution of a program x, and this datafile can be processed by means of standard means on the basis of its structure. At that, in case of absence of the datafile with the demanded program x the procedure call returns $Failed while at using of the third optional argument h – an arbitrary expression – the datafile with the program x uploaded into the current directory determined by the call Directory[], is removed from this directory; also the datafile \"$TempFile$\" is removed if it is empty or implementation of the program x was terminated abnormally."

FullUserTools::usage =
"The procedure call FullUserTools[x] returns the list of names in string format, that enter in definition of the active user block/function/module x; in addition, the first element of the list is a context of these tools. Whereas in a case of tools with different contexts the call returns the nested list of sublists of the above format. In turn, the procedure call FullUserTools[x, y] through the optional argument y – an undefinite variable – returns 2–element list whose the first element defines the list of tools without usages, and the second element defines the unidentified tools."

FullToolsCalls::usage =
"Unlike the FullUserTools procedure the FullToolsCalls procedure provides the analysis of the user block, function or module regarding existence in its definition of calls of both the user and the system means. The procedure call FullToolsCalls[x] returns the list of names, whose calls are in definition of the active user block/function/module x; in addition, the first element of the list is a context of these tools. While in case of means with different contexts the procedure call returns the nested list of sublists of the above format. In case of absence in a x definition of the user or system calls the procedure call FullToolsCalls[x] returns the empty list, i.e. {}."

FullToolsCallsM::usage =
"Unlike the previous procedure the FullToolsCallsM procedure provides the above analysis of the user block, function or module of the same name. The procedure call FullToolsCallsM[x] returns the nested list of results of application of the FullToolsCalls procedure to subobjects (blocks, functions, modules) which compose an object of the same name x. The order of elements in the returned list corresponds to an order of definitions of the subobjects returned by the call Definition[x]."

AllCalls::usage =
"Unlike the previous FullToolsCallsM procedure the AllCalls procedure provides the analysis of the user block, function or module in the context of full form of calls entering in it. The procedure call AllCalls[x] returns the nested list of sublists containing the full form of calls entering in definitions of subobjects that compose an object of the same name or a simple object x. The order of elements in the returned list corresponds to an order of definitions of the subobjects returned by the call Definition[x]."

MemberLN::usage =
"The procedure call MemberLN[L, x] returns the list of ListList–type, for which each sublist defines number of a level of nesting of a nested list L by its first element, along with number of occurrences of an expression x into the given level by its second element."

ExpArgs::usage =
"The call ExpFunc[G, {x, y, z, ...}] provides expansion of the tuple of formal arguments of a function, module or block G by arguments {x, y, z, ...} to the right from the tuple, with returning Null, i.e. nothing, and activation in the current session of the updated definition of the object G. Expansion of the tuple of formal arguments is made for object of G only on variables from the list {x, y, z, ...} which aren't neither its formal arguments, nor local variables; otherwise any expansion isn't made. List of elements {x, y, z, ...} for updating can be both symbols in string format, and names of arguments with tests attributed by it for their admissibility. In addition, the call ExpArgs[G, x] on inadmissible object G, in particular, on system functions, or on the empty list x is returned unevaluated."

Npackage::usage =
"The function call Npackage[w] returns the list of names in string–format of objects whose definition contain in a package w activated in the current session. In a case of inactivity in the current session of a package x or its absence the function call returns the value $Failed."

DefInPackage::usage =
"For test of a package loaded into the current session or a not loaded package being in a m–datafile, concerning existence

in it of global and local objects exists the procedure DefInPackage. The call DefInPackage[x], where x defines a m–datafile or a full path to it or the context associated with a package, returns the nested list, whose first element defines a context of the package, the second – the list of local variables, and the third – the list of global variables of the package x. If x doesn't define a package or a context, the call DefInPackage[x] is returned unevaluated. In case of a fictitious context x the procedure call returns $Failed."

FullCalls::usage =
"The call FullCalls[x] returns the list whose the first element is the context associated with a package loaded into the current session, whereas its other elements are symbols from the package that are used by a procedure or a function x. In case of use by x of symbols from some packages, the call returns the nested list from lists of the above type."

FullCalls1::usage =
"The call FullCalls1[x] returns the nested list whose the first element is the result of the call FullCalls1[x] of the above procedure, whereas the second defines the names of procedures/functions, used by x, excluding the means belonging to the loaded user packages."

FindFileObject::usage =
"In some cases there is a problem of finding of a m−file containing definition of some x−object, active in the current session. The given problem is solved successfully with procedure, whose the call FindFileObject[x] returns the list of the files containing definition of object x, including the standard help on it; at absence of such m−files the call of procedure returns the empty list. The call FindFileObject[x, y, z, ...] with unessential arguments {y, z, ...} − the names in string−format of devices of external memory with direct access − provides search of m−files on the specified devices instead of search on all file system of a computer in case of call of procedure with 1 argument. On undefined argument x the call is returned unevaluated."

UserLib::usage =
"The procedure UserLib provides a series of main functions for maintenance of the user library of procedures/functions located in a txt−datafile L. The procedure call UserLib[L, f] executes the functions with a library datafile given by its name L or by full path to it according to the second factual argument that represents 2−element list f in the following way, namely:

{\"names\", \"list\"} − the return of the list of names of procedures/functions, whose definitions are in a library datafile L; in case of the empty datafile the procedure call is returned unevaluated;

{\"print\", \"all\"} − the print on the screen of all contents of a library datafile L; whereas in case of the empty datafile the procedure call is returned unevaluated;

{\"print\", \"Name\"} − the print on the screen of definition of a procedure/function with name Name, being in a library datafile L; for the empty datafile the procedure call is returned unevaluated; at absence in the library datafile L of required facility the procedure call returns the Null, i.e. nothing; in that case the procedure call prints the message of the following general form \"Name is absent in Library L\";

{\"add\", \"Name\"} − saving in a library datafile L in Append−mode of a procedure or function with name Name; its definition should be preliminary evaluated in the current session of the package in Input−mode;

{\"load\", \"all\"} − upload into the current session of all means whose definitions are in a library datafile L; whereas in case of the empty datafile the procedure call is returned unevaluated;

{\"load\", \"Name\"} − upload into the current session a procedure/function with name Name, whose definition is in a library datafile L; whereas in case of the empty datafile L the procedure call is returned unevaluated; in case of absence in the datafile L of a required facility the procedure call returns the Null, i.e. nothing, printing the message of the following general form \"Name is absent in Library L\".
In other cases the procedure call is returned unevaluated."

SequenceQ::usage = "The call Sequence[x] returns the True if x is an object defined by the function Sequence, and the False otherwise; at that, name x of the object is coded in string−format."

SortQ::usage = "The call SortQ[x] returns value True if x is a string, sorted symbol−by−symbol, and value False otherwise."

SortLS::usage =
"The function SortLS is an expansion of standard function Sort onto separate strings and integers. The call SortLS[x, y] allows a list with elements of type {Numeric, Symbol, String}, a string and an integer as the first argument x; the second optional argument y has the same sense, as and for the function Sort."

EmptyFileQ::usage =
"The call EmptyFileQ[f] returns True if a datafile f is empty, and False otherwise. If a datafile f is absent in file system of the computer, the call EmptyFileQ[f] returns the $Failed. Moreover, if in the course of search of the datafile f its multiplicity in file system of the computer is detected, all datafiles from list of the found datafiles are tested, including also datafiles that are located in the Recycle Bin directory. At that, the procedure call EmptyFileQ[f, y] with two actual arguments where optional argument y – an expression, returns the nested 2–element list whose first sublist defines emptiness/nonemptiness (True/False) of the datafile f in the list of datafiles of the same name whereas the second sublist defines full paths

to the datafiles f of the same name. At that, between both sublists the one–to–one correspondence takes place."

Contexts1::usage = "The call Contexts1[] returns the list of contexts corresponding to packages whose components are activated in the current session. The procedure is a modification of the standard function Contexts."

ContextQ::usage =
"The following simple function provides testing of a string for its admissibility as a context; in addition, the admissibility is considered from the point of view of syntactic correctness only. The call of function ContextQ[x] returns True, if x – a potentially allowable context, and False otherwise."

ContextSymbol::usage = "The call ContextSymbol[x] returns the context associated with a symbol x."

ContextToSymbol::usage =
"Whereas the procedure call ContextToSymbol[x, y] returns the list of format {x, {y}}, ascribing in the current session a context y to a definite symbol x. The symbol with a new context keeps attributes and usage of an old symbol. In particular, the given means is quite useful in the case of necessity of saving of objects in mx–files."

ToContext::usage =
"Unlike the above ContextToSymbol procedure the procedure call ToContext[x, y] returns nothing, providing the assignment of a context y to a symbol or their list x. At the same time it must be kept in mind that a procedure call ToContext[x, y], providing assignment of a new context y to symbols x, at the same time doesn't keep the attributes and options attributed to them. It is necessary to do it outside of the ToContext procedure."

ContextToSymbol1::usage =
"The ContextToSymbol1 procedure provides assignment of the given context to a definite or undefinite symbol. The procedure call ContextToSymbol1[x, y, z] returns Null, i.e. nothing, providing assignment of a certain y context to a symbol x; at that, the third optional argument z – the string, defining for x the usage; at its absence for an undefinite symbol x the usage – empty string, i.e. \"\", while for a definite symbol x the usage has view \"Help on x\". At that along with possibility of assignment of the given context to symbols the ContextToSymbol1 procedure is an useful enough means for extension by new means of the user package contained in a mx–file. The technology of similar updating is as follows. On the first step a file x of mx–format with the user's package having y context is uploaded into the current session by the function call Get[x]. Then, in the same session the definition of a new means f with its usage u which describes the given means is evaluated. At last, by the procedure call ContextToSymbol1[f, y, u] the assignment of a y context to the symbol f along with its usage u is provided. Moreover, the usage u can be directly coded in the procedure call, or be determined by a certain string u. At last, by the function call DumpSave[x, y] the saving in the mx–file x of all objects having y context is provided. Similar approach provides a rather effective mechanism of updating in the context of both definitions and usages of the means entering the user's package which is located in a mx–file."

ContextToSymbol2::usage =
"The next useful procedure in a certain relation extends the ContextToSymbol1 procedure and provides assignment of the given context to a definite symbol. The successful procedure call ContextToSymbol2[x, y] returns two–element list of the format {Cont`x, y}, where Cont – a previous context of a definite x symbol and y – its new context, providing assignment of a certain y context to the definite symbol x; while the procedure call ContextToSymbol2[x, y, z] with optional third argument z – an arbitrary expression – also returns two–element list of the above format {Cont`x, y}, providing assignment of certain y context to the definite symbol x with saving of symbol definition x along with its attributes and usage in datafile \"x.mx\".
Therefore, before a procedure call ContextToSymbol2[x, y] for change of the existing context of a symbol x onto a new symbol y in the current session it is necessary to evaluate definition of a symbol x and its usage in the format x::usage = \"Help on an object x.\" (if the usage exists) with assignment to the symbol x of necessary attributes.
At that along with possibility of assignment of the given context to symbols the ContextToSymbol2 procedure is an useful enough means for extension by new means of the user package contained in a mx–file. The technology of similar updating is as follows. On the first step the definition of a new means x with its usage u which describes the given means is evaluated along with ascribing of the necessary attributes. Then, by means of the procedure call ContextToSymbol2[x, y, j] the assignment of a y context to a symbol x along with its usage with saving of the updated object x in \"x.mx\" file is provided. At last, the function call Get[p] loads in the current session the revised user package located in a mx–file p with y context, whereas the subsequent call DumpSave[p, y] saves in the updated mx–file p all objects having y context, including the objects that earlier have been obtained by the procedure call ContextToSymbol2[x, y] or Get[\"x.mx\"]. Such approach provides a rather effective mechanism of updating of the user packages located in mx–files."

ContextRepMx::usage =
"The procedure call ContextRepMx[x, y] provides replacement of the context j of a mx–file x by a new context y, returning the list of format {x, j, y} where x defines file with result of such replacement, j – an old context and y – a new context. At that, if context in file x is absent, the call returns $Failed. The call ContextRepMx[x, y, z] where z defines a mx–file with result of replacement returns {z, j, y}."

ContextActQ::usage =
"The simple enough function ContextActQ provides check of a context for the purpose of its presence in the list of contexts of the current session. The call ContextActQ[w] returns True if a context w is in the list of all contexts of the current session; otherwise, False is returned."

FindFileContext::usage =
"The call of standard function Context[x] returns the context which is associated with a symbol x. Meanwhile, a m−file with the package containing the given context is represented as interesting enough. Call FindFileContext[x] returns the list of files with the packages containing the given context x; at absence of such files the call of procedure returns the empty list, i.e. {}. In addition, the call FindFileContext[x, y, z, ...] with optional arguments {y, z, ...} − the names in string−format of devices of external memory with direct access − provides search of required files on the specified devices instead of search on all file system of the computer in case of the call of procedure with one actual argument."

FindFileContext1::usage = "Depending on the status of a context x the procedure call FindFileContext1[x] returns the following values:
{\"Current\", {m−files}} – the context x is the current and is in m−files;
\"Current\" – the context x is the current, but isn't associated with m−files;
{m−files} – the context x is in m−files, but not in the list $ContextPath;
{} – the context x is formally correct, but not factual."

ContextInFile::usage =
"As an essential enough addition to the above procedures FindFileContext and FindFileContext1 is the ContextInFile procedure providing search of datafiles of the types {\"cdf\", \"m\", \"mx\", \"nb\", \"tr\"} containing definitions of packages with the given context. The procedure call ContextInFile[x, y] returns the list of full paths to datafiles of the indicated types containing definitions of packages with a context x. At that, search is executed in a directory, defined by the second optional argument y; in its absence the search of datafiles is executed in the \"C:\" directory. Return of the empty list, i.e. {}, determines absence of the sought−for datafiles in the given path of search."

SeqUnion::usage = "The call SeqUnion[x, y, z, ...] returns the
result of union of sequences <x, y, z, ...> where arbitrary expressions may be as arguments."

Rename::usage =
"In the regular mode the procedure Rename[x, y] returns Null−value, i.e. nothing, providing replacement of a name x of a defined object onto a name y with preservation of all attributes of the x−object. If y−argument defines a name of an defined object or of an undefined name with attributes attributed to it, the procedure call is returned unevaluated. Meanwhile, the Rename procedure successfully processes also the objects x of the same name of the type {Block, Function, Module}."

Rename1::usage =
"The Rename1 procedure is a quite useful modification of the above procedure, being based on procedure Definition2. The call Rename1[x, y] is similar to the call Rename[x, y] whereas the call Rename1[x, y, z] with the third optional z−argument – an arbitrary expression – performs the same functions as the call Rename1[x, y] without change of an initial object x."

RenameH::usage =
"As against previous procedure Rename, the procedure RenameH provides in the certain measure a selective renaming of the same procedures/functions on the basis of their headings. Successful call RenameH[x, y] returns Null, i.e. nothing, renaming an object with heading x onto a name y with preservation of all attributes; in addition, the initial object with heading x is removed from the current session of the package. Moreover, the call RenameH[x, y, z] with the third optional argument z − any expression is supposed − renames object with heading x onto a name y with preservation of attributes; meanwhile, the initial object with heading x is kept in the current session of the package. In a case of erroneous situations $Failed is returned."

RenBlockFuncMod::usage =
"The call RenBlockFuncMod[x, y] in string format returns a new name of a block/function/module x determined by the following format Unique [y] <>H, where y – an arbitrary symbol, whereas H – one of symbols {\"B\", \"F\", \"M\"} depending on the type of an object x or the type of a subobject composing it in case of the object of the same name x. In addition, the object x is removed from the current session while the result of this renaming keeps options and attributes of the initial object x. Moreover, the symbol y can be definite, since its definition doesn't change at the procedure call."

RenBlockFuncMod1::usage =
"The procedure call RenBlockFuncMod1[x, y] returns a new name of a block/function/module x, where y – an arbitrary undefinite symbol; at that, y saves options and attributes of the initial object x. Furthermore, an object x can be of the same name whereas the initial symbol x becomes as an undefinite symbol."

DefAttributesH::usage =
"Procedure DefAttributesH expands the functions SetAttributes and ClearAttributes on a case of the same procedures/functions with various headings and with the any attributes ascribed to it. The call DefAttributesH [x, y, z, p, h, ...] returns Null, i.e. nothing, ascribing {y = \"Set\"} or deleting {y = \"Clear\"} for object with a heading x the attributes defined by arguments {z, p, ...}. Whereas in attempt of assignment or deletion of attributes nonexistent in the current version of system, the procedure call returns the list, whose the first element is $Failed, whereas the second element – the list of expressions, different from the current attributes."

AttributesH::usage = "AttributesH is inverse function to the above procedure DefAttributesH.
The call AttributesH[x] returns the list of attributes, ascribed to an object with heading x."

DefaultsQ::usage =

"The call DefaultsQ[x] returns True if definition of a block, function or a module x contains defaults for formal arguments, and False otherwise. Whereas the call DefaultsQ[x, y], where the second argument y – an indefinite variable – through y additionally returns the list of the used types of defaults {\"_.\", \"_:\"}."

WhatObj::usage =
"The call WhatObj[x] returns the value depending on location of an x–symbol, activated in the current session, namely: \"Undefined\" – undefined object; \"System\" – standard function of the system; \"CS\" – object, whose definition had been evaluated in the current session; \"Context`\" – context defining a package which had been activated in the current session, and which contains the definition of the x–symbol; otherwise, the procedure call is returned unevaluated."

Subs::usage =
"The procedure call Subs[x, y, z] returns the result of substitutions into an expression x of expressions z instead of occurrences in it of subexpressions y. In addition, if as x any correct expression admitted by Math–language then as a single substitution or their set coded as y ≡ {y1, y2, ..., yn} and z ≡ {z1, z2, ..., zn} are allowable as the second and third arguments determining substitutions of the format y –> z, defining a set of substitutions {y1 –> z1, y2 –> z2, ..., yn –> zn} carried out consistently."

Subs1::usage =
"A simple enough Subs1 function can be considered as a certain extension and complement of the previous Subs procedure. The function call Subs1[x, {y, z}] returns the result of replacement of all occurrences of an subexpression y of an expression x onto an expression z; at that, the function call qua of the second argument allows both the simple 2–element list, and the list of ListList–type."

Subs2::usage =
"A simple enough Subs2 function can be considered as a certain extension and complement of the previous Subs procedure and function Subs1. The function call Subs2[x, y, z] returns the result of replacement of all occurrences of an subexpression y of an expression x onto an expression z. The function admits a number of interesting enough extensions."

Subs1Q::usage = "The function call Subs1Q[x, y] returns True if a call Subs1[x, y] is allowable, and False otherwise."

Substitution::usage =
"Substitution is an integrated procedure of the above means Subs, Subs1 and Subs2. The procedure call Substitution[x, y, z] returns the result of substitutions into an arbitrary expression x of an expression z instead of occurrences in it of all subexpressions y. In addition, if as expression x any correct expression admitted by Math–language is used whereas as a single substitution or their set coded as the lists y ≡ {y1, y2, ..., yn} and z ≡ {z1, z2, ..., zn} of the same length are allowable as the second and third arguments determining substitutions of the format y –> z, defining a set of substitutions y1 –> z1, y2 –> z2, ..., yn –> zn} carried out consistently. If the lists y and z have different length the procedure call is returned unevaluated, while at absence of allowable substitutions the initial expression x is returned."

Substitution1::usage =
"The Substitution1 procedure can be considered as an analog of the above Substitution procedure. The procedure call Substitution1[x, y, z] returns the result of substitutions into an arbitrary expression x of an expression z instead of occurrences in it of all subexpressions y. In addition, if as expression x any correct expression admitted by Math–language is used whereas as a single substitution or their set coded as the lists y ≡ {y1, y2, ..., yn} and z ≡ {z1, z2, ..., zn} of the same length are allowable as the second and third arguments determining substitutions of the format y –> z, defining a set of substitutions y1 –> z1, y2 –> z2, ..., yn –> zn} carried out consistently. If the lists y and z have different length the procedure call is returned unevaluated, while at absence of allowable substitutions the initial expression x is returned."

Integrate2::usage =
"On the basis of the previous Subs1 function an useful enough procedure has been realized, whose call Integrate2[x, y] provides integrating of an arbitrary expression x on the subexpressions determined by a sequence y. At that, the procedure with the returned result by means of Simplify function performs a sequence of algebraic and other transformations and returns the simplest form it finds."

Diff1::usage =
"The procedure call Diff1[x,y,z, ...] that is also realized on the basis of the Subs1 function returns the differentiation result of an arbitrary expression x on the generalized {y,z, ...} variables which can be an arbitrary expressions. The result is returned in the simplified form on the basis of the Simplify function."

Diff2::usage =
"The procedure call Diff2[x, y, z, h, ...] that is realized on the basis of the Substitution1 function returns the differentiation result of an arbitrary expression x on the generalized {y, z, h, ...} variables which can be an arbitrary expressions. The result is returned in the simplified form on the basis of the Simplify function."

MaxNestLevel::usage = "The call MaxNestLevel[L] returns the maximal level of nesting of a list L; in addition, a list whose elements are not lists (pure list) has level of nesting 0."

MaxLevel::usage = "The call MaxLevel[L] returns the maximal level of nesting of a list L; in addition, a list whose elements are not lists (pure list) has level of nesting 0."

ListLevels::usage =
"The call ListLevels[L] returns the list of levels of nesting of a list L; in addition, if L is the empty list, i.e. {}, the list 0 is returned."

Arity::usage =
"On blocks/functions/modules with indefinite number of arguments the call Arity[x] returns \"Undefined\" value, on the system functions – \"System\" while on the objects having fixed number of the actual arguments, their quantity is returned; in other cases the call is returned unevaluated. Let's note that the Arity procedure processes the special situation \"the procedures of the same name with various headings\", returning the list of arities of subobjects composing object x. In addition, between this list and the list of definitions of subobjects that is returned by the call Definition[x] there is one–to–one correspondence. On inadmissible arguments the procedure call is returned unevaluated."

Arity1::usage = "The Arity1 procedure – a rather effective equivalent analog of the Arity procedure."

ArityBFM::usage =
"On a block/function/module with indefinite number of arguments the call ArityBFM[x] returns \"Undefined\" value, while on the objects having fixed number of the factual arguments, their quantity is returned. Let's note that the ArityBFM procedure processes the special situation \"the procedures of the same name with various headings\", returning the list of arities of subobjects composing object x. In addition, between this list and the list of definitions of subobjects that is returned by the call Definition[x] there is one–to–one correspondence. On factual arguments different from the above type the procedure call is returned unevaluated. In addition, under the \"function\" a function of classical type (i.e. with heading) is realized."

ArityBFM1::usage =
"The ArityBFM1procedure provides more detailed information on the arity of a block/function/module. The procedure call ArityBFM1[x] returns the arity (number of formal arguments) of a block/function/module x in the form of 4–element list whose the first, the second and the third elements determine amount of formal arguments with patterns \"_\", \"__\" and \"___\" accordingly. While the fourth element defines common number of formal arguments that can be different from the number of factual arguments. Thus, the x heading should has classical type, i.e. it should not include non–standard, but acceptable methods of headings definition. The unsuccessful procedure call returns $Failed."

AritySystemFunction::usage =
"The procedure call AritySystemFunction[x] generally returns the 2–element list whose first element defines number of formal arguments whereas the second element – number of admissible factual arguments. Whereas the procedure call AritySystemFunction[x, y] with the second optional argument y – an indefinite variable – through it returns the generalized template of formal arguments. The infinity symbol says about arbitrary number of factual arguments. Return of the one–element list speaks about equality of quantities of formal and the allowed actual arguments."

AritySystemFunction1::usage =
"The AritySystemFunction1 procedure is similar to the AritySystemFunction procedure, but is based on other basis. The procedure call AritySystemFunction1[x] generally returns the 2–element list whose the first and the second element define minimal and maximal number of admissible factual arguments accordingly. In case of their coincidence 1–element list is returned. Whereas the procedure call AritySystemFunction1[x, y] with the second optional argument y – an indefinite symbol – through it returns the generalized template of formal arguments. The infinity symbol says about arbitrary number of admissible factual arguments. An unsuccessful procedure call returns $False."

MessagesOut::usage =
"In some cases the processing of messages generated by the calls of blocks, functions and modules in the program mode, for example, in a module body is required. The problem is solved by the procedure whose call MessagesOut[x] returns the message generated by a G block/function/module call given in the format x = \"G[...]\". Whereas the procedure call MessagesOut[x, y] with the second optional argument – an indefinite symbol – through it returns the messages in the format {MessageName, \"Message\"}. In the absence of any messages the empty list, i.e. {}, is returned. At that, if for the user object G (block, function, module) is ascribed only usage, the usage is returned. An unsuccessful procedure call returns $Failed."

ReplaceAll1::usage =
"The call ReplaceAll1[x, y, z] returns the result of substitution of all entries of an subexpression y onto an expression z in an expression x. The procedure expands standard function ReplaceAll."

ReplaceAll2::usage =
"Unlike the Replace4 procedure, the call ReplaceAll2[x, r] returns the result of application to an expression x of a rule r (a –> b) or of consecutive application of rules from the list r; any expressions are allowed as the left parts of the rules."

ReplaceSubLists::usage =
"The call ReplaceSubLists[x, y] returns the result of replacement of elements (including adjacent elements) of a list x on the basis of a rule or list of rules y. In addition, the lists can act as parts of the rules."

ReplaceProc::usage =
"The call ReplaceProc[P, w] returns the definition in string format of a procedure – result of application to a procedure P of transformation rules w (a rule or their list); in addition, from rules w are eliminated the rules whose left parts coincide with formal arguments of the procedure P."

StringReplace1::usage =
"The call StringReplace1[S, L, P] returns the result of substitution in a string S of strings or expressions from a list P instead of its substrings, defined by positions of a nest list L of ListList−type. The procedure well supplements the standard function StringReplace."

StringReplace2::usage =
"The call StringReplace2[S, s, Exp] returns the result of replacement of all occurrences in a string S of its substrings s onto an expression Exp; in addition, the replaced substrings s should not be limited by letters. If the string S does not contain occurrences of s, the call returns the initial string S whereas on empty string S the empty string is returned."

StringReplace3::usage =
"The call StringReplace3[S, x, x1, y, y1, z, z1, ...] returns the result of replacement of all occurrences in a string S of its substrings {x, y, z, ...} onto strings {x1, y1, z1, ...} accordingly; in addition, the replaced substrings {x, y, z, ...} should not be limited by letters. If the string S does not contain occurrences of {x, y, z, ...}, the call returns the initial string S whereas on empty string S the empty string is returned."

StringReplaceS::usage =
"The call StringReplaceS[S, s1, s2] returns the result of replacement of all occurrences in a string S of its substrings s1 that are limited from the left and the right by strings \"x\" (StringLength[\"x\"] = 1) from the given lists L and R accordingly, onto substrings s2. If the string S does not contain of occurrences s1, the procedure call returns the source string S."

StringReplacePart1::usage =
"Unlike the standard StringReplacePart function the StringReplacePart1 procedure provides replacements in strings not on the basis of positions of the replaced substrings, but according to ordinal numbers of their occurrences. The procedure call StringReplacePart1[x, y, z, p] returns the result of replacement in a string x of substrings y onto a string z according to ordinal number(s) of occurrences of substrings y into the string x which are defined by an integer or an integer list p. In case of impossibility of such replacement, for example, the list p contains numbers that are different from order numbers of occurrences of y in x, the call returns the initial string x. The procedure ignores inadmissible elements of the list p, without initiating an erroneous situation."

Replace1::usage =
"The call Replace1[x, r] returns the result of applying of a rule r or their list to undefined variables of an expression x (to all or selectively). In case of detection by procedure of empty rules from r, the message with hint of the list of rules which appeared empty, i.e. whose left parts do not belong to the list of independent variables of an expression x is printed."

Replace2::usage =
"The call Replace2[x, y, z] returns the result of replacement of all occurences of a subexpression y in an expression x onto an expression z. In case of impossibility to do such replacement the call returns $Failed."

Replace3::usage =
"The call Replace3[x, y, z] returns the result of replacement of all occurences of subexpressions y in an expression x onto expressions z. In addition, separate expressions or their lists can act as arguments {y, z}. Thus, in case of arguments {y, z} in the form of the list for them is chosen the common length determined by relation Min[Map[Length, {y, z}]], allowing to avoid the possible special and erroneous situations which demand the program processing, but with print of the relevant diagnostic information."

Replace4::usage =
"The call Replace4[x, a -> b] returns the result of application to an expression x of substitution a -> b on condition that its left part a is an arbitrary expression. At absence in expression x of the occurrences of subexpression a initial expression x is returned."

BlockQ::usage = "The call BlockQ[x] returns the True if x is a Block and the False otherwise."

BlockQ1::usage = "The call BlockQ1[x] returns the True if x is a Block and the False otherwise."

DefFunc::usage =
"The call DefFunc[x] in a compact format returns the definition of an x−object contained in a package or a notebook loaded into the current session. The object name is given in string−format or Symbol−format depending on the object type (procedure, function, global variable, procedure variable etc.)."

DefFunc3::usage =
"The call DefFunc3[x] returns the list of definitions, given in string format, of x−procedures of the same name with the various headings which were activated in the current session of the package; In addition, the order of definitions of the returned list corresponds to the order of their output by standard function Definition of the package."

DefOpt::usage =
"The function Definition[x] in a lot of cases returns definition of a x−object together with a context corresponding to it that at enough large definitions becomes poorly foreseeable and less acceptable for the subsequent program processing. On the other hand, our procedures DefFunc, DefFunc1, DefFunc2, DefFunc3 also appear unsuitable, if necessary to obtain the definitions of some procedural variables, in particular, $TypeProc. And only the call of procedure DefOpt[x] returns

definition of a x−object in an optimum format irrespective of type of the user object. On system functions and other string expressions the call DefOpt[x] returns \"Null\". In addition, the call DefOpt[x] not only returns the optimum format of definition of an object x, but also evaluates it in the current session what in some cases is rather useful; furthermore, the name of an object x is coded in string−format. At the same time it must be kept in mind that procedure DefOpt is useless for the procedures/functions of the same name, i.e. having several definitions with different headings."

DefOpt1::usage =
"The DefOpt1 procedure is seemed as effective enough for receiving of definition of the user procedure/function in optimized format in the above sense, i.e. without context. The procedure call DefOpt1[x] on a system function x returns its name, on an user function or procedure returns its optimized code in string format whereas on other values of argument x $Failed is returned."

OptDefinition::usage = "The call of procedure OptDefinition[x] returns the definition of a procedure or function x in an optimum format, i.e without context."

DefOptimum::usage =
"The procedure call DefOptimum[x] returns the definition of a procedure or function x in an optimum format, i.e without context associated with a package containing x. This definition becomes active in the current session."

OptDefPackage::usage =
"The procedure call OptDefPackage[x] returns the 2−element list whose first element − the list with names of means of the user package with context x, uploaded into the current session whose definitions received by the Definition function are optimized in the above sense while the second element − the list with names of means with a context x whose definitions aren't optimized in the above sense. In turn, the procedure call OptDefPackage[x, y] with the second optional y argument − an arbitrary expression − returns the above 2−element list, in addition converting the means of the second sublist into means with the optimized format in the current session. The procedure call on an inactive context x returns $Failed."

SymbolsFromMx::usage =
"A rather useful SymbolsFromMx procedure that allows to check a mx−file with the user package for entering of symbols in it without uploading of the datafile into the current session. The procedure call SymbolsFromMx[x] returns the three−element list whose first element − the list with names of means of the user package, contained in a mx−file x, whose definitions received by the system Definition function are not optimized in the above sense, while the second element represents the list with names of means with the context ascribed to the file x; at last, the third element represents the list with names of system means used by the user package located in the x file. In turn, the procedure call SymbolsFromMx[x, y] with the second optional y argument − an indefinite symbol − additionally returns the 2−element list whose the first element − the list with names of means of the user package, located in a mx−file x, whose definitions received by the system Definition function are not optimized in the above sense, while the second element represents the list with names of means of the user package x, whose definitions obtained by the Definition function are optimized in the above sense considered in detail in our book [8]."

OptimizedDefPackage::usage =
"The OptimizedDefPackage procedure allows to check the user package located in a mx−file x for existing in it of means whose definitions in case of uploading of the file in the current session are optimized in the above sense. The procedure call OptimizedDefPackage[x] returns the two−element list whose the first element − the list with names of means of the user package, located in a mx−file x whose definitions are optimized in the above sense, while the second element represents the list with names of means that are not optimized. At the same time, the procedure call converts all means of a package x whose definitions will be optimized in the current session. The procedure can be applied both to the unloaded and the uploaded mx−file. At that, if the mx−file is unloaded, than this file remains as unloaded."

OptimizedMfile::usage =
"The procedure call OptimizedMfile[x, y] ensures the saving of tools definitions with x context that are optimized in the above sense in a m−file y with returning y. In turn, the call OptimizedMfile[x, y, z] additionally through the third argument z − an indefinite symbol − returns the 3−element list whose the first element defines list of tools whose definitions are optimized, the second element defines list of tools whose definitions are not optimized, and the third element defines the list of temporary symbols which have been generated at upload of the user package with context x into the current session."

DefOp::usage =
"The call DefOp[w] in string−format returns the type of the assignment applied to a name x encoded in string−format too: (1) \"Undefined\" − the name w is not defined, (2) $Failed − a name x defines object different from a simple variable, (3) \"=\" − to the name w immediate assignment, and (4) \":=\" − to a name w the lazy assignment has been applied. Whereas the call DefOp[w,y] through optional second argument − the undefined variable y − is returned an expression assigned to the symbol w."

SuffPref::usage =
"The call SuffPref[S, s, N] returns the True if N=1 and a substring s or a string from the list s begins a string S, the True if N=2 and a substring s or a string from the list s ends a string S, and returns the True if N=3 and a substring s or a string from the list s limits a string S from both ends; otherwise the False is returned."

Gather1::usage =
"The call Gather1[L, n] returns the nested list formed on the basis of a list L of ListList−type by means of grouping of sublists

of L by its n−th element. The procedure is an extension of the standard function Gather of the package."

Gather2::usage =
"Similarly to procedure Gather1 the procedure Gather2 − one more modification of standard procedure Gather. The call Gather2[L] returns either the simple list, or the list of ListList−type that defines only multiple elements of a list L with their multiplicities. At absence of multiple elements in L the call of procedure returns the empty list, i.e. {}."

GatherStrLetters::usage = "The call GatherStrLetters[x] of a rather simple function returns the result of gathering of the letters of a string x into substrings of identical letters."

CharacterQ::usage =
"The call CharacterQ[x] returns the True if x is a symbol with character code from range 0..255, and the False otherwise."

Characters1::usage = "The Characters1 extends the standard function Characters. The call Characters1[x] gives the list of characters in a string x//ToString."

HowAct::usage =
"The call HowAct[Q] returns the value True if Q is an object active in the current session, and the False otherwise. In many cases the procedure HowAct is more suitable than standard function ValueQ, including local variables in procedures."

UndefinedQ::usage =
"The procedure call UndefinedQ[x] returns True if x is a definite symbol, and False otherwise. In addition, unlike the HowAct function the UndefinedQ procedure returns True only on condition of empty definition for the tested symbol irrespective of existence for it of options and attributes. The given distinction is a rather essential, in particular, in case of programming of procedures at which as formal arguments undefinite variables through which additional results of procedures calls are returned are used. In this case it is necessary to use the format \"y_ /; ! HowAct[y]\" in the procedures headings."

DefinedActSymbols::usage = "The function call DefinedActSymbols[] returns the list of the names of symbols different from temporary and undefined symbols in the current session."

SubsPosSymb::usage =
"The call SubsPosSymb[x, n, y, z] returns the substring of a string x which is limited from the right (from the left) by a position n and from the left (from the right) by characters from a list y. In addition, the search in the string x is done from left to right (z = 1) and from right to the left (z = 0) starting with position n. The procedure call on inadmissible actual arguments is returned unevaluated."

Locals::usage =
"The call Locals[x] returns the list whose elements in string−format represent the local variables together with their initial values of a block or a module x. While the call Locals[x, y] with the second optional argument y – an indefinite variable – provides the return through y in addition or the simple 2−element list, or the ListList−type list with 2−element sublists, whose first elements define names of local variables of the block/module x in string−format, whereas the second element – the initial values ascribed to them in string−format; lack of the initial values is defined by symbol \"No\". If the object x has no local variables,the call Locals[x,y] returns the empty list,i.e.{},the same result is returned through the second argument. Moreover,on a typical function the call of the procedure returns \"Function\" value."

PosIntQ::usage = "The call PosIntQ[n] returns True if n − a positive number; otherwise, False is returned."

PosIntListQ::usage = "The call PosIntListQ[W] returns True if W − a list of positive numbers; otherwise, False is returned."

MapNestList::usage =
"The procedure call MapNestList[x, f, y] returns result of application to all nesting levels of a list x of a function f with transfer of the arguments determined by optional argument y to it."

StrSub::usage =
"The call StrSub[x, y, z, n] returns the result of replacement in a string x of occurrences of a substring y onto a substring z according to numbers n of such occurrences defined by either a positive number or their list. In case of absence of occurrences y into string x or inadmissible value n the call of procedure StrSub returns value $Failed."

Locals1::usage =
"The call Locals1[P] returns the list of names of local variables in string−format of a block or a module P, if local variables are absent for P, the call returns the empty list, i.e. {}. In addition, in case of an object P of the same name, the call returns the nested list whose elements are bijective relative to the elements of list PureDefinition[x]."

Locals2::usage =
"The call Locals2[P] returns the list of names of local variables in string−format without initial values ascribed to them of a block or a module P, if local variables are absent for P, the call returns the empty list, i.e. {}. In addition, in case of an object P of the same name, the call returns the nested list whose elements are bijective relative to the elements of list PureDefinition[x]. If type of P is different from {Block, Module}, the call Locals2[P] is returned unevaluated."

Locals3::usage =
"The procedure Locals3 essentially uses our procedure StringToList, and its call Locals3[x] returns the list whose elements in string format represent the local variables of a block/module/function x and/or 2–element lists of such variables whose the first element – a local variable while the second element – its initial value if it exists, of course. Whereas, on typical functions x the procedure call Locals3[x] returns \"Function\"."

Globals::usage =
"The procedure call Globals[P] returns the list of global variables in string–format of a procedure P. If a procedure P has no global variables, the procedure call Globals[P] returns the empty list, i.e. {}. It is necessary to note, the procedure Globals[P] as a global variable understands the name of an object in the body of a procedure P for which assignment by operators {`:=`, `=`} is made and which is distinct from local variables of the main procedure P. Therefore the situation when the local variable of a subprocedure in procedure P can be determined by procedure Globals[P] as a global variable is quite probable."

Globals1::usage =
"The procedure call Globals1[P] returns the list of global variables in string–format of a procedure P. If a procedure P has no global variables, the procedure call Globals1[P] returns the empty list, i.e. {}. In addition, the procedure is extension of procedure Globals onto case of an procedure P of arbitrary structure, allowing subprocedures of any level of nesting in own body."

Globals2::usage =
"The simple enough function Globals2 distributes action of the procedures Globals and Globals1 onto procedures of an arbitrary type; the call Globals2[x] returns the list of names in string–format of global variables of a procedure x. If a procedure x has no global variables, the call Globals2[x] returns the empty list, i.e. {}."

VarsInBlockMod::usage =
"The procedure call VarsInBlockMod[x] returns the 4–element list, whose first element defines the list of formal arguments, the second element defines the list of local variables, the third element defines the list of global variables and the fourth element defines the list of the variables other than system variables and names."

Spos::usage = "The call Spos[x, y, p, d] returns position number of the first occurence of an one–letter string y into a string x to the left (d=0) or to the right (d=1) from the given position p."

SortNL::usage =
"The call SortNL[L, p, b] sorts a listlist L on elements in a p–position of its sublists; b = {Greater|Less}. On inadmissible second argument p the corresponding message is printed; whereas on inadmissible first or third argument the function call is returned unevaluated."

PrefixQ::usage = "The call PrefixQ[x, y] returns True if x is prefix of a string y, and False otherwise."

SuffixQ::usage = "The call SuffixQ[x, y] returns True if x is suffix of a string y, and False otherwise."

SortNL1::usage =
"The call SortNL1[L, p, b] returns the result of sorting of a ListList L on elements in a p–position of its sublists on the basis of their unique decimal codes defined by the procedure GC; b={Greater|Less}. In a certain sense the procedure SortNL1 extends the procedure SortNL."

SortNestList::usage =
"The SortNestList[x, p, y] returns the result of sorting of a nested numeric or symbolic list x on elements in a p–position of its sublists on the basis of sorting function defined by a comparison y; y={Greater|Less} for the numeric lists, and y={SymbolGreater|SymbolLess} for the symbolic lists. In addition, a nested list with the nesting level 1 is supposed as argument x; otherwise, the call returns the initial list x."

SortLpos::usage =
"The call SortLpos[L, n, SF] sorts a nested L on elements in a n–position of its sublists; SF={Greater|Less}. In addition, the nested list L can be different from list of ListList–type. The SortLpos procedure programmatically processes the main arising special and wrong situations, by returning on them $Failed."

SortString::usage = "The call SortString[x, F] sorts a string x symbol–by–symbol according to a sorting function F={Greater|Less}."

D1::usage = "The call D1[x, y] returns the result of differentiation of an expression x by a subexpression y. In case of impossibility to do such differentiation the call returns $Failed."

Df::usage = "The call Df[x, y] returns the result of differentiation of x on y. The procedure expands standard function D."

Df1::usage =
"The call Df1[x, y] returns the result of differentiation of x on y. The procedure expands standard function D. The procedure is an useful modification of the procedure Df and is more effective than procedure Df in a series of cases."

Df2::usage =
"The call Df2[x, y] returns the result of differentiation of x on y. The procedure expands standard function D. The procedure

is an useful modification of the procedure Df and is more effective than procedure Df in a series of cases."

Difff::usage =
"The call Diff[x, y, z, ...] returns the differentiation result of an expression x on the generalized {y, z, ...} variables which can be an arbitrary expressions. The result is returned in the simplified form on the basis of the Simplify function."

Diff::usage =
"The call Diff[x, y, z, ...] returns the differentiation result of an expression x on the generalized {y, z, ...} variables which can be an arbitrary expressions. The result is returned in the simplified form on the basis of the Simplify function."

Int::usage = "The call Int[x, y] returns the result of integration of x on y. The procedure expands standard function Integrate."

Int1::usage =
"The call Int1[x, y] returns the result of integration of x on y. The procedure expands standard function Integrate. The procedure is an useful modification of the procedure Int and is more effective than procedure Int in a series of cases."

Integrate1::usage = "The call Integrate1[x, y] returns the result of integration of an expression x by a subexpression y. In case of impossibility to do such integration the call returns $Failed."

Integral1::usage =
"The call Integral1[x, y, z, ...] returns the result of integration of an expression x on the generalized {y, z, ...} variables which can be an arbitrary expressions. The result is returned in the simplified form on the basis of the Simplify function."

Integral2::usage =
"The call IntegraL[x, y, z, ...] returns the integration result of an expression x on the generalized {y, z, ...} variables which can be an arbitrary expressions. The result is returned in the simplified form on the basis of the Simplify function."

NamesProc::usage =
"The call NamesProc[] returns the list of names of the user blocks, functions and modules directly activated in the current session."

FunctionQ::usage =
"The call FunctionQ[x] returns True, if x is a function of any type (traditional or pure), and False otherwise. In addition, the name x of an object can be coded both in symbolical, and in string formats; in the second case the correct testing of the object x is provided, allowing multiplicity of definitions, i.e. object x can be of the same name in the sense stated above."

QFunction::usage =
"The call QFunction[x] returns True on a traditional function x and on x objects generated by the Compile function along with pure functions, and False otherwise. In addition, as the traditional function the function determined by construction of the format J[x_, y_, ...] {:=|=} W(x, y, ...) is understood."

QFunction1::usage =
"The call QFunction1[x] returns True on a traditional function x and on x objects, generated by the Compile function, and False otherwise; moreover, on an object of the same name x the True value is returned only if all its components are traditional functions and/or are generated by the Compile function. The call QFunction1[x] assumes coding of actual argument x in string−format. In addition, as the traditional function the function determined by construction of the format J[x_, y_, ...] {:=|=} W(x, y, ...) is understood."

PureFuncQ::usage = "The call PureFuncQ[F] returns True if F is a pure function, and False otherwise."

Mapp::usage =
"The call Mapp[F, L, g] generalizes standard function Map on number of arguments the greater than 2. For example, Mapp[a, {b, c, d}, x, y, z] == {a[b, x, y, z], a[c, x, y, z], a[d, x, y, z]}; Mapp[StringReplace, {\"812345265\", \"72345957\"}, {\"2\" −> \"V\", \"5\" −> \"G\"}] == {\"81V34GV6G\", \"7V34G9G7\"}. In addition, the user procedure/function, a standard function or a symbol F can be as a factual argument F."

Mapp1::usage = "The call Mapp1[x, y] unlike the call Map[x,y] of the standard function returns the result of application of a procedure/function x to all elements of a list y, regardless of their membership to the list levels."

FunCompose::usage =
"The call FunComposep[L, x] returns the nested function from the given list of functions L from a variable (expression) x."

PackNames::usage =
"The PackNames procedure solves the problem of obtaining of the names list of the objects whose definitions with usages are in a package, being in a datafile of the format {\"m\", \"nb\"}. In addition, it is supposed that package uploading into the current session isn't obligatory. The given problem is solved by a quite useful procedure, whose call PackNames[x] returns the names list of the above objects in a package, being in a datafile x of the format {\"m\", \"nb\"}."

PackNames1::usage =
"The PackNames1 procedure solves the problem of obtaining of the names list of the objects whose definitions with

usages are in a package, being in a datafile of the mx–format. In addition, it is supposed that package loading into the current session isn't obligatory. The given problem is solved by a quite useful procedure, whose call PackNames1[x] returns the names list of the above objects in a package, being in a datafile x of the mx–format whose definition are associated with the context of a mx–file x, for example, by means of Definition function."

PackNames2::usage =
"The PackNames2 procedure using Shortest function, whose call Shortest[h] is a pattern object that matches the shortest sequence consistent with a pattern h, solves the problem of obtaining of the names list of objects whose definitions are in a package, being in a file of the nb–format. At that, it is supposed that package loading into the current session isn't obligatory. The given problem is solved by means of a quite useful procedure, whose call PackNames2[x] returns the names list of the above objects in a package, being in a file x of the nb–format."

Closes::usage = "The call Closes[x] closes a datafile x including the closed and null files without printing of erroneous messages."

RenDirFile::usage =
"The procedure RenDirFile is an extension of standard functions RenameFile and RenameDirectory. Procedure RenDirFile[x, y] provides renaming of an element x (directory or datafile) in situ irrespective of its type and attributes with preservation of its type and all its attributes; at that, as argument y a new name of the element x is used. Therefore the successful procedure call RenDirFile[x, y] returns the full path to a renamed element x. In the case of existence of an element y the message \"Directory/datafile <y> already exists\" is returned. In other unsuccessful cases the procedure call returns the $Failed or is returned unevaluated."

DelEl::usage =
"The call DelEl[L, w, N] returns a list L truncated from the left by elements w for N = 1, a list L truncated from the right by elements w for N = 2, and for N = 3 truncated from both ends; at other values N the call is returned unevaluated."

StrDelEnds::usage =
"The call StrDelEnds[S, h, p] returns a string S truncated from the left by symbols h for p = 1, a string S truncated from the right by symbols h for p = 2, and for p = 3 truncated from both ends; at other values p the procedure call is returned unevaluated."

StrDelEnds1::usage =
"The call StrDelEnds1[S, h, p] returns a string S truncated from the left by symbols h for p = 1, a string S truncated from the right by symbols h for p = 2, and for p = 3 truncated from both ends; at other values p the procedure call is returned unevaluated."

StreamsU::usage =
"The call StreamsU[] returns the list containing Stream–objects active in the current session excluding system streams."

CloseAll::usage = "The call CloseAll[] closes all streams different from system streams and returns list of the closed files."

Close1::usage =
"The Close1 procedure generalizes the standard Close function. The call Close1[x, y, z, ...] closes all existing datafiles from the list {x, y, z, ...} irrespective of quantity of streams on which they were open by various qualifiers, returning their list. In other cases the call on admissible actual arguments returns the empty list, i.e. { } whereas on inadmissible actual arguments the call is returned unevaluated."

Close2::usage =
"The procedure Close2 is a functional analog of the above procedure Close1. The procedure call Close2[x, y, z, ...] closes all off really–existing datafiles in a list {x, y, z, ...} irrespective of quantity of streams on which they have been opened by various files qualifiers with returning their list. In other cases the call on admissible actual arguments returns the empty list, i.e. {} whereas on inadmissible actual arguments a call is returned unevaluated."

MixCaseQ::usage = "The call MixCaseQ[x] returns the True if string x contains letters on different registers and the False otherwise. If x contains special symbols only the \"Special Characters\" is returned."

FileOpenQ::usage =
"The call FileOpenQ[F] returns the list {{R, F, Channel}, ...}, if file F is open for reading/writing (R = {\"read\"|\"write\"}), defines actually file F in the stylized format (LowerCase + all occurrences \"\\\" are replaced on \"/\") and element Channel defines the logic channel per which the file F was open; if file F is closed False is returned, if file F is absent $Failed is returned."

FileOpenQ1::usage =
"Procedure FileOpenQ1 is an enough useful extension of the procedure FileOpenQ. The procedure call FileOpenQ1[F] returns the nested list of the format {{R,x,y,...,z}, {{R,x1,y1,...,z1}} if a datafile F is open for reading or writing (R = {\"in\" | \"out\"}), and F defines the datafile in any format (Register + \"/\" and/or \"\\\"); if the datafile F is closed or is absent, the empty list is returned, i.e. {}. Moreover, sublists {x, y, ..., z} and {x1, y1, ..., z1} define datafiles or full paths to them that are open for reading and writing respectively. Moreover, if in the current session all user datafiles are closed, except system files, the call FileOPenQ1[x] on an arbitrary string x returns $Failed. The datafiles and paths to them are returned in formats which are defined in the list returned by the function call Streams[], irrespective of the format of the datafile F."
InOutFiles::usage = "The call InOutFiles[] returns 2–element list whose the first element defines the list of datafiles opened for reading while the second element defines the list of datafiles

opened for writing; in the absence of the user's open datafiles the empty list is returned."

OpenFileQ::usage =
"The call OpenFileQ[x] returns True if a datafile x accurate within standard format coincides with a datafile from the list InOutFiles[], otherwise False is returned; in addition, the call OpenFileQ[x, y] through an undefined symbol y returns the 2−element list with all such datafiles in case of the main result True. The first sublist defines the datafiles opened for reading whereas the second sublist defines the datafiles opened for writing."

Need::usage =
"The call Need[x] loads a package which corresponds to a x−context into the current session provided that the corresponding datafile of the format {\".m\"|\".mx\"} is in one of the catalogs determined by the system variable $Path, with True return; otherwise the call returns value $Failed. While the call Need[x, y] loads a package which corresponds to a x−context into the current session provided that the corresponding datafile of the format {\".m\"|\".mx\"} is or in one of the subdirectories determined by the system variable $Path, or is defined by argument y, with True return; otherwise value $Failed is returned."

UpdatePath::usage = "The call UpdatePath[x] expands a list of directories defined by the global variable $Path onto directories defined by a list or a string x."

UpdatePackages::usage =
"The call UpdatePackages[P] expands a list of packages defined by the global variable $Packages onto a package P or their list. Argument P defines the context of a package P, or the list of contexts of appropriate packages. The procedure call returns Null, i.e. nothing."

UpdateContextPaths::usage =
"The call UpdateContextPaths[P] expands a list of paths defined by the global variable $ContextPaths onto a context P or their list. The procedure call returns Null, i.e. nothing."

Adrive::usage =
"The call Adrive[] returns the list of names of all active devices of external memory in string−format in the current session."

Adrive1::usage =
"Procedure Adrive1 extends the procedure Adrive. The call Adrive1[] returns the 2−element nested list, whose first sublist contains names of all active devices of external memory in string−format whereas the second sublist contains names of all inactive devices of external memory."

SetDir::usage =
"The procedure call SetDir[x] on an existing subdirectory x does it current while a nonexistent subdirectory is previously created and then it is defined as the current subdirectory. At that, if the factual x−argument at the call SetDir[x] is determined by a chain without name of the IO device, for example, \"aa\\\\...\\\\bb\", then a chain of the subdirectories Directory[] <> \"aa\\\\...\\\\bb\" is created that determines a full path to the created current subdirectory. The unsuccessful procedure call returns $Failed, for example, at an inactive IO device."

SetDir1::usage =
"The SetDir1 – an extension of the procedure SetDir. In case of lack of the device for the given chain of directories w the call SetDir1[w] returns the created chain of subdirectories on the device having the greatest possible volume of available memory among all active devices of direct access in the current session."

Nobj::usage =
"The call of procedure Nobj[x, y] returns the list of names in string−format of objects, saved earlier in a file x by means of function Save whereas through the second actual argument y the list of headings in string−format of the given objects is returned. Such decision is essential enough since in a file x can be the same objects with various headings which identify uniqueness of an object."

Aobj::usage =
"The procedure call Aobj[w, y] activates in the current session all objects with a name y from a m−file w, which had been earlier created by means of the chain of functions GUI: `File –> Save As –> Mathematica Package (∗.m)`, returning Null. Moreover, as the second argument y for the Aobj procedure a single symbol or their list can act. Besides that is supposed, before preservation in the m−file x all definitions of objects in the current document were calculated in separate Input−paragraphs and should have headings. The successful call of the Aobj procedure returns nothing with printing of the message concerning means which were loaded from the m−file x or which are absent in the datafile x."

Aobj1::usage =
"The Aobj1 procedure is a rather useful extension of the previous Aobj procedure. Like the Aobj procedure the Aobj1 procedure also is used for activation in the current session of the objects which are in a m−datafile which is earlier created by means of chain `File –> Save As –> Mathematica Package (∗.m)` of the GUI commands. The successful call Aobj1[x, y] returns Null, i.e. nothing with output of the messages concerning those means that were uploaded from a m−file x and that are absent in the datafile. Moreover, as the second argument y at the procedure call Aobj1 the separate symbol or their list can be. Besides that is supposed, before saving in a m−datafile x all definitions of objects in the saved document should be evaluated in separate Input−paragraphs on the basis of delayed assignments however existence of headings not required. Right there it should

be noted that for ability of correct processing of the m–files created in the specified manner the predetermined $AobjNobj variable is used, that provides correct processing of the datafiles containing the procedures, in particular, Aobj and Aobj1."

Map1::usage = "The call Map1[{F, G, H,...}, {x, y, z,...}] returns the list of the format {F[x, y, z,...], G[x, y, z,...], H[x, y, z,...],...}."
Map2::usage = "The call Map2[F, {a, b, c,...}, {x, y, z,...}] returns the list of the format {F[a, x, y, z,...], F[b, x, y, z,...], F[c, x, y, z,...],...}."
Map3::usage = "The call Map3[F, H, {x, y, z, h,...}] returns the list of the format {F[H, x], F[H, y], F[H, z], F[H, h],...}."
Map4::usage = "The call Map4[F, {x, y, z, h,...}, H] returns the list of the format {F[x,H], F[y, H], F[z, H], F[h, H],...}."
Map5::usage =
"The call Map5[F, {{x1, y1, ...}, {x2, y2, ...}, {x3, y3, ...}, ...}] returns the list of the format {F[x1, y1, ...], F[x2, y2, ...], F[x3, y3, ...], ...}."
Map6::usage = "The call Map6[F, {{x1, y1, ...}, {x2, y2, ...}, {x3, y3, ...}, ...}] returns the list of the format {F[x1, y1, ...], F[x2, y2, ...], F[x3, y3, ...], ...}, if a function F is presented in short form <#1,#2, ..., #3,...> &."

Map7::usage = "The call Map7[F, G, H, ..., V, {a, b, c, ..., v}], where F, G, H, ..., V – the arbitrary symbols and {a, b, c, ..., v} – the list of arbitrary expressions, returns the list of the format {F[G[H[... V[a]]]] ...], F[G[H[... V[b]]]] ...], F[G[H[... V[c]]]] ...], ..., F[G[H[... V[v]]]] ...]}."

Map8::usage =
"The call Map8[F, G, H, ..., V, {a, b, c, ..., v}], where F, G, H, ..., V – symbols and {a, b, c, ..., v} – the list of arbitrary expressions, returns the list of the format {F[a, b, c, ..., v], G[a, b, c, ..., v], H[a, b, c, ..., v], ..., V[a, b, c, ..., v]}."

Map9::usage = "The call Map9[F, {a, b, c, ..., v}, {a1, b1, c1, ..., v1}], where F – a symbol and {a, b, c, ..., v}, {a1, b1, c1, ..., v1} – the lists of arbitrary expressions, returns the list of the format {F[a, a1], F[b, b1], F[c, c1], ..., F[v, v1]}."

Map10::usage =
"The call Map10[[F, x, {a, b, ..., v}, c1, c2, ..., cn], where F – a symbol whereas x and {a, b, c, ..., v} – an arbitrary expression and the list of arbitrary expressions accordingly, and c1, c2, ..., cn – optional arguments – returns the list of the format {F[x, a, c1, c2, ...], F[x, b, c1, c2, ...], F[x, c, c1, c2, ...], ..., F[x, v, c1, c2, ...]}."

Map11::usage =
"The call Map11[F, {{x, y, z, ...}, {a, b, c, ...}, ...}, h], where F – a symbol whereas {x, y, z, ...}, {a, b, c, ...}, ... and h – arbitrary expressions returns the list of the format {{F[x, h], F[y, h], F[z, h], ...}, {F[a, h], F[b, h], F[c, h], ...}, ...}."

Map12::usage =
"The call Map12[F, {{a, b, c, ..., v}, {a1, b1, c1, ..., v1}, ..., p,...,{ap, bp, cp, ..., vp}}], where F – an arbitrary symbol, whereas the second factual argument – the nested list of any nesting of arbitrary expressions accordingly – returns the list of the format {Map[F, {a, b, c, ..., v}], Map[F, {a1, b1, c1, ..., v1}], ..., F[p] ,..., Map[F, {ap, bp, cp, ..., vp}]}."

Map13::usage =
"The call Map13[F, {{a, b, c, ..., v}, {a1, b1, c1, ..., v1}, ..., {ap, bp, cp, ..., vp}}], where F – an arbitrary symbol, whereas the second factual argument – the nested list of ListList–type of arbitrary expressions accordingly – returns the list of the format {F[a, a1, a2, ..., ap], F[b, b1, b2, ..., bp], F[c, c1, c2, ..., cp], ..., F[v, v1, v2, ..., vp]}."

Map14::usage =
"The call Map14[F, {a, b, c, ..., v}, y], where F – an arbitrary symbol, whereas the second factual argument – the list of arbitrary expressions, and y is an arbitrary expression – returns the list of the format {F[a, y], F[b, y], F[c, y], F[d, y], ..., F[v, y]}. In addition, use in call Map14[F, {a, b, c, ..., v}, y, t] of the optional fourth actual argument – any expression – returns result of the following format, namely: {\"F[a, y]\", \"F[b, y]\", \"F[c, y]\", \"F[d, y]\",..., \"F[v, y]\"}."

Map15::usage = "The call Map15[F1, F2, ..., Fp, W], where Fj – the arbitrary symbols (j=1..p), whereas W – an arbitrary expression returns the list of the format F1[t], F2[t], F3[t], F4[t], ..., Fp[t]}."

Map16::usage =
"The call Map16[F, {a, b, ..., v}, c1, c2, ..., cn], where F – a symbol whereas {a, b, c, ..., v} – the list of arbitrary expressions, and c1, c2, ..., cn – optional arguments accordingly – returns the list of the format {F[a, c1, c2, ...], F[b, c1, c2, ...], F[c, c1, c2, ...], ..., F[v, c1, c2, ...]}."

Map17::usage =
"The call Map17[F, {a –> b, c –> d, ...}], where F – a symbol whereas {a –> b, c –> d, ...} – the list of rules returns the list of the format {F[a] –> F[b], F[c] –> F[d], ...} without demanding any additional explanations in view of its transparency."

Map18::usage = "The call Map18[x, y], where x – the list {x1, x2, ..., xn} of symbols and y – the list {y1, y2, ..., yp} of any expressions, returns the nested list of the following format, namely:

{{x1[y1], x1[y2], ..., x1[yp]}, {x2[y1], x2[y2],..., x2[yp]}, ..., {xn[y1], xn[y2], ..., xn[yp]}}

The result returned by the function call Map18[x, y] is transparent enough and of special explanations doesn't demand."

Map19::usage =
"The call Map19[x, y], where x – the list {x1, x2, ..., xj, ..., xp} of symbols and y – the list {y1, y2, ..., yj, ..., yp} of arbitrary expressions,

returns the nested list of the following format, namely: {x1[y1], x2[y2], ..., xj[yj], ..., xp[yp]} (j = 1..p). If yj = {a, b, c, ...} then xj[a, b, c, ...]. The result returned by the function call Map19[x, y] is transparent enough and of special explanations doesn't demand."

Map20::usage =
"The call Map20[x, y], where x – the list {x1, x2, ..., xj, ..., xp} of symbols and y – the nested list {{y1}, {y2}, ..., {yj, ..., yjn}, ..., {yp}} of NestList–type of arbitrary expressions, returns the nested list of the following format, namely: {x1[y1], x2[y2], ..., xj[yj, ..., yjn], ..., xp[yp]} (j = 1..p). If yj = {a, {b}, c, ...} then xj[a, {b}, c, ...]. The result returned by the function call Map20[x, y] is transparent enough and of special explanations doesn't demand."

CDir::usage = "The call CDir[d] creates a chain of subdirectories defined by string d; in addition, if drive does not exist the first drive is chosen from list of existing active drivers that has maximal size of free space."

FileFormat1::usage =
"The procedure call FileFormat1[x] returns the simple or nested list, first element of a simple list defines the full path to a datafile x while the second element – its format that is recognized by the FileFormat function; at that, the required datafile can be located in any directory of file system of the computer; absence of a datafile x initiates the return of the empty list, i.e. {}. Moreover, at finding several datafiles with an identical name the nested list whose sublists have the specified format is returned."

FileFormat2::usage =
"The procedure FileFormat2 is an extension of the procedure FileFormat1 in case of the root directories of direct–access volumes. The call of procedure FileFormat2[x] returns the format of a datafile x, located in any catalogue of file system of a computer, similarly to standard function FileFormat; in addition, absence of the datafile x initiates return of the empty list; otherwise, the simple or nested list is returned, at that the first element of such list defines full path to the datafile x whereas the second – its format, recognizable by function FileFormat. Moreover, at finding of several files with identical name x, the nested list whose sublists have the specified format is returned. If x – the root directory of a direct–access volume the \"Directory\" is returned."

FileFormat3::usage =
"The call FileFormat3[x] returns the type of a datafile which has been set by a name or a qualifier x. In addition, if the file has an expansion, it will be the type of datafile x. Whereas the call FileFormat3[x, y] with the second optional argument – an expression y – in case of the datafile without expansion of the name returns the full name of the file with the extension defined for it, at the same time renaming the file x to the evaluated format. In a certain relation the Format3 procedure complements the procedures Format1 and Format2."

LoadExtProg::usage =
"The successful call LoadExtProg[x] provides search in file system of the computer of a program x given by the full name with its subsequent copying into the subdirectory defined by the call Directory[]. The successful procedure call LoadExtProg[x] searches out a datafile x in file system of the computer and copies it into the directory defined by the call Directory[], returning Directory[]<>\"\\\"<>x if the datafile already was in this subdirectory or has been copied into this directory. In addition the first directory containing the found datafile x supplements the list of the directories determined by the predetermined $Path variable. Whereas the procedure call LoadExtProg[x, y] with the second optional argument y – an undefinite variable – in addition through y returns the list of all full paths to the found datafile x without modification of the directories list determined by the predetermined $Path variable. In the case of absence of opportunity to find a required datafile x $Failed is returned."

Range1::usage = "The call Range1[J1, Jp] returns the list {J1, J2, J3, ..., Jp}, where J is a name and p is an positive integer. In addition, actual arguments {J1, Jp} are encoded in the format $xxx_yyyN (N = {0..p|1..p})."

Range2::usage = "The call Range2[J , p] returns the list {J1, J2, J3, ..., Jp}, where J is a name and p is an positive integer."

Range3::usage = "The call Range3[J , p] returns the list {J1_, J2_, J3_, ..., Jp_}, where J is a name and p is an positive integer."

Range4::usage =
"The calls Range4[x], Range4[x, y], Range4[x, y, z] on numerical arguments are equivalent to the calls Range[x], Range[x, y], Range[x, y, z] of standard function accordingly, at the same time as the call Range4[x, y] on symbols in string–format with codes 32 ÷ 4096 is equivalent to the call CharacterRange[x, y], the call Range4[x] returns the list of symbols with codes that are smaller or equal ToCharacterCode[x]."

Range5::usage =
"The function call Range5[x] returns a list that is formed on the basis of tuple x consisting from simple elements and spans."

AssignToList::usage = "The call AssignToList[L, Z, n] assigns values of elements of a list L to the corresponding elements of the generated list {Z1, Z2, ..., Zn} (n >= 1), returning the list of assigned values."

AssignL::usage =
"The function call AssignL[x, y] provides correct assignment to elements (to all or the given elements) of an arbitrary expression or expressions from the list y, modeling assignments on the basis of constructions of the format {x, y, z, …}[[n]] = Ex and {x, y, z, …}[[n ;; p]] = {Exn, Exn+1, …, Exp} and to them similar which the system doesn't support while the function

call AssignL[x, y, w] where w – an expression – provides the correct delayed assignments of the above–stated kind, as visually illustrates the previous fragment. At that, the function call on inadmissible appointments returns $Failed."

TransListList::usage = "The call TransListList[x] returns result of transposition of a list x of ListList–type. Procedure TransListList operates by the principle of the switch."

CALL::usage =
"The call CALL[G [x, ...], d] of the CALL procedure provides return of the result of the call G[x, ...] of a procedure/function G provided that its definition is in the datafile \"G.txt \" from a library directory which is defined by the second optional argument d. In case of the procedure call with one argument, the current directory defined by the call Directory[] is understood as the library directory. Moreover, if the datafile \"G.txt \" is absent then the procedure CALL returns $Failed. If procedure/function G is already activated in the current session, the datafile \"G.txt \" is ignored, and the call CALL[G[x, ...]] treats active object with name G. If the datafile \"G.txt \" contains definitions of several procedures/functions G of the same name, the call CALL[G[x, ...], d] is made concerning its last definition."

CALLmx::usage =
"The CALLmx procedure is an useful extension of the CALL procedure; its call provides saving in a library directory of definitions of objects in the form of mx–datafiles with possibility of their subsequent loading in the current session or other session of the system Mathematica. The call CALLmx[y, 1, d] returns the list, whose first element defines a library catalog, whereas the others define names of objects from the argument y (a separate name or their list) whose definitions are evaluated in the current session; the optional argument d defines the directory in which the evaluated definitions of objects from y in the form of mx–datafiles with names \"Name.mx \" (where Name – the name of an object determined by argument y) will be located; in case of absence of argument d the library catalog is determined by the call Directory[]. While the call CALLmx[y, 2, d] provides loading in the current session of objects, whose names are defined by argument y, from the library catalog determined by the third argument d or its absence."

CALLmxH::usage =
"The CALLmxH procedure is one of possible modifications of the CALLmx procedure supporting work with help database of the user library. For help database the list structure of the following format is used, namely:

Help[] := {F1::usage = \"Help on object F1\", ..., Fn::usage = \"Help on object Fn\"}

Whereas the format of the procedure call becomes CALLmxH[{F1,...,Fn}, {1|2}, Help, d], when everything the told relative to the first, second and fourth arguments concerning the call CALLmx[{F1,...,Fn}, {1|2}, d] of the CALLmx procedure completely is upheld, whereas the argument Help defines a help database of a library. At such approach the call CALLmxH[{F1,...,Fn}, 1, Help, d] (where the argument d is optional) creates the set of mx–files with definitions of means of the user library and the mx–file with help database. All subsequent loadings of the library by the call CALLmxH[{F1,...,Fn }, 2, Help, d] not only provide in the current session the availability of definitions of means {F1,...,Fn} that are situated in the user library, but also their usages from its help database."

DirQ::usage = "The call DirQ[d] returns the True if string d defines a real directory, and the False otherwise. The procedure expands the conventional testing function DirectoryQ onto case of different codings of directories."

DirEmptyQ::usage = "The call DirEmptyQ[d] returns True if a directory d is empty; otherwise, False is returned. The call is returned unevaluated if d is not a real directory."

DirFD::usage =
"In addition to the procedure DirEmptyQ, the procedure DirFD[d] provides return of the nested list whose first element defines the list of subdirectories of the first level of a directory d whereas the second – the list of datafiles of the directory d. If the directory d is empty, the call of procedure returns the empty list, i.e. {}."

DirFull::usage =
"In addition to the DirFD procedure the DirFull procedure represents a quite certain interest, whose call DirFull[d] returns the list of all full paths to the subdirectories and files contained in a directory d and its subdirectories; the first element of this list – the directory d. While on an empty directory d the call DirFull[d] returns the empty list, i.e. {}. The fragment below represents source code of the DirFull procedure with examples of its usage."

FindSubDir::usage =
"Procedure FindSubDir provides search in the file system of a computer or in the file system of the given devices of external memory with the direct access, given by their names in string–format, the full paths containing a subdirectory x, given by its name in string–format. In addition, the call FindSubDir[x] returns the list of full paths within of all file system of a computer, whereas the call FindSubDir[x, y, z, ...] – within of the file system of devices {y, z, ...} only."

TypeFilesD::usage =
"In addition to the DirFull procedure the call TypeFilesD[d] of the procedure TypeFilesD returns the sorted list of types of the files located in a directory d with returning of \"undefined\" on datafiles without of a name extension. At that, the datafiles located in the directory d and in all its subdirectories of an arbitrary nesting level are considered. Moreover, on the empty directory d the procedure call TypeFilesD[d] returns the empty list, i.e. {}."

CopyFileToDir

DirName::usage =
"The call DirName[F] returns the \"None\" if F is a real directory, the path to a directory containing a datafile F, and $Failed otherwise. The procedure expands the standard function DirectoryName onto case of real existence of F. Moreover, search is done within all file system of the computer, but not within only system of subdirectories determined by the predetermined $Path variable."

CopyDir::usage =
"The call CopyDir[d1, d2] copies a full directory d1 into a directory d2; if the target directory exists the directory d1 is copied as its subdirectory with returning path to the new directory. The function expands the standard function CopyDirectory onto case of existing target directories."

CopyFileToDir::usage =
"The procedure call CopyFileToDir[x, y] provides copying of a datafile or directory x into a directory y with return of the full path to the copied datafile or directory. If the copied datafile already exists, it isn't updated if the target directory already exists, the directory x is copied into its subdirectory of the same name."

CopyFile1::usage =
"The CopyFile1 procedure is an useful enough generalization of the standard CopyFile function whose call CopyFile[w1, w2] returns the full name of the datafile it copies to and $Failed if it cannot do the copy; in addition, datafile w1 must already exist whereas datafile w2 must not. Thus, in the program mode the standard function is insufficiently convenient. The procedure call CopyFile1[w1, w2] returns the full path to the datafile that had been copied. In contrast to the standard function, the procedure call CopyFile1[w1, w2] returns the list of format {w2, w2∗}, where w2 – the full path to the copied datafile and w2∗ – the full path to the previously existing copy of a datafile \"xyz.abc\" in the format \"xyz.abc\", if the target directory for w2 already contained datafile w1. In the case w1 ≡ w2 (where identity is considered up to standardized paths of both files) the procedure call returns the standardized path to the datafile w1, doing nothing. In the other successful cases the call CopyFile[w1, w2] returns the full path to the copied datafile. Even, if the nonexistent path to the target directory for the copied datafile is defined, then such path taking into account available devices of direct access will be created."

DelDirFile::usage =
"Similarly to standard functions DeleteFile and DeleteDirectory, the call DelDirFile[x] deletes from file system of a computer a file or a directory x, returning Null, i.e. nothing. Whereas the call DelDirFile[x, y] with the second optional argument y – an arbitrary expression – deletes from file system of a computer a file or a directory x, irrespectively from existence of attribute Read−only."

DelDirFile1::usage =
"The DelDirFile1 procedure presents an useful enough extension of the DelDirFile procedure onto the case of open datafiles in addition to the Read−only attribute for both separate datafiles, and the datafiles being in the deleted directory. The call DelDirFile1[x] is fully equivalent to the call DelDirFile1[x, y], providing removal of a datafile or a directory irrespective of openness of the separate datafile and the Read−only attribute ascribed to it, or existence of similar datafiles in the removed directory."

Attrib::usage =
"The successful procedure call Attrib[f, \"Attr\"] returns the list of attributes of a given datafile or directory f in the context Archive (\"A\"), Read−only (\"R\"), Hidden (\"H\") and System (\"S\"). At that, also other attributes inherent to the system datafiles and directories are possible; thus, in particular, on the main directories of devices of external memory \"Drive f\", while on a nonexistent directory or datafile the message \"f isn't a directory or datafile\" is returned. At that, the call is returned in the form of the list of the format {x, y, ..., z, F} where the last element determines a full path to a datafile or directory f; the datafiles and subdirectories of the same name can be in various directories, however processing of attributes is made only concerning the first datafile/ directory from the list of the objects of the same name. If the full path to a datafile/directory f is defined as the first argument of the Attrib procedure, specifically only this object is processed. The elements of the returned list that precede its last element determine attributes of a processed directory or datafile. The procedure call Attrib[f, {}] returns Null, i.e. nothing, canceling all attributes for a processed datafile/directory f whereas the procedure call Attrib[f, {\"x\", \"y\",..., \"z\"}] where x, y, z∈{\"−A\", \"−H\", \"−S\", \"−R\", \"+A\", \"+H\", \"+S\", \"+R\"}, also returns Null, i.e. nothing, setting/cancelling the attributes of the processed datafile/directory f determined by the second argument. At impossibility to execute processing of attributes the procedure call Attrib[f,x] returns the corresponding messages. The Attrib procedure allows to carry out processing of attributes of both the file, and the directory located in any place of file system of the computer."

Attrib1::usage =
"The Attrib1 procedure in many respects is similar to the Attrib procedure both in the functional, and in the descriptive relation, but the Attrib1 procedure has certain differences. The successful procedure call Attrib[f, \"Attr\"] returns the list of attributes in string format of a directory or datafile f in the context Archive (\"A\"), Read−only (\"R\"), Hidden (\"H\"), System (\"S\"). The procedure call Attrib[f, {}] returns Null, i.e. nothing, canceling all attributes for the processed datafile/directory f whereas the procedure call Attrib[f, {\"x\", \"y\",..., \"z\"}] where x, y, z∈{\"−A\", \"−H\", \"−S\", \"−R\", \"+A\", \"+H\", \"+S\", \"+R\"}, also returns Null, i.e. nothing, setting/cancelling the attributes of the processed datafile/directory f determined by the second argument, while call Attrib1[f, x, y] with the 3rd optional argument y – an expression – in addition deletes the program file \"attrib.exe\" from the directory determined by the call Directory[]. At that, the possibility of removal of the \"attrib.exe\" program file from the directory which is defiined by the call Directory[] after a call of the Attrib1 procedure leaves file Mathematica system unchanged."

Attribs::usage =
"This procedure provides only two functions – (1) obtaining the list of the attributes ascribed to a datafile or directory, and (2) removal of all ascribed attributes. The call Attribs[x] returns the list of attributes in string format which are ascribed to a datafile or directory x. On the main directories of volumes of direct access the procedure call Attribs returns $Failed. While the call Attribs[x, y] with the second optional argument y – an expression – deletes all attributes which are ascribed to a datafile or directory x with returning at a successful call 0. At that, the file \"attrib.exe\" is removed from the directory defined by the call Directory[] after a call of the procedure."

DelAllAttribs::usage =
"The successful call DelAllAttribs[x] returns 0, providing the removal of all attributes, ascribed to a datafile or a directory x."

InsertN::usage =
"The procedure call InsertN[S, L, n] returns the result of inserting into string S of substrings, defined by a list L after its positions defined by a list n. In a case n = {<1 | StringLength(S)} the corresponding substring from L is located before the beginning of string S or in its end accordingly. It is supposed that actual arguments L and n can contain different number of elements; in such case odd elements of n is ignored. In addition, processing of a string S is carried out concerning the list m of positions for inserts defined according to the following relation m = DeleteDuplicates[Sort[n]]."

DelSubStr::usage =
"The procedure call DelSubStr[S, L] returns the result of deleting from a string S of substrings, defined by a list L of its extremal positions, and extreme symbols defined by list L of their positions."

GV::usage = "The call GV[] returns the list of special symbols and global variables of the package at the current session."

Names1::usage =
"The call Names1[] returns the nested 4−element list, whose first element defines the list of names of the procedures, the second − the list of names of functions, the third element − the list of names whose definitions have been evaluated in the current session of the package, whereas the fourth element defines the list of other names associated with the current session."

CurrentNames::usage =
"The procedure call CurrentNames[] returns the nested list, whose elements define sublists of names in string format of means of the current session which by the first element are identified by their context other than the context \"System'\". While the procedure call CurrentNames[x] where x – an undefinite symbol – through x returns the nested list of the above format whose first element of sublists is a context whereas the others define symbols with this context that have no definitions, i.e. so−called concomitant symbols of different types."

ExtrPackName::usage =
"The call ExtrPackName[F, N] returns definition of an object N along with its calculation which is contained in a m−file F with a package, doing this definition as an accessible in the current session. If format of file F is distinct from the m−format, the call returns the value `$Failed` whereas at absence in the file F of requested object N the call returns the corresponding message."

SearchFile::usage = "The call SearchFile[F] returns the list containing full paths to a datafile F found in file system of a computer; if the datafile F has not been found, the empty list is returned."

SearchFile1::usage =
"The SearchFile1 procedure is a functional analog of the SearchFile procedure. The call SearchFile[F] returns the list of full paths to a datafile F found within file system of the computer; in the case of absence of the required file F the procedure call SearchFile[F] returns the empty list, i.e. {}. The procedure SearchFile1 essentially uses the procedure Adrive that is used by a number of our means of access. It should be noted that speed of both procedures generally very essentially depends on the sizes of file system of the computer, first of all, if a required datafile isn't defined by the full path and isn't in the directories determined by the $Path variable. Moreover, in this case the search is done even in the \"C:\\$Recycle.Bin\" directory."

SearchDir::usage =
"The call SearchDir[d] returns the list of all paths in file system of the computer which are completed by a subdirectory d; in case of lack of such paths the procedure call SearchDir[d] returns the empty list, i.e. {}. In combination with the procedures FindFile1 and FileExistsQ1 the SearchDir procedure is useful at working with file system of the computer, as confirms their usage for the solution of tasks of similar type."

NbName::usage =
"The call NbName[] returns the list of notebooks names loaded into the current session; in addition, order of notebooks is in accord with order of their loadings; at that, the first element of the list presents notebook active in the current session."

AllCurrentNb::usage =
"The AllCurrentNb procedure is a rather useful means at operating with nb−documents in the current session. The procedure call AllCurrentNb[] returns the nested list of ListList−type, the first element of each sublist defines a name in string format of nb−document activated in the current session whereas the second element determines the nb−file from which it was uploaded into the current session. If nb−document was created in the current session without saving, the second element of the sublist is the message \"nb has been not saved\". At that, the first sublist of the returned list defines the current

nb–document. If the returned list contains one sublist, it is converted in simple 2–element list of the above format."

NotebookSave1::usage =
"The NotebookSave1 procedure adjoins directly the procedure AllCurrentNb, significantly extending standard NotebookSave function. At that, the given procedure provides saving in a nb–file of an arbitrary nb–document opened in the current session. Moreover, the procedure provides saving of nb–documents earlier saved in nb–files, along with nb–documents opened only in the current session without their preliminary saving in nb–files, including documents \"Untitled–n\" and \"Messages\". The procedure call NotebookSave1[x, y] returns the path to a nb–file determined by the second y argument into which has been unloaded a nb–document x opened in the current session. The saving of nb–documents opened in the current session is done by the following manner, namely:
(1) if nb–document has been opened in the current session from a nb–file then it will be saved in the same nb–file, but not in a datafile y;
(2) if nb–document has been opened onle in the current session without preliminary saving in nb–file then it will be saved in a datafile y; at that, nb–document x receives name y in the current session, and it opens in a new window;
(3) if a nb–document x is not contained among nb–documents opened in the current session then the procedure call returns Null, i.e. nothing."

OpenCurrentNb::usage =
"The procedure call OpenCurrentNb[x] in the program mode allows to open window with a nb–document x opened in the current session with returning Null, i.e. nothing. At that, an existing nb–file or cdf–file, or a nb–document opened in the current session can be as argument x. Along with nb–documents the OpenCurrentNb procedure successfully processes CDF–documents too."

NbDocumentQ::usage =
"The procedure call NbDocumentQ[x] returns True if x is a real nb–document opened in the current session or located in a nb–file x, otherwise False is returned. Whereas the procedure call NbDocumentQ[x, y] with the second optional argument y – an indefinite variable – through y returns the nb–document type, namely: \"current\" (current nb–document), \"opened\" (nb–document, opened in the current session), or \"file\" (nb–document located in a nb–file x) if the main return is True, otherwise y is returned indefinite. Along with nb–documents the NbDocumentQ procedure successfully processes CDF–documents too."

NbCurrentQ::usage =
"The function NbCurrentQ is a simplistic version of the procedure NbDocumentQ. The function call NbCurrentQ[x] returns True, if x nb–document is opened in the current session, and False otherwise. Along with nb–documents the NbCurrentQ function successfully processes CDF–documents too."

FileCurrentNb::usage =
"The the procedure call FileCurrentNb[x] returns the datafile containing x nb–document uploaded in the current session; if the nb–document wasn't open in the current session the $Failed is returned, whereas on a document X with the name \"Untitled–n\" or \"Messages\" the message \"X wasn't saved\" is returned. Along with nb–documents the FileCurrentNb procedure successfully processes CDF–documents too."

NbFileEvaluate::usage =
"The function call NbFileEvaluate[x] evaluates all the calculated cells in notebook located in a datafile x of the format {\"cdf\", \"nb\"} with its opening in the new window. At that, the call evaluates the notebook as if all cells had been evaluated with hot keys \"Shift+Enter\". Messages, print output and other side effects are placed in the notebook along with outputs. If the call is used on a datafile x that is not open, the Mathematica will invisibly open the file, evaluate it entirely, save, and close the datafile. Whereas the function call NbFileEvaluate[x, y] with the second argument y – an arbitrary expression – opens a file x, evaluates it entirely and will leave the notebook completely unmodified."

MfileEvaluate::usage =
"The procedure call MfileEvaluate[x, y] evaluates a package located in a m–file x and depending on value of the second argument y performs the following actions concerning the m–file x with a package, namely:
y=1 – returns the list of all objects names whose definitions are in the m–file x;
y=2 – returns the 2–element list whose the first element defines the list of objects names of m–file that are available in the current session while the second element – the list of names of m–file that are not available in the current session;
y=3 – returns the context ascribed to a package located in the m–file x;
y=4 – prints the usages concerning all objects located in the m–file x in the format:
Name
The usage concerning Name.
Meantime, the procedure call MfileEvaluate[x, y] where x defines a m–file not contained a package or m–file with a package containing the definition of the MfileEvaluate procedure returns the $Failed."

MfileLoad::usage =
"The procedure call MfileLoad[x] evaluates the package located in a m–file x, returning Null, i.e. nothing. Whereas the call MfileLoad[x, y] with the second optional argument y – an indefinite variable – thru y additionally returns the 3–element list whose elements define the context ascribed to the package x, list of main objects names whose definitions are in the package, and the list of names with the given context accordingly. In particular, the third element can define objects of the system packages built into the system as it takes place for the

Combinatorica package, options, etc. The procedure call on a m–file not contained a package returns $Failed."

LoadFile::usage =
"In a lot of cases there is a necessity of loading into the current session of files {nb, m, mx, txt}–types or files of ASCII–format without extension of the name, that are located in one of directories of file system of a computer. The given problem is solved by means of procedure LoadFile, whose call LoadFile[F] loads into the current session a datafile given by its full name F with extension {\".m\", \".nb\", \".mx\", \".txt\"} or at all without expansion. Moreover, at a finding of the list of such datafiles with identical name F the loading of the first file of the list with return of the corresponding message is made whereas through the global variable $Load$Files$ the procedure returns the list of all F datafiles found during search. In the case of lack of F datafiles through the $Load$Files$ variable the empty list, i.e. {} is returned. The procedure handles basic erroneous and especial situations."

IsFileOpen::usage =
"The call IsFileOpen[F] returns True if a datafile F, defined by the name or the full path, is open, and False otherwise. If the argument F doesn't define an existing datafile, the call is returned unevaluated. Whereas the call IsFileOpen[F, h] with the second optional argument h – an indefinite variable – returns through h the nested list whose sublists have format {{\"read\", \"write\"}, {The list of streams on which datafile F is open on the reading/recording}} if the main result is True."

AcNb::usage = "The call AcNb[] returns full name of the current document earlier saved as a nb–file."

StrStr::usage = "The call StrStr[x] returns an expression x in string format if x is different from string; otherwise, the double string obtained from an expression x is returned."

CNames::usage =
"The call CNames[x] returns the list of all short names in a package with a context x, that have in it definitions, whereas the call CNames[x, y] in addition through optional argument y – an undefinite variable – returns the list of all undefined short names in the package with context x."

VarExch::usage =
"The procedure VarExch[L] provides the exchange of values of variables with the corresponding exchange of all their attributes. For example, for L = {x, y} variables x and y, having values 68 and 63, should receive values 63 and 68 accordingly with a corresponding exchange of all of their attributes. The procedure VarExch[L] solves the given problem{task}, returning the value Null, i.e. nothing. The list of two names of variables which should be exchanged by values and attributes in string–format along with the nested list from sublists of the specified type are supposed as argument L; in any case all elements of pairs should be defined; otherwise, the procedure call returns the value Null with printing of the corresponding diagnostic message."

VarExch1::usage =
"The procedure VarExch1[L] similarly to the procedure VarExch[L] provides the exchange of values of variables with the corresponding exchange of all their attributes. The procedure call returns the value Null, i.e. nothing. The list of two names of variables which should be exchanged by values and attributes in string–format along with the nested list from sublists of the specified type are supposed as argument L; in any case all elements of pairs should be defined; otherwise, the procedure call is returned unevaluated without printing of any diagnostic message."

ListListQ::usage = "The call ListListQ[L] returns the True if L defines a list of lists of identical length, and the False otherwise."

RestListList::usage =
"The procedure call RestListList[x] returns the result of restructuring of a list {{x1, x2, ..., xn}, {y1, y2, ..., yn}, ..., {z1, z2, ..., zn}} of ListList–type into the list of the same type of the view {{x1, y1, ..., z1}, {x2, y2, ..., z2}, ...}."

RestListList1::usage =
"The procedure RestListList1 is an analog of the RestListList procedure. The procedure call RestListList1[x] returns the result of restructuring of a list {{x1, x2, ..., xn}, {y1, y2, ..., yn}, ..., {z1, z2, ..., zn}} of ListList–type into the list of the same type of the view {{x1, y1, ..., z1}, {x2, y2, ..., z2}, ...}."

FullNestListQ::usage =
"The procedure call FullNestListQ[x] returns True if all elements of a list x are the nested sublists, and False otherwise."

StrExprQ::usage = "The call StrExprQ[x] returns the True if a string x contains a correct expression, and the False otherwise."

ExpressionQ::usage =
"The call SyntaxQ[w] of standard Mathematica function returns True if a w string corresponds to syntactically correct input for a single expression, and returns False otherwise. At that, the function tests only syntax of expression, ignoring its semantics at its evaluation. While the ExpressionQ procedure along with the syntax provides testing of an expression regarding semantic correctness. The procedure call ExpressionQ[w] returns True if a string w corresponds to syntactically and semantically correct single expression, and returns False otherwise."

ExprsInStrQ::usage =
"The call ExprsInStrQ[x, y] returns True if a string x contains the correct expressions, and False otherwise. In addition, through the

second optional argument – an undefined variable y – the list of all correct expressions contained in the string x is returned."

SeqQ::usage = "The call SeqQ[x] returns the True if x is an Seq−object (sequence), and the False otherwise."

SeqToList::usage = "The call SeqToList[x] returns the result of converting of a Seq−object x into List."

ListToSeq::usage = "The call ListToSeq[x] returns the result of converting of a list x into Seq−object."

Head1::usage = "The call Head1[x] returns the heading of an expression x in the
context {Block, Function, Module, System, Symbol, Head[x], PureFunction}. At that, on the objects
of the same name that have one name with several definitions the procedure call returns $Failed."

Head2::usage =
"The Head2 procedure seems as an useful enough means that is a modification of the Head1 procedure and that is based
on the previous ProcFuncTypeQ procedure and the standard Head function. The procedure call Head2[x]
returns the heading or the type of an object x, given in string format. In principle, the type of an object can
quite be considered as a heading in its broad understanding. The Head2 procedure serves to such problem
generalizing the standard Head function and returning the heading of an expression x in the context of
{Block, CompiledFunction, Function, Module, PureFunction, ShortPureFunction, Symbol, System, Head[x]}."

Head3::usage =
"The Head3 function expands the standard Head function and the above procedures Head1, Head2 provided that a tested
expression x is considered accurate to the sign; as a whole, the call Head3[x] is similar to the call Head1[x]."

FullFormF::usage = "The call FullFormF[] returns the list of the most often used
equivalents of functions and operators used in the full form of representation of expressions."

ReductRes::usage =
"The function call ReductRes[P, a] returns the reduced result a of a procedure call P. By the reduced result is understood
a result without context associating with the user package containing the procedure P."

ReplaceProcBody::usage =
"The call of function ReplaceProcBody[x, y] that is based on the procedure PartProc, returns Null, i.e. nothing,
and provides replacement of the body of a procedure x onto a new body y, given in
string−format. Furthermore, the updated object x is activated in the current session."

SeqIns::usage =
"The call SeqIns[x, y, z] returns the inserting result of an element y (list, Seq−object, etc.) into a Seq−object x according
to the given position z (z <= 0 − before x, z> = Length[x] − after x; otherwise, after z−position in x)."

SeqDel::usage = "The call SeqDel[x, y] returns the deletion result of an element y (list, Seq−object, etc.) from a Seq−object x."

LeftFold::usage = "The call LeftFold[F, id, s] procedure iterates procedure
F over string s, composing the successive results from the left with the initial value id."

RightFold::usage = "The RightFold[F, id, s] procedure iterates procedure
F over string s, composing the successive results from the right with the initial value id."

IsPermutation::usage =
"The call IsPermutation[x] returns the True if x is a permutation; otherwise, the False is returned. A string is a permutation
if, and only if, each character in the string occurs exactly once. The call IsPermutation[\"\"] returnes the True."

CatN::usage = "The call CatN[s, n] returns the result of n−fold concatenation of a string s."

LongestCommonSubString::usage = "The procedure call LongestCommonSubString[x,
y] returns the list of substrings of maximal length that are common for strings x and y."

LongestCommonSubSequence::usage =
"The procedure LongestCommonSubSequence[x, y] is similar to the procedure LongestCommonSubString[x,
y] however its call returns the list of maximal SubSequences common for strings x and y."

LongestCommonSubsequence1::usage =
"In contrast to system function LongestCommonSubsequence the procedure call LongestCommonSubsequence1[x,
y, J] in mode IgnoreCase –> J ∈ {True, False} of search of strings finds the longest contiguous
substrings common to the strings x and y. Whereas the call LongestCommonSubsequence1[x, y, J, t]
additionally through an undefinite variable t returns the list of all common contiguous substrings."

LongestCommonSubsequence2::usage =
"The LongestCommonSubsequence2 procedure is an extension of the LongestCommonSubsequence1 procedure in the case of a

finite number of strings in which the search for longest common continuous substrings must be done. The procedure call LongestCommonSubsequence2[x, y, ..., h, J] in mode IgnoreCase –> J ∈ {True, False} finds the longest contiguous substrings common to the strings x, y, ..., h. At that, the setting {True|False} for the last argument J is equivalent to the option IgnoreCase –> {True|False} that treats lowercase and uppercase letters as equivalent or not at search of substrings."

PalindromeQ1::usage = "The call PalindromeQ1[x] returns the True if a string x is palindrome; otherwise, the False is returned."

MaximalPalindromicSubstring::usage = "The procedure call MaximalPalindromicSubstring[x]
returns the list of substrings of a string x of maximal length that are palindromes."

BinaryListQ::usage =
"The call BinaryListQ[L] returns the True if L defines a binary list including its possible sublists, and the False otherwise."

SubDelStr::usage =
"The call SubDelStr[x, L] provides deletion from a string x of all substrings whose numbers of positions are defined by a LisiList L."

ToString1::usage =
"The call ToString1[x] returns the result of correct converting of an arbitrary x−expression into string format irrespective
of presence in expression x of string components. The procedure extends the standard function ToString."

ToString2::usage = "The call ToString2[x] returns the result ToString1[x] if
x – not a list; otherwise, ToString1[x] under condition that ToString1 has Listable−attribute."

ToString3::usage = "The call ToString3[x] serves for converting of an expression x in string InputForm format."

ToString4::usage =
"The call ToString4[x] is analogous to the call ToString1[x] if x – a symbol, otherwise \"(\" <> ToString1[x] <> \")\" is returned."

DefToString::usage =
"The procedure call DefToString[x] in string format returns the definition of an object whose name x is encoded in string format."

ToStringRule::usage = "The call ToStringRule[x] returns a rule or list of rules x whose
left and right parts are in string format. In addition, the right parts of rules are edged by brackets."

ToStringRule1::usage = "The call ToStringRule1[x] returns a rule or list of rules x whose left and right parts are in string format."

StringSplit1::usage =
"As against function StringSplit[x, y], the call of procedure StringSplit1[x, y] carries out semantic splitting of a string x by a symbol y
onto elements of the returned list. In this case the semantics consists in what in the returned list are located substrings of
string x which contain correct expressions; in absence of such substrings the call of procedure returns the empty list, i.e. {}."

StringSplit2::usage =
"As against function StringSplit[x, y], the call of procedure StringSplit2[x, y] includes in the returned list the substrings of a string x
which begin with delimiter y. In case of absence of a delimiter y into string x, the call StringSplit2[x, y] returns {x}."

SubStr::usage =
"The call SubStr[S, p, a, b, R] returns a substring of a string S that is limited from the left from p−position by the first symbol
different from symbol a or from elements of list a, and is limited from the right from p−position by the first symbol different
from symbol b or from elements of list b; whereas in erroneous case through argument R an appropriate warning is returned."

NamesCS::usage =
"The call NamesCS[P, Pr, Pobj] returns the Null−value, i.e. nothing, whereas through three arguments P, Pr and Pobj are returned
the list of contexts corresponding to packages loaded into current session, the list of the user procedures whose definitions
were activated in the current session, and the nested list accordingly. In addition, the nested list has the following structure,
namely: the first element of each sublist defines the context corresponding to an appropriate package loaded into the current
session, whereas its other elements defines names of objects of the package which were activated in the current session."

NamesContext::usage =
"The call NamesContext[x] returns the list of names in string–format of program objects of the current session which
are associated with a context x. In case of absence of this context the empty list, i.e. {}, is returned
whereas in case of value x, different from a context, the procedure call is returned unevaluated."

Bits::usage =
"On a tuple of actual arguments <x, P> where x − a string of length 1 (character) and P − an integer in the range 0..8, the call Bits[x, P]
returns binary representation of x in the form of the list if P=0, and the P−th bit of such representation of x otherwise. Whereas
on a tuple of actual arguments <x, P> where x − a nonempty binary list of length no more than 8 and P=0, the call Bits[x, P]
returns a symbol that corresponds to the given binary list x; in other cases the call of procedure is returned unevaluated."

Nconcat::usage = "The call Nconcat[x] returns the concatenation of elements of a list x – integers > 0; furthermore, the

list Flatten[x] is considered instead of x, providing operations with the nested lists of any level of nesting."

StringTrim1::usage =
"The procedure StringTrim1 is an useful extension of standard function StringTrim. The call StringTrim1[x,y,z] returns the result of truncation of a string x by a substring y on the left, and by a substring z on the right. In case y = z = \"\" the call StringTrim1[x, \"\", \"\"] is equivalent to the call StringTrim[x]."

StringTrim2::usage =
"The procedure StringTrim2 is an useful extension of standard function StringTrim and procedure StringTrim1. the procedure call StringTrim2[x, y, z] returns the result of truncation of a string x by symbols y on the left (z = 1), on the right (z = 2) or both ends (z = 3).. In case x = \"\" the procedure call returns the empty string, i.e. \"\"; at that, a single character or a list of them can act as an argument y."

TrueCallQ::usage =
"In contrast to the above procedures TestArgsTypes, TestArgsTypes1 that provide the differentiated testing of the actual arguments received by a tested object for their admissibility, the simple function TrueCallQ provides testing of correctness of the call of an object of type {Block, Function, Module} as a whole; the call TrueCallQ[x, args] returns True if the call x[args] is correct, and False otherwise. At that, the lack of the fact of the unevaluated call, and lack of the special or erroneous situations distinguished by Mathematica is understood as a correctness of the call."

TestProcCalls::usage =
"The call TestProcCalls[x, y] returns the nested list, whose elements have format {j, \n\, True|False}, where j – order number of an argument, \n\ – formal argument in string format, {True|False} – value which defines the admissibility (True) or inadmissibility (False) of the actual value determined by the list y and received by a formal argument {j, n} in the call point of an object x. It is supposed that the object x defines the fixed number of formal arguments and lengths of lists, defining by formal arguments and y, are identical, otherwise the procedure call returns $Failed."

SymbolToList::usage = "The call SymbolToList[x] returns the result of converting of a symbol x into list."

GenRules::usage =
"In Mathematica, the transformation rules generally are defined by the function Rule, whose call Rule[a, b] returns the transformation rule in the format a –> b. For dynamic generation of such rules the GenRules procedure can be quite useful, whose call GenRules[x, y] depending on a type of its arguments returns a rule or the list of rules; the call GenRules[x, y, z] with the third optional argument z – any expression – returns the list with one rule or the nested list of ListList–type. Depending on the format of coding of the procedure call the following result is returned, namely:
GenRules[{x, y, z, ...}, a] ⟹ {x –> a, y –> a, z –> a, ...}
GenRules[{x, y, z, ...}, a, g] ⟹ {{x –> a}, {y –> a}, {z –> a}, ...}
GenRules[{x, y, z, ...}, {a, b, c, ...}] ⟹ {x –> a, y –> b, z –> c, ...}
GenRules[{x, y, z, ...}, {a, b, c, ...}, g] ⟹ {{x –> a}, {y –> b}, {z –> c}, ...}
GenRules[x, {a, b, c, ...}] ⟹ {x –> a}
GenRules[x, {a, b, c, ...}, g] ⟹ {x –> a}
GenRules[x, a] ⟹ {x –> a}
GenRules[x, a, g] ⟹ {x –> a}."

GenRules1::usage =
"In Mathematica, the transformation rules generally are defined by the function Rule, whose call Rule[a, b] returns the transformation rule in the format a –> b. For dynamic generation of such rules the GenRules procedure can be quite useful, whose call GenRules[x, y] depending on a type of its arguments returns a rule or the list of rules; the call GenRules[x, y, z] with the third optional argument z – any expression – returns the list with one rule or the nested list of ListList–type. Along with the above rules of type a –> b Mathematica allows application of the so–called delayed rules (RuleDelayed) of type a :> b or a :⇢ b which are realized only at the moment of their application. For the purpose of generation of lists of transformation rules of this type, the GenRules procedure, presented above in which the Rule function is replaced with the RuleDelayed function can be used, or its modification GenRules1 adapted for use of one or another function by the corresponding coding of the 3rd actual argument is used. In the call GenRules1[x, y, h, z], where x, y, z – arguments which are completely similar to the arguments of the GenRules procedure with the same names, whereas the argument h determines the mode of generation of the list of the usual or delayed rules on the basis of the value \"r\" received by it (simple rule) or \"rd\" (delayed rule). Depending on the format of coding of the procedure call the following result is returned (only for the delayed rules, for the usual rules see the above procedure GenRules, i.e. argument h has value \"rd\"), namely:
GenRules[{x, y, z, ...}, \"rd\", a] ⟹ {x :⇢ a, y :⇢ a, z :⇢ a, ...}
GenRules[{x, y, z, ...}, a, \"rd\", h1] ⟹ {{x :⇢ a}, {y :⇢ a}, {z :⇢ a}, ...}
GenRules[{x, y, z, ...}, {a, b, c, ...}, \"rd\"] ⟹ {x :⇢ a, y :⇢ b, z :⇢ c, ...}
GenRules[{x, y, z, ...}, {a, b, c, ...}, \"rd\", h1] ⟹ {{x :⇢ a}, {y :⇢ b}, {z :⇢ c}, ...}
GenRules[x, {a, b, c, ...}, \"rd\"] ⟹ {x :⇢ a}
GenRules[x, {a, b, c, ...}, \"rd\", h1] ⟹ {x :⇢ a}
GenRules[x, a, \"rd\"] ⟹ {x :⇢ a}
GenRules[x, a, \"rd\", h1] ⟹ {x :⇢ a}."

GenRules2::usage =

"The call GenRules2[x, y] where x – a non–nested list and y – an expression different from list, or a list depending on type of the second argument y generates the list of transformation rules of the following formats respectively, namely:
GenRules[{x, y, z, …}, a] ⟹⟹ {x –> a, y –> a, z –> a, …}
GenRules[{x, y, z, …}, {a, b, c, …}] ⟹⟹ {x –> a, y –> b, z –> c, …}"

RevRules::usage = "The call RevRules[x] returns the rule or the list of rules that are reverse to the rules determined by argument x – a rule of format \"a –> b\" or their list."

$Load$Files$::usage = "The predefined variable $Load$Files$ is used by the procedure LoadFile."

WhatType::usage =
"The procedure call WhatType[x] returns type of an object x of one of base data types {\"Module\", \"DynamicModule\", \"Block\", \"Complex\", \"Integer\", \"Rational\", \"Real\", \"Times\", \"Plus\", \"List\", \"Power\", \"And\", \"Rule\", \"Condition\", \"StringJoin\", \"UndirectedEdge\", \"Alternatives\", etc.}. Procedure WhatType does not support exhaustive testing of types, however on its basis it is enough simply to expand a class of checked types of the data."

TwoHandQ::usage =
"The procedure call TwoHandQ[x] returns the True if an expression x has one of the following types, namely: {\"+\", \">=\", \"<=\", \"&&\", \"||\", \"–\", \"^\", \"**\", \"<\", \"==\", \"!=\", \">\", \"–>\"}, and the False otherwise. In addition, in case of the call TwoHandQ[x, y] through the second argument y the type of x is returned if the main result of the call is True."

Iff::usage =
"The procedure Iff[x,...] is equivalent to the standard function If[x,...] in the limits of number of arguments 2..4, returning the unevaluated call in case of other number of arguments. In addition, it is necessary to have in view, that all actual arguments, since the second one, are coded in string format in order to prevent their premature calculation at the call Iff[x,...] when actual arguments are calculated/simplified."

Args::usage =
"The call Args[F] returns the list of formal arguments of a block/function/module x along with the pure functions and function Compile. In addition, the format of the returned result is defined by type of an object x, namely:

– the list of formal arguments is returned on the Compile function with the types ascribed to them;
– on a module/block and typical function the list of formal arguments with the tests ascribed to them for testing admissibility of the actual arguments or without them is returned; in addition, the Args procedure processes the situation \"the objects of the same name with various headings\", returning the nested list of formal arguments concerning of all subobjects of an object in the order determined by the function Definition;
– on a pure function in the short format the list of deputies of formal arguments {#1,..., #n} in string format, while for a standard pure function the list of formal arguments in string format is returned.

Moreover, the call Args[W, h] with the second optional argument h – any admissible expression or any their sequence – can act, returns the result, similar to the call with the first argument, with that difference what all formal arguments are encoded in string format, but without the types ascribed to arguments and tests for testing of their admissibility. On an inadmissible actual argument W the call Args[W] is returned unevaluated."

Args1::usage =
"The procedure call Args1[x] returns simple or the nested list, whose elements are 2–element lists, whose first element represents the list of formal arguments with the types and tests, ascribed to them while the second – an object type in the context {\"Module\", \"Block\", \"Function\"}. As argument x the objects on which BlockFuncModQ[x] returns True are allowed. On an unacceptable argument x the procedure call Args1[x] is returned unevaluated."

ArgsBFM::usage =
"The call ArgsBFM[x] returns the list of formal arguments in string format of a block/function/module x, whereas the call ArgsBFM[x, y] with the second optional argument y – indefinite variable – additionally returns through it the list of formal arguments of the block/function/module x with tests ascribed to them in string format."

ArgsBFM1::usage =
"The procedure is an useful modification of the ArgsBFM procedure. The procedure call ArgsBFM1[x] generally returns the list of ListList–type whose 2–element sublists in string format define the name of a formal argument (the first element) and its admissibility test (the second element) of a x block/function/module. Lack of the test is coded as \"Arbitrary\"; at the same time, under lack of the test is understood not only its actual lack, but also the optional and default patterns ascribed to a formal argument. The given procedure successfully processes the objects of the same name too, i.e. the objects with several headings."

ArrayInd::usage =
"The procedure call ArrayInd[H] returns the list of the form {\"H[n1]=c1\", ..., \"H[nk]=ck\"}, where H – a name of an array in string–format which has been created by means of assignments H[n]{=|:=}Value_n. If H is not an array of the above type, the call of procedure ArrayInd returns the empty list or the list with definition of an object H given in string–format."

MdP::usage =

"The call MdP[x] returns a simple 2–element list, in which the first element – an object name in string format and the second element – number of headings with such name (if x defines a procedure/function/block activated in the current session; in the absence of similar object $Failed is returned); the nested list whose 2–element sublists have the structure described above (if an object x defines the list of the modules/functions/blocks activated in the current session), the nested list of the previous format (if x is empty, defining the list of all functions/blocks/modules activated in the current session); in the absence of the functions/modules/blocks activated in the current session the call MdP returns $Failed."

HeadingQ::usage =
"The procedure call HeadingQ[x] returns True if an object x, given in string format, can be considered as a syntactic correct heading; otherwise False is returned; in case of inadmissible argument x the call HeadingQ[x] is returned unevaluated."

HeadingQ1::usage =
"The procedure call HeadingQ1[x] returns True if the actual argument x, given in string format, can be considered as a syntactically correct heading; otherwise False is returned; in case of inadmissible argument x the call HeadingQ[x] is returned unevaluated [3]."

HeadingQ2::usage =
"Analogously to the procedures HeadingQ and HeadingQ1, the procedure call HeadingQ2[x] returns True if actual argument x, set in string format, can be considered as a syntactically correct heading; otherwise False is returned [8]."

HeadingQ3::usage = "The call HeadingQ3[x] returns True if an actual argument x, set in string format, can be considered as a syntactically correct heading; otherwise, the call returns False [8]."

TestHeadingQ::usage =
"The procedure call TestHeadingQ[x] returns True if x represents the heading in string format of a block/module/function, and False otherwise. Whereas the procedure call TestHeadingQ[x, y] with the second optional y argument – an undefinite variable – through it additionally returns information specifying the reasons of the incorrectness of x heading. Meanwhile, on the x objects of the same name (multiple objects) the procedure call returns $Failed; extension of the procedure onto case of the multiple objects does not cause much difficulty. From the detailed analysis follows, that the TestHeadingQ procedure on the opportunities surpasses the testing procedures HeadingQ ÷ HeadingQ3."

StandHead::usage =
"Realization of algorithms of a number of the procedures that significantly use the headings requires the coding of headings in the format corresponding to the system agreements at evaluation of definitions of a procedure/function/block. For automation of representation of a heading in the standard format the StandHead procedure can be quite useful whose call StandHead[h] returns the heading of a block/ procedure/function in the format corresponding to the system agreements at evaluation of its definition."

ExtrProcFunc::usage =
"The call ExtrProcFunc[h] returns an unique name of a generated block/function/ procedure that in the list of definitions has a heading h; otherwise, $Failed is returned. The procedure is characteristic in that leaves all definitions of a symbol HeadName[h] without change. At that, the returned object saves all options and attributes ascribed to the symbol HeadName[h]."

RepStandFunc::usage =
"The procedure call RepStandFunc[x, y, z] returns the call of a means y of the same name with a standard function y, and whose definition is defined in string format by the argument x, on argument z of its actual arguments. At the same time, such call of the RepStandFunc procedure is once–only in the sense that after the call the initial state of the standard function y remains without change."

SetAttributes1::usage =
"The call SetAttributes1[x, y] expands the standard function SetAttributes onto the form of representation of the first argument x by which can be indexed variables and lists, etc., in particular, providing setting of attributes y for elements of a list x."

AttrOpts::usage =
"The procedure call AttrOpts[x] returns the 2–element nested list whose first element determines options whereas the second element defines the list of the attributes ascribed to a symbol x of type {Block, Funcion, Module}. On a symbol x without options and attributes ascribed to it, the call AttrOpts[x] returns {{}, {}}."

$Line1::usage =
"The global variable $Line1 as against the standard global variable $Line defines the general number of Out–paragraphs of the current session of Mathematica, including results of calculation of the user packages loaded into it from datafiles of the format {\"cdf\", \"nb\"}."

$ProcType::usage =
"The procedure variable $ProcType is used only in the body of a block or a module and receives value of type in string–format of the object containing it in the context of {\"Block\", \"Module\"}; outside of objects of the specified type the variable $ProcType accepts the \"ToString\" value which doesn't have special value."

$TypeProc::usage =
"The procedure variable $TypeProc is used only in the body of a procedure of any type and receives value of type in string−format of the given procedure in the context of \"Block\", \"Module\" and \"DynamicModule\"; outside of a procedure the variable has value $Failed, whereas use of the variable outside of procedures is incorrectly, causing exigent condition with return $Failed."

$CallProc::usage =
"To $TypeProc the procedural variable $CallProc directly adjoins whose call returns contents of the body in string−format of a block or a module containing it, in moment of its call. In addition, for the module the body with local variables with the symbols \"$\" ascribed to them, whereas for the block its body in standard format is returned. The variable call outside of a block or a module receives \"StringTake[ToString1[Stack[_][[1]]], {10, −2}]\" value."

ModuleQ::usage =
"The procedure call ModuleQ[M, y] returns the value True if an object M an object M given by a symbol is a module; otherwise, the value False is returned. In addition, in case of True through the second argument y the type of module is returned, namely: \"Module\" or \"DynamicModule\". In case False y is returned unevaluated. In addition, the procedure call on a tuple of incorrect actual arguments is returned unevaluated. In other cases the call ModuleQ[M, y] returns False."

ModuleQ1::usage = "The call ModuleQ1[x] returns the True if x is a Module and the False otherwise."

ModuleQ2::usage = "The call ModuleQ2[x] returns the True if x is a Module and the False otherwise."

FuncBlockModQ::usage =
"The call FuncBlockModQ[x, y] returns True, if x – a symbol defining an object of type {Function, Module, Block}; in addition, in the presence for symbol x of several definitions the True is returned only when all its definitions accompanying x, will generate an object of the same type. Whereas through the 2nd argument y – an undefinite variable – the object type from the point of view of {\"Function\", \"Block\", \"Module\"} is returned. If the symbol x defines object of the same name, whose definitions associate with subobjects of different types, the call FuncBlockModQ[x, y] returns False whereas through the 2nd argument the value \"Multiple\" is returned."

Affiliate::usage = "The procedure call Affiliate[x] returns the context for an arbitrary symbol x, given in string−format; the value \"Undefined\" is returned for a symbol fully undefined in the current session."

ListListGroup::usage = "The procedure call ListListGroup[x, n] returns the nested list − result of grouping of a ListList−list x on the basis of n−th elements of sublists composing x."

CallsInProc::usage =
"The call CallsInProc[x] returns the nested 3−element list whose first element – the list of names in string−format of system means, the second element – the list of the user means in string−format that are contained in an object x of type {block, function, module} whereas the third element – the list of the contexts corresponding to the user means of the second element."

Border::usage = "The procedure call Border[x] returns the the border of a string x, where the border of a string x is the maximal prefix of x that is also a suffix of x."

Restart::usage =
"The procedure call Restart[] returns nothing, deleting from the current session all objects defined in it. Moreover, from the given list are excluded the objects whose definitions are in the downloaded packages. While the call Restart[x] with optional argument x – a context defining the user package which has been loaded into the current session – also returns nothing, additionally deleting from the current session all objects whose definitions are contained in the mentioned user package. Moreover, the system objects are not affected by the Restart. The procedure is easily extended to the case when the list of contexts is used as the argument x. The call Restart[x] deletes from the current session all objects with context x."

ObjType::usage =
"The call of procedure ObjType[x] returns the type of an object x in the context of {Function, Module, Block or DynamicModule}, in other cases the type of the expression assigned to a symbol x in the current session by operators {: =, =} is returned.In addition, the procedure ObjType ascribes to the type Function not only especially functional objects, but also definitions of following format Name[x _, y _, z _, ...]: = Expression; in this case the call returns the list of the following format, namely: {\"Name[x _,y _,z _,...]\", {Function, Head [Expression]}}."

ObjInCurrentNb::usage =
"The procedure call ObjInCurrentNb[] returns the nested list of the objects activated in a Nb−document of the current session. In addition, the objects are grouped according to the contexts attributed to them. In each sublist of the returned list the first element defines the context while the others define names of objects in string format that have this context."

ObjInCurrentNb1::usage =
"The ObjInCurrentNb1 procedure is a rather useful extension of the previous ObjInCurrentNb procedure. Similarly, the procedure call ObjInCurrentNb1[x] returns the nested list of objects located in a nb−document x opened in the current session or in a nb−file x. These objects are grouped according to the contexts attributed to them. At that, in each sublist of the returned list the first element defines the context while the others define names of objects in string format that have this

context. Furthermore, an attempt to analyze nb–document x opened in the current session without saving in a nb–file returns the result of analysis in the current nb–file (i.e. nb–document in which the call ObjInCurrentNb1[x] was done)."

PartialSums::usage =
"The procedure call PartialSums[L] returns the list of partial sums of the elements in a list L. In addition, if symbol L had been given in string–format then an initial list L is updated in situ."

ToDefOptPF::usage =
"As a result of loading of the user package into the current session the definitions of a part of its means x, that are returned by the standard Definition function, contain constructs of the format Context[x] <> ToString[x] <> \"`\", sometimes it is essentially complicates both review of the definition, and its processing. The call ToDefOptPF[x] returns the definition of a procedure/function x really by converting it in the current session into the optimum format which isn't containing mentioned context constructions. In addition, the attributes, attributed to object (procedure or function) x, remain. Moreover, the procedure ToDefOptPF correctly processes also so–called objects of the same name, i.e. objects which under one name have more than one definition with various headings."

IFk::usage =
"The procedure IFk implements a Maple clause <if L1 then V1 elif L2 then V2 elif L3 then V3 elif L4 then V4 ... else Vk end if (1)>, that is very useful for programming the branching algorithms. The procedure call IFk uses any number of actual arguments more than one which are 2–element lists of the format {Lj , Vj}, excepting the last (1). As a last argument any correct expression of language is allowable; in addition, check of Lj on the boolean type is not made. The procedure call IFk on a tuple of correct actual arguments returns the result that is equivalent to execution of the corresponding Maple–clause (1)."

IFk1::usage =
"Procedure IFk1 – a certain useful expansion of the procedure IFk which as against IFk admits only boolean expressions as actual arguments Lj; otherwise, the call is returned unevaluated. In the rest, the procedures IFk and IFk1 are functionally identical."

NamesNbPackage::usage =
"The call NamesNbPackage[F] returns the list of objects names saved as a package in a file F of nb–format; at that, it is presupposed that the objects are supplied by usages. In the absence of such objects the empty list is returned."

ToList::usage = "The procedure call ToList[x] returns the result of converting of an expression x into list."

PartProc::usage =
"In a series of the problems caused by a processing of string representation of definitions of procedures, the problem of partition of the given representation onto two basic components,namely: the body of procedure and its bounding box with a closing procedural bracket \"]\" represents the certain interest. The call of procedure PartProc[P] returns the 2–element list, whose first element in string–format represents the bounding box with a closing procedural bracket \"]\" of a procedure; the place of the procedure body occupies the substring \"Procedure Body\" whereas the second element of the list in string–format represents the body of the procedure P. In addition, under the bounding box with the closing procedural bracket \"]\" is understood the construction of the format \"Heading: = Module[{locals}, ...]\". At erroneous situations the call is returned unevaluated, or $Failed is returned."

Try::usage =
"The Try function is similar to try–предложению of the Maple, providing processing of x depending on the messages initiated by the evaluation of x. Furthermore, it is necessary to note that all messages initiated by such calculation of an expression x, should be activated in the current session. The call of the function has the following format, namely:

Try[\"x–expression\", y}}]

where the first argument x defines an expression x in string format whereas the second argument defines the message associated with a possible special or erroneous situation at calculation of expression x. In case evaluation of an expression x is correct, the result of its evaluation (for example, the procedure call) is returned, otherwise the nested list of the format {y, {Mes}} is returned where Mes defines the system message generated as a result of processing of a erroneous or special situation by the system. The function Try proved itself as a rather convenient means for processing of special and erroneous situations at programming of a number of applied and system problems."

NamesNbPackage1::usage =
"The call NamesNbPackage1[F] returns the list of objects names saved as a package in a file F of nb–format; at that, it is presupposed that the objects are supplied by usages. In the absence of such objects the empty list is returned. The Procedure is an useful modification of procedure NamesNbPackage."

NamesMPackage::usage =
"The call NamesMPackage[F] returns the list of objects names in string–format that have been saved as a package in datafile F of m–format; at that, it is presupposed that the objects are supplied by usages. In the absence of such objects the empty list is returned. The procedure is an useful expansion of procedures NamesNbPackage and NamesNbPackage1 in case of m–files."

LoadNameFromM::usage =
"The call LoadNameFromM[F, N] returns the Null, i.e. nothing, loading into the current session the evaluated definitions of an object N or the objects from a list N that were saved as the package with context in a datafile F of m–format."

RhsLhs::usage =
"The procedure call RhsLhs[x, y] depending on value {\"Lhs\", \"Rhs\"} of 2−nd argument y returns the left or right part of expression x accordingly in relation to operator Head[x] whereas the call RhsLhs[x, y, t] in addition through an undefined variable t returns operator Head[x] concerning which partition of expression x onto the left and right part was made. In addition, in case of impossibility of partition of an expression x onto parts the call of procedure RhsLhs is returned unevaluated."

RhsLhs1::usage =
"This procedure is a functionally equivalent modification of the procedure RhsLhs. The procedure call RhsLhs1[x, y] depending on value {\"Lhs\", \"Rhs\"} of 2−nd argument y returns the left or right part of expression x accordingly in relation to operator Head[x] whereas the call RhsLhs[x, y, t] in addition through an undefined variable t returns operator Head[x] concerning which partition of the expression x onto the left and right part was made. In addition, in case of impossibility of partition of an expression x onto parts the call of procedure RhsLhs1 is returned unevaluated."

ExtrCall::usage =
"The call ExtrCall[x, y] returns the True if the user block, function or module y contains the calls of a block/function/module x, and the False otherwise. If the call as argument x defines the list of blocks/functions/modules then sublist x of calls of of blocks/functions/modules of x that are contained in the object y is returned. In case the first optional argument x is absent, the call ExtrCall[y] returns the list of the system means, that compose definition of the user function, block or module y. Meanwhile, it must be kept in mind that the ExtrCall procedure correctly processes only unique objects, but not the objects of the same name, by returning on the last ones the value $Failed."

WhichN::usage =
"Simple procedure WhichN allows an arbitrary even number of arguments similar to standard function Which, otherwise returning the unevaluated call. In the rest the procedure WhichN is analogous to the standard function Which. Procedure appears useful enough in case of dynamic generation of a Which−object."

SeqToList1::usage = "The call of simple function SeqToList1[a, b, c, ...] returns the list of its actual arguments."

SetPathSeparator::usage =
"The call of procedure SetPathSeparator[x] determines a separator \"\\\" or \"/\" for paths to files/directories for the period of the current session with the package, returning a new separator x."

FileDirStForm::usage =
"The call of procedure FileDirStForm[x] returns a name or a path to a datafile x in the standard format − the symbols composing names of files and paths to them are coded in lower case, whereas as separators the standard one \"\\\" is used."

StandPath::usage =
"The call of procedure StandPath[x] returns the path to a datafile x or a directory in the standard format − the symbols composing names of files and paths to them are coded in lower case, whereas the standard separators \"\\\" are used as separators. Moreover, the procedure StandPath for testing of admissibility of argument x as a real path uses the procedure PathToFileQ that represents an independent interest and provides the correctness of processing of the paths containing the blank symbols."

StandStrForm::usage = "The function call StandStrForm[x] returns the result of converting of all letters of a string x into lowercase with replacements of symbols \"/\" onto symbols \"\\\"."

DirFilePaths::usage =
"The DirFilePaths procedure provides search in the given directory of chains of subdirectories and datafiles containing a string x as own components. The call DirFilePaths[x, y] returns the 2−element list whose first element is a list of full paths to subdirectories of a directory y which contain components x whereas the second element is the list of full paths to datafiles whose names coincide with a string x. In the absence of the second optional argument y the procedure call instead of it supposes BootDrive1[][[1]] <> \"*.*\"."

SeqToString::usage = "The call of simple function SeqToString[a, b, c, ...] returns the list of its actual arguments in string format."

ContextMfile::usage =
"The call ContextMfile[x] returns the context associated with a package, located in a m−file x, given by the name or full path to it."

ContextMfile1::usage =
"Meanwhile, the ContextMfile procedure provides search only of the first context in a m−file with a package whereas generally multiple contexts can be associated with a package. The ContextMfile1 procedure provides the solution of this question in case of multiple contexts. The procedure call ContextMfile1[x] returns the list of the contexts or single context associated with a datafile x of formats {\"m\", \"tr\"}, in case of lack of contexts the empty list, i.e. {} is returned. Furthermore, the additional tr−format allows to carry out search of contexts in the system datafiles containing contexts. Moreover, in case FileExistsQ[x] = False the search of a datafile x is done in file system of the computer as a whole."

ContextNBfile::usage =
"As distinct from procedure ContextMfile, the call ContextNBfile[x] returns the context associated with a package, located

in a datafile x of the format {\"cdf\", \"nb\"}, given by the name or full path to it. If a datafile x of the format {\"cdf\", \"nb\"} does not contain a context identifier, the call of procedure ContextNBfile[x] returns $Failed."

StrFromStr::usage =
"In a lot of cases at processing of string constructions it is necessary to take of them substrings, limited by the symbol {\"}, i.e. \"strings in strings\". This problem is solved by means of the procedure, whose call StrFromStr[x] returns the list of such substrings located in a string x; otherwise, the call StrFromStr[x] returns the empty list, i.e. {}."

Un::usage = "The simple function Un[x] is a software implementation of the standard function Unique[x]."

Unique1::usage = "The call Unique1[x, y] returns an unique name in string−format that depends on the second argument or its absence, at the same time assigning a value x to this name."

UniqueV::usage =
"The procedure call UniqueV[X, y] returns a name in string format \"Xn\" of an unique variable of the current session to which value y was ascribed, where X – a symbol, n – an integer and y – an arbitrary expression."

ListAssign::usage =
"The call ListAssign[x, y] provides assignment of values of a list x to the generated variables of format y$nnn, returning the nested list, whose first element defines the list of the generated variables \"y$nnn\" in string format, whereas the second − the list of values of the list which were assigned for corresponding variables of the first list."

ListAppValue::usage = "The call of simple function ListAppValue[x, y] provides assignment of a value y to each element of a list x."

OverLap::usage =
"The procedure call OverLap[x, y] returns the length of the overlap between x and y in linear time. This is defined to be the length of the longest suffix of x that is a prefix of y; in addition, if the third argument had been coded then through it the overlap is returned. In case of especial or erroneous situations the call is returned unevaluated."

ExpLocals::usage =
"The ExpLocals procedure is intended for extension of the list of local variables of blocks and modules. Expansion is defined by the list of strings defining names of new local variables and/or local variables with initial values ascribed to them. It must be kept in mind, elements of the given list need to be encoded in string−format in order to avoid assignment to them of values of the variables of the current session of the same name and/or calculations which are defined by initial values ascribed to them. The call ExpLocals[x, y] returns the list of local variables with initial values ascribed to to them in string−format from the list y by which local variables of a block or a module x should be expanded. In addition, generally this list can be less than the given list y (or at all empty) inasmuch as the variables which are available in object x as formal arguments or local variables are excluded from it. The resultant object keeps all options and attributes of initial object x."

ExtrExpr::usage =
"In a certain relation to the ExprOfStr procedure also the ExtrExpr procedure adjoins, whose call ExtrExpr[S, n, m] returns omnifarious correct expressions in string format which are contained in the substring of a string S limited by positions with numbers n and m. In the absence of correct expressions the empty list, i.e. {} is returned."

ExprOnLevels::usage =
"The call ExprOnLevels[x] returns the nested list whose sublists represent subexpressions of an expression x that are located on its levels starting with the first up to the last accordingly."

XOR1::usage =
"Operation 'x XOR1 y' with two positive integers x and y is defined as bit−by−bit operation XOR without a carrying over into the high−order digits with binary equivalents of the given integers x, y; in addition, a length l of binary representation is defined by length of representation of the maximal integer, i.e. l = max{|x|,|y|}. The call XOR1[{x1,x2,...,xp}] returns the result of application of the above operation XOR1 to integers x1,x2,...,xp."

ReprodXOR1::usage =
"The procedure call ReprodXOR1[S,n,m] returns 2−element list whose the first element defines the number of iterations of a structure with template size n that generates from a string S a configuration containing m copies of S while the second element defines real number of copies. The call with fourth optional argument g additionally through g (indefinite variable) returns the resultant configuration."

ReprodXOR11::usage =
"Procedure ReprodXOR1 provides simulation of dynamics in 1−HS of initial configurations containing symbols from alphabet A={0,1,2,...,a} where a is a prime no more then 7. While for case of arbitrary alphabet A allowing to use the initial configuration with symbols from alphabet A={0,1,2,...,a} (a – an arbitrary integer) as an initial configuration can be used a modification of the above procedure ReprodXOR1 whose call ReprodXOR11[S,n,m] returns two−element list whose the first element defines the number of steps of a structure with neighbourhood template size n which generates from a list S defining the initial configuration a configuration containing m copies of configuration S while the second element defines real number of such copies. At that, the procedure call ReprodXOR11[S,n,m,g] with fourth optional argument g additionally through actual argument g (indefinite variable) returns the list defining the resultant configuration."

SelfReprod::usage =
"A successful procedure call SelfReprod[c, n, p, m], implemented in the environment of system Mathematica, returns the number of iterations of a linear global transition function with neighbourhood index X={0,1,...,n–1} and alphabet A={0,1,...,p–1} (p – an apbitrary integer) that was required to generate m copies of an initial configuration c. In case of a rather long run of the procedure, it can be interrupted, by monitoring through the list {d, t} the reality of obtaining the required number of copies of configuration c, where d – number of iterations and t – quantity of initial configuration c."

SelfReprod1::usage =
"The procedure call SubConf[Ltf, Cf, p, w] on the basis of the local transition function which is defined by the list Ltf of parallel substitutions along with the finite configuration Cf, number of the demanded copies p and a finite configuration w returns 2–element list whose the first element defines number of steps used by the structure 1–HS(a, n) with local transition function Ltf for generation p copies of configuration Cf from an initial configuration w whereas the second element defines number of really obtained copies of configuration Cf."

HSD::usage =
"The procedure call HSD[A, dl, C, L, n, p] returns the configuration in the list form which is result of generation from configuration C on step p of a local transition function L given by the list of substitutions of a 1–HSD with alphabet A, the delays d and neighbourhood template size n."

ReprodHSD::usage =
"Procedure ReprodHSD allows experimentally to study in a structure 1–HSD with alphabet A, the delays d, neighbourhood template size n and a local transition function ltf given by the list of substitutions the self–reproductivity of an initial configuration Cf given in the list form. The procedure call ReprodHSD[A,dl,Cf,L,n,p,v] returns the 2–element list whose first element defines the demanded quantity of copies p of a finite configuration Cf given in the list form, and the second element defines the quantity of copies in the Moore sense of the configuration Cf that have been really obtained in process of generation of 1–HSD on interval out of no more than v steps."

ReprodHSM::usage =
"The procedure ReprodHSM has six formal arguments, namely: Cf – a finite configuration in the list format, n – the size of neighbourhood index of a structure, p – memory depth, m – modulus of congruence, g – number of the required copies of configuration Cf in generated configurations, v – the limiting number of steps of the structure. The call ReprodHSM[Cf, n, p, m, g, v] returns 2–element list whose first element is g and the second defines the real number of copies of the configuration Cf in generated configurations. At long non–performance of the condition (g) the procedure execution will end via v steps or can be interrupted manually."

ReprodHSM1::usage =
"In contrast to the above procedure ReprodHSM the call ReprodHSM1[Cf, n, p, m, g, v] returns 3–element list whose first element is g, the second element defines the real number of copies of the configuration Cf, and third element – number of the demanded steps of the structure."

ReprodHSwVni::usage =
"For computer research of self–reproducibility in 1–dimensional homogeneous structures with variable neighbourhood index, the procedure ReprodHSwVni can be useful enough. The call ReprodHSwVni[A, Cf, m, g, v] returns 3–element list whose first element defines the desired number (g) of copies of a configuration Cf in an alphabet A, the second defines the real number of copies of Cf, and third element defines number of structure steps. Other arguments of the ReprodHSwVni are analogous to the arguments of same name in the above procedure ReprodHSM."

SubConf::usage =
"The procedure call SubConf[Ltf, Cf, p, w] on the basis of the local transition function which is defined by the list Ltf of parallel substitutions along with the finite configuration Cf, number of the demanded copies p and a finite configuration w returns 2–element list whose the first element defines number of steps used by the structure 1–HS(2, n) with local transition function Ltf for generation p copies of configuration Cf from an initial configuration w whereas the second element defines number of really obtained copies of configuration Cf."

ReadFullFile::usage =
"The call ReadFullFile[F] returns the contents of a datafile F with change of symbols \"\\r\\n\" of carriage return and form feed onto the symbols \"\" (empty string); if the datafile F is absent in file system of a computer, the call returns the value $Failed. In addition, the call ReadFullFile[F, y] with the second optional argument y – an indefinite variable – returns through it the full name or full path to the datafile F; if y is a string, it replaces in the returned contents of the datafile F symbols \"\\r\\n\" onto a string y; In case of other types of argument y change of symbols \"\\r\\n\" onto the symbols \"\" are done."

ReadFullFile1::usage =
"The call ReadFullFile1[F] returns the contents of a datafile F with change of symbols \"\\r\\n\" of carriage return and form feed onto the symbols \"\" (empty string); if the datafile F is absent in file system of a computer, the call is returned unevaluated. In case of inadmissible type of a datafile the call returns the empty string."

ToLTF::usage =
"The procedure call ToLTF[A, n, G, p] returns the list of parallel substitutions defining a local transition function of 1–HS

```
      with alphabet A, template size n and p=Length[A]; at that, G is the name of a function defined as G[x1,...,xn]."

SEQ::usage =
  "Procedure SEQ serves as an analogue of the same procedure of the package Maple. The procedure call SEQ[x, y, z] returns the
      list of values x[y] where y changes or in limits z=m;;n, or in limits z=m;;n;;p with step p; in addition, values {m,
      n, p} can accept only positive numerical values; at m <= n the value p is considered positive, otherwise negative.
      In case of zero or negative value of 3−rd argument the procedure call SEQ[x, y, z] is returned unevaluated."



Begin["`ExtrExpr`"]
ExtrExpr[S_ /; StringQ[S], N_ /; IntegerQ[N], M_ /; IntegerQ[M]] :=
  Module[{a = StringLength[S], b, c, h = {}, k, j}, If[! (1 ≤ M ≤ a && N ≤ M), {},
      b = StringTake[S, {N, M}]; Do[ Do[c = Quiet[StringTake[b, {j, M − N − k + 1}]]; If[ExpressionQ[c], AppendTo[h, c];
            Break[], Null], {k, 0, M − N + 1}], {j, 1, M − N}];
      DeleteDuplicates[Select[Map[StringTrim2[#, {"+", "−", " " , "_"}, 3] &, h], ! MemberQ[{"", " "}, #] &]]]]
End[]

Begin["`ExtrCall`"]
ExtrCall[z___, y_ /; BlockFuncModQ[y]] := Module[{b, p, g, x, a = Join[CharacterRange["A", "Z"], CharacterRange["a", "z"]]},
    If[{z} == {}, p = PureDefinition[y]; If[ListQ[p], Return[$Failed]];
      g = ExtrVarsOfStr[p, 2]; g = Select[g = Map[" " <> # <> "[" &, g], ! StringFreeQ[p, #] &];
      g = Select[Map[If[SystemQ[p = StringTake[#, {2, −2}]], p] &, g], ! SameQ[#, Null] &];
      If[Length[g] == 1, g[[1]], g],
      b[x_] := Module[{c = DefFunc3[ToString[y]], d, h, k = 1, t = {}}, h = StringPosition[c, ToString[x] <> "["];
          If[h == {}, Return[False], d = Map[First, h]; For[k, k ≤ Length[d], k++,
              AppendTo[t, If[! MemberQ[a, StringTake[c, {d[[k]] − 1, d[[k]] − 1}]], True, False]]]; t[[1]]];
      If[! ListQ[z], b[z], Select[z, b[#] &]]]
End[]

Begin["`ExtensionHeading`"]
ExtensionHeading[x_ /; BlockFuncModQ[x], y__] :=
  Module[{a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}], c = If[StringQ[x], Symbol[x], x], d, atr = Attributes[x]},
    ClearAttributes[x, atr]; d = Args[x]; Clear[x]; d = Map[Join[#, {y}] &, d]; d = Map[ToString1, Map[c @@ # &, d]];
    Do[ToExpression[StringReplace[a[[k]], b[[k]] → d[[k]], 1]], {k, 1, Length[a]}]; Map[ToExpression, d]; ReduceArgs[x];
    SetAttributes[x, atr]]
End[]

Begin["`QBlockMod`"]
QBlockMod[x_] := Module[{a = Flatten[{PureDefinition[x]}], b, c = True, k = 1},
    If[MemberQ[{"System", $Failed}, a[[1]]], False, b = Flatten[{HeadPF[x]}];
      While[k ≤ Length[a], If[! SuffPref[StringReplace[a[[k]], b[[k]] <> " := " → "", 1], {"Module[{", "Block[{"}, 1], c = False;
          Break[]];
        k++]; c]]
End[]

Begin["`RenDirFile`"]
RenDirFile[x_, y_ /; StringQ[y]] :=
  Module[{a, c = {"/", "\\"}, b = StringTrim[x, {"/", "\\"}]}, If[FileExistsQ[b] || DirectoryQ[b], c = StringTrim[y, c];
      a = If[FileExistsQ[b], RenameFile, RenameDirectory];
      If[PathToFileQ[b] && PathToFileQ[c] && FileNameSplit[b][[1 ;; −2]] == FileNameSplit[c][[1 ;; −2]],
        Quiet[Check[a[b, c], "Directory/datafile <" <> y <> "> already exists"]], If[PathToFileQ[b] && ! PathToFileQ[c],
          Quiet[Check[a[b, FileNameJoin[Append[FileNameSplit[b][[1 ;; −2]], StringReplace[c, {"/" −> "", "\\" −> ""}]]]],
              "Directory/datafile <" <> y <> "> already exists"]], If[! PathToFileQ[b] && ! PathToFileQ[c],
            Quiet[Check[a[b, StringReplace[c, {"/" −> "", "\\" −> ""}]], "Directory/datafile <" <> y <> "> already exists"]],
            $Failed]]], Defer[RenDirFile[x, y]]]]
End[]

Begin["`CharacterQ`"]
CharacterQ[c_] := ! NumberQ[c] && StringLength[ToString[c]] == 1
End[]

Begin["`Characters1`"]
Characters1[x_] := Characters[x // ToString]
End[]

Begin["`$Load$Files$`"]
$Load$Files$ := 78
End[]
```

```
Begin["`FileExistsQ1`"]
FileExistsQ1[x__ /; StringQ[{x}[[1]]]] := Module[{a = SearchFile[{x}[[1]]], b = {x}},
    If[a == {}, False, If[Length[b] == 2 && ! HowAct[b[[2]]], ToExpression[ToString[b[[2]]] <> " = " <> ToString1[a]], Null];
      True]]
End[]

Begin["`InOutFiles`"]
InOutFiles[] := Module[{a = Map[ToString1, Streams[]][[3 ;; -1]], b = {}, c = {}},
    If[a == {}, {}, Map[If[SuffPref[#, "InputStream[", 1], AppendTo[b, StringTake[#, {13, Flatten[StringPosition[#, "\", "]][[1]]}]],
          AppendTo[c, StringTake[#, {14, Flatten[StringPosition[#, "\", "]][[1]]}]]] &, a];
      Map[ToExpression, {b, c}]]]
End[]

Begin["`OpenFileQ`"]
OpenFileQ[x_ /; StringQ[x], y___] := Module[{a = InOutFiles[], b = {}, c = {}},
    If[a == {}, False, Map[If[ToLowerCase[StandPath[x]] == ToLowerCase[StandPath[#]], AppendTo[b, #]] &, a[[1]]];
      Map[If[ToLowerCase[StandPath[x]] == ToLowerCase[StandPath[#]], AppendTo[c, #]] &, a[[2]]];
      If[{b, c} == {{}, {}}, False, If[{y} ≠ {} && SymbolQ[y] && ! HowAct[y], y = {b, c}];
        True]]]
End[]

Begin["`RestoreDelPackage`"]
RestoreDelPackage[f_ /; StringQ[f] && MemberQ[{"cdf", "m", "mx", "nb"}, FileExtension[f]], s_String] :=
  Module[{a = FilesDistrDirs[BootDrive[][[1]] <> ":\\Recycler"][[1 ;; -2]], b, c, d = {}, k = 1, j, p, t, h = {}},
    For[k, k ≤ Length[a], k++, b = a[[k]];
      For[j = 2, j ≤ Length[b], j++, If[SameQ[FileExtension[b[[j]]], FileExtension[f]] &&
          SameQ[ContextFromFile[b[[1]] <> b[[j]]], s], AppendTo[d, b[[1]] <> b[[j]]]]]];
    If[d == {}, $Failed, {b = {}, k = 1}; For[k, k ≤ Length[d], k++, AppendTo[b,
        CopyFile[d[[k]], CreateDirectory[Directory[] <> "\\$" <> ToString[k] <> "\\" <> FileNameTake[d[[k]]]]];
      DeleteFile[d[[k]]]; a = DirectoryName[d[[k]]]; If[! SameQ[a, ""], DeleteDirectory[a, DeleteContents → True]]; b]]
End[]

Begin["`RestoreDelFile`"]
RestoreDelFile[f_ /; StringQ[f] || ListQ[f], r_ /; StringQ[r]] :=
  Module[{b = ToString[Unique["ag"]], c, p = $ArtKr$, t = Map[StandPath, Flatten[{f}]], h}, ClearAll[$ArtKr$];
    If[FileExistsQ1["$recycle.bin", $ArtKr$], d = $ArtKr$[[1]];
      $ArtKr$ = p, Return[$Failed]]; Run["Dir " <> d <> "/B/S/L > " <> b];
    If[EmptyFileQ[b], $Failed, Quiet[CreateDirectory[r]]; c = ReadList[b, String]]; DeleteFile[b];
    h[x_, y_] := If[FileExistsQ[x] && SuffPref[x, "\\" <> y, 2], CopyFileToDir[x, StandPath[r]], "Null"];
    c = Select[Flatten[Outer[h, c, t]], ! SameQ[#, "Null"] &]]
End[]

Begin["`Restart`"]
Restart[x___] := Module[{}, Map[{Quiet[ClearAttributes[#, Protected]], Quiet[Remove[#]]} &,
      If[{x} ≠ {} && MemberQ[Select[Contexts[], StringCount[#, "`"] == 1 &], x],
        {Write["$590$", 90], Close["$590$"], Names[x <> "*"]}[[-1]], Names["`*"]]];
    If[Quiet[Check[Read["$590$"], 500]] == 90, DeleteFile[Close["$590$"]];
      Unprotect[$Packages, Contexts]; {$Packages, $ContextPath} = Map[Complement[#, {x}] &, {$Packages, $ContextPath}];
      Contexts[] = Select[Contexts[], Quiet[Check[StringTake[#, {1, StringLength[x]}], "Null"]] ≠ x &];
      Protect[$Packages, Contexts], Null];]
End[]

Begin["`TypeActObj`"]
TypeActObj[] := Module[{a = Names["`*"], b = {}, c, d, h, p, k = 1}, Quiet[For[k, k ≤ Length[a], k++, h = a[[k]];
        c = ToExpression[h]; p = "0" <> ToString[Head[c]];
        If[! StringFreeQ[h, "$"] || p === Symbol && Definition[c] === Null, Continue[],
          b = Append[b, {h, If[ProcQ[c], "0Procedure", If[Head1[c] === Function, "0Function", p]]}]]]];
    a = Quiet[Gather1[Select[b, ! #[[2]] === Symbol &], 2]];
    a = ToExpression[
        StringReplace[ToString1[Map[DeleteDuplicates, Map[Sort, Map[Flatten, a]]]], "AladjevProcedures`TypeActObj`" -> ""]];
    Append[{}, Do[a[[k]][[1]] = StringTake[a[[k]][[1]], {2, -1}], {k, Length[a]}]]; a]
End[]

Begin["`Gather2`"]
Gather2[x_ /; ListQ[x]] := Module[{a = Select[Gather[Flatten[x]], Length[#] > 1 &], b = {}},
    If[a == {}, Return[{}], Do[b = Append[b, {a[[k]][[1]], Length[a[[k]]]}], {k, Length[a]}]];
    If[Length[b] > 1, b, First[b]]]
End[]
```

```
Begin["`GatherStrLetters`"]
GatherStrLetters[x_ /; StringQ[x]] :=
  StringJoin[Map[StringJoin, Map[FromCharacterCode, Gather[ToCharacterCode[x], #1 == #2 &]]]]
End[]

Begin["`BitGet1`"]
BitGet1[x___, n_ /; IntegerQ[n] && n ≥ 0, p_ /; IntegerQ[p] && p > 0 || ListQ[p]] :=
  Module[{b = 1, c = {}, d, a = ToExpression[Characters[IntegerString[n, 2]]], h = If[ListQ[p], p, {p}]},
    For[b, b ≤ Length[a], b++, c = Append[c, If[MemberQ[h, b], a[[b]], Null]]];
    If[! HowAct[x], x = Length[a], Null]; Select[c, ToString[#] ≠ "Null" &]]
End[]

Begin["`WhichN`"]
WhichN[x__] := Module[{a = {x}, c = "Which[", d, k = 1}, d = Length[a]; If[OddQ[d], Defer[WhichN[x]],
      ToExpression[For[k, k ≤ d, k++, c = c <> ToString[a[[k]]] <> ","]; StringTake[c, {1, -2}] <> "]"]]]
End[]

Begin["`Ind`"]
Ind[x_] := Module[{a = ToString[InputForm[x]], b, c}, b = Flatten[StringPosition[a, {"[", "]"}]]; If[b == {} || StringTake[a, -1] != "]", x,
      Quiet[Check[Map[ToExpression, {StringTake[a, {1, b[[1]] - 1}], "{" <> StringTake[a, {b[[1]] + 1, b[[-1]] - 1}] <> "}"}], x]]]
End[]

Begin["`StringPosition1`"]
StringPosition1[x_ /; StringQ[x], y_] :=
  Module[{b = {}, c = 1, d, a = Flatten[{If[ListQ[y], Map[ToString, y], ToString[y]]}]}, For[c, c ≤ Length[a], c++, d = a[[c]];
      AppendTo[b, {d, Flatten[StringPosition[x, d]]}]]; Sort[Select[b, #[[2]] ≠ {} &]]]
End[]

Begin["`SubsPosSymb`"]
SubsPosSymb[x_ /; StringQ[x], n_ /; PosIntQ[n], y_ /; ListQ[y] && DeleteDuplicates[Map[CharacterQ, y]] == {True},
    z_ /; z == 0 || z == 1] := Module[{a = "", k = n, b}, If[n > StringLength[x],
      Return[Defer[SubsPosSymb[x, n, y, z]]], While[If[z == 0, k ≥ 1, k <= StringLength[x]], b = StringTake[x, {k, k}];
        If[! MemberQ[y, b], If[z == 0, a = b <> a, a = a <> b], Break[]]; If[z == 0, k--, k++]]; a]]
End[]

Begin["`StrOfSymb`"]
StrOfSymb[x_ /; StringQ[x], y_ /; PureFuncQ[y]] := StringJoin[Select[Characters[x], y]]
End[]

Begin["`TwoHandQ`"]
TwoHandQ[x__] := Module[{a = ToString[InputForm[{x}[[1]]]],
      b = {"+", ">=", "<=", "&&", "||", "-", "^", "**", "<", "==", "!=", ">", "->"}, c, d = {x}}, c = StringPosition[a, b];
    If[StringFreeQ[a, "->"] && StringFreeQ[a, ">="] && Length[c] > 2 || Length[c] == 0,
      False, If[Length[d] > 1 && ! HowAct[d[[2]]] && ! ProtectedQ[d[[2]]],
        ToExpression[ToString[d[[2]]] <> "=" <> ToString[Head[{x}[[1]]]]], Return[Defer[TwoHandQ[x]]]];
      True]]
End[]

Begin["`GC`"]
GC[x_] := ToExpression[StringReplace[ToString[ToCharacterCode[ToString[InputForm[x]]]], {"{" → "", "," → "", "}" → ""}]]
End[]

Begin["`ExprOfStr`"]
ExprOfStr[x_ /; StringQ[x], n_ /; IntegerQ[n] && n > 0, m_ /; MemberQ[{-1, 1}, m], L_ /; ListQ[L]] :=
  Module[{a = "", b, k}, If[n ≥ StringLength[x], Return[Defer[ExprOfStr[x, n, m, L]]], Null];
    For[k = n, If[m == -1, k ≥ 1, k ≤ StringLength[x]], If[m == -1, k--, k++],
      If[m == -1, a = StringTake[x, {k, k}] <> a, a = a <> StringTake[x, {k, k}]];
      If[! SyntaxQ[a], Null, If[If[m == -1, k == 1, k = StringLength[x]] ||
            MemberQ[L, Quiet[StringTake[x, If[m == -1, {k - 1, k - 1}, {k + 1, k + 1}]]]], Return[a], Null]]];
    $Failed]
End[]

Begin["`HeadToCall`"]
HeadToCall[h_ /; HeadingQ[h]] :=
  Module[{a = HeadName[h], b}, b = "{" <> StringTake[StringReplace[h, a <> "[" → "", 1], {1, -2}] <> "}";
    b = StrToList[b];
    b = Select[b, ! StringFreeQ[#, "_"] &];
    b = Map[StringTake[#, {1, Flatten[StringPosition[#, "_"]][[1]] - 1}] &, b];
```

```
    a <> "[" <> StringTake[ToString[b], {2, −2}] <> "]"]
End[]

Begin["`ExprOfStr1`"]
ExprOfStr1[x_ /; StringQ[x], n_ /; IntegerQ[n], p_ /; MemberQ[{−1, 1}, p]] :=
  Module[{a = Quiet[StringTake[x, {n, n}]], b = StringLength[x], k},  If[n ≥ 1 && n ≤ b && ! SameQ[a, $Failed],
      For[k = If[p == −1, n − 1, n + 1], If[p == −1, k ≥ 1, k ≤ b], If[p == −1, k−−, k++], If[! SyntaxQ[a],
        If[p == −1, a = StringTake[x, {k, k}] <> a, a = a <> StringTake[x, {k, k}]]; Continue[], Return[a]]]; $Failed, $Failed]]
End[]

Begin["`ListableQ`"]
ListableQ[x_] := MemberQ[Quiet[Check[Attributes[x], {}]], Listable]
End[]

Begin["`LevelsList`"]
LevelsList[x_ /; ListQ[x]] := Module[{a = ToString[x], b = 1}, Do[a = StringTake[a, {2, −2}];
      If[! SyntaxQ[a], Break[], b++], {k, 1, Infinity}]; b]
End[]

Begin["`ListableC`"]
ListableC[x_ /; SystemQ[x] || ProcQ[x] || QFunction[ToString[x]]] :=
  Module[{a = Attributes[x], b = If[MemberQ[Attributes[x], Protected], "Protected", "Null"]}, If[MemberQ[a, Listable], a, Unprotect[x];
      SetAttributes[x, Listable]; If[b == "Protected", Protect[x]]; Attributes[x]]]
End[]

Begin["`ListAssignP`"]
ListAssignP[x_ /; ListQ[x], n_ /; PosIntQ[n] || PosIntListQ[n], y_] :=
  Module[{a = DeleteDuplicates[Flatten[{n}]], b = Flatten[{y}], c, k = 1},
    If[a[[−1]] > Length[x], Return[Defer[ListAssignP[x, n, y]]], c = Min[Length[a], Length[b]]];
    While[k ≤ c, Quiet[Check[ToExpression[ToString[x[[a[[k]]]]] <> " = " <> ToString1[If[ListQ[n], b[[k]], y]]], Null]];
      k++];
    If[NestListQ1[x], x[[−1]], x]]
End[]

Begin["`ListNumericQ`"]
ListNumericQ[x_ /; ListQ[x]] := SameQ[DeleteDuplicates[Map[NumericQ, N[Flatten[x]]]], {True}]
End[]

Begin["`IntegerListQ`"]
IntegerListQ[x_] := ListQ[x] && Length[Select[Flatten[x], IntegerQ[#] &]] == Length[Flatten[x]]
End[]

Begin["`Split1`"]
Split1[x_ /; ListQ[x], y_] := Module[{a, b, c = {}, d, h, k = 1},
    If[MemberQ3[x, y] || MemberQ[x, y], a = If[ListQ[y], Sort[Flatten[Map[Position[x, #] &, y]]], Flatten[Position[x, y]]];
      h = a; If[a[[1]] ≠ 1, PrependTo[a, 1]]; If[a[[−1]] ≠ Length[x], AppendTo[a, Length[x]]]; d = Length[a];
      While[k ≤ d − 1, AppendTo[c, x[[a[[k]] ;; If[k == d − 1, a[[k + 1]], a[[k + 1]] − 1]]]]; k++];
      If[h[[−1]] == Length[x], AppendTo[c, {x[[−1]]}]]; c, x]]
End[]

Begin["`SplitList`"]
SplitList[L_ /; ListQ[L], x_] := Module[{a = Flatten[{x}], b = ToString[Unique["$a$"]], c, d, h}, c = Map[ToString[#] <> b &, a];
    d = StringJoin[Map[ToString[#] <> b &, L]]; h = Select[StringSplit[d, c], # ≠ "" &]; h = Map[StringReplace[#, b → ","] &, h];
    h = ToExpression[Map["{" <> StringTake[#, {1, −2}] <> "}" &, h]]; Remove[b]; If[Length[h] == 1, h[[1]], h]]
End[]

Begin["`SplitList1`"]
SplitList1[x_ /; ListQ[x], y_ /; ListQ[y], z_ /; ListQ[z]] :=
  Module[{c, a = Map12[ToString, {x, y, z}], b = ToString[Unique["$"]]}, c = Map11[StringJoin, a, b];
    c = Map[StringJoin, c];
    c = SubsString1[c[[1]], {c[[2]], c[[3]]}, StringQ[#] &, 0]; ToExpression[Map11[StringSplit, c, b]]]
End[]

Begin["`Intersection1`"]
Intersection1[x__ /; DeleteDuplicates[Map[ListQ[#] &, {x}]] == {True}, Ig_ /; MemberQ[{False, True}, Ig]] :=
  Module[{b = Length[{x}], c = {}, d = {}}, Do[AppendTo[c, Map[StringQ, {x}[[j]]]], {j, 1, b}];
    If[DeleteDuplicates[Flatten[c]] ≠ {True}, $Failed,
      If[Ig == False, Intersection[x], Do[AppendTo[d, Map[{j, #, ToUpperCase[ToString[#]]} &, {x}[[j]]]], {j, 1, b}];
        c = Map[DeleteDuplicates, Gather[Flatten[Join[d], 1], #1[[3]] == #2[[3]] &]]; c = Flatten[Select[c, Length[#] ≥ b &], 1];
```

```
        c = If[DeleteDuplicates[Map[#[[1]] &, c]] ≠ Range[1, b], {}, DeleteDuplicates[Map[#[[2]] &, c]]]]]
End[]

Begin["`IntersectStrings`"]
IntersectStrings[x__ /; DeleteDuplicates[Flatten[Map[StringQ, {x}]]] == {True}, Ig_ /; MemberQ[{False, True}, Ig]] :=
  Module[{a = {}, b = If[Length[{x}] == 1, {x, x}, {x}], c, d = {}, h}, If[x == "", {}, c = Length[b];
      h = Subsets[b, 2][[c + 2 ;; -1]];
      Do[{ClearAll[p], LongestCommonSubsequence1[h[[k]][[1]], h[[k]][[2]], Ig, p], AppendTo[a, p]}, {k, 1, Length[h]}];
      h = DeleteDuplicates[ToExpression["Intersection1[" <> StringTake[ToString1[a], {2, -2}] <> "," <> ToString[Ig] <> "]"]];
      Flatten[Map[Sort, Gather[h, StringLength[#1] == StringLength[#2] &]]]]]
End[]

Begin["`ElemOnLevels`"]
ElemOnLevels[x_ /; ListQ[x]] := Module[{a, b, c, d, p = 0, k, j = 1}, If[! NestListQ1[x], Flatten[{0, x}], {a, c, d} = {x, {}, {}};
      While[NestListQ1[a], b = {p++};
        For[k = 1, k ≤ Length[a], k++, If[! ListQ[a[[k]]], AppendTo[b, a[[k]]];
            AppendTo[c, k]]];
        AppendTo[d, b]; a = Flatten[Delete[a, Map[List, c]], 1];
        {b, c} = {{}, {}};
        j++]; AppendTo[d, Flatten[{p++, a}]]]]
End[]

Begin["`ElemsOnLevelList`"]
ElemsOnLevelList[x_ /; ListQ[x]] := Module[{a = LevelsOfList[x], b = Flatten[x], c = {}, d = {}, k = 1, t},
    Do[AppendTo[c, {a[[t = k++]], b[[t]]}], {Length[a]}]; c = Map[Flatten, Gather[c, #1[[1]] == #2[[1]] &]]; k = 1;
    Do[AppendTo[d, {c[[t = k++]][[1]], Select[c[[t]], EvenQ[Flatten[Position[c[[t]], #]][[1]]] &]}], {Length[c]}]; d]
End[]

Begin["`LevelsOfList`"]
LevelsOfList[x_ /; ListQ[x]] := Module[{L, L1, t, p, k, h, g}, If[x == {}, {}, If[! NestListQ1[x], Map[#^0 &, Range[1, Length[x]]],
        If[FullNestListQ[x], Map[#^0 &, Range[1, Length[Flatten[x]]]],
          {p, h, L, L1, g} = {1, FromCharacterCode[2], x, {}, {}}; ClearAll[t];
          Do[For[k = 1, k <= Length[L], k++, If[! ListQ[L[[k]]] && ! SuffPref[ToString[L[[k]]], h, 1], AppendTo[g, 0];
                AppendTo[L1, h <> ToString[p]], If[! ListQ[L[[k]]] && ! SuffPref[ToString[L[[k]]], h, 1], AppendTo[g, 0];
                  AppendTo[L1, L[[k]]], AppendTo[g, 1]; AppendTo[L1, L[[k]]]]]];
            If[! MemberQ[g, 1], L1, L = Flatten[L1, 1]; L1 = {}; g = {}; p++], {Levels[x, t]; t}];
          ToExpression[Map[StringDrop[#, 1] &, L]]]]]
End[]

Begin["`ListExprHeadQ`"]
ListExprHeadQ[x_ /; ListQ[x], h_] := Length[x] == Length[Select[x, Head[#] === h &]]
End[]

Begin["`$ProcType`"]
$ProcType := ToString[Stack[][[1]]]
End[]

Begin["`SubsStrLim`"]
SubsStrLim[x_ /; StringQ[x], y_ /; StringQ[y] && StringLength[y] == 1, z_ /; StringQ[z] && StringLength[z] == 1] :=
  Module[{a, b = x <> FromCharacterCode[6], c = y, d = {}, p, j, k = 1, n, h},
    If[! StringFreeQ[b, y] && ! StringFreeQ[b, z], a = StringPosition[b, y];
      n = Length[a]; For[k, k ≤ n, k++, p = a[[k]][[1]]; j = p;
        While[h = Quiet[StringTake[b, {j + 1, j + 1}]]; h ≠ z, c = c <> h; j++]; c = c <> z;
        If[StringFreeQ[StringTake[c, {2, -2}], {y, z}], AppendTo[d, c]]; c = y]];
    Select[d, StringFreeQ[#, FromCharacterCode[6]] &]]
End[]

Begin["`StringDependAllQ`"]
StringDependAllQ[s_ /; StringQ[s], a_ /; StringQ[a] || ListQ[a] && ! MemberQ[Map[StringQ, a], False]] :=
  DeleteDuplicates[Map[StringFreeQ[s, #] &, If[StringQ[a], {a}, a]]] == {False}
End[]

Begin["`SubsStrLim1`"]
SubsStrLim1[x_ /; StringQ[x], y_ /; StringQ[y] && StringLength[y] == 1, z_ /; StringQ[z] && StringLength[z] == 1] :=
  Module[{a, b = x <> FromCharacterCode[6], c = y, d = {}, p, j, k = 1, n, m, h},
    If[! StringFreeQ[b, y] && ! StringFreeQ[b, z], a = StringPosition[b, y];
      {n, m} = {Length[a], StringLength[x]};
      For[k, k ≤ n, k++, p = a[[k]][[1]];
        For[j = p + 1, j ≤ m, j++,
```

```
            h = StringTake[b, {j, j}];
            If[h ≠ z, c = c <> h, c = c <> z;
              If[Length[DeleteDuplicates[Map10[StringCount, c, {y, z}]]] == 1, AppendTo[d, c];
                c = y;
                Break[], Continue[]]]]]]; d]
End[]

Begin["`PosIntQ`"]
PosIntQ[n_] := If[IntegerQ[n] && n > 0, True, False]
End[]

Begin["`ShortPureFuncQ`"]
ShortPureFuncQ[x_] := PureFuncQ[ToExpression[If[StringQ[x], x, ToString[x]]]] &&
    StringTake[StringTrim[ToString[If[StringQ[x], ToExpression[x], ToString[x]]]], {-1, -1}] == "&"
End[]

Begin["`PosIntListQ`"]
PosIntListQ[L_ /; ListQ[L]] := SameQ[DeleteDuplicates[Map[PosIntQ, L]], {True}]
End[]

Begin["`ExtProgExe`"]
ExtProgExe[x_ /; StringQ[x], y_ /; StringQ[y], h___] := Module[{a = "$TempFile$", b = Directory[] <> "\\" <> x, c},
    Empty::datafile = "Datafile $TempFile$ is empty; the datafile had been deleted.";
    If[FileExistsQ[b], c = Run[x, " ", y, " > ", a], c = LoadExtProg[x];
      If[c === $Failed, Return[$Failed]]; c = Run[x, " ", y, " > ", a];
      If[{h} ≠ {}, DeleteFile[b]]];
    If[c ≠ 0, DeleteFile[a];
      $Failed, If[EmptyFileQ[a], DeleteFile[a]; Message[Empty::datafile], Directory[] <> "\\" <> a]]]
End[]

Begin["`MfilePackageQ`"]
MfilePackageQ[x_] := If[FileExistsQ[x] && FileExtension[x] == "m",
    StringDependAllQ[ReadFullFile[x], {"(* ::Package:: *)", "(* ::Input:: *)", "::usage", "BeginPackage[\"", "EndPackage[]"}], False]
End[]

Begin["`PackageUsages`"]
PackageUsages[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] :=
  Module[{a = StringSplit[ReadString[x], {"(**)", "*)\r\n(*"}], b, c, d, f},
    b = Select[a, ! StringFreeQ[#, {"::usage=", "::usage = "}] &]; c = FileNameSplit[x];
    d = FileBaseName[c[[-1]]] <> "_Usages.txt";
    f = FileNameJoin[Join[c[[1 ;; -2]], {d}]];
    Map[{WriteString[f, StringReplace[#, "::usage" -> ""]], WriteString[f, "\n\n"]} &, b]; Close[f]]
End[]

Begin["`CountOptions`"]
CountOptions[h___] := Module[{a = SystemOptions[], b = {}, d, c = 1, k}, While[c ≤ Length[a], d = a[[c]];
      AppendTo[b, If[ListQ[Part[d, 2]], {Part[d, 1], Length[Part[d, 2]]}, d]];
      c++]; b = Flatten[Gather[b, Head[#1] == Head[#2] &], 1];
    If[{h} == {}, b, If[HowAct[h], Defer[CountOptions[h]],
        d = 0; Do[If[ListQ[b[[k]]], d = d + b[[k]][[2]], d = d + 1], {k, Length[b]}]]; {h} = {d}; b]]
End[]

Begin["`RepStandFunc`"]
RepStandFunc[x_ /; StringQ[x], y_ /; SymbolQ[y], z__] :=
  Module[{a = Unique[y], b}, ToExpression[StringReplace[x, ToString[y] <> "[" → ToString[a] <> "[", 1]]; b = a[z];
    Remove[a]; b]
End[]

Begin["`SysUserSoft`"]
SysUserSoft[x_ /; BlockFuncModQ[x]] := Module[{b, s = {}, u = {}, h = Args[x, 6], c,
      a = Flatten[{PureDefinition[x]}][[1]], d = If[QFunction[x], {}, LocalsGlobals1[x]]}, b = ExtrVarsOfStr[a, 2, 90];
    c = Select[b, ! MemberQ[Flatten[{ToString[x], h, "True", "False", "$Failed", Quiet[d[[1]]], Quiet[d[[3]]]}], #] &];
    Map[If[Quiet[SystemQ[#]], AppendTo[s, #], If[BlockFuncModQ[#], AppendTo[u, #]]] &, c]; c = Map[Gather, {s, u}];
    c = {Map[Flatten[#] &, Map[{#, Length[#]} &, c[[1]]]], Map[Flatten[#] &, Map[{#, Length[#]} &, c[[2]]]]};
    c = {Map[DeleteDuplicates[#] &, c[[1]]], Map[DeleteDuplicates[#] &, c[[2]]]}; If[Flatten[c] == {}, {}, c]]
End[]

Begin["`SetAttributes1`"]
SetAttributes1[x_, y_] := ToExpression["SetAttributes[SetAttributes, Listable]; SetAttributes[" <>
```

```
      ToString[x] <> ", " <> ToString[y] <> "]; ClearAttributes[SetAttributes, Listable]"]
End[]

Begin["`RemovePackage`"]
RemovePackage[x_ /; ContextQ[x]] := Module[{a = CNames[x], b = ClearAttributes[{$Packages, Contexts}, Protected]},
    Quiet[Map[Remove, a]]; $Packages = Select[$Packages, StringFreeQ[#, x] &];
    Contexts[] = Select[Contexts[], StringFreeQ[#, x] &]; SetAttributes[{$Packages, Contexts}, Protected];
    $ContextPath = Select[$ContextPath, StringFreeQ[#, x] &];]
End[]

Begin["`RemoveContext`"]
RemoveContext[at_ /; ContextQ[at], x__] :=
  Module[{a = {}, b, c = {}, d = Map[ToString, {x}], f = "$Art27$Kr20$", Attr}, d = Intersection[CNames[at], d];
    b = Flatten[Map[PureDefinition, d]]; If[b == {}, $Failed, Attr := Map[{#, Attributes[#]} &, d];
      Do[AppendTo[a, StringReplace[b[[k]], at <> d[[k]] <> "`" -> ""]], {k, 1, Length[d]}];
      Write[f, a]; Close[f]; Do[AppendTo[c, at <> d[[k]]], {k, 1, Length[d]}];
      c = Flatten[c]; Map[{ClearAttributes[#, Protected], Remove[#]} &, d]; Map[ToExpression, Get[f]];
      DeleteFile[f]; Map[ToExpression["SetAttributes[" <> #[[1]] <> "," <> ToString[#[[2]]] <> "]"] &, Attr]; ]]
End[]

Begin["`DeletePackage`"]
DeletePackage[x_] := Module[{a}, If[! MemberQ[$Packages, x], $Failed, a = Names[x <> "*"];
      Map[ClearAttributes[#, Protected] &, Flatten[{"$Packages", "Contexts", a}]]; Quiet[Map[Remove, a]];
      $Packages = Select[$Packages, # ≠ x &]; $ContextPath = Select[$ContextPath, # ≠ x &];
      Contexts[] = Select[Contexts[], StringCount[#, x] == 0 &]; Quiet[Map[Remove, Names[x <> "*"]]];
      Map[SetAttributes[#, Protected] &, {"$Packages", "Contexts"}];]]
End[]

Begin["`DelOfPackage`"]
DelOfPackage[x_ /; ContextQ[x],
    y_ /; SymbolQ[y] || (ListQ[y] && DeleteDuplicates[Map[SymbolQ, y]] == {True}), z___] := Module[{a, b, c},
    If[! MemberQ[$Packages, x], $Failed, If[Set[b, Intersection[Names[x <> "*"], a = Map[ToString, Flatten[{y}]]]] == {}, $Failed,
        If[{z} ≠ {} && StringQ[z] && SuffPref[z, ".mx", 2],
          ToExpression["DumpSave[" <> ToString1[z] <> "," <> ToString[b] <> "]"];
          c = {z, b}, c = b];
        ClearAttributes[b, Protected];
        Map[Remove, b]; c]]]
End[]

Begin["`ContMxFile`"]
ContMxFile[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y___] :=
  Module[{a = ReadFullFile[x], b = "CONT", c = "ENDCONT", d = "`", h, t},
    h = Flatten[StringPosition[a, {b, c}]][[1 ;; 4]];
    h = StringReplace[StringTake[a, {h[[2]] + 1, h[[3]] - 2}], "┼" -> ""];
    h = StringJoin[Select[Characters[h], SymbolQ[#] &]] <> d; If[h == "", {},
      If[MemberQ[$Packages, h] && {y} ≠ {}, {h, CNames[h]}, If[! MemberQ[$Packages, h] && {y} ≠ {}, Quiet[Get[x]];
          {{h, CNames[h]}, RemovePackage[h]}[[1]], t = SubsString[a, {h, "`"}]; t = Select[t, ! MemberQ[ToCharacterCode[#], 0] &];
          {h, Sort[DeleteDuplicates[Map[StringReplace[#, {h → "", "`" → ""}] &, t]]]}]]]]
End[]

Begin["`ContMxFile1`"]
ContMxFile1[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a = ReadFullFile[x], b = "CONT", c = "ENDCONT", d, h, t},
    h = Flatten[StringPosition[a, {b, c}]][[1 ;; 4]];
    h = StringReplace[StringTake[a, {h[[2]] + 1, h[[3]] - 2}], "┼" -> ""];
    h = StringJoin[Select[Characters[h], SymbolQ[#] &]] <> "`"; If[h == "", {}, d = StringPosition[a, h][[2 ;; -1]];
      d = Map[StringTrim[#, "`"] &, Map[SubStrToSymb[a, #[[2]] + 1, "`", 1] &, d]];
      {h, Sort[Select[d, StrAllSymbNumQ[#] &]]}]]
End[]

Begin["`ContMxFile2`"]
ContMxFile2[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a = $Packages, b = "AvzAgnVsvArtKr`", c, d = Unique["ag"], g, h}, h = ToString[d];
    g = IsPackageQ[x, d]; If[g === $Failed, $Failed,
      If[g === True, {d, AladjevProcedures`CNames[d], ToExpression["Remove[" <> h <> "]"]},
        ToExpression["InputForm[BeginPackage[\"AvzAgnVsvArtKr`\"];EndPackage[]]"];
        Off[General::shdw];
        Get[x]; c = $Packages[[1]];
        b = {c, AladjevProcedures`CNames[c]}; AladjevProcedures`RemovePackage[c]; On[General::shdw]; b]]]
```

```
End[]

Begin["`ContMxW7`"]
ContMxW7[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a = FromCharacterCode[Select[BinaryReadList[x], # ≠ 0 &]], b = "CONT", c = "ENDCONT", d = "`", h, t, g = {}, k, f, n},
    h = StringPosition[a, {b, c}][[1 ;; 2]];
    If[h[[1]] − h[[2]] == {−3, 0}, $Failed, t = StringTrim[StringTake[a, {h[[1]][[2]] + 2, h[[2]][[1]] − 2}]];
      a = StringTake[a, {h[[2]][[2]] + 1, −1}]; f = StringPosition[a, t];
      Map[{c = "", For[k = #[[2]] + 1, k ≤ StringLength[a], k++, n = StringTake[a, {k, k}];
            If[n == d, Break[], c = c <> n]];
          If[StringFreeQ[c, StringTake[t, {1, −2}]], AppendTo[g, c], Null]} &, f]; {t, Select[Sort[g], StrAllSymbNumQ[#] &]}]]
End[]

Begin["`DiffContexts`"]
DiffContexts[x_ /; SymbolQ[x] && ! UnevaluatedQ[HeadPF, x], y___] := Module[
    {a = {"(*BeginPackage[\"↕`\"]*)", "(*⇵::usage=\"\"" <> "*)", "(*Begin[\"`−`\"]*)", "(*┘*)", "(*End[]*)", "(*EndPackage[]*)"},
      b = Map[FromCharacterCode, {18, 15, 6, 4}], c = Definition2[x][[1 ;; −2]], d, h = ToString[x],
      k = 1, j, t = {}, p, f = {}, z}, If[Length[c] < 2, Context[x], z = HeadPF[x]; Clear[x];
      For[k, k ≤ Length[c], k++, d = {};
        For[j = 1, j ≤ Length[a], j++,
          AppendTo[d, StringReplace[a[[j]], {b[[1]] → h <> ToString[k], b[[2]] → h, b[[3]] → h, b[[4]] → c[[k]]}]]];
        AppendTo[t, p = h <> ToString[k] <> ".m"]; AppendTo[f, {h <> ToString[k] <> "`", z[[k]]}];
        Map[{BinaryWrite[p, ToCharacterCode[#][[3 ;; −3]]], BinaryWrite[p, {32, 10}]} &, d];
        Close[p];
        Quiet[Get[p]]]; If[{y} ≠ {}, Map[DeleteFile, t], Null];
      Reverse[f]]]
End[]

ContextToSymbol[x_ /; AladjevProcedures`SymbolQ[x], y_ /; AladjevProcedures`ContextQ[y], z___] :=
  Module[{a, b = ToString[x]}, Off[General::shdw];
    a = StringReplace["BeginPackage[\"AvzAgnVsvArtKr`\"]\n90::usage=73\nBegin[\"`90`\"]\n500\nEnd[]\nEndPackage[]",
        {"AvzAgnVsvArtKr`" → y, "73" → If[AladjevProcedures`PureDefinition[x] === $Failed, "\"\"",
            If[{z} ≠ {} && StringQ[z], AladjevProcedures`ToString1[z], AladjevProcedures`ToString1["Help on " <> b]]], "90" → b,
          "500" → If[AladjevProcedures`PureDefinition[x] === $Failed, b, AladjevProcedures`PureDefinition[x]]}];
    Remove[x]; ToExpression[a]; On[General::shdw]]

Begin["`PackageFileQ`"]
PackageFileQ[x_] := If[StringQ[x] && FileExistsQ[x] && MemberQ[{"cdf", "m", "mx", "nb"}, FileExtension[x]],
    If[SameQ[ContextFromFile[x], $Failed], False, True], False]
End[]

Begin["`MxPackNames`"]
MxPackNames[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{b, a = FromCharacterCode[Select[ToCharacterCode[ReadFullFile[x]], # > 31 &]], c, d, g = {}, k, j},
    b = StringPosition[a, {"CONT", "ENDCONT"}][[1 ;; 2]];
    b = StringTake[a, {b[[1]][[2]] + 2, b[[2]][[1]] − 1}]; b = Map[#[[2]] + 1 &, StringPosition[a, b][[2 ;; −1]]];
    For[k = 1, k ≤ Length[b], k++, c = ""; For[j = b[[k]], j < Infinity, j++, d = StringTake[a, {j, j}]; If[d == "`", Break[], c = c <> d]];
      AppendTo[g, c]]; Sort[Select[g, StringFreeQ[#, {"[", "("}] &]][[2 ;; −1]]]
End[]

Begin["`PackReplaceQ`"]
PackReplaceQ[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y___] := Module[{a = $Packages, b = ContentOfMfile[x], c},
    c = Select[Select[Flatten[Map[Quiet[CNames[#]] &, a]], StringFreeQ[#, "`"] &], MemberQ[b, #] &];
    If[{y} ≠ {} && ! HowAct[y], y = c, Null]; If[c == {}, False, True] ]
End[]

Begin["`LangHoldFuncQ`"]
LangHoldFuncQ[x_] :=
  If[SystemQ[x] && Intersection[Quiet[Check[Attributes[x], False]], {HoldAll, HoldFirst, HoldRest}] ≠ {}, True, False]
End[]

Begin["`AddMxFile`"]
AddMxFile[
    x_ /; ListQ[x] && Length[x] ≥ 1 && DeleteDuplicates[Map[FileExistsQ[#] && FileExtension[#] == "mx" &, x]] == {True} &&
        DeleteDuplicates[Map[! MemberQ[$Packages, ContextInMxFile[#]] &, x]] == {True}, y_ /; ContextQ[y], z___] :=
  Module[{a = {}, b}, Do[AppendTo[a, ContextForPackage[x[[k]], y]], {k, 1, Length[x]}];
    DumpSave[b = StringTake[y, {1, −2}] <> ".mx", y]; If[{z} ≠ {}, RemovePackage[y], Null]; {y, b}]
End[]
```

```
Begin["`ContextForPackage`"]
ContextForPackage[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; ContextQ[y] && ! MemberQ[$Packages, y]] :=
  Module[{a = ContextInMxFile[x], b, c}, If[a === $Failed, $Failed,
      If[a == y, {y, x}, If[MemberQ[$Packages, a], b = 74, Get[x]]; Map[ContextToSymbol[#, y] &, CNames[a]];
        DumpSave[c = FileNameJoin[Reverse[ReplacePart[Reverse[FileNameSplit[x]], 1 –> FileBaseName[x] <> "$.mx"]]], y];
        RemovePackage[a];
        If[b == 74, Null, RemovePackage[y]]; {y, c}]]]
End[]

Begin["`DefInPackage`"]
DefInPackage[x_ /; MfilePackageQ[x] || ContextQ[x]] :=
  Module[{a, b = {"Begin[\"`", "`\"]"}, c = "BeginPackage[\"", d, p, g, t, k = 1, f, n = x},
    Label[Avz];
    If[ContextQ[n] && Contexts[n] ≠ {}, f = "$Kr20Art27$";
      Save[f, x]; g = FromCharacterCode[17]; t = n <> "Private`"; a = ReadFullFile[f, g];
      DeleteFile[f]; d = CNames[n]; p = SubsString[a, {t, g}];
      p = DeleteDuplicates[Map[StringCases[#, t ~~ Shortest[___] ~~ "[" <> t ~~ Shortest[___] ~~ " := "] &, p]];
      p = Map[StringTake[#, {StringLength[t] + 1, Flatten[StringPosition[#, "["]][[1]] – 1}] &, Flatten[p]];
      {n, DeleteDuplicates[p], d}, If[FileExistsQ[n], a = ReadFullFile[n]; f = StringTake[SubsString[a, {c, "`\"]"}], {15, –3}][[1]];
        If[MemberQ[$Packages, f], n = f; Goto[Avz]]; b = StringSplit[a, "*)(*"];
        d = Select[b, ! StringFreeQ[StringReplace[#, " " → ""], "::usage="] &];
        d = Map[StringTake[#, {1, Flatten[StringPosition[#, "::"]][[1]] – 1}] &, d];
        p = DeleteDuplicates[Select[b, StringDependAllQ[#, {"Begin[\"`", "`\"]"}] &]];
        p = MinusList[Map[StringTake[#, {9, –4}] &, p], {"Private"}];
        t = Flatten[StringSplit[SubsString[a, {"Begin[\"`Private`\"]", "End[]"}], "*)(*"]];
        If[t == {}, {f, MinusList[d, p], p}, g = Map[StringReplace[#, " " → ""] &, t[[2 ;; –1]]]; g = Select[g, ! StringFreeQ[#, ":="] &];
          g = Map[StringTake[#, {1, Flatten[StringPosition[#, ":"]][[1]] – 1}] &, g];
          g = Map[Quiet[Check[StringTake[#, {1, Flatten[StringPosition[#, "["]][[1]] – 1}], #]] &, g]; {f, g, d}], $Failed]]]
End[]

Begin["`ContentOfMfile`"]
ContentOfMfile[f_ /; FileExistsQ[f] && FileExtension[f] == "m"] := Module[{b, a = ReadFullFile[f]}, b = StringSplit[a, {"(*", "*)"}];
    b = Select[b, ! StringFreeQ[#, {"Begin[\"`", "`\"]"}] && StringFreeQ[#, "BeginPackage["] &];
    b = Flatten[Map[StringCases[#, "\"`" ~~ __ ~~ "`\""] &, b]]; b = DeleteDuplicates[Map[StringTake[#, {3, –3}] &, b]];
    Sort[Select[b, StringFreeQ[#, {"=", ",", "`", "[", "]", "(", ")", "^", "^", ";", "{", "}", "\\", "/"}] &]]]
End[]

Begin["`ContentOfMfile1`"]
ContentOfMfile1[f_ /; FileExistsQ[f] && FileExtension[f] == "m"] :=
  Sort[DeleteDuplicates[Select[Map[StringTake[#, {9, –4}] &, SubsString[ReadFullFile[f], {"Begin[\"`", "`\"]"}]],
      StringFreeQ[#, {"=", ",", "`", "[", "]", "(", ")", "^", ";", "{", "}", "\\", "/"}] &]]]
End[]

Begin["`ContentOfMfile2`"]
ContentOfMfile2[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] :=
  Sort[DeleteDuplicates[Map[StringReplace[#, {"Begin[\"`" –> "", "`\"]" –> ""}] &,
      SubsString[StringReplace[ReadFullFile[x], {" " → "", "\n" → "", "\t" –> "", "\r" –> ""}], {"Begin[\"`", "`\"]"}]]]]
End[]

Begin["`ContextDef`"]
ContextDef[x_ /; SymbolQ[x]] := Module[{a = $ContextPath, b = ToString[x], c = {}},
    Do[If[! SameQ[ToString[Quiet[Check[ToExpression["Definition1[" <> ToString1[a[[k]] <> b] <> "]"], "Null"]]], "Null"],
        AppendTo[c, {a[[k]] <> b, If[ToString[Definition[b]] != "Null", "Global`" <> b, Nothing]}]], {k, 1, Length[a]}];
    Flatten[c]]
End[]

Begin["`MainContexts`"]
MainContexts[] := DeleteDuplicates[Map[StringTake[#, {1, Flatten[StringPosition[#, "`"]][[1]]}] &, Contexts[]]]
End[]

Begin["`ContextsInModule`"]
ContextsInModule[x_ /; BlockQ[x] || ModuleQ[x]] :=
  Module[{a = Select[ExtrVarsOfStr[PureDefinition[x], 1], ! MemberQ[{"Block", "Module"}, #] &], b, c, d, h, Sf},
    Sf[y_] := Flatten[{Select[y, ! StringFreeQ[#, "`"] &], Select[y, StringFreeQ[#, "`"] &]}]; ClearAll[h]; NotSubsProcs[x, h];
    a = MinusList[a, Flatten[{b = Args[x, 90], c = Locals1[x], h}]];
    h = {PrependTo[b, "Args"], PrependTo[c, "Locals"], PrependTo[h, "SubProcs"]};
    Flatten[{h, Map[Sf, Map[DeleteDuplicates, Map[Flatten, Gather[Map[{#, Context[#]} &, a], #1[[2]] == #2[[2]] &]]]], 1]]
End[]
```

```
Begin["`ContextToFileName1`"]
ContextToFileName1[y___] :=
  Module[{a = Directory[], b = SetDirectory[$InstallationDirectory], c, d}, c = FileNames[{"*.m"}, {"*"}, Infinity];
    d = Sort[DeleteDuplicates[Map[StringTake[#, {Flatten[StringPosition[#, "\\"]][[-1]] + 1, -3}] <> "`" &, c]]];
    SetDirectory[a]; If[{y} ≠ {} && ContextQ[{y}[[1]]],
      Map[b <> "\\" <> #1 &, Select[c, SuffPref[#, "\\" <> StringTake[y, {1, -2}] <> ".m", 2] &]], d]]
End[]

Begin["`FullCalls`"]
FullCalls[x_ /; ProcQ[x] || FunctionQ[x]] :=
  Module[{a = {}, b, d, c = "::usage = ", k = 1}, Save[b = ToString[x], x]; For[k, k < Infinity, k++, d = Read[b, String];
      If[d === EndOfFile, Break[], If[StringFreeQ[d, c], Continue[],
        AppendTo[a, StringSplit[StringTake[d, {1, Flatten[StringPosition[d, c]][[1]] - 1}], " /: "][[1]]]]]];
    a = Select[a, SymbolQ[#] &]; DeleteFile[Close[b]]; a = Map[{#, Context[#]} &, DeleteDuplicates[a]];
    a = If[Length[a] == 1, a, Map[DeleteDuplicates, Map[Flatten, Gather[a, #1[[2]] === #2[[2]] &]]]];
    {d, k} = {{}, 1};
    While[k ≤ Length[a], b = Select[a[[k]], ContextQ[#] &]; c = Select[a[[k]], ! ContextQ[#] &]; AppendTo[d, Flatten[{b, Sort[c]}]];
      k++];
    d = MinusList[If[Length[d] == 1, Flatten[d], d], {ToString[x]}]; If[d == {Context[x]}, {}, d]]
End[]

Begin["`Rule1`"]
Rule1[x__] := Module[{a = {x}, b = Length[{x}]},
    If[b == 2, a[[1]] → a[[2]], If[b > 2 && EvenQ[b], Map[a[[#]] → a[[# + 1]] &, Select[Range[1, b], OddQ[#] &]], If[b == 1 &&
        ListQ[x] && EvenQ[Length[x]], Map[x[[#]] → x[[# + 1]] &, Select[Range[1, Length[x]], OddQ[#] &]], Defer[Rule[x]]]]]]
End[]

Begin["`FullCalls1`"]
FullCalls1[x_ /; ProcQ[x] || FunctionQ[x]] := Module[{a = {}, b, c = "", d, k = 1, n, p}, Save[b = ToString[x], {x, c}];
    For[k, k < Infinity, k++, d = Read[b, String];
      If[d === EndOfFile, Break[], If[d != " ", c = c <> d, If[n = Flatten[StringPosition[c, " := "]]; n ≠ {},
          If[Quiet[HeadingQ[p = StringTake[c, {1, n[[1]] - 1}]]], a = Append[a, Quiet[HeadName[StringTake[c, {1, n[[1]] - 1}]]]]];
        c = ""]]]; DeleteFile[Close[b]];
    {b = FullCalls[x], Select[MinusList[a, {ToString[x]}], ! MemberQ[Flatten[b], #] &]}]
End[]

Begin["`StrSub`"]
StrSub[x_ /; StringQ[x], y_ /; StringQ[y], z_ /; StringQ[z], n_ /; PosIntQ[n] || PosIntListQ[n]] :=
  Module[{a = StringPosition[x, y], b = Sort[Flatten[{n}]]}, If[a == {} || Length[a] < b[[-1]], $Failed, StringReplacePart[x, z, Part[a, b]]]]
End[]

Begin["`StrToList`"]
StrToList[x_ /; StringQ[x]] :=
  Module[{a, b = {}, c = {}, d, h, k = 1, j, y = If[StringTake[x, {1, 1}] == "{" && StringTake[x, {-1, -1}] == "}", StringTake[x, {2, -2}], x]},
    a = DeleteDuplicates[Flatten[StringPosition[y, "="]] + 2];
    d = StringLength[y]; If[a == {}, Map[StringTrim, StringSplit[y, ","]], While[k ≤ Length[a], c = ""; j = a[[k]];
        For[j, j ≤ d, j++, c = c <> StringTake[y, {j, j}];
          If[! SameQ[Quiet[ToExpression[c]], $Failed] && (j == d || StringTake[x, {j + 1, j + 1}] == ","),
            AppendTo[b, c → ToString[Unique[$Art$Kr$]]];
            Break[]]];
        k++];
      h = Map[StringTrim, StringSplit[StringReplace[y, b], ","]]; Map14[StringReplace, h, RevRules[b]]]]
End[]

Begin["`ArrayInd`"]
ArrayInd[x_ /; StringQ[x]] := Module[{a = StringSplit[ToString[InputForm[DefFunc[x]]], "\n \n"]}, If[a == {"Null"}, {}, a]]
End[]

Begin["`StrOfSymblQ`"]
StrOfSymblQ[x_ /; StringQ[x], A_ /; ListQ[A]] :=
  If[DeleteDuplicates[Map3[MemberQ, Map[ToString, A], Characters[x]]] == {True}, True, False]
End[]

Begin["`StrPartition`"]
StrPartition[x_ /; StringQ[x], y_ /; StringQ[y] && StringLength[y] == 1 ||
      ListQ[y] && DeleteDuplicates[Map[StringQ[#] &, y]] == {True} && DeleteDuplicates[Map[StringLength[#] &, y]] == {1}] :=
  Module[{a = DeleteDuplicates[Flatten[StringPosition[x, y]]], b = {}}, Flatten[AppendTo[b, Map[StringTake[x, {1, #}] &, a]]]]
End[]
```

```
Begin["`Un`"]
Un[x___] := Module[{a},
    If[{x} == {}, Return[Symbol["$" <> ToString[$ModuleNumber]]], a[y_] := If[StringQ[y], Symbol[y <> ToString[$ModuleNumber]],
          If[Head[y] == Symbol, Symbol[ToString[y] <> "$" <> ToString[$ModuleNumber]], y]];
      If[ListQ[x], Map[a, Flatten[x]], a[x]]]]
End[]

Begin["`Unique1`"]
Unique1[x_, y___] := Module[{a = Unique[y], b}, b = ToString[a];
    ToExpression[ToString[a] <> "=" <> ToString1[x]]; b]
End[]

Begin["`ListSymbolQ`"]
ListSymbolQ[x_ /; ListQ[x]] := SameQ[DeleteDuplicates[SymbolQ /@ Flatten[x]], {True}]
End[]

Begin["`MemberQL`"]
MemberQL[x_ /;  ListQ[x], y_, z___] := Module[{a = ElemOnLevels[x], b, c}, If[! NestListQ[a], a = {a}, Null];
    b = Map[If[Length[#] == 1, Null, {#[[1]], Count[#[[2 ;; -1]], y]}] &, a];
    b = Select[b, ! SameQ[#, Null] && #[[2]] ≠ 0 &];
    If[b == {}, False, If[{z} ≠ {} && ! HowAct[z], z = b, Null]; True]]
End[]

Begin["`IsMonotonic`"]
IsMonotonic[x__] := Module[{a = {x}, b = {x}[[1]], c = If[StringQ[{x}[[1]]], ToCharacterCode[{x}[[1]]], "Err"]},
    If[c === "Err", Return[Defer[IsMonotonic[x]]], If[c == Sort[c, #1 > #2 &],
        If[Length[a] > 1 && ! HowAct[a[[2]]], ToExpression[ToString[a[[2]]] <> "=" <> "\"Decrease\""], Null];
        True, If[c == Sort[c, #1 < #2 &], If[Length[a] > 1 && ! HowAct[a[[2]]],
            ToExpression[ToString[a[[2]]] <> "=" <> "\"Increase\""], Null];
          True, False]]]
End[]

Begin["`ListStrQ`"]
ListStrQ[x_ /; ListQ[x]] := Length[Select[x, StringQ[#] &]] == Length[x] && Length[x] ≠ 0
End[]

Begin["`Contexts1`"]
Contexts1[] := Module[{a = {}, b = Contexts[], c, k = 1}, For[k, k ≤ Length[b], k++, c = b[[k]];
      If[Length[DeleteDuplicates[Flatten[StringPosition[c, "`"]]]] == 1 && StringTake[c, {-1, -1}] == "`", a = Append[a, c], Next[]]]; a]
End[]

Begin["`UserPackTempVars`"]
UserPackTempVars[x_ /; ContextQ[x]] := Module[{a = {}, p, b = {}, d = {}, c = Names[x <> "*"], h = {}},
    Quiet[Map[{p = Definition2[#], If[UnevaluatedQ[Definition2, #], AppendTo[d, #],
              If[p[[2]] == {} && p[[1]] == "Undefined", AppendTo[a, #], If[p[[2]] == {Temporary}, AppendTo[b, #], 6]]]} &, c];
    Map[{p = Flatten[StringPosition[#, "$"]], If[p[[-1]] == StringLength[#], AppendTo[a, #],
          If[IntegerQ[ToExpression[StringTake[#, {p[[-1]] + 1, StringLength[#]}]]], AppendTo[h, #]]]} &, b];  {d, a, h}]
End[]

Begin["`TempInPack`"]
TempInPack[x_ /; ContextQ[x]] := Select[Names[x <> "*"], TemporaryQ[#] &]
End[]

Begin["`$UserContexts`"]
$UserContexts := Select[Map[If[Flatten[UserPackTempVars[#][[2 ;; 3]]] ≠ {}, #] &,
      Select[$Packages, ! MemberQ[{"Global`", "System`"}, #] &]], ! SameQ[#, Null] &]
End[]

Begin["`SystemSymbols`"]
SystemSymbols[] := Module[{a = Names["*"], b = Join[Map[FromCharacterCode, Range[63 488;
            63 596]], CNames["Global`"]], c = ActUcontexts[590]}, MinusList[a, Join[b, Flatten[Map[CNames[#] &, c]]]]]
End[]

Begin["`SubLists`"]
SubLists[x_ /; ListQ[x]] := Module[{a, b, c = {}, k = 1}, If[! NestListQ1[x], {}, a = ToString[x];
      b = DeleteDuplicates[Flatten[StringPosition[a, "{"]]];
      While[k ≤ Length[b], AppendTo[c, SubStrSymbolParity1[StringTake[a, {b[[k]], -1}], "{", "}"][[1]]];
        k++];
      DeleteDuplicates[ToExpression[c[[2 ;; -1]]]]]]
```

```
End[]

Begin["`ContextQ`"]
ContextQ[x_] :=
  StringQ[x] && StringLength[x] > 1 && Quiet[SymbolQ[Symbol[StringTake[x, {1, -2}]]]] && StringTake[x, {-1, -1}] == "`"
End[]

Begin["`ContextToSymbol`"]
ContextToSymbol[x_ /; SymbolQ[x], y_ /; ContextQ[y]] :=
  Module[{a = ToString[x],  b = Flatten[{PureDefinition[x]}], c, d, h = Attributes[x], f},
    If[b === {$Failed}, $Failed, AppendTo[$ContextPath, y];
      f = a <> ".mx";  Attributes[x] = {};  Quiet[ToExpression[Map[y <> # &, b]]];  c = ToExpression[y <> a];
      ToExpression[y <> a <> "::usage = " <> ToString1[d = ToExpression[a <> "::usage"]]];  DumpSave[f, {c, d}];  Remove[x];  Get[f];
      ToExpression[" SetAttributes[" <> a <> "," <> ToString[h] <> "]"];  DeleteFile[f];  {c, y}]]
End[]

Begin["`ToContext`"]
ToContext[x_ /; SymbolQ[x] || ListQ[x] && DeleteDuplicates[Map[SymbolQ, x]] == {True}, y_ /; ContextQ[y]] :=
  Module[{h = Flatten[{x}], a, attr, b, c}, a = Map[PureDefinition, h];
    attr = Map[Attributes, h];  Map[ClearAttributes[#, Protected] &, h];  Map[Remove, h];
    Quiet[ToExpression[{"BeginPackage[\"" <> y <> "\"]", ToString[Flatten[a]], "Map[Attributes[#[[1]]] = #[[2]]&," <>
          "Partition[Riffle[" <> ToString[h] <> "," <> ToString[attr] <> "], 2]]", "EndPackage[]"}]]; ]
End[]

Begin["`ContextToSymbol1`"]
ContextToSymbol1[x_ /; AladjevProcedures`SymbolQ[x], y_ /; AladjevProcedures`ContextQ[y], z___] :=
  Module[{a, b = ToString[x]}, Off[General::shdw];
    a = StringReplace["BeginPackage[\"AvzAgnVsvArtKr`\"]\n90::usage=74\nBegin[\"`90`\"]\n500\nEnd[]\nEndPackage[]",
        {"AvzAgnVsvArtKr`" → y, "74" → If[AladjevProcedures`PureDefinition[x] === $Failed, "\"\"",
            If[{z} ≠ {} && StringQ[z], AladjevProcedures`ToString1[z], AladjevProcedures`ToString1["Help on " <> b]]], "90" → b,
          "500" → If[AladjevProcedures`PureDefinition[x] === $Failed, b, AladjevProcedures`PureDefinition[x]]}];
    Remove[x];  ToExpression[a];  On[General::shdw]]
End[]

Begin["`ContextToSymbol2`"]
ContextToSymbol2[x_ /; SymbolQ[x], y_ /; ContextQ[y], z___] :=
  Module[{a = Flatten[{PureDefinition[x]}], b = ToString[x], c = ToString[x] <> ".mx", d, h, t}, If[a === {$Failed}, $Failed,
      t = If[! StringQ[Set[t, ToExpression[b <> "::usage"]]], Nothing, y <> b <> "::usage = " <> ToString1[t]];  a = Flatten[{a, t}];
      Map[ToExpression, Map[StringReplace[#, b <> "[" → y <> b <> "[", 1] &, a]];
      d = Attributes[x];  AppendTo[$ContextPath, y];  DumpSave[c, y];  Unprotect[x, h];  ClearAll[x, h];  Get[c];
      ToExpression["SetAttributes[" <> y <> b <> "," <> ToString[d] <> "]"];  If[{z} == {}, DeleteFile[c];  {x, y}, {x, y}]]]
End[]

Begin["`ContextSymbol`"]
ContextSymbol[x_ /; SymbolQ[x]] :=
  Select[Map[If[MemberQ[CNames[#], ToString[x]] || MemberQ[CNames[#], # <> ToString[x]], #] &,
      DeleteDuplicates[$ContextPath]], ! SameQ[#1, Null] &]
End[]

Begin["`ContextRepMx`"]
ContextRepMx[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; ContextQ[y], z___] :=
  Module[{a = ContextMXfile[x], b, c, t}, If[SameQ[a, $Failed], $Failed,
      If[{z} ≠ {} && FileExtension[z] == "mx", c = z, c = x];  If[MemberQ[$Packages, a], t = 74, Get[x]];
      b = CNames[a];  Map[Quiet[ContextToSymbol[#, y]] &, b];
      ToExpression["RemovePackage[" <> ToString1[a] <> "]"];  DumpSave[c, y];
      If[t == 74, Get[c], RemovePackage[y]];  {c, a, y}]]
End[]

Begin["`RemoveNames`"]
RemoveNames[x___] := Module[{a = Select[Names["`*"], ToString[Definition[#]] ≠ "Null" &], b},
    ToExpression["Remove[" <> StringTake[ToString[MinusList[a, Select[a, ProcQ[#] ||
                  ! SameQ[ToString[Quiet[DefFunc[#]]], "Null"] || Quiet[Check[QFunction[#], False]] &]]], {2, -2}] <> "]"];
    b = Select[a, ProcQ[#] &];  {b, MinusList[a, b]}]
End[]

Begin["`RhsLhs`"]
RhsLhs[x__] :=
  Module[{a = Head[{x}[[1]]], b = ToString[InputForm[{x}[[1]]]], d, h = {x}, c = {{Greater, ">"}, {GreaterEqual, ">="}, {Span, ";;"},
      {And, "&&"}, {LessEqual, "<="}, {Unequal, "!="}, {Or, "||"}, {Rule, "->"}, {Less, "<"},
```

```
        {Plus, {"+", "−"}}, {Power, "^"}, {Equal, "=="}, {NonCommutativeMultiply, "∗∗"}, {Times, {"∗", "/"}}}},
    If[Length[h] < 2 || ! MemberQ[{"Lhs", "Rhs"}, h[[2]]], Return[Defer[RhsLhs[x]]], Null];
    If[! MemberQ[Select[Flatten[c], ! StringQ[#] &], a] || a == Symbol, Return[Defer[RhsLhs[x]]], Null];
    d = StringPosition[b, Flatten[Select[c, #[[1]] == a &], 1][[2]]]; a = Flatten[Select[c, #[[1]] == a &]];
    If[Length[h] ≥ 3 && ! HowAct[h[[3]]], ToExpression[ToString[h[[3]]] <> "=" <> ToString1[a]], Null];
    ToExpression[
      If[h[[2]] == "Lhs", StringTrim[StringTake[b, {1, d[[1]][[1]] − 1}]], StringTrim[StringTake[b, {d[[1]][[2]] + 1, −1}]]]]]
End[]

Begin["`RhsLhs1`"]
RhsLhs1[x_, y__] := Module[{a = Head[x], b = {x, y}, d,
      c = {{Greater, ">"}, {GreaterEqual, ">="}, {And, "&&"}, {LessEqual, "<="}, {Unequal, "!="}, {Or, "||"}, {Rule, "−>"}, {Less, "<"},
        {Plus, {"+", "−"}}, {Power, "^"}, {Equal, "=="}, {Span, ";;"}, {NonCommutativeMultiply, "∗∗"}, {Times, {"∗", "/"}}}},
    If[! MemberQ[Flatten[c], a], Return[Defer[RhsLhs1[x, y]]], d = Level[x, 1]];
    If[Length[b] > 2 && ! HowAct[b[[3]]],
      ToExpression[ToString[b[[3]]] <> " = " <> ToString1[Flatten[Select[c, #[[1]] === a &]]]], Null];
    If[b[[2]] == "Lhs", d[[1]], If[b[[2]] == "Rhs", d[[2]], Defer[RhsLhs1[x, y]]]]]
End[]

Begin["`BitSet1`"]
BitSet1[n_ /; IntegerQ[n] && n ≥ 0, p_ /; ListQ[p]] :=
  Module[{b = 1, c, d, a = ToExpression[Characters[IntegerString[n, 2]]], h = If[ListListQ[p], p, {p}]},
    If[ListListQ[h] && Length[Select[h, Length[#] == 2 && IntegerQ[#[[1]]] && IntegerQ[#[[2]]] && MemberQ[{0, 1}, #[[2]]] &]] ==
        Length[h], Null, Return[Defer[BitSet1[n, p]]]];
    For[b, b ≤ Length[h], b++, {c, d} = {h[[b]][[1]], h[[b]][[2]]};
      If[c ≤ Length[a], a[[c]] = d, Null]]; Sum[a[[k]]∗2^(Length[a] − k), {k, Length[a]}]]
End[]

Begin["`OP`"]
OP[expr_] := Module[{a = ToString[InputForm[expr]], b = {}, c, d, k, h},
    If[StringTake[a, {−1, −1}] == "]", a = Flatten[Ind[expr]], a = DeleteDuplicates[Quiet[ToList[expr]]]];
    Label[ArtKr]; d = Length[a];
    For[k = 1, k ≤ Length[a], k++, h = a[[k]]; c = Quiet[ToList[h]];
      If[MemberQ[DeleteDuplicates[c], $Failed], AppendTo[b, Ind[h]], AppendTo[b, c]]];
    a = DeleteDuplicates[Flatten[b]];
    If[d == Length[a], Sort[a], b = {}; Goto[ArtKr]]]
End[]

Begin["`Op`"]
Op[x_] := Module[{a = {}, b}, If[ListQ[x], a = x, Do[a = Insert[a, Part[x][[b]], −1], {b, Length[x]}]]; a]
End[]

Begin["`FullFormF`"]
FullFormF[] := {And, AngleBracket, CapitalDifferentialD, CenterDot, Complex, Condition, Congruent, Decrement, DifferentialD,
    DoubleVerticalBar, DirectedEdge, Element, Equal, Equivalent, Exists, ForAll, Function, GreaterFullEqual, Greater,
    GreaterEqual, Implies, Increment, Infinity, Intersection, LeftArrow, LessEqual, Less, LessFullEqual, LessLess,
    MessageName, MinusPlus, Not, NotDoubleVerticalBar, NotCongruent, NotElement, NotExists, NotReverseElement,
    NotSubset, NotSuperset, Or, Part, Pattern, Plus, PlusMinus, Power, Product, Proportion, Proportional, Rational, Repeated,
    ReverseElement, RoundImplies, Rule, RuleDelayed, SameQ, Span, StringJoin, Subset, SubsetEqual, Sum, Superset,
    SupersetEqual, TildeEqual, TildeTilde, Times, UndirectedEdge, Unequal, Union, UpArrow, VerticalBar, VerticalSeparator}
End[]

Begin["`UnDefVars`"]
UnDefVars[x_] := Select[OP[x], Quiet[ToString[Definition[#]]] == "Null" &]
End[]

Begin["`MinusLists`"]
MinusLists[x_ /; ListQ[x], y_ /; ListQ[y], z_ /; MemberQ[{1, 2}, z]] :=
  Module[{a, b, c, k = 1, n}, If[z == 1, Select[x, ! MemberQ[y, #] &],
      a = Intersection[x, y]; b = Map[Flatten, Map[Flatten[Position[x, #] &], a]]; c = Map[Count[y, #] &, a];
      n = Length[b]; For[k, k ≤ n, k++, b[[k]] = b[[k]][[1 ;; c[[k]]]]]; c = Map[GenRules[#, Null] &, b];
      Select[ReplacePart[x, Flatten[c]], ! SameQ[#, Null] &]]]
End[]

Begin["`UnDefVars1`"]
UnDefVars1[x_] := Select[ExtrVarsOfStr[ToString[x], 2], ! SystemQ[#] &]
End[]

Begin["`MultEntryList`"]
```

```
MultEntryList[x_ /; ListQ[x]] := Map[{#[[1]], Length[#]} &, Gather[Flatten[x]]]
End[]

Begin["`ContextActQ`"]
ContextActQ[x_ /; ContextQ[x]] := MemberQ[DeleteDuplicates[Contexts1[]], x]
End[]

Begin["`ReloadPackage`"]
ReloadPackage[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y___List, t___] :=
  Module[{a = NamesMPackage[x], b = ContextMfile[x], c = "$Art27$Kr20$.txt",
      d = If[{y} ≠ {}, ToExpression[Map14[StringJoin, Map[ToString, y], "[", 90]], {}], p, k = 1},
    Put[
      c];
    While[k ≤ Length[a], p = a[[k]];
      PutAppend[StringReplace[ToString1[ToExpression["Definition[" <> p <> "]"]], b <> p <> "`" → ""], c]; k++];
    If[d == {}, ToExpression["Clear[" <> StringTake[ToString[a], {2, -2}] <> "]"], Null]
      While[b != "EndOfFile", b = ToString[Read[c]]; If[b === "EndOfFile", Break[]]; If[d == {}, Quiet[ToExpression[b]];
          Continue[],
          If[If[{t} == {}, MemberQ, ! MemberQ][d, StringTake[b, {1, Quiet[StringPosition[b, "[", 1][[1]][[1]]]}]], Quiet[ToExpression[b]];
            Break[], Continue[]]]]; DeleteFile[Close[c]]]
End[]

Begin["`ReloadPackage1`"]
ReloadPackage1[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y_: 0, t_: 0] :=
  Module[{a = NamesMPackage[x], b = ReadFullFile[x], c, d = Map[ToString, Flatten[{y}]]},
    c = Flatten[Map[SubsString[b, {"*)(*Begin[\"`" <> # <> "`\"]*)(*", "*)(*End[]*)"}, 90] &, a]];
    c = Map[StringReplace[#, "*)(*" → ""] &, c]; Map[If[d == {"0"}, Quiet[ToExpression[#]],
        If[ListQ[y], If[{t} == {0}, If[MemberQ[d, StringTake[#, Flatten[StringPosition[#, {"[", " :=", "="}]][[1]] - 1]],
            ToExpression[#], If[! MemberQ[d, StringTake[#, Flatten[StringPosition[#, {"[", " :=", "="}]][[1]] - 1]],
              Quiet[ToExpression[#]]]]]] &, c];]
End[]

Begin["`DefWithContext`"]
DefWithContext[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] := Module[{a = ContextMfile[x], b, c = {}, d = {}}, b = CNames[a];
    Map[If[StringFreeQ[Definition4[#], a <> # <> "`"], AppendTo[c, #], AppendTo[d, #]] &, b]; {c, d}]
End[]

Begin["`TypeWinMx`"]
TypeWinMx[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a, b, c, d}, If[StringFreeQ[$OperatingSystem, "Windows"], $Failed,
      a = StringJoin[Select[Characters[ReadString[x]], SymbolQ[#] || Quiet[IntegerQ[ToExpression[#]]] || # == "-" &]];
      d = Map[FromCharacterCode, Range[2, 27]]; b = StringPosition[a, {"CONT", "ENDCONT"}];
      If[b[[1]][[2]] == b[[2]][[2]],
        c = StringCases1[a, {"Windows", "ENDCONT"}, "___"], b = StringPosition[a, {"Windows", "CONT"}];
        c = StringTake[a, {b[[1]][[1]], b[[2]][[2]]}]];
      c = StringReplace[c, Flatten[{GenRules[d, ""], "ENDCONT" -> "", "CONT" -> ""}]];
      If[ListQ[c], c[[1]], c]]
End[]

Begin["`Ver`"]
Ver[] := Module[{a = "$Art24$Kr17$.txt", b}, Run["Ver > " <> a]; b = Read[a, String]; DeleteFile[Close[a]]; b]
End[]

Begin["`$Version1`"]
$Version1 := StringSplit[$Version, " "][[1]]
End[]

Begin["`$Version2`"]
$Version2 := StringTrim[ReadString[$InstallationDirectory <> "\\.VersionID"]]
End[]

Begin["`PCOS`"]
PCOS[] := Module[{a = ToString[Unique["agn"]], b}, Run["SYSTEMINFO > " <> a];
    b = Map[StringSplit[#] &, ReadList[a, String][[1 ;; 2]]]; DeleteFile[a]; b = Map[#[[3 ;; -1]] &, b];
    {b[[1]][[1]], StringReplace[ListToString[b[[2]], " "], "\"" -> ""]}]
End[]

Begin["`OSplatform`"]
OSplatform[] := Module[{a = "$Art27$Kr20$", b = {}}, Run["systeminfo > " <> a]; Do[AppendTo[b, Read[a, String]], 3];
```

```
    DeleteFile[Close[a]];  b = Map[StringSplit[#, ":"] &, b[[2 ;; 3]]];  Map[StringTrim[#[[2]]] &, b]]
End[]

Begin["`DO`"]
DO[x_, y_, k_] := Module[{a = x, b = Op[y], c, d = 1, R = {}},  c := Length[b] + 1;
    While[d < c, R = Insert[R, a /. k  −> b[[d]],  −1];
      a := x;
      d ++]; R]
End[]

Begin["`CopyFileToDir`"]
CopyFileToDir[x_ /; PathToFileQ[x], y_ /; DirQ[y]] :=
  Module[{a, b}, If[DirQ[x], CopyDir[x, y], If[FileExistsQ[x], a = FileNameSplit[x][[−1]];
      If[FileExistsQ[b = y <> "\\" <> a], b, CopyFile[x, b]], $Failed]]]
End[]

Begin["`CopyFile1`"]
CopyFile1[x_ /; FileExistsQ[x], y_ /; StringQ[y]] := Module[{a, b, c, d, p = StandPath[x], j = StandPath[y]},
    b = StandPath[If[Set[a, DirectoryName[j]] === "", Directory[], If[DirectoryQ[a], a, CDir[a]]]];
    If[p == j, p, If[FileExistsQ[j], Quiet[Close[j]];
      Quiet[RenameFile[j, c = StringReplace[j, Set[d, FileBaseName[j]] −> "$" <> d <> "$"]]];
      CopyFileToDir[p, b];
      {b <> "\\" <> FileNameTake[p], j = b <> "\\" <> c}, CopyFileToDir[p, b]]]
End[]

Begin["`Head1`"]
Head1[x_] :=
  Module[{a = PureDefinition[x]}, If[ListQ[a], $Failed, If[a === "System", System, If[BlockQ[x], Block, If[ModuleQ2[x], Module,
      If[PureFuncQ[x], PureFunction, If[Quiet[Check[FunctionQ[x], False]], Function, Head[x]]]]]]]
End[]

Begin["`Head2`"]
Head2[x_] := Module[{a = Quiet[Check[ProcFuncTypeQ[ToString[x]], {Head[x]}]], b},
    If[SameQ[a[[−1]], "System"], "System", If[SameQ[a[[−1]], "Expression"], Head[x], If[ListQ[a], b = a[[−1]];
      If[Length[b] == 1, b[[1]], b]]]]
End[]

Begin["`Head3`"]
Head3[x_] := If[Part[x, 1] === −1, Head1[−1 * x], Head1[x]]
End[]

Begin["`ReductRes`"]
ReductRes[x_ /; SymbolQ[x], y_] := ToExpression[StringReplace[ToString[y], Context[x] <> ToString[x] <> "`" −> ""]]
End[]

Begin["`ListListGroup`"]
ListListGroup[x_ /; ListListQ[x],  n_ /; IntegerQ[n] && n > 0] :=
  Module[{a = {}, b = {}, k = 1}, If[Length[x[[1]]] < n, Return[Defer[ListListGroup[x, n]]],
      For[k, k ≤ Length[x], k ++, AppendTo[b, x[[k]][[n]]];  b = DeleteDuplicates[Flatten[b]]];
    For[k = 1, k ≤ Length[b], k ++, AppendTo[a, Select[x, #[[n]] == b[[k]] &]]]; a]
End[]

Begin["`ReplaceLevelList`"]
ReplaceLevelList[x_ /; ListQ[x], n_ /; IntegerQ[n] && n > 0 || IntegerListQ[n], y_, z___] :=
  Module[{a, b = SetAttributes[ToString4, Listable], c = Flatten[x], d, h = FromCharacterCode[2], k, p = {},
      g = FromCharacterCode[3], u = FromCharacterCode[4], m, n1 = Flatten[{n}], y1 = Flatten[{y}], z1},
    If[{z} ≠ {}, z1 = Flatten[{z}], z1 = y1];  If[Length[DeleteDuplicates[Map[Length, {n1, y1, z1}]]] ≠ 1, $Failed,
      a = ToString4[x];
      SetAttributes[StringJoin, Listable];
      b = ToExpression[ToString4[StringJoin[u, a, g <> h]]];
       b = ToString[b];  m = StringPosition[b, h];  d = LevelsOfList[x];  b = StringReplacePart[ToString[b], Map[ToString, d], m];
      For[k = 1, k ≤ Length[n1], k ++, AppendTo[p, u <> Map[ToString4, y1][[k]] <> g <> Map[ToString, n1][[k]] →
          If[{z} == {}, "", Map[ToString, z1][[k]]]]];  b = StringReplace[b, p];
       d = Map[g <> # &, Map[ToString, d]];  Map[ClearAttributes[#, Listable] &, {StringJoin, ToString4}];
      ToExpression[StringReplace[b, Flatten[{u −> "", ", }" −> "}", "{," −> "{", ", ," −> ",",  " , ," −> "",  " , " −> "", GenRules[d, ""]}]]]]]
End[]

Begin["`ReplaceListCond`"]
ReplaceListCond[x_ /; ListQ[x],
```

```
    f_ /; SymbolQ[f] || ListQ[f] && DeleteDuplicates[Map[SysFuncQ[#] || FunctionQ[#] &, f]] == {True}, y__] := Module[
    {a, b, func}, func[x1_ /; ListQ[x1], f1_, y1_] := Map[If[! UnevaluatedQ[f1, #] && f1[#] || f1[#], Replace[#, # → y1], #] &, x1];
    If[! ListQ[f], func[x, f, y], {a, b} = {x, Flatten[{y}]}; Quiet[Do[a = func[a, f[[k]], If[Length[b] == 1, b[[1]], b[[k]]],
        {k, 1, If[Length[b] == 1, Length[f], Min[{Length[f], Length[b]}]]}]]; a]]
End[]

Begin["`PositionsListCond`"]
PositionsListCond[x_ /; ListQ[x],
    f_ /; SymbolQ[f] || ListQ[f] && DeleteDuplicates[Map[SysFuncQ[#] || FunctionQ[#] &, f]] == {True}] :=
  Module[{a, b = {}}, Do[If[MemberQ[Map[#[x[[k]]] &, Flatten[{f}]], True], AppendTo[b, k], Null], {k, 1, Length[x]}]; b]
End[]

Begin["`FreeSpaceVol`"]
FreeSpaceVol[x_ /; MemberQ3[Join[CharacterRange["a", "z"], CharacterRange["A", "Z"]], Flatten[{x}]]] :=
  Module[{a = ToString[Unique["ag"]], c, d = Flatten[{x}], f},
    f[y_] := Module[{n, b}, n = Run["Dir " <> y <> ":\\ > " <> a];
      If[n ≠ 0, {y, "Device is not ready"}, b = StringSplit[ReduceAdjacentStr[ReadFullFile[a], " ", 1]][[-3 ;; -2]];
        {y, ToExpression[StringReplace[b[[1]], "ÿ" -> ""]], b[[2]]}]]; c = Map[f, d]; DeleteFile[a]; If[Length[c] == 1, c[[1]], c]]
End[]

Begin["`VolDir`"]
VolDir[x_ /; DirectoryQ[x] || MemberQ[Map[# <> ":" &, Adrive[]], ToUpperCase[x]]] :=
  Module[{a = ToString[Unique["agn"]], b, c, d = StandPath[x]}, b = Run["DIR /S " <> d <> " > " <> a];
    If[b ≠ 0, $Failed, c = Map[StringTrim, ReadList[a, String][[-2 ;; -1]]]]; DeleteFile[a];
    c = Map[StringTrim, Mapp[StringReplace, c, {"ÿ" -> "", "bytes" -> "", "free" -> ""}]];
    ToExpression[Map[StringSplit[#][[-1]] &, c]]]
End[]

Begin["`DirsFiles`"]
DirsFiles[x_ /; DirectoryQ[x] || MemberQ[Map[# <> ":" &, Adrive[]], ToUpperCase[x]]] :=
  Module[{a = ToString[Unique["agn"]], b = {x}, c = {}, d = StandPath[x], f},
    If[Run["DIR /A/B/S " <> d <> " > " <> a] ≠ 0, $Failed, f = ReadList[a, String]; DeleteFile[a];
      Map[If[DirectoryQ[#], AppendTo[b, #], If[FileExistsQ[#], AppendTo[c, #], Null]] &, f]; {b, c}]]
End[]

Begin["`GroupNames`"]
GroupNames[L_] := Module[{a = If[ListQ[L], L, {L}], c, d, p, t, b = {{"Null", "Null"}}, k = 1}, For[k, k ≤ Length[a], k++, c = a[[k]];
      d = Head2[c];
      t = Flatten[Select[b, #[[1]] === d &]];
      If[t == {} && (d === Symbol  && Attributes[c] === {Temporary}),
        AppendTo[b, {Temporary, c}], If[t == {}, AppendTo[b, {d, c}], p = Flatten[Position[b, t]][[1]];
          AppendTo[b[[p]], c]]]; b = b[[2 ;; -1]]; b = Gather[b, #1[[1]] == #2[[1]] &];
    b = Map[DeleteDuplicates[Flatten[#]] &, b]; If[Length[b] == 1, Flatten[b], b]]
End[]

Begin["`LeftFold`"]
LeftFold[F_, id_, s_ /; StringQ[s]] := Module[{a = StringLength[s], b = ToString[F] <> "[", c = "", k = 1},
    For[k, k ≤ a, k++, c = c <> b <> ToString1[StringTake[s, {k, k}]] <> ", "]; ToExpression[c <> ToString[id] <> CatN["]", a]]]
End[]

Begin["`RightFold`"]
RightFold[F_, id_, s_ /; StringQ[s]] := Module[{a = StringLength[s], b = ToString[F] <> "[", c = "", k = 1},
    For[k, k ≤ a, k++, c = c <> b <> ToString1[StringTake[s, {-k, -k}]] <> ", "]; ToExpression[c <> ToString[id] <> CatN["]", a]]]
End[]

Begin["`Names1`"]
Names1[x___ /; {x} == {}] :=
  Module[{c = 1, d, h, b = {{}, {}, {}, {}}, a = Select[Names["`*"], StringTake[#, {1, 1}] ≠ "$" &]}, While[c ≤ Length[a], d = a[[c]];
      If[ProcQ[d], AppendTo[b[[1]], d],
        If[Quiet[Check[QFunction[d], False]], AppendTo[b[[2]], d], h = ToString[Quiet[DefFunc[d]]];
          If[! SameQ[h, "Null"] && h == "Attributes[" <> d <> "] = {Temporary}", AppendTo[b[[3]], d]], AppendTo[b[[4]], d]]];
      c++]; b]
End[]

Begin["`CurrentNames`"]
CurrentNames[x___] := Module[{a = Complement[Names["*"], Names["System`*"]], b, c, d = {}}, b = Map[{Context[#], #} &, a];
    c = Gather[b, #1[[1]] == #2[[1]] &]; c = Map[DeleteDuplicates[Flatten[#]] &, c];
    If[{x} ≠ {} && ! HowAct[x], Map[Do[AppendTo[d,
          {#[[1]], If[MemberQ[{$Failed, "Undefined"}, PureDefinition[#[[k]]]], #[[k]], Nothing]}], {k, 2, Length[#]}] &, c];
```

```
    x = Map[DeleteDuplicates[Flatten[#]] &, Gather[d, #1[[1]] == #2[[1]] &]], Null]; c]
End[]

Begin["`OverLap`"]
OverLap[x_ /; StringQ[x], y__] := Module[{a, b, c = {}, d = {x, y}, p = 0, h = {}, k = 1},
  If[! StringQ[d[[2]]], Return[Defer[OverLap[{x, y}]]], {a, b} = Map[StringLength, {x, d[[2]]}];
  For[k, k ≤ Min[a, b], k++, If[StringTake[d[[2]], k] ≠ StringTake[x, -k], Continue[], h = Append[h, k]]];
  If[p == Min[a, b], Return[$Failed], If[Length[d] ≥ 3, If[! HowAct[d[[3]]], ToExpression[ToString[d[[3]]] <> "=" <>
        ToString1[StringTake[d[[2]], If[h == {}, Return[$Failed], h[[-1]]]]]], Return[Defer[OverLap[{x, y}]]], Null]]];
  If[h == {}, Return[$Failed], h[[-1]]]]
End[]

Begin["`NbCallProc`"]
NbCallProc[x_ /; BlockFuncModQ[x]] :=
 Module[{a = SubsDel[StringReplace[ToString1[DefFunc[x]], "\n \n" -> ";"], "`" <> ToString[x] <> "`", {"[", ","}, -1]}, Clear[x];
  ToExpression[a]]
End[]

Begin["`ExtrFromM`"]
ExtrFromM[x_ /; FileExistsQ[x] && FileExtension[x] == "m",
  y_ /; SymbolQ[y] || ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}, z___] :=
 Module[{a = ReadString[x], b = DeleteDuplicates[Map[ToString, Flatten[{y}]]], c, d = {}, h = {}, g = {}, s = {}, t = {}, k = 1},
  c = Map[Flatten[{StringCases[a, "(*" <> # <> "[" ~~ Shortest[X__] ~~ "*)"],
        StringCases[a, "(*" <> # <> "::usage" ~~ Shortest[X__] ~~ "*)"]} &, b];
  For[k, k ≤ Length[c], k++, If[c[[k]] == {}, AppendTo[t, b[[k]]], If[Length[c[[k]]] == 1,
    If[SuffPref[c[[k]][[1]], "(*" <> b[[k]] <> "::usage", 1], AppendTo[d, b[[k]]], AppendTo[h, b[[k]]]],
    AppendTo[s, b[[k]]]; AppendTo[g, {StringTake[c[[k]][[1]], {3, -3}], StringTake[c[[k]][[2]], {3, -3}]}]]];
  Quiet[Map[ToExpression, g]]; If[{z} ≠ {} && ! HowAct[z], z = {s, d, h, t}, Null]; ]
End[]

Begin["`ExtrOfMfile`"]
ExtrOfMfile[f_ /; FileExistsQ[f] && FileExtension[f] == "m", s_ /; StringQ[s] || ListQ[s], z___] :=
 Module[{Vsv, p = {}, v, m}, m = ReadFullFile[f];
  If[StringFreeQ[m, Map["(*Begin[\"`" <> # <> "`\"]*)" &, Map[ToString, s]]], $Failed,
   Vsv[x_, y_] := Module[{a = m, b = FromCharacterCode[17], c = FromCharacterCode[24],
      d = "(*Begin[\"`" <> y <> "`\"]*)", h = "(*End[]*)", g = {}, t}, a = StringReplace[a, h → c];
     If[StringFreeQ[a, d], $Failed, While[! StringFreeQ[a, d], a = StringReplace[a, d → b, 1];
       t = StringTake[SubStrSymbolParity1[a, b, c][[1]], {4, -4}];
       t = StringReplace[t, {"(*" → "", "*)" → ""}]; AppendTo[g, t]; a = StringReplace[a, b → "", 1]; Continue[]];
      {g, ToExpression[g[[-1]]]}]];
   If[StringQ[s], v = Quiet[Check[Vsv[f, s][[1]], $Failed]], Map[{v = Quiet[Check[Vsv[f, #][[1]], $Failed]], AppendTo[p, v]} &,
     Map[ToString, s]]]; If[{z} ≠ {} && ! HowAct[z], z = If[StringQ[s], v, p]];]]
End[]

Begin["`ExtrDefFromM`"]
ExtrDefFromM[x_ /; SymbolQ[x], y_ /; FileExistsQ[y] && FileExtension[y] == "m", z___] :=
 Module[{a = ReadString[y], b = "(*Begin[\"`$`\"]*)", c = "(*End[]*)", d = ToString[x], h = "(*$::usage=" , c1, h1},
  c = StringCases[a, Shortest[StringReplace[b, "$" → d] ~~ __ ~~ c]];
  h = StringCases[a, Shortest[StringReplace[h, "$" → d] ~~ __ ~~ "*)"]];
  h1 = If[h == {}, "Usage for " <> d <> " is absent", StringReplace[h[[1]], {"(*" -> "", "*)" -> "", d <> "::usage=" → ""}]];
  c1 = If[c == {}, "Definition for " <> d <> " is absent", c = StringReplace[c[[1]], {"\r\n" -> "", "(*" -> "", "*)" -> ""}];
     c = StringTake[c, {StringLength[d] + 12, -6}]];
  If[h == {} && c == {}, $Failed, If[{z} ≠ {}, Map[ToExpression, {d <> "::usage=" <> h1, c}], Null]; Table[{ToExpression[h1], c1}, 1]]]
End[]

Begin["`ExtrFromNBfile`"]
ExtrFromNBfile[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]], n_ /; StringQ[n]] :=
 Module[{a = ToString1[Get[x]], b = "`" <> n <> "`", c, d, p},
  If[StringFreeQ[a, "\\\"`" <> n <> "`\\\"\""], $Failed, a = StringTake[a, {Flatten[StringPosition[a, "Notebook["]][[1]], -1}];
   c = StringCases[a, Shortest[
       "RowBox[{\"Begin\", \"[\", \"\\\"" <> b <> "\\\"\", \"]\"}]" ~~ __ ~~ "RowBox[{\"End\", \"[\", \"]\"}]" ]][[1]];
   p = If[StringFreeQ[c, "[IndentingNewLine]"], 74, 69]; c = StringReplace[c, ", \"\\[IndentingNewLine]" -> ""];
   d = Map[#[[1]] &, StringPosition[c, "RowBox"]]; c = StringTake[c, {d[[2]], If[p == 74, d[[-1]] - 9, d[[-2]] - 4]}];
   d = DisplayForm[ToExpression[c]];
   CDFDeploy[n, d, "Target" -> "CDFPlayer"];
   d = StringTake[ToString[Import[n <> ".cdf", "Plaintext"]], {1, -5}];
   DeleteFile[{n <> ".cdf", n <> ".png"}]; d]]
End[]
```

```
Begin["`ExtrFromNBfile1`"]
ExtrFromNBfile1[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]], n_ /; StringQ[n], m___] :=
  Module[{a = ToString[InputForm[Get[x]]], c = "`" <> n <> "`", b = "RowBox[{\"End\", \"[\", \"]\"}]", d, h, t},
    d = StringCases[a, c ~~ Shortest[X__] ~~ b];
    If[d == {}, {}, d = StringTake[d, {Flatten[StringPosition[d, "RowBox[{""]][[1]], -1}];
      d = StringReplace[d[[1]], b -> ""]; h = d; While[! SyntaxQ[h], h = StringTake[h, {1, -2}]];
      NotebookEvaluate[NotebookOpen[x, Visible → False], EvaluationElements → All];
      h = DisplayForm[ToExpression[t = h]]; CDFDeploy[n, h, "Target" -> "CDFPlayer"]; DeleteFile[ n <> ".png"];
      If[{m} ≠ {} && ! HowAct[m], m = Directory[] <> "\\" <> n <> ".cdf", Null]; h]]
End[]

Begin["`ExtrFromNBfile2`"]
ExtrFromNBfile2[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]], n_ /; StringQ[n]] :=
  Module[{a = ContextFromFile[x], b, c, d},
    If[a === $Failed, $Failed, d = If[PathToFileQ[x], x, Directory[] <> "\\" <> x];
      If[MemberQ[$Packages, a], c = Map[StringReplace[#, a <> n <> "`" -> ""] &, Flatten[{PureDefinition[n]}]];
        If[ListQ[c] && Length[c] == 1, c[[1]], c], Quiet[Activate[ExtrFromNBfile1[d, n]]];
        If[! MemberQ[CNames[a], n], $Failed, b = Flatten[{PureDefinition[n]}];
          c = Map[StringReplace[#, a <> n <> "`" -> ""] &, b]; RemovePackage[a]; Unprotect[n];
          Quiet[Remove[n]]; If[ListQ[c] && Length[c] == 1, c[[1]], c]]]]
End[]

Begin["`ExtrFromNBfile3`"]
ExtrFromNBfile3[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]],
    n_ /; StringQ[n] || ListQ[n] && n ≠ {} && DeleteDuplicates[Map[StringQ[#] &, Flatten[{n}]]] == {True}, y___] :=
  Module[{a = Get[x], b, c, d = {}, g = {}, h = Flatten[{n}], k = 1}, b = DisplayForm[a];
    CDFDeploy[c = "$Art27$Kr20$.cdf", b, "Target" -> "CDFPlayer"];
    b = ToString1[Import[c, "Plaintext"]];
    Quiet[DeleteFile[{c, FileBaseName[c] <> ".png"}]];
    For[k, k ≤ Length[h], k++, If[StringFreeQ[b, "Begin[\\\"`" <> h[[k]] <> "`\\\"]"], AppendTo[g, {h[[k]], $Failed}],
        c = DeleteDuplicates[StringCases[b, Shortest["Begin[\\\"`" <> h[[k]] <> "`\\\"]" ~~ __ ~~ "End[]"]]];
        c = Map[StringTake[#, {Flatten[StringPosition[#, h[[k]] <> "[""]][[1]], -6}] &, c];
        c = Map[StringReplace[#, {"\\r" -> "", "\\n" -> "", "\\" -> ""}] &, c]; AppendTo[d, c]]];
    d = Flatten[d]; If[{y} ≠ {} && ! HowAct[y], y = If[Length[g] == 1, g[[1]], g], Null]; If[Length[d] == 1, d[[1]], d]]
End[]

Begin["`CdfNbToText`"]
CdfNbToText[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]], y___] :=
  Module[{a = Get[x], b, c}, b = DisplayForm[a];
    CDFDeploy[c = "$Art27$Kr20$.cdf", b, "Target" -> "CDFPlayer"];
    b = ToString1[Import[c, "Plaintext"]];
    Quiet[DeleteFile[{c, FileBaseName[c] <> ".png"}]];
    c = DirectoryName[x] <> FileBaseName[x] <> ".txt";
    b = StringReplace[b, {"\\r" → "", "\\n" -> "\n", "\\" -> ""}]; WriteString[c, b]; Close[c]; If[{y} ≠ {}, {c, b}, c]]
End[]

Begin["`EvaluateCdfNbFile`"]
EvaluateCdfNbFile[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]], y___] :=
  NotebookEvaluate[NotebookOpen[x, If[{y} ≠ {}, Visible → True, Visible → False]], EvaluationElements → All]
End[]

Begin["`ContentsCdfNb`"]
ContentsCdfNb[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]]] :=
  Module[{a = ToString1[Get[x]], b = "RowBox[{\"Begin\", \"[\", \"\\\"`\", c = "`\\\"\", \"]\"}]", d},
    d = StringCases[a, b ~~ Shortest[X__] ~~ c];
    Sort[Map[StringReplace[#, {b -> "", c -> ""}] &, d]]]
End[]

Begin["`ContentsMx`"]
ContentsMx[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a = ContextInMxFile[x], b}, If[a === $Failed, $Failed, If[MemberQ[$Packages, a], CNames[a], Get[x];
      b = CNames[a]; RemovePackage[a]; b]]]
End[]

Begin["`ContentOfNbCdf`"]
ContentOfNbCdf[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]], y___] :=
  Module[{a = ToString[Get[x]], b, c = {}, d = {}}, b = Sort[DeleteDuplicates[Select[Map[StringTrim[#, {" \"`", "`\""}] &, StringCases[a,
        {Shortest[" \"`" ~~ __ ~~ "`\""], Shortest["`" ~~ __ ~~ "`"]}, Overlaps → True]], SymbolQ[#] && # ≠ "Private" &]]];
    If[{y} ≠ {} && ! HowAct[y], Map[If[! StringFreeQ[a, "{" <> # <> ", ::, usage}"], AppendTo[c, #], AppendTo[d, #]] &, b];
```

```
      y = {c, d, Select[Select[Map[SubsPosSymb[a, # - 1, {"{"}, 0] &, Map[First, StringPosition[a, ", ::, usage}"]]], ! MemberQ[b, #] &],
            SymbolQ[#] && # != "p"  &]};  b, b]]
End[]

Begin["`TestCdfNbFile`"]
TestCdfNbFile[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]], y___] :=
  Module[{a = ToString[InputForm[Get[x]]], b = "RowBox[{\"Begin\", \"[\", \"\\\"`\",
      c = "`\\\"\", \"]\"}]", d = "RowBox[{\"590\", \"::\", \"usage\"}]" , h = {}, p},
    p = StringCases[a, b ~~ Shortest[X__] ~~ c]; p = Map[StringReplace[#, {b -> "", c -> ""}] &, p];
    Map[If[StringFreeQ[a, StringReplace[d, "590" → #]], AppendTo[h, #], Null] &, p];
    If[{y} ≠ {} && ! HowAct[y], y = Sort[p], Null];  h]
End[]

Begin["`UsagesCdfNb`"]
UsagesCdfNb[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]],
    y_ /; SymbolQ[y] || ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}] :=
  Module[{a = ToString[InputForm[Get[x]]], b = "RowBox[{RowBox[{\"590\", \"::\", \"usage\"}], \"=\", ", c = "}], \"
          \"}]", b1 = "RowBox[{RowBox[{\"590\", \"::\", \"usage\"}], \"=\", ", c1 = "}]", f, p, g},
    f[t_] := Module[{z = StringReplace[b, "590" → t], z1 = StringReplace[b1, "590" → t], h, d},
      d = Join[StringCases[a, z ~~ Shortest[X__] ~~ c], StringCases[a, z1 ~~ Shortest[X__] ~~ c1]];
      If[d == {}, {}, ToExpression[ToExpression[StringReplace[d[[1]], {z -> "", c -> "", z1 -> "", c1 -> ""}]]]];
    g = Map[{p = f[#], If[p ===  {}, # <> "::" <> "No" <> "\n", # <> "::" <> p <> "\n"]}[[2]] &, Map[ToString, Flatten[{y}]]];
    If[Length[g] == 1, g[[1]], StringJoin[g]]
End[]

Begin["`ActivateMeansFromCdfNb`"]
ActivateMeansFromCdfNb[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]],
    y_ /; SymbolQ[y] || ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}] :=
  Module[{a = Map[ToString, Flatten[{y}]], b, c}, b = UsagesCdfNb[x, a];
    b = StringSplit["\n" <> b, Map["\n" <> # <> "::" -> # <> "::" &, a]];
    Do[ToExpression[b[[2 * k - 1]] <> "usage=" <> ToString1[b[[2 * k]]]], {k, 1, Length[b] / 2}];
    Map[Quiet[{ClearAll[c], ExtrFromNBfile1[x, #, c], DeleteFile[c]}] &, a];]
End[]

Begin["`ExtrFromMfile`"]
ExtrFromMfile[x_ /; FileExistsQ[x] && FileExtension[x] == "m",
    y_ /; SymbolQ[y] || ListQ[y] && DeleteDuplicates[Map[SymbolQ, y]] == {True}] :=
  Module[{a = ReadString[x], b, c, d, d1, n}, b = StringSplit[a, {"(**)", "(* ::Input:: *)"}];
    b = Map[If[! StringFreeQ[#, {"::usage=", "BeginPackage[\"",  "End[]"}], #, Null] &, b];
    b = Select[b, ! SameQ[#, Null] &];  c = Map[ToString, Flatten[{y}]];  d = Map["Begin[\"`" <> # <> "`\"]" &, c];
    d1 = Map[ "(*" <> # <> "::usage=" &, c];  b = Select[b, ! StringFreeQ[#, Join[d, d1]] &];  b = Map[StringTake[#, {3, -5}] &, b];
    c = Map[If[SuffPref[#, d1, 1], StringTake[#, {3, -1}], n = StringReplace[#, GenRules[d, ""]];
          n = StringReplace[n, "*)\r\n(*" -> ""];  StringTake[n, {3, -6}]] &, b];  ToExpression[c];  c]
End[]

Begin["`AtomicQ`"]
AtomicQ[x_] := Module[{F}, If[Map[F, x] === x, True, False]]
End[]

Begin["`SymbolGreater`"]
SymbolGreater[x_ /; SymbolQ[x], y_ /; SymbolQ[y]] :=
  Greater[ToExpression[StringJoin[Map[ToString, ToCharacterCode[ToString[x]]]]],
    ToExpression[StringJoin[Map[ToString, ToCharacterCode[ToString[y]]]]]]
End[]

Begin["`SymbolLess`"]
SymbolLess[x_ /; SymbolQ[x], y_ /; SymbolQ[y]] := Less[ToExpression[StringJoin[Map[ToString, ToCharacterCode[ToString[x]]]]],
    ToExpression[StringJoin[Map[ToString, ToCharacterCode[ToString[y]]]]]]
End[]

Begin["`StrFromStr`"]
StrFromStr[x_ /; StringQ[x]] := Module[{a = "\"", b, c = {}, k = 1}, b = DeleteDuplicates[Flatten[StringPosition[x, a]]];
    For[k, k ≤ Length[b] - 1, k++, c = Append[c, ToExpression[StringTake[x, {b[[k]], b[[k + 1]]}]]];  k = k + 1]; c]
End[]

Begin["`StringStringQ`"]
StringStringQ[x_] := If[! StringQ[x], False, If[SuffPref[x, "\"", 1] && SuffPref[x, "\"", 2], True, False]]
End[]
```

```
Begin["`StringLevels`"]
StringLevels[x_String] := Module[{a = x, n = -1}, While[StringQ[a], a = ToExpression[a];
      n++;
      Continue[]]; n]
End[]

Begin["`Npackage`"]
Npackage[x_ /; StringQ[x]] := If[MemberQ[Contexts1[], x],
    Sort[Select[Names[x <> "*"], StringTake[#, -1] ≠ "$" && ToString[Definition[#]] ≠ "Null" &]], $Failed]
End[]

Begin["`LocObj`"]
LocObj[x_] := Module[{a = Head1[x], b}, b[y_] := Context[y]; If[a === Module, {x, "Module", b[x]},
      If[a === Function, {x, "Function", b[x]}, If[SystemQ[x], {x, "SFunction", b[x]}, {x, "Expression", "Global"}]]]]
End[]

Begin["`Affiliate`"]
Affiliate[x_ /; StringQ[x]] := Module[{a = Quiet[Context[x]]}, If[ToString[a] === "Context[" <> x <> "]", "Undefined",
      If[MemberQ[Contexts[], a] && ToString[Quiet[DefFunc[x]]] == "Null" || Attributes[x] === {Temporary}, "Undefined", a]]]
End[]

Begin["`LoadNameFromM`"]
LoadNameFromM[F_ /; FileExistsQ[F] && FileExtension[F] == "m"  && StringTake[ToString[ContextFromFile[F]], -1] == "`",
    p_ /; StringQ[p] || ListStringQ[p]] := Module[{a = ReadFullFile[F], b = {}, c = "*)(*End[]*)", d, h = Flatten[{p}]},
    b = Map[SubsString[a, {"(*Begin[\"`" <> # <> "`\"]*)(*", c}, 90] &, h];
    b = If[Length[b] == 1, Flatten[b], Map[#[[1]] &, b]]; Map[ToExpression, Map[StringReplace[#, "*)(*" → " "] &, b]];]
End[]

Begin["`OpenFiles`"]
OpenFiles[x___String] :=
  Module[{a = Streams[][[3 ;; -1]], b, c, d, h1 = {"read"} , h2 = {"write"} }, If[a == {}, {}, d = Map[{Part[#, 0], Part[#, 1]} &, a];
      b = Select[d, #[[1]] == InputStream &]; c = Select[d, #[[1]] == OutputStream &];
      b = Map[DeleteDuplicates, Map[Flatten, Gather[Join[b, c], #1[[1]] == #2[[1]] &]]];
      b = Map[Flatten, Map[If[SameQ[#[[1]], InputStream], AppendTo[h1, #[[2 ;; -1]]], AppendTo[h2, #[[2 ;; -1]]]] &, b]];
      If[{x} == {}, b,
        If[SameQ[FileExistsQ[x], True], c = Map[Flatten, Map[{#[[1]], Select[#, StandPath[#] === StandPath[x] &]} &, b]];
          If[c == {{"read"}, {"write"}}, {}, c = Select[c, Length[#] > 1 &]; If[Length[c] > 1, c, c[[1]]], {}]]]]]
End[]

Begin["`FilesDistrDirs`"]
FilesDistrDirs[x_ /; DirQ[x]] := Module[{a = {}, b, c = FromCharacterCode[17], d, g, h = {}, f = "$Art27Kr20$", k = 1, t},
    Run["Dir " <> StandPath[x] <> " /A/B/OG/S > " <> f];
    For[k, k < Infinity, k++, b = Read[f, String]; If[SameQ[b, EndOfFile], DeleteFile[Close[f]];
        Break[], AppendTo[a, b]]];
    b = Gather[PrependTo[a, StringReplace[x, "/" -> "\\"]], DirQ[#1] === DirQ[#2] &];
    d = {Sort[Map[StringJoin[#, "\\"] &, b[[1]]], StringCount[#1, "\\"] ≥ StringCount[#2, "\\"] &], Quiet[Check[b[[2]], {}]]};
    a = Map[ToLowerCase, Flatten[d]];
    For[k = 1, k ≤ Length[d[[1]]], k++, t = ToLowerCase[d[[1]][[k]]];
      AppendTo[h, g = Select[a, SuffPref[#, t, 1] && StringFreeQ[StrDelEnds[#, t, 1], "\\"] &]];
      a = MinusList[a, g]]; a = {}; For[k = 1, k ≤ Length[h], k++, b = h[[k]];
      AppendTo[a, {b[[1]], Map[StrDelEnds[#, b[[1]], 1] &, b[[2 ;; -1]]]}]; Map[Flatten[#] &, a]]
End[]

Begin["`DirFilePaths`"]
DirFilePaths[x_ /; StringQ[x], y_: BootDrive1[][[1]] <> "\\*.*"] :=
  Module[{c = {}, h, d = {}, b = ToString[Unique["avz"]], a = StringTrim[StandStrForm[x], "\\"]},
    Run["DIR /A/B/S " <> StandPath[y] <> " > " <> b];
    h = ReadList[b, String];
    DeleteFile[b]; Map[If[! StringFreeQ[StandPath[#], {"\\" <> a <> "\\", "\\" <> a}],
        If[DirectoryQ[#], AppendTo[c, #], AppendTo[d, #]], Null] &, h];
    {c, d}]
End[]

Begin["`MathematicaDF`"]
MathematicaDF[] :=
  Module[{a = "Art27$Kr20$", b = {}, c = "", d}, Run["Dir " <> " /A/B/S " <> StrStr[$InstallationDirectory] <> " > " <> a];
    While[! SameQ[c, EndOfFile], c = Read[a, String];
      Quiet[If[DirectoryQ[c], AppendTo[b, "dir"], If[FileExistsQ[c], d = ToLowerCase[FileExtension[c]];
          AppendTo[b, If[d === "", "NoExtension", d]], AppendTo[b, "NoFile"]]]]; DeleteFile[Close[a]];
```

```
    Sort[Map[{#[[1]], Length[#]} &, Gather[b, #1 === #2 &]], Order[#1[[1]], #2[[1]]] == 1 &]]
End[]

Begin["`Memory`"]
Memory[x___] := Module[{a = "$Memory$", b, c = "mem.exe", d}, b =  $InstallationDirectory <> "\\" <> c;
    If[FileExistsQ[b], c = Run[c, " > ", a], d = LoadExtProg[c];
      If[d === $Failed, Return[$Failed], c = Run[c, " > ", a]]];
    If[{x} != {}, DeleteFile[b]];
    If[c != 0, DeleteFile[a];
      $Failed, c = Map[StringTrim, ReadList[a, String]];
      DeleteFile[a]; c]]
End[]

Begin["`EmptyFileQ`"]
EmptyFileQ[f_ /; StringQ[f], y___] := Module[{a, b, c, d = {}, k = 1}, If[FileExistsQ[f], b = {f}, c = Art27$Kr20;
      ClearAll[Art27$Kr20];
      a = FileExistsQ1[f, Art27$Kr20]];
    If[! a, Return[$Failed], b = Art27$Kr20; Art27$Kr20 = c];
    While[k ≤ Length[b], AppendTo[d, Quiet[Close[b[[k]]]];
        If[Quiet[Read[b[[k]]]] === EndOfFile, Quiet[Close[b[[k]]];
            True], Quiet[Close[b[[k]]]];
          False]]; k++];
    d = If[Length[d] == 1, d[[1]], d];  If[{y} ≠ {}, {d, If[Length[b] == 1, b[[1]], b]}, d]]
End[]

Begin["`DeleteFile1`"]
DeleteFile1[x_ /; StringQ[x] || ListQ[x], y___] := Module[{a = Map[ToString, Flatten[{x}]], b, c = $ArtKr$, d, p, t},
    b = If[! StringFreeQ[Ver[], " XP "], FilesDistrDirs[BootDrive[][[1]] <> ":\\Recycler"][[1]],
        p = 90; ClearAll[$ArtKr$]; If[FileExistsQ1["$recycle.bin", $ArtKr$], d = $ArtKr$[[1]], Return[$Failed]];
        b = SortBy[Select[Flatten[FilesDistrDirs[d]], DirectoryQ[#] &], Length[#] &][[2]]; $ArtKr$ = c;
    c = Map[StandPath, Map[If[StringFreeQ[#, ":"], Directory[] <> "\\" <> #, #] &, a]];
    t = Map[Run["Copy /Y " <> # <> " " <> If[p == 90, b <> FileNameSplit[#][[-1]], b[[1]]]] &, c];
    t = Position[t, 1];  c = If[t ≠ {}, MinusList[c, b = Extract[c, t]], Null];
     If[t ≠ {} && {y} ≠ {} && ! HowAct[y], Quiet[y = b], Quiet[y = {}]];  Map[{Attrib[#, {}], Quiet[DeleteFile[#]]} &, c];  0]
End[]

Begin["`ClearRecycler`"]
ClearRecycler[x___] := Module[{b = {}, c = "", k = 2, d = BootDrive[][[1]] <> ":\\Recycler", a}, a = FilesDistrDirs[d];
    If[{x} ≠ {} && x === "ALL", Run["Dir " <> d <> " /A/B/S/W > Dir.txt"];
      While[! SameQ[c, EndOfFile], c = Read["Dir.txt", String];  AppendTo[b, c];  Continue[]];
      DeleteFile[Close["Dir.txt"]];  b = b[[1 ;; -2]];
      For[k = 1, k ≤ Length[b], k++, c = b[[k]];
        If[DirectoryQ[c], DeleteDirectory[c, DeleteContents → True], Run["Del /F /A[:AHR] " <> c]]];
      0,  a = FilesDistrDirs[d][[1]];
If[a == {"c:\\recycler\\"}, 0,  While[k ≤ Length[a], c = FileBaseName[a[[k]]];
          If[c ≠ "desktop" && ! StringMatchQ[c, "info" ~~ ___ ~~ NumberString] &&
              ! StringMatchQ[c, "dc" ~~ ___ ~~ NumberString], AppendTo[b, a[[k]]]];
          k++];
        c = Map[StringJoin[a[[1]], #] &, b];
        c = Map[Run["Del " <> StandPath[#]] &, c]; If[MemberQ[{{0}, {}}, Flatten[c]], 0, 0]]]
End[]

Begin["`ClearRecyclerBin`"]
ClearRecyclerBin[x___] := Module[{a, b = ToString[Unique["ag"]], c = $ArtKr$, d, p}, ClearAll[$ArtKr$];
    If[! FileExistsQ1["$recycle.bin", $ArtKr$], $Failed, d = StandPath[$ArtKr$[[1]]];
      $ArtKr$ = c;  Run["Dir " <> d <> "/B/S/L > " <> b];  p = ReadList[b, String];  DeleteFile[b];
      If[p == {}, Return[$Failed],
        Map[If[{x} == {}, If[SuffPref[a = FileNameSplit[#][[-1]], "$", 1] || a === "desktop.ini", Null,  Attribs[#, 90];
                  If[FileExistsQ[#], Quiet[Check[DeleteFile[#], DeleteDirectory[#, DeleteContents → True]]],
                    Quiet[Check[DeleteDirectory[#, DeleteContents → True], DeleteFile[#]]]]],
                If[FileNameSplit[#][[-1]] == "desktop.ini", Null, Attribs[#, 90];
                  If[DirQ[#], Run["RD /S/Q " <> #], Run["DEL /F/Q " <> #]]]] &, p]; ]]
End[]

Begin["`StandPath`"]
StandPath[x_ /; StringQ[x]] :=
  Module[{a, b = "", c, k = 1}, If[MemberQ[Flatten[Outer[StringJoin, CharacterRange["a", "z"], {":/", ":\\"}]], c = ToLowerCase[x]],
      StringReplace[c, "/" -> "\\"],
      If[PathToFileQ[x], a = FileNameSplit[StringReplace[ToLowerCase[ToLowerCase[x]], "/" → "\\"]];
```

```
        For[k, k ≤ Length[a], k++, c = a[[k]]; If[! StringFreeQ[c, " "], b = b <> StrStr[c] <> "\\", b = b <> c <> "\\"]];
        StringTake[b, {1, -2}], ToLowerCase[x]]]]
End[]

Begin["`NestQL`"]
NestQL[L_ /; ListQ[L]] := MemberQ[Map[ListQ[#] &, L], True]
End[]

Begin["`SelectPos`"]
SelectPos[x_ /; ListQ[x], y_ /; ListQ[y] && DeleteDuplicates[Map[IntegerQ[#] && # > 0 &, y]] == {True}, z_ /; MemberQ[{1, 2}, z]] :=
  Select[x, If[If[z == 2, Equal, Unequal][Intersection[Flatten[Position[x, #]], y], {}], False, True] &]
End[]

Begin["`ElemsList`"]
ElemsList[x_ /; ListQ[x], y_ /; ListQ[y]] :=
  Module[{a = Select[y, ! ListQ[#] &], b = Select[y, ListQ[#] &], c = "", k = 1}, If[a == {} && b == {}, x,
    If[a == {}, Quiet[Check[ToExpression[ToString[x] <> "[[" <> StringTake[ToString[b], {3, -3}] <> "]]"], $Failed]],
      If[b == {}, c = ToString[x];
        While[k ≤ Length[a], c = c <> "[[" <> ToString[a[[k]]] <> "]]";
          k++];
        Quiet[Check[ToExpression[c], $Failed]],
        c = ToString[x]; While[k ≤ Length[a], c = c <> "[[" <> ToString[a[[k]]] <> "]]"; k++];
        Quiet[Check[ToExpression[c <> "[[" <> StringTake[ToString[b], {3, -3}] <> "]]"], $Failed]]]]]
End[]

Begin["`PosListTest`"]
PosListTest[L_ /; ListQ[L], p_ /; PureFuncQ[p]] :=
  Module[{a = {}, k = 1}, While[k ≤ Length[L], If[Select[{L[[k]]}, p] ≠ {}, AppendTo[a, k]];
    k++]; a]
End[]

Begin["`FirstPositionsList`"]
FirstPositionsList[x_ /; ListQ[x]] := DeleteDuplicates[Map[#[[1]][[1]] &, Map[SequencePosition[x, {#}] &, x]]]
End[]

Begin["`SeqQ`"]
SeqQ[x_] :=
  Module[{a = ToString[x]}, If[StringLength[a] ≥ 4 && StringTake[a, {1, 4}] == "Seq[" && StringTake[a, {-1, -1}] == "]", True, False]]
End[]

Begin["`SeqToList`"]
SeqToList[x_ /; SeqQ[x]] := ToExpression["{" <> StringTake[ToString[x], {5, -2}] <> "}"]
End[]

Begin["`ListCount`"]
ListCount[x_ /; ListQ[x], y__List] := StringCount[ToString[x], Mapp[StringTake, Map[ToString, {y}], {2, -2}]]
End[]

Begin["`ListToSeq`"]
ListToSeq[x_ /; ListQ[x]] := ToExpression["Seq[" <> StringTake[ToString[x], {2, -2}] <> "]"]
End[]

Begin["`TransListList`"]
TransListList[L_ /; ListListQ[L]] := Module[{a = Length[L], b = Length[L[[1]]], c = {}, d = {}, k = 1, j},
    For[k, k ≤ b, k++, For[j = 1, j ≤ a, j++, d = Append[d, L[[j]][[k]]]]; c = Append[c, d]; d = {}];
    c]
End[]

Begin["`ElemLevelsL`"]
ElemLevelsL[x_ /; ListQ[x]] := Module[{a = x, c = {}, m = {}, n, k = 0}, While[NestListQ1[a], n = Select[a, ! ListQ[#] &];
      AppendTo[c, {k++, Complement[n, m]}];
      m = n;
      a = Flatten[a, 1]; Continue[]];
    Append[c, {k++, Complement[a, m]}]]
End[]

Begin["`ElemLevelsN`"]
ElemLevelsN[x_ /; ListQ[x]] := Module[{a = x, c = {}, m = 0, n, k = 0}, While[NestListQ1[a], n = Length[Select[a, ! ListQ[#] &]];
      AppendTo[c, {k++, n - m}]; m = n; a = Flatten[a, 1]; Continue[]]; Append[c, {k++, Length[a] - m}]]
```

```
End[]

Begin["`SeqIns`"]
SeqIns[x_ /; SeqQ[x], y_, z_ /; IntegerQ[z]] := Module[{a = SeqToList[x], b = {}, c = If[SeqQ[y], SeqToList[y], y]},
    If[z ≤ 0, b = Append[c, a], If[z ≥ Length[a], b = Append[a, c], b = Join[a[[1 ;; z]], c, a[[z + 1 ;; -1]]]]];
    ListToSeq[Flatten[b]]]
End[]

Begin["`UniqueV`"]
UniqueV[x_ /; SymbolQ[x], y_] := Module[{a = ToString[Unique[ToString[x]]]}, ToExpression[a <> "= " <> ToString1[y]];
    a]
End[]

Begin["`SubProcs`"]
SubProcs[P_ /; BlockModQ[P]] := Module[
    {b, c = {}, d, t, h, k = 1, p = {}, g = {}, a = Flatten[{PureDefinition[P]}][[1]]}, b = StringPosition[a, {"] := Block[{", "] := Module[{"}];
    For[k, k ≤ Length[b], k++, d = b[[k]]; AppendTo[p, ExprOfStr[a, d[[1]], -1, {" ", ",", ";"}]];
      AppendTo[c, h = ExprOfStr[a, d[[1]], -1, {" ", ",", ";"}] <> " := " <> ExprOfStr[a, d[[1]] + 5, 1, {" ", ",", ";"}];
        t = Flatten[StringPosition[h, "["]];
        h = Quiet[StringReplacePart[h, ToString[Unique[ToExpression[StringTake[h, {1, t[[1]] - 1}]]]], {1, t[[1]] - 1}]];
        AppendTo[g, StringTake[h, {1, Flatten[StringPosition[h, "["]][[1]] - 1}]];
        h]; Map[ToExpression, c];
    {p, Map[ToExpression, g]}]
End[]

Begin["`SubProcs1`"]
SubProcs1[x_ /; BlockFuncModQ[x]] :=
  Module[{b = {}, c, d, k = 1, a = Flatten[{PureDefinition[x]}]}, For[k, k ≤ Length[a], k++, c = a[[k]];
      d = StringPosition[c, {"] := Module[{", "] := Block[{"}]; If[d == {}, Continue[]];
      AppendTo[b, {StringTake[c, {1, d[[1]][[1]]}], Length[d] - 1}]]; If[Length[b] == 1, Flatten[b], b]]
End[]

Begin["`SubProcs2`"]
SubProcs2[y_, z___] := Module[{n = {}, m = 1, SB, v = Flatten[{PureDefinition[y]}]}, If[BlockFuncModQ[y], SB[x_String] :=
        Module[{b = "Module[", c, d, h, g = "", t, k, p, q, j, s, w, a = Map[#[[1]] &, StringPosition[x, "Module[{"]]}, If[a == {}, Return[]];
          If[Length[a] == 1, Return[$Failed], d = Map[# - 5 &, a]]; c = {StringTake[x, {1, d[[1]]}]};
          For[k = Length[a], k > 1, k--, h = b;
            g = "";
            t = ""; For[j = a[[k]] + 7, j < Infinity, j++, h = h <> StringTake[x, {j, j}];
              If[SameQ[Quiet[Check[ToExpression[h], "Error"]], "Error"],
                Continue[], For[j = d[[k]], j > 1, j--, g = StringTake[x, {j, j}] <> g;
                  If[SameQ[Quiet[Check[ToExpression[g], "Error"]], "Error"], Continue[], Break[]]];
                While[j > 1, p = StringTake[x, {j, j}]; If[! SameQ[p, " "], t = p <> t, Break[]];
                  j--]; p = StringPosition[x, " " <> t <> "["][[1]];
                s = Flatten[SubStrSymbolParity1[StringTake[x, {p[[1]], -1}], "[", "]"]]; w = 1;
                While[w ≤ Length[s] - 1, q = s[[w]]; If[! StringFreeQ[q, "_"], s = t <> q <> " := Module" <> s[[w + 1]];
                    Break[]];
                  w++];
                AppendTo[c, s];
                Break[]]];
            c]; For[m, m ≤ Length[v], m++, AppendTo[n, SB[v[[m]]]]];
        n = Select[n, ! SameQ[#, Null] &]; If[n == {}, $Failed, n = If[Length[n] == 1, n[[1]], n];
          If[{z} ≠ {}, ToExpression[n]];
          n], $Failed]]
End[]

Begin["`NotSubsProcs`"]
NotSubsProcs[x_ /; ModuleQ[x], y___] :=
  Module[{a = SubsProcs[x], at = Attributes[x], b, c, d, h, t = {}}, If[a == {}, {}, b = PureDefinition[x]; ClearAttributes[x, at];
      c = HeadPF[x]; d = Args[x, 90]; h = ToString[Unique["a"]];
      ToExpression[StringReplace[b, c → h <> "[" <> StringTake[ToString[Map[# <> "_" &, d]], {2, -2}] <> "]", 1]];
      Quiet[Check[ToExpression[h <> "[" <> StringTake[ToString[d], {2, -2}] <> "]"], Null]];
      d = Map[StringTake[#, {1, Flatten[StringPosition[#, "[", 1]][[1]] - 1}] &, a];
      If[{y} ≠ {} && ! HowAct[y], y = d, Null];
      Map[If[UnevaluatedQ[HeadPF, #] || ! StringFreeQ3[HeadPF[#], "$_"], AppendTo[t, #], Null] &, d];
      ToExpression[b]; t = MinusList[d, t];
      SetAttributes[x, at]; {t, Quiet[Map[ClearAll, Flatten[{h, d}]]]}[[1]]]]
End[]
```

```
Begin["`$ProcName`"]
$ProcName := Module[{d = "$$ArtKr$$", a, b, c, t = "", k}, a = ToString1[Stack[_]];
    d = Flatten[StringPosition[a, d]][[1]];
    b = Flatten[StringPosition[a, "$$NameProc$$"]][[1]]; If[b > d || ToString[b] == "", Return["UndefinedName"], k = b];
    For[k = b, k ≤ d, k++, c = StringTake[a, {k, k}]; If[MemberQ[{",", "}"}, c], Break[], t = t <> c;
        Continue[]]];
    {b = ToExpression[ToExpression[StringSplit[t, "="][[2]]]], HeadPF[b]}]
End[]

Begin["`ModToPureFunc`"]
ModToPureFunc[x_ /; QBlockMod[x]] := Module[{a, Atr = Attributes[x], c, d, p, O = Options[x],
      b = Flatten[{PureDefinition[x]}][[1]], n = "$$$" <> ToString[x], k = 1, j, t, q = {}}, ToExpression["$$$" <> b];
    c = LocalsGlobals1[Symbol[n]]; a = Args[Symbol[n], 90]; d = StringReplace[PureDefinition[n], HeadPF[n] <> " := " –> "", 1];
    ToExpression["ClearAll[" <> n <> "]"];
    If[c[[3]] ≠ {}, Return[{$Failed, "Globals", c[[3]]}]]; c = Map[{#, ToString[Unique[#]]} &, Join[a, c[[1]]]];
    While[k ≤ Length[c], p = c[[k]];
      d = StringReplaceS[d, p[[1]], p[[2]]];
      k++]; d = ToString[ToExpression[d]];
    t = Map[ToString, UnDefVars[ToExpression[d]]];
    t = Map[StringTake[#, {1, If[StringFreeQ[#, "$"], –1, Flatten[StringPosition[#, "$"]][[1]] – 1]}] &, t];
    k = 1;
    While[k ≤ Length[t], j = 1;
      While[j ≤ Length[c], If[t[[k]] == c[[j]][[2]], AppendTo[q, c[[j]][[1]]]];
        j++];
      k++];
    k = 1;
    While[k ≤ Length[c], p = c[[k]]; d = StringReplaceS[d, p[[2]], p[[1]]]; k++];
    If[p = MinusList[q, a];
      p ≠ {}, {$Failed, "Locals", p}, ToExpression["ClearAll[" <> n <> "]"];
      n = "$$$" <> ToString[x];
      ToExpression[n <> " := Function[" <> ToString[a] <> ", " <> d <> "]"];
      If[Atr ≠ {}, ToExpression["SetAttributes[" <> n <> "," <> ToString[Atr] <> "]"]];
      If[O ≠ {}, ToExpression["SetOptions[" <> n <> "," <> ToString[O] <> "]"]]; n]]
End[]

Begin["`SubProcs3`"]
SubProcs3[y_, z___] := Module[{u = {}, m = 1, Sv, v = Flatten[{PureDefinition[y]}]},
    If[BlockFuncModQ[y], Sv[S_String] := Module[{a = ExtrVarsOfStr[S, 1], b, c = {}, d, t = 2,
          k = 1, cc = {}, n, p, j, h = {StringTake[S, {1, Flatten[StringPosition[S, " := "]][[1]] – 1}]}}, a =
          Select[a, ! SystemQ[Symbol[#]] && ! MemberQ[{ToString[G]}, #] &]; b = StringPosition[S, Map[" " <> # <> "[" &, a]];
        p = Select[a, ! StringFreeQ[S, " " <> # <> "["] &];
        b = Flatten[Map[SubStrSymbolParity1[StringTake[S, {#[[1]], –1}], "[", "]"] &, b]];
        For[j = 1, j ≤ Length[p], j++, n = p[[j]]; For[k = 1, k ≤ Length[b] – 1, k++, d = b[[k]];
            If[! StringFreeQ[d, "_"] && StringTake[b[[k + 1]], {1, 1}] == "[",
              AppendTo[c, Map[n <> d <> " := " <> # <> b[[k + 1]] &, {"Block", "Module"}]]]]];
        c = DeleteDuplicates[Flatten[c]];
        For[k = 1, k ≤ Length[c], k++, d = c[[k]];
          If[! StringFreeQ[S, d], AppendTo[h, d], AppendTo[cc, StringTake[d, {1, Flatten[StringPosition[d, " := "]][[1]] – 1}]]]];
        {h, cc} = Map[DeleteDuplicates, {h, cc}]; p = Map[StringTake[#, {1, Flatten[StringPosition[#, "["]][[1]]}] &, h];
        cc = Select[Select[cc, ! SuffPref[#, p, 1] &], ! StringFreeQ[S, #] &]; If[cc == {}, h, For[k = 1, k ≤ Length[cc], k++, p = cc[[k]];
            p = StringCases[S, p <> " := " ~~ __ ~~ ";"]; AppendTo[h, StringTake[p, {1, Flatten[StringPosition[p, ";"]][[1]] – 1}]]]];
        Flatten[h]];
      For[m, m ≤ Length[v], m++, AppendTo[u, Sv[v[[m]]]]]; u = Select[u, ! SameQ[#, Null] &]; u = If[Length[u] == 1, u[[1]], u];
      If[{z} ≠ {}, ToExpression[u]];
      u, $Failed]]
End[]

Begin["`BFMSubsQ`"]
BFMSubsQ[x_, y___] :=
  Module[{a, b, c, d = {}, k = 1, p, h, g = {}}, If[! BlockModQ[x], False, {a, b} = Map[Flatten, {{PureDefinition[x]}, {HeadPF[x]}}];
    For[k, k ≤ Length[a], k++, p = a[[k]]; p = StringReplace[p, b[[k]] <> " := " –> "", 1];
      c = Select[ExtrVarsOfStr[p, 1], ! SystemQ[#] &]; h = Flatten[Map[StrSymbParity[p, " " <> #, "[", "]"] &, c]];
      h = Select[h, SuffPref[#, Map[StringJoin[" " <> # <> "["] &, c], 1] && ! StringFreeQ[#, "_"] &];
      AppendTo[g, {b[[k]], Length[h]}];
      AppendTo[d, {If[h ≠ {}, True, False], b[[k]]}]]; If[{y} ≠ {} && ! HowAct[y], y = g];
    If[Length[d] == 1, d[[1]], d]]]
End[]

Begin["`CallQ`"]
```

```
CallQ[x_] := Module[{a = ToString[If[Quiet[Part[x, 1]] === -1, Part[x, 1] * x, x]], b, c}, b = Flatten[StringPosition[a, "["]];
    If[b == {}, False, c = b[[1]];
      If[SymbolQ[StringTake[a, {1, c - 1}]] && StringTake[a, {c + 1, c + 1}] ≠ "[" && StringTake[a, -1] == "]", True, False]]]
End[]

Begin["`StructProcFunc`"]
StructProcFunc[x_ /; BlockFuncModQ[x]] :=
  Module[{c, d, h = {}, p, k = 1, t, b = Flatten[{HeadPF[x]}], a = Flatten[{PureDefinition[x]}]},
    c = Map9[StringReplace, a, Map[StringJoin[#, " := "] → "" &, b]];
    While[k ≤ Length[b], d = c[[k]]; If[SuffPref[d, "Module[{", 1], t = "Module", If[SuffPref[d, "Block[{", 1], t = "Block", t = ""]];
      If[t ≠ "", AppendTo[h, {t, b[[k]], p = SubStrSymbolParity1[d, "{", "}"][[1]];
            StrToList[p], StringReplace[StringTake[d, {1, -2}], t <> "[" <> p <> ", " → ""]}],
        AppendTo[h, {"Function", b[[k]], StringReplace[d, b[[k]] <> " := " → ""]}]]; k++]; If[Length[h] == 1, h[[1]], h]]
End[]

Begin["`AssignL`"]
AssignL[x_, y_, z___] := Quiet[If[{z} ≠ {}, x := y, x = y]]
End[]

Begin["`SubsProcQ`"]
SubsProcQ[x_, y_, z___] := Module[{a, b, k = 1, j = 1, Res = {}},
    If[BlockModQ[x] && BlockFuncModQ[y], {a, b} = Map[Flatten, {{Definition4[ToString[x]]}, {Definition4[ToString[y]]}}];
      For[k, k ≤ Length[b], k++, For[j, j ≤ Length[a], j++,
          If[! StringFreeQ[a[[j]], b[[k]]], AppendTo[Res, {StringTake[a[[j]], {1, Flatten[StringPosition[a[[j]], " := "]][[1]] - 1}],
              StringTake[b[[k]], {1, Flatten[StringPosition[b[[k]], " := "]][[1]] - 1}]}], Continue[]]];
      If[Res ≠ {}, If[{z} ≠ {} && ! HowAct[z], z = If[Length[Res] == 1, Res[[1]], Res]; True], False], False]]
End[]

Begin["`ReplaceProcBody`"]
ReplaceProcBody[x_ /; BlockModQ[x], y_ /; StringQ[y]] := ToExpression[StringReplace[PureDefinition[x], ProcBody[x] → y]]
End[]

Begin["`ReplaceProc`"]
ReplaceProc[x_ /; ProcQ[x], r_ /; DeleteDuplicates[Map[RuleQ, Flatten[{r}]]] == {True}] :=
  Module[{a = Definition2[x], b = HeadPF[x], c, d = Flatten[{r}]},
    c = ToExpression["Hold[" <> StringTrim[a[[1]], b <> " := "] <> "]"]; d = Select[d, ! MemberQ[Args1[x], ToString[Part[#, 1]]] &];
    c = ToString1[ReplaceAll[c, d]]; b <> " := " <> StringTake[c, {6, -2}]]
End[]

Begin["`CallSave`"]
CallSave[x_ /; FileExistsQ[x], y_ /; SymbolQ[y] || ListQ[y], z_ /; ListQ[z]] :=
  Module[{a = StringReplace[StringTake[ToString[InputForm[ReadString[x]]], {2, -2}], "\\r\\n\\r\\n" → "\\r\\n \\r\\n"],
      b, c, d, nf, u, p, t, v, s = Map[ToString, Flatten[{y}]]}, b = StringSplit[a, "\\r\\n \\r\\n"];
    nf[g_] := StringTake[g, {1, Flatten[StringPosition[g, "[", 1]][[1]] - 1}];
    c = Select[b, SuffPref[#, p = Flatten[{Map4[StringJoin, s, "["], Map4[StringJoin, s, " /: "]}], 1] &];
    {d, u, t, v} = {{}, {}, Map[# <> " /: " &, s], {}};
    Map[If[SuffPref[#, t, 1], AppendTo[u, DelSuffPref[StringReplace[StringTrim[#, t], "\\" -> ""], "rn", 2]], AppendTo[d, #]] &, c];
    Map[ToExpression, {d, u}]; If[d == {}, $Failed, If[Length[d] == 1, Symbol[nf[d[[1]]]][Sequences[z]],
        If[Length[DeleteDuplicates[Map[nf[#] &, d]]] == 1, Map[Symbol[nf[#]][Sequences[z]] &, d][[1]], Map[nf[#] &, d]]]]]
End[]

Begin["`DefOnHead`"]
DefOnHead[x_ /; HeadingQ[x]] :=
  Module[{a, b, c, d, h = Quiet[Check[RedSymbStr[StringReplace[StandHead[x], "," → ", "], " ", " "], x]]}, a = HeadName[h];
    b = Definition2[ToExpression[a]];
    c = Select[b, SuffPref[#, Map3[StringJoin, h, {" := ", " = "}], 1] &];
    d = Select[b, SuffPref[#, Quiet[Map3[StringJoin, "Options[" <> a <> "]", {" = ", " := "}]], 1] &];
    If[MemberQ[b, "Undefined"], $Failed, If[d == {}, AppendTo[c, b[[-1]]], Join[c, {d[[1]], b[[-1]]}]]]]
End[]

Begin["`StandHead`"]
StandHead[x_ /; HeadingQ[x] || HeadingQ1[x]] := Module[{a = HeadName[x], b}, b = StringReplace[x, a <> "[" → "", 1];
    b = ToString1[ToExpression["{" <> StringTake[b, {1, -2}] <> "}"]]; a <> "[" <> StringTake[b, {2, -2}] <> "]"]
End[]

Begin["`ExtrProcFunc`"]
ExtrProcFunc[x_ /; HeadingQ[x]] := Module[{a = StandHead[x], c, d, b = HeadName[x], g, p}, c = Definition2[ToExpression[b]];
    If[c[[1]] == "Undefined", $Failed, d = Select[c, SuffPref[#, a <> " := ", 1] &]; c = ToString[Unique[b]];
      If[d ≠ {}, ToExpression[c <> d[[1]]]; g = AttrOpts[b]; p = c <> b; Options[p] = g[[1]]; SetOptions[p, g[[1]]];
```

```
        ToExpression["SetAttributes[" <> p <> "," <> ToString[g[[2]]] <> "]"]; Clear[c]; p, Clear[c]; $Failed]]]
End[]

Begin["`SeqToList1`"]
SeqToList1[x___] := {x}
End[]

Begin["`SeqDel`"]
SeqDel[x_ /; SeqQ[x], y_] :=
  Module[{a = SeqToList[x], b = If[SeqQ[y], SeqToList[y], y]}, ListToSeq[Select[a, ! MemberQ[Flatten[{b}], #] &]]]
End[]

Begin["`Default1`"]
Default1[x_ /; SymbolQ[x], y_ /; PosIntListQ[y], z_ /; ListQ[z]] :=
  Module[{a = Min[Map[Length, {y, z}]], k = 1}, While[k ≤ a, Default[x, y[[k]]] = z[[k]];
      k++];]
End[]

Begin["`Defaults`"]
Defaults[x_ /; BlockFuncModQ[x],
    y_ /; ListQ[y] && Length[y] == 2 || ListListQ[y] && DeleteDuplicates[Map[IntegerQ[#[[1]]] &, y]] == {True}] :=
  Module[{a = Flatten[{Definition2[x]}], atr = Attributes[x], b = Flatten[{HeadPF[x]}][[1]],
    c = Args[x], d, g = If[ListListQ[y], y, {y}], p, h = {}, k = 1, q, t, u},
   c = Select[Map[ToString, If[NestListQ[c], c[[1]], c]], # ≠ "$Failed" &];
   If[Max[Map[#[[1]] &, g]] ≤ Length[c] && Min[Map[#[[1]] &, g]] ≥ 1, q = Map[#[[1]] &, g];
    d = StringReplace[a[[1]], b → "", 1];
    While[k ≤ Length[q], p = c[[q[[k]]]]; t = StringSplit[p, "_"];
     If[MemberQ[q, q[[k]]],  u = If[Length[t] == 2, t[[2]] = StringReplace[t[[2]], " /; " → ""];
          If[Quiet[ToExpression["{" <> t[[1]] <> "=" <> ToString[g[[k]][[2]]] <> "," <> t[[2]] <> "}"]][[2]] ||
            Quiet[Head[g[[k]][[2]]] === Symbol[t[[2]]]], True, False
          ], True];
       If[u, c[[q[[k]]]] = StringTake[p, {1, Flatten[StringPosition[p, "_"]][[2]]}] <> "."]; k++];
    ClearAllAttributes[x];
    Clear[x];
    k = 1;
    While[k ≤ Length[q],
     ToExpression["Default[" <> ToString[x] <> ", " <> ToString[q[[k]]] <> "]" <> " = " <> ToString1[g[[k]][[2]]]];
     k++];
    ToExpression[ToString[x] <> "[" <> StringTake[ToString[c], {2, -2}] <> "]" <> d];
    Map[ToExpression, MinusList[a, {a[[1]]}]];
    SetAttributes[x, atr], $Failed]]
End[]

Begin["`DefaultsM`"]
DefaultsM[x_ /; BlockFuncModQ[x],
    y_ /; ListQ[y] && Length[y] == 2 || ListListQ[y] && DeleteDuplicates[Map[IntegerQ[#[[1]]] &, y]] == {True}] :=
  Module[{a = Flatten[{PureDefinition[x]}], ArtKr, atr = Attributes[x], g = If[ListListQ[y], y, {y}], q, k = 1},
    ClearAllAttributes[x]; ClearAll[x]; q = Map[#[[1]] &, g];
    While[k ≤ Length[g],
     ToExpression["Default[" <> ToString[x] <> ", " <> ToString[g[[k]][[1]]] <> "]" <> " = " <> ToString1[g[[k]][[2]]]];
     k++];
    ArtKr[s_ /; StringQ[s], def_ /; ListQ[def]] := Module[{n = Unique[AVZ], b, c, d, t, j = 1, h}, h = ToString[n] <> ToString[x];
       ToExpression[ToString[n] <> s];
       b = HeadPF[h];  d = StringReplace[PureDefinition[h], b → ""];
       c = Select[Map[ToString, Args[h]], # ≠ "$Failed" &];
       While[j ≤ Length[c], If[MemberQ[q, j], t = c[[j]];  c[[j]] = StringTake[t, {1, Flatten[StringPosition[t, "_"]][[2]]}] <> "."];
        j++];
       ToExpression[ToString[x] <> "[" <> StringTake[ToString[c], {2, -2}] <> "]" <> d];
       ClearAll[h, n]];
    k = 1;
    While[k ≤ Length[a], ArtKr[a[[k]], g]; k++]; SetAttributes[x, atr]]
End[]

Begin["`SelfReprod`"]
SelfReprod[c_ /; StringQ[c], n_ /; IntegerQ[n], p_ /; IntegerQ[p], m_ /; IntegerQ[m]] :=
  Module[{a, h = StringJoin[Map[ToString, NestList[Sin, 0, n - 1]]], b = "0", k}, a = h <> c <> h;  d = 1;
    Label[AVZ];  For[k = StringLength[a] - n + 1, k ≥ 1, k--,
      b = ToString[Mod[Total[ToExpression[Characters[StringTake[a, {k, k + n - 1}]]]], p]] <> b];  a = "0" <> b;  t = StringCount[a, c];
    If[t ≥ m, d, d++;  Goto[AVZ]]]
```

```
End[]

Begin["`SelfReprod1`"]
SelfReprod1[Ltf_ /; ListQ[Ltf], Cf_ /; StringQ[Cf], p_ /; IntegerQ[p], h_ /; StringQ[h]] :=
  Module[{n = StringLength[Part[Ltf[[1]], 1]], a, b, c = "", k, d = 0, t},
    a = StringJoin[Map[ToString, NestList[Sin, 0, n − 2]]]; b = a <> h <> a;
    Label[AVZ]; For[k = StringLength[b] − n + 1, k ≥ 1, k −−, c = StringReplace[StringTake[b, {k, k + n − 1}], Ltf] <> c];
    b = a <> c <> a;
    c = "";
    d = d + 1; If[t = StringCount[b, Cf];
      t ≥ p, {d, t}, Goto[AVZ]]]
End[]

Begin["`DefaultValues1`"]
DefaultValues1[x_ /; BlockFuncModQ[x]] :=
  Module[{d = {}, h, k, a = {SetAttributes[String, Listable]}, b = Map[ToString, Args[x]], c = Map[ToString, DefaultValues[x]]},
    ClearAttributes[ToString, Listable];
    If[b ≠ {}, For[a = 1, a ≤ Length[b], a ++, h = b[[a]];
        If[! StringFreeQ[h, "_:"], AppendTo[d,
            ToExpression["{" <> ToString[a] <> "} :> " <> StringTake[h, {Flatten[StringPosition[h, "_:"]][[2]] + 1, −1}]]]]]];
    If[c ≠ {}, If[c ≠ {}, c = ToExpression[Mapp[StringReplace, Mapp[StringReplace, c,
                {"HoldPattern[Default[" <> ToString[x] −> "{", "]]" −> "}"}], {"{, " −> "{", "{}" −> "{2016}"}]]];
      h = c[[1]][[1]]; If[Op[h] == {2016},
        a =
          {};
        For[k = 1, k ≤ Length[b], k ++, AppendTo[a, ToExpression[ToString[{k}] <> " :> " <> ToString[c[[1]][[2]]]]]];
        c = a];
      If[PosIntListQ[h] && Length[h] > 1, a = {}; b = h;
        For[k = 1, k ≤ Length[b], k ++, AppendTo[a, ToExpression[ToString[{k}] <> " :> " <> ToString[c[[1]][[2]]]]]];
        c = a]];
    If[d == {} && c == {}, Return[{}], c = Sort[Join[d, c], Op[#1][[1]][[1]] ≤ Op[#2][[1]][[1]] &]];
    {k, h} = {1, {}}; While[k ≤ Length[c] − 1, AppendTo[h, If[Op[c[[k]]][[1]] == Op[c[[k + 1]]][[1]], k + 1]];
      k ++];
    Select[ReplacePart[c, Mapp[Rule, Select[h, # ≠ "Null" &], Null]], ! SameQ[#, Null] &]]
End[]

Begin["`DefaultValues2`"]
DefaultValues2[x_ /; BlockFuncModQ[x], y_ /; IntegerListQ[y], z_ /; ListQ[z]] :=
  Module[{a = Attributes[x], b = Map[ToString, Args[x]],
      c = HeadPF[x], d = PureDefinition[x], p, h = {}, y1, z1, g = {}}, ClearAttributes[x, a];
    p = Min[Map[Length, {y, z}]]; y1 = y[[1 ;; p]]; z1 = z[[1 ;; p]]; If[y1 == {}, Null,
      Do[AppendTo[h, If[! MemberQ[y1, k], b[[k]], If[SuffPref[b[[k]], "_", 2], AppendTo[g, {k} −> z1[[k]]];
              b[[k]] <> ":" <> ToString1[z1[[k]]], If[SuffPref[b[[k]], "_.", 2], AppendTo[g, {k} −> z1[[k]]];
                StringTake[b[[k]], {1, −2}] <> ":" <> ToString1[z1[[k]]], b[[k]]]]]], {k, 1, Length[b]}]; Clear[x];
      ToExpression[StringReplace[d, c → ToString[x] <> "[" <> StringTake[ToString[h], {2, −2}] <> "]", 1]]; SetAttributes[x, a]; g]]
End[]

Begin["`Sequences`"]
Sequences[x__] := Module[{a = Flatten[{x}], b}, b = "Sequence[" <> ToString1[a] <> "]"; a = Flatten[StringPosition[b, {"{", "}"}]];
    ToExpression[StringReplace[b, {StringTake[b, {a[[1]], a[[1]]}] → "", StringTake[b, {a[[−1]], a[[−1]]}] → ""}]]]
End[]

Begin["`Sq`"]
Sq[x_ /; ListQ[x]] := ToExpression["Sequence[" <> StringTake[ToString1[x], {2, −2}] <> "]"]
End[]

Begin["`MemberQ1`"]
MemberQ1[L_ /; ListQ[L], x_, y_ /; ! HowAct[y]] :=
    Module[{a = Flatten[L], b = L, c = 0, p = {}}, While[! b == {}, If[MemberQ[b, x], p = Append[p, c], Null];
        b = Select[b, ListQ[#] &]; b = Flatten[b, 1]; c = c + 1]; If[p == {}, False, y = p; True]];
End[]

Begin["`Iff`"]
Iff[x_, y__ /; StringQ[y]] := Module[{a = {x, y}, b}, b = Length[a];
    If[b == 1 || b >= 5, Defer[Iff[x, y]],
      If[b === 2, If[x, ToExpression[y]],
        If[b == 3, If[x, ToExpression[y], ToExpression[a[[3]]]],
          If[b == 4, If[x, ToExpression[a[[2]]], ToExpression[a[[3]]], ToExpression[a[[4]]], Null]]]]]
End[]
```

```
Begin["`GotoLabel`"]
GotoLabel[x_ /; BlockModQ[x]] :=
  Module[{b, c = {{}, {}, {}}, d, p, a = Flatten[{PureDefinition[x]}][[1]], k = 1, j, h, v = {}, t}, b = ExtrVarsOfStr[a, 1];
    b = DeleteDuplicates[Select[b, MemberQ[{"Label", "Goto"}, #] &]];
    If[b == {}, c, d = StringPosition[a, Map[" " <> # <> "[" &, {"Label", "Goto"}]];
      t = StringLength[a];
      For[k, k ≤ Length[d], k++, p = d[[k]];
        h = "";
        j = p[[2]]; While[j ≤ t, h = h <> StringTake[a, {j, j}];
          If[StringCount[h, "["] == StringCount[h, "]"], AppendTo[v, StringTake[a, {p[[1]] + 1, p[[2]] − 1}] <> h]; Break[]];
          j++]];
      h = DeleteDuplicates[
          v];
      {Select[h, SuffPref[#, "Goto", 1] &], Select[h, SuffPref[#, "Label", 1] &],
        Gather[Sort[v], #1 == StringReplace[#2, "Label[" −> "Goto[", 1] &]}]]
End[]

Begin["`MemberT`"]
MemberT[L_ /; ListQ[L], x_] := Length[Select[Flatten[L], SameQ[#, x] &]]
End[]

Begin["`MemberQ2`"]
MemberQ2[L_ /; ListQ[L], x_, y_ /; ! HowAct[y]] :=
  Module[{b = Flatten[L], c = 0, k = 1}, If[MemberQ[b, x], For[k, k ≤ Length[b], k++, If[b[[k]] === x, c = c + 1, Next[]]];
      y = c;
      True, False]]
End[]

Begin["`MemberQ3`"]
MemberQ3[x_ /; ListQ[x], y_ /; ListQ[y], z___] := If[{z} ≠ {}, MemberQ[SubsStrLim[ToString[x], "{", "}"], ToString[y]],
    SameQ[DeleteDuplicates[Map3[MemberQ, Flatten[x], Flatten[y]]], {True}]]
End[]

Begin["`MemberQ4`"]
MemberQ4[x_ /; ListQ[x], y_, z___] :=
  If[ListQ[y], Count[(MemberQ[x, #1] &) /@ y, True] >= If[{z} === {}, 1, If[IntegerQ[z], z, 1]], MemberQ[x, y]]
End[]

Begin["`IFk`"]
IFk[x__] := Module[{a = {x}, b, c = "", d = "If[", e = "]", h = {}, k = 1}, b = Length[a];
    If[For[k, k ≤ b − 1, k++, AppendTo[h, b ≥ 2 && ListQ[a[[k]]] && Length[a[[k]]] == 2]];
      DeleteDuplicates[h] != {True}, Return[Defer[IFk[x]]], k = 1];
    For[k, k ≤ b − 1, k++, c = c <> d <> ToString[a[[k]][[1]]] <> "," <> ToString[a[[k]][[2]]] <> ","];
    c = c <> ToString[a[[b]]] <> StringMultiple[e, b − 1];
    ToExpression[c]]
End[]

Begin["`IFk1`"]
IFk1[x__] := Module[{a = {x}, b, c = "", d = "If[", e = "]", h = {}, k = 1}, b = Length[a];
    If[For[k, k ≤ b − 1, k++, AppendTo[h, b ≥ 2 && ListQ[a[[k]]] && Length[a[[k]]] == 2]];
      DeleteDuplicates[h] != {True}, Return[Defer[IFk1[x]]], {h, k} = {{}, 1}];
    If[For[k, k ≤ b − 1, k++, AppendTo[h, a[[k]][[1]]]];
      Select[h, ! MemberQ[{True, False}, #] &] ≠ {}, Return[Defer[IFk1[x]]], k = 1];
    For[k = 1, k ≤ b − 1, k++, c = c <> d <> ToString[a[[k]][[1]]] <> "," <> ToString[a[[k]][[2]]] <> ","];
    c = c <> ToString[a[[b]]] <> StringMultiple[e, b − 1];
    ToExpression[c]]
End[]

Begin["`ListToString`"]
ListToString[x_ /; ListQ[x] || StringQ[x], y_ /; StringQ[y]] := Module[{a, b = {}, c, d, k = 1}, If[ListQ[x], a = Flatten[x];
      For[k, k < Length[a], k++, c = a[[k]];
        AppendTo[b, ToString1[c] <> y]];
      a = StringJoin[Append[b, ToString1[a[[−1]]]]], a = FromCharacterCode[14];
      d = a <> StringReplace[x, y → a] <> a;
      c = Sort[DeleteDuplicates[Flatten[StringPosition[d, a]]]];
      For[k = 1, k < Length[c], k++, AppendTo[b, StringTake[d, {c[[k]] + 1, c[[k + 1]] − 1}]]];
      ToExpression[b]]]
End[]
```

```
Begin["`GroupIdentMult`"]
GroupIdentMult[x_ /; ListQ[x]] := Module[{a = Gather[x], b, c}, b = Map[{DeleteDuplicates[#][[1]], Length[#]} &, a];
    b = Map[DeleteDuplicates[#] &, Map[Flatten, Gather[b, SameQ[#1[[2]], #2[[2]]] &]]];
    b = Map[{{#[[1]]}, Sort[#[[2 ;; -1]]]} &, Map[Reverse, Map[If[Length[#] > 2, Delete[Append[#, #[[2]]], 2], #] &, b]]];
    b = Sort[b, #1[[1]][[1]] > #2[[1]][[1]] &]; If[Length[b] == 1, Flatten[b, 1], b]]
End[]

Begin["`MemberLN`"]
MemberLN[L_ /; NestQL[L], x_] := Module[{a = L, b = {}, c = 0, d, k, p = 0, h = {}}, While[a ≠ {}, c = c + 1;
      For[k = 1, k ≤ Length[a], k++, d = a[[k]];
        If[d === x, p = p + 1, If[ListQ[d], AppendTo[b, d], Null]]]; h = Append[h, {c, p}];
      a = Flatten[b, 1];
      b = {};
      p = 0]; h]
End[]

Begin["`SEQ`"]
SEQ[x_, y_ /; SymbolQ[y], z_ /; Head[z] == Span] :=
  Module[{a = ToString[z], b = {}, c, d = ToString[y], p}, c = ToExpression[StringSplit[a, " ;; "]];
    If[DeleteDuplicates[Map[NumberQ, c]] ≠ {True} || DeleteDuplicates[Map[Positive, c]] ≠ {True},
      Return[Defer[SEQ[x, y, z]]], If[Length[c] > 2 && c[[3]] == 0, Return[Defer[SEQ[x, y, z]]], If[c[[1]] ≤ c[[2]], p = 1, p = 2]]];
    For[y = c[[1]], If[p == 1, y ≤ c[[2]], y >= c[[2]] - If[p == 1 && Length[c] == 2 || p == 2 && Length[c] == 2, 0, c[[3]] - 1]],
      If[Length[c] == 2, If[p == 1, y++, y--], If[p == 1, y += c[[3]], y -= c[[3]]]], b = Append[b, x]];
    {ToExpression["Clear[" <> d <> "]"], b}[[2]]
End[]

Begin["`PrefCond`"]
PrefCond[x_ /; StringQ[x], y_ /; StringQ[y]] :=
  Module[{a = Flatten[StringPosition[x, y]]}, If[a == {}, "", StringTake[x, {1, a[[1]] - 1}]]]
End[]

Begin["`AttrOpts`"]
AttrOpts[x_ /; BlockFuncModQ[x]] :=
  Module[{a = Definition2[x], b, c, d}, b = a[[-1]]; c = Select[a, SuffPref[#, "Options[" <> ToString[x] <> "]", 1] &];
    If[c == {}, d = c, d = StringSplit[c[[1]], " := "][[2]]]; {ToExpression[d, b}]
End[]

Begin["`MxToTxt`"]
MxToTxt[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; StringQ[y], z___] :=
  Module[{a = ContextMXfile[x], b, c}, LoadMyPackage[x, a]; b = CNames[a];
    Map[{Write[y, Definition[#]], Write[y]} &, b]; Close[y];
    If[MemberQ[{z}, "Del"], RemovePackage[a]]; c = Select[{z}, ! HowAct[#] && ! SameQ[#, "Del"] &];
    If[c ≠ {}, ToExpression[ToString[c[[1]]] <> "=" <> ToString[b]]];]
End[]

Begin["`MxToTxt1`"]
MxToTxt1[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; StringQ[y], z___] :=
  Module[{a = ContextFromFile[x], c}, LoadMyPackage[x, a];
    Map[PutAppend[Definition[#], "OK!", y] &, CNames[a]];
    If[MemberQ[{z}, "Del"], RemovePackage[a]]; c = Select[{z}, ! HowAct[#] && ! SameQ[#, "Del"] &];
    If[c ≠ {}, ToExpression[ToString[c[[1]]] <> "=" <> ToString[b]]];]
End[]

Begin["`MxToTxt2`"]
MxToTxt2[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; StringQ[y]] :=
  Module[{a = ContextFromFile[x], c}, If[! MemberQ[$Packages, a], c = 78];
    Get[x];
    Map[PutAppend[Definition[#], "OK!", y] &, CNames[a]]; If[c === 78, RemovePackage[a]]; "OK!"]
End[]

Begin["`MxToMpackage`"]
MxToMpackage[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y___] :=
  Module[{a = ContextInMxFile[x], b, c = {}, d, f, u, s, h}, If[a === $Failed, $Failed, If[MemberQ[$Packages, a], h = 74, Quiet[Get[x]]];
      If[{y} == {}, f = FileBaseName[x] <> ".m", f = FileBaseName[If[StringQ[y], y, x]] <> ".m"];
      b = CNames[a]; Map[If[SameQ[Set[u, ToString[ToExpression[# <> "::usage"]]], # <> "::usage"], Null,
            AppendTo[c, # <> "::usage = " <> u]] &, b]; If[c == {}, $Failed, c = Flatten[Map[{"(*" <> # <> "*)", "(**)"} &, c]];
        d = Flatten[Map[{"(*Begin[\"`" <> # <> "`\"]*)", Map["(*" <> #1 <> ";*)" &, Flatten[{PureDefinition[#]}]],
              "(*" <> "SetAttributes[" <> # <> ", Attributes[" <> # <> "]]" <> "*)", "(*End[]*)", "(**)"} &, b]]; s = OpenWrite[f];
```

```
        Map[WriteLine[s, #] &, Join[{"(* ::Package:: *)", "(**)", "(* ::Input:: *)", "(**)", "(*BeginPackage[\""  <> a <> "\"]*)", "(**)"},
            c, d, {"(*EndPackage[]*)"}]]; If[! SameQ[h, 74], RemovePackage[a], Null]; Close[s]]]
End[]

Begin["`MxFileToMfile`"]
MxFileToMfile[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; StringQ[y] && FileExtension[y] == "m"] :=
  Module[{a = ContextFromFile[x], b, c, k = 1}, Get[x];
    b = CNames[a];
    WriteString[y, "(* ::Package:: *)", "\n", "(* ::Input:: *)", "\n", "(*BeginPackage[\"" <> a <> "\"]*)", "\n"];
    While[k ≤ Length[b], c = b[[k]] <> "::usage";
      WriteString[y, "(*" <> c <> " = " <> ToString1[ToExpression[a <> c]], "*)", "\n"];
      k++]; k = 1;
    While[k ≤ Length[b], c = b[[k]];
      WriteString[y, "(*Begin[\"`" <> c <> "`\"]*)", "\n", "(*" <> PureDefinition[a <> c] <> "*)", "\n", "(*End[]*)", "\n"];
      k++]; WriteString[y, "(*EndPackage[]*)", "\n"];
    Map[{Clear1[2, a <> # <> "::usage"], Clear1[2, a <> #]} &, b]; $ContextPath = MinusList[$ContextPath, {a}]; Close[y]]
End[]

Begin["`MfileToMx`"]
MfileToMx[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] := Module[{a = ContextFromFile[x], b, d, c = ToString1[x <> "x"]},
    If[MemberQ[$ContextPath, a], ToExpression["DumpSave[" <> c <> "," <> ToString1[a] <> "]"];
      x <> "x", b = ReadList[x, String]; d = Select[Map[StringReplace[#, {"(*" → "", "*)" → ""}] &, b[[3 ;; -1]]], # ≠ "" &];
      Quiet[ToExpression[d]];
      ToExpression["DumpSave[" <> c <> "," <> ToString1[a] <> "]"];
      Map[Clear1[2, a <> #] &, CNames[a]]; $ContextPath = MinusList[$ContextPath, {a}];
      x <> "x"]]
End[]

Begin["`Globals`"]
Globals[P_ /; ProcBMQ[P]] := Module[{c, d = {}, p, g = {}, k = 1, b = ToString1[DefFunc[P]], a = Sort[Locals1[P]]},
    If[P === ExprOfStr, Return[{}], c = StringPosition[b, {" := ", " = "}][[2 ;; -1]]];
    For[k, k ≤ Length[c], k++, p = c[[k]]; AppendTo[d, ExprOfStr[b, p[[1]], -1, {" ", ",", "\"", "!", "{"}]]];
    For[k = 1, k ≤ Length[d], k++, p = d[[k]];
      If[p ≠ "$Failed" && p ≠ " ", AppendTo[g,
          If[StringFreeQ[p, {"{", "}"}], p, StringSplit[StringReplace[p, {"{" → "", "}" → ""}]], ","]], Null]];
    g = Flatten[g]; d = {}; For[k = 1, k ≤ Length[g], k++, p = g[[k]];
      AppendTo[d, If[StringFreeQ[p, {"[", "]"}], p, StringTake[p, {1, Flatten[StringPosition[p, "["]][[1]] - 1}]]]];
    g = d;
    d = {}; For[k = 1, k ≤ Length[g], k++, p = g[[k]]; AppendTo[d, StringReplace[p, {"," → "", " " → ""}]]];
    d = Sort[Map[StringTrim, DeleteDuplicates[Flatten[d]]]]; Select[d, ! MemberQ[If[ListListQ[a], a[[1]], a], #] &]]
End[]

Begin["`Globals1`"]
Globals1[P_ /; ProcQ[P]] :=
  Module[{a = SubProcs[P], b, c, d = {}}, {b, c} = Map[Flatten, {Map[Locals1, a[[2]]], Map[Globals, a[[2]]]}];
    MinusList[DeleteDuplicates[c], b]]
End[]

Begin["`Globals2`"]
Globals2[x_ /; ProcQ[x] || ModuleQ[x] || BlockQ[x]] := ExtrNames[x][[3]]
End[]

Begin["`VarsInBlockMod`"]
VarsInBlockMod[x_ /; BlockQ[x] || ModuleQ[x]] :=
  Module[{a = Locals1[x], b = Globals[x], c, d}, c = ExtrVarsOfStr[StringReplace[PureDefinition[x], HeadPF[x] <> " := " -> "", 1], 1];
    c = Select[c, ! Quiet[Check[SystemQ[#], True]] &]; {d = Args[x, 90], a, b, MinusList[c, Flatten[{a, b, d}]]}]
End[]

Begin["`GlobalToLocal`"]
GlobalToLocal[x_ /; QBlockMod[x], y___] :=
  Module[{a = LocalsGlobals1[x], b, c}, If[Intersection[a[[1]], a[[3]]] == a[[3]] || a[[3]] == {}, x, b = Join[a[[2]], MinusList[a[[3]], a[[1]]]];
      c = "$$$" <> StringReplace[PureDefinition[x], ToString[a[[2]]] → ToString[b], 1];
      If[{y} ≠ {} && ! HowAct[y], y = {a[[1]], a[[3]]}]; ToExpression[c]; Symbol["$$$" <> ToString[x]]]]
End[]

Begin["`GlobalToLocalM`"]
GlobalToLocalM[x_ /; QBlockMod[x]] := Module[
    {d, h = "$$$", k = 1, n, p = {}, b = Attributes[x], c = Options[x], a = Flatten[{PureDefinition[x]}]}, While[k ≤ Length[a], d = a[[k]];
      n = h <> ToString[x]; ToExpression[h <> d]; GlobalToLocal[Symbol[n]]; AppendTo[p, PureDefinition["$$$" <> n]];
```

```
      ToExpression["ClearAll[" <> n <> "]"]; k++];  ClearAllAttributes[x]; ClearAll[x];
    ToExpression[Map[StringReplace[#, "$$$$$$" -> "", 1] &, p]];  SetAttributes[x, b];
    If[c ≠ {}, SetOptions[x, c]];]
End[]

Begin["`LocalsGlobals`"]
LocalsGlobals[x_ /; ProcQ[x]] := {Locals[x], Globals1[x]}
End[]

Begin["`LocalsGlobals1`"]
LocalsGlobals1[x_ /; QBlockMod[x]] :=
  Module[{c = "", d, j, h = {}, k = 1, p, G, L, a = Flatten[{PureDefinition[x]}][[1]], b = Flatten[{HeadPF[x]}][[1]]},
    b = StringReplace[a, {b <> " := Module[" → "", b <> " := Block[" -> ""}, 1];
    While[k ≤ StringLength[b], d = StringTake[b, {k, k}];
      c = c <> d;  If[StringCount[c, "{"] == StringCount[c, "}"], Break[]]; k++];
    b = StringReplace[b, c <> "," -> "", 1];  L = If[c == "{}", {}, StrToList[StringTake[c, {2, -2}]]];
     d = StringPosition[b, {" := ", " = "}]; d = (#1[[1]] - 1 &) /@ d;
    For[k = 1, k ≤ Length[d], k++, c = d[[k]]; p = "";  For[j = c, j ≥ 1, j--, p = StringTake[b, {j, j}] <> p;
        If[! Quiet[ToExpression[p]] === $Failed && StringTake[b, {j - 1, j - 1}] == " ", AppendTo[h, p];
          Break[]]];
    G = Flatten[(If[StringFreeQ[#1, "{"], #1, StrToList[StringTake[#1, {2, -2}]]] &) /@
          (StringTake[#1, {1, Quiet[Check[Flatten[StringPosition[#1, "["]][[1]], 0]] - 1}] &) /@ h];
    b = (If[StringFreeQ[#1, " = "], #1, StringTake[#1, {1, Flatten[StringPosition[#1, " = "]][[1]] - 1}]] &) /@ L;
    d = DeleteDuplicates[Flatten[(StringSplit[#1, ", "] &) /@ MinusList[G, b]]];
    d = Select[d, ! Quiet[SystemQ[#1]] &&
          ! MemberQ[Flatten[{"\\", "#", "\"", "", "+", "-", ToString /@ Range[0, 9]}], StringTake[#1, {1, 1}]] &];
    {Select[b, ! MemberQ[ToString /@ Range[0, 9], StringTake[#1, {1, 1}]] &], L, MinusList[d, b]}]
End[]

Begin["`LocalsGlobalsM`"]
LocalsGlobalsM[x_ /; QBlockMod[x]] :=
  Module[{b = "$$90$", c, d = {}, k = 1, a = Flatten[{PureDefinition[x]}]}, While[k ≤ Length[a], c = b <> ToString[x];
      ToExpression[b <> a[[k]]];
      AppendTo[d, LocalsGlobals1[c]];  ToExpression["Clear[" <> c <> "]"];
      k++];  If[Length[d] == 1, d[[1]], d]]
End[]

Begin["`ArgsTypes`"]
ArgsTypes[x_ /; CompileFuncQ[x] || BlockFuncModQ[x] || PureFuncQ[x]] :=
  Module[{a = Args[x], c = {}, d = {}, k = 1}, If[CompileFuncQ[x], a = Mapp[StringSplit, Map[ToString, a], "_"];
      If[Length[a] == 1, a[[1]], a], If[PureFuncQ[x], a = Map[{#, "Arbitrary"} &, a];
        If[Length[a] == 1, a[[1]], a], SetAttributes[ToString, Listable];
        a = Map[ToString, a];
        ClearAttributes[ToString, Listable]; a = If[NestListQ[a], a, {a}];
        For[k, k ≤ Length[a], k++, c = Append[c, Mapp[StringSplit, Mapp[StringSplit, a[[k]], "_ /; "], {"___", "__", "_"}]]];
        c;
        For[k = 1, k ≤ Length[c], k++, d = Append[d, Map[Flatten, c[[k]]]]]; c = {};
        For[k = 1, k ≤ Length[d], k++,
          c = Append[c, Map[If[Length[#] == 1, {#[[1]], "Arbitrary"}, {#[[1]], StringReplace[#[[2]], "\\" -> ""]}] &, d[[k]]]]];
        c = Map[If[Length[#] == 1, #[[1]], #] &, c];  If[Length[c] == 1, c[[1]], c]]]
End[]

Begin["`FormalArgs`"]
FormalArgs[x_] := Module[{a, b = Quiet[Part[x, 1]]}, If[CallQ[x], a = ToString[If[b === -1, Part[x, 1] * x, x]];
      ToExpression["{" <> StringTake[a, {Flatten[StringPosition[a, "["]][[1]] + 1, -2}] <> "}"], $Failed]]
End[]

Begin["`Try`"]
Try[x_ /; StringQ[x], y_] := Quiet[Check[ToExpression[x], {y, $MessageList}]]
End[]

Begin["`Riffle1`"]
Riffle1[x_ /; ListListQ[x]] := Module[{a = {}}, Do[AppendTo[a, Map[#[[k]] &, x]], {k, 1, Length[x[[1]]]}]; a]
End[]

Begin["`ActBFMuserQ`"]
ActBFMuserQ[x___ /; If[{x} == {}, True, If[Length[{x}] == 1 && ! HowAct[x], True, False]]] :=
  Module[{b = {}, c = 1, d, h, a = Select[Names["`*"], ! UnevaluatedQ[Definition2, #] &]},
    For[c, c ≤ Length[a], c++, h = Quiet[ProcFuncTypeQ[a[[c]]]];
```

```
      If[h[[1]], AppendTo[b, {a[[c]], h[[-1]]}], Null]]; If[b == {}, False, If[{x} ≠ {}, x = If[Length[b] == 1, b[[1]], b]; True]]
End[]

Begin["`CNames`"]
CNames[x_ /; ContextQ[x], y___] :=
  Module[{a = Names[StringJoin[x, "*"]], b}, b = Select[a, Quiet[ToString[Definition[ToString[#1]]]] ≠ "Null" &];
    If[{y} ≠ {} && PureDefinition[y] === $Failed,
      y = Map[StringTrim[#, x] &, Sort[DeleteDuplicates[Select[a, Quiet[Check[PureDefinition[#], $Failed]] === $Failed &]]]];
    Map[StringTrim[#, x] &, Select[b, Quiet[Attributes[#] ≠ {Temporary} && ToString[Definition[#]]] ≠ "Null" &]]]
End[]

Begin["`IsFileOpen`"]
IsFileOpen[F_ /; FileExistsQ[F], h___] := Module[{a = OpenFiles[F]}, If[a == {}, False, If[{h} ≠ {} && ! HowAct[h], h = a, Null];
      True]]
End[]

Begin["`IsPackageQ`"]
IsPackageQ[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y___] :=
  Module[{a = ReadFullFile[x], b = "CONT", c = "ENDCONT", d, g = $Packages},
    If[! StringContainsQ[a, "CONT" ~~ __ ~~ "ENDCONT"], $Failed,
      d = StringPosition[a, {b, c}];
      d = StringTake[a, {d[[1]][[2]] + 1, d[[2]][[1]] - 1}]; d = Select[Map[If[! StringFreeQ[d, #], #, Null] &, g], ! SameQ[#, Null] &];
      If[{y} ≠ {} && ! HowAct[y], y = If[d == {}, {}, d[[1]]], Null]; If[d ≠ {}, True, False]]]
End[]

Begin["`ReplaceOut`"]
ReplaceOut[x_ /; PosIntQ[x] || PosIntListQ[x], y___] :=
  Module[{a = Flatten[{x}], b = Flatten[{y}], k = 1}, If[b ≠ {}, If[Length[a] ≠ Length[b], Defer[ReplaceOut[x, y]], Unprotect[Out];
        For[k, k ≤ Length[a], k++, Out[a[[k]]] = b[[k]]]; Protect[Out];, ClearOut[x]]]
End[]

Begin["`ExprQ`"]
ExprQ[x_ /; StringQ[x]] := If[Quiet[ToExpression[x]] === $Failed, False, True]
End[]

Begin["`$AobjNobj`"]
$AobjNobj := "(* ::Input:: *)"
End[]

Begin["`Cost`"]
Cost[x_] := Module[{f = {Plus, Times, Power, Indexed, Function}, a = ToString[InputForm[x]], b = {{"+", "-"}, {"*", "/"}, "^"},
    c, d = {}, h, k = 1, j, t}, If[StringFreeQ[a, Flatten[{b, "["}]], 0, c = Map[StringCount[a, #] &, b];
    While[k ≤ 3, h = c[[k]];
      If[h ≠ 0, AppendTo[d, {f[[k]], h}]];
      k++]; If[Set[b, StringCount[a, "[["] > 0, AppendTo[d, {f[[4]], b}]];
    t = StringPosition[a, "["]; If[t ≠ {}, t = Map[#[[1]] &, t];
      t = Select[Map[If[StringTake[a, {# - 1, # - 1}] ≠ "[" && StringTake[a, {# + 1, # + 1}] ≠ "[", #] &, t], ! SameQ[#, Null] &]];
    If[t ≠ {}, AppendTo[d, {f[[5]], Length[t]}]]; b = StringPosition[a, "("]; {t, b, h} = {0, Map[#[[1]] &, b], StringLength[a]};
    For[k = 1, k ≤ Length[b], k++, c = ""; For[j = b[[k]], j ≤ h, j++, c = c <> StringTake[a, {j, j}];
        If[StringCount[c, "{"] === StringCount[c, "}"],
          If[ExpressionQ[c], Continue[], If[NumberQ[Interpreter["Number", Positive][c]], t = t + 1]];
          Break[]]];
    d = If[t ≠ 0 && d[[1]][[1]] === Plus, d[[1]][[2]] = d[[1]][[2]] - t;
        d, d]; Plus[Sequences[Map[#[[2]] * #[[1]] &, d]]]]
End[]

Begin["`ListStrToStr`"]
ListStrToStr[x_ /; ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True}, p___] :=
  Module[{a = ""}, If[{p} == {}, Do[a = a <> x[[k]] <> ", ", {k, Length[x]}];
      StringTake[a, {1, -3}], StringJoin[x]]]
End[]

Begin["`StringToList`"]
StringToList[x_ /; StringQ[x]] :=
  Module[{a, b, c = {}, d = {}, k, j, t = "", t1, n = 1}, b = DeleteDuplicates[Flatten[StringPosition[x, ","]]];
    PrependTo[b, 0]; AppendTo[b, StringLength[x] + 1];
    Do[AppendTo[c, StringTake[x, {b[[k]] + 1, b[[k + 1]] - 1}]], {k, 1, Length[b] - 1}]; a = Length[c];
    Do[For[j = n, j ≤ a, j++, t = t <> c[[j]] <> ","; t1 = StringTrim[t, {",", " "}];
        If[SyntaxQ[t1], AppendTo[d, t1]; t = ""; n = j + 1; Break[]]], {k, 1, a}]; d]
```

```
End[]

Begin["`StandStrForm`"]
StandStrForm[x_ /; StringQ[x]] := StringReplace[ToLowerCase[x], "/" -> "\\"]
End[]

Begin["`RedSymbStr`"]
RedSymbStr[x_ /; StringQ[x], y_ /; SymbolQ1[y], z_ /; StringQ[z]] :=
  Module[{a = StringPosition[x, y], b}, If[StringFreeQ[x, y], x, b = Map[#[[1]] &, a];
    b = Sort[DeleteDuplicates[Map[Length, Split[b, #2 - #1 == 1 &]]], Greater];
    StringReplace[x, GenRules[Map3[StringMultiple, y, b], z]]]
End[]

Begin["`SubCfEntries`"]
SubCfEntries[x_ /; StringQ[x], n_ /; IntegerQ[n] && n >= 1, y_ : False] :=
  Module[{a = Characters[x], b = x, c = {}, d = {}}, If[n > StringLength[x], $Failed, Do[c = Join[c, Partition[Characters[b], n, n]];
      c = Gather[Map[StringJoin, c], #1 == #2 &]; Map[AppendTo[d, If[Length[#] > 1, #[[1]], Nothing]] &, c];
      c = {}; b = StringTake[b, {2, -1}]; If[StringLength[b] < n, Break[], Null], Infinity];
    d = DeleteDuplicates[Flatten[d]]; d = Map[{StringCount[x, #, Overlaps -> y], #} &, d]; If[Length[d] > 1 || d == {}, d, d[[1]]]]]
End[]

Begin["`ReduceList`"]
ReduceList[L_ /; ListQ[L], x_, z_, t_ /; MemberQ[{1, 2}, t]] :=
  Module[{a = Map[Flatten, Map[Position[L, #] &, Flatten[{x}]]], b = {}, m = Flatten[{x}], n = Flatten[{z}], k = 1},
    n = If[Length[m] > Length[n], PadRight[n, Length[m], 1], n];
    For[k, k ≤ Length[a], k++, If[Length[a[[k]]] ≥ n[[k]], AppendTo[b, a[[k]]], Null]];
    For[k = 1, k ≤ Length[a], k++, a[[k]] = If[t == 1, a[[k]][[1 ;; Length[a[[k]]] - n[[k]]]], a[[k]][[-Length[a[[k]]] + n[[k]] ;; -1]]]];
    Select[ReplacePart[L, GenRules[Flatten[a], Null]], ! SameQ[#, Null] &]]
End[]

Begin["`ReduceArgs`"]
ReduceArgs[x_ /; BlockFuncModQ[x]] :=
  Module[{a, b, c = Flatten[{PureDefinition[x]}], h = Flatten[{HeadPF[x]}], d = {}, d1 = {}, d2 = {}, atr = Attributes[x]}, ClearAll[b];
    ClearAttributes[x, atr]; a = ArgsBFM[x, b]; ClearAll[x]; Do[AppendTo[d, Join[a[[k]], b[[k]]]], {k, 1, Length[c]}];
    d = Map[{#[[1 ;; Length[#]/2]], #[[Length[#]/2 + 1 ;; -1]]} &, d];
    Do[AppendTo[d1, FirstPositionsList[d[[k]][[1]]]], {k, 1, Length[d]}];
    Do[AppendTo[d2, Part[d[[k]][[2]], d1[[k]]]], {k, 1, Length[d]}];
    d2 = Map[ToString[x] <> "[" <> StringTake[#, {2, -2}] <> "]" &, Map[ToString, d2]];
    Do[ToExpression[StringReplace[c[[k]], h[[k]] → d2[[k]], 1]], {k, 1, Length[c]}]; SetAttributes[x, atr]]
End[]

Begin["`MultipleArgsQ`"]
MultipleArgsQ[x_ /; BlockFuncModQ[x]] := Or @@ Map[Length[#] > Length[DeleteDuplicates[#]] &, ArgsBFM[x]]
End[]

Begin["`$TypeProc`"]
$TypeProc := CheckAbort[If[$Art27$Kr20$ = Select[{Stack[Module], Stack[Block], Stack[DynamicModule]}, # ≠ {} &];
      If[$Art27$Kr20$ == {}, Clear[$Art27$Kr20$]; Abort[], $Art27$Kr20$ = ToString[$Art27$Kr20$[[1]][[1]]]];
      SuffPref[$Art27$Kr20$, "Block[{", 1], Clear[$Art27$Kr20$];
      "Block", If[SuffPref[$Art27$Kr20$, "Module[{", 1] && ! StringFreeQ[$Art27$Kr20$, "DynamicModule"], Clear[$Art27$Kr20$];
        "DynamicModule", Clear[$Art27$Kr20$]; "Module"]], $Failed]
End[]

Begin["`$CallProc`"]
$CallProc := StringTake[ToString1[Stack[_][[1]]], {10, -2}]
End[]

Begin["`ProtectedQ`"]
ProtectedQ[x_] := If[MemberQ[Attributes1[x], Protected], True, False]
End[]

Begin["`StringMultiple`"]
StringMultiple[s_ /; StringQ[s], p_ /; IntegerQ[p] && p ≥ 1, h___] :=
  Module[{a = {s}}, Map[AppendTo[a, {If[{h} ≠ {}, ToString[h], ""] <> s, #}[[1]]] &, Range[1, p - 1]];
    StringJoin[a]]
End[]

Begin["`StringMultiple1`"]
StringMultiple1[x_ /; StringQ[x], n_ /; IntegerQ[n]] := Module[{a = ""}, Map[{a = a <> x, #} &, Range[n]]; a]
```

```
End[]

Begin["`SingleDefQ`"]
SingleDefQ[x_] := If[Length[Flatten[{PureDefinition[x]}]] == 1 || SameQ["System", PureDefinition[x]], True, False]
End[]

Begin["`StringDependQ`"]
StringDependQ[x_ /; StringQ[x], y_ /; StringQ[y] || ListStrQ[y], z___] :=
  Module[{a = Map[StringFreeQ[x, #] &, Flatten[{y}]], b = {}, c = Length[y], k = 1},
    If[DeleteDuplicates[a] == {False}, True, If[{z} ≠ {} && ! HowAct[z], z = Select[Flatten[y], StringFreeQ[x, #] &]];
      False]]
End[]

Begin["`StringDependQ1`"]
StringDependQ1[x_ /; StringQ[x], y_ /; ListStringQ[y]] :=
  Module[{a = x, b, k = 1}, For[k, k ≤ Length[y], k++, b = Flatten[StringPosition[a, y[[k]]]];
      If[b ≠ {}, a = StringTake[a, {b[[2]] + 1, -1}], Return[False]]]; True]
End[]

Begin["`Border`"]
Border[x_ /; StringQ[x]] := Module[{a = Floor[StringLength[x]/2], b = {}, c, k = 1}, For[k, k ≤ a, k++, c = StringTake[x, k];
      If[c === StringTake[x, -k], b = Append[b, c], Continue[]]]; If[b == {}, "", b[[-1]]]]
End[]

Begin["`SubStrSymbolParity`"]
SubStrSymbolParity[x_ /; StringQ[x], y_ /; CharacterQ[y], z_ /; CharacterQ[z], d_ /; MemberQ[{0, 1}, d],
    t___ /; t == {} || PosIntQ[{t}[[1]]] := Module[{a, b = {}, c = {y, z}, k = 1, j, f, m = 1, n = 0, p, h},
    If[{t} == {}, f = x, f = StringTake[x, If[d == 0, {t, StringLength[x]}, {1, t}]]];
    If[Map10[StringFreeQ, f, c] ≠ {False, False} || y == z, Return[], a = StringPosition[f, If[d == 0, c[[1]], c[[2]]]]];
    For[k, k ≤ Length[a], k++, j = If[d == 0, a[[k]][[1]] + 1, a[[k]][[2]] - 1]; h = If[d == 0, y, z];
      While[m ≠ n, p = Quiet[Check[StringTake[f, {j, j}], Return[$Failed]]];
        If[p == y, If[d == 0, m++, n++];
          If[d == 0, h = h <> p, h = p <> h], If[p == z, If[d == 0, n++, m++];
            If[d == 0, h = h <> p, h = p <> h], If[d == 0, h = h <> p, h = p <> h]]];
        If[d == 0, j++, j--]];
      AppendTo[b, h]; m = 1; n = 0; h = ""]; b]
End[]

Begin["`SubStrSymbolParity1`"]
SubStrSymbolParity1[x_ /; StringQ[x], y_ /; CharacterQ[y], z_ /; CharacterQ[z]] := Module[
    {a = DeleteDuplicates[Flatten[StringPosition[x, y]]], b = DeleteDuplicates[Flatten[StringPosition[x, z]]], c = {}, d, k = 1, j, p},
    If[a == {} || b == {}, $Failed, For[k, k ≤ Length[a], k++, p = StringTake[x, {a[[k]], a[[k]]}];
        For[j = a[[k]] + 1, j ≤ StringLength[x], j++, p = p <> StringTake[x, {j, j}];
          If[StringCount[p, y] == StringCount[p, z], AppendTo[c, p]; Break[]]]]; c]]
End[]

Begin["`StrSymbParity`"]
StrSymbParity[S_ /; StringQ[S], S1_ /; StringQ[S1], x_ /; StringQ[x] && StringLength[x] == 1,
    y_ /; StringQ[y] && StringLength[y] == 1] := Module[{a = StringPosition[S, S1], b = {}, c = S1, d, k = 1, j}, If[x == y || a == {}, {},
      For[k, k ≤ Length[a], k++, For[j = a[[k]][[2]] + 1, j ≤ StringLength[S], j++, c = c <> StringTake[S, {j, j}];
          If[StringCount[c, x] ≠ 0 && StringCount[c, y] ≠ 0 && StringCount[c, x] === StringCount[c, y],
            AppendTo[b, c]; c = S1; Break[]]]]];
    b]
End[]

Begin["`CorrectSubString`"]
CorrectSubString[x_ /; StringQ[x], y_ /; IntegerQ[y], z___] :=
  Module[{a = "", b = StringLength[x], k, t = 0}, If[y ≥ 1 && y ≤ b || x == "", If[{z} ≠ {}, Do[a = StringTake[x, {k, k}] <> a;
          If[SyntaxQ[a], t = 1; Break[]], {k, y, 1, -1}], Do[a = a <> StringTake[x, {k, k}]; If[SyntaxQ[a], t = 1; Break[]], {k, y, b}]];
      If[t == 1, a, $Failed], $Failed]]
End[]

Begin["`ActRemObj`"]
ActRemObj[x_ /; StringQ[x], y_ /; MemberQ[{"Act", "Rem"}, y]] :=
  Module[{a = $HomeDirectory <> "\\" <> x <> ".$ArtKr$", b, c = ToString[Definition4[x]]}, If[c === "$Failed", $Failed,
      If[HowAct[x] && y == "Rem", b = OpenWrite[a];
        WriteString[b, c]; Close[b]; ClearAllAttributes[x]; Remove[x]; "Remove",
        If[! HowAct[x] && y == "Act", If[FileExistsQ[a], b = OpenRead[a];
            Read[b];
```

```
            Close[b];
            DeleteFile[a];
            "Activate", Return[Defer[ActRemObj[x, y]]]]]]]]
End[]

Begin["`ActUcontexts`"]
ActUcontexts[x___] := Module[
    {a = MinusList[$Packages, {"System`", "Global`"}], b = FileNames[{"*.m", "*.tr"}, $InstallationDirectory, Infinity], c, d = {}, k, j},
    c = DeleteDuplicates[Map[StringTake[#, {1, Flatten[StringPosition[#, "`"]][[1]]}] &, a]]; If[{x} == {},
      For[k = 1, k ≤ Length[c], k++, For[j = 1, j ≤ Length[b], j++, If[FileBaseName[b[[j]]] <> "`" == c[[k]] ||
            MemberQ[Quiet[ContextMfile1[b[[j]]]], c[[k]]], AppendTo[d, c[[k]]]; Break[]]]]; MinusList[c, d],
      c = Map[StringTake[#, {1, -2}] &, c];
      For[k = 1, k ≤ Length[c], k++,
        For[j = 1, j ≤ Length[b], j++, If[FileBaseName[b[[j]]] == c[[k]] && MemberQ[{"m", "tr"}, FileExtension[b[[j]]]],
          AppendTo[d, c[[k]]]; Break[]]]]; Map[# <> "`" &, MinusList[c, d]]]
End[]

Begin["`SysContexts`"]
SysContexts[] := Module[{a = Contexts[], b = ActUcontexts[590]}, Select[a, ! SuffPref[#, b, 1] &]]
End[]

Begin["`AllContexts`"]
AllContexts[y___] :=
  Module[{a = Directory[], b = SetDirectory[$InstallationDirectory], c, h}, b = FileNames[{"*.m", "*.tr"}, {"*"}, Infinity];
    h[x_] := StringReplace[StringJoin[Map[FromCharacterCode, BinaryReadList[x]]],
        {"\n" -> "", "\r" -> "", "\t" -> "", " " -> "", "{" -> "", "}" -> ""}];
    c = Flatten[Select[Map[StringCases[h[#], {"BeginPackage[\"" ~~ Shortest[W__] ~~ "`\"]", "Needs[\"" ~~ Shortest[W__] ~~ "`\"]",
                  "Begin[\"" ~~ Shortest[W__] ~~ "`\"]", "Get[\"" ~~ Shortest[W__] ~~ "`\"]", "Package[\"" ~~ Shortest[W__] ~~
                    "`\"]"}] &, b], # ≠ {} &]]; c = DeleteDuplicates[Select[c, StringFreeQ[#, {"*", "="}] &]]; SetDirectory[a];
    c = Flatten[Append[Select[Map[StringReplace[StringTake[#, {1, -2}], {"Get[" -> "", "Begin[" -> "",
                    "Needs[" -> "", "BeginPackage[" -> "", "Package[" -> ""}, 1] &, c],
          # != "\"`Private`\"" &], {"\"Global`\"", "\"CloudObjectLoader`\""}]];
    c = Map[ToExpression, Sort[Select[DeleteDuplicates[Flatten[Map[If[StringFreeQ[#, ","], #, StringSplit[#, ","]] &, c]]],
          StringFreeQ[#, {"<>", "]", "["}] && StringTake[#, {1, 2}] != "\"`" &]]];
    If[{y} ≠ {}, MemberQ[c, y], c]]
End[]

Begin["`MultipleContexts`"]
MultipleContexts[x_] :=
  Module[{a = DeleteDuplicates[Map[StringTake[#, {1, Flatten[StringPosition[#, "`"]][[1]]}] &, Contexts[]]], b},
    b = Map[{#, NamesContext[#]} &, a];
    Map[If[MemberQ4[#[[2]], {ToString[x], #[[1]] <> ToString[x]}], #[[1]], Nothing] &, b]]
End[]

Begin["`ContextsCS`"]
ContextsCS[] :=
  Sort[DeleteDuplicates[Flatten[Map[If[StringLength[#] == 1 || StringFreeQ[#, "`"], {"System`", "Global`"}, StringTake[
          #, {1, Flatten[StringPosition[#, "`"]][[-1]]}]] &, Names[]]]]]
End[]

Begin["`WhatType`"]
WhatType[x_ /; StringQ[x]] := Module[{a = Quiet[Head[ToExpression[x]]], b = t, d, c = $Packages}, If[a === Symbol, Clear[t];
      d = Context[x];
      If[d == "Global`", d = Quiet[ProcFuncBlQ[x, t]];
        If[d === True, Return[{t, t = b}[[1]]], Return[{"Undefined", t = b}[[1]]]],
        If[d == "System`", Return[{d, t = b}[[1]]], Null]], Return[{ToString[a], t = b}[[1]]]];
    If[Quiet[ProcFuncBlQ[x, t]], If[MemberQ[{"Module", "DynamicModule", "Block"}, t], Return[{t, t = b}[[1]]], t = b;
      ToString[Quiet[Head[ToExpression[x]]]]], t = b; "Undefined"]]
End[]

Begin["`ExprOnLevels`"]
ExprOnLevels[x_] := Module[{a = {}, k = 1}, While[k ≤ Depth[x], a = Append[a, MinusList[Level[x, k], Level[x, k - 1]]];
      k++];
    a[[1 ;; -2]]]
End[]

Begin["`Integral1`"]
Integral1[x_, y__] :=
  Module[{d = {}, t = {}, k = 1, h = x, n = g, a = Map[ToString, Map[InputForm, {y}]], c = {}, b = Length[{y}]}, Clear[g];
```

```
    While[k ≤ b, AppendTo[c, Unique[g]]; AppendTo[d, ToString[c[[k]]]];
      AppendTo[t, a[[k]] → d[[k]]];
      h = ToExpression[StringReplace[ToString[h // InputForm], t[[k]]]]; h = Integrate[h, c[[k]]];
      h = ReplaceAll[h, Map[ToExpression, Part[t[[k]], 2] → Part[t[[k]], 1]]];
      k++];
    g = n;
    Map[Clear, c];
    Simplify[h]]
End[]

Begin["`ExprComp`"]
ExprComp[x_, z___] :=
  Module[{a = {x}, b, h = {}, F, q, t = 1}, F[y_List] := Module[{c = {}, d, p, k, j = 1}, For[j = 1, j ≤ Length[y], j++, k = 1;
          While[k < Infinity, p = y[[j]]; a = Quiet[Check[Part[p, k], $Failed]];
            If[a === $Failed, Break[], If[! SameQ[a, {}], AppendTo[c, a]]]; k++]]; c];
    q = F[a]; While[q ≠ {}, AppendTo[h, q]; q = Flatten[Map[F[{#}] &, q]]];
    If[{z} ≠ {} && ! HowAct[z], z = Map[Select[#, ! NumberQ[#] &] &, h]];
    Sort[Select[DeleteDuplicates[Flatten[h], Abs[#1] === Abs[#2] &], ! NumberQ[#] &]]]
End[]

Begin["`SortString`"]
SortString[x_ /; StringQ[x], y_ /; MemberQ[{Greater, Less}, y]] :=
  StringJoin[Sort[Characters[x], y[ToCharacterCode[#1][[1]], ToCharacterCode[#2][[1]]] &]]
End[]

Begin["`SortLS`"]
SortLS[x_ /; ListQ[x] || StringQ[x] || IntegerQ[x], y___] :=
  If[ListQ[x], Sort[x, y], If[StringQ[x], Evaluate, ToExpression][StringJoin[Sort[Characters[If[StringQ[x], x, ToString[x]]], y]]]]
End[]

Begin["`HelpPrint`"]
HelpPrint[n_ /; IntegerQ[n]] := Module[{a = Out[n], b, k = n + 1}, Print[ToString[a]]; While[k < Infinity,
      b = ToString[ToExpression["%" <> ToString[k]]]; If[! StringFreeQ[b, a], Return["End of HelpBase on context " <> a], Print[b]];
      k++]]
End[]

Begin["`HelpBasePac`"]
HelpBasePac[x_ /; ContextQ[x], y___] :=
  (If[{y} != {}, Information[#1], ToExpression[StringJoin["?", #1]]] & ) /@ NamesContext[x]
End[]

Begin["`UserLib`"]
UserLib[L_ /; FileExistsQ[L], f_ /; ListQ[f]] :=
  Module[{a, b = "", c, d = 0}, If[f[[1]] === "print" && f[[2]] === "all" && ! EmptyFileQ[L],
      FilePrint[L], If[f[[1]] === "print" && f[[2]] ≠ "all" && ! EmptyFileQ[L], a = OpenRead[L];
        While[b ≠ "EndOfFile", b = Read[a, String];
          If[SuffPref[b, f[[2]] <> "[", 1], Print[b]; d = 1;
            Continue[], If[d == 1, If[b === "\"$$$$$$\"", Close[a]; Break[], Print[b]; Continue[]], Continue[]]]];
        If[d == 1, Null, Print[f[[2]] <> " is absent in Library " <> StrStr[L]]],
        If[f[[1]] === "add", PutAppend[ToExpression["Definition[" <> f[[2]] <> "]"], L];
          PutAppend["$$$$$$", L], If[f[[1]] === "load" && f[[2]] === "all" && ! EmptyFileQ[L], Get[L];,
            If[f[[1]] === "load" && f[[2]] ≠ "all" && ! EmptyFileQ[L], a = OpenRead[L]; c = "";
              While[b ≠ "EndOfFile", b = Read[a, String];
                If[SuffPref[b, f[[2]] <> "[", 1], c = c <> b; d = 1;
                  Continue[], If[d == 1, If[b === "\"$$$$$$\"", Close[a];
                      Break[], c = c <> b;
                      Continue[]], Continue[]]]];
              If[d == 1, ToExpression[c], Print[f[[2]] <> " is absent in Library " <> StrStr[L]]],
              If[f[[1]] === "names" && f[[2]] === "list" && ! EmptyFileQ[L], a = OpenRead[L]; c = {};
                While[b ≠ "EndOfFile", b = Read[a, String];
                  If[Quiet[StringTake[b, {1, 1}]] ≠ " " && b ≠ "\"$$$$$$\"", c = Append[c, StringTake[b,
                        {1, Flatten[StringPosition[b, "["]][[1]] - 1}]], Continue[]]]; Close[a]; Return[c], Defer[UserLib[L, f]]]]]]]]]
End[]

Begin["`UnevaluatedQ`"]
UnevaluatedQ[F_ /; SymbolQ[F], x___] := Module[{a = Quiet[Check[F[x], "error", F::argx]]}, If[a === "error", "ErrorInArgs",
      If[ToString1[a] === ToString[F] <> "[" <> If[{x} == {}, "", ListStrToStr[Map[ToString1, {x}]] <> "]"], True, False]]]
End[]
```

```
Begin["`StringTake1`"]
StringTake1[x_ /; StringQ[x], y_] := Module[{a = Map[ToString, Map[InputForm, y]], b = {}, c, k = 1}, c = Sort[StringPosition[x, a]];
    b = {StringTake[x, {1, c[[1]][[1]] − 1}]};
    For[k, k ≤ Length[c] − 1, k ++, b = Append[b, StringTake[x, {c[[k]][[2]] + 1, c[[k + 1]][[1]] − 1}]]];
    Select[Append[b, StringTake[x, {c[[k]][[2]] + 1, −1}]], # ≠ "" &]]
End[]

Begin["`StringTake2`"]
StringTake2[x_ /; StringQ[x], y_] := Module[{a = Map[ToString, Map[InputForm, y]], b = {}, k = 1},
    For[k, k ≤ Length[a], k ++, b = Append[b, ToString1[a[[k]]] <> "−>" <> "\",\""]];
    StringSplit[ StringReplace[x, ToExpression[b]], ","]]
End[]

Begin["`StringTake3`"]
StringTake3[x_ /; StringQ[x], y_] :=
  Module[{a = FromCharacterCode[2], b = Map[ToString, Flatten[{y}]]}, StringSplit[StringReplace[x, GenRules[b, a]], a]]
End[]

Begin["`ListStrList`"]
ListStrList[x_ /; StringQ[x] || ListQ[x]] :=
  Module[{a = FromCharacterCode[2]}, If[StringQ[x] && ! StringFreeQ[x, a], Map[ToExpression, StringSplit[x, a]],
      If[ListQ[x], StringTake[StringJoin[Map14[StringJoin, Map[ToString1, x], a]], {1, −2}], x]]]
End[]

Begin["`ExtrVarsOfStr`"]
ExtrVarsOfStr[S_ /; StringQ[S], t_ /; MemberQ[{1, 2}, t], x___] :=
  Module[{k, j, d = {}, p, a = StringLength[S], q = Map[ToString, Range[0, 9]], h = 1, c = "",
      L = Characters["`!@#%^&∗(){}:\"\\/|<>?~−=+[];:'., 1234567890"], R = Characters["`!@#%^&∗(){}:\"\\/|<>?~=[];:'., "]}, Label[G];
    For[k = h, k ≤ a, k ++, p = StringTake[S, {k, k}]; If[! MemberQ[L, p], c = c <> p; j = k + 1;
        While[j ≤ a, p = StringTake[S, {j, j}]; If[! MemberQ[R, p], c = c <> p, AppendTo[d, c];
            h = j;
            c = "";
            Goto[G]]; j ++]]];
    AppendTo[d, c];
    d = Select[d, ! MemberQ[q, #] &]; d = Select[Map[StringReplace[#, {"+" −> "", "−" −> "", "_" −> ""}] &, d], # ≠ "" &];
    d = Flatten[Select[d, ! StringFreeQ[S, #] &]]; d = Flatten[Map[StringSplit[#, ", "] &, d]];
    If[t == 1, Flatten, Sort][If[{x} ≠ {}, Flatten, DeleteDuplicates][Select[d, ! MemberQ[{"\\", "#", ""}, StringTake[#, {1, 1}]] &]]]]
End[]

Begin["`Rename`"]
Rename[x_String /; HowAct[x], y_ /; ! HowAct[y]] :=
  Module[{a, b = Flatten[{PureDefinition[x]}], c, d}, If[! SameQ[b, {$Failed}], a = Attributes[x];
      c = ClearAllAttributes[x]; d = StringLength[x]; c = Map[ToString[y] <> StringTake[#, {d + 1, −1}] &, b];
      Map[ToExpression, c]; Clear[x]; SetAttributes[y, a]]]
End[]

Begin["`Rename1`"]
Rename1[x_String /; HowAct[x], y_ /; ! HowAct[y], z___] :=
  Module[{a = Attributes[x], b = Definition2[x][[1 ;; −2]], c = ToString[y]}, b = Map[StringReplacePart[#, c, {1, StringLength[x]}] &, b];
    ToExpression[b]; ToExpression["SetAttributes[" <> c <> ", " <> ToString[a] <> "]"];
    If[{z} == {}, ToExpression["ClearAttributes[" <> x <> ", " <> ToString[a] <> "]; Remove[" <> x <> "]"], Null]]
End[]

Begin["`RenameH`"]
RenameH[x_ /; HeadingQ1[x], y_ /; ! HowAct[y], z___] :=
  Module[{c, a = HeadName[x], d = StandHead[x], b = ToExpression["Attributes[" <> HeadName[x] <> "]"]},
    c = Flatten[{PureDefinition[a]}];
    If[c == {$Failed}, $Failed, If[c == {}, Return[$Failed], ToExpression["ClearAllAttributes[" <> a <> "]"]];
      ToExpression[ToString[y] <> DelSuffPref[Select[c, SuffPref[#, d <> " := ", 1] &][[1]], a, 1]];
      If[{z} == {}, RemProcOnHead[d]];
      If[! SameQ[PureDefinition[a], $Failed], ToExpression["SetAttributes[" <> ToString[a] <> ", " <> ToString[b] <> "]"]];
      ToExpression["SetAttributes[" <> ToString[y] <> ", " <> ToString[b] <> "]"];]]
End[]

Begin["`RenBlockFuncMod`"]
RenBlockFuncMod[x_ /; BlockFuncModQ[x], y_ /; SymbolQ[y]] :=
  Module[{t = {}, h, a = Options[x], b = Attributes[x], k = 1, n, c = Flatten[{PureDefinition[x]}], d = Flatten[{HeadPF[x]}]},
    For[k, k ≤ Length[c], k ++, h = StringReplace[c[[k]], StringJoin[d[[k]], " := "] −> ""];
      h = If[SuffPref[h, "Module[{", 1], "M", If[SuffPref[h, "Block[{", 1], "B", "F"]]; n = ToString[Unique[y]] <> h;
```

```
      AppendTo[t, n];
      ToExpression[StringReplace[c[[k]], ToString[x] <> "[" → n <> "[", 1]];
      If[a ≠ {}, ToExpression["SetOptions[" <> n <> ", " <> ToString[a] <> "]"]];
      If[b ≠ {}, ToExpression["SetAttributes[" <> n <> ", " <> ToString[b] <> "]"]]]; ClearAllAttributes[x];
    ClearAll[x];
    If[Length[t] == 1, t[[1]], t]]
End[]

Begin["`RenBlockFuncMod1`"]
RenBlockFuncMod1[x_ /; BlockFuncModQ[x], y_ /; SymbolQ[y] && ! HowAct[y]] := Module[{a = Options[x], b = Attributes[x]},
    ToExpression[StringReplace[PureDefinition[x], ToString[x] <> "[" → ToString[y] <> "[", 1]];
    ClearAttributes[x, b]; ClearAll[x]; If[a == {}, Null, SetOptions[y, a]]; SetAttributes[y, b]; y]
End[]

Begin["`DefAttributesH`"]
DefAttributesH[x_ /; HeadingQ[x], y_ /; MemberQ[{"Set", "Clear"}, y], z___] := Module[{a}, If[AttributesQ[{z}, a = Unique[g]],
      ToExpression[y <> "Attributes[" <> HeadName[x] <> "," <> ToString[{z}] <> "]"], {$Failed, a}]]
End[]

Begin["`AttributesH`"]
AttributesH[x_ /; HeadingQ[x]] := Attributes1[Symbol[HeadName[x]]]
End[]

Begin["`SymbolQ`"]
SymbolQ[x_] := ! SameQ[Quiet[Check[ToExpression["Attributes[" <> ToString[x] <> "]"], $Failed]], $Failed]
End[]

Begin["`SymbolsOfString`"]
SymbolsOfString[x_ /; StringQ[x], y___] :=
  Module[{a = Flatten[Map[Range[Sequences[#]] &, {{32, 35}, {37, 47}, {58, 64}, {91, 91}, {93, 96}, {123, 126}}]], b},
    b = DeleteDuplicates[StringSplit[StringReplace[x, GenRules[Map[FromCharacterCode, a], " "]]]];
    If[{y} == {}, b, If[y == 1, Select[b, ! MemberQ3[Range[48, 57], ToCharacterCode[#]] &],
        Select[Select[b, ! Quiet[SystemQ[#]] &], NameQ[#] &]]]
End[]

Begin["`TestBFM`"]
TestBFM[x_] := Module[{a = Flatten[{PureDefinition[x]}], b, d, h, p, k, j, t = {}},
    If[MemberQ[{$Failed, "System"}, a[[1]]], Return[$Failed], b = Flatten[{HeadPF[x]}];
      For[k = 1, k ≤ Length[a], k++, d = a[[k]]; p = Map[b[[k]] <> # &, {" := ", " = "}];
        h = StringReplace[d, {p[[1]] → "", p[[2]] → ""}, 1];
        If[SuffPref[h, "Module[{", 1], t = AppendTo[t, "Module"], If[SuffPref[h, "Block[{", 1], t = AppendTo[t, "Block"],
            If[SuffPref[h, "DynamicModule[{", 1], t = AppendTo[t, "DynamicModule"], t = AppendTo[t, "Function"]]]]]];
    If[Length[t] == 1, t[[1]], t]]
End[]

Begin["`SortRevStr`"]
SortRevStr[x_ /; StringQ[x], y_ /; MemberQ[{Reverse, Sort}, y], z___] := Module[{a = Characters[x]},
    If[y === Reverse, StringJoin[y[a]], StringJoin[If[{z} ≠ {} && z === SymbolGreater, Sort[a, z[#1, #2] &], Sort[a]]]]]
End[]

Begin["`TestArgsCall`"]
TestArgsCall[x_ /; BlockFuncModQ[x], y___] :=
  Module[{a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}], c = "$$$", n = ToString[x], d, p, h = {}, k = 1},
    While[k ≤ Length[b], d = c <> n;
      ToExpression[c <> b[[k]] <> ":=90"];
      p = Symbol[d][y];
      ToExpression["Remove[" <> d <> "]"];
      If[p === 90, AppendTo[h, a[[k]]]];
      k++];
    If[Length[h] == 1, h[[1]], h]]
End[]

Begin["`TestArgsLocals`"]
TestArgsLocals[x_ /; BlockFuncModQ[x], y___] :=
  Module[{a = ArgsBFM[x, 90], b, a1, b1, c1 = {}, d1 = {}, c, g, n, f, s, p = Flatten[{PureDefinition[x]}]},
    b = If[Length[p] > 1, Locals1[x], {Locals1[x]}];
    g = Length[p]; b = Map[If[# === "Function", {}, #] &, b]; a1 = Map[Length[#] == Length[DeleteDuplicates[#]] &, a];
    b1 = Map[Length[#] == Length[DeleteDuplicates[#]] &, b];
    Do[AppendTo[c1, Intersection[a[[k]], b[[k]]] == {}], {k, 1, g}]; c = If[MemberQ[Flatten[Set[n, {a1, b1, c1}]], False], False, True];
```

```
    d1 = RestListList[n];  s = Map[If[And @@ #, Nothing, #] &, d1];
    f[z_] := MemberQ[z, False];
    If[{y} ≠ {} && ! HowAct[y] && ! c, y = {p[[PositionsListCond[d1, f]]], If[Length[s] == 1, s[[1]], s]}, Null];  c]
End[]

Begin["`TestFactArgs`"]
TestFactArgs[x_ /; ProcQ[x] || QFunction[x], y__] := Module[{b, c = {}, d, k = 1, p = {y}, a = Flatten[{HeadPF[x]}][[1]]},
    b = StrToList["{" <> StringTake[a, {StringLength[ToString[x]] + 2, -2}] <> "}"];
    b = Map[StringSplit[#, "_"] &, b];
    If[Length[b] == Length[p] && StringFreeQ[a, "__"], While[k ≤ Length[b], d = b[[k]];
        If[Length[d] == 1, AppendTo[c, True],
          If[Length[d] == 2 && SymbolQ[d[[2]]], AppendTo[c, Head[p[[k]]] === Symbol[d[[2]]]], If[SuffPref[d[[2]], " /; ", 1],
              AppendTo[c, ToExpression[StringReplace3[StringTake[d[[2]], {5, -1}], d[[1]], ToString[p[[k]]]]]]]];  k++];  c, $Failed]]
End[]

Begin["`SymbolQ1`"]
SymbolQ1[x_]  :=  If[Length[ToCharacterCode[ToString[x]]] == 1,  True,  False]
End[]

Begin["`CatN`"]
CatN[s_ /; StringQ[s], n_ /; IntegerQ[n] && n ≥ 1] := Module[{a = "", k = 1}, For[k, k ≤ n, k++, a = a <> s]; a]
End[]

Begin["`PartProc`"]
PartProc[P_ /; BlockModQ[P]] := Module[{a = ProcBody[P]}, {StringReplace[PureDefinition[P], a → "Procedure Body", 1], a}]
End[]

Begin["`HeadPF`"]
HeadPF[x_ /; BlockFuncModQ[x]] :=
  Module[{a = Select[Flatten[{PureDefinition[x]}], ! SuffPref[#, "Default[", 1] &], b, c = ToString[x]},
    b = Map[StringTake[#, {1, Flatten[StringPosition[#, {" := ", " = "}]][[1]] - 1}] &, a];
    If[Length[b] == 1, b[[1]], b]]
End[]

Begin["`Headings`"]
Headings[x_ /; BlockFuncModQ[x]] :=
  Module[{c = {{"Block"}, {"Function"}, {"Module"}}, k = 1, a = Flatten[{PureDefinition[x]}], n, d, h, p, t},
    While[k ≤ Length[a], d = a[[k]]; n = ToString[Unique["agn"]]; ToExpression[n <> d];
      ClearAll[p];  h = HeadPF[t = n <> ToString[x]];
      d = StringTake[h, {StringLength[n] + 1, -1}];
      BlockFuncModQ[t, p];
      If[p == "Block", AppendTo[c[[1]], d], If[p == "Function", AppendTo[c[[2]], d], AppendTo[c[[3]], d]]];
      ToExpression["Remove[" <> t <> "," <> n <> "]"]; k++];
    c = Select[c, Length[#] > 1 &];  If[Length[c] == 1, c[[1]], c]]
End[]

Begin["`CompileFuncQ`"]
CompileFuncQ[x_] := If[SuffPref[ToString[InputForm[Definition2[x]]], "Definition2[CompiledFunction[{", 1], True, False]
End[]

Begin["`HeadingsPF`"]
HeadingsPF[x___ /; SameQ[x, {}]] := Module[{a = {}, b, c = {{"Block"}, {"Function"}, {"Module"}}, d = {}, k = 1, t},
    Map[If[Quiet[Check[BlockFuncModQ[#], False]], AppendTo[a, #], Null] &, Names["`*"]];
    b = Map[Headings[#] &, a];  While[k ≤ Length[b], t = b[[k]];
      If[NestListQ[t], d = Join[d, t], AppendTo[d, t]]; k++];
    Map[If[#[[1]] == "Block", c[[1]] = Join[c[[1]], #[[2 ;; -1]]],
        If[#[[1]] == "Function", c[[2]] = Join[c[[2]], #[[2 ;; -1]]], c[[3]] = Join[c[[3]], #[[2 ;; -1]]]]] &, d];
    c = Select[c, Length[#] > 1 &];  If[Length[c] == 1, c[[1]], c]]
End[]

Begin["`DelRestPF`"]
DelRestPF[r_ /; MemberQ[{"d", "r"}, r], x___] :=
  Module[{b, c, p, f = "$Art27Kr20$.mx", a = Quiet[Select[{x}, BlockFuncModQ[#] &]], k = 1}, If[r == "d", b = Map[Definition2, a];
      Save[f, b];
      Map[ClearAllAttributes, a];
      Map[Remove, a];, c = Get[f];
      DeleteFile[f];  For[k, k ≤ Length[c], k++, p = c[[k]];
        ToExpression[p[[1 ;; -2]]];
        ToExpression["SetAttributes[" <>
```

```
            StringTake[p[[1]], {1, Flatten[StringPosition[p[[1]], "["][[1]] - 1}] <> "," <> ToString[p[[-1]]] <> "]"]]]
End[]

Begin["`DelRestPF1`"]
DelRestPF1[r_ /; MemberQ[{"d", "r"}, r], f_ /; StringQ[f], x___] :=
  Module[{a = Quiet[Select[{x}, BlockFuncModQ[#] &]], b, c, p, k = 1}, If[r == "d", b = Map[Definition2, a];
      Save[f, b];
      Map[ClearAllAttributes, a];
      Map[Remove, a];, c = Get[f];
      DeleteFile[f];  For[k, k ≤ Length[c], k++, p = c[[k]];
        ToExpression[p[[1 ;; -2]]];
        ToExpression["SetAttributes[" <>
            StringTake[p[[1]], {1, Flatten[StringPosition[p[[1]], "["][[1]] - 1}] <> "," <> ToString[p[[-1]]] <> "]"]]]
End[]

Begin["`SystemQ`"]
SystemQ[S_] := If[Off[Definition::ssle];
    ! ToString[Definition[S]] === Null && MemberQ[Names["System`*"],  ToString[S]], On[Definition::ssle];
    True, On[Definition::ssle]; False]
End[]

Begin["`SysFunctionQ`"]
SysFunctionQ[x_] := Module[{a}, If[! SymbolQ[x], False, If[! SystemQ[x], False, a = SyntaxInformation[x];
      If[a === {}, False, If[Select[a[[1]][[2]], ! SameQ[#, OptionsPattern[]] &] === {}, False, True]]]]
End[]

Begin["`SeqUnion`"]
SeqUnion[x__] := Sequence[x]
End[]

Begin["`IndexedQ`"]
IndexedQ[x_] := Module[{a = Quiet[ToString[x]], b}, b = StringPosition[a, "[["];
    If[StringTake[a, {-2, -1}] == "]]"  && + SymbolQ[StringTake[a, {1, b[[1]][[1]] - 1}]], True, False]]
End[]

Begin["`Decomp`"]
Decomp[x_] := Module[{b = {}, c = DeleteDuplicates[Flatten[Level[x, Infinity]], Abs[#1] === Abs[#2] &], k}, Label[ArtKr];
    For[k = 1, k ≤ Length[c], k++, b = Append[b, If[AtomQ[c[[k]]], c[[k]], {Level[c[[k]], -1], Head[c[[k]]]}]];
    b = DeleteDuplicates[Flatten[b], Abs[#1] === Abs[#2] &];  If[c == b, Return[b], c = b;  b = {};  Goto[ArtKr]]]
End[]

Begin["`Index`"]
Index[x_ /; IndexedQ[x]] :=
  Module[{a = Quiet[ToString[x]], b, c}, b = Flatten[StringPosition[a, "[["]];  ToExpression[StringTake[a, {b[[2]] + 1, -3}]]]
End[]

Begin["`IndexQ`"]
IndexQ[x_ /; StringQ[x]]  :=  If[! SuffPref[x, "[[", 1] && StringDependQ[x, {"[[", "]]"}] &&
      EvenQ[StringCount[x, {"[[", "]]"}]] && StringTake[x, {-2, -1}] == "]]", True, False]
End[]

Begin["`Indices`"]
Indices[x_ /; IndexQ[x]] := Module[{a = "]]", b = "[[", h = x, c, d = {}, g, t, k, y, p = StringLength[x]},
    While[p ≥ 4, For[k = p - 2, k ≥ 1, k--, c = StringTake[h, {k, p}];
        If[StringCount[c, a] == StringCount[c, b], d = Append[d, {k, c}];
          Break[]]; k = d[[-1]][[1]]; h = StringTake[h, {1, k - 1}];
      If[IndexQ[h], p = StringLength[h]; Continue[], Break[]]];
    g = Reverse[Map[{#[[1]], #[[2]], #[[1]] + StringLength[#[[2]]]} &, d]]; y = g[[-1]][[2]];
    For[k = Length[g] - 1, k ≥ 1, k--, If[g[[k]][[3]] == g[[k + 1]][[1]], y = g[[k]][[2]] <> y]]; y]
End[]

Begin["`PatternQ`"]
PatternQ[x_] := If[Head[x] === Pattern, True, False]
End[]

Begin["`DefOp`"]
DefOp[x_ /; StringQ[x] && SymbolQ[x] || SymbolQ[ToExpression[x]], y___] := Module[{a = PureDefinition[x], b = {y}, c, d},
    If[a === $Failed, "Undefined", If[SuffPref[a, x <> "[", 1], $Failed, c[h_] := StringReplace[a, x <>  " " <> d <> " " -> "", 1];
        If[SuffPref[a, x <> " = ", 1], d = "=", d = ":="];  If[b ≠ {} && ! HowAct[y], y = c[d]]; d]]
```

```
End[]

Begin["`DirEmptyQ`"]
DirEmptyQ[d_ /; DirQ[d]] := Module[{p = StandPath[StringReplace[d, "/" -> "\\"]], a = "$DirFile$", b, c, h = " 0 File(s)  "},
    b = Run["Dir " <> p <> If[SuffPref[p, "\\", 2], "", "\\"] <> "*.* > " <> a];
    If[b ≠ 0, $Failed, Do[c = Read[a, String], {6}]];  DeleteFile[Close[a]];  ! StringFreeQ[c, h]]
End[]

Begin["`GenRules`"]
GenRules[x_, y_, z___] := Module[
    {a, b = Flatten[{x}], c = Flatten[If[Map[ListQ, {x, y}] == {True, False}, PadLeft[{}, Length[x], y], {y}]]}, a = Min[Map[Length, {b, c}]];
    b = Map9[Rule, b[[1 ;; a]], c[[1 ;; a]]];  If[{z} == {}, b, b = Map[List, b];  If[Length[b] == 1, Flatten[b], b]]
End[]

Begin["`GenRules1`"]
GenRules1[x_, y_, h_ /; h == "r" || h == "rd", z___] := Module[
    {a, b = Flatten[{x}], c = Flatten[If[Map[ListQ, {x, y}] == {True, False}, PadLeft[{}, Length[x], y], {y}]]}, a = Min[Map[Length, {b, c}]];
    b = Map9[If[h == "r", Rule, RuleDelayed], b[[1 ;; a]], c[[1 ;; a]]];  If[{z} == {}, b, b = Map[List, b];  If[Length[b] == 1, Flatten[b], b]]
End[]

Begin["`GenRules2`"]
GenRules2[x_ /; ListQ[x], y_] := If[ListQ[y], Map[Rule[x[[#]], y[[#]]] &, Range[1, Min[Length[x], Length[y]]]],
    Map[Rule[x[[#]], y] &, Range[1, Length[x]]]]
End[]

Begin["`RevRules`"]
RevRules[x_ /; RuleQ[x] || ListQ[x] && DeleteDuplicates[Map[RuleQ, x]] == {True}] :=
  Module[{a = Flatten[{x}], b}, b = Map[#[[2]] → #[[1]] &, a];
    If[Length[b] == 1, b[[1]], b]]
End[]

Begin["`VarExch`"]
VarExch[L_List /; Length[L] == 2 || ListListQ[L] && Length[L[[1]]] == 2] :=
  Module[{Kr, k = 1}, Kr[p_ /; ListQ[p]] := Module[{a = Map[Attributes, p], b, c, m, n},
        ToExpression[{"ClearAttributes[" <> StrStr[p[[1]]] <> "," <> ToString[a[[1]]] <> "]",
            "ClearAttributes[" <> StrStr[p[[2]]] <> "," <> ToString[a[[2]]] <> "]"}];
        {b, c} = ToExpression[{"ToString[Definition[" <> StrStr[p[[1]]] <> "]]", "ToString[Definition[" <> StrStr[p[[2]]] <> "]]"}];
        If[MemberQ[{b, c}, "Null"],
          Print[VarExch::"Both arguments should be defined but uncertainty had been detected: ", p];  Return[], Null];
        {m, n} = Map4[StringPosition, Map[StrStr, {b, c}], {" := ", " = "}];
        {n, m} = {StringTake[b, {1, m[[1]][[1]] - 1}] <> StringTake[c, {n[[1]][[1]], -1}],
            StringTake[c, {1, n[[1]][[1]] - 1}] <> StringTake[b, {m[[1]][[1]], -1}]};
        ToExpression[{n, m}];
        Map[ToExpression, {"SetAttributes[" <> StrStr[p[[1]]] <> "," <> ToString[a[[2]]] <> "]",
            "SetAttributes[" <> StrStr[p[[2]]] <> "," <> ToString[a[[1]]] <> "]"}];
    If[! ListListQ[L], Kr[L], For[k, k ≤ Length[L], k++, Kr[L[[k]]]];]
End[]

Begin["`NestListQ`"]
NestListQ[x_] := SameQ[ListQ[x] && DeleteDuplicates[Map[ListQ, x]], {True}]
End[]

Begin["`NestListQ1`"]
NestListQ1[x_] := ! SameQ[ListQ[x] && Length[Select[x, ListQ[#] &]], 0]
End[]

Begin["`MapNestList`"]
MapNestList[x_ /; ListQ[x], f_ /; SymbolQ[f], y___] :=
  Module[{a = ToString1[x], b, c, d, h = FromCharacterCode[3]}, b = StringPosition[a, "{"];
    d = StringReplacePart[a, ToString[f] <> "[" <> h, b]; b = StringPosition[d, "}"];
    ToExpression[
      StringReplace[StringReplacePart[d, If[{y} ≠ {}, "},  " <>  StringTake[ToString1[{y}], {2, -2}] <>  "]", "}]"], b], h -> "{"]]]
End[]

Begin["`AttributesQ`"]
AttributesQ[x_ /; ListQ[x], y___] :=
  Module[{a, b = {}}, Map[If[Quiet[Check[SetAttributes[a, #], $Failed]] === $Failed, AppendTo[b, #]] &, x];
    If[b ≠ {}, If[{y} ≠ {} && ! HowAct[y], y = b];  False, True]]
End[]
```

```
Begin["`RuleQ`"]
RuleQ[x_] := If[MemberQ[{Rule, RuleDelayed}, Head[x]], True, False]
End[]

Begin["`QmultiplePF`"]
QmultiplePF[x_, y___] := Module[{a = Flatten[{PureDefinition[x]}]},
    If[MemberQ[{{"System"}, {$Failed}}, a] || Length[a] == 1, False, If[{y} ≠ {} && ! HowAct[y], y = If[Length[a] == 1, a[[1]], a];
      True]]
End[]

Begin["`VarExch1`"]
VarExch1[L_List /; Length[L] == 2 || ListListQ[L] && Length[L[[1]]] == 2] := Module[{Art, k = 1, d}, Art[p_List] :=
      Module[{a = Quiet[Check[Map[Attributes, p], $Aborted]], b, c, m, n}, If[a == $Aborted, Return[Defer[VarExch1[L]]], Null];
        If[HowAct[$Art$], b = $Art$; Clear[$Art$]; m = 1, Null];
        If[HowAct[$Kr$], c = $Kr$; Clear[$Kr$]; n = 1, Null];
        ToExpression[{"ClearAttributes[" <> StrStr[p[[1]]] <> "," <> ToString[a[[1]]] <> "]",
            "ClearAttributes[" <> StrStr[p[[2]]] <> "," <> ToString[a[[2]]] <> "]"}];
        ToExpression[{"Rename[" <> StrStr[p[[1]]] <> "," <> "$Art$" <> "]", "Rename[" <> StrStr[p[[2]]] <> "," <> "$Kr$" <> "]"}];
        ToExpression["Clear[" <> StrStr[p[[1]]] <> "," <> StrStr[p[[2]]] <> "]"];
        ToExpression[
          {"Rename[" <> StrStr["$Kr$"] <> "," <> p[[1]] <> "]", "Rename[" <> StrStr["$Art$"] <> "," <> p[[2]] <> "]"}];
        Map[ToExpression, {"SetAttributes[" <> StrStr[p[[1]]] <> "," <> ToString[a[[2]]] <> "]",
            "SetAttributes[" <> StrStr[p[[2]]] <> "," <> ToString[a[[1]]] <> "]"}];
        If[m == 1, $Art$ = b, Null];
        If[n == 1, $Kr$ = c, Null];]; If[! ListListQ[L], Art[L], For[k, k ≤ Length[L], k++, Art[L[[k]]]]]
End[]

Begin["`ClearCS`"]
ClearCS[x_ /; MemberQ[{ClearAll, Remove}, x]] :=
  Module[{a = Join[Names["Global`*"], {"a", "b", "c", "d", "h", "k", "p", "S", "x", "y"}]},
    Quiet[Mapp[ClearAttributes, a, Protected]]; Quiet[Map[x, a]];]
End[]

Begin["`StringReplace1`"]
StringReplace1[S_ /; StringQ[S], L_ /; ListListQ[L] && Length[L[[1]]] == 2 && MatrixQ[L, IntegerQ] && Sort[Map[Min, L]][[1]] ≥ 1,
    P_ /; ListQ[P]] := Module[{a = {}, b, k = 1},
    If[Sort[Map[Max, L]][[-1]] ≤ StringLength[S] && Length[P] == Length[L], Null, Return[Defer[StringReplace1[S, L, P]]]];
    For[k, k ≤ Length[L], k++, b = L[[k]];
      a = Append[a, StringTake[S, {b[[1]], b[[2]]}] → ToString[P[[k]]]];
    StringReplace[S, a]]
End[]

Begin["`ProcFuncBlQ`"]
ProcFuncBlQ[x_, y_ /; ! HowAct[y]] := Module[{a = ToString[HeadPF[x]], b = ToString[y] <> " = ", c = PureDefinition[x]},
    If[ListQ[c], False, If[SuffPref[a, "HeadPF[", 1], If[SuffPref[a, " & ]", 2], y = "PureFunction";
          True, False], If[HeadingQ[a], If[SuffPref[c, a <> " := Module[{", 1],
            y = "Module"; True, If[SuffPref[c, a <> " := Block[{", 1], y = "Block";
              True, If[SuffPref[c, a <> " := DynamicModule[{", 1], y = "DynamicModule"; True, y = "Function"; True]]], False]]]]
End[]

Begin["`ProcFuncBlQ1`"]
ProcFuncBlQ1[x_, y_ /; ! HowAct[y]] := Module[{a = Flatten[{HeadPF[x]}], b = {}, c = Flatten[{PureDefinition[x]}], d = {}, k},
    Do[If[SuffPref[ToString[x], " & ", 2], AppendTo[b, "PureFunction"]; AppendTo[d, True],
        If[c[[k]] === $Failed, AppendTo[b, $Failed]; AppendTo[d, False],
          If[SuffPref[c[[k]], a[[k]] <> " := Module[{", 1], AppendTo[b, "Module"]; AppendTo[d, True],
            If[SuffPref[c[[k]], a[[k]] <> " := Block[{", 1], AppendTo[b, "Block"]; AppendTo[d, True],
              If[SuffPref[c[[k]], a[[k]] <> " := DynamicModule[{", 1], AppendTo[b, "DynamicModule"]; AppendTo[d, True],
                AppendTo[b, "Function"]; AppendTo[d, True]]]]], {k, 1, Length[c]}];
    {b, d} = Map[If[Length[#] == 1, #[[1]], #] &, {b, d}]; y = b; d]
End[]

Begin["`VizContext`"]
VizContext[x_ /; ContextQ[x]] := Module[{a = CNames[x]},
    If[a == {}, $Failed, CreateDocument[Map[StringReplace[#, x –> ""] &, Map[ToString1, Map[Definition, a]]]]]]
End[]

Begin["`VizContentsNB`"]
VizContentsNB[x_] := Module[{}, If[StringQ[x] && FileExistsQ[x] && FileExtension[x] == "nb", CopyToClipboard[Get[x]];
```

```
      Paste[], $Failed]]
End[]

Begin["`StringReplace2`"]
StringReplace2[S_ /; StringQ[S], s_ /; StringQ[s], Exp_] :=
  Module[{b, c, d, a = Join[CharacterRange["A", "Z"], CharacterRange["a", "z"]], k = 1},
    b = Quiet[Select[StringPosition[S, s], ! MemberQ[a, StringTake[S, {#[[1]] - 1, #[[1]] - 1}]] &&
                ! MemberQ[a, StringTake[S, {#[[2]] + 1, #[[2]] + 1}]] &]];  StringReplacePart[S, ToString[Exp], b]]
End[]

Begin["`StringReplace3`"]
StringReplace3[S_ /; StringQ[S], x__] := Module[{b = S, c, j = 1, a = Map[ToString, {x}]},
    c = Length[a];  If[OddQ[c], S, While[j ≤ c/2, b = StringReplace2[b, a[[2 * j - 1]], a[[2 * j]]];
        j ++]; b]]
End[]

Begin["`StringStringQ`"]
StringStringQ[x_] := If[! StringQ[x], False, If[SuffPref[x, "\"", 1] && SuffPref[x, "\"", 2], True, False]]
End[]

Begin["`StringJoin1`"]
StringJoin1[x_ /; ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True}] :=
  Module[{a = x, b = Length[x], c = "", k = 1}, While[k ≤ b - 1, c = c <> a[[k]] <> ", ";
      k ++];
    c <> a[[-1]]]
End[]

Begin["`ExprPatternQ`"]
ExprPatternQ[x_] := ! StringFreeQ[ToString[FullForm[x]], {"BlankSequence[]", "BlankNullSequence[]", "Blank[]"}]
End[]

Begin["`ReadFullFile`"]
ReadFullFile[f_ /; StringQ[f], y___] := Module[{a, b = $Art6Kr$}, If[FileExistsQ[f], a = f, ClearAll[$Art6Kr$];
      If[! FileExistsQ1[f, $Art6Kr$], Return[$Failed], a = $Art6Kr$[[1]]]]; $Art6Kr$ = b;
    StringReplace[StringJoin[Map[FromCharacterCode, BinaryReadList[a]]],
      "\r\n" → If[{y} ≠ {}, If[StringQ[y], y, If[! HowAct[y], y = a;  "", ""]], ""]]]
End[]

Begin["`ReadFullFile1`"]
ReadFullFile1[x_ /; FileExistsQ[x]] := StringReplace[Quiet[Check[ReadString[x], ""]], "\r\n"  ->  ""]
End[]

Begin["`BinaryListQ`"]
BinaryListQ[L_ /; ListQ[L]] := MemberQ[{{0}, {1}, {0, 1}}, Sort[DeleteDuplicates[Flatten[L]]]]
End[]

Begin["`ToString1`"]
ToString1[x_] := Module[{a = "$$Art27$Kr20$$.txt", b = "", c, k = 1}, Write[a, x]; Close[a];
    For[k, k < Infinity, k ++, c = Read[a, String];
      If[SameQ[c, EndOfFile], Return[DeleteFile[Close[a]];
          b], b = b <> StrDelEnds[c, " ", 1]]]]
End[]

Begin["`ToString2`"]
ToString2[x_] := Module[{a}, If[ListQ[x], SetAttributes[ToString1, Listable];  a = ToString1[x];
      ClearAttributes[ToString1, Listable];
      a, ToString1[x]]]
End[]

Begin["`ToString3`"]
ToString3[x_] := StringReplace[ToString1[x], "\"" -> ""]
End[]

Begin["`ToString4`"]
ToString4[x_] := If[! SymbolQ[x], "(" <> ToString1[x] <> ")", ToString1[x]]
End[]

Begin["`DefToString`"]
DefToString[x_ /; StringQ[x]] := Module[{a = Definition[x], b = ToString[Unique["agn"]], c}, Write[b, a];
```

```
    Close[b];
    a = ReadString[b];
    DeleteFile[b];
    If[Set[c, Attributes[x]] == {}, a, b = StringPosition[a, "\r\n \r\n"][[1]];
      a = StringTake[a, {b[[2]] + 1, -1}] <> StringTake[a, b] <> "Attributes[" <> ToString[x] <> "] = " <> ToString[c] <> ";";
    StringReplace[a, {"\r\n \r\n" -> "\r\n", "\r\n\r\n" -> ""}]]
End[]

Begin["`Arity`"]
Arity[P_ /; SystemQ[P] || CompileFuncQ[P] || BlockFuncModQ[P] || PureFuncQ[P]] :=
  Module[{a}, If[SystemQ[P], "System", a = Args[P];
      Mapp[SetAttributes, {ToString, StringFreeQ}, Listable];  a = Map[ToString, a];
      a = Map[If[DeleteDuplicates[StringFreeQ[#, "__"]] === {True}, Length[#], "Undefined"] &, If[NestListQ[a], a, {a}]];
      Mapp[ClearAttributes, {ToString, StringFreeQ}, Listable]; If[Length[a] == 1, a[[1]], a]]]
End[]

Begin["`Arity1`"]
Arity1[P_ /; SystemQ[P] || CompileFuncQ[P] || PureFuncQ[P] || BlockFuncModQ[P]] :=
  Module[{a}, If[SystemQ[P], "System", a = Args[P];  a = Mapp1[ToString, a];
      a = Map[If[DeleteDuplicates[StringFreeQ[#, "__"]] === {True}, Length[#], "Undefined"] &, If[NestListQ[a], a, {a}]];
      If[Length[a] == 1, a[[1]], a]]]
End[]

Begin["`ArityBFM`"]
ArityBFM[x_ /; BlockFuncModQ[x]] := Module[{a = Flatten[{HeadPF[x]}], b},
    b = Map[If[! StringFreeQ[#, {"__", "___"}], "Undefined", Length[ToExpression[
                "{" <> StringTake[StringReplace[#, ToString[x] <> "[" → "", 1], {1, -2}] <> "}"]]] &, a];  If[Length[b] == 1, b[[1]], b]]
End[]

Begin["`ArityBFM1`"]
ArityBFM1[x_Symbol] := Module[{a, b, c, d = {}}, Map[ClearAll, {a, c}];
    If[DeleteDuplicates[Flatten[{ProcFuncBlQ1[x, c]}]] ≠ {True}, $Failed, Quiet[Check[ArgsBFM[x, a], Return[$Failed]]];
      Do[b = {0, 0, 0};  Map[If[! StringFreeQ[#, "___"], b[[3]] ++, If[! StringFreeQ[#, "__"], b[[2]] ++, b[[1]] ++]] &, a[[k]]];
        AppendTo[d, AppendTo[b, Plus[0, Sequences[b]]]];  b = {0, 0, 0}, {k, 1, Length[a]}];  If[Length[d] == 1, d[[1]], d]]]
End[]

Begin["`AritySystemFunction`"]
AritySystemFunction[x_, y___] := Module[{a, b, c, d, g, p}, If[! SysFunctionQ[x], $Failed, a = SyntaxInformation[x];
      If[a == {}, $Failed,  c = {0, 0, 0, 0};
        b = Map[ToString, a[[1]][[2]]];
        b = Map[If[SuffPref[#, "{", 1], "_", If[SuffPref[#, {"Optional", "OptionsPattern"}, 1], "_.", #]] &, b];
        If[{y} ≠ {} && ! HowAct[y], y = Map[ToExpression[ToString[Unique["x"]] <> #] &, b], Null];
        Map[If[# == "_", c[[1]] ++, If[# == "_.", c[[2]] ++, If[# == "__", c[[3]] ++, c[[4]] ++]]] &, b];
        {p, c, d, g} = c;  d = DeleteDuplicates[{p + d, If[d ≠ 0 || g ≠ 0, Infinity, p + c]}];
        If[d ≠ {0, Infinity}, d, Quiet[x[Null]];
          If[Set[p, Messages[x]] == {}, d,  p = Select[Map[ToString, Map[Part[#, 2] &, p]], ! StringFreeQ[#, " expected"] &];
            p = Map[StringTake[#, {Flatten[StringPosition[#, ";"]][[1]] + 1, -2}] &, p];
            p = ToExpression[Flatten[StringCases[p, DigitCharacter ..]]];  If[p == {}, {1}, If[Length[p] > 1, {Min[p], Max[p]}, p]]]]]]]
End[]

Begin["`AritySystemFunction1`"]
AritySystemFunction1[x_ /; SystemQ[x], y___] :=
  Module[{a = FindFile1["functioninformation.m", $InstallationDirectory], b, c, d = "" , k, p, g = {0, 0, 0, 0}},
    a = StringReplace[StringReplace[ReadString[a], {"\n"  -> "", "\r" -> ""}], "    " -> " "];
    If[StringFreeQ[a, c = ToString1[p = ToString[x]]], $Failed, c = Flatten[StringPosition[a, "{" <> c]][[1]];
      For[k = c, k < Infinity, k ++, d = d <> StringTake[a, {k, k}];  If[SyntaxQ[d], Break[]]];
      d = ToExpression[d][[2]];  If[{y} ≠ {} && ! HowAct[y], y = d, Null];  d = Map[ToString, d];
      d = Flatten[Map[If[SuffPref[#, {"Optional", "OptionsPattern"}, 1], "_.", If[SuffPref[#, "{", 1], "_", #]] &, d]];
      Map[If[# == "_", g[[1]] ++, If[# == "_.", g[[2]] ++, If[# == "__", g[[3]] ++, g[[4]] ++]]] &, d];
      {p, c, d, g} = g;   d = Flatten[DeleteDuplicates[{p + d, If[d ≠ 0 || g ≠ 0, Infinity, p + c]}]]]]
End[]

Begin["`MessagesOut`"]
MessagesOut[x_ /; StringQ[x], y___] := Module[{a, b}, If[! (SyntaxQ[x] && CallQ[x]), $Failed, a = HeadName1[x];
      If[BlockFuncModQ[a] || SysFunctionQ[Symbol[a]], If[BlockFuncModQ[a], Null, ToExpression["Messages[" <> a <> "]={}"]];
        Quiet[ToExpression[x]];  b = ToExpression["Messages[" <> a <> "]"];
        If[BlockFuncModQ[a], Null, ToExpression["Messages[" <> a <> "]={}"]];
        If[{y} ≠ {} && ! HowAct[y], y = Select[Map[{Part[#[[1]], 1], Part[#, 2]} &, b], ! SameQ[#[[1]], $Off[]] &];
          y = If[Length[y] == 1, y[[1]], y], Null];  b, $Failed]]]
```

```
End[]

Begin["`$InBlockMod`"]
$InBlockMod :=
  Quiet[ Check[StringTake[If[Stack[Block] ≠ {}, ToString[InputForm[Stack[Block][[1]]]],  If[Stack[Module] ≠ {}, StringReplace3[
              ToString[InputForm[Stack[Module][[1]]]], Sequences[Riffle[Select[StringReplace[StringSplit[
                        StringTake[SubStrSymbolParity1[ToString[InputForm[Stack[Module][[1]]]], "{", "}"][[1]], {2, −2}], " "],
                      "," → ""], StringTake[#, −1] == "$" &], Mapp[StringTake, Select[StringReplace[StringSplit[
                         StringTake[SubStrSymbolParity1[ToString[InputForm[Stack[Module][[1]]]], "{", "}"][[1]], {2, −2}], " "],
                       "," → ""], StringTake[#, −1] == "$" &], {1, −2}]]]]], $Failed], {10, −2}], Null]]
End[]

Begin["`DefFunc`"]
DefFunc[x_ /; SymbolQ[x] || StringQ[x]] :=
  Module[{a = GenRules[Map14[StringJoin, {"Global`", Context[x]}, ToString[x] <> "`"], ""],
      b = StringSplit[ToString[InputForm[Definition[x]]], "\n \n"]}, ToExpression[Map[StringReplace[#, a] &, b]];  Definition[x]]
End[]

Begin["`SyntaxLength1`"]
SyntaxLength1[x_ /; StringQ[x], y___] :=
  Module[{a = "", b = 1, d, h = {}, c = StringLength[x]}, While[b ≤ c, d = SyntaxQ[a = a <> StringTake[x, {b, b}]];
      If[d, AppendTo[h, StringTrim2[a, {"+", "−", " "}, 3]]];  b++];  h = DeleteDuplicates[h];
    If[{y} ≠ {} && ! HowAct[{y}[[1]]], {y} = {h}];  If[h == {}, 0, StringLength[h[[−1]]]]]
End[]

Begin["`PrefixQ`"]
PrefixQ[x_ /; StringQ[x], y_ /; StringQ[y]] :=
  If[x == "", True, If[! StringFreeQ[y, x] && StringTake[y, StringLength[x]] == x, True, False]]
End[]

Begin["`PureDefinition`"]
PureDefinition[x_, t___] :=
  Module[{b, c, d, h = ToString[x] <> " /: Default[", a = If[UnevaluatedQ[Definition2, x], $Failed, Definition2[x]]},
    If[a === $Failed, Return[$Failed]];  b = a[[1 ;; −2]];
    c = If[SuffPref[b[[−1]], Map3[StringJoin, "Options[" <> ToString[x] <> "]", {" = ", " := "}], 1], b[[1 ;; −2]], b];
    If[{t} ≠ {} && ! HowAct[t], d = MinusList[a, c];  t = Join[If[Length[d] > 1, d, Flatten[d]], Select[a, SuffPref[#, h, 1] &]]];
    c = Select[c, ! SuffPref[#, h, 1] &];
    If[Length[c] == 1, c[[1]], c]]
End[]

Begin["`MathPackages`"]
MathPackages[h___] :=
  Module[{c = $InstallationDirectory, b, a = "$Kr20Art27$", d}, d = Run["Dir " <> StandPath[c] <> "/A/B/O/S > $Kr20Art27$"];
    If[d ≠ 0, $Failed, d = ReadList[a, String]; DeleteFile[a];
      b = Map[If[! DirectoryQ[#] && FileExtension[#] == "m", FileBaseName[#], "Null"] &, d];
      b = Select[b, # ≠ "Null" &];
      b = MinusList[DeleteDuplicates[b], {"init", "PacletInfo"}];
      If[{h} ≠ {} && ! HowAct[h], h = {$Version, $LicenseType,
          StringJoin[StringSplit[StringReplace[DateString[$LicenseExpirationDate], " " → "*"], "*"][[1 ;; −2]]]}];
    Sort[b]]
End[]

Begin["`SystemPackages`"]
SystemPackages[y___] := Module[{a, b}, a = FileNames["*.m", $InstallationDirectory, Infinity];
    b = Quiet[DeleteDuplicates[Map[{FileBaseName[#], ContextMfile[#]} &, a]]];  b = Select[b, # ≠ {} &];
    If[{y} ≠ {} && ! HowAct[y], y = Select[Map[If[SameQ[#[[2]], $Failed], #[[1]] &, b], ! SameQ[#, Null] &]];
    Select[b, ! SameQ[#[[2]], $Failed] &]]
End[]

Begin["`RedundantLocals`"]
RedundantLocals[x_ /; BlockFuncModQ[x]] := Module[{a, b, c, p, g, k = 1, j, v, t = {}, z = ""}, {a, b} = {PureDefinition[x], Locals1[x]};
    If[StringQ[a], If[b == {}, True, p = Map[#[[1]] &, StringPosition[a, {"} = ", "} := "}]];
        p = Select[p, ! MemberQ[{"{\"}", " \"}"}, StringTake[a, {# − 2, #}]] &];
        c = Map[Map3[StringJoin, #, {" := ", " = "}] &, b];
        g = Select[b, StringFreeQ[a, Map3[StringJoin, #, {" := ", " = "}]] &];
        While[k ≤ Length[p], v = p[[k]];
          For[j = v, j ≥ 1, j −−, z = StringTake[a, {j, j}] <> z;
            If[! SameQ[Quiet[ToExpression[z]], $Failed], AppendTo[t, z]]];
          z = ""; k++];
```

```
        t = MinusList[g, Flatten[Map[StrToList, t]]];
        If[t == {}, t, p = Select[Map[" " <> # <> "[" &, t], ! StringFreeQ[a, #] &];
          g = {};
          For[k = 1, k ≤ Length[p], k++, v = StringPosition[a, p[[k]]];
            v = Map[#[[2]] &, v];
            z = StringTake[p[[k]], {2, −2}];
            c = 1;
            For[j = c, j ≤ Length[v], j++,
              For[b = v[[j]], b ≤ StringLength[a], b++, z = z <> StringTake[a, {b, b}];
                If[! SameQ[Quiet[ToExpression[z]], $Failed], AppendTo[g, z]; c = j + 1; z = StringTake[p[[k]], {2, −2}]; Break[]]]]];
          MinusList[t, Map[HeadName[#] &, Select[g, HeadingQ1[#] &]]]],
      "Object <" <> ToString[x] <> "> has multiple definitions"]]
End[]

Begin["`RedundantLocalsM`"]
RedundantLocalsM[x_ /; BlockFuncModQ[x]] :=
  Module[{b = {}, d, k = 1, a = Flatten[{PureDefinition[x]}], c = ToString[Unique["g"]]}, While[k ≤ Length[a], d = c <> ToString[x];
      ToExpression[c <> a[[k]]]; AppendTo[b, If[QFunction[d], "Function", RedundantLocals[d]]];
      ToExpression["ClearAll[" <> d <> "]"]; k++]; Remove[c]; If[Length[b] == 1, b[[1]], b]]
End[]

Begin["`SuffixQ`"]
SuffixQ[x_ /; StringQ[x], y_ /; StringQ[y]] :=
  If[x == "", True, If[! StringFreeQ[y, x] && StringTake[y, −StringLength[x]] == x, True, False]]
End[]

Begin["`SubsString`"]
SubsString[s_ /; StringQ[s], y_ /; ListQ[y], pf___] := Module[{a = "", b, c, k = 1},
    If[Set[c, Length[y]] < 2, s, b = Map[ToString1, y]; While[k ≤ c − 1, a = a <> b[[k]] <> "~~ Shortest[__] ~~ "; k++];
      a = a <> b[[−1]]; b = StringCases[s, ToExpression[a]];
      If[{pf} ≠ {} && PureFuncQ[pf], Select[b, pf],
        If[{pf} ≠ {}, Map[StringTake[#, {StringLength[y[[1]]] + 1, −StringLength[y[[−1]]] − 1}] &, b], Select[b, StringQ[#] &]]]]
End[]

Begin["`SubsString1`"]
SubsString1[s_ /; StringQ[s], y_ /; ListQ[y], pf_ /; IntegerQ[pf] || PureFuncQ[pf], t_ /; MemberQ[{0, 1}, t], r___] :=
  Module[{c, h, a = "", b = Map[ToString1, y], d = s, k = 1}, If[Set[c, Length[y]] < 2, s, If[{r} ≠ {}, b = Map[StringReverse, Reverse[b]];
      d = StringReverse[s]]]; While[k ≤ c − 1, a = a <> b[[k]] <> "~~ Shortest[__] ~~ ";
    k++]; a = a <> b[[−1]];
    h = StringCases[d, ToExpression[a]];
    If[t == 0, h = Map[StringTake[#, {StringLength[b[[1]]] − 1, −StringLength[b[[−1]]] + 1}] &, h]];
    If[PureFuncQ[pf], h = Select[h, pf]]; If[{r} ≠ {}, Reverse[Map[StringReverse, h]], h]]
End[]

Begin["`ReduceAdjacentStr`"]
ReduceAdjacentStr[x_ /; StringQ[x], y_ /; StringQ[y], n_ /; IntegerQ[n], z___] := Module[{a = {}, b = {},
    c = Append[StringPosition[x <> FromCharacterCode[0], y, IgnoreCase → If[{z} ≠ {}, True, False]], {0, 0}], h, k}, If[c == {}, x,
    Do[If[c[[k]][[2]] + 1 == c[[k + 1]][[1]], b = Union[b, {c[[k]], c[[k + 1]]}], b = Union[b, {c[[k]]}]; a = Union[a, {b}]; b = {}],
      {k, 1, Length[c] − 1}]; a = Select[a, Length[#] >= n &];
    a = Map[Quiet[Check[{#[[1]], #[[−1]]}, Nothing]] &, Map[Flatten, Map[#[[−Length[#] + n ;; −1]] &, a]]];
    StringReplacePart[x, "", a]]
End[]

Begin["`SubStrToSymb`"]
SubStrToSymb[x_ /; StringQ[x], n_ /; IntegerQ[n], y_ /; StringQ[y] && y != "", p_ /; MemberQ[{0, 1}, p]] :=
  Module[{a, b = StringLength[x], c, d, k}, If[n ≤ 0 || n ≥ b || StringFreeQ[x, y], $Failed, c = StringTake[x, {n}];
    For[If[p == 0, k = n − 1, k = n + 1], If[p == 0, k ≥ 1, k ≤ b], If[p == 0, k−−, k++],
      If[Set[d, StringTake[x, {k}]] != y, If[p == 0, c = d <> c, c = c <> d], Break[]]];
    If[k < 1 || k > b, $Failed, If[p == 0, c = y <> c, c = c <> y]]]]
End[]

Begin["`StrAllSymbNumQ`"]
StrAllSymbNumQ[x_ /; StringQ[x]] := ! MemberQ[Map[SymbolQ[#] || Quiet[IntegerQ[ToExpression[#]]] &, Characters[x]], False]
End[]

Begin["`DirFD`"]
DirFD[d_ /; DirQ[d]] :=
  Module[{a = "$DirFile$", b = {{}, {}}, c, h, t, p = StandPath[StringReplace[d, "/" → "\\"]]}, If[DirEmptyQ[p], Return[{}], Null];
    c = Run["Dir " <> p <> " /B " <> If[SuffPref[p, "\\", 2], "", "\\"] <> "*.* > " <> a];
```

```
    t = Map[ToString, ReadList[a, String]]; DeleteFile[a];
    Map[{h = d <> "\\" <> #; If[DirectoryQ[h], AppendTo[b[[1]], #], If[FileExistsQ[h], AppendTo[b[[2]], #], Null]]} &, t]; b]
End[]

Begin["`TypeFilesD`"]
TypeFilesD[x_ /; DirQ[x]] := Module[
    {a = "$Art27Kr20$", c, b = StandPath[StringReplace[x, "/" → "\\"]], d = {}, p}, If[DirEmptyQ[x], {}, Run["Dir /S/B/A ", b, " > ", a];
      c = Map[ToString, ReadList[a, String]]; DeleteFile[a];
      Sort[Select[DeleteDuplicates[Map[If[DirectoryQ[#], Null, If[FileExistsQ[#], p = ToLowerCase[ToString[FileExtension[#]]];
                  If[! SameQ[p, ""], p, "undefined"]], Null] &, c]], ! SameQ[#, Null] &]]]
End[]

Begin["`DefFunc3`"]
DefFunc3[x_ /; BlockFuncModQ[x]] := Module[{a = Attributes[x], b}, ClearAllAttributes[x];
    b = StringSplit[StrStr[InputForm[DefFunc[x]]], "\n \n"]; SetAttributes[x, a]; If[Length[b] == 1, b[[1]], b]]
End[]

Begin["`SubsProcs`"]
SubsProcs[x_ /; BlockModQ[x]] :=
  Module[{d, s = {}, g, k = 1, p, h = "", v = 1, R = {}, Res = {}, a = PureDefinition[x], j, m = 1, n = 0, b = {" := Module[{", " := Block[{"},
      c = ProcBody[x]}, For[v, v ≤ 2, v++, If[StringFreeQ[c, b[[v]]], Break[], d = StringPosition[c, b[[v]]]];
      For[k, k ≤ Length[d], k++, j = d[[k]][[2]]; While[m ≠ n, p = StringTake[c, {j, j}];
          If[p == "[", m++;
            h = h <> p, If[p == "]", n++;
              h = h <> p, h = h <> p]];
          j++]; AppendTo[Res, h];
        m = 1;
        n = 0;
        h = ""];
      Res = Map10[StringJoin, If[v == 1, " := Module[", " := Block["], Res]; g = Res;
      {Res, m, n, h} = {{}, 1, 0, "]"};
      For[k = 1, k ≤ Length[d], k++, j = d[[k]][[1]] − 2; While[m ≠ n, p = StringTake[c, {j, j}];
          If[p == "]", m++;
            h = p <> h, If[p == "[", n++;
              h = p <> h, h = p <> h]];
          j−−]; AppendTo[Res, h];
        s = Append[s, j];
        m = 1;
        n = 0;
        h = "]"];
      Res = Map9[StringJoin, Res, g];
      {g, h} = {Res, ""};
      Res = {}; For[k = 1, k ≤ Length[s], k++, For[j = s[[k]], j ≥ 1, j−−, p = StringTake[c, {j, j}];
          If[p == " ", Break[], h = p <> h]];
        AppendTo[Res, h];
        h = ""]; AppendTo[R, Map9[StringJoin, Res, g]];
      {Res, m, n, k, h, s} = {{}, 1, 0, 1, "", {}}];
    R = If[Length[R] == 2, R, Flatten[R]];
    If[Length[R] == 1, R[[1]], R]]
End[]

Begin["`StrNestedMod`"]
StrNestedMod[x_ /; ModuleQ[x], y___] :=
  Module[{a = SubProcs[x][[2]], c, d, g, h = {}, k = 1, p, t, n}, If[Length[a] == 1, Map[Remove, a];
      {ToString[x]},
      c = Map[{d = DeleteDuplicates[Flatten[StringPosition[#, "$"]]], StringTake[#, {1, d[[−1]] − 1}]}[[−1]] &, Map[ToString, a]];
      b = Map[Symbol, c];
      For[k, k ≤ Length[a], k++, AppendTo[h, b[[k]] = {c[[k]], Select[Locals[a[[k]]], MemberQ[c, #] &]}];
      h = Select[h, #[[2]] ≠ {} &]; Map[Remove, a];
      If[{y} ≠ {} && ! HowAct[y], y = c, Null];
      For[k = Length[h], k ≥ 2, k−−,
        If[MemberQ[h[[k − 1]][[2]], h[[k]][[1]]], h[[k − 1]][[2]] = AppendTo[h[[k − 1]][[2]], h[[k]][[−1]]]]]; h[[1]]]]
End[]

Begin["`Bits`"]
Bits[x_, P_ /; IntegerQ[P]] := Module[{a, k}, If[StringQ[x] && StringLength[x] == 1,
      If[1 ≤ P ≤ 8, PadLeft[IntegerDigits[ToCharacterCode[x][[1]], 2], 8][[P]],
        If[P == 0, PadLeft[IntegerDigits[ToCharacterCode[x][[1]], 2], 8], Defer[Bits[x, P]]]],
      If[BinaryListQ[x] && 1 <= Length[Flatten[x]] <= 8, a = Length[x];
```

```
        FromCharacterCode[Sum[x[[k]] * 2 ^ (a - k), {k, 1, a}]], Defer[Bits[x, P]]]]]
End[]

Begin["`ListsAssign`"]
ListsAssign[x_ /; ListQ[x], y_ /; ListQ[y]] :=
  Module[{a = Min[Map[Length, {x, y}]], b, c, d = {}, k = 1}, If[a == 0, Return[x], Off[Set::setraw];
      Off[Set::write]; Off[Set::wrsym]]; While[k ≤ a, {b, c} = {x[[k]], y[[k]]};
      AppendTo[d, b = c]; k++];
    x = {Sequences[d[[1 ;; a]]], Sequences[x[[a + 1 ;; -1]]]}; On[Set::setraw]; On[Set::write]; On[Set::wrsym];
    x]
End[]

Begin["`ObjType`"]
ObjType[x_] :=
  Module[{a, b, c, d = {}, h}, If[ToString1[HeadPF[x]] === StringJoin["HeadPF[", ToString1[x], "]"] || SymbolQ[HeadPF[x]],
      Return[Head[x]], b = Flatten[{Definition1[x]}]; c = Length[b]];
 Do[AppendTo[d, h = StringSplit[b[[k]], " := "]; {h[[1]], If[PrefixQ["Module[{", h[[2]]], Module,
          If[PrefixQ["Block[{", h[[2]]], Block, If[PrefixQ["Function[", h[[2]]], Function,
      If[PrefixQ["DynamicModule[{", h[[2]]], DynamicModule, {Function, Head[ToExpression[h[[2]]]]}]]]]}], {k, c}]; Flatten[d, 1]]
End[]

Begin["`ObjInCurrentNb`"]
ObjInCurrentNb[] :=
  Module[{b, c = {}, d, k, h = {}, a = StringReplace[ToString[NotebookGet[]], {" " → "", "\n" → "", "\r" → "", "\t" → ""}]},
    b = StringCases[a, Shortest["RowBox[{RowBox[{" ~~ __ ~~ ",[,"]];
    b = DeleteDuplicates[Select[b, StringFreeQ[#, {"Cell", "BoxData"}] && StringCount[#, "RowBox"] == 2 &]];
    b = Map[StringReplace[#, {"RowBox[{" → "", ",[," → ""}] &, b];
    Map[If[Set[d, Quiet[Check[Context[#], $Failed]]] === $Failed, Null, AppendTo[c, {#, d}]] &, b];
    c = Map[DeleteDuplicates, Map[Flatten, Gather[c, #1[[2]] == #2[[2]] &]]]; If[c == {}, {}, c = If[NestListQ[c], c, {c}];
      Do[AppendTo[h, Sort[c[[k]], ContextQ[#] &]], {k, 1, Length[c]}]; c = {};
      Do[AppendTo[c, Flatten[{h[[k]][[1]], Select[h[[k]][[2 ;; -1]], ! SameQ[PureDefinition[#], $Failed] &]}]], {k, 1, Length[h]}];
      h = Map[Flatten[{#[[1]], Sort[#[[2 ;; -1]]]}] &, c]; If[Length[h] == 1, Flatten[h], h]]
End[]

Begin["`ObjInCurrentNb1`"]
ObjInCurrentNb1[
    x_ /; MemberQ[Map[#[[1]] &, AllCurrentNb[]], FileNameTake[x]] || FileExistsQ[x] && FileExtension[x] == "nb"] :=
  Module[{a, b, c = {}, d, k, h = {}, f, cnb = NbName[][[1]]}, a = If[x === NbName[][[1]], ToString[NotebookGet[]],
      If[MemberQ[Map[#[[1]] &, AllCurrentNb[]], FileNameTake[x]],
        k = Flatten[Select[AllCurrentNb[], #[[1]] === FileNameTake[x] && #[[2]] != "nb has been not saved" &]]; If[k == {},
          f = If[SuffPref[x, ".nb", 2], x, x <> ".nb"]; f = Directory[] <> "\\" <> f; NotebookSave1[FileNameTake[x], f];
          b = ReadFullFile[f]; {OpenCurrentNb[cnb], b}[[2]], ReadFullFile[k[[2]]]], ReadFullFile[x]]];
    If[a === $Failed, $Failed, a = StringReplace[a, {" " -> "", "\n" -> "", "\r" -> "", "\t" -> "", "\\\\" -> "", "\"" -> ""}];
      b = Join[SubsString[a, {"RowBox[{", ",[,"}], SubsString[a, {"<`", "`>"}]];
      b = DeleteDuplicates[Select[b,
          StringFreeQ[#, {"Cell", "BoxData"}] && StringCount[#, "RowBox"] == 1 || StringFreeQ[#, "`"] &]];
      b = Map[StringReplace[#, {"RowBox[{" -> "", ",[," -> ""}] &, b]; b = Select[b, SymbolQ[#] &];
      If[b == {}, {}, Map[If[Set[d, Quiet[Check[Context[#], $Failed]]] === $Failed, Null, AppendTo[c, {#, d}]] &, b];
        c = Map[DeleteDuplicates, Map[Flatten, Gather[c, #1[[2]] == #2[[2]] &]]];
        c = If[NestListQ[c], c, {c}]; Do[AppendTo[h, Sort[c[[k]], ContextQ[#] &]], {k, 1, Length[c]}]; c = {};
        Do[AppendTo[c, Flatten[{h[[k]][[1]], Select[h[[k]][[2 ;; -1]], ! SameQ[PureDefinition[#], $Failed] &]}]], {k, 1, Length[h]}];
        h = Map[Flatten[{#[[1]], Sort[#[[2 ;; -1]]]}] &, c]; c = {};
        Map[If[Length[#] > 1, AppendTo[c, #], Null] &, h]; If[Length[c] == 1, Flatten[c], c]]]
End[]

Begin["`OpenCurrentNb`"]
OpenCurrentNb[x_ /; StringQ[x] && (SuffPref[x, {".cdf", ".nb"}, 2] || SuffPref[x, "Untitled-", 1]) || x == "Messages"] :=
  Module[{a = Notebooks[], b, c = {}, d, k, h},
    If[FileExistsQ[x] && Select[AllCurrentNb[], #[[1]] == FileNameTake[x] &] == {}, NotebookOpen[x], h = FileNameTake[x];
      b = Flatten[Map[SubsString[#, {"<<", ">>"}, 0] &, Map[ToString, a]]];
      d = Map[Quiet[Check[NotebookFileName[#], "nb has been not saved"]] &, a];
      Do[AppendTo[c, {b[[k]], d[[k]]}], {k, 1, Length[b]}]; b = If[Length[c] == 1, Flatten[c], c]; If[NbName[][[1]] == h, Null,
        If[Select[b, #[[1]] == h &] == {}, $Failed,
          c = Partition[Riffle[Map[#[[1]] &, b], a], 2]; SetSelectedNotebook[Select[c, #[[1]] == h &][[1]][[2]]]];]]
End[]

Begin["`NbDocumentQ`"]
NbDocumentQ[x_, t___] := Module[{a, b = FileNameTake[x], c},
    If[StringQ[x] && (SuffPref[x, {".cdf", ".nb"}, 2] || SuffPref[x, "Untitled-", 1]), If[Select[AllCurrentNb[], #[[1]] == b &] != {},
```

```
        If[{t} ≠ {} && ! HowAct[t], If[NbName[][[1]] == b, t = "current", t = "opened"], Null];
        True, a = AllCurrentNb[]; On[Definition::notfound]; c = Quiet[Check[Get[x], False]]; Off[Definition::notfound];
        If[c[[0]] === Notebook, If[{t} ≠ {} && ! HowAct[t], t = "file", Null]; True, False]], False]]
End[]

Begin["`NbCurrentQ`"]
NbCurrentQ[x_] := If[StringQ[x] && (SuffPref[x, {".cdf", ".nb"}, 2] || SuffPref[x, "Untitled−", 1]) || x == "Messages",
   If[Select[Map[ToString, Notebooks[]], ! StringFreeQ[#, "<<" <> x <> ">>"] &] ≠ {}, True, False], False]
End[]

Begin["`FileCurrentNb`"]
FileCurrentNb[x_ /; StringQ[x] && (SuffPref[x, {".cdf", ".nb"}, 2] || SuffPref[x, "Untitled−", 1]) || x == "Messages"] :=
  Module[{a = Notebooks[], b, c, k}, b = Map[ToString, a];
    Do[If[! StringFreeQ[b[[k]], "<<" <> x <> ">>"], c = Return[Quiet[Check[NotebookDirectory[a[[k]]], "500"]]], c = "90"],
      {k, 1, Length[a]}]; If[DirectoryQ[c], c <> x, If[c == "90", $Failed, x <> " was not saved"]]]
End[]

Begin["`NbFileEvaluate`"]
NbFileEvaluate[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]], y___] :=
  Quiet[Quiet[NotebookEvaluate[NotebookOpen[x], InsertResults → If[{y} ≠ {}, False, True]]]]
End[]

Begin["`MfileEvaluate`"]
MfileEvaluate[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y_ /; MemberQ[{1, 2, 3, 4}, y]] := Module[{a, b, c, d, h, f},
   f[z_] := Select[z, "Attributes[" <> # <> "] = {Temporary}" != Quiet[ToString[Definition[#]]] &];
   a = StringCases[ReadString[x], "BeginPackage[\"" ~~ Shortest[___] ~~ "`\"]"];
   a = Map[Quiet[Check[StringTake[#, {15, −3}], Null]] &, a];
   a = Select[a, Complement[Characters[#], Join[CharacterRange["A", "Z"], CharacterRange["a", "z"], {"`"}]] == {} &];
   If[MemberQ[{{}, {"AladjevProcedures`"}}, a], $Failed,
     Quiet[NotebookEvaluate[NotebookOpen[x, Visible → False]]]; b = a[[1]];
     h = Names[b <> "*"]; c = Sort[DeleteDuplicates[Map[StringReplace[#, b −> ""] &, h]]];
      If[y == 1, f[c], If[y == 2, d = {{}, {}};
         Map[If[Quiet[ToString[Definition[#]]] != "Null", AppendTo[d[[1]], #], AppendTo[d[[2]], #]] &, c];
         If[d[[1]] == {}, Null, c = Map[Quiet[ToExpression[StringReplace[DefToString[#], {b −> "", # <> "`" −> ""}]]] &, d[[1]]]];
         d = Map[StringReplace[#, b −> ""] &, d];
         Map[Sort, Map[f, d]], If[y == 3, b, Map[{Print[StringReplace[#, b −> ""]], ToExpression["?" <> ToString[#]]} &, h]; ]]]]
End[]

Begin["`MfileLoad`"]
MfileLoad[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y___] :=
  Module[{a = ReadString[x], b, c, d, h, p}, b = Flatten[StringPosition[a, "BeginPackage[\""]];
    If[b == {}, $Failed,
      c[z_] := Select[z,
          ! MemberQ[{"Null", "Attributes[" <> # <> "] = {Temporary}", False}, Quiet[Check[ToString[Definition[#]], False]] &];
      d = StringCases[a, "BeginPackage[\"" ~~ Shortest[___] ~~ "`\"]"];
      d = Map[Quiet[Check[StringTake[#, {15, −3}], Null]] &, d];
      d = Select[d, Complement[Characters[#], Join[CharacterRange["A", "Z"], CharacterRange["a", "z"], {"`"}]] == {} &][[1]];
       Quiet[ToExpression[StringReplace[StringTake[a, {b[[1]], −1}], {"(*" −> "", "*)" −> ""}]]];
      b = Map[StringReplace[#, d −> ""] &, Names[d <> "*"]];
      Map[Quiet[ClearAttributes[#, {Protected, ReadProtected}]] &, b]; If[{y} ≠ {} && SameQ[ToString[Definition[y]], "Null"],
        y = {d, Set[p, c[b]], Select[Complement[b, p], "Attributes[" <> # <> "] = {Temporary}" != Quiet[ToString[Definition[#]]] &]};,
        Null]]]
End[]

Begin["`StrDelEnds1`"]
StrDelEnds1[S1_ /; StringQ[S1], h_ /; StringQ[h], p_ /; MemberQ[{1, 2, 3}, p]] :=
  Module[{a, k, s}, If[! SuffPref[S1, h, p], S1, If[p == 3, StringTrim[S1, h ...], {s, a, k} = {S1, StringLength[h], 1};
        While[SuffPref[s, h, p], s = StringTake[s, If[p == 1, {a + 1, −1}, {1, StringLength[s] − a}]]; k ++ 1]; s]]]
End[]

Begin["`StrDelEnds`"]
StrDelEnds[S1_ /; StringQ[S1], h_ /; StringQ[h], p_ /; MemberQ[{1, 2, 3}, p]] :=
  Module[{a, k, s}, If[! SuffPref[S1, h, p], S1, If[p == 3, StringTrim[S1, h ...], {s, a, k} = {S1, StringLength[h], 1};
        While[SuffPref[s, h, p], s = StringTake[s, If[p == 1, {a + 1, −1}, {1, StringLength[s] − a}]]; k ++ 1]; s]]]
End[]

Begin["`StringTrim1`"]
StringTrim1[x_ /; StringQ[x], y_ /; StringQ[y], z_ /; StringQ[z]] :=
  Module[{a = x}, If[y == z == "", a = StringTrim[a], If[SuffPref[a, y, 1], a = StringReplace[a, y → "", 1]];
```

```
        If[SuffPref[a, z, 2], a = StringReplace[a, z → "", StringCount[a, z]]]]; a]
End[]

Begin["`StringTrim2`"]
StringTrim2[x_ /; StringQ[x], y_ /; StringQ[y] || DeleteDuplicates[Map[StringQ, Flatten[{y}]]] == {True},
    z_ /; MemberQ[{1, 2, 3}, z]] := Module[{a = x, b = Flatten[{y}], m = 0, n = 0}, Do[If[z == 1,
        If[SuffPref[a, b, 1], a = StringDrop[a, 1]; m = 1, m = 0],
         If[z == 2, If[SuffPref[a, b, 2], a = StringDrop[a, −1]; n = 1, n = 0],
          If[SuffPref[a, b, 1], a = StringDrop[a, 1]; m = 1, m = 0]; If[SuffPref[a, b, 2], a = StringDrop[a, −1]; n = 1, n = 0]]];
       If[m + n == 0, Return[a], Null], Infinity]]
End[]

Begin["`StringReplaceS`"]
StringReplaceS[S_ /; StringQ[S], s1_ /; StringQ[s1], s2_ /; StringQ[s2]] := Module[
    {a = StringLength[S], b = StringPosition[S, s1], c = {}, k = 1, p, L = Characters["`!@#%^&*(){}:\"\\/|<>?~−=+[];'., 1234567890"],
      R = Characters["`!@#%^&*(){}:\"\\/|<>?~=+[];'., "]}, If[b == {}, S, While[k ≤ Length[b], p = b[[k]];
        If[Quiet[(p[[1]] == 1 && p[[2]] == a) || (p[[1]] == 1 && MemberQ[R, StringTake[S, {p[[2]] + 1, p[[2]] + 1}]]) || (MemberQ[L,
                    StringTake[S, {p[[1]] − 1, p[[1]] − 1}]] && MemberQ[R, StringTake[S, {p[[2]] + 1, p[[2]] + 1}]]) || (p[[2]] == a &&
                  MemberQ[L, StringTake[S, {p[[1]] − 1, p[[1]] − 1}]])], c = Append[c, p]; k++]; StringReplacePart[S, s2, c]]]
End[]

Begin["`StringReplacePart1`"]
StringReplacePart1[x_ /; StringQ[x], y_ /; StringQ[y], z_ /; StringQ[z],
    n_ /; IntegerQ[n] || ListQ[n] && DeleteDuplicates[Map[IntegerQ[#] &, n]] == {True}] :=
  Module[{a = StringPosition[x, y]}, If[a == {}, x,
      StringReplacePart[x, z, Map[Quiet[Check[a[[#]], Nothing]] &, Flatten[{n}]]]]]
End[]

Begin["`Range1`"]
Range1[x_, y_] := Module[{a, b, c, h},
    {a, b, c} = {Characters[ToString[x]], Characters[ToString[y]], Join[CharacterRange["a", "z"], CharacterRange["A", "Z"], {"$", "_"}]};
    h[z_] := Module[{t = Length[z], n, m, d}, For[t, t ≥ 1, t−−, d = z[[t]];
          If[! MemberQ[c, d], Next[], n = StringJoin[z[[1 ;; t]]]; m = StringJoin[z[[t + 1 ;; −1]]]; Break[]]]; {n, m}];
    a = Flatten[{h[a], h[b]}];
    If[a[[1]] ≠ a[[3]] || ! HowAct[a[[1]]] || ! HowAct[a[[3]]] || a[[2]] === "" || a[[4]] === "" || ToExpression[a[[2]] <> ">" <> a[[4]]],
      Return[Defer[Range1[x, y]]], b = Range[ToExpression[a[[2]]], ToExpression[a[[4]]]]];
    ToExpression[Map3[StringJoin, a[[1]], Map[ToString, b]]]]
End[]

Begin["`Range2`"]
Range2[x_, y_ /; IntegerQ[y] /; y ≥ 1] :=
  Module[{a = {}, b = Range[1, y], k = 1}, For[k, k ≤ Length[b], k++, a = Append[a, ToString[x] <> ToString[b[[k]]]]];
    ToExpression[a]]
End[]

Begin["`CallListable`"]
CallListable[x_ /; SystemQ[x] || BlockFuncModQ[x], y___] :=
  Module[{a = Attributes[x]}, If[MemberQ[a, Listable], x[Flatten[{y}]], SetAttributes[x, Listable];
      {x[Flatten[{y}]], ClearAttributes[x, Listable]}[[1]]]]
End[]

Begin["`SysFuncQ`"]
SysFuncQ[x_] := If[UnevaluatedQ[Definition2, x], False, If[SameQ[Definition2[x][[1]], "System"], True, False]]
End[]

Begin["`SysFuncQ1`"]
SysFuncQ1[x_] := MemberQ[Names["System`*"], ToString[x]]
End[]

Begin["`Clear1`"]
Clear1[x_ /; MemberQ[{1, 2}, x], y___] := Module[{a = {y}, b, c, d, k = 1}, If[y === {}, Null, For[k, k ≤ Length[a], k++, b = a[[k]];
        d = Quiet[ToExpression["Attributes[" <> ToString1[b] <> "]"]];
        ToExpression["Quiet[ClearAttributes[" <> ToString1[b] <>
            ", " <> ToString[d] <> "]" <> "; Clear" <> If[x == 1, "", "All"] <> "[" <> ToString1[b] <> "]]"]];
      If[x == 2, Null, Quiet[Check[ToExpression["SetAttributes[" <> ToString1[b] <> ", " <> ToString[d] <> "]"], $Failed]]]]]
End[]

Begin["`Range3`"]
Range3[x_, y_ /; IntegerQ[y] /; y ≥ 1] :=
```

```
  Module[{a = {}, b = Range[1, y], k = 1}, For[k, k ≤ Length[b], k++, a = Append[a, ToString[x] <> ToString[b[[k]]] <> "_"]];
    ToExpression[a]]
End[]

Begin["`Range4`"]
Range4[x_, y___] := Module[{a = Select[Flatten[{x, If[{y} ≠ {}, {y}, Null]}], # ≠ "Null" &], b},
    If[SameQ[DeleteDuplicates[Map[NumberQ, a]], {True}], Range[Sequences[a]], b = Map[FromCharacterCode, Range[32, 4096]];
      If[Length[{x, y}] == 2 && MemberQ3[b, {x, y}], CharacterRange[x, y],
        If[Length[{x, y}] == 1 && StringQ[x], Select[b, ToCharacterCode[#][[1]] ≤ ToCharacterCode[x][[1]] &], $Failed]]]]
End[]

Begin["`Range5`"]
Range5[x__] := Flatten[Map[If[Head[#] === Span, Range[Part[#, 1], Part[#, 2]], #] &, Flatten[{x}]]]
End[]

Begin["`Gather1`"]
Gather1[L_ /; ListListQ[L], n_ /; IntegerQ[n]] := Module[{a = {}, b = {}, c, k},
    If[! (1 ≤ n && n ≤ Length[L[[1]]]), Return[Defer[Gather1[L, n]]], Do[a = Append[a, L[[k]][[n]]], {k, 1, Length[L]}]];
    a = Map[List, DeleteDuplicates[a]];  For[k = 1, k ≤ Length[a], k++, a[[k]] = Select[L, #[[n]] == a[[k]][[1]] &]];
    a]
End[]

Begin["`Gather2`"]
Gather2[x_ /; ListQ[x]] := Module[{a = Select[Gather[Flatten[x]], Length[#] > 1 &], b = {}},
    If[a == {}, Return[{}], Do[b = Append[b, {a[[k]][[1]], Length[a[[k]]]}], {k, Length[a]}]];
    If[Length[b] > 1, b, First[b]]]
End[]

Begin["`SubStr`"]
SubStr[S_ /; StringQ[S], p_ /; IntegerQ[p], a_ /; CharacterQ[a] || ListQ[a] && DeleteDuplicates[Map[CharacterQ, a]] == {True},
    b_ /; CharacterQ[b] || ListQ[b] && DeleteDuplicates[Map[CharacterQ, b]] == {True},
    R_ /; ! HowAct[R]] := Module[{c = Quiet[StringTake[S, {p, p}]], k, t},
    If[p ≥ 1 && p ≤ StringLength[S], For[k = p + 1, k ≤ StringLength[S], k++, t = StringTake[S, {k, k}];
        If[If[CharacterQ[b], t ≠ b, ! MemberQ[b, t]], c = c <> t; Continue[], Break[]]];
      For[k = p − 1, k ≥ 1, k−−, t = StringTake[S, {k, k}];
        If[If[CharacterQ[a], t ≠ a, ! MemberQ[a, t]], c = t <> c;  Continue[], Break[]]];  c,
      R = "Argument p should be in range 1.." <> ToString[StringLength[S]] <> " but received " <> ToString[p];
      $Failed]]
End[]

Begin["`Tuples1`"]
Tuples1[x_ /; ListQ[x], n_ /; IntegerQ[n]] :=
  Module[{c = "", d = "", f = Res, h = {}, t, t$$, j = 1, a = Map[ToString1, Map[ToString, x]], m = Length[x], b = ""}, Res = {};
    For[j = 1, j ≤ n, j++, h = Append[h, "t$$" <> ToString[j]];
      b = b <> "For[t$$" <> ToString[j] <> "=1, t$$" <> ToString[j] <> " <= " <> ToString[m] <> ", t$$" <> ToString[j] <> "++, ";
      d = d <> "]"];
    For[j = 1, j ≤ n, j++, c = c <> ToString[a] <> "[[t$$" <> ToString[j] <> "]]" <> ","];
    c = "Res = Append[Res, ToString1[StringJoin[{" <> StringTake[c, {1, −2}] <> "}]]]";
    ToExpression[b <> c <> d]; {Res, Res = f}[[1]]]
End[]

Begin["`AssignToList`"]
AssignToList[y_ /; ListQ[y], z_ /; ! HowAct[z], n_ /; IntegerQ[n] && n ≥ 1] :=
  Module[{a, k = 1}, If[Length[y] < n, Return[Defer[AssignToList[y, z, n]]], a = Range2[z, n];
      For[k, k ≤ n, k++, ToExpression[ToString[a[[k]]] <> " = " <> ToString[y[[k]]]]]]; a]
End[]

Begin["`BlockToModule`"]
BlockToModule[x_ /; SymbolQ[x]] := Module[{a = Definition2[x], b, c, d, h = {}, k = 1, n, m},
    If[ListQ[a] && a[[1]] == "System" || UnevaluatedQ[Definition2, x], $Failed, b = a[[−1]];
      ClearAllAttributes[x]; c = a[[1 ;; −2]]; d = Flatten[{HeadPF[x]}];
      For[k, k ≤ Length[d], k++, {n, m} = {c[[k]], d[[k]]};
        If[SuffPref[n, {m <> " := Block[{", m <> " = Block[{"}, 1],
          AppendTo[h, StringReplace[n, "Block[{" −> "Module[{", 1]], AppendTo[h, n]]];
      Map[ToExpression, h];
      SetAttributes[x, b]]]
End[]

Begin["`BlockFuncModQ`"]
```

```
BlockFuncModQ[x_, y___] :=
  Module[{b, c, a = Flatten[{PureDefinition[x]}][[1]]}, If[MemberQ[{$Failed, "System"}, a], False, b = StringSplit[a, {" := ", " = "}, 2];
    If[StringFreeQ[b[[1]], "["], False, c = If[SuffPref[b[[2]], "Module[{", 1], "Module",
        If[SuffPref[b[[2]], "Block[{", 1], "Block", "Function"]]; If[{y} ≠ {} && ! HowAct[y], y = c];
      True]]]
End[]

Begin["`BlockModQ`"]
BlockModQ[x_, y___] := Module[{a = Flatten[{PureDefinition[x]}][[1]], b, c, s = FromCharacterCode[6]},
  If[MemberQ[{$Failed, "System"}, a], False, b = StringReplace[a, {" := " → s, " = " → s}, 1];
    b = StringTake[b, {Flatten[StringPosition[b, s]][[1]] + 1, -1}];
    c = If[SuffPref[b, "Module[{", 1], "Module", If[SuffPref[b, "Block[{", 1], "Block"]];
    If[{y} ≠ {} && ! HowAct[y], y = c]; If[c === Null, False, True]]]
End[]

Begin["`ActBFM`"]
ActBFM[] := Select[Sort[Names["Global`*"]], ! TemporaryQ[#] && BlockFuncModQ[#] &]
End[]

Begin["`DelEl`"]
DelEl[L_ /; ListQ[L], x_, N_ /; MemberQ[{1, 2, 3}, N]] :=
  Module[{a, b = Length[L], k, Art}, Art[Z_, y_, p_ /; MemberQ[{1, 2}, p]] := Module[{b = Length[Z], k},
      If[b == 0, Defer[DelEl[Z, y, p]], If[p == 1, For[k = 1, k ≤ b, k++, If[Z[[k]] === y, Null, Return[Take[Z, {k, b}]]]],
          For[k = b, 1 ≤ k, k--, If[Z[[k]] === y, Null, Return[Take[Z, {1, k}]]]], Null]]];
    If[N == 1 || N == 2, Art[L, x, N], Art[Art[L, x, 1], x, 2]]]
End[]

Begin["`BootDrive`"]
BootDrive[] := Module[{a = Select[Adrive[], FileExistsQ[# <> ":\\" <> "Boot.ini"] &], b, c, d = "", h = {}, t},
  If[a == {}, Return[$Failed], b = a[[1]] <> ":\\" <> "Boot.ini"]; Label[avz]; c = Read[b, String];
  If[c === EndOfFile, Close[b]; Return[{a[[1]], If[Length[h] == 1, h[[1]], h]}],
    If[SuffPref[c, "default=", 1], d = "\\" <> StrDelEnds[StringTake[c, {StringPosition[c, "\\"][[1]][[1]] + 1, -1}], " ", 2] <> "=";
      Goto[avz], If[d == "" || StringFreeQ[c, {d, "\\WINDOWS="}], Goto[avz],
        t = Flatten[StringPosition[c, DeleteDuplicates[{d, "\\WINDOWS=", "/"}]]];
        t = ToExpression[StringTake[c, {t[[2]] + 1, If[Length[t] == 2, 0, t[[3]] - 1}]]; h = Append[h, t]; Goto[avz]]]]]
End[]

Begin["`BootDrive1`"]
BootDrive1[] := Mapp[Part, GetEnvironment[{"SystemDrive", "SystemRoot", "OS"}], 2]
End[]

Begin["`FindSubDir`"]
FindSubDir[x_ /; StringQ[x], y___] :=
  Module[{b = {}, c = "", k = 1, p, t, a = If[{y} == {}, Adrive[], {y}], f = "Art27Kr20.txt", h = ToLowerCase[x]},
    While[k ≤ Length[a], Run["Dir ", a[[k]] <> ":\\", " /B/S/L > " <> f];
      While[! SameQ[c, "EndOfFile"], c = ToString[Read[f, String]]; t = FileNameSplit[c];
        p = Flatten[Position[t, h]];
        If[p ≠ {} && DirectoryQ[FileNameJoin[t[[1 ;; p[[1]]]]]], AppendTo[b, c]]; Continue[]];
      Closes[f];
      c = "";
      k++];
    {DeleteFile[f], b}[[2]]]
End[]

Begin["`$TestArgsTypes`"]
$TestArgsTypes = 50 090
End[]

Begin["`OpSys`"]
OpSys[] := Module[{a = ToString[Unique["s"]], b}, Run["SystemInfo >" <> a];
    b = StringTrim[StringTake[ReadList[a, String][[2]], {9, -1}]];
    DeleteFile[a]; b]
End[]

Begin["`TestArgsTypes`"]
TestArgsTypes[P_ /; ModuleQ[P] || BlockQ[P] || QFunction[P], y_] :=
  Module[{c, d = {}, h, k = 1, a = Map[ToString, Args[P]], b = ToString[InputForm[y]]}, ClearAll["$TestArgsTypes"];
    If[! SuffPref[b, ToString[P] <> "[", 1], Return[y],
      c = Map[ToString1, ToExpression["{" <> StringTake[b, {StringLength[ToString[P]] + 2, -2}] <> "}"]]];
```

```
    If[Length[a] ≠ Length[c], $TestArgsTypes = "Quantities of formal and factual arguments are different";
      $Failed, For[k, k ≤ Length[a], k++, d = Append[d, ToExpression["{" <> c[[k]] <> "}" <> " /. " <> a[[k]] <> " –> True"]]];
      d = Map[If[ListQ[#], #[[1]], #] &, d]; h = Flatten[Map3[Position, d, Cases[d, Except[True]]]];
      h = Map[{#, If[ListQ[d[[#]]], Flatten[d[[#]], 1], d[[#]]]} &, h];
      $TestArgsTypes = If[Length[h] == 1, h[[1]], h]; $Failed]]
End[]

Begin["`TestArgsTypes1`"]
TestArgsTypes1[P_ /; ModuleQ[P] || BlockQ[P] || QFunction[P], y_] :=
  Module[{c, d = {}, h, k = 1, n, p, w, w1, a = Quiet[ArgsTypes[P]], g = Map[ToString1, Args[P]], b = ToString[InputForm[y]]},
    a = Map[{#[[1]], StringReplace[#[[2]], "\\\\" → ""]} &, a];
    ClearAll["$TestArgsTypes", "$$Art$Kr$$"];
    If[! SuffPref[b, ToString[P] <> "[", 1], Return[y],
      c = Map[ToString1, ToExpression["{" <> StringTake[b, {StringLength[ToString[P]] + 2, –2}] <> "}"]]];
    If[Length[a] ≠ Length[c], Return[$TestArgsTypes = "Quantities of formal and factual arguments are different";
        $Failed], w = Map[StringTake[#, {1, StringPosition[#, "_"][[1]][[1]] – 1}] &, g];
      w1 = Map[ToString, Unique[w]]; While[k ≤ Length[w], ToExpression[w1[[k]] <> " = " <> w[[k]]];
        k++];
      Map[ClearAll, w]; For[k = 1, k ≤ Length[a], k++, p = a[[k]];
        If[p[[2]] === "Arbitrary", d = Append[d, True],
          If[StringFreeQ[g[[k]], " /; "], If[ToExpression["Head[" <> c[[k]] <> "] === " <> p[[2]]],
              d = Append[d, True], d = Append[d, False]], $$Art$Kr$$ = ToExpression[p[[1]]];
            n = ToExpression[{p[[1]] <> " = " <> c[[k]], p[[2]]}]; ToExpression[p[[1]] <> " = " <> "$$Art$Kr$$"];
            If[n[[–1]], d = Append[d, True], d = Append[d, False]]]]];
    h = DeleteDuplicates[Flatten[Map3[Position, d, Cases[d, Except[True]]]]];
    h = Map[{#, If[ListQ[c[[#]]], Flatten[c[[#]], 1], c[[#]]]} &, h]; $TestArgsTypes = If[Length[h] == 1, h[[1]], h];
    k = 1;
    While[k ≤ Length[w], ToExpression[w[[k]] <> " = " <> w1[[k]]]; k++]; Remove["$$Art$Kr$$"]; $Failed]
End[]

Begin["`TestArgsTypes2`"]
TestArgsTypes2[P_ /; ModuleQ[P] || BlockQ[P] || QFunction[P], y__] :=
  Module[{a = Quiet[ArgsTypes[P]], b = Map[ToString1, {y}], c = {y}, d = {}, k = 1, p},
    If[Length[c] ≠ Length[a], "Quantities of formal and factual arguments are different",
      For[k, k ≤ Length[c], k++, p = a[[k]];
        AppendTo[d, If[p[[2]] === "Arbitrary", True, If[SymbolQ[p[[2]]], ToString[Head[c[[k]]]] === p[[2]],
            ToExpression[StringReplace[p[[2]], {"[" <> p[[1]] <> "]" → "[" <> b[[k]] <> "]",
                " " <> p[[1]] <> " " → " " <> b[[k]] <> " ", " " <> p[[1]] <> "]" → " " <> b[[k]] <> "]"}]]]]];
      If[MemberQ[d, False], Partition[Riffle[{y}, d], 2], {True, P[y]}]]]
End[]

Begin["`TrueCallQ`"]
TrueCallQ[x_ /; BlockFuncModQ[x], y__] := Quiet[Check[If[UnevaluatedQ[x, y], False, x[y]; True], False]]
End[]

Begin["`PrevNextVar`"]
PrevNextVar[x_ /; SymbolQ[x], t_ /; IntegerQ[t], y___] := Module[{a = ToString[x], b, c = {}, d, k, n},
    b = Characters[a];
    n = Length[b];
    For[k = n, k ≥ 1, k––, If[IntegerQ[ToExpression[b[[k]]]], AppendTo[c, b[[k]]], d = StringJoin[b[[1 ;; k]]]; Break[]]];
    k = ToExpression[c = StringJoin[Reverse[c]]]; If[SameQ[k, Null] || {y} == {} && k – t ≤ 0, x,
      If[c == "", x, ToExpression[d <> ToString[k + If[{y} ≠ {}, t, –t]]]]]]
End[]

Begin["`ContextMfile`"]
ContextMfile[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] :=
  Module[{b, a = ReadFullFile[x], c}, b = SubsString[a, {"BeginPackage[\"", "\"]"}]; c = If[b ≠ {}, StringTake[b, {14, –2}]];
    If[b === {}, $Failed, c = Flatten[StringSplit[c, ","]];
      c = Select[Quiet[ToExpression[c]], ContextQ[#] &]; If[Length[c] > 1, c, c[[1]]]]]
End[]

Begin["`ContextMfile1`"]
ContextMfile1[x_ /; MemberQ[{"m", "tr"}, FileExtension[x]]] :=
  Module[{a = ReadFullFile1[If[FileExistsQ[x], x, Flatten[{FindFile1[x]}][[1]]]], b = "BeginPackage[", c, d},
    If[a === {}, {}, c = StringPosition[a, b]; If[c == {}, {}, d = SubStrToSymb[StringTake[a, {Flatten[c][[2]], –1}], 1, "]", 1];
        d = StringReplace[StringTake[d, {2, –2}], {"{" –> "", "}" –> ""}];
        d = Map[ToExpression, StrToList[d]]; If[Length[d] == 1, d[[1]], d]]]
End[]
```

```
Begin["`ContextFromMx`"]
ContextFromMx[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a = StringJoin[Select[Characters[ReadString[x]], SymbolQ[#] || IntegerQ[#] || # == "`" &]],
      b, d = Map[FromCharacterCode, Range[2, 27]]},
    If[StringFreeQ2[a, {"CONT", "ENDCONT"}], $Failed, b = StringCases1[a, {"CONT", "ENDCONT"}, "___"];
      If[b == {}, $Failed, StringReplace[b, Flatten[{GenRules[d, ""] ,  "ENDCONT" -> "", "CONT" -> ""}]][[1]]]]]
End[]

Begin["`ContextsInFiles`"]
ContextsInFiles[x_ /; FileExistsQ[x] && MemberQ[{"m", "mx", "cdf", "nb"}, FileExtension[x]]] :=
  Module[{a = StringReplace[ReadFullFile[x], {" " → "", "\n" → "", "\t" -> ""}], b, b1, d = Flatten[{Range[65, 90], Range[96, 122]}]},
    If[FileExtension[x] == "m",  b = StringCases[a, Shortest["BeginPackage[\"" ~~ __ ~~ "`\"]"]];
      If[b === {}, b = $Failed, b = Map[StringTake[#, {15, -3}] &, b][[1]];
        b1 = StringCases[a,  Shortest["BeginPackage[\"" ~~ __ ~~ "`\"}]"]]; If[b1 === {}, b1 = $Failed,
          If[! StringFreeQ[b1[[1]], {"{", "}"}], b1 = StringReplace[b1[[1]], {"{" -> "", "}" -> "", "\"" -> ""}];
            b1 = StringSplit[StringTake[b1, {14, -2}], ","], b1 = $Failed]]];
      b = Flatten[{b, b1}];
      SetAttributes[ContextQ, Listable];  b = Select[b, DeleteDuplicates[Flatten[ContextQ[{#}]]] == {True} &];
      If[ListQ[b] && Length[b] == 1, b[[1]], b],
      If[FileExtension[x] == "mx", Quiet[Check[b = StringCases[a, Shortest["CONT" ~~ __ ~~ "ENDCONT"]][[1]];
            b = Flatten[Map[ToCharacterCode, Characters[StringReplace[b, {"CONT" → "", "ENDCONT" → ""}]]]];
            StringJoin[Map[FromCharacterCode[#] &, Select[b, MemberQ[d, #] &]]], $Failed]],
        Quiet[Check[a = StringCases[a, Shortest["BeginPackage\",\"[\",\"\\\"\\<" ~~ __ ~~ "`\\>\\\"\",\"]\"}]"]][[1]];
            a = StringReplace[a, "BeginPackage" → ""];  b = Flatten[Map[ToCharacterCode, Characters[a]]];
            StringJoin[Map[FromCharacterCode[#] &, Select[b, MemberQ[d, #] &]]], $Failed]]]]
End[]

Begin["`ContextInMfile`"]
ContextInMfile[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] :=
  Module[{a = StringReplace[ReadFullFile[x], {" " → "", "\n" → "", "\t" -> "", "\r" -> ""}],
      b, c, d = Flatten[{Range[65, 90], Range[96, 123], {44, 125}}]},
    b = Flatten[Map[SubsString[a, {#, "\"]"}] &, {"Begin[\"", "Package[\"", "Get[\"", "Needs[\""}]];
    c = Flatten[Map[SubsString[a, {#, "\"}]"}] &, {"Begin[\"", "Package[\"", "Get[\"", "Needs[\""}]];
    b = Map[StringReplace[#, {"\"" -> "", "{" -> "", "}" -> ""}] &, Map[StringReplace[StringTake[#, {1, -2}],
          {"Begin[\"" -> "", "Get[\"" -> "", "Needs[\"" -> "", "Package[\"" -> ""}, 1] &, Join[b, c]]];
    DeleteDuplicates[Flatten[Map[StringSplit[#, ","] &, Select[b,
          Complement[DeleteDuplicates[ToCharacterCode[#]], d] == {} && ! SuffPref[#, "`", 1] &]]]]
End[]

Begin["`ContextFromMx1`"]
ContextFromMx1[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a = StringJoin[Select[Characters[ReadString[x]], SymbolQ[#] || IntegerQ[#] || # == "`" &]],
      b, d = Map[FromCharacterCode, Range[2, 27]]}, If[StringFreeQ2[a, {"CONT", "ENDCONT"}] ||
        StringCases1[a, {"CONT", "ENDCONT"}, "___"] == {}, $Failed, b = StringPosition[a, {"CONT", "ENDCONT"}];
      If[b == {}, $Failed, StringReplace[StringTake[a, {b[[1]][[1]], b[[2]][[2]]}],
          Flatten[{GenRules[d, ""] ,  "ENDCONT" -> "", "CONT" -> ""}]]]]
End[]

Begin["`ContextNBfile`"]
ContextNBfile[x_ /; FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x]]] :=
  Module[{a = ""}, While[! SameQ[a , EndOfFile], a =
Read[x, String]; If[! StringFreeQ[a, "BeginPackage"],
a = Quiet[ToExpression[ToExpression[StringSplit[a, ","][[3]]]]; Break[]; Continue[]];
    Close[x]; If[! ContextQ[a] || SameQ[a , EndOfFile], $Failed, a]]
End[]

Begin["`ContextMXfile`"]
ContextMXfile[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a , b = Flatten[Map7[Range, Sequences, {{48, 57}, {65, 90}, {96, 122}}]], c}, a = BinaryReadList[x];
    a = a[[1 ;; If[Length[a] ≥ 500, 500, Length[a]]]];
  c = Flatten[Map3[PosSubList, a, {{67, 79, 78, 84}, {69, 78, 68, 67, 79, 78, 84}}]];
    If[Length[c] < 5, $Failed, FromCharacterCode[Select[a[[c[[2]] + 1 ;; c[[5]] - 1]], MemberQ[b, #1] & ]]]
End[]

Begin["`ContextInMxFile`"]
ContextInMxFile[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] := Module[
    {a = StringReplace[ReadFullFile[x], {" " → "", "\n" → "", "\t" -> "", "\r" -> ""}], b = Flatten[{Range[65, 90], Range[96, 122]}]},
    If[StringCount[SubsString[a, {"Mathematica", "ENDCONT"}]][[1]], "CONT"] == 1, $Failed,
      FromCharacterCode[
```

```
        Select[ToCharacterCode[Sort[DeleteDuplicates[Map[StringReplace[#, {"ENDCONT" -> "", "CONT" -> ""}] &,
                SubsString[a, {"CONT", "ENDCONT"}]]]][[1]], MemberQ[b, #] &]]]
End[]

Begin["`ContextFromFile`"]
ContextFromFile[x_ /; StringQ[x]] :=
  If[Quiet[FileExistsQ[x] ] && MemberQ[{"cdf", "m", "mx", "nb"}, FileExtension[x]], Quiet[ToExpression[
      StringJoin["Context", ToUpperCase[If[FileExtension[x] == "cdf", "nb", FileExtension[x]]], "file[", ToString1[x],
   "]"]]], $Failed]
End[]

Begin["`ToContextPath`"]
ToContextPath[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] :=
  Module[{a = ReadFullFile[x], b = "BeginPackage[", c}, c = StrSymbParity[a, b, "[", "]"];
    c = Map[If[! StringFreeQ[#, {"`\"]", "`\"}]"}], StringTake[#, {14, -2}]] &, c];
    c = ToExpression[Flatten[Map[StringSplit[#, ","] &, c]]];
    c = DeleteDuplicates[Map[If[ListQ[#], #[[1]], #] &, c]]; $ContextPath = DeleteDuplicates[Join[$ContextPath, c]]; $ContextPath]
End[]

Begin["`SymbolsContext`"]
SymbolsContext[x_ /; ContextQ[x]] := Quiet[Select[Map[StringReplace[#, x -> ""] &, Names[x <> "*"]],
      ToString[Definition[#]] ≠ "Null" && Attributes[#] ≠ {Temporary} &]]
End[]

Begin["`DefFromPackage`"]
DefFromPackage[x_ /; SymbolQ[x]] := Module[{a = Context[x], b = "", c = "", d = ToString[x], k = 1, h, p},
    If[MemberQ[{"Global`", "System`"}, a], Return[Attributes[x]], h = a <> d;
      ToExpression["Save[" <> ToString1[d] <> "," <> ToString1[h] <> "]"]];
    For[k, k < Infinity, k++, c = Read[d, String];
      If[c === " ", Break[], b = b <> c]]; p = StringReplace[RedSymbStr[b, " ", " "], h <> "`" → ""];
    {c, k, b} = {"", 1, ""};
    For[k, k < Infinity, k++, c = Read[d, String];
      If[c === " " || c === EndOfFile, Break[], b = b <> If[StringTake[c, {-1, -1}] == "\\", StringTake[c, {1, -2}], c]]];
    DeleteFile[Close[d]]; {p, StringReplace[b, " /: " <> d → ""], Attributes[x]}]
End[]

Begin["`DefFromM`"]
DefFromM[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y_ /; SymbolQ[y], z___] := Module[{a = ReadList[x, String], b, c, d},
    {b, c} = {"(*Begin[\"`" <> ToString[y] <> "`\"]*)", "(*End[]*)"};
    d = StringJoin[Map[StringTake[#, {3, -3}] &, Flatten[SubListsMin[a, b, c, "r", 78]]]];
    If[{z} ≠ {}, ToExpression[d];
      d, d]]
End[]

Begin["`TestProcCalls`"]
TestProcCalls[x_ /; BlockFuncModQ[x], y_ /; ListQ[y]] :=
  Module[{d, p, a = Args[x], b = {}, r, c = "_ /; ", k = 1, v}, a = Map[ToString1, If[NestListQ[a], a[[1]], a]];
    If[Length[a] ≠ Length[y] || MemberQ[Map[! StringFreeQ[#, "__"] &, a], True],
      $Failed, v = If[NestListQ[v = Args[x, 90]], v[[1]], v];
      For[k, k ≤ Length[a], k++, p = a[[k]]; AppendTo[b,
          If[StringTake[p, {-1, -1}] == "_", True, If[! StringFreeQ[p, c], d = StringSplit[p, c];
              r = ToExpression[d[[1]]];
              {ToExpression[{d[[1]] <> "=" <> ToString1[y[[k]]], d[[2]]}][[2]], ToExpression[d[[1]] <> "=" <> ToString[r]]}[[1]],
              d = StringSplit[p, "_"];
              ToString[Head[y[[k]]]] == d[[2]]]]]]; {k, d} = {1, Partition[Riffle[v, b], 2]}; While[k ≤ Length[d], PrependTo[d[[k]], k];
        k++]; d]]
End[]

Begin["`ClearAllAttributes`"]
ClearAllAttributes[x__] := Map[Quiet[ClearAttributes[#, Attributes[#]];] &, {x}][[1]]
End[]

Begin["`ProcFuncCS`"]
ProcFuncCS[] := Quiet[Map3[Select, Names["`*"], {BlockQ[#] &, FunctionQ[#] &, ModuleQ[Symbol[#]] &}]]
End[]

Begin["`FindFile1`"]
FindFile1[x_ /; StringQ[x], y___] :=
  Module[{a = If[{y} ≠ {} && PathToFileQ[y], {y}, Map[# <> ":\\" &, Adrive[]]], c, d = {}, k = 1, b = "\\" <> ToLowerCase[x]},
```

```
    For[k, k ≤ Length[a], k ++,
      c = Map[ToLowerCase, Quiet[FileNames["*", a[[k]], Infinity]]];  d = Join[d, Select[c, SuffPref[#, b, 2] && FileExistsQ[#] &]]];
    If[Length[d] == 1, d[[1]], d]]
End[]

Begin["`FindFile2`"]
FindFile2[x_ /; DirectoryQ[x], y_ /; StringQ[y]] :=
  Select[FileNames["*.*", x, Infinity], StandPath[FileNameSplit[#][[-1]]] === StandPath[y] &]
End[]

Begin["`NamesMPackage`"]
NamesMPackage[f_ /; IsFile[f] && FileExtension[f] == "m" && ! SameQ[ContextFromFile[f], $Failed]] :=
  Module[{c, d, Res = {}, s = "::usage=\"", a = OpenRead[f]}, Label[c];
    d = Read[a, String];  If[SuffPref[d, "(*Begin[\"`", 1] || d === EndOfFile, Close[a];
      Return[Sort[DeleteDuplicates[Res]]], If[SuffPref[d, "(*", 1] && ! StringFreeQ[d, s],
        AppendTo[Res, StringTake[d, {3, Flatten[StringPosition[d, s]][[1]] - 1}]];  Goto[c], Goto[c]]]]
End[]

Begin["`PackageMxCont`"]
PackageMxCont[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y___] :=
  Module[{a, b, c, d}, If[IsPackageQ[x] === $Failed, $Failed, a = ContextsInFiles[x];
      If[a === $Failed, $Failed,
        If[MemberQ[$Packages, a], d = 74, d = 69];  c = If[StringFreeQ[AladjevProcedures`OSplatform[][[1]], " 7 "],
            AladjevProcedures`ContMxFile[x, 90], AladjevProcedures`ContMxW7[x]];
        If[c == {} || c[[2]] == {}, b = {}, b = a];  If[{y} ≠ {} && ! AladjevProcedures`HowAct[y] && ! SameQ[a, {}], y = c, Null];
        If[d == 69, AladjevProcedures`RemovePackage[a], Null];  b]]]
End[]

Begin["`CurrentPackageQ`"]
CurrentPackageQ[x_ /; ContextQ[x]] := MemberQ[$Packages, x]
End[]

Begin["`ReplaceAll1`"]
ReplaceAll1[x_, y_, z_] := Module[{a, b, c},
    If[! HowAct[y], x /. y -> z, c = If[MemberQ[{Plus, Times, Power}, Head[z]], "(" <> ToString[InputForm[z]] <> ")", ToString[z]];
      {a, b} = Map[ToString, Map[InputForm, {x, y}]];
      If[StringLength[b] == 1, ReplaceAll[x, y -> z], ToExpression[StringReplace[a, b -> c]]]]]
End[]

Begin["`ReplaceAll2`"]
ReplaceAll2[x_, r_ /; RuleQ[r] || ListRulesQ[r]] :=
  Module[{a = x, k = 1}, If[RuleQ[r], Replace4[x, r], While[k ≤ Length[r], a = Replace4[a, r[[k]]];
        k ++]; a]]
End[]

Begin["`Replace1`"]
Replace1[x_, y_ /; Head[y] == Rule || ListQ[y] && DeleteDuplicates[Map[Head, y]] == {Rule}] :=
  Module[{a = x // FullForm // ToString, b = UnDefVars[x], c, h = {}, d = ToStringRule[DeleteDuplicates[Flatten[{y}]]], p, l, r, k = 1},
    p = Map14[RhsLhs, d, "Lhs"]; c = Select[p, ! MemberQ[Map[ToString, b], #] &];
    If[c ≠ {}, Print["Rules " <> ToString[Flatten[Select[d, MemberQ[c, RhsLhs[#, "Lhs"]] &]]] <> " are vacuous"]];
    While[k ≤ Length[d], l = RhsLhs[d[[k]], "Lhs"];
      r = RhsLhs[d[[k]], "Rhs"];
      h = Append[h, {"[" <> l → "[" <> r, " " <> l → " " <> r, l <> "]" → r <> "]"}]; k ++];
    Simplify[ToExpression[StringReplace[a, Flatten[h]]]]]
End[]

Begin["`Replace2`"]
Replace2[x_, y_, z_] := Module[{a = Map7[ToString, FullForm, {x, y, z}], b}, SetAttributes[ToString1, Listable];
    b = ToExpression[StringReplace[a[[1]], a[[2]] → a[[3]]]];
    If[DeleteDuplicates[StringFreeQ[ToString1[Flatten[{b}]], "\""]] == {True}, ClearAttributes[ToString1, Listable];
      b, ClearAttributes[ToString1, Listable]; $Failed]]
End[]

Begin["`Replace3`"]
Replace3[x_, y_, z_] := Module[{a = Flatten[{y}], b = Flatten[{z}], c, t = x, k = 1}, c = Min[Map[Length, {a, b}]];
    If[c < Length[a], Print["Subexpressions " <> ToString1[a[[c + 1 ;; -1]]] <> " were not replaced"]];
    For[k, k ≤ c, k ++, t = Subs[t, a[[k]], b[[k]]]];  t]
End[]
```

```
Begin["`Replace4`"]
Replace4[x_, r_ /; RuleQ[r]] := Module[{a, b, c, h}, {a, b} = {ToString[x // InputForm], Map[ToString, Map[InputForm, r]]};
    c = StringPosition[a, Part[b, 1]];
    If[c == {}, x,
      If[Head[Part[r, 1]] === Plus,
        h = Map[If[(#[[1]] === 1 || MemberQ[{" ", "(", "[", "{"}, StringTake[a, {#[[1]] - 1, #[[1]] - 1}]]) &&
                  (#[[2]] === StringLength[a] || MemberQ[{" ", ")", "]", "}", ","}, StringTake[a, {#[[2]] + 1, #[[2]] + 1}]]), #] &, c],
        h = Map[If[(#[[1]] === 1 || ! Quiet[SymbolQ[StringTake[a, {#[[1]] - 1, #[[1]] - 1}]]]) &&
                  (#[[2]] === StringLength[a] || ! Quiet[SymbolQ[StringTake[a, {#[[2]] + 1, #[[2]] + 1}]]]), #] &, c]];
      h = Select[h, ! SameQ[#, Null] &]; ToExpression[StringReplacePart[a, "(" <> Part[b, 2] <> ")", h]]]]
End[]

Begin["`ReplaceSubLists`"]
ReplaceSubLists[x_ /; ListQ[x], y_ /; RuleQ[y] || ListRulesQ[y]] :=
  Module[{a, f, d = FromCharacterCode[2016]}, f[z_ /; ListQ[z]] := StringJoin[Map[ToString1[#] <> d &, z]];
    a = Map[f[Flatten[{#[[1]]}]] → f[Flatten[{#[[2]]}]] &, Flatten[{y}]]; ToExpression[StringSplit[StringReplace[f[x], a], d]]]
End[]

Begin["`FuncToPure`"]
FuncToPure[x_ /; QFunction[ToString[x]], y___] :=
  Module[{a = HeadPF[x], b = Map[ToString, Args[x]], c = Definition2[x][[1]], d, k = 1, h, t, g = {}, p},
    d = Map[First, Mapp[StringSplit, b, "_"]];
    p = StringTrim[c, a <> " := "]; h = "Hold[" <> p <> "]"; {t, h} = {Length[b], ToExpression[h]};
    While[k ≤ t, AppendTo[g, d[[k]] <> " –> #" <> ToString[k]]; k++];
    h = ToString1[ReplaceAll[h, ToExpression[g]]]; g = StringTake[h, {6, –2}];
    ToExpression[
      If[{y} ≠ {}, "Function[" <> If[Length[b] == 1, StringTake[ToString[d], {2, –2}], ToString[d]] <> ", " <> p <> "]", g <> " &"]]]
End[]

Begin["`BlockFuncModVars`"]
BlockFuncModVars[x_ /; BlockFuncModQ[x]] :=
  Module[{d, t, c = Args[x, 90], a = If[QFunction[x], {}, LocalsGlobals1[x]], s = {"System"},
      u = {"Users"}, b = Flatten[{PureDefinition[x]}][[1]], h = {}}, d = ExtrVarsOfStr[b, 2];
    If[a == {}, t = Map[If[Quiet[SystemQ[#]], AppendTo[s, #], If[BlockFuncModQ[#], AppendTo[u, #], AppendTo[h, #]]] &, d];
      {s, u = Select[u, # ≠ ToString[x] &], c, MinusList[d, Join[s, u, c, {ToString[x]}]]},
      Map[If[Quiet[SystemQ[#]], AppendTo[s, #], If[BlockFuncModQ[#], AppendTo[u, #], AppendTo[h, #]]] &, d];
      {Select[s, ! MemberQ[{"$Failed", "True", "False"}, #] &], Select[u, # ≠ ToString[x] && ! MemberQ[a[[1]], #] &],
        c, a[[1]], a[[3]], Select[h, ! MemberQ[Join[a[[1]], a[[3]], c, {"System", "Users"}], #] &]}]]
End[]

Begin["`StrStr`"]
StrStr[x_] := If[StringQ[x], "\"" <> x <> "\"", ToString[x]]
End[]

Begin["`IsPermutation`"]
IsPermutation[x_ /; StringQ[x]] := Module[{a = Characters[x]}, If[Length[a] == Length[DeleteDuplicates[a]], True, False]]
End[]

Begin["`Nvalue`"]
Nvalue[x_] := Module[{a = {}, b = Names["`*"], k = 1},
    For[k, k ≤ Length[b], k++, If[ToExpression[b[[k]]] == x, AppendTo[a, b[[k]]], Next[]]]; Select[a, ! SuffPref[#, "Global`", 1] &]
]
End[]

Begin["`Nvalue1`"]
Nvalue1[x_] := Module[{a = {}, b = Select[Names["*"], StringFreeQ[#, "$"] &], c, k = 1},
    While[k ≤ Length[b], c = ToExpression["Attributes[" <> ToString1[b[[k]]] <> "]"];
      If[! MemberQ[c, Protected], AppendTo[a, b[[k]]], Null]; k++]; Select[a, ToExpression[#] === x &]]
End[]

Begin["`DelSuffPref`"]
DelSuffPref[x_ /; StringQ[x], y_ /; StringQ[y], n_ /; MemberQ[{1, 2, 3}, n]] :=
  Module[{a = StringLength[y]}, If[! SuffPref[x, y, n], x, StringTake[x, {{a + 1, –1}, {1, –(a + 1)}, {a + 1, –(a + 1)}}[[n]]]]]
End[]

Begin["`MaxLevel`"]
MaxLevel[x_ /; ListQ[x]] := Module[{a = x, k = 0}, While[NestListQ1[a], k++;
      a = Flatten[a, 1];
      Continue[]]; k]
```

```
End[]

Begin["`ListLevels`"]
ListLevels[x_ /; ListQ[x]] := Module[{a = x, b, c = {}, k = 1}, If[! NestListQ1[x], {0}, While[NestListQ1[a], b = Flatten[a, 1];
        If[Length[b] ≥ Length[a], AppendTo[c, k++], AppendTo[c, k]]; a = b; Continue[]]; c]]
End[]

Begin["`HowAct`"]
HowAct[x_] := If[Quiet[Check[ToString[Definition[x]], True]] === "Null", False,
   If[ProtectedQ[x], True, If[Quiet[ToString[Definition[x]]] === "Attributes[" <> ToString[x] <> "] = {Temporary}", False, True]]]
End[]

Begin["`UndefinedQ`"]
UndefinedQ[x_] :=  If[SymbolQ[x], If[MemberQ[{$Failed, "Undefined"}, PureDefinition[x]], True, False], False]
End[]

Begin["`DefinedActSymbols`"]
DefinedActSymbols[] := Select[Names["`*"], ! (TemporaryQ[#] || UndefinedQ[#]) &]
End[]

Begin["`PathToFileQ`"]
PathToFileQ[x_ /; StringQ[x]] :=
  If[StringLength[x] >= 3, If[MemberQ[Join[CharacterRange["a", "z"], CharacterRange["A", "Z"]], StringTake[x, 1]] &&
        StringTake[x, {2, 2}] == ":" && And[Map3[StringFreeQ, x, {"/", "\\"}]] ≠ {True, True}, True, False], False]
End[]

Begin["`SortLpos`"]
SortLpos[L_ /; NestListQ[L], n_ /; IntegerQ[n], SF_] :=
  If[! 1 ≤ n ≤ Min[Map[Length, L]], $Failed, If[! MemberQ[{Greater, Less}, SF], $Failed,
      If[DeleteDuplicates[Map[NumericQ[#] &, Flatten[L]]] ≠ {True}, $Failed, Sort[L, SF[#2[[n]], #1[[n]]] &]]]]
End[]

Begin["`SubListsMin`"]
SubListsMin[L_ /; ListQ[L], x_, y_, t_ /; MemberQ[{"r", "l"}, t], z___] :=
  Module[{a, b, c, d = {}, k = 1, j}, {a, b} = Map[Flatten, Map3[Position, L, {x, y}]];
    If[a == {} || b == {} || a == {} && b == {} || L == {}, {},
      b = Select[Map[If[If[t == "r", Greater, Less][#, a[[1]]], #] &, b], ! SameQ[#, Null] &];
      For[k, k ≤ Length[a], k++, j = 1;
        While[j ≤ Length[b], If[If[t == "r", Greater, Less][b[[j]], a[[k]]], AppendTo[d, If[t == "r", a[[k]] ;; b[[j]], b[[j]] ;; a[[k]]]];
            Break[]]; j++]];
      d = Sort[d, Part[#1, 2] − Part[#1, 1] <= Part[#2, 2] − Part[#2, 1] &];
      d = Select[d, Part[#, 2] − Part[#, 1] == Part[d[[1]], 2] − Part[d[[1]], 1] &];
      d = Map[L[[#]] &, d]; d =  If[{z} ≠ {}, Map[#[[2 ;; −2]] &, d], d]; If[Length[d] == 1, Flatten[d], d]]]
End[]

Begin["`NamesNbPackage`"]
NamesNbPackage[f_ /; IsFile[f] && FileExtension[f] == "nb" && ! SameQ[ContextFromFile[f], $Failed]] := Module[{Res = {}, Tr},
   Tr[x_ /; StringQ[x]] := Module[{c, d, h, g = "\"::\"", v = "\"=\"", p = "\"usage\"", a = OpenRead[x], s = "   RowBox[{"}, Label[c];
      d = Read[a, String];  If[d === EndOfFile, Close[a];  Return[Res], Null];
      If[DeleteDuplicates[Map3[StringFreeQ, d, {s, g, p, v}]] == {False} && SuffPref[d, s, 1], h = Flatten[StringPosition[d, g]];
        AppendTo[Res, StringTake[d, {12, h[[1]] − 3}]]; Goto[c], Goto[c]]];  Map[ToExpression, Sort[Tr[f]]]]
End[]

Begin["`ExcessVarsPack`"]
ExcessVarsPack[x_ /; ContextQ[x] && MemberQ[$Packages, x]] :=
  Module[{a = {{}, {}, {}, {}, {}}, b = Sort[DeleteDuplicates[MinusList[Names[x <> "*"], CNames[x]]]]},
    Map[ If[TemporaryQ[#] && PureDefinition[#] ≠ $Failed,  AppendTo[a[[1]], #], If[TemporaryQ[#] && PureDefinition[#] ===
                $Failed, AppendTo[a[[2]], #], If[! SameQ[PureDefinition[#], $Failed] && Context[#] == x ,  AppendTo[a[[3]], #],
              If[SameQ[PureDefinition[#], $Failed] && Context[#] == x, AppendTo[a[[4]], #], AppendTo[a[[5]], #]]]]] &, b];  a]
End[]

Begin["`NamesNbPackage1`"]
NamesNbPackage1[f_ /; IsFile[f] && FileExtension[f] == "nb" && ! SameQ[ContextFromFile[f], $Failed]] :=
  Module[{c, d, g = "::", a = OpenRead[f], p = "usage", v = "=", Res = {}, s = "   RowBox[{"}, Label[c];
    d = Read[a, String];  If[d === EndOfFile, Close[a];  Return[Sort[Map[ToExpression, Res]]],
      If[DeleteDuplicates[Map3[StringFreeQ, d, {s, g, p, v}]] == {False} && SuffPref[d, s, 1],
        AppendTo[Res, StringReplace[StringSplit[d, ","][[1]], s → ""]];  Goto[c]];  Goto[c]]]
End[]
```

```
Begin["`RedMfile`"]
RedMfile[x_ /; FileExistsQ[x] && FileExtension[x] == "m", p_ /; SymbolQ[p], r_ /; MemberQ[{"add", "delete", "replace"}, r]] :=
  Module[{a = ReadList[x, String], d = ToString[p], h, save, b = "(*Begin[\"`" <> ToString[p] <> "`\"]*)", c = "(*End[]*)"},
    If[MemberQ[! ContentOfMfile[x], ToString[p]] && r == "delete" ||
        MemberQ[{"add", "replace"}, r] && ! (ProcQ[p] || QFunction[p]), $Failed,
      save[q_] := Module[{f, k = 1}, f = DirectoryName[x] <> FileBaseName[x] <> "$.m";
        While[k ≤ Length[q], WriteString[f, q[[k]], "\n"]; k++]; Close[f]];
      If[! MemberQ[a, "(* ::Package:: *)"], $Failed, If[r === "delete", h = Select[a, SuffPref[#, "(*" <> d <> "::usage", 1] &];
        If[h == {}, x, a = Select[a, ! SuffPref[#, "(*" <> d <> "::usage", 1] &]; d = SubListsMin[a, b, c, "r"]; d = MinusList[a, d];
          save[d]], If[r === "add" && Select[a, SuffPref[#, "(*" <> d <> "::usage=", 1] &] == {} && Head[p::usage] == String &&
            (ProcQ[p] || FunctionQ[p]), h = PosListTest[a, SuffPref[#, {"(*BeginPackage[", "(*EndPackage[]"}, 1] &];
          a = Insert[a, "(*" <> d <> "::usage=" <> ToString1[p::usage] <> "*)", h[[1]] + 1];
          a = Flatten[Insert[a, {"(*Begin[\"`" <> d <> "`\"]*)", "(*" <> PureDefinition[p] <> "*)", "(*End[]*)"}, h[[2]] + 1]];
          save[a], If[r === "replace" && Head[p::usage] == String && (ProcQ[p] || QFunction[p]),
            h = PosListTest[a, SuffPref[#, "(*Begin[\"`" <> d <> "`\"]*)", 1] &];
            If[h == {}, $Failed, a[[h[[1]] ;; h[[1]] + 2]] = {"(*Begin[\"`" <> d <> "`\"]*)", "(*" <> PureDefinition[p] <> "*)",
                  "(*End[]*)"}; h = PosListTest[a, SuffPref[#, "(*" <> d <> "::usage=", 1] &];
              a[[h[[1]]]] = "(*" <> d <> "::usage=" <> ToString1[p::usage] <> "*)";
              save[Flatten[a]]]]], x]]]
End[]

Begin["`RedMxFile`"]
RedMxFile[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; StringQ[y] && SymbolQ[y],
    r_ /; MemberQ[{"add", "delete", "replace"}, r], f___] := Module[{a, c, c1 = ContextFromFile[x], c2, save, t},
    If[MemberQ[$Packages, c1] && CNames[c1] ≠ {}, t = 74, Get[x]];
    save[z_] := Module[{p = DirectoryName[z] <> FileBaseName[z] <> "$.mx"},
        ToExpression["DumpSave[" <> ToString1[p] <> "," <> ToString1[c1] <> "]"];
        p];
    If[r == "delete" && MemberQ[CNames[c1], y], Unprotect[y]; Remove[y]; c = save[x],
      If[r == "replace" && MemberQ[a, y] && {f} ≠ {} && FileExistsQ[f] && FileExtension[f] == "mx", Unprotect[y];
        Remove[y];
        Get[f]; c = save[x],
        If[r == "add" && {f} ≠ {} && FileExistsQ[f] && FileExtension[f] == "mx", Unprotect[y];
          Remove[y]; Get[f]; c = save[x], c = $Failed]]]; DeleteFile[f]; If[t == 74, Null, RemovePackage[c1]]; c]
End[]

Begin["`ProcsAct`"]
ProcsAct[] := Module[{a = Names["*"], b = Names["System`*"], c, d = {}, k = 1, j, h, t,
      g = {{"Module"}, {"Block"}, {"DynamicModule"}, {"Function"}, {"Others"}}}, c = Select[a, ! MemberQ[b, #] &];
    c = Select[c, ToString[Definition[#]] ≠ "Null" &&
          ToString[Definition[#]] != "Attributes[" <> ToString[#] <> "] = {Temporary}" &&
          ! MemberQ[{ToString[#] <> " = {Temporary}", ToString[#] <> " = {Temporary}"}, ToString[Definition[#]]] &];
    For[k, k ≤ Length[c], k++, h = c[[k]];
      ClearAll[t];
      Quiet[ProcQ1[Symbol[h], t]];
      t = Quiet[Check[t[[2]][[1]], $Failed]];
      If[t === "Module", AppendTo[g[[1]], h], If[t === "Block", AppendTo[g[[2]], h],
        If[t === "DynamicModule", AppendTo[g[[3]], h], If[QFunction[h], AppendTo[g[[4]], h], AppendTo[g[[5]], h]]]]]; g]
End[]

Begin["`NamesCS`"]
NamesCS[P_ /; ! HowAct[P], Pr_ /; ! HowAct[Pr], Pobj_ /; ! HowAct[Pobj]] :=
  Module[{a = Quiet[Select[Map[ToExpression, Names["`*"]], ProcQ[#] &]],
      b = Contexts[], c = $Packages, d, k = 1, p, n, m, h}, {P, Pr} = {c, a};
    c = Map[List, c]; For[k, k ≤ Length[b], k++, For[p = 1, p ≤ Length[c], p++, n = b[[k]];
        m = c[[p]][[1]]; If[n === m, Null, If[SuffPref[n, m, 1], d = StringReplace[n, b → ""];
          If[d == "", Null, c[[p]] = Append[c[[p]], ToExpression[StringTake[StringReplace[n, b → ""], {1, -2}]]]]], Continue[]]]];
    c = Map[DeleteDuplicates, c]; For[k = 1, k ≤ Length[c], k++, h = c[[k]];
      If[Length[h == 1], h = Null, h = Select[h, StringQ[#] || ToString[Quiet[DefFunc[#]]] ≠ "Null" &]]];
    Pobj := Select[c, Length[#] > 1 && ! # === Null &];
    Pobj = Mapp[Select, Pobj, If[! StringQ[#], True, If[StringTake[#, -1] == "`", True, False]] &];]
End[]

Begin["`NamesContext`"]
NamesContext[x_ /; ContextQ[x]] := Module[{b, c = {}, k = 1, h, a = Names[x <> "*"]}, While[k ≤ Length[a], b = a[[k]];
      h = Quiet[ToString[ToExpression["Definition[" <> b <> "]"]]];
      If[h ≠ "Null" && h ≠ "Attributes[" <> b <> "] = {Temporary}" && ! SuffPref[b, "a$", 1], AppendTo[c, a[[k]]]];
      k++]; c]
End[]
```

```
Begin["`ContentObj`"]
ContentObj[x_ /; BlockQ[x] || FunctionQ[x] || ModuleQ[x]] :=
  Module[{a = ToString1[FullDefinition[x]], b}, a = StringCases[a, X__ ~~ " /: " ~~ X__];
    If[a == {}, {}, a = Map[StringTake[#, {1, Flatten[StringPosition[#, " /: "]][[1]] - 1}] &, a][[2 ;; -1]];
      a = Map[DeleteDuplicates, Map[Flatten, Gather[Map[{Context[#], #} &, a], #1[[1]] == #2[[1]] &]]];
      a = Map[Flatten[{#[[1]], Sort[#[[2 ;; -1]]]}] &, a]; If[Length[a] == 1, a[[1]], a]]
End[]

Begin["`Map3`"]
Map3[F_ /; SymbolQ[F], g_, L_ /; ListQ[L]] := Map[Symbol[ToString[F]][g, #] &, L]
End[]

Begin["`Map5`"]
Map5[F_, L_ /; NestListQ[L]] := Map[F[Sequences[#]] &, L]
End[]

Begin["`SortQ`"]
SortQ[s_ /; StringQ[s]] := If[s === StringJoin[Sort[Characters[s]]], True, False]
End[]

Begin["`Map6`"]
Map6[F_ /; PureFuncQ[F], L_ /; ListListQ[L]] :=
  Module[{a, b = Length[L], k = 1, c = Length[L[[1]]], d = {}, h, p}, h = StringTake[ToString[F], {1, -4}];
    For[k, k ≤ b, k++, a = {};
      AppendTo[d,
        StringReplace[h, Flatten[{For[p = 1, p ≤ c, p++, AppendTo[a, "#" <> ToString[p] -> ToString[L[[k]][[p]]]]], a}][[2 ;; -1]]]];
    ToExpression[d]]
End[]

Begin["`Map7`"]
Map7[x__ /; SameQ[DeleteDuplicates[Map[SymbolQ, {x}]], {True}], L_ /; ListQ[L]] :=
  Map[FunCompose[Reverse[Map[Symbol, Map[ToString, {x}]]], #] &, L]
End[]

Begin["`Map8`"]
Map8[x__ /; DeleteDuplicates[Map[SymbolQ, {x}]] === {True}, y_ /; ListQ[y]] := Map[Symbol[ToString[#]][Sequences[y]] &, {x}]
End[]

Begin["`Map9`"]
Map9[F_ /; SymbolQ[F], x_ /; ListQ[x], y_ /; ListQ[y]] := If[Length[x] == Length[y], Map13[F, {x, y}], Defer[Map9[F, x, y]]]
End[]

Begin["`Map10`"]
Map10[F_ /; SymbolQ[F], x_, L_ /; ListQ[L], y___] := Map[Symbol[ToString[F]][x, #, Sequences[{y}]] &, L]
End[]

Begin["`Map11`"]
Map11[x_ /; SymbolQ[x], y_ /; ListQ[y], z_] := Map[If[ListQ[#], Map[x[#1, z] &, #], x[#, z]] &, y]
End[]

Begin["`Map12`"]
Map12[F_ /; SymbolQ[F], x_ /; NestListQ1[x]] :=
  Module[{a = ToString1[x], b = ToString[F] <> "@", c}, c = StringReplace[a, {"{" → "{" <> b, ", " → "," <> b}];
    c = StringReplace[c, b <> "{" -> "{"];
    ToExpression[c]]
End[]

Begin["`Map13`"]
Map13[x_ /; SymbolQ[x], y_ /; ListListQ[y]] :=
  Module[{a = Length[y], b = Length[y[[1]]], k, j, c = {}, d = {}}, For[k = 1, k ≤ b, k++, For[j = 1, j ≤ a, j++, AppendTo[c, y[[j]][[k]]]];
      AppendTo[d, Apply[x, c]]; c = {}]; d]
End[]

Begin["`Map14`"]
Map14[x_ /; SymbolQ[x], y_ /; ListQ[y], z_, t___] :=
  Module[{a = {}, k = 1}, If[y == {}, Return[$Failed], While[k ≤ Length[y], a = Append[a,
            If[{t} == {}, ToExpression, ToString][ToString[x] <> "[" <> ToString1[y[[k]]] <> ", " <> ToString1[z] <> "]"]]; k++]];
    a]
```

```
End[]

Begin["`Map15`"]
Map15[x__ /; SameQ[DeleteDuplicates[Map[SymbolQ, {x}]], {True}], y_] := Map[#[y] &, {x}]
End[]

Begin["`Map16`"]
Map16[f_ /; SymbolQ[f], l_ /; ListQ[l], x___] :=
  Quiet[(f[#1, FromCharacterCode[6]] & ) /@ 1 /. FromCharacterCode[6] -> Sequence[x]]
End[]

Begin["`Map17`"]
Map17[x_, y_ /; RuleQ[y] || ListRulesQ[y]] := If[RuleQ[y], Map[x, y], Map[Map[x, #] &, y]]
End[]

Begin["`Map18`"]
Map18[x_ /; ListQ[x] && DeleteDuplicates[Map[SymbolQ[#] &, x]] == {True}, y_ /; ListQ[y]] := Map[Map[#, y] &, x]
End[]

Begin["`Map19`"]
Map19[x_ /; ListQ[x], y_ /; ListQ[y]] :=
  Module[{a = {}, k = 1}, If[Length[x] == Length[y] && DeleteDuplicates[Map[SymbolQ[#] &, x]] == {True},
      Do[AppendTo[a, Flatten[{x[[k]], y[[k]]}]], {k, Length[x]}];
      Map[#[[1]] @@ #[[2 ;; -1]] &, a], $Failed]]
End[]

Begin["`Map20`"]
Map20[x_ /; ListQ[x], y_ /; NestListQ[y]] := If[Length[x] ≠ Length[y], $Failed, Map[#[[1]] @@ #[[2]] &, Partition[Riffle[x, y], 2]]]
End[]

Begin["`EquExprPatt`"]
EquExprPatt[x_, y_ /; ExprPatternQ[y]] := Module[{c, d = {}, j, t, v = {}, k = 1, p, g = {}, s = {},
      a = Map[FullForm, Map[Expand, {x, y}]], b = Mapp[MinusList, Map[OP, Map[Expand, {x, y}]], {FullForm}],
      z = SetAttributes[ToString, Listable], w}, {b, c} = ToString[{b, a}];
    p = StringPosition[c[[2]], {"Pattern[", "Blank[]]"}];
    While[k = 2 * k - 1; k ≤ Length[p], AppendTo[d, StringTake[c[[2]], {p[[k]][[1]], p[[k + 1]][[2]]}]]; k++];
    {t, k} = {ToExpression[d], 1}; While[k ≤ Length[t], AppendTo[v, StringJoin[ToString[Op[t[[k]]]]]];
      k++];
    v = ToString[v]; v = Map13[Rule, {d, v}]; v = StringReplace[c[[2]], v];
    b = Quiet[Mapp[Select, b, ! SystemQ[#] || BlockFuncModQ[ToString[#]] &]];
    {b, k, j} = {ToString[b], 1, 1}; While[k ≤ Length[b[[1]]], z = b[[1]][[k]];
      AppendTo[g, {"[" <> z <> "," → "[w", " " <> z <> "," → " w", "[" <> z <> "]" → "[w]", " " <> z <> "]" → " w]"}];
      k++];
    While[j ≤ Length[b[[2]]], z = b[[2]][[j]];
      AppendTo[s, {"[" <> z <> "," → "[w", " " <> z <> "," → " w", "[" <> z <> "]" → "[w]", " " <> z <> "]" → " w]"}];
      j++];
    ClearAttributes[ToString, Listable]; z = Map9[StringReplace, {c[[1]], v}, Map[Flatten, {g, s}]];
    SameQ[z[[1]], StringReplace[z[[2]], Join[GenRules[Flatten[Map[# <> "," &, Map[ToString, t]]], "w"], GenRules[
          Flatten[Map[# <> "]" &, Map[ToString, t]]], "w"], GenRules[Flatten[Map[# <> ")" &, Map[ToString, t]]], "w)"]]]]]
End[]

Begin["`EquExprPatt1`"]
EquExprPatt1[x_, y_] := Module[{c, d = {}, t, v = {}, k = 1, j, p, g = {}, s = {}, a = Map[FullForm, Map[Expand, {x, y}]],
      b = Mapp[MinusList, Map[OP, Map[Expand, {x, y}]], {FullForm}], z = SetAttributes[ToString, Listable], w},
    {b, c} = ToString[{b, a}]; p = StringPosition[c[[2]], {"Pattern[", "Blank[]]"}];
    While[k = 2 * k - 1; k ≤ Length[p], AppendTo[d, StringTake[c[[2]], {p[[k]][[1]], p[[k + 1]][[2]]}]]; k++];
    {t, k} = {ToExpression[d], 1}; While[k ≤ Length[t], AppendTo[v, StringJoin[ToString[Op[t[[k]]]]]];
      k++];
    v = ToString[v]; v = Map13[Rule, {d, v}]; v = StringReplace[c[[2]], v];
    b = Quiet[Mapp[Select, b, ! SystemQ[#] || ProcQ[#] || QFunction[ToString[#]] &]]; {b, k, j} = {ToString[b], 1, 1};
    While[k ≤ Length[b[[1]]], z = b[[1]][[k]];
      AppendTo[g, {"[" <> z <> "," → "[w", " " <> z <> "," → " w", "[" <> z <> "]" → "[w]", " " <> z <> "]" → " w]"}]; k++];
    While[j ≤ Length[b[[2]]], z = b[[2]][[j]];
      AppendTo[s, {"[" <> z <> "," → "[w", " " <> z <> "," → " w", "[" <> z <> "]" → "[w]", " " <> z <> "]" → " w]"}]; j++];
    ClearAttributes[ToString, Listable];
    z = Map9[StringReplace, {c[[1]], v}, Map[Flatten, {g, s}]];
    SameQ[z[[1]], StringReplace[z[[2]], Join[GenRules[Flatten[Map[# <> "," &, Map[ToString, t]]], "w"], GenRules[
          Flatten[Map[# <> "]" &, Map[ToString, t]]], "w"], GenRules[Flatten[Map[# <> ")" &, Map[ToString, t]]], "w)"]]]]]
End[]
```

```
Begin["`PureFuncQ`"]
PureFuncQ[F_] := Quiet[StringTake[ToString1[F], {-3, -1}] == " & " && ! StringFreeQ[ToString[F], "#"] ||
    SuffPref[ToString[InputForm[F]], "Function[", 1]]
End[]

Begin["`PureToFunc`"]
PureToFunc[x_ /; PureFuncQ[x], y_ /; ! HowAct[y]] :=
  Module[{a = Sort[Select[ExprComp[x], Head[#] === Slot &]], b = {}, c = {}, k}, Do[AppendTo[c, Unique["x"]], {k, 1, Length[a]}];
    Do[AppendTo[b, a[[k]] → c[[k]]], {k, 1, Length[a]}];
    Simplify[ToExpression[StringTake[ToString[y @@ Map[ToExpression, Map[# <> "_" &, Map[ToString, c]]]] <>
          ":=" <> ToString1[ReplaceAll[x, b]], {1, -4}]]]]
End[]

Begin["`Definition1`"]
Definition1[x_] := Module[{a, b, c, h = If[StringQ[x], ToExpression[x], x]},
    If[! SymbolQ[h], a = $Failed, If[SystemQ[h], a = "Null", b = Quiet[{Attributes[h], ClearAllAttributes[h]}[[1]]];
        Quiet[Off["Definition"::"notfound"];
          a = ToString1[ToExpression["Definition[" <> ToString1[h] <> "]"]]];
        If[! a == "Null",
          c = Flatten[{HeadPF[h]}];
          a = StringSplit[a, c];
          a = Partition[Riffle[a, c], 2];
          a = Map[StringJoin[#[[2]], #[[1]]] &, a];
          Quiet[On["Definition"::"notfound"]]; SetAttributes[h, b]]], Null];
    If[SystemQ[h] && a === $Failed, "Null", If[Length[a] == 1, a[[1]], a]]]
End[]

Begin["`Definition2`"]
Definition2[x_ /; SameQ[SymbolQ[x], HowAct[x]]] :=
  Module[{a, b = Attributes[x], c}, If[SystemQ[x], Return[{"System", Attributes[x]}], Off[Part::partw]];
    ClearAttributes[x, b];
    Quiet[a = ToString[InputForm[Definition[x]]];
      Mapp[SetAttributes, {Rule, StringJoin}, Listable];
      c = StringReplace[a, Flatten[{Rule[StringJoin[Contexts1[], ToString[x] <> "`"], ""]}]];
      c = StringSplit[c, "\n \n"]; Mapp[ClearAttributes, {Rule, StringJoin}, Listable];
      SetAttributes[x, b]; a = AppendTo[c, b];
      If[SameQ[a[[1]], "Null"] && a[[2]] == {}, On[Part::partw];
        {"Undefined", Attributes[x]}, If[SameQ[a[[1]], "Null"] && a[[2]] ≠ {} && ! SystemQ[x], On[Part::partw];
          {"Undefined", Attributes[x]}, If[SameQ[a[[1]], "Null"] && a[[2]] ≠ {} && a[[2]] ≠ {}, On[Part::partw];
            {"System", Attributes[x]}, On[Part::partw]; a]]]]
End[]

Begin["`Definition3`"]
Definition3[x_ /; SymbolQ[x], y_ /; ! HowAct[y]] := Module[{a = Attributes[x], b = Definition2[x]}, If[b[[1]] == "System", y = x;
      {"System", a}, b = Definition2[x][[1 ;; -2]];
      ClearAttributes[x, a]; If[BlockFuncModQ[x, y], ToExpression[b];
        SetAttributes[x, a]; Definition[x], SetAttributes[x, a]; Definition[x]]]]
End[]

Begin["`Definition4`"]
Definition4[x_ /;  StringQ[x]] :=
  Module[{a}, a = Quiet[Check[Select[StringSplit[ToString[InputForm[Quiet[Definition[x]]]], "\n"], # ≠ " " && # ≠ x &], $Failed]];
    If[a === $Failed, $Failed, If[SuffPref[a[[1]], "Attributes[", 1], a = AppendTo[a[[2 ;;  -1]], a[[1]]]];  If[Length[a] ≠ 1, a, a[[1]]]]
End[]

Begin["`OptRes`"]
OptRes[x_ /; SymbolQ[x], y_] := If[Mapp[SetAttributes, {Rule, StringJoin}, Listable];
    StringQ[y] && StringFreeQ[y, "`" <> ToString[x] <> "`"], Mapp[ClearAttributes, {Rule, StringJoin}, Listable];
    y, {If[StringStringQ[y], ToExpression, Evaluate][StringReplace[ToString1[y], Flatten[
          {Rule[StringJoin[Contexts1[], ToString[x] <> "`"], ""]}]]], Mapp[ClearAttributes, {Rule, StringJoin}, Listable]}[[1]]]
End[]

Begin["`Def`"]
Def[x_ /; StringQ[x]] := Module[{a}, If[! SymbolQ[x] || SystemQ[x], $Failed, a = Definition2[x][[1 ;; -2]];
      If[Length[a] == 1, a[[1]], a]]]
End[]

Begin["`Def1`"]
```

```
Def1[x_ /; StringQ[x]] := If[! SymbolQ[x] || SystemQ[x], $Failed, Definition2[x][[1 ;; -2]]]
End[]

Begin["`Attributes1`"]
Attributes1[x__] := Module[{a = Map[Quiet[Check[Attributes[#], {}]] &, {x}]}, If[Length[a] == 1, a[[1]], a]]
End[]

Begin["`ProcContent`"]
ProcContent[x_ /; BlockFuncModQ[x]] := Module[{a, b = SubProcs[x][[1]], f},
    f[y_] := Module[{a1 = "$Art2720Kr$", b1 = "", c = {y}, d, h = "", p},
        Save[a1, y]; While[! SameQ[b1, EndOfFile], b1 = Read[a1, String]; Map[ClearAll, SubProcs[x][[2]]];
          If[! MemberQ[{" ", "EndOfFile"}, ToString[b1]], h = h <> ToString[b1];
            Continue[], d = Flatten[StringPosition[h, " := ", 1]]];
          If[d == {} , h = "";
            Continue[], p = StringTake[h, {1, d[[1]] - 1}];
            If[! SameQ[Quiet[ToExpression[p]], $Failed],
              AppendTo[c, StringTake[p, {1, Flatten[StringPosition[p, "[", 1]][[1]] - 1}]]; h = "", Null]]];
        a1 = Map[ToExpression, {DeleteFile[Close[a1]], c}[[2]]]; DeleteDuplicates[a1]; a = f[x];
    {x, If[Length[a] > 1, a[[2 ;; -1]], {}], If[Length[b] > 1, Map[ToExpression, Map[HeadName, b[[2 ;; -1]]]], {}]}]
End[]

Begin["`CallsInMean`"]
CallsInMean[x_ /; SymbolQ[x]] := Module[{a = Context[x], b, c}, If[! MemberQ[$Packages[[1 ;; -3]], a],
      $Failed, b = StringCases[ToString[FullDefinition[x]], "\n \n" ~~ X__ ~~ " /: " ~~ X__ ~~ "::usage =" ];
      {ToString[x], Select[Sort[DeleteDuplicates[Map[StringTake[#, {4, Flatten[StringPosition[#, " /: "]][[1]] - 1}] &, b]]],
          # ≠ ToString[x] &]}]]
End[]

Begin["`CallsInMeansPackage`"]
CallsInMeansPackage[x_ /; ContextQ[x]] :=
  Module[{a = {{}, {}}, b, c, d = 0}, If[! MemberQ[$Packages, x], $Failed, Map[{b = Quiet[CallsInMean[#]], If[b === $Failed, b = Null,
              If[b[[2]] == {}, AppendTo[a[[1]], #], AppendTo[a[[2]], #]], If[Set[c, Length[b[[2]]]] > d, d = c, Null]]} &, CNames[x]];
      If[a == {{}, {}}, {}, {a, b}]]]
End[]

Begin["`HeadName`"]
HeadName[x_ /; HeadingQ[x] || HeadingQ1[x]] := StringTake[x, {1, StringPosition[x, "[", 1][[1]][[1]] - 1}]
End[]

Begin["`HeadName1`"]
HeadName1[x_ /; StringQ[x]] :=
  Module[{a}, If[! ExpressionQ[ToString[Unique["gs"]] <> x <> ":=74"], $Failed, If[SuffPref[x, "]", 2] &&
          ! SuffPref[x, "[", 1] && SymbolQ[a = StringTake[x, {1, Flatten[StringPosition[x, "["]][[1]] - 1}]], a, $Failed]]]
End[]

Begin["`UnDef`"]
UnDef[x_, y___] := Module[{a = {y}, b = Head2[x]}, If[a ≠ {} && ! HowAct[y], y = b]; If[b === Symbol, True, False]]
End[]

Begin["`RemovedQ`"]
RemovedQ[x_] := Module[{a = DeleteDuplicates[(StringTake[#1, {1, Flatten[StringPosition[#1, "`"]][[1]]}] &) /@ Contexts[]], b},
    If[MemberQ4[Flatten[(NamesContext[#1] &) /@ a], ToString[x]], False, True]]
End[]

Begin["`CsProcsFuncs`"]
CsProcsFuncs[] := Select[CNames["Global`"], ProcQ[#] || FunctionQ[#] &]
End[]

Begin["`ToDefOptPF`"]
ToDefOptPF[x_ /; BlockFuncModQ[x]] := Module[{a, b, c, k = 1, p = ToString[x]}, a = ToExpression["Attributes[" <> p <> "]"];
    b = ToExpression["ClearAllAttributes[" <> p <> "]"]; c = StringSplit[ToString[InputForm[Definition[x]]], "\n \n"];
    While[k ≤ Length[c], ToExpression[StringReplace[c[[k]], Context[x] <> p <> "`" → ""]]; k++];
    ToExpression["SetAttributes[" <> p <> "," <> ToString[Sequence[a]] <> "]"];
    If[Length[c] == 1, c[[1]], c]]
End[]

Begin["`CsProcsFuncs1`"]
CsProcsFuncs1[] := Module[{a = CsProcsFuncs[], b, c}, b = Map[Definition2, ToExpression[a]];
    c = Quiet[Mapp[Select, b, StringFreeQ[#1, ToString[#1] <> "Options[" <> ToString[#1] <> "] := "] &]];
```

```
    Select[Map9[List, a, Map[Length, c]], ! MemberQ[#, "CsProcsFuncs1"] &]]
End[]

Begin["`Args`"]
Args[P_, z___] := Module[{a, b, c, d = {}, k = 1, Vt}, If[CompileFuncQ[P] || BlockFuncModQ[P] || PureFuncQ[P],
      Vt[y_ /; ListQ[y]] := Module[{p = 1, q = {}, t}, While[p ≤ Length[y], q = Append[q, t = ToString[y[[p]]];
            StringTake[t, {1, StringPosition[t, "_"][[1]][[1]] − 1}]]; p++]; q];
      If[CompileFuncQ[P], a = StringSplit[ToString[InputForm[Definition2[P]]], "\n \n"][[1]];
        b = Quiet[SubStrSymbolParity1[a, "{", "}"]];
        b = Select[b, ! StringFreeQ[#, "_"] || ! StringFreeQ[a, " Function[" <> #] &];
        b = Mapp[StringSplit, b, ", "];
        b = Mapp[StringReplace, b, {"{" → "", "}" → ""}]; b = Mapp[Select, b, StringFreeQ[#, "Blank$"] &];
        c = b[[2]]; For[k, k ≤ Length[c], k++, d = Append[d, c[[k]] <> b[[1]][[k]]]]; d = ToExpression[d];
        If[{z} == {}, d, Flatten[Map[Vt, {d}]]], If[BlockFuncModQ[P], a = Flatten[{HeadPF[P]}];
          For[k, k ≤ Length[a], k++,
            d = Append[d, If[{z} ≠ {}, Vt[ToExpression["{" <> StringTake[a[[k]], {StringLength[ToString[P]] + 2, −2}] <> "}"]],
                ToExpression["{" <> StringTake[a[[k]], {StringLength[ToString[P]] + 2, −2}] <> "}"]]]];
          If[Length[d] == 1, d[[1]], d], a = StringTake[StringReplace[ToString[InputForm[Definition2[P]]],
              "Definition2[" → "", 1], {1, −2}];
          If[SuffPref[a, "Function[{", 1], b = SubStrSymbolParity1[a, "{", "}"];
            b = Select[b, ! StringFreeQ[a, "Function[" <> #] &][[1]];
            a = StringSplit[StringTake[b, {2, −2}], ", "], a = StringReplace[a, "#" → "$$$$$"];
            a = Map[ToString, UnDefVars[ToExpression[a]]];
            Map[ToString, ToExpression[Mapp[StringReplace, a, "$$$$$" → "#"]]]]], $Failed]]
End[]

Begin["`Args1`"]
Args1[x_ /; BlockFuncModQ[x]] :=
  Module[{b = 1, c = {}, h = {}, d, p, t, a = Flatten[{PureDefinition[x]}]}, For[b, b ≤ Length[a], b++, t = ToString[Unique["agn"]];
      p = t <> ToString[x];
      ToExpression[t <> a[[b]]];
      d = Unique["avz"];
      AppendTo[h, {ToString[d], p, t}]; AppendTo[c, {Args[p], BlockFuncModQ[p, d]; d}];
      d = ToUpperCase[d]]; Map[Remove, Flatten[h]]; If[Length[c] == 1, c[[1]], c]]
End[]

Begin["`ArgsBFM`"]
ArgsBFM[x_ /; BlockFuncModQ[x], y___] := Module[{a = Flatten[{HeadPF[x]}], b, c = {}, d = {}, n = ToString[x] <> "[", k = 1, p},
    b = Map[ToExpression["{" <> StringTake[#, {StringLength[n] + 1, −2}] <> "}"] &, a];
    c = Map[Map[ToString, #] &, b];
    While[k ≤ Length[c], p = c[[k]];
      AppendTo[d, Map[StringTake[#, {1, Flatten[StringPosition[#, "_"]][[1]] − 1}] &, p]];
      k++];
    If[{y} ≠ {} && ! HowAct[y], y = c]; d]
End[]

Begin["`ArgsBFM1`"]
ArgsBFM1[x_ /; BlockFuncModQ[x]] := Module[{b, c, a = ToString2[Args[x]], d, g = {}, t}, a = If[QmultiplePF[x], a, {a}];
    Do[{b, c, d} = {{}, {}, {}}; t = a[[j]];
      Do[AppendTo[b, If[ListQ[t[[k]]], Map[StringSplit[#, {"/;", "__", "___", "_", ":", "_:", "_."}] &, t[[k]]],
            StringSplit[t[[k]], {"/;", "__", "___", "_", ":", "_:", "_."}]]], {k, 1, Length[t]}];
      Do[AppendTo[c, If[NestListQ[b[[k]]], Map[Map[If[#1 == " " || #1 == "", Nothing, StringTrim[#1]] &, #] &, b[[k]]],
            Map[If[# == " " || #1 == "", Nothing, StringTrim[#]] &, b[[k]]]]], {k, 1, Length[b]}];
      Do[AppendTo[d, If[NestListQ[c[[k]]], Map[If[Length[#1] == 1, {#1[[1]], "Arbitrary"}, #1] &, c[[k]]],
            If[Length[c[[k]]] == 1, {c[[k]][[1]], "Arbitrary"}, c[[k]]]]], {k, 1, Length[c]}]; d = If[Length[d] == 1, d[[1]], d];
      AppendTo[g, d], {j, 1, Length[a]}]; If[Length[g] == 1, g[[1]], g]]
End[]

Begin["`ExtrName`"]
ExtrName[x_ /; StringQ[x], n_ /; IntegerQ[n] && n > 0, m_ /; MemberQ[{−1, 1}, m]] := Module[{a, b, c = "", k,
      d = Flatten[{CharacterRange["a", "z"], CharacterRange["A", "Z"], CharacterRange["𝒶", "𝓏"], CharacterRange["𝒜", "𝒵"],
            CharacterRange["α", "ϑ"], CharacterRange["A", "φ"], "$", CharacterRange["ɑ", "ʒ"], Map[ToString, Range[0, 9]]}]},
    For[k = If[m == 1, n + 1, n − 1], If[m == 1, k ≤ StringLength[x], k ≥ 1], If[m == 1, k++, k−−], a = StringTake[x, {k, k}];
      If[MemberQ[d, a], If[m == 1, c = c <> a, c = a <> c];
        Continue[], Break[]]]; c]
End[]

Begin["`ExtrNames`"]
ExtrNames[x_ /; ProcQ[x]] := Module[{a = BlockToModule[x], b, c, d, f, p = {}, g, k = 1}, {f, a} = {ToString[Locals[x]], Locals1[x]};
```

```
    {b, c} = {HeadPF[x], PureDefinition[x]};  g = StringReplace[c, {b <> " := Module[" → "", ToString[f] <> ", " → ""}];
    d = Map[If[ListQ[#], #[[1]], #] &, StringPosition[g, {" := ", " = "}]];
    For[k, k ≤ Length[d], k++, AppendTo[p, ExtrName[g, d[[k]], -1]]];
    p = Select[p, # ≠ "" &]; {a, Complement[a, p], Complement[p, a]}]
End[]

Begin["`DefOpt`"]
DefOpt[x_ /; StringQ[x]] := Module[{a = If[SymbolQ[x],
          If[SystemQ[x], b = "Null", ToString1[Definition[x]], "Null"]], b, c}, If[! SameQ[a, "Null"], b = Quiet[Context[x]]];
    If[! Quiet[ContextQ[b]], "Null", c = StringReplace[a, b <> x <> "`" -> ""]; ToExpression[c];  c]]
End[]

Begin["`DefOpt1`"]
DefOpt1[x_] := Module[{a = ToString[x], b, c}, If[! SymbolQ[a], $Failed,
      If[SystemQ[x], x, If[ProcQ[a] || FunctionQ[a], b = Attributes[x];
          ClearAttributes[x, b];
          c = StringReplace[ToString1[Definition[x]], Context[a] <> a <> "`" -> ""];  SetAttributes[x, b];  c, $Failed]]]]
End[]

Begin["`DefOptimum`"]
DefOptimum[x_ /; Quiet[ProcQ[x] || FunctionQ[x]]] :=
  Module[{a = "", b = ToString[x], c, d = Context[x], f = Attributes[x], k = 1}, ClearAttributes[x, f];
    Save[b, x];
    For[k = 1, k < Infinity, k++, c = Read[b, String];  If[c === " ", Break[], a = a <> StringReplace[c, StringJoin[d, b, "`"] → ""]]];
    DeleteFile[Close[b]];  Clear[x];  ToExpression[a];  SetAttributes[x, f];  Definition[x]]
End[]

Begin["`OptDefPackage`"]
OptDefPackage[x_ /; ContextQ[x], y___] := Module[{a = {{}, {}}, b}, If[MemberQ[$Packages, x],
      Map[If[StringFreeQ[ToString[Definition[#]], x <> # <> "`"], AppendTo[a[[1]], #], AppendTo[a[[2]], #]] &, CNames[x]];
      If[{y} ≠ {}, Map[{ClearAttributes[#, b = Attributes[#]], ToExpression[PureDefinition[#]], SetAttributes[#, b]} &, a[[2]]], Null];
      a, $Failed]]
End[]

Begin["`SymbolsFromMx`"]
SymbolsFromMx[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y___] :=
  Module[{a = ReadFullFile[x], b = Map[FromCharacterCode, Range[0, 31]], c, d = ContextFromMx[x], h, p = {{}, {}}, t},
    h = StringCases[a, Shortest[d ~~ __ ~~ "`"]];
    h = Sort[DeleteDuplicates[Select[Map[StringTake[#, {StringLength[d] + 1, -2}] &, h], SymbolQ[#] &]]];  c = StringSplit[a, "♂"];
    c = Map[StringTrim[StringReplace[#, GenRules[b, ""]]] &, c];  c = Select[c, StringFreeQ[#, " "] && PrintableASCIIQ[#] &];
    c = Select[c, SymbolQ[#] &];  c = Select[c, {b = StringTake[#, {1, 1}], LetterQ[b] && UpperCaseQ[b]}[[2]] &];
    Map[If[SystemQ[#], AppendTo[p[[2]], #], AppendTo[p[[1]], #]] &, c];
    a = {h, t = Sort[Select[p[[1]], StringFreeQ[#, "`"] &]], Sort[p[[2]]]};
    If[{y} ≠ {} && ! HowAct[y], If[MemberQ[$Packages, d],
        y = {h, MinusList[Sort[DeleteDuplicates[Select[t, Context[#] == d && ! TemporaryQ[#] &]]], h]}, Get[x];
        y = {h, MinusList[Sort[DeleteDuplicates[Select[t, Context[#] == d && ! TemporaryQ[#] &]]], h]};  RemovePackage[d]], Null];
    a]
End[]

Begin["`OptDefinition`"]
OptDefinition[x_ /; Quiet[ProcQ[x] || FunctionQ[x]]] :=
  Module[{c = $Packages, d, b = Definition2[x][[-1]], a = Definition2[x][[1 ;; -2]]}, ClearAllAttributes[x];
    d = Map[StringJoin[#, ToString[x] <> "`"] &, c];
    ToExpression[Map[StringReplace[#, GenRules[d, ""]] &, a]];  SetAttributes[x, b]; Definition[x]]
End[]

Begin["`OptimizedDefPackage`"]
OptimizedDefPackage[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a = ContextFromFile[x], b, c = {{}, {}}}, If[a === $Failed, $Failed, If[MemberQ[$Packages, a], b = 74, Get[x]];
      Quiet[Map[If[StringFreeQ[ToString[Definition[#]], a <> # <> "`"], AppendTo[c[[1]], #],
            {AppendTo[c[[2]], #], Map[ToExpression, Flatten[{PureDefinition[#]}]]}] &, CNames[a]]];
      If[b === 74, Null, RemovePackage[a]];  c]]
End[]

Begin["`OptimizedMfile`"]
OptimizedMfile[x_ /; ContextQ[x], y_ /; FileExtension[y] == "m", z___] :=
  Module[{a = Save[y, x], b = CNames[x], c, c1, d, d1, g, p, t, m = {}, n = {}}, c = ReadString[y];
    t = Map[x <> # <> "`" &, b];  c1 = StringReplace[c, GenRules[t, ""]];
    d = DeleteDuplicates[StringCases[c1, "Attributes[" ~~ X_ ~~ "$] = {Temporary}"]];
```

```
    If[{z} ≠ {} && ! HowAct[z],
      Map[{g = StringReplace[#, {x -> "", "`" -> ""}], If[StringFreeQ[c, #], AppendTo[m, g], AppendTo[n, g]]} &, t];
      d1 = Sort[Flatten[Map[StringTake[p = StringCases[#, "[" ~~ __ ~~ "]"], {2, -2}] &, d]]]; z = {m, n, d1}, Null];
    c = StringReplace[c1, GenRules[d, ""]]; WriteString[y, c]; Close[y]]
End[]

Begin["`ProcActCallsQ`"]
ProcActCallsQ[x_ /; BlockFuncModQ[x], y___] := Module[{a, b, c = {}, d, k = 1, h = "::usage = "}, Save[b = "Art27$Kr20", x];
    For[k, k < Infinity, k++, d = Read[b, String]; If[SameQ[d, EndOfFile], Break[],
        If[! StringFreeQ[d, h], AppendTo[c, StringSplit[StringTake[d, {1, Flatten[StringPosition[d, h]][[1]] - 1}], " /: "][[1]]]]]];
    DeleteFile[
      Close[
        b]];
    c = Select[c, SymbolQ[#] &]; b = If[MemberQ[c, ToString[x]], Drop[c, 1], c]; If[{y} ≠ {} && ! HowAct[{y}[[1]]], {y} = {b}];
    If[b == {}, False, True]]
End[]

Begin["`SubDelStr`"]
SubDelStr[x_ /; StringQ[x], L_ /; ListListQ[L]] :=
  Module[{k = 1, a = {}}, If[! L == Select[L, ListQ[#] && Length[#] == 2 &] || L[[-1]][[2]] > StringLength[x] || L[[1]][[1]] < 1,
      Return[Defer[SubDelStr[x, L]]], For[k, k ≤ Length[L], k++, a = Append[a, StringTake[x, L[[k]]] -> ""]];
      StringReplace[x, a]]]
End[]

Begin["`ListListQ`"]
ListListQ[L_] := If[ListQ[L] && L ≠ {} && Length[L] >= 1 &&
      Length[Select[L, ListQ[#] && Length[#] == Length[L[[1]]] &]] == Length[L], True, False]
End[]

Begin["`RestListList`"]
RestListList[x_ /; ListListQ[x]] := Module[{a, b = Length[x[[1]]], c = Length[x], d = {}}, Do[a = {};
      Do[AppendTo[a, x[[j]][[k]]], {j, 1, c}];
      AppendTo[d, a], {k, 1, b}]; d]
End[]

Begin["`RestListList1`"]
RestListList1[x_ /; ListListQ[x]] := Module[{a = Length[x], b = Length[x[[1]]], c = {}, d, k}, d = Map[k + b * # &, Range[0, a - 1]];
    Do[AppendTo[c, Flatten[x][[d]]], {k, 1, b}]; c]
End[]

Begin["`ListRulesQ`"]
ListRulesQ[x_ /; ListQ[x]] := DeleteDuplicates[Map[RuleQ[#] &, Flatten[x]]] === {True}
End[]

Begin["`AcNb`"]
AcNb[] := StringSplit[NotebookFileName[], {"\\", "/"}][[-1]]
End[]

Begin["`FileDirStForm`"]
FileDirStForm[x_ /; StringQ[x]] := Module[{a = StringReplace[ToLowerCase[x], "/" → "\\"]},
    If[StringLength[a] == 3 && StringTake[a, {2, 2}] == ":", a, StrDelEnds[a, "\\", 2]]]
End[]

Begin["`SetPathSeparator`"]
SetPathSeparator[x_ /; MemberQ[{"/", "\\"}, x]] := Module[{}, Unprotect[$PathnameSeparator];
    $PathnameSeparator = x;
    SetAttributes[$PathnameSeparator, Protected]]
End[]

Begin["`PartialSums`"]
PartialSums[L_ /; ListQ[L] || StringQ[L] && ListQ[ToExpression[L]]] :=
  Module[{a = {}, b = ToExpression[L], k = 1, j}, For[k, k ≤ Length[b], k++, AppendTo[a, Sum[b[[j]], {j, k}]]];
    If[StringQ[L], ToExpression[L <> " =" <> ToString[a]], a]]
End[]

Begin["`ListAssign`"]
ListAssign[x_ /; ListQ[x], y_ /; SymbolQ[y]] := Module[{a = {}, b}, Do[a = Append[a, Unique[y]], {k, Length[x]}];
    b = Map[ToString, a];
    ToExpression[ToString[a] <> "=" <> ToString1[x]];
```

```
    {b, a}]
End[]

Begin["`FullNestListQ`"]
FullNestListQ[x_ /; ListQ[x]] := Module[{a = ListLevels[x][[-1]], b = x, c}, If[a == 0, False, Do[If[! NestListQ[b], c = False;
          Return[], c = True;
          b = Flatten[b, 1]], {a}]; c]]
End[]

Begin["`ListAppValue`"]
ListAppValue[x_ /; ListQ[x], y_] := Quiet[x = PadLeft[{}, Length[x], y]]
End[]

Begin["`LoadFile`"]
LoadFile[F_ /; StringQ[F]] := Module[{a, b, c}, If[! MemberQ[{"nb", "m", "mx", "txt", ""}, FileExtension[F]],
      Return["File <" <> F <> "> has an inadmissible type"], a = Flatten[{FindFile[F]}]; a = If[a === {$Failed}, SearchFile[F], a];
      $Load$Files$ = a]; If[a == {}, Return["File <" <> F <> "> has not been found"],
      Quiet[Check[Get[$Load$Files$[[1]]], c = {$Failed}, {Syntax::sntxc, Syntax::sntxi}]];
      If[c === {$Failed}, "File <" <> $Load$Files$[[1]] <> "> has inadmissible syntax",
        "File <" <> $Load$Files$[[1]] <> "> has been loaded;\n$Load$Files$ defines the list with full paths to the found files."],
      Return["File <" <> F <> "> has not been found"]]]
End[]

Begin["`NbName`"]
NbName[] := Map[StringReplace[#, {"NotebookObject[<<" -> "", "]" -> "", ">>" -> ""}] &, Map[ToString, Notebooks[]]]
End[]

Begin["`AllCurrentNb`"]
AllCurrentNb[] := Module[{a = Notebooks[], b, c = {}, d, k}, b = Flatten[Map[SubsString[#, {"<<", ">>"}, 0] &, Map[ToString, a]]];
    d = Map[Quiet[Check[NotebookFileName[#], "nb has been not saved"]] &, a];
    Do[AppendTo[c, {b[[k]], d[[k]]}], {k, 1, Length[b]}]; c = Select[c, #[[1]] ≠ "Messages" &]; If[Length[c] == 1, Flatten[c], c]]
End[]

Begin["`NotebookSave1`"]
NotebookSave1[x_ /; StringQ[x], y_ /; StringQ[y]] :=
  Module[{a = Notebooks[], b, c = {}, d, k}, b = Flatten[Map[SubsString[#, {"<<", ">>"}, 0] &, Map[ToString, a]]];
    d = Map[Quiet[Check[NotebookFileName[#], $Failed]] &, a];
    Do[AppendTo[c, {b[[k]], d[[k]]}], {k, 1, Length[b]}]; c = If[Length[c] == 1, Flatten[c], c];
    Do[If[c[[k]][[1]] == FileNameTake[x] && c[[k]][[2]] === $Failed, NotebookSave[a[[k]], y];
        Return[y], If[c[[k]][[1]] == FileNameTake[x], NotebookSave[a[[k]], c[[k]][[2]]]; Return[c[[k]][[2]]], Null]], {k, 1, Length[c]}]]
End[]

Begin["`RealProcQ`"]
RealProcQ[x_] := Module[{a, b = " = ", c, d, k = 1, p}, If[! ProcQ[x], False, If[ModuleQ[x], True, a = Locals1[x];
        c = PureDefinition[x]; d = Map[#[[1]] - 1 &, StringPosition[c, b]];
        p = Map[ExprOfStr[c, #, -1, {" ", "{", "["}] &, d]; p = DeleteDuplicates[Flatten[Map[StrToList, p]]]; If[p == a, True, False]]]]
End[]

Begin["`ContOfContext`"]
ContOfContext[x_ /; ContextQ[x]] :=
  Module[{b = {}, c = {}, h, k = 1, a = Select[CNames[x], # ≠ "a" &]}, If[a == {}, $Failed, While[k ≤ Length[a], h = a[[k]];
        If[StringFreeQ[StringReplace[ToString[Definition4[h]], "\\n \\n" → ""], x <> h <> "`"], AppendTo[c, h], AppendTo[b, h]];
        k++];
      {b, c}]]
End[]

Begin["`SearchDir`"]
SearchDir[d_ /; StringQ[d]] :=
  Module[{a = Adrive[], b = "\\" <> ToLowerCase[StringTrim[d, ("\\" | "/") ...]] <> "\\", c, t = {}, g = {}, k = 1, p, v},
    For[k, k ≤ Length[a], k++, p = a[[k]]; c = Map[ToLowerCase, Quiet[FileNames["*", p <> ":\\", Infinity]]];
      Map[If[! StringFreeQ[#, b] || SuffPref[#, b, 2] && DirQ[#], AppendTo[t, #], Null] &, c];
    For[k = 1, k ≤ Length[t], k++, p = t[[k]] <> "\\";
      a = StringPosition[p, b];
      If[a == {}, Continue[], a = Map[#[[2]] &, a];
        Map[If[DirectoryQ[v = StringTake[p, {1, # - 1}]], AppendTo[g, v], Null] &, a]]]; DeleteDuplicates[g]]
End[]

Begin["`SearchFile`"]
SearchFile[F_ /; StringQ[F]] := Module[{a, b, f, dir, h = StringReplace[ToUpperCase[F], "/" → "\\"], k},
```

```
    {a, b, f} = {Map[ToUpperCase[#] <> ":\\" &, Adrive[]], {}, ToString[Unique["d"]] <> ".txt"};
    dir[y_ /; StringQ[y]] := Module[{a, b, c, v}, Run["Dir " <> "/A/B/S " <> y <> " > " <> f]; c = {};
        Label[b]; a = StringReplace[ToUpperCase[ToString[v = Read[f, String]]], "/" → "\\"];
        If[a == "ENDOFFILE", Close[f];
          DeleteFile[f]; Return[c], If[SuffPref[a, h, 2], If[FileExistsQ[v], AppendTo[c, v]];
            Goto[b], Goto[b]]]];
    For[k = 1, k ≤ Length[a], k++, AppendTo[b, dir[a[[k]]]]]; Flatten[b]]
End[]

Begin["`SearchFile1`"]
SearchFile1[x_ /; StringQ[x]] := Module[{a, b, c, d, f = {}, k = 1}, If[PathToFileQ[x], If[FileExistsQ[x], x, {}],
      a = $Path; f = Select[Map[If[FileExistsQ[# <> "\\" <> ToUpperCase[x]], #, "Null"] &, a], # ≠ "Null" &];
      If[f ≠ {}, f, d = Map[# <> ":\\" &, Adrive[]]; For[k, k ≤ Length[d], k++, a = Quiet[FileNames["*", d[[k]], Infinity]];
          f = Join[f, Select[Map[If[FileExistsQ[#] && SuffPref[ToUpperCase[#], "\\" <> ToUpperCase[x], 2], #, "Null"] &, a],
              # ≠ "Null" &]]]; If[f == {}, {}, f]]]]
End[]

Begin["`ExtrPackName`"]
ExtrPackName[F_ /; StringQ[F], N_ /; StringQ[N]] :=
  Module[{a, b, c, d, Art, Kr}, If[FileExistsQ[F] && FileExtension[F] == "m" && StringTake[ToString[ContextFromFile[F]], -1] == "`",
      a = OpenRead[F], Return[$Failed]];
    If[Read[a, String] ≠ "(* ::Package:: *)", Close[a];
      $Failed, {c, d} = {"", StringReplace["(*Begin[\"`Z`\"]*)", "Z" → N]}]; Label[Art];
    b = Read[a, String];
    If[b === EndOfFile, Close[a];
      Return["Definition of " <> N <> " is absent in file <" <> F <> ">"], Null]; If[b ≠ d, Goto[Art], Label[Kr];
      b = StringTake[Read[a, String], {3, -3}];
      c = c <> b <> " "; If[b == "End[]", Close[a]; Return[ToExpression[StringTake[c, {1, -8}]]], Goto[Kr]]]
End[]

Begin["`GV`"]
GV[] := Select[Names["*"], StringTake[#, 1] == "$" &]
End[]

Begin["`SubsDel`"]
SubsDel[S_ /; StringQ[S], x_ /; StringQ[x],
    y_ /; ListQ[y] && DeleteDuplicates[Map[StringQ, y]] == {True} && Plus[Sequences[Map[StringLength, y]]] == Length[y],
    p_ /; MemberQ[{-1, 1}, p]] :=
  Module[{b, c = x, d, h = StringLength[S], k}, If[StringFreeQ[S, x], Return[S], b = StringPosition[S, x][[1]]];
    For[k = If[p == 1, b[[2]] + 1, b[[1]] - 1], If[p == 1, k ≤ h, k >= 1], If[p == 1, k++, k--],
      d = StringTake[S, {k, k}]; If[MemberQ[y, d] || If[p == 1, k == 1, k == h], Break[], If[p == 1, c = c <> d, c = d <> c];
        Continue[]];
    StringReplace[S, c → ""]]
End[]

Begin["`ExpArgs`"]
ExpArgs[f_ /; BlockFuncModQ[f],
    x_ /; ListQ[x] && DeleteDuplicates[Map[! StringFreeQ[ToString[#], "_"] || StringQ[#] &, x]] == {True}] :=
  Module[{a, b, c, d, t, h, g = {}, k = 1}, a = Flatten[{Definition4[ToString[f]]}];
    b = Args[f, 90]; b = If[NestListQ[b], b[[1]], b]; d = Locals1[f];
    d = If[NestListQ[d], d[[1]], d]; c = Flatten[{HeadPF[f]}][[1]]; t = Map[ToString, x];
    h = Map[#[[1]] &, Map[StringSplit[#, "_"] &, t]];
    b = Join[b, d]; While[k ≤ Length[h], If[! MemberQ[b, h[[k]]], d = t[[k]];
        AppendTo[g, If[StringFreeQ[d, "_"], d <> "_", d]]]; k++]; If[g == {}, Return[], g = ToString[g];
      d = StringTake[c, {1, -2}] <> ", " <> StringTake[g, {2, -2}] <> "]"; ClearAllAttributes[f]; ClearAll[f];
      a[[1]] = StringReplace[a[[1]], c -> d, 1]; Map[ToExpression, a];]
End[]

Begin["`RemProcOnHead`"]
RemProcOnHead[x_ /; HeadingQ[x] || HeadingQ1[x] || ListQ[x] && DeleteDuplicates[Map[HeadingQ[#] &, x]] == {True}] :=
  Module[{b, c, d, p, a = HeadName[If[ListQ[x], x[[1]], x]], y},
    If[! MemberQ[Names["*"], a] || ! HowAct[a], $Failed, b = Definition2[a];
      c = b[[1 ;; -2]]; d = b[[-1]]; ToExpression["ClearAttributes[" <> a <> "," <> ToString[d] <> "]"];
      y = Map[StandHead, Flatten[{x}]];
      p = Select[c, ! SuffPref[#, y, 1] &]; ToExpression["Clear[" <> a <> "]"];
      If[p == {}, "Done", ToExpression[p]; ToExpression["SetAttributes[" <> a <> "," <> ToString[d] <> "]"]; "Done"]]
End[]

Begin["`RemProcOnHead1`"]
```

```
RemProcOnHead1[x_ /; HeadingQ[x] || ListQ[x] && DeleteDuplicates[Map[HeadingQ, x]] == {True}] :=
  Module[{a = HeadName[Flatten[{x}][[1]]], b, c},
    c = Flatten[{PureDefinition[a]}]; If[c === {$Failed}, Print["Symbol <" <> a <> "> is undefined."]; "Done",
      b = ToExpression["Attributes[" <> a <> "]"]; c = Select[c, ! SuffPref[#, Flatten[{x}], 1] &];
      ToExpression["ClearAllAttributes[" <> a <> "]"];
      ToExpression["ClearAll[" <> a <> "]"];
      If[c ≠ {}, ToExpression[c];
        ToExpression["SetAttributes[" <> a <> "," <> ToString[b] <> "]"]]; "Done"]]
End[]

Begin["`ComplexQ`"]
ComplexQ[x_] := If[NumberQ[N[x]] && Im[x] ≠ 0, True, False]
End[]

Begin["`Attrib`"]
Attrib[F_ /; StringQ[F],
    x_ /; ListQ[x] && DeleteDuplicates[Map3[MemberQ, {"−A", "−H", "−S", "−R", "+A", "+H", "+S", "+R"}, x]] == {True} || x == {} ||
        x == "Attr"] := Module[{a, b = "attrib ", c, d = " > ", h = "attrib.exe", p, f, g,  t, v},  a = ToString[v = Unique["ArtKr"]];
    If[Set[t, LoadExtProg["attrib.exe"]] === $Failed, Return[$Failed], Null];
    If[StringLength[F] == 3 && DirQ[F] && StringTake[F, {2, 2}] == ":", Return["Drive " <> F],
      If[StringLength[F] == 3 && DirQ[F], f = StandPath[F], If[FileExistsQ1[StrDelEnds[F, "\\", 2], v], g = v;
          f = StandPath[g[[1]]]; Clear[v], Return["<" <> F <> "> is not a directory or a datafile"]]];
    If[x === "Attr", Run[b <> f <> d <> a], If[x === {}, Run[b <> " −A −H −S −R " <> f <> d <> a],
        Run[b <> StringReplace[StringJoin[x], {"+" −> " +", "−" −> " −"}] <> " " <> f <> d <> a]];
    If[FileByteCount[a] == 0, Return[DeleteFile[a]], d = Read[a, String]; DeleteFile[Close[a]]];
    h = StringSplit[StringTrim[StringTake[d, {1, StringLength[d] − StringLength[f]}]]];  Quiet[DeleteFile[t]];
    h = Flatten[h /. {"HR" −> {"H", "R"}, "SH" −> {"S", "H"}, "SHR" −> {"S", "H", "R"}, "SRH" −> {"S", "R", "H"},
          "HSR" −> {"H", "S", "R"}, "HRS" −> {"H", "R", "S"}, "RSH" −> {"R", "S", "H"}, "RHS" −> {"R", "H", "S"}}];
    If[h === {"File", "not", "found", "−"} || MemberQ[h, "C:\\Documents"], "Drive " <> f, {h, g[[1]]}]]
End[]

Begin["`Attrib1`"]
Attrib1[F_ /; StringQ[F],
    x_ /; ListQ[x] && DeleteDuplicates[Map3[MemberQ, {"−A", "−H", "−S", "−R", "+A", "+H", "+S", "+R"}, x]] == {True} ||
        x == {} || x == "Attr", y___] := Module[{a = "$ArtKr$", b = "attrib ", c, d = " > ", h = "attrib.exe",
      p, f, g = Unique["agn"]}, If[LoadExtProg["attrib.exe"] === $Failed, Return[$Failed], Null];
    If[StringLength[F] == 3 && DirQ[F] && StringTake[F, {2, 2}] == ":", Return["Drive " <> F],
      If[StringLength[F] == 3 && DirQ[F], f = StandPath[F], If[FileExistsQ1[StrDelEnds[StringReplace[F, "/" −> "\\"], "\\", 2], g],
          f = StandPath[g[[1]]];  Clear[g], Return["<" <> F <> "> is not a directory or a datafile"]]];
    If[x === "Attr", Run[b <> f <> d <> a], If[x === {}, Run[b <> " −A −H −S −R " <> f <> d <> a],
        Run[b <> StringReplace[StringJoin[x], {"+" −> " +", "−" −> " −"}] <> " " <> f <> d <> a]];
    If[FileByteCount[a] == 0, Return[DeleteFile[a]], d = Read[a, String]; DeleteFile[Close[a]]];
    h = StringSplit[StringTrim[StringTake[d, {1, StringLength[d] − StringLength[f]}]]];
    If[{y} != {}, Quiet[DeleteFile[Directory[] <> "\\" <> "attrib.exe"]], Null];
    h = Flatten[h /. {"HR" −> {"H", "R"}, "SH" −> {"S", "H"}, "SHR" −> {"S", "H", "R"}, "SRH" −> {"S", "R", "H"},
          "HSR" −> {"H", "S", "R"}, "HRS" −> {"H", "R", "S"}, "RSH" −> {"R", "S", "H"}, "RHS" −> {"R", "H", "S"}}];
    If[h === {"File", "not", "found", "−"} || MemberQ[h, "C:\\Documents"], "Drive " <> f, h]]
End[]

Begin["`Attribs`"]
Attribs[x_ /; FileExistsQ[x] || DirectoryQ[x], y___] := Module[{b, a = StandPath[x], c = "attrib.exe", d = ToString[Unique["g"]], df},
    If[DirQ[x] && StringLength[x] == 3 && StringTake[x, {2, 2}] == ":", $Failed,
      df[] := Quiet[DeleteFile[Directory[] <> "\\" <> c]];  If[! FileExistsQ[c], LoadExtProg[c]];
      If[{y} == {}, Run[c <> " " <> a <> " > ", d];
        df[];
        b = Characters[StringReplace[StringTake[Read[d, String], {1,  −StringLength[a] − 1}], " " −> ""]];
        DeleteFile[Close[d]];
        b, a = Run[c <> " −A −H −R −S " <> a];
        df[]; a]]]
End[]

Begin["`DelAllAttribs`"]
DelAllAttribs[x_ /; FileExistsQ[x] || DirectoryQ[x]] :=
  Module[{a = StandPath[x], b, c = "Attrib.exe"}, If[DirQ[x] && StringLength[x] == 3 && StringTake[x, {2, 2}] == ":",
      $Failed, If[! FileExistsQ[$InstallationDirectory <> "\\" <> c], LoadExtProg[c]];
      Run[c <> " −A −H −R −S " <> a]]]
End[]

Begin["`DirQ`"]
```

```
DirQ[d_ /; StringQ[d]] := DirectoryQ[StringReplace[d, "/" → "\\"]]
End[]

Begin["`DirName`"]
DirName[F_ /; StringQ[F]] := If[DirQ[F], "None", If[! FileExistsQ1[F], $Failed, Quiet[Check[FileNameJoin[FileNameSplit[F][[1;
            −2]]], "None"]]]]
End[]

Begin["`FileQ`"]
FileQ[f_ /; StringQ[f]] := Module[{d = Adrive[], s = {}, k = 1, a = ToLowerCase[StringReplace[Flatten[OpenFiles[]], "\\\\" −> "/"]],
    b = ToLowerCase[StringReplace[Directory[], "\\" −> "/"]], c = ToLowerCase[StringReplace[f, "\\" −> "/"]]},
  For[k, k ≤ Length[d], k++, AppendTo[s, d[[k]] <> ":"]];
  If[StringLength[c] < 2 || ! MemberQ[ToLowerCase[s], StringTake[c, {1, 2}]], c = b <> "/" <> c, Null];
  If[DirQ[c], False, If[MemberQ[a, c], True, If[Quiet[OpenRead[c]] === $Failed, False, Close[c]; True]]]]
End[]

Begin["`CopyDir`"]
CopyDir[d_ /; StringQ[d], p_ /; StringQ[p]] := CopyDirectory[d, If[DirQ[p], p <> "\\" <> FileNameSplit[d][[−1]], p]]
End[]

Begin["`SequenceQ`"]
SequenceQ[s_ /; StringQ[s]] := Module[{a, b = " := ", c = " = ", d = Quiet[ToString[Definition[s]]]},
  Quiet[Check[If[StringTake[StringReplace[d, {s <> b → "", s <> c → ""}], {1, 9}] == "Sequence[", True, False], False]]]
End[]

Begin["`DirQ`"]
DirQ[d_ /; StringQ[d]] := DirectoryQ[StringReplace[d, "/" → "\\"]]
End[]

Begin["`StrExprQ`"]
StrExprQ[x_ /; StringQ[x]] := Module[{a = True}, a && Quiet[Check[ToExpression[x], a = False]]; a]
End[]

Begin["`ExpressionQ`"]
ExpressionQ[x_ /; StringQ[x]] := Module[{a, b, c, f = "$Art$Kr$.mx"}, If[SyntaxQ[x], b = SymbolsOfString[x, 90];
    If[b != {}, ToExpression["DumpSave[\"$Art$Kr$.mx\"" <> "," <> ToString1[b] <> "]"], Null];
    c = Quiet[Check[ToExpression[x], a]]; Quiet[{Get[f], DeleteFile[f]}]; If[SameQ[c, a], False, True], False]]
End[]

Begin["`ExprsInStrQ`"]
ExprsInStrQ[x_ /; StringQ[x], y___] :=
  Module[{a = {}, c = 1, d, j, b = StringLength[x], k = 1}, For[k = c, k ≤ b, k++, For[j = k, j ≤ b, j++, d = StringTake[x, {k, j}];
      If[! SymbolQ[d] && SyntaxQ[d], AppendTo[a, d]]]; c++];
   a = Select[Map[StringTrim, Map[StringTrim2[#, {"−", "+", " "}, 3] &, a]], ExpressionQ[#] &];
   If[a == {}, False, If[{y} ≠ {} && ! HowAct[{y}[[1]]], y = DeleteDuplicates[a]]; True]]
End[]

Begin["`ToStringRule`"]
ToStringRule[x_ /; ListQ[x] && DeleteDuplicates[Map[Head, x]] == {Rule} || Head[x] == Rule] :=
  Module[{a = Flatten[{x}], b = {}, c, k = 1}, While[k ≤ Length[a], c = a[[k]];
    b = Append[b, ToString[RhsLhs[c, "Lhs"]] → "(" <> ToString[RhsLhs[c, "Rhs"]] <> ")"];
    k++]; If[ListQ[x], b, b[[1]]]]
End[]

Begin["`ToStringRule1`"]
ToStringRule1[x_ /; ListQ[x] && DeleteDuplicates[Map[Head, x]] == {Rule} || Head[x] == Rule] :=
  Module[{a = Flatten[{x}], b}, b = Map[ToString[#[[1]]] −> ToString[#[[2]]] &, a];
   If[Length[b] == 1, b[[1]], b]]
End[]

Begin["`CALL`"]
CALL[x_, y___] := Module[{a = ToString[x, InputForm], b, c}, b = StringTake[a, Flatten[StringPosition[a, "["]][[1]] − 1];
   If[! MemberQ[{"Null", $Failed}, Definition4[b]],
     x, c = If[{y} === {}, Directory[], If[StringQ[y] && DirectoryQ[y], y, Directory[]]] <> "\\" <> b <> ".txt";
     If[! FileExistsQ[c], $Failed,  ToExpression[ReadList[c, "String"][[−1]]]; x]]]
End[]

Begin["`CALLmx`"]
CALLmx[y_, z_ /; MemberQ[{1, 2}, z], d___] :=
```

```
  Module[{a = If[{d} === {}, Directory[], If[StringQ[d] && DirectoryQ[d], d, Directory[]]],
      b = Map[ToString, If[ListQ[y], y, {y}]], c = {}, h, k = 1, s},
    If[z == 1, While[k ≤ Length[b], s = b[[k]]; h = a <> "\\" <> s <> ".mx";
        If[! MemberQ[{"Null", $Failed}, Definition4[s]],
          ToExpression["DumpSave[" <> ToString1[h] <> "," <> ToString[s] <> "]"]; AppendTo[c, s]]; k++]; Prepend[c, a],
      While[k ≤ Length[b], s = b[[k]];
        h = a <> "\\" <> s <> ".mx";
        If[FileExistsQ[h], Get[h];
          AppendTo[c, s]];
        k++]; c]]
End[]

Begin["`CALLmxH`"]
CALLmxH[y_, z_ /; MemberQ[{1, 2}, z], help_, d___] :=
  Module[{a = If[{d} === {}, Directory[], If[StringQ[d] && DirectoryQ[d], d, Directory[]]],
      b = Map[ToString, If[ListQ[y], y, {y}]], c = {}, h, k = 1, s, t},
    t = a <> "\\" <> ToString[help] <> ".mx";
    If[z == 1, While[k ≤ Length[b], s = b[[k]]; h = a <> "\\" <> s <> ".mx";
        If[! MemberQ[{"Null", $Failed}, Definition4[s]], ToExpression["DumpSave[" <> ToString1[h] <> "," <> ToString[s] <> "]"];
          ToExpression["DumpSave[" <> ToString1[t] <> "," <> ToString1[help] <> "]"]; AppendTo[c, s]]; k++]; Prepend[c, a],
      While[k ≤ Length[b], s = b[[k]];
        h = a <> "\\" <> s <> ".mx";
        If[FileExistsQ[h], Get[h];
          Get[t];
          help[];
          AppendTo[c, s]];
        k++]; c]]
End[]

Begin["`Table1`"]
Table1[L_ /; ListListQ[L], x_] := Module[{a = {}, c = L, d = {}, k = 1, b = Length[L]},
    If[ListListQ[L] && Length[L[[1]]] == 2, For[k, k ≤ b, k++, AppendTo[a, L[[k]][[1]]];
        AppendTo[d, L[[k]][[2]]]]; {a, d} = Map[DeleteDuplicates, {a, d}];
      If[x === "index", a, If[x === "entry", d, If[ListQ[x] && Length[x] == 2,
            If[! MemberQ[a, x[[1]]], AppendTo[c, x], Select[Map[If[#1[[1]] === x[[1]] && ! SameQ[x[[2]], Null],
                    x, If[#[[1]] === x[[1]] && x[[2]] === Null, Null, #]] &, L], ! SameQ[#, Null] &]],
            Quiet[Check[Select[Map[If[#[[1]] === x, #[[2]]] &, L], ! SameQ[#, Null] &][[1]], $Failed]]]], $Failed]]
End[]

Begin["`TabLib`"]
TabLib[Lib_ /; FileExistsQ[Lib] && FileExtension[Lib] == "mx", x_, y___] :=
  Module[{a = Get[Lib], b, c}, If[MemberQ[{"index", "entry"}, x], Table1[LibBase, x], Map[ToExpression, LibBase[[1]][[2]]];
      If[ListQ[x] && Length[x] == 2,
        c = If[SameQ[x[[2]], Null], x, {x[[1]], PureDefinition[x[[1]]]}]; b = Table1[LibBase, c];
        If[! SameQ[b, $Failed], LibBase = b;
          ToExpression["DumpSave[" <> ToString1[Lib] <> "," <> "LibBase]"]],
        If[StringQ[x] && ! StringFreeQ[x, "::usage = "], c = Quiet[LibBase[[1]][[2]] = AppendTo[LibBase[[1]][[2]], x]];
          LibBase = ReplacePart[LibBase, {1, 2} → c];
          ToExpression["DumpSave[" <> ToString1[Lib] <> "," <> "LibBase]"],
          If[Table1[LibBase, x] === $Failed, $Failed, b = Table1[LibBase, x];
            If[! SameQ[b, $Failed], ToExpression[b]; x[y]], $Failed]]]]
End[]

Begin["`UsageBase`"]
UsageBase[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y___] := Module[{a = Get[x], b, c = 6, d, k = 1},
    If[{y} == {}, Map[ToExpression, BaseHelp];,
      If[Length[{y}] == 1 && SameQ[y, "?"], BaseHelp,
        If[Length[{y}] == 1 && SymbolQ[y],
          BaseHelp = Select[BaseHelp, StringTake[#1, {1, Flatten[StringPosition[#, "::usage"]][[2]]}] ≠ ToString[y] <> "::usage" &];
          ToExpression["DumpSave[" <> ToString1[x] <> "," <> "BaseHelp]"];,
          If[Length[{y}] == 1 && StringQ[y] && ! StringFreeQ[y, {"::usage=", "::usage ="}],
            If[BaseHelp == {}, AppendTo[BaseHelp, y],
              b = StringTake[y, {1, Flatten[StringPosition[y, "::usage"]][[2]]}];
              For[k, k ≤ Length[BaseHelp], k++, d = BaseHelp[[k]];
                If[StringTake[d, {1, Flatten[StringPosition[d, "::usage"]][[2]]}] === b, BaseHelp[[k]] = y; c = 90; Break[]]];
              If[c ≠ 90, AppendTo[BaseHelp, y]]];
            ToExpression["DumpSave[" <> ToString1[x] <> "," <> "BaseHelp]";]]]]]
End[]
```

```
Begin["`ClearOut`"]
ClearOut[x_ /; PosIntQ[x] || PosIntListQ[x]] := Module[{a = Flatten[{x}], k = 1}, Unprotect[Out];
    For[k, k ≤ Length[a], k++, (Out[a[[k]]] =.];
    Protect[Out];]
End[]

Begin["`RemovePF`"]
RemovePF[x_ /; HeadingQ1[x] || ListQ[x] && DeleteDuplicates[Map[HeadingQ1, x]] == {True}] :=
  Module[{b, c = {}, d, p, k = 1, j, a = DeleteDuplicates[Map[HeadName, Flatten[{x}]]]},
    b = Map[If[UnevaluatedQ[Definition2, #], {"90", {}}, Definition2[#]] &, a];
    For[k, k ≤ Length[a], k++, p = b[[k]];  AppendTo[c, Select[Flatten[
          {p[[1 ;; -2]], "SetAttributes[" <> a[[k]] <> ", " <> ToString[p[[-1]]] <> "]"}], ! SuffPref[#, Map[StandHead, x], 1] &]]];
    Map[ClearAllAttributes, a]; Map[Remove, a]; Map[ToExpression, c]; a = Definition2[b = HeadName[x]];
    If[a[[1]] === "Undefined", ToExpression["ClearAttributes[" <> b <> "," <> ToString[a[[2]]] <> "]"], Null]]
End[]

Begin["`ClearValues`"]
ClearValues[x_ /; ListQ[x], y___] :=
  Select[Map[If[{y} == {}, Remove, ClearAll], Select[Names["`*"], MemberQ[x, ToExpression[#]] &]], # ≠ "Null" &]
End[]

Begin["`ClearContextVars`"]
ClearContextVars[x_ /; MemberQ[Select[Contexts[], StringCount[#, "`"] == 1 &], x], y___] :=
  Map[If[Context[#] == x, ClearAttributes[#, Protected];
      If[{y} ≠ {}, ClearAll, Remove][#];  #, Nothing] &, Names["`*"]]
End[]

Begin["`VarsValues`"]
VarsValues[x_ /; ListQ[x]] := Select[Names["`*"], MemberQ[x, ToExpression[#]] &]
End[]

Begin["`FileFormat1`"]
FileFormat1[x_ /; StringQ[x]] := Module[{a}, If[FileExistsQ[x], {x, FileFormat[x]}, a = SearchFile[x];
    If[a == {}, {}, a = Map[{#, FileFormat[#]} &, a];  If[Length[a] == 1, a[[1]], a]]]
End[]

Begin["`FileFormat2`"]
FileFormat2[x_ /; StringQ[x]] := Module[{a, b = {}, c, k = 1}, If[StringLength[x] == 3,
    If[MemberQ[{":/", ":\\"}, StringTake[x, -2]] && MemberQ[Adrive[], ToUpperCase[StringTake[x, 1]]], Return["Directory"],
      Null], If[DirectoryQ[x], Return["Directory"], a = SearchFile[x]];  If[a == {}, Return[{}], For[k, k ≤ Length[a], k++, c = a[[k]];
        AppendTo[b, {c, FileFormat[c]}]]]];  If[Length[b] == 1, b[[1]], b]]
End[]

Begin["`FileFormat3`"]
FileFormat3[x_ /; FileExistsQ[x], t___] := Module[{b, c, a = FileExtension[x]},
  If[a ≠ "", ToUpperCase[a], c = If[Quiet[StringTake[Read[x, String], {1, 5}]] === "%PDF-", {Close[x], "PDF"}[[-1]], Close[x];
      b = ReadFullFile[x];
      If[! StringFreeQ[b, {"MSWordDoc", "Microsoft Office Word"}], "DOC", If[! StringFreeQ[b, ".opendocument.textPK"], "ODT",
        If[! StringFreeQ[b, {"!DOCTYPE HTML ", "text/html"}], "HTML", If[MemberQ3[Range[0, 255], DeleteDuplicates[
            Flatten[Map[ToCharacterCode[#] &, DeleteDuplicates[Characters[b]]]]]], "TXT", Undefined]]]];
    If[{t} ≠ {}, Quiet[Close[x]];  RenameFile[x, x <> "." <> c], c]]
End[]

Begin["`QSaveGUI`"]
QSaveGUI[x_] := If[FileExistsQ[x] && FileExtension[x] == "m", SuffPref[ReadFullFile[x], "(* ::Package:: *)", 1], False]
End[]

Begin["`LoadExtProg`"]
LoadExtProg[x_ /; StringQ[x], y___] :=
  Module[{a = Directory[], b = Unique["agn"], c, d, h}, If[PathToFileQ[x] &&  FileExistsQ[x], CopyFileToDir[x, Directory[]],
    If[PathToFileQ[x] &&  ! FileExistsQ[x], $Failed,  d = a <> "\\" <> x; If[FileExistsQ[d], d,
      h = FileExistsQ1[x, b];  If[h, CopyFileToDir[b[[1]], a];
        If[{y} == {}, AppendTo[$Path, FileNameJoin[FileNameSplit[b[[1]]][[1 ;; -2]]], y = b];  d, $Failed]]]]]
End[]

Begin["`CallsInProc`"]
CallsInProc[P_ /; BlockFuncModQ[P]] := Module[{b, c = {}, k = 1, a = ToString[FullDefinition[P]], TN},
  TN[S_ /; StringQ[S], L_ /; ListQ[L] && Length[Select[L, IntegerQ[#] &]] == Length[L] && L ≠ {}] := Module[
    {a1 = "", c1, b1 = {}, k1, p = 1}, For[p, p ≤ Length[L], p++, For[k1 = L[[p]] - 1, k1 ≠ 0, k1--, c1 = StringTake[S, {k1, k1}];
```

```
            a1 = c1 <> a1; If[c1 === " ", a1 = StringTake[a1, {2, -1}];
              If[Quiet[Check[Symbol[a1], False]] === False, a1 = "";
                Break[], AppendTo[b1, a1];
                a1 = "";
                Break[]]]]; b1];
    b = TN[a, b = DeleteDuplicates[Flatten[StringPosition[a, "["]]]][[2 ;; -1]];
    b = Sort[DeleteDuplicates[Select[b,
          StringFreeQ[#, "`"] && ! MemberQ[{"Block", ToString[P], "Module"}, #] && ToString[Definition[#]] ≠ "Null" &]]];
    k = Select[b, SystemQ[#] &];
    c = MinusList[b, Flatten[{k, ToString[P]}]]; {k, c, DeleteDuplicates[Map[Context, c]]}
End[]

Begin["`ProcLocals`"]
ProcLocals[x_ /; ProcQ[x], z___] := Module[{a = DefOpt[ToString[x]], b = ArtKr, c, m = 1, n = 0, k, p, h = ""}, Clear[ArtKr];
    ProcQ1[ToString[x], ArtKr]; c = StringPosition[a, ArtKr <> "["]; ArtKr = b; k = c[[1]][[2]] + 1;
    While[m != n, p = StringTake[a, {k, k}]; If[p == "{", m++; h = h <> p, If[p == "}", n++; h = h <> p, h = h <> p]]; k++];
    If[{z} ≠ {} && Definition1[{z}[[1]]] === "Null", z = k + 2]; "{" <> h]
End[]

Begin["`ProcBody`"]
ProcBody[x_ /; BlockFuncModQ[x]] :=
  Module[{c, p, d = {}, k = 1, a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}]}, While[k ≤ Length[a], p = a[[k]];
      c = Mapp[Rule, Map[b[[k]] <> " := " <> # &, {"Block[", "Module[", ""}], ""]; c = StringReplace[p, c, 1];
      AppendTo[d, If[BlockModQ[x], StringTake[StringReplace[c, SubStrSymbolParity1[c, "{", "}"][[1]] <> ", " -> "", 1], {1, -2}], c]];
      k++];
    If[Length[d] == 1, d[[1]], d]]
End[]

Begin["`FindFileContext`"]
FindFileContext[x_ /; ContextQ[x], y___] :=
  Module[{b = {}, c = "", d = StringJoin["BeginPackage[", StrStr[x], "]"], s = {}, k = 1, j = 1, a = If[{y} == {}, Adrive[], {y}],
  f = "$Kr20_Art27$.txt"}, While[k <= Length[a], Run["Dir ", StringJoin[a[[k]], ":\\*.*"], StringJoin[" /A/B/O/S > ", f]];
   While[! c === EndOfFile, c = Read[f, String];
        If[! DirQ[c] && FileExtension[c] == "m", AppendTo[b, c]]; j++]; c = ""; j = 1; k++]; k = 1;
  While[k <= Length[b], c = ToString[ReadFullFile[b[[k]]]]; If[! StringFreeQ[c, d], AppendTo[s, b[[k]]]]; k++];
    DeleteFile[Close[f]]; s]
End[]

Begin["`FindFileContext1`"]
FindFileContext1[x_ /; ContextQ[x]] := Module[{a = FindFileContext[x], b = If[MemberQ[$Packages, x], "Current", {}]},
    If[a ≠ {} && ! SameQ[b, {}], {b, a}, If[a ≠ {} && SameQ[b, {}], a, If[a == {} && ! SameQ[b, {}], b, {}]]]]
End[]

Begin["`ContextInFile`"]
ContextInFile[x_ /; ContextQ[x], y___] := Module[{b, d, h, Tav, c = "$Art27Kr20$"}, If[{y} ≠ {} && DirQ[y],
      Run["DIR " <> StandPath[y] <> "/A/B/O/S > $Art27Kr20$"], Run["DIR C:\\  /A/B/O/S > $Art27Kr20$"]];
    d = ReadList[c, String]; DeleteFile[c];
    Tav[t_ /; ListQ[t]] := Module[{m, v = {}, k, z, a = "BeginPackage[" <> ToString1[x] <> "]"},
        Map[If[FileExistsQ[#] && MemberQ[{"cdf", "nb", "m", "mx", "tr"}, FileExtension[#]],
            If[MemberQ[{"tr", "m"}, FileExtension[#]] && ! StringFreeQ[ReadFullFile[#], a],
              AppendTo[v, #], If[MemberQ[{"cdf", "nb"}, FileExtension[#]], {m, h, k} = {0, "", 1};
                For[k, k < Infinity, k++, h = Read[#, String];
                  If[h === EndOfFile, Close[#];
                    Break[], If[! StringFreeQ[h, "BeginPackage"] && ! StringFreeQ[h, x], m = 90;
                      Close[#];
                      Break[], Continue[]]]];
                If[m == 90, AppendTo[v, #], Null],
                If[FileExtension[#] == "mx", z = StringPosition[ReadFullFile[#], {"CONT", "ENDCONT"}];
                  If[! StringFreeQ[StringTake[ReadFullFile[#], {z[[1]][[1]], z[[2]][[1]]}], " " <> x <> " "]; AppendTo[v, #], Null]]],
            Null] &, t]; v]; Tav[d]]
End[]

Begin["`FindFileObject`"]
FindFileObject[x_ /; ! SameQ[ToString[DefOpt[ToString[x]]], "Null"], y___] :=
  Module[{b = {}, c = "", s = {}, d, k = 1, a = If[{y} == {}, Adrive[], {y}], f = "ArtKr",
      h = "(*Begin[\"`" <> ToString[x] <> "`\"]*)", p = "(*" <> ToString[x] <> "::usage=", t}, While[k ≤ Length[a],
      Run["Dir ", a[[k]] <> ":\\", " /B/S/L > " <> f]; While[! SameQ[c, "EndOfFile"], c = ToString[Read[f, String]];
        If[StringTake[c, {-2, -1}] == ".m", AppendTo[b, c]]; Continue[]]; Quiet[Close[f]]; c = ""; k++]; k = 1;
    While[k ≤ Length[b], If[Select[ReadList[b[[k]], String], ! StringFreeQ[#, h] && StringFreeQ[#, p] &] ≠ {}, AppendTo[s, b[[k]]]];
```

```
      k++];
    {DeleteFile[f], s}[[2]]]
End[]

Begin["`NamesFromMx`"]
NamesFromMx[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a = ContextFromFile[x], b}, If[MemberQ[$ContextPath, a], CNames[a], Get[x];
      b = CNames[a]; Map[Close1[2, a <> #] &, b]; $ContextPath = MinusList[$ContextPath, {a}]; b]]
End[]

Begin["`NamesFromMx1`"]
NamesFromMx1[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{c, d = {}, p, h = "", k = 1, j, m, n, a = ContextFromFile[x], b = ToString[ReadFullFile[x]]},
    b = StringJoin[Map[FromCharacterCode, Select[ToCharacterCode[b], # > 32 && # < 128 &]]];
    {n, m} = Map[StringLength, {a, b}]; c = Map[#[[1]] + n &, StringPosition[b, a]][[2 ;; -1]];
    While[k ≤ Length[c], For[j = c[[k]], j ≤ m, j++, p = StringTake[b, {j, j}]; If[p == "`", AppendTo[d, h];
          h = "";
          Break[], h = h <> p]];
      k++]; Sort[MinusList[Select[d, SymbolQ[#] &], {"Private"}]]]
End[]

Begin["`NamesFromMx2`"]
NamesFromMx2[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] := Module[{a = ToString[ReadFullFile[x]], b},
    b = Select[ToCharacterCode[a], # == 255 || (# > 31 && # < 123 && ! MemberQ[Flatten[{Range[37, 47], Range[91, 95]}], #]) &];
    b = ReduceList[b, 255, 1, 1]; b = Select[Quiet[SplitList[b, 96]], # ≠ {} &]; b = Quiet[Map[FromCharacterCode, b]];
    b = DeleteDuplicates[Select[b, SymbolQ[#] &]];
    Sort[Select[b, ! MemberQ[{"Private", "System"}, #] && StringFreeQ[#, {StringTake[ContextFromFile[x], {1, -2}], "ÿ"}] &]]]
End[]

Begin["`SuffPref`"]
SuffPref[S_ /; StringQ[S], s_ /; StringQ[s] || ListQ[s]  && DeleteDuplicates[Map[StringQ, s]] == {True},
    n_ /; MemberQ[{1, 2, 3}, n]] := Module[{a, b, c, k = 1}, If[StringFreeQ[S, s], False, b = StringLength[S];
      c = Flatten[StringPosition[S, s]]; If[n == 3 && c[[1]] == 1 && c[[-1]] == b, True,
        If[n == 1 && c[[1]] == 1, True, If[n == 2 && c[[-1]] == b, True, False]]]]]
End[]

Begin["`CDir`"]
CDir[d_ /; StringQ[d]] :=
  Module[{a}, Quiet[If[StringTake[d, {2, 2}] == ":", If[MemberQ[a, StringTake[d, 1]], CreateDirectory[d], a = Adrive[];
          CreateDirectory[Sort[a, FreeSpaceVol[#1] ≥ FreeSpaceVol[#2] &][[1]] <> StringTake[d, {2, -1}]]], CreateDirectory[d]]]
End[]

Begin["`Map2`"]
Map2[F_ /; SymbolQ[F], c_ /; ListQ[c], d_ /; ListQ[d]] := Map[Symbol[ToString[F]][#, Sequences[d]] &, c]
End[]

Begin["`Map1`"]
Map1[x_ /; ListQ[x] && SameQ[DeleteDuplicates[Map[SymbolQ[#] &, x]], {True}], y_List] :=
  Map[Symbol[ToString[#]][Sequences[y]] &, x]
End[]

Begin["`PalindromeQ1`"]
PalindromeQ1[x_ /; StringQ[x]] := If[x == StringReverse[x], True, False]
End[]

Begin["`MaximalPalindromicSubstring`"]
MaximalPalindromicSubstring[x_ /; StringQ[x]] :=
  Module[{b = {}, c, k = 1, a = Select[Map[StringJoin, Subsets[Characters[x]]], # ≠ "" &]},
    For[k, k ≤ Length[a], k++, If[PalindromeQ[a[[k]]], b = Append[b, a[[k]]], Null]];
    c = Sort[Map[StringLength, b]][[-1]]; Select[b, StringLength[#] == c &]]
End[]

Begin["`SubsBstr`"]
SubsBstr[S_ /; StringQ[S], x_ /; CharacterQ[x], y_ /; CharacterQ[y]] :=
  Module[{a = {}, c, h, n, m, s = S, p, t}, c[s_, p_, t_] := DeleteDuplicates[Map10[StringFreeQ, s, {p, t}]] == {False};
    While[c[s, x, y], n = StringPosition[s, x, 1][[1]][[1]]; s = StringTake[s, {n, -1}]; m = StringPosition[s, y, 1];
      If[m == {}, Return[], m = m[[1]][[1]]]; AppendTo[a, h = StringTake[s, {1, m}]]; s = StringReplace[s, h → ""]; Continue[]]; a]
End[]
```

```
Begin["`Map4`"]
Map4[F_ /; SymbolQ[F], L_ /; ListQ[L], x_] := Map[Symbol[ToString[F]][#, x] &, L]
End[]

Begin["`Levels`"]
Levels[x_, h_ /; ToString[Definition[h]] == "Null"] := Module[{a = {}, b, k = 1}, While[k < Infinity, b = Level[x, k];
      If[a == b, Break[], a = b];
      k++];
    h = k - 1; a]
End[]

Begin["`Aobj`"]
Aobj[x_ /; FileExistsQ[x] && StringTake[x, -2] == ".m",
    y_ /; SymbolQ[y] || ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}] := Module[
    {a, b = "(*", c = "*)", d = $AobjNobj, p = {Read[x, String], Close[x]}[[1]], h = Mapp[StringJoin, Map[ToString, Flatten[{y}]], "["],
      k, j, g, s, t = {}, v = {}}, If[p ≠ "(* ::Package:: *)", $Failed, a = ReadFullFile[x];
      If[StringFreeQ[a, d], $Failed, a = StringSplit[a, d][[2 ;; -1]];
        a = Map[StringReplace[#, {b -> "", c -> ""}] &, a]; a = Select[a, SuffPref[#, h, 1] &];
        For[k = 1, k ≤ Length[h], k++, g = h[[k]]; For[j = 1, j ≤ Length[a], j++, s = a[[j]]; c = StrSymbParity[s, g, "[", "]"];
            c = If[c == {}, False, HeadingQ1[Quiet[ToString[ToExpression[c[[1]]]]]] || HeadingQ[c[[1]]]];
            If[SuffPref[s, g, 1] && c, AppendTo[t, s];
              AppendTo[v, StringTake[g, {1, -2}]]]]]; Map[ToExpression, t];
        If[v ≠ {}, Print["Software for " <> ToString[v] <> " is downloaded"],
          Print["Software for " <> ToString[Flatten[{y}]] <> " was not found"]]]]
End[]

Begin["`Aobj1`"]
Aobj1[x_ /; FileExistsQ[x] && StringTake[x, -2] == ".m",
    y_ /; SymbolQ[y] || ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}] :=
  Module[{a, c = "*)(*", d = $AobjNobj, k, t = {}, g = {}, h = Map[ToString, Flatten[{y}]], p, j = 1, v},
    a = StringSplit[ReadFullFile[x], d][[2 ;; -1]];
    a = Map[StringTake[#, {3, -3}] &, a]; For[j, j ≤ Length[h], j++, p = h[[j]];
      For[k = 1, k ≤ Length[a], k++, If[SuffPref[a[[k]], Map[StringJoin[p, #] &, {"[", "=", ":"}], 1],
          AppendTo[t, StringReplace[a[[k]], c -> ""]]; AppendTo[g, p], Null]]]; v = {t, MinusList[h, g]};
    If[v[[1]] ≠ {}, ToExpression[v[[1]]]; Print["Software for " <> ToString[g] <> " is downloaded"], Null];
    If[v[[2]] ≠ {}, Print["Software for " <> ToString[v[[2]]] <> " was not found"], Null]]
End[]

Begin["`Nobj`"]
Nobj[x_ /; FileExistsQ[x] && StringTake[x, -2] == ".m", y_ /; ! HowAct[y]] := Module[{a, b, c, d, p, h, t, k = 1},
    If[FileExistsQ[x] && MemberQ[{"Table", "Package"}, Quiet[FileFormat[x]]], {a, b, d, h} = {OpenRead[x], {}, "90", {}};
      While[! SameQ[d, "EndOfFile"], d = ToString[Read[a, String]];
        If[! SuffPref[d, " ", 1], If[! StringFreeQ[d, "::usage = \""],
            AppendTo[b, StringSplit[StringTake[d, {1, Flatten[StringPosition[d, "::usage"]][[1]] - 1}], " /: "][[1]]],
            p = Quiet[Check[StringTake[d, {1, Flatten[StringPosition[d, {" := ", " = "}]][[1]] - 1}], $Failed]];
            If[! SameQ[p, $Failed], If[SymbolQ[p] && StringFreeQ[p, {" ", "{", "`"}] ||
                  StringFreeQ[p, {" ", "{", "`"}] && HeadingQ1[p] === True, AppendTo[b, p]]]];
        k++];
      Close[
        a];
      b = Sort[DeleteDuplicates[b]];
      h = Select[b, ! SymbolQ[#] &]; t = Map[If[SymbolQ[#], #, HeadName[#]] &, h];
      b = MinusList[b, h];
      b = Sort[DeleteDuplicates[Join[b, t]]]; y = MinusList[
          Sort[DeleteDuplicates[Join[h, Select[Map[If[! UnevaluatedQ[HeadPF, #], HeadPF[#]] &, b], ! SameQ[#, Null] &]]]], b];
      b, $Failed]]
End[]

Begin["`FullUserTools`"]
FullUserTools[x_ /; BlockFuncModQ[x], y___] := Module[{a, b, c, d, p = {}, n = {}}, Save[Set[a, ToString[x] <> ".txt"], x];
    b = ReadString[a];
    DeleteFile[a];
    c = StringSplit[b, "\r\n \r\n"]; b = Select[c, ! StringFreeQ[#, "::usage = \""] &];
    d = MinusList[c, b]; c = Map[StringSplit[#, " /: ", 2][[1]] &, b]; d = Map[StringSplit[#, " := "][[1]] &, d];
    Quiet[Map[If[HeadingQ[#], AppendTo[p, HeadName[#]], AppendTo[n, #]] &, d]]; {a, p} = {Join[c, p], MinusList[c, p]};
    b = Map[MinusList[#, {ToString[x]}] &, {a, p}][[1]]; b = DeleteDuplicates[Map[{#, Context[#]} &, b]];
    b = Gather[b, #1[[2]] == #2[[2]] &];
    b = Map[Sort[DeleteDuplicates[Flatten[#]]] &, b]; d = Map[Sort[#, ContextQ[#1] &] &, b];
    d = Map[Flatten[{#[[1]], Sort[#[[2 ;; -1]]]}] &, d]; d = If[Length[d] == 1, d[[1]], d]; If[{y} != {} && ! HowAct[y], y = {p, n};
```

```
      d, d]]
End[]

Begin["`FullToolsCalls`"]
FullToolsCalls[x_ /; BlockFuncModQ[x]] := Module[{a = Flatten[{PureDefinition[x]}][[1]], b, c = {}, d, k = 1, g = {}, p, j, n},
    b = Gather[Map[#[[1]] &, StringPosition[a, "["], Abs[#1 − #2] == 1 &]; b = Flatten[Select[b, Length[#] == 1 &]];
    For[k, k ≤ Length[b], k++, n = "";
      For[j = b[[k]] − 1, j ≥ 0, j−−,
        If[SymbolQ[p = Quiet[StringTake[a, {j}]]] || IntegerQ[Quiet[ToExpression[p]]], n = p <> n, AppendTo[c, n];
          Break[]]]];
    c = MinusList[c, Join[Locals[x], Args[x, 90], {"Block", "Module"}]];
    c = Map[{#, Quiet[Context[#]]} &, MinusList[c, {ToString[x]}]]; b = Map[Sort[DeleteDuplicates[Flatten[#]]] &, c];
    d = Map[Flatten, Gather[Map[Sort[#, Quiet[ContextQ[#1]] &] &, b], #1[[1]] == #2[[1]] &]];
    d = Map[Flatten[{#[[1]], Sort[#[[2 ;; −1]]]}] &, Map[DeleteDuplicates, d]];
    d = If[Length[d] == 1, d[[1]], d]; Select[d, ! Quiet[SameQ[#[[1]], Context[""]]] &]]
End[]

Begin["`FullToolsCallsM`"]
FullToolsCallsM[x_ /; BlockFuncModQ[x]] := Module[{a = Flatten[{PureDefinition[x]}], b, c = {}, k = 1, n = ToString[x]},
    If[Length[a] == 1, FullToolsCalls[x],
      For[k, k ≤ Length[a], k++, b = ToString[Unique["sv"]];
        ToExpression[StringReplace[a[[k]], n <> "[" → b <> "[", 1]];
        AppendTo[c, FullToolsCalls[b]];
        Quiet[Remove[b]]]; c = Map[If[NestListQ[#] && Length[#] == 1, #[[1]], #] &, c];
      Map[If[! NestListQ[#] && Length[#] == 1, {}, If[NestListQ[#] && Length[#] > 1, #, Select[#, Length[#] > 1 &]]] &, c]]]
End[]

Begin["`AllCalls`"]
AllCalls[x_ /; BlockFuncModQ[x]] := Module[{a1, ArtKr, k1, b1, c1 = {}, d1, m = ToString[x]},
    a1 = Flatten[{PureDefinition[x]}];
    ArtKr[y_] := Module[{a = Flatten[{PureDefinition[y]}][[1]], b, c = {}, d, k, g = {}, p, j, n},
        b = Gather[Map[#[[1]] &, StringPosition[a, "["], Abs[#1 − #2] == 1 &]; b = Flatten[Select[b, Length[#] == 1 &]];
        For[k = 1, k ≤ Length[b], k++, n = "";
          For[j = b[[k]] − 1, j ≥ 0, j−−,
            If[SymbolQ[p = Quiet[StringTake[a, {j}]]] || IntegerQ[Quiet[ToExpression[p]]], n = p <> n, AppendTo[c, n];
              Break[]]]];
        For[k = 1, k ≤ Length[b], k++, For[j = b[[k]], j ≤ StringLength[a], j++,
            SubStrSymbolParity1[StringTake[a, {j, StringLength[a]}], "[", "]"][[1]];
            AppendTo[g, SubStrSymbolParity1[StringTake[a, {j, StringLength[a]}], "[", "]"][[1]]]; Break[]]];
        n = Select[Map[StringJoin[#] &, Partition[Riffle[c, g], 2]], # ≠ HeadPF[y] &]; If[FunctionQ[y], n, n[[2 ;; −1]]]];
    If[Length[a1] == 1, ArtKr[x], For[k1 = 1, k1 ≤ Length[a1], k1++, b1 = ToString[Unique["v"]];
        ToExpression[StringReplace[a1[[k1]], m <> "[" → b1 <> "[", 1]];
        AppendTo[c1, ArtKr[b1]]; Quiet[Remove[b1]]]; c1]]
End[]

Begin["`SetDir`"]
SetDir[x_ /; StringQ[x]] := Module[{a, b, c}, If[StringLength[x] == 1 || StringLength[x] ≥ 2 && StringTake[x, {2, 2}] ≠ ":",
      Quiet[SetDirectory[Quiet[CreateDirectory[StringReplace[Directory[] <> "\\" <> x, "\\\\" → "\\"]]]]],
      If[Run["Dir " <> StringTake[x, {1, 2}]] ≠ 0, $Failed, b = FileNameSplit[x]; c = b[[1]];
        Do[c = c <> "\\" <> b[[k]]; Quiet[CreateDirectory[c]], {k, 2, Length[b]}]; SetDirectory[c]]]
End[]

Begin["`SetDir1`"]
SetDir1[x_ /; StringQ[x]] := Module[{a = Quiet[Check[SetDirectory[x], $Failed]], b, c}, If[! SameQ[a, $Failed], a, b = Adrive[];
      c = Map[FreeSpaceVol, b]; c = SortNL1[c, 2, Greater]; SetDir[StringJoin[c[[1]][[1]], StringTake[x, {2, −1}]]], a]]
End[]

Begin["`Adrive`"]
Adrive[] := Module[{a, b, c, d}, {a, b} = {CharacterRange["A", "Z"], {}};
    Do[ d = Directory[]; c = a[[k]];
      AppendTo[b, If[Quiet[SetDirectory[c <> ":\\"]] === $Failed, Nothing, SetDirectory[d]; c]], {k, 1, Length[a]}]; Sort[b]]
End[]

Begin["`Adrive1`"]
Adrive1[] := Module[{a = CharacterRange["A", "Z"], b = {}, c, d = {}, h}, Do[c = a[[k]];
      If[DirQ[c <> ":\\"], AppendTo[b, c], Null], {k, 1, Length[a]}];
    Do[h = Directory[]; c = a[[k]];
      AppendTo[d, If[Quiet[SetDirectory[c <> ":\\"]] === $Failed, Nothing, SetDirectory[h]; c]], {k, 1, Length[a]}];
    Map[Sort, {d, MinusList[b, d]}]]
```

```
End[]

Begin["`ListPosition`"]
ListPosition[x_ /; ! NestListQ1[x], y_ /; ListQ[y]] :=
  Module[{a = {}, c = Length[y], k = 1}, While[k ≤ c, AppendTo[a, Flatten[Position[x, y[[k]]]]];
      k++]; a]
End[]

Begin["`Save1`"]
Save1[x_ /; StringQ[x], y_ /; DeleteDuplicates[Map[StringQ, Flatten[{y}]]][[1]]] :=
  Module[{Rs, t = Flatten[{y}], k = 1}, Rs[n_, m_] := Module[{b, c = ToString[Unique[b]],
      a = If[SymbolQ[m], Save[n, m], If[StringFreeQ[m, "["], $Failed, StringTake[m, {1, Flatten[StringPosition[m, "["]][[1]] - 1}]]]},
    If[a === Null, Return[], If[a === $Failed, Return[$Failed], If[SymbolQ[a], b = DefFunc3[a], Return[$Failed]]]];
    If[Length[b] == 1, Save[n, a], b = Select[b, SuffPref[#, m, 1] &]];
    If[b ≠ {}, b = c <> b[[1]], Return[$Failed]];
    ToExpression[b];
    a = c <> a; ToExpression["Save[" <> ToString1[n] <> "," <> ToString1[a] <> "]"];
    BinaryWrite[n, StringReplace[ToString[StringJoin[Map[FromCharacterCode, BinaryReadList[n]]]], c -> ""]];
    Close[n];]; For[k, k ≤ Length[t], k++, Rs[x, t[[k]]]]]
End[]

Begin["`Save2`"]
Save2[x_ /; StringQ[x], y_ /; SymbolQ[y] || ListQ[y]] := If[FileExistsQ[x], Save[x, "\\r\\n \\r\\n"];
    Save[x, y], Save[x, y]]
End[]

Begin["`DuplicateLocalsQ`"]
DuplicateLocalsQ[P_ /; BlockModQ[P], y___] := Module[{a, b = Locals1[P]}, If[b == {}, False, b = If[NestListQ[b], b[[1]], b];
      a = Select[Gather2[b], #[[2]] > 1 &]; If[a == {}, False, If[{y} ≠ {} && ! HowAct[y], y = a]; True]]]
End[]

Begin["`DuplicateLocals`"]
DuplicateLocals[x_ /; BlockModQ[x]] :=
  Module[{a = Locals1[x]}, a = Select[Map[{#, Count[a, #]} &, DeleteDuplicates[a]], #[[2]] > 1 &];
    a = Sort[a, ToCharacterCode[#1[[1]]][[1]] < ToCharacterCode[#2[[1]]][[1]] &]; If[Length[a] > 1 || a == {}, a, a[[1]]]]
End[]

Begin["`DelDuplLocals`"]
DelDuplLocals[x_ /; BlockQ[x] || ModuleQ[x], y___] := Module[{a = Locals[x], b = {}, c = {}, d, p},
    If[a == {}, x, d = Attributes[x];
      ClearAttributes[x, d]; Map[If[StringFreeQ[#, "="], AppendTo[b, #], AppendTo[c, #]] &, a];
      b = DeleteDuplicates[b]; c = DeleteDuplicates[Map[StringSplit[#, " = ", 2] &, c], #1[[1]] == #2[[1]] &];
      p = Map[#[[1]] &, c];
      b = Select[b, ! MemberQ[p, #] &]; c = Map[StringJoin[#[[1]], " = ", #[[2]]] &, c];
      b = StringRiffle[Join[b, c], ", "];
      p = PureDefinition[x]; ToExpression[StringReplace[p, StringRiffle[a, ", "] → b, 1]];
      SetAttributes[x, d]; If[{y} ≠ {} && ! HowAct[y], y = MinusList[a, StrToList[b]], Null]; x]]
End[]

Begin["`DelDuplLocalsM`"]
DelDuplLocalsM[x_ /; BlockModQ[x], y___] := Module[{a = Flatten[{PureDefinition[x]}], h = ToString[x], b, c = {}, d, p, z = {}},
    If[Length[a] == 1, DelDuplLocals[x, y],
      If[{y} ≠ {} && ! HowAct[y] || {y} === {}, b = Attributes[x]; ClearAttributes[x, b], Return[Defer[DelDuplLocalsM[x, y]]]];
      Map[{AppendTo[c, d = ToString[Unique["vgs"]]], ToExpression[StringReplace[#, h <> "[" → d <> "[", 1]]} &, a];
      p = Map[PureDefinition[#] &, Map[{DelDuplLocals[#, y], Quiet[AppendTo[z, y]], Clear[y]}[[1]] &, c]];
      ToExpression[Map[StringReplace[#, GenRules[Map[# <> "[" &, c], h <> "["], 1] &, p]];
      SetAttributes[x, b]; Map[Remove[#] &, c]; If[{y} ≠ {}, y = z, Null]; x]]
End[]

Begin["`ReplaceLocals`"]
ReplaceLocals[x_ /; BlockModQ[x], y_ /; DeleteDuplicates[Map[StringQ, Flatten[{y}]]] == {True}] :=
  Module[{a = Locals[x], b = Flatten[{y}], c = PureDefinition[x], d, p, h = HeadPF[x], k, j, f},
    f[z_ /; StringQ[z]] := If[StringFreeQ[z, "="], z, StringTrim[StringTake[z, {1, Flatten[StringPosition[z, "="]][[1]] - 1}]]];
    d = StringReplace[c, h <> " := " -> ""];
    d = StringTake[d, {If[ModuleQ[x], 8, 7], -1}]; p = SubStrSymbolParity1[d, "{", "}"][[1]];
    For[k = 1, k ≤ Length[a], k++, For[j = 1, j ≤ Length[b], j++, If[f[a[[k]]] == f[b[[j]]], a[[k]] = b[[j]]]]];
    ToExpression[StringReplace[c, p -> ToString[a], 1]]]
End[]
```

```
Begin["`OptimalLocals`"]
OptimalLocals[x_ /; BlockQ[x] || ModuleQ[x]] := Module[{a = DelDuplLocals[x], at, b, c, d, p, t, h, k, j, f}, a = Locals[x];
    b = RedundantLocals[x]; If[StringQ[b], Return[b], Null];
    If[a == {} || b == {}, x, t = a; c = PureDefinition[x]; h = HeadPF[x]; at = Attributes[x]; ClearAttributes[x, at];
      f[z_ /; StringQ[z]] := If[StringFreeQ[z, "="], z, StringTrim[StringTake[z, {1, Flatten[StringPosition[z, "="]][[1]] - 1}]]];
      d = StringReplace[c, h <> " := " → ""]; d = StringTake[d, {If[ModuleQ[x], 8, 7], -1}]; p = SubStrSymbolParity1[d, "{", "}"][[1]];
      For[k = 1, k ≤ Length[b], k++, For[j = 1, j ≤ Length[t], j++, If[f[t[[j]]] == b[[k]], t = Delete[t, j]]]];
      ToExpression[StringReplace[c, p → ToString[t], 1]]; SetAttributes[x, at]; x]]
End[]

Begin["`OptimalLocalsM`"]
OptimalLocalsM[x_ /; BlockFuncModQ[x]] := Module[{a = PureDefinition[x], atr = Attributes[x],
      b = ClearAllAttributes[x], c = FromCharacterCode[6], d = {}, p = 1, t = {}, h = ToString[x]},
    If[StringQ[a], OptimalLocals[h]; SetAttributes[x, atr]; Definition[x],
      b = Map[{AppendTo[d, h <> c <> ToString[p++]],
            ToExpression[StringReplace[#, h <> "[" → h <> c <> ToString[p - 1] <> "[", 1]]} &, a];
      b = Select[DeleteDuplicates[Flatten[b]], ! SameQ[#, Null] &];
      Map[If[BlockQ[#] || ModuleQ[#], OptimalLocals[#], Null] &, b];
      Map[ToExpression, Map[StringReplace[PureDefinition[#], GenRules[# <> "[", h <> "["], 1] &, b]]; Map[ClearAll, b];
      SetAttributes[x, atr]; Definition[x]]]
End[]

Begin["`OptimalLocalsN`"]
OptimalLocalsN[x_ /; BlockModQ[x]] := Module[{b, c, d, h, t, a = SubsProcs[x], k = 1}, If[a === {}, OptimalLocals[x];
      Definition[x], b = Attributes[x];
      ClearAllAttributes[x];
      c = PureDefinition[x];
      d = Flatten[SubsProcs[x]]; t = Map[StringTake[#, {1, Flatten[StringPosition[#, "["]][[1]] - 1}] &, d];
      h = Select[t, ! MemberQ[{"Undefined", $Failed}, PureDefinition[#]] &];
      If[h ≠ {}, ToExpression["Save[\"$Art27Kr18$.m\"," <> ToString[h] <> "]"]; Map[ClearAllAttributes, h]; Map[Clear, h], Null];
      For[k, k ≤ Length[d], k++, ToExpression[d[[k]]];
        OptimalLocals[t[[k]]]; c = StringReplace[c, d[[k]] → PureDefinition[t[[k]]], 1]; ToExpression["Clear[" <> t[[k]] <> "]"]];
      ToExpression[c];
      OptimalLocals[x]; Quiet[Check[Get["$Art27Kr18$.m"], Null]];
      Quiet[Check[DeleteFile["$Art27Kr18$.m"], Null]]; SetAttributes[x, b]; Definition[x]]]
End[]

Begin["`ProcUniToMod`"]
ProcUniToMod[x_ /; BlockModQ[x]] := Module[{a = Flatten[{PureDefinition[x]}], b = Attributes[x], c = ClearAllAttributes[x]},
    Clear[x];
    c = {};
    Map[{ToExpression[#], OptimalLocalsN[x], AppendTo[c, PureDefinition[x]], Clear[x]} &, a];
    Map[ToExpression[StringReplace[#, " := Block[{" → " := Module[{"]] &, c]; SetAttributes[x, b]; Definition[x]]
End[]

Begin["`ReplaceSubProcs`"]
ReplaceSubProcs[x_ /; BlockModQ[x], y_ /; ListQ[y] && Length[y] == 2 || ListListQ[y] && Length[y[[1]]] == 2, z___] :=
  Module[{a = Flatten[SubsProcs[x]], b, c, h, g, f, k}, If[a == {}, OptimalLocals[x];
      Definition[x], b = Attributes[x]; ClearAllAttributes[x]; f = PureDefinition[x]; c = SubProcs[x]; Map[ClearAll[#] &, c[[2]]];
      h = Map[HeadName[#] &, c[[1]]][[2 ;; -1]]; c = Select[If[ListListQ[y], y, {y}],
          BlockModQ[#[[1]]] && BlockFuncModQ[#[[2]]] && MemberQ[h, ToString[#[[1]]]] || #[[2]] == "" &];
      c = ToStringRule1[Map[#[[1]] → #[[2]] &, c]]; c = If[RuleQ[c], {c}, c];
      g = {};
      For[k = 1, k ≤ Length[c], k++, AppendTo[g,
          Select[a, SuffPref[#, Part[c[[k]], 1] <> "[", 1] &][[1]] → If[Part[c[[k]], 2] == "", "", PureDefinition[Part[c[[k]], 2]]]]];
       ToExpression[StringReplace[StringReplace[f, g], {" , " -> "", " ; " -> ""}]];
      If[{z} ≠ {}, Map[ClearAllAttributes, h]; Map[ClearAll, h], Null]; OptimalLocalsN[x]; SetAttributes[x, b]; Definition[x]]]
End[]

Begin["`UpdateContextPaths`"]
UpdateContextPaths[x_ /; StringQ[x] || ListStringQ[x]] :=
  Module[{}, $ContextPath = DeleteDuplicates[Flatten[ReleaseHold[Map[AppendTo[$ContextPath, #] &, Flatten[{x}]]]]]; ]
End[]

Begin["`UpdatePackages`"]
UpdatePackages[x_ /; StringQ[x] || ListStringQ[x]] := Module[{}, ClearAttributes[$Packages, Protected];
    $Packages = DeleteDuplicates[Flatten[ReleaseHold[Map[AppendTo[$Packages, #] &, Flatten[{x}]]]]];
    SetAttributes[$Packages, Protected];]
End[]
```

```
Begin["`LoadMyPackage`"]
LoadMyPackage[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_] :=
  Module[{a, Cn, Ts, k = 1}, Ts[g_] := Module[{p = "$Art26Kr18$.txt", b = "", c, d, v = 1}, Write[p, g];
        Close[p];
        While[v < Infinity, c = Read[p, String]; If[SameQ[c, EndOfFile], Close[p];
            DeleteFile[p]; Return[b], b = b <> c]; Continue[]]];
    Cn[t_] := Module[{s = Names[StringJoin[t, "*"]], b}, b = Select[s, Quiet[ToString[Definition[ToString[#1]]] ≠ "Null" &]];
    Quiet[Get[x]];
    a = Cn[y];
    While[k ≤ Length[a],
      Quiet[ToExpression[StringReplace[StringReplace[Ts[ToExpression["Definition[" <> a[[k]] <> "]"]], y → ""], a[[k]] <> "`" → ""]]];
      k++]]
End[]

Begin["`LoadPackage`"]
LoadPackage[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a}, Quiet[ToExpression["Off[shdw::Symbol]"]; Get[x]; a = ToExpression["Packages[][[1]]"];
      ToExpression["LoadMyPackage[" <> "\"" <> x <> "\"" <> "," <> "\"" <> a <> "\"" <> "]"];
      ToExpression["On[shdw::Symbol]"]]]
End[]

Begin["`MfileEvaluate`"]
MfileEvaluate[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y_ /; MemberQ[{1, 2, 3, 4}, y]] := Module[{a, b, c, d, h},
    a = StringCases[ReadString[x], "BeginPackage[\"" ~~ Shortest[___] ~~ "`\"]"];
    a = Map[Quiet[Check[StringTake[#, {15, -3}], Null]] &, a];
    a = Select[a, Complement[Characters[#], Join[CharacterRange["A", "Z"], CharacterRange["a", "z"], {"`"}]] == {} &];
    If[MemberQ[{{}, {"AladjevProcedures`" }}, a], $Failed,
      Quiet[NotebookEvaluate[NotebookOpen[x, Visible → False]]]; b = a[[1]];
      h = Names[b <> "*"]; c = Sort[DeleteDuplicates[Map[StringReplace[#, b -> ""] &, h]]];
      If[y == 1, c, If[y == 2, d = {{}, {}}; Map[If[ToString[Definition[#]] != "Null", AppendTo[d[[1]], #], AppendTo[d[[2]], #]] &, h];
          If[d[[1]] == {}, Null, c = Map[Quiet[ToExpression[StringReplace[DefToString[#], {b -> "", # <> "`" -> ""}]]] &, d[[1]]]];
          Map[StringReplace[#, b -> ""] &, d], If[y == 3, b, Map[{Print[#], ToExpression["?" <> ToString[#]]} &, h]; ]]]]
End[]

Begin["`Usages`"]
Usages[x_ /; StringQ[x], y___] := Module[{a, b, h = ""}, If[! FileExistsQ[x], Put[x], Null];
    If[{y} == {} && ! EmptyFileQ[x], While[! SameQ[h, EndOfFile], Quiet[ToExpression[h = Read[x, Expression]]]];
      Close[x];, If[{y} == {} && EmptyFileQ[x], $Failed, If[Quiet[Check[ListQ[y], False]] && {y} ≠ {} && ListSymbolQ[y],
          a = DeleteDuplicates[Select[y, Head[#::usage] === String &]];
          If[a ≠ {}, PutAppend[Sequences[Map[ToString[#] <> "::usage = " <> "\"" <> #::usage <> "\"" &, a]], x], $Failed],
          If[! Quiet[Check[ListQ[y], False]], b = DeleteDuplicates[Reverse[ReadList[x, Expression]]];
            Put[Sequences[Select[b, ! SuffPref[#1, Map[ToString[#] <> "::usage" &, Flatten[{y}]], 1] &]], x], $Failed]]]]
End[]

Begin["`Usages1`"]
Usages1[x_ /; ContextQ[x]] := DeleteDuplicates[Map[{Print[#], ToExpression["?" <> #]} &, CNames[x]]][[1]]
End[]

Begin["`UsagesMNb`"]
UsagesMNb[x_ /; FileExistsQ[x] && MemberQ[{"m", "nb"}, FileExtension[x]]] :=
  Module[{a, b, c}, If[FileExtension[x] == "m", a = Select[ReadList[x, String], ! StringFreeQ[#, "::usage="] &];
      a = Map[StringTake[#, {3, -3}] &, a];
      a = Map[If[SymbolQ[StringTake[#, {1, Flatten[StringPosition[#, "::usage="]][[1]] - 1}]], #] &, a];
      Select[a, ! SameQ[#, Null] &], c = "$.m"; b = ContextFromFile[x];
      ToExpression["Save[" <> StrStr[c] <> ", " <> StrStr[b] <> "]"];
      b = Select[Quiet[ReadList["$.m", Expression]], ! MemberQ[{Null, {Temporary}}, #] &];
      DeleteFile["$.m"]; b]]
End[]

Begin["`UpdatePath`"]
UpdatePath[x_ /; StringQ[x] || ListStringQ[x]] := Module[{}, ClearAttributes[$Path, Protected];
    $Path = DeleteDuplicates[Flatten[AppendTo[$Path, x]]];
    SetAttributes[$Path, Protected]]
End[]

Begin["`StreamFiles`"]
StreamFiles[] := Module[{a = Map[ToString1, StreamsU[]], b = {}, w = {"out"}, r = {"in"}, c, k = 1},
    If[a == {}, Return["AllFilesClosed"], For[k, k ≤ Length[a], k++, c = a[[k]];
```

```
        If[SuffPref[c, "Out", 1], AppendTo[w, StrFromStr[c]], AppendTo[r, StrFromStr[c]]]]];
    c = Select[Map[Flatten, {r, w}], Length[#] > 1 &]; If[Length[c] == 1, c[[1]], c]]
End[]

Begin["`Need`"]
Need[x__] :=
  Module[{a = Directory[], c, p, d = {x}[[1]], f, b = If[Length[{x}] > 1 && StringQ[{x}[[2]]], {x}[[2]], "Null"]}, If[! ContextQ[d], $Failed,
      If[b == "Null", Quiet[Check[Get[d], $Failed]],
        If[b ≠ "Null" && ! MemberQ[{"m", "mx"}, FileExtension[b]] || ! FileExistsQ[b],
          $Failed, If[MemberQ[$Packages, d], True, CopyFile[b, f = a <> "\\" <> FileNameSplit[b][[-1]]];
            Get[f]; DeleteFile[f]; True]]]]
End[]

Begin["`DefaultsQ`"]
DefaultsQ[x_ /; BlockFuncModQ[x], y___] :=
  Module[{a = Args[x], b = {"_.", "_:"}, c = {}, d, k = 1}, a = Map[ToString, If[NestListQ[a], a[[1]], a]];
    While[k ≤ Length[a], d = a[[k]];
      If[! StringFreeQ[d, b[[1]]], AppendTo[c, b[[1]]], If[! StringFreeQ[d, b[[2]]], AppendTo[c, b[[2]]]]]; k++];
    If[c == {}, False, If[{y} ≠ {} && ! HowAct[y], y = DeleteDuplicates[Flatten[c]]]; True]]
End[]

Begin["`FileOpenQ`"]
FileOpenQ[F_ /; StringQ[F]] :=
  Module[{A, a = FileType[F], b, d, x = inputstream, y = outputstream, c = Map[ToString1, StreamsU[]],
      f = ToLowerCase[StringReplace[F, "\\" → "/"]]}, A[t_] := Module[{a1 = ToString1[t], b1 = StringLength[ToString[Head[t]]]},
        ToExpression["{" <> StrStr[Head[t]] <> "," <> StringTake[a1, {b1 + 2, -2}] <> "}"]];
    If[MemberQ[{Directory, None}, a], $Failed, Clear[inputstream, outputstream];
      d = ToExpression[ToLowerCase[StringReplace[ToString1[Map[A, StreamsU[]]], "\\\\" → "/"]]];
      a = Select[d, #[[2]] === f &];
      If[a == {}, {inputstream, outputstream} = {x, y}; False,
        a = {ReplaceAll[a, {inputstream → "read", outputstream → "write"}], {inputstream, outputstream} = {x, y}}[[1]]]]; If[
      Length[a] == 1, a[[1]], a]]
End[]

Begin["`FileOpenQ1`"]
FileOpenQ1[F_ /; StringQ[F]] := Module[{a = StreamFiles[], b, c, d, k = 1, j},
    If[a === "AllFilesClosed", Return[False], c = StringReplace[ToLowerCase[F], "/" -> "\\"];
      b = Mapp[StringReplace, Map[ToLowerCase, a], "/" -> "\\"]]; For[k, k ≤ 2, k++,
      For[j = 2, j ≤ Length[b[[k]]], j++, If[Not[SuffPref[b[[k]][[j]], c, 2] || SuffPref[b[[k]][[j]], "\\" <> c, 2]], a[[k]][[j]] = Null;
          Continue[], Continue[]]]];
    b = Mapp[Select, a, ! # === Null &]; b = Select[b, Length[#] > 1 &];
    If[Length[b] == 1, b[[1]], b]]
End[]

Begin["`FunCompose`"]
FunCompose[L_ /; ListQ[L], x_] := Module[{a, k = 2}, a = L[[1]]@x; For[k, k ≤ Length[L], k++, a = L[[k]]@a];
    a]
End[]

Begin["`MixCaseQ`"]
MixCaseQ[x_ /; StringQ[x]] := Module[{a, b, k}, {a, b} = {Characters[x], {}};
    For[k = 1, k ≤ Length[a], k++, If[! LetterQ[a[[k]]], Null, b = Append[b, If[UpperCaseQ[a[[k]]], 1, 2]]]];
    b = Length[DeleteDuplicates[b]]; If[b == 0, "Special Characters", If[b == 1, False, True]]]
End[]

Begin["`Closes`"]
Closes[x_] := Quiet[Check[Close[x], Null]]
End[]

Begin["`CloseAll`"]
CloseAll[] := Map[Close, StreamsU[]]
End[]

Begin["`Close1`"]
Close1[x___String] :=
  Module[{a = Streams[][[3 ;; -1]], b = {x}, c = {}, k = 1, j}, If[a == {} || b == {}, {}, b = Select[{x}, FileExistsQ[#] &];
      While[k ≤ Length[a], j = 1;
        While[j ≤ Length[b], If[ToUpperCase[StringReplace[a[[k]][[1]], {"\\" → "", "/" → ""}]] ==
              ToUpperCase[StringReplace[b[[j]], {"\\" → "", "/" → ""}]], AppendTo[c, a[[k]]]]; j++]; k++];
```

```
      Map[Close, c]; If[Length[b] == 1, b[[1]], b]]]
End[]

Begin["`Close2`"]
Close2[x___String] := Module[{a = Streams[][[3 ;; -1]], b = {}, c, d = Select[{x}, StringQ[#] &]},
    If[d == {}, {}, c[y_] := ToLowerCase[StringReplace[y, "/" -> "\\"]]; Map[AppendTo[b, Part[#, 1]] &, a];
      d = DeleteDuplicates[Map[c[#] &, d]]; Map[Close, Select[b, MemberQ[d, c[#]] &]]]]
End[]

Begin["`StreamsU`"]
StreamsU[] := Streams[][[3 ;; -1]]
End[]

Begin["`UprocQ`"]
UprocQ[x_ /; SymbolQ[x]] := Module[{a = Unique["agn"], b}, If[SingleDefQ[x], b = ProcQ1[x, a];
      {b, a[[2]][[1]]}, False]]
End[]

Begin["`Mapp`"]
Mapp[F_ /; ProcQ[F] || SysFuncQ[F] || SymbolQ[F], Expr_, x___] :=
  Module[{a = Level[Expr, 1], b = {x}, c = {}, h, g = Head[Expr], k = 1},
    If[b == {}, Map[F, Expr], h = Length[a];
      For[k, k ≤ h, k++, AppendTo[c, ToString[F] <> "[" <> ToString1[a[[k]]] <> ", " <> ListStrToStr[Map[ToString1, {x}]] <> "]"]];
      g @@ Map[ToExpression, c]]]
End[]

Begin["`Mapp1`"]
Mapp1[F_ /; SymbolQ[F], L_ /; ListQ[L]] := Module[{a = Attributes[F], b}, SetAttributes[F, Listable];
    b = Map[F, L];
    ClearAllAttributes[F];
    SetAttributes[F, a]; b]
End[]

Begin["`MapInSitu`"]
MapInSitu[x_, y_ /; StringQ[y]] := ToExpression[y <> "=" <> ToString[Map[x, ToExpression[y]]]]
End[]

Begin["`MapInSitu1`"]
MapInSitu1[x_, y_] := ToExpression[ToString[Args[MapInSitu, 78]] <> "=" <> ToString[Map[x, y]]]
End[]

Begin["`MapInSitu2`"]
MapInSitu2[x_, y_] := Module[{a = Map[x, y], b = ToString[y], c = Select[Names["`*"], StringFreeQ[#, "$"] &], d = {}, k = 1, h},
    For[k, k ≤ Length[c], k++, h = c[[k]];
      If[ToString[ToExpression[h]] === b, d = Append[d, h], Null]];
    For[k = 1, k ≤ Length[d], k++, h = d[[k]]; ToExpression[h <> " = " <> ToString[a]]];
    a]
End[]

Begin["`ToList`"]
ToList[expr_] :=
  Module[{a, b, c = {}, d, k = 1, n}, If[StringQ[expr], Characters[expr], If[ListQ[expr], expr, a = ToString[InputForm[Map[b, expr]]];
      d = StringSplit[a, ToString[b] <> "["];
      For[k, k ≤ Length[d], k++, n = d[[k]];
        c = Append[c, StringTake[n, {1, Flatten[StringPosition[n, "]"]][[-1]] - 1}]]];
      ToExpression[c]]]]
End[]

Begin["`FunctionQ`"]
FunctionQ[x_] := If[StringQ[x], PureFuncQ[ToExpression[x]] || QFunction1[x], PureFuncQ[x] || QFunction[x]]
End[]

Begin["`QFunction1`"]
 QFunction1[x_ /; StringQ[x]] := Module[{a, c = ToString[Unique["agn"]], b, p, d = {}, h = {}, k = 1},
    If[UnevaluatedQ[Definition2, x], False, If[SysFuncQ[x], False, a = Definition2[x][[If[Options[x] == {}, 1 ;; -2, 1 ;; -3]]];
      For[k, k ≤ Length[a], k++, p = c <> ToString[k]; AppendTo[h, p <> x];
        ToExpression[p <> a[[k]]]; AppendTo[d, QFunction[Symbol[p <> x]]]]; Map[Remove, Flatten[{c, h}]];
      If[DeleteDuplicates[d] == {True}, True, False]]]]
End[]
```

```
Begin["`CurrentNb`"]
CurrentNb[] := StringTake[StringCases[ToString[NotebookSelection[]], "<<" ~~ __ ~~ ">>"][[1]], {3, -3}]
End[]

Begin["`QFunction`"]
QFunction[x_] := Module[{a = Quiet[Definition2[x][[1]]], b = ToString3[HeadPF[x]]},
    If[! SingleDefQ[x], False, If[SuffPref[ToString1[a], {"CompiledFunction[", "Function["}, 1] || SuffPref[ToString1[a], " & ", 2],
        True, If[SameQ[a, x], False, If[SuffPref[b, "HeadPF[", 1], False, b = Map3[StringJoin, b, {" := ", " = "}];
            If[MemberQ[{SuffPref[StringReplace[a, b → ""], "Module[", 1], SuffPref[StringReplace[a, b → ""], "Block[", 1]}, True],
              False, True]]]]]]
End[]

Begin["`CompActPF`"]
CompActPF[x_ /; BlockFuncModQ[x]] :=
  Module[{b = {}, c = "", d, a = ToDefOptPF[x], f = ToString[x] <> ".txt", h = ""}, Put[FullDefinition[x], f];
    Quiet[While[! SameQ[h, EndOfFile], h = Read[f, String]; If[h ≠ " ", c = c <> h;
          If[HeadingQ[d = StringTake[c, {1, Flatten[StringPosition[c, " := "]][[1]] - 1}]], AppendTo[b, d];
            c = ""];
          Continue[]]]]; DeleteFile[Close[f]];
    {Map[HeadName, b], b}]
End[]

Begin["`CompActPF1`"]
CompActPF1[x_ /; BlockFuncModQ[x]] :=
  Module[{d = {}, k = 1, b = Args[x, 90], a = Flatten[{PureDefinition[x]}][[1]], c = Locals1[x], p},
    {b, c} = {If[NestListQ[b], b[[1]], b], If[NestListQ[c], c[[1]], c]};
    a = Select[ExtrVarsOfStr[a, 2], ! MemberQ[Flatten[{ToString[x], Join[b, c, {"Block", "Module"}]}], #] &];
    While[k ≤ Length[a], p = a[[k]];
      AppendTo[d, If[BlockFuncModQ[p], {p, HeadPF[p]}, If[SystemQ[p], {p, "System"}, {p, "Undefined"}]]];
      k++];
    a = Map[Flatten, Gather[d, ! StringFreeQ[#1[[2]], "_"] && ! StringFreeQ[#2[[2]], "_"] &]];
    b = Map[Flatten, Gather[a, #1[[2]] == "System" && #2[[2]] == "System" &]];
    d = Map[Flatten, Gather[b, #1[[2]] == "Undefined" && #2[[2]] == "Undefined" &]];
    Map[If[#[[-1]] == "System", Prepend[MinusList[#, {"System"}], "System"],
        If[#[[-1]] == "Undefined", Prepend[MinusList[#, {"Undefined"}], "Undefined"], #]] &, d]]
End[]

Begin["`NamesProc`"]
NamesProc[] := Select[Sort[Names["`*"]], Quiet[BlockFuncModQ[#]] &&
      ToString[Definition[#]] ≠ "Null" && ToString[Definition[#]] ≠ "Attributes[" <> ToString[#] <> "] = {Temporary}" &&
      ! MemberQ[{ToString[#] <> " = {Temporary}", ToString[#] <> " = {Temporary}"}, ToString[Definition[#]]] &]
End[]

Begin["`DeleteOptsAttr`"]
DeleteOptsAttr[x_ /; BlockFuncModQ[x], y___] := Module[{b, a = Definition2[x], c = "Options[" <> ToString[x] <> "]"}, b = a[[-1]];
    ClearAllAttributes[x];
    ClearAll[x];
    ToExpression[Select[a, StringFreeQ[ToString[#], c] &]]; If[{y} == {}, If[b ≠ {}, SetAttributes[x, b]]]]
End[]

Begin["`ScanLikeProcs`"]
ScanLikeProcs[x_: {}] := Module[{b = {}, c = {}, d, h, k = 1,
      a = Select[Names["`*"], StringFreeQ[#, "$"] && Quiet[Check[BlockFuncModQ[#], False]] &]}, Off[Definition::ssle];
    If[a == {}, Return[{}], For[k, k ≤ Length[a], k++, d = Definition2[a[[k]]][[1 ;; -2]];
        If[Length[d] > 1, AppendTo[b, Map[StringTake[#, {1, Flatten[StringPosition[#, " := "]][[1]] - 1}] &, d]];
          AppendTo[c, a[[k]]]]]]; On[Definition::ssle]; If[! HowAct[x], x = b, Null]; c]
End[]

Begin["`SubsStr`"]
SubsStr[x_ /; StringQ[x], y_ /; StringQ[y], h_ /; ListQ[h], t_ /; MemberQ[{0, 1}, t]] :=
  Module[{a = Map[ToString, h], b}, If[StringFreeQ[x, y], Return[x], b = If[t == 1, Map3[StringJoin, y, a], Mapp[StringJoin, a, y]]];
    If[StringFreeQ[x, b], Return[x], StringReplace[x, Map9[Rule, b, h]]]]
End[]

Begin["`SubsDel`"]
SubsDel[S_ /; StringQ[S], x_ /; StringQ[x],
    y_ /; ListQ[y] && DeleteDuplicates[Map[StringQ, y]] == {True} && Plus[Sequences[Map[StringLength, y]]] == Length[y],
    p_ /; MemberQ[{-1, 1}, p]] :=
```

```
  Module[{b, c = x, d, h = StringLength[S], k}, If[StringFreeQ[S, x], Return[S], b = StringPosition[S, x][[1]]];
    For[k = If[p == 1, b[[2]] + 1, b[[1]]-1], If[p == 1, k ≤ h, k ≥ 1], If[p == 1, k++, k--], d = StringTake[S, {k, k}];
      If[MemberQ[y, d] || If[p == 1, k == 1, k == h], Break[], If[p == 1, c = c <> d, c = d <> c]; Continue[]]]; StringReplace[S, c --> ""]]
End[]

Begin["`ExpLocals`"]
ExpLocals[P_ /; ModuleQ[P] || BlockQ[P], L_ /; ListQ[L] && DeleteDuplicates[Map[StringQ, L]] == {True}] :=
  Module[{a = Flatten[{PureDefinition[P]}][[1]], b = Locals1[P], c = Args[P, 90], d, p, p1, h, Op = Options[P], Atr = Attributes[P], t},
    Quiet[d = Map[If[StringFreeQ[#, {" = ", "="}], #, StringSplit[#, {" = ", "="}][[1]]] &, L];
      p = Locals[P];
      h = MinusList1[d, Flatten[{b, c}]];
      If[h == {}, Return[{}]]; ClearAll[t];
      BlockModQ[P, t]; h = Flatten[Map[Position[d, #] &, h]]; d = Join[p, c = Map[L[[#]] &, h]];
      ToExpression["ClearAllAttributes[" <> ToString[P] <> "]"];
      ClearAll[P]; ToExpression[StringReplace[a, t <> "[" <> ToString[p] → t <> "[" <> ToString[d], 1]]];
    If[Op ≠ {}, SetOptions[P, Op]]; SetAttributes[P, Atr]; c]
End[]

Begin["`MinusList`"]
MinusList[x_ /; ListQ[x], y_ /; ListQ[y]] := Select[x, ! MemberQ[y, #] &]
End[]

Begin["`MinusList1`"]
MinusList1[x_ /; ListQ[x], y_ /; ListQ[y]] :=
  Module[{a = x, k = 1}, While[k ≤ Length[y], a = ReplacePart[a, Flatten[Position[a, y[[k]]]][[1]] → Null];
      k++];
    Select[a, ! SameQ[#, Null] &]]
End[]

Begin["`PosSubList`"]
PosSubList[x_ /; ListQ[x], y_ /; ListQ[y]] :=
  Module[{a = ToString1[x], b = ToString1[y], c = FromCharacterCode[16], d}, d = StringTake[b, {2, -2}];
    If[! StringFreeQ[a, d], b = StringReplace[a, d → c <> "," <> StringTake[ToString1[y[[2 ;; -1]]], {2, -2}]];
      Map[{#, # + Length[y] - 1} &, Flatten[Position[ToExpression[b], ToExpression[c]]], {}]]
End[]

Begin["`Df`"]
Df[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y], Simplify[Subs[D[Subs[x, y, a], a], a, y]]]]
End[]

Begin["`Df1`"]
Df1[x_, y_] :=
  Module[{a, b, c = "$$Art27$Kr20$$"}, If[! HowAct[y], D[x, y], {a, b} = Map[ToString, Map[InputForm, {x, y}]]; Simplify[
        ToExpression[StringReplace[ToString[InputForm[D[ToExpression[StringReplace[a, b → c]], ToExpression[c]]]], c → b]]]]]
End[]

Begin["`D1`"]
D1[x_, y_] := Module[{a, b, c}, c = Replace2[D[b = Replace2[x, y, a];
          If[b === $Failed, Return[$Failed], b], a], a, y];
    If[b === $Failed, $Failed, c]]
End[]

Begin["`Diff`"]
Diff[x_, y__] :=
  Module[{c = {}, d = {}, b = Length[{y}], t = {}, k = 1, h = x, n = g, a = Map[ToString, Map[InputForm, {y}]]}, Clear[g];
    While[k ≤ b, AppendTo[c, Unique[g]]; AppendTo[d, ToString[c[[k]]]];
      AppendTo[t, a[[k]] → d[[k]]]; h = ToExpression[StringReplace[ToString[h // InputForm], t[[k]]]];
      h = D[h, c[[k]]]; h = ReplaceAll[h, Map[ToExpression, Part[t[[k]], 2] → Part[t[[k]], 1]]];
      k++];
    g = n;
    Map[Clear, c]; h]
End[]

Begin["`InsertN`"]
InsertN[S_ /; StringQ[S], L_ /; ListQ[L], n_ /; ListQ[n] && Length[n] == Length[Select[n, IntegerQ[#] &]]] :=
  Module[{a = Map[ToString, L], b, c = FromCharacterCode[2], d = Characters[S], k = 1, p, m = DeleteDuplicates[Sort[n]]},
    b = Map[c <> ToString[#] &, Range[1, Length[d]]]; b = Riffle[d, b];
    p = Min[Length[a], Length[m]];
    While[k ≤ p, If[m[[k]] < 1, PrependTo[b, a[[k]]], If[m[[k]] > Length[d], AppendTo[b, a[[k]]],
```

```
        b = ReplaceAll[b, c <> ToString[m[[k]]] → a[[k]]]]]; k++]; StringJoin[Select[b, ! SuffPref[#, c, 1] &]]]
End[]

Begin["`IsFile`"]
IsFile[x_ /; StringQ[x]] := If[FileExistsQ[x], If[! DirectoryQ[x], True, False], False]
End[]

Begin["`StringSplit1`"]
StringSplit1[x_ /; StringQ[x], y_ /; StringQ[y]] := Module[{a = StringSplit[x, y], b, c = {}, d, p, k = 1, j = 1}, d = Length[a];
  Label[G]; For[k = j, k ≤ d, k++, p = a[[k]];
    If[! SameQ[Quiet[ToExpression[p]], $Failed], AppendTo[c, p], b = a[[k]]; For[j = k, j ≤ d − 1, j++, b = b <> y <> a[[j + 1]];
      If[! SameQ[Quiet[ToExpression[b]], $Failed], AppendTo[c, b]; Goto[G], Null]]]]; Map[StringTrim, c]]
End[]

Begin["`StringSplit2`"]
StringSplit2[x_ /; StringQ[x], y_ /; StringQ[y]] :=
  Module[{a = FromCharacterCode[17]}, StringSplit[StringReplace[x, y → a <> y], a]]
End[]

Begin["`DirFull`"]
DirFull[x_ /; DirQ[x]] :=
  Module[{a = "$Art27Kr20$", b = StandPath[StringReplace[x, "/" → "\\"]], c, d}, If[DirEmptyQ[x], {}, Run["Dir /S/B/A ", b, " > ", a];
    c = Map[ToString, ReadList[a, String]]; DeleteFile[a]; Prepend[c, b]]]
End[]

Begin["`DelDirFile`"]
DelDirFile[x_ /; StringQ[x] && DirQ[x] || FileExistsQ[x], y___] :=
  Module[{c, f, a = {}, b = "", k = 1}, If[DirQ[x] && If[StringLength[x] == 3 && StringTake[x, {2, 2}] == ":", False, True],
    If[{y} == {}, Quiet[DeleteDirectory[x, DeleteContents → True]], a = {};
      b = ""; c = StandPath[x]; f = "$Art2618Kr$"; Run["Dir " <> c <> " /A/B/OG/S > " <> f];
      Attribs[c, 90]; For[k, k < Infinity, k++, b = Read[f, String];
        If[SameQ[b, EndOfFile], DeleteFile[Close[f]];
          Break[], Attribs[b, 90]]];
      DeleteDirectory[x, DeleteContents −> True]], If[FileExistsQ[x], If[{y} ≠ {}, Attribs[x, 90]];
      Quiet[DeleteFile[x]], $Failed]]]
End[]

Begin["`DelDirFile1`"]
DelDirFile1[x_ /; StringQ[x] && FileExistsQ[x] || DirQ[x] && If[StringLength[x] == 3 && StringTake[x, {2, 2}] == ":", False, True]] :=
  Module[{a = {}, b = "", c = StandPath[x], d, f = "$Art590Kr$", k = 1}, If[DirQ[x], Run["Dir " <> c <> " /A/B/OG/S > " <> f];
    Attribs[c, 90]; For[k, k < Infinity, k++, b = Read[f, String];
      If[SameQ[b, EndOfFile], DeleteFile[Close[f]]; Break[], Attribs[b, 90]; Close2[b]]];
    DeleteDirectory[x, DeleteContents −> True], Close2[x]; Attribs[x, 90]; DeleteFile[x]]]
End[]

Begin["`WhatValue`"]
WhatValue[x_] :=
  If[SystemQ[x], {"System", x}, If[! SameQ[Definition2[ToString[x]][[1]], ToString[x]], {"Local", x}, {"Undefined", x}]]
End[]

Begin["`ParVar`"]
ParVar[x_ /; SymbolQ[x], y_ /; ListQ[y]] := Module[{a = {}, b, k = 1}, For[k, k ≤ Length[y], k++, AppendTo[a, Unique[x]]];
  b = ToString[a];
  {b, ToExpression[b <> "=" <> ToString1[y]]}[[1]]]
End[]

Begin["`MaxParts`"]
MaxParts[x_] := Length[Op[x]]
End[]

Begin["`ProcCalls`"]
ProcCalls[x_ /; StringQ[x]] :=
  Module[{a = Select[StringSplit[ToString[InputForm[Definition[x]]], "\n"], # ≠ " " && # ≠ x && ! SuffPref[#, x <> " := ", 1] &],
    b = Attributes[x]},
  a = If[SuffPref[a[[1]], "Attributes[", 1], a[[2 ;; −1]], a];
  ClearAttributes[x, b];
  Clear[x];
  Map[ToExpression, a];
  SetAttributes[x, b]]
```

```
End[]

Begin["`ProcCalls1`"]
ProcCalls1[x_ /; StringQ[x]] := Module[{a = Flatten[{PureDefinition[x]}], c = Attributes[x], d = {}}, ClearAttributes[x, c];
    Do[If[TestDefBFM[a[[k]], x], AppendTo[d, a[[k]]], Null], {k, 1, Length[a]}]; Clear[x]; ToExpression[d]; SetAttributes[x, c]]
End[]

Begin["`TestDefBFM`"]
TestDefBFM[x_ /; StringQ[x], y_ /; StringQ[y]] := Module[{a = ToString[Unique["S"]], b, c}, b = a <> y;
    ToExpression[a <> x];
    c = BlockFuncModQ[b];
    ToExpression["Remove[" <> b <> "]"]; c]
End[]

Begin["`WhatObj`"]
WhatObj[x_ /; SymbolQ[x]] := Module[{a = Quiet[Context[x]], t}, If[a === "System`", "System",
      If[a === "Global`", If[MemberQ[{$Failed, "Undefined"}, PureDefinition[x]], "Undefined", "CS"], a]]
End[]

Begin["`HeadingQ`"]
HeadingQ[x_ /; StringQ[x]] := Module[{a, b, c, k = 1, m = True, n = True},
    If[StringTake[x, {-1, -1}] == "]" && StringCount[x, {"[", "]"}] == 2 && ! StringFreeQ[StringReplace[x, " " → ""], "[]"], Return[m],
      If[! Quiet[StringFreeQ[RedSymbStr[x, "_", "_"], "[_]"]], Return[! m]]];
    Quiet[Check[ToExpression[x], Return[False]]];
    If[DeleteDuplicates[Map3[StringFreeQ, x, {"[", "]"}]] === {False}, c = StringPosition[x, "["][[1]][[2]];
      If[c == 1, Return[False], a = StringTake[x, {c, -1}]], Return[False]];
    b = StringPosition[a, "["][[1]][[1]]; c = StringPosition[a, "]"][[-1]][[1]];
    a = "{" <> StringTake[a, {b + 1, c - 1}] <> "}"; a = Map[ToString, ToExpression[a]];
    If[DeleteDuplicates[Mapp[StringFreeQ, a, "_"]] == {False}, Return[True]];
    If[{c, a} == {2, {}}, Return[True], If[a == {} || StringTake[a[[1]], {1, 1}] == "_", Return[False], For[k, k ≤ Length[a], k ++, b = a[[k]];
          If[StringReplace[b, "_" → ""] ≠ "" && StringTake[b, {-1, -1}] == "_" ||
              ! StringFreeQ[b, "_"] || ! StringFreeQ[b, "_:"] || ! StringFreeQ[b, "_."], m = True, n = False]]];
      m && n]]
End[]

Begin["`HeadingQ1`"]
HeadingQ1[x_ /; StringQ[x]] :=
  Module[{b, c = {}, d, h = ToString[Unique["sv"]], k, a = Quiet[StringTake[x, {Flatten[StringPosition[x, "[", 1]][[1]] + 1, -2}]]},
    If[StringFreeQ[x, "["], False, b = StringSplit1[a, ","];
      For[k = 1, k ≤ Length[b], k ++, d = b[[k]];
        AppendTo[c, If[StringFreeQ[d, "_"], False, If[MemberQ[ToString /@ {Complex, Integer, List, Rational, Real, String, Symbol},
              StringTake[d, {Flatten[StringPosition[d, "_"]][[-1]] + 1, -1}]], True, HeadingQ[h <> "[" <> d <> "]"]]]];
      If[DeleteDuplicates[c] == {True}, True, False]]
End[]

Begin["`HeadingQ2`"]
HeadingQ2[x_ /; StringQ[x]] :=
  Module[{a, b, c, d = ToString[Unique["agn"]]}, {a, b} = Map[DeleteDuplicates, Map[Flatten, Map3[StringPosition, x, {"[", "]"}]]];
If[StringLength[x] == b[[-1]] && SymbolQ[c = StringTake[x, {1, a[[1]] - 1}]],
      Quiet[Check[ToExpression[StringReplace[x, c <> "[" → d <> "[", 1] <> " := 72"], False]];
      c = Map[SyntaxQ, ArgsTypes[d]];
      ToExpression["Remove[" <> d <> "]"]; If[DeleteDuplicates[c] === {True}, True, False], False]]
End[]

Begin["`HeadingQ3`"]
HeadingQ3[x_ /; StringQ[x]] := Module[{a = "AvzRansIan", b}, Clear[AvzRansIan];
    b = Quiet[ToExpression[a <> StringTake[x, {Flatten[StringPosition[x, "["]][[1]], -1}]
          <> ":=90"]]; If[SameQ[b, Null], Clear[a]; HeadingQ2[x], Clear[a]; False]]
End[]

Begin["`TestHeadingQ`"]
TestHeadingQ[x_ /; StringQ[x], y___] :=
  Module[{a, b, c = Map[ToString, {Arbitrary, Integer, Real, Complex, List, Symbol, String}], d = {}, g = {}, f, h},
    f[t_] := If[{y} ≠ {} && ! HowAct[y], y = t, Null];
    h = Quiet[Check[StringReplace[x, HeadName1[x] -> "", 1], $Failed]]; If[h === $Failed, f[x];
      False, If[! SyntaxQ[x], f[x]; False, h = StringDelete[h, " "];
        If[! (SuffPref[h, "[", 1] && SuffPref[h, "]", 2]) || StringFreeQ[h, "_"] || ! StringFreeQ[h, "_;"], f[h];
          False, a = ToString[Unique["gs"]]; b = Quiet[Check[ToExpression[a <> h <> ":=74"], $Failed]];
          If[b === $Failed, f[h]; Remove[a]; False, b = Quiet[Check[Args[a, 90], $Failed]];
```

```
            If[b === $Failed, f[h]; Remove[a];  False,  b = Quiet[Check[ArgsTypes[a], $Failed]];
              If[b === $Failed, f[h];  Remove[a];  False,  b = If[! ListListQ[b], {b}, b];
                Map[If[MemberQ[c, #[[2]]] || #[[2]] == "." || SuffPref[#[[2]], ":", 1] || StringMatchQ[#[[2]], X__ ~~ ":" ~~ Y__] ||
                      MemberQ[{True, False}, Quiet[Check[ToExpression[#[[2]]], $Failed]]],
                    AppendTo[d, True], AppendTo[g, #]] &, b];  If[g == {}, f[{}];  True, f[g];
                False]]]]]]]
End[]

Begin["`Df2`"]
Df2[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y], Simplify[ReplaceAll1[D[ReplaceAll1[x, y, a], a], a, y]]]]
End[]

Begin["`Difff`"]
Difff[x_, y__] := Module[{a = x, a1, a2, a3, b = Length[{y}], c = {}, d, k = 1, n = g}, Clear[g];
    While[k ≤ b, d = {y}[[k]];
      AppendTo[c, Unique[g]]; a1 = Replace4[a, d → c[[k]]];
      a2 = D[a1, c[[k]]]; a3 = Replace4[a2, c[[k]] → d];
      a = a3; k++]; g = n; Simplify[a3]]
End[]

Begin["`Int`"]
Int[x_, y_] := Module[{a}, If[! HowAct[y], Integrate[x, y], Simplify[Subs[Integrate[Subs[x, y, a], a], a, y]]]]
End[]

Begin["`Integral2`"]
Integral2[x_, y__] := Module[{a = x, a1, a2, a3, b = Length[{y}], c = {}, d, k = 1, n = g}, Clear[g];
    While[k ≤ b, d = {y}[[k]]; AppendTo[c, Unique[g]];
      a1 = Replace4[a, d → c[[k]]]; a2 = Integrate[a1, c[[k]]];
      a3 = Replace4[a2, c[[k]] → d]; a = a3; k++]; g = n; Simplify[a3]]
End[]

Begin["`MdP`"]
MdP[x___] := Module[{a = Select[Names["`*"], BlockFuncModQ[#] &], b = {}, c, d}, d = Flatten[Map[ToString, {x}]];
    a = If[a == {}, {}, If[d == {}, a, If[MemberQ4[a, d], Intersection[a, d], {}]]]; If[a == {}, $Failed,
      c = Map[AppendTo[b, {#, Length[Flatten[{PureDefinition[#]}]]}] &, a][[-1]]; If[Length[c] > 1, c, c[[1]]]]]
End[]

Begin["`ModLibraryPath`"]
ModLibraryPath[x_ /; DirQ[x]] := Module[{a = $LibraryPath, b = Attributes[$LibraryPath]}, ClearAttributes[$LibraryPath, b];
    a = Insert[a, x, -1]; $LibraryPath = a; SetAttributes[$LibraryPath, b]]
End[]

Begin["`ModuleQ`"]
ModuleQ[x_, y___ /; y == Null  || SymbolQ[y] && ! HowAct[y]] :=
  Module[{a = PureDefinition[x], b}, If[ListQ[a] || a == "System" || a === $Failed, False, b = HeadPF[x];
      If[SuffPref[a, b <> " := " <> "Module[{", 1], If[{y} ≠ {}, y = "Module"];
        True, If[SuffPref[a, b <> " := " <> "DynamicModule[{", 1], If[{y} ≠ {}, y = "DynamicModule"];
          True, False]]]]
End[]

Begin["`ModuleQ1`"]
ModuleQ1[x_] := If[SymbolQ[x] && TestBFM[x] === "Module", True, False]
End[]

Begin["`ModuleQ2`"]
ModuleQ2[x_] := Module[{a  =  If[SymbolQ[x], Flatten[{PureDefinition[x]}][[1]], $Failed], b},
    If[MemberQ[{$Failed, "System"}, a], False,  b  = Map14[StringJoin, {" := ", " = "}, "Module[{"];
      If[SuffPref[a, Map3[StringJoin, HeadPF[x], b], 1], True, False]]]
End[]

Begin["`FuncBlockModQ`"]
FuncBlockModQ[x_ /; SymbolQ[x], y_ /; ! HowAct[y]] :=
  Module[{b, c, m, n, a = PureDefinition[x]}, If[MemberQ[{"System", $Failed}, a], False, a = Flatten[{a}];
      b = Flatten[{HeadPF[x]}];
      c = Join[Mapp[StringJoin, b, " := "], Mapp[StringJoin, b, " = "]];
      c = GenRules[c, ""]; c = StringReplace[a, c];
      {m, n} = Map[Length, {Select[c, SuffPref[#, "Block[{", 1] &], Select[c, SuffPref[#, "Module[{", 1] &]}];
      If[Length[a] == m, y = "Block";
        True, If[Length[a] == n, y = "Module";
```

```
            True, If[m + n == 0, y = "Function";
              True, y = "Multiple"; False]]]]
End[]

Begin["`Int1`"]
Int1[x_, y_] := Module[{a}, If[! HowAct[y], Integrate[x, y], Simplify[ReplaceAll1[Integrate[ReplaceAll1[x, y, a], a], a, y]]]]
End[]

Begin["`Integrate1`"]
Integrate1[x_, y_] := Module[{a, b, c}, c = Replace2[Integrate[b = Replace2[x, y, a];
          If[b === $Failed, Return[$Failed], b], a], a, y];
    If[b === $Failed, $Failed, c]]
End[]

Begin["`SortNL`"]
SortNL[L_ /; ListListQ[L] && DeleteDuplicates[Map[NumericQ[#] &, Flatten[L]]] == {True}, p_ /; IntegerQ[p],
    b_ /; MemberQ[{Greater, Less}, b]] := If[p ≥ 1 && p ≤ Length[L[[1]]], Sort[L, b[#1[[p]], #2[[p]]] &],
    Print[SortNL::"incorrect second argument, should lay in interval ", {1, Length[L[[1]]]}]]
End[]

Begin["`SortNL1`"]
SortNL1[L_ /; ListListQ[L], p_ /; IntegerQ[p], b_ /; MemberQ[{Greater, Less}, b]] := If[p ≥ 1 && p ≤ Length[L[[1]]],
    Sort[L, b[GC[#1[[p]]], GC[#2[[p]]]] &], Print[SortNL::"incorrect second argument, should lay in interval ", {1, Length[L[[1]]]}]]
End[]

Begin["`SortNestList`"]
SortNestList[x_ /; NestListQ[x], p_ /; PosIntQ[p], y_] := Module[{a = DeleteDuplicates[Map[Length, x]], b},
    b = If[SameQ[DeleteDuplicates[Map[ListNumericQ, x]], {True}] && MemberQ[{Greater, Less}, y], y,
        If[SameQ[DeleteDuplicates[Map[ListSymbolQ, x]], {True}] && MemberQ[{SymbolGreater, SymbolLess}, y], y],
        Return[Defer[SortNestList[x, p, y]]]];
    If[Min[a] ≤ p ≤ Max[a], Sort[x, b[#1[[p]], #2[[p]]] &], Defer[SortNestList[x, p, y]]]]
End[]

Begin["`SeqToString`"]
SeqToString[h___] := StringTake[ToString1[{h}], {2, -2}]
End[]

Begin["`Spos`"]
Spos[x_ /; StringQ[x], y_ /; StringQ[y], p_ /; IntegerQ[p], dir_ /; IntegerQ[dir]] :=
  Module[{a, b, c}, If[StringFreeQ[x, y], Return[0], If[StringLength[y] > 1 || dir ≠ 0 && dir ≠ 1, Return[False], b = StringLength[x]]];
    If[p < 1 || p > b, False, If[p == 1 && dir == 0, c = 0,
        If[p == b && dir == 1, c = 0, If[dir == 0, For[a = p, a ≥ 1, a -= 1, If[StringTake[x, {a}] == y, Return[a], c]],
            For[a = p, a ≤ b, a += 1, If[StringTake[x, {a}] == y, Return[a], c]]]]];
    If[a == 0 || a == b + 1, 0, a]]
End[]

Begin["`StringPat`"]
StringPat[x_ /; StringQ[x] || ListStringQ[x], y_ /; MemberQ[{"_", "__", "___"}, y]] :=
  Module[{a = "", b}, If[StringQ[x], x, b = Map[ToString1, x];
      ToExpression[StringJoin[Map[# <> "~~" <> y <> "~~" &, b[[1 ;; -2]]], b[[-1]]]]]]
End[]

Begin["`StringCases1`"]
StringCases1[x_ /; StringQ[x], y_ /; StringQ[y] || ListStringQ[y], z_ /; MemberQ[{"_", "__", "___"}, z]] :=
  Module[{b, c = "", d, k = 1},
    Sort[Flatten[Map[DeleteDuplicates, If[StringQ[y], {StringCases[x, y]}, {StringCases[x, StringPat[y, z], Overlaps -> All]}]]]]
End[]

Begin["`StringCases2`"]
StringCases2[x_ /; StringQ[x], y_ /; ListQ[y] && y ≠ {}] :=
  Module[{a, b = "", c = Map[ToString, y]}, Do[b = b <> ToString1[c[[k]]] <> "~~__~~", {k, 1, Length[c]}];
    a = ToExpression[StringTake[b, {1, -7}]];  StringCases[x, Shortest[a]]]
End[]

Begin["`StringFreeQ1`"]
StringFreeQ1[x_ /; StringQ[x], y_ /; StringQ[y] || ListStringQ[y], z_ /; MemberQ[{"_", "__", "___"}, z]] :=
  If[StringQ[y], StringFreeQ[x, y], If[StringCases1[x, y, z] == {}, True, False]]
End[]
```

```
Begin["`StringFreeQ2`"]
StringFreeQ2[x_ /; StringQ[x], y_ /; StringQ[y] || ListQ[y] && DeleteDuplicates[Map[StringQ[#] &, y]] == {True}] :=
  ! MemberQ[Map[StringFreeQ[x, #] &, Flatten[{y}]], False]
End[]

Begin["`StringFreeQ3`"]
StringFreeQ3[x_, y_ /; StringQ[y]] := StringFreeQ[ToString1[x], y]
End[]

Begin["`SelectStrings`"]
SelectStrings[x_ /; ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True},
    y_ /; ListQ[y] && DeleteDuplicates[Map[StringQ, y]] == {True}, Ig_: True] :=
  Select[x, StringFreeQ2[If[Ig == True, ToUpperCase[#], #], If[Ig == True, Map[ToUpperCase, y], y]] &]
End[]

Begin["`SelectContains`"]
SelectContains[x_ /; ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True},
    y_ /; ListQ[y] && DeleteDuplicates[Map[StringQ, y]] == {True}, r_ /; MemberQ[{False, True}, r], Ig_: True] :=
  Module[{a = Riffle1[Map[StringFreeQ[x, #, IgnoreCase → Ig] &, y]], b},
    b = Map[ToExpression["And(" <> StringTake[ToString[#], {2, -2}] <> ")"] &, a];
    Map[If[#[[2]] === r, #[[1]], Nothing] &, Riffle1[{x, b}]]]
End[]

Begin["`MaxNestLevel`"]
MaxNestLevel[L_ /; ListQ[L]] := Module[{a = Flatten[L], b = L, c = 0}, While[! a == b, b = Flatten[b, 1];
      c = c + 1]; c]
End[]

Begin["`Avg`"]
Avg[] := Module[{b, a = ToString[ToExpression[ToString[InputForm[Stack[_][[1]]]]]],
    a = If[! SuffPref[a, {"Module[", "Block["}, 1], "Module[{}," <> a <> "]", a]; a = StringReplace[a, "$" -> ""];
    a = StringReplace[a, If[SuffPref[a, "Block[", 1], "Block[", "Module["] → "", 1]; a = SubStrSymbolParity1[a, "{", "}"][[1]];
    If[a == "{}", {}, b = StrToList[StringTake[a, {2, -2}]]; b = Map[StringSplit[#, " = "] &, b];
      Map[If[Length[#] == 1, {#[[1]], "None"}, #] &, b]]]
End[]

Begin["`LongestCommonSubString`"]
LongestCommonSubString[x_ /; StringQ[x], y_ /; StringQ[y]] := Module[{a, b, c, d}, If[StringLength[x] > StringLength[y], a = x;
      b = y, a = y;
      b = x];
    d = DeleteDuplicates[Select[Map[StringJoin, Subsets[Characters[b]]], ! StringFreeQ[a, #] && # ≠ "" &]];
    c = Select[Select[d, ! StringFreeQ[b, #] &], ! StringFreeQ[a, #] &];
    If[c == {}, c, Sort[Select[c, StringLength[#] == Max[Map[StringLength, c]] &]]]]
End[]

Begin["`LongestCommonSubSequence`"]
LongestCommonSubSequence[x_ /; StringQ[x], y_ /; StringQ[y]] := Module[{a, b, c},
    {a, b} = {Select[Map[StringJoin, Subsets[Characters[x]]], # ≠ "" &], Select[Map[StringJoin, Subsets[Characters[y]]], # ≠ "" &]};
    c = Intersection[a, b]; If[c == {}, c, Select[c, StringLength[#] == Max[Map[StringLength, c]] &]]]
End[]

Begin["`LongestCommonSubsequence1`"]
LongestCommonSubsequence1[x_ /; StringQ[x], y_ /; StringQ[y], Ig_ /; MemberQ[{False, True}, Ig], t___] :=
  Module[{a = Characters[x], b = Characters[y], c, d, f},
    f[z_, h_] := Map[If[StringFreeQ[h, #], Nothing, #] &, Map[StringJoin[#] &, Subsets[z]][[2 ;; -1]]];
    c = Gather[d = Sort[If[Ig == True, Intersection1[f[a, x], f[b, y], Ig], Intersection[f[a, x], f[b, y]]],
          StringLength[#1] ≤ StringLength[#2] &], StringLength[#1] == StringLength[#2] &];
    If[{t} ≠ {} && ! HowAct[t], t = d, Null]; c = If[c == {}, {}, c[[-1]]]; If[c == {}, {}, If[Length[c] == 1, c[[1]], c]]]
End[]

Begin["`LongestCommonSubsequence2`"]
LongestCommonSubsequence2[x__ /; DeleteDuplicates[Flatten[Map[StringQ, {x}]]] == {True}, Ig_ /; MemberQ[{False, True}, Ig]] :=
  Module[{a = {}, b = {x}, c = Length[{x}], d = {}, h}, h = Subsets[b, 2][[c + 2 ;; -1]];
    Do[AppendTo[a, LongestCommonSubsequence1[h[[k]][[1]], h[[k]][[2]], Ig]], {k, 1, Length[h]}]; a = DeleteDuplicates[Flatten[a]];
    Do[If[DeleteDuplicates[Flatten[Map[! StringFreeQ[#, a[[k]], IgnoreCase → Ig] &, b]]] == {True}, AppendTo[d, a[[k]]], Null],
      {k, 1, Length[a]}]; d]
End[]

Begin["`DelSubStr`"]
```

```
DelSubStr[S_ /; StringQ[S], L_ /; ListQ[L] && MemberQ[{0, 1}, MaxNestLevel[L]]] :=
  Module[{a = If[MaxNestLevel[L] == 1, Select[Sort[Map[Sort, L]],
        MemberQ[Range[1, StringLength[S]], First[#]] && MemberQ[Range[1, StringLength[S]], First[#]] &], Map[List, Sort[L]]],
    c = S, d = {}, h, k = 1}, If[Sort[Flatten[L]][[-1]] > StringLength[S], S, For[k, k ≤ Length[a], k++,
      d = Append[d, If[Length[a[[k]]] == 2, a[[k]], {a[[k]][[1]], a[[k]][[1]]}]]]; d = Sort[d];
    For[k = 1, k ≤ Length[a], k++, h = d[[k]]; c = StringReplacePart[c, "", h];
      d = d - Last[h] + First[h] - 1]; c]]
End[]

Begin["`Locals`"]
Locals[x_ /; BlockFuncModQ[x], R___] :=
  Module[{c, d = {}, p, q, t, a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}], k = 1, Sg, Sv},
    Sg[y_ /; StringQ[y]] := Module[{h = 1, v = {}, j, s = "", z = StringLength[y] - 1}, Label[avz];
      For[j = h, j ≤ z, j++, s = s <> StringTake[y, {j, j}];
        If[! SameQ[Quiet[ToExpression[s]], $Failed] && StringTake[y, {j + 1, j + 1}] == ",", AppendTo[v, s];
          h =
            j +
              3;
          s = "";
          Goto[avz]]]; AppendTo[v, s <> StringTake[y, {-1, -1}]];
      Map[If[Quiet[StringTake[#, {1, 2}]] === ", ", StringTake[#, {3, -1}], #] &, v]];
    c = Flatten[Map[Map3[StringJoin, #, {" := ", " = "}] &, b]]; c = Map[StringReplace[#, Map[Rule[#, ""] &, c]] &, a];
    For[k, k ≤ Length[a], k++, p = c[[k]];
      If[SuffPref[p, "Module[{", 1], t = 8, If[SuffPref[p, "Block[{", 1], t = 7, t = 0; AppendTo[d, "Function"]]];
      If[t ≠ 0, AppendTo[d, SubStrSymbolParity1[StringTake[p, {t, -1}], "{", "}"][[1]]]]; Continue[]];
    d = Map[StringReplace[#, {"{" -> "{$$90$$", ", " -> ", $$90$$", "= " -> "= $$90$$"}] &, d];
    d = Map[If[MemberQ[{"Function", "{}"}, #], #, Sg[StringTake[#, {2, -2}]]] &, d];
    d = Map[If[FreeQ[Quiet[ToExpression[#]], $Failed], #, StringJoin1[#]] &, d];
    d = Map[If[# === {""}, "{}", #] &, Mapp[StringReplace, d, {"$$90$$" -> "", "\\" -> ""}]]; Map[ClearAll, Names["`$$90$$*"]];
    If[{R} ≠ {} && ! HowAct[R], Sv[y_List] := Map[Flatten, Map[{q = StringPosition[#, " = "],
          If[q == {}, {#, "No"}, {StringTake[#, {1, q[[1]][[1]] - 1}], StringTake[#, {q[[1]][[2]] + 1, -1}]}]}][[2 ;; -1]] &, y]];
     R = {}; Do[AppendTo[R, If[ListQ[d[[k]]], Sv[d[[k]]], d[[k]]]], {k, 1, Length[d]}]; R = If[ListQ[HeadPF[x]], R, R[[1]]], Null];
    d = If[Length[d] == 1, Flatten[d], d]; If[d == {"{}"}, {}, d]]
End[]

Begin["`Locals1`"]
Locals1[x_ /; BlockFuncModQ[x]] := Module[{a, b = {}, c, k = 1, kr},
    kr[y_ /; ListQ[y]] := Module[{d = {}, v = Flatten[y], j = 1}, While[j ≤ Length[v] / 2, AppendTo[d, v[[2 * j - 1]]];
        j++]; d];
    ClearAll[a]; Locals[x, a]; If[NestListQ1[a], For[k, k ≤ Length[a], k++, c = a[[k]];
      AppendTo[b, If[MemberQ[{"{}", "Function"}, c], c, kr[c]]]]; If[StringQ[PureDefinition[x]], Flatten[b], b], kr[a]]]
End[]

Begin["`Locals2`"]
Locals2[x_ /; QBlockMod[x]] :=
  Module[{c = {}, d, p, h, q, k = 1, g = {}, a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}], Sv}, While[k ≤ Length[a],
      AppendTo[c, d = StringReplace[a[[k]], Mapp[Rule, Map[b[[k]] <> " := " <> # &, {"Module[", "Block["}], ""], 1];
        Quiet[SubStrSymbolParity[d, "{", "}", 1][[-1]]]]; k++];
    Sv[y_List] := Map[Flatten, Map[{q = StringPosition[#, " = "],
          If[q == {}, #, {StringTake[#, {1, q[[1]][[1]] - 1}], StringTake[#, {q[[1]][[2]] + 1, -1}]}][[1]]]}[[2 ;; -1]] &, y]];
    c = Locals[x]; c = If[NestListQ1[c], c, {c}];
     Do[AppendTo[g, Map[#[[1]] &, If[ListQ[c[[k]]], Quiet[Check[Sv[c[[k]]], c[[k]]]], c[[k]]]]], {k, 1, Length[c]}];
    g = If[Length[b] > 1, g, If[Length[g[[1]]] > 1, g[[1]], g[[1]][[1]]]]; If[StringQ[g], {g}, g]]
End[]

Begin["`Locals3`"]
Locals3[x_ /; BlockFuncModQ[x]] := Module[{a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}], c = {}, d, h, q, g = {}, S},
    Do[d = StringReplace[a[[k]], {b[[k]] <> " := " -> "", b[[k]] <> " = " -> "" }, 1];
      h = If[SuffPref[d, {"Module[", "Block["}, 1], StringTrim[d, {"Module[", "Block["}], "Function"];
      AppendTo[c, If[h == "Function", "Function", CorrectSubString[h, 1]]], {k, 1, Length[a]}];
    c = Map[If[MemberQ[{"Function", "{}"}, #], #, StringToList[StringTake[#, {2, -2}]]] &, c];
    S[y_] := Map[{q = StringPosition[#, " = "],
          If[q == {}, #, {StringTake[#, {1, q[[1]][[1]] - 1}], StringTake[#, {q[[1]][[2]] + 1, -1}]}]}[[2 ;; -1]] &, y];
    Do[AppendTo[g, If[MemberQ[{"Function", "{}"}, c[[k]]], c[[k]], Map[#[[1]] &, S[c[[k]]]]]], {k, 1, Length[c]}];
    Flatten[g, LevelsList[g] - 1]]
End[]

Begin["`SyntCorProcQ`"]
SyntCorProcQ[x_ /; BlockModQ[x]] := Module[{d, h, c = $KrArt$, b = PureDefinition[x], a = HeadPF[x]}, ClearAll[$KrArt$];
```

```
    $KrArt$ = ProcFuncTypeQ[ToString[x]][[2]][[1]];
    h = Quiet[Check[Locals2[x], Locals1[x]]];  h = If[h === {}, "{}", ToString[h]];  d = a <> " := " <> $KrArt$ <> "[" <> h;
    d = StringReplace[b, d --> "", 1];  $KrArt$ = c; ! MemberQ[{"]", ", Null]"}, d]]
End[]

Begin["`Subs`"]
Subs[x_, y_, z_] := Module[{d, k = 2, subs},
    subs[m_, n_, p_] :=
      Module[{a, b, c, h, t}, If[! HowAct[n], m /. n → p, {a, b, c, h} = First[{Map[ToString, Map[InputForm, {m, n, p, 1/n}]]}];
          t = Simplify[ToExpression[StringReplace[StringReplace[a, b → "(" <> c <> ")"], h --> "1/" <> "(" <> c <> ")"]]];
          If[t === m, m /. n → p, t]]];
    If[! ListQ[y] && ! ListQ[z], subs[x, y, z], If[ListQ[y] && ListQ[z] && Length[y] == Length[z], d = subs[x, y[[1]], z[[1]]];
       For[k, k ≤ Length[y], k++, d = subs[d, y[[k]], z[[k]]]]; d, Defer[Subs[x, y, z]]]]
End[]

Begin["`Subs1`"]
Subs1[x_, y_ /; ListQ[y] && Length[y] == 2 || ListListQ[y]] := ToExpression[
    StringReplace[ToString[FullForm[x]], Map[ToString[FullForm[#[[1]]]] → ToString[FullForm[#[[2]]]] &, If[ListListQ[y], y, {y}]]]]
End[]

Begin["`Subs2`"]
Subs2[x_, y_, z_] := If[Numerator[y] == 1 && ! SameQ[Denominator[y], 1], Subs1[x, {y, z}], Subs[x, y, z]]
End[]

Begin["`Subs1Q`"]
Subs1Q[x_, y_] := SameQ[x, Subs1[Subs1[x, y], If[ListListQ[y], Map[Reverse, y], Reverse[y]]]]
End[]

Begin["`Substitution`"]
Substitution[x_, y_, z_] := Module[{d, k = 2, subs, subs1},
    subs[m_, n_, p_] :=
      Module[{a, b, c, h, t}, If[! HowAct[n], m /. n → p, {a, b, c, h} = First[{Map[ToString, Map[InputForm, {m, n, p, 1/n}]]}];
          t = Simplify[ToExpression[StringReplace[StringReplace[a, b → "(" <> c <> ")"], h --> "1/" <> "(" <> c <> ")"]]];
          If[t === m, m /. n → p, t]]];
    subs1[m_, n_, p_] := ToExpression[StringReplace[ToString[FullForm[m]], ToString[FullForm[n]] → ToString[FullForm[p]]]];
    If[! ListQ[y] && ! ListQ[z], If[Numerator[y] == 1 && ! SameQ[Denominator[y], 1], subs1[x, y, z], subs[x, y, z]],
      If[ListQ[y] && ListQ[z] && Length[y] == Length[z],
        If[Numerator[y[[1]]] == 1 && ! SameQ[Denominator[y[[1]]], 1], d = subs1[x, y[[1]], z[[1]]], d = subs[x, y[[1]], z[[1]]]];
        For[k, k ≤ Length[y], k++, If[Numerator[y[[k]]] == 1 && ! SameQ[Denominator[y[[k]]], 1],
            d = subs1[d, y[[k]], z[[k]]], d = subs[d, y[[k]], z[[k]]]]];  d, Defer[Substitution[x, y, z]]]]
End[]

Begin["`Substitution1`"]
Substitution1[x_, y_, z_] := Module[{y1, z1, d, k = 2, subst},
    subst[m_, n_, p_] :=
      ToExpression[StringTake[StringReplace[ToString1[ReplaceAll[HoldAll[m], n → p]], "HoldAll[" --> ""], {1, -2}]];
    {y1, z1} = Map[Flatten, {{y}, {z}}];  If[ListQ[y1] && ListQ[z1] && Length[y1] == Length[z1], d = subst[x, y1[[1]], z1[[1]]];
      For[k, k ≤ Length[y1], k++, d = subst[d, y1[[k]], z1[[k]]]];  d, Defer[Substitution1[x, y, z]]]
End[]

Begin["`Integrate2`"]
Integrate2[x_, y__] := Module[{a, b, c = Map[Unique["gs"] &, Range[1, Length[{y}]]], d}, a = Riffle[{y}, c];
    a = If[Length[{y}] == 1, a, Partition[a, 2]];
    d = Integrate[Subs1[x, a], Sequences[c]];
    {Simplify[Subs1[d, If[ListListQ[a], Map[Reverse, a], Reverse[a]]]], Map[Remove, c]}[[1]]
End[]

Begin["`Diff1`"]
Diff1[x_, y__] := Module[{a, b, c = Map[Unique["gs"] &, Range[1, Length[{y}]]], d}, a = Riffle[{y}, c];
    a = If[Length[{y}] == 1, a, Partition[a, 2]];
    d = D[Subs1[x, a], Sequences[c]];  {Simplify[Subs1[d, If[ListListQ[a], Map[Reverse, a], Reverse[a]]]], Map[Remove, c]}[[1]]
End[]

Begin["`Diff2`"]
Diff2[x_, y__] := Module[{a = x, b = {y}, c, k}, Do[a = Substitution1[D[Substitution1[a, b[[k]], c], c], c, b[[k]]], {k, 1, Length[b]}];
    Simplify[a]]
End[]

Begin["`SubsList`"]
```

```
SubsList[x_ /; ListQ[x], y_, z_] := Module[{a = FromCharacterCode[2016], b, c, d},
    b = StringJoin[Map[ToString1[#] <> a &, x]]; c = Map[StringJoin[Map[ToString1[#] <> a &, Flatten[{#1}]]] &, {y, z}];
    c = ToExpression[StringSplit[SubsString[b, {c[[1]], c[[2]]}], a]]; If[Length[c] == 1, c[[1]], c]]
End[]

Begin["`BlockQ`"]
BlockQ[x_] := Module[{a = If[SymbolQ[x], Flatten[{PureDefinition[x]}][[1]], $Failed], b},
    If[MemberQ[{$Failed, "System"}, a], False, b = Map14[StringJoin, {" := ", " = "}, "Block[{"];
      If[SuffPref[a, Map3[StringJoin, HeadPF[x], b], 1], True, False]]]
End[]

Begin["`BlockQ1`"]
BlockQ1[x_] := If[SymbolQ[x] && TestBFM[x] === "Block", True, False]
End[]

Begin["`SymbolToList`"]
SymbolToList[x_ /; SymbolQ[x]] := Map[ToExpression, Characters[ToString[x]]]
End[]

Begin["`Nconcat`"]
Nconcat[h_ /; PosIntQ[h] || PosIntListQ[Flatten[h]]] := If[PosIntQ[h], h, ToExpression[StringJoin[ToString2[Flatten[h]]]]]
End[]

Begin["`FreeQ1`"]
FreeQ1[ex1_, ex2_] := Module[{h}, Quiet[FreeQ[Subs[ex1, ex2, h = Unique["ArtKr"]], h]]]
End[]

Begin["`FreeQ2`"]
FreeQ2[x_, p_] := If[ListQ[p], If[DeleteDuplicates[Map10[FreeQ, x, p]] === {True}, True, False], FreeQ[x, p]]
End[]

Begin["`Uprocs`"]
Uprocs[] := Module[{a, b, c, d, h, g, k, t1, t2}, a := "_$Art27_Kr20$_.txt"; {c, g} = {{}, {}}; Save[a, "`*"];
    b := Map[ToString, Flatten[DeleteDuplicates[ReadList[a, String]]]];
    For[k = 1, k ≤ Length[b], If[StringCount[First[b[[{k}]]], " := Module[{"] ≠ 0 && StringTake[First[b[[{k}]]], {1}] ≠ " " ||
          StringCount[First[b[[{k}]]], " := Block[{"] ≠ 0 && StringTake[First[b[[{k}]]], {1}] ≠ " ",
        AppendTo[c, First[b[[{k}]]]], Null]; k = k + 1];
    For[k = 1, k ≤ Length[c], d = Quiet[First[c[[{k}]]]];
      h = Quiet[Symbol[StringTake[d, First[First[StringPosition[d, "["]]] - 1]]];
      t1 = If[StringCount[d, " := Module[{"] ≠ 0, Module, Block];
      t2 = Quiet[StringTake[d, Last[First[StringPosition[d, "]"]]]]];
      If[BlockModQ[h], AppendTo[g, {h, t2, t1}], Null]; k = k + 1]; DeleteFile[a]; g]
End[]

Begin["`ActCsProcFunc`"]
ActCsProcFunc[] := Module[{a = Names["Global`*"], b = {"Procedure"}, c = {"Function"}, h = {}, d, k = 1, t, v},
    Map[If[TemporaryQ[#] || HeadPF[#] === #, Null, AppendTo[h, ToString[t = Unique["g"]]];
          v = BlockFuncModQ[#, t]; If[v && MemberQ[{"Block", "Module"}, t], AppendTo[b, {#, HeadPF[#]}],
            If[v && t === "Function", AppendTo[c, {#, HeadPF[#]}]]]] &, a]; Map[Remove, h]; {b, c}]
End[]

Begin["`TemporaryQ`"]
TemporaryQ[x_] :=
  If[SymbolQ[x], MemberQ[{"Attributes[" <> ToString[x] <> "] = {Temporary}", "Null"}, ToString[Definition[x]]], False]
End[]

Begin["`ListOp`"]
ListOp[x_ /; ListQ[x], y_ /; ListQ[y], z_ /; HowAct[z]] := Module[{a = Length[x], b = Length[y], c, d = {}, k = 1}, c = Min[a, b];
    For[k, k ≤ c, k++, d = Append[d, z[x[[k]], y[[k]]]]];
    Flatten[{d, x[[c + 1 ;; -1]], y[[c + 1 ;; -1]]}]]
End[]

Begin["`ListTrim`"]
ListTrim[L_ /; ListQ[L], expr_] := Module[{a = Flatten[{expr}], k = 1, j = Length[L]},
    For[k, k ≤ Length[L], k++, If[! MemberQ[a, L[[k]]], Break[]]]; For[j, j ≥ 1, j--, If[! MemberQ[a, L[[j]]], Break[]]];
    L[[k ;; j]]]
End[]

Begin["`ProcQ`"]
```

```
ProcQ[x_] := Module[{a, atr = Quiet[Attributes[x]], b, c, d, h}, If[! SymbolQ[x], False,
      If[SystemQ[x], False, If[UnevaluatedQ[Definition2, x], False, If[ListQ[atr] && atr ≠ {}, ClearAllAttributes[x];
          a = StringReplace[ToString[InputForm[Definition[x]]], Context[x] <> ToString[x] <> "`" –> ""];
          Quiet[b = StringTake[a, {1, First[First[StringPosition[a, {" := Block[{", " :=Block[{"}] – 1]]}];
            c = StringTake[a, {1, First[First[StringPosition[a, {" := Module[{", " :=Module[{"}] – 1]]}];
            d = StringTake[a, {1, First[First[StringPosition[a, {" := DynamicModule[{", " :=DynamicModule[{"}] – 1]]}]];
          If[b === ToString[HeadPF[x]], SetAttributes[x, atr]; True, If[c === ToString[HeadPF[x]], SetAttributes[x, atr];
              True, If[d === ToString[HeadPF[x]], SetAttributes[x, atr]; True, SetAttributes[x, atr]; False]]]]]]]
End[]

Begin["`ProcQ1`"]
ProcQ1[x_, y___ /; y === Null || SymbolQ[y] && ! HowAct[y]] :=
  Module[{a = Quiet[Check[Flatten[{PureDefinition[x]}], $Failed]], b = StringLength[ToString[x]], c, g = ToString[Unique["agn"]],
      h = {}, p = $$$74, k = 1, t = {}, v = {}}, If[SubsetQ[{$Failed, "System"}, a], False, For[k, k ≤ Length[a], k ++,
        Clear[$$$74]; ToExpression[g <> StringTake[a[[k]], {b + 1, –1}]]; AppendTo[h, c = ProcQ[g]]; AppendTo[v, g];
        BlockFuncModQ[g, $$$74]; AppendTo[t, If[c && $$$74 == "Function", "DynamicModule", $$$74]];
        g = ToString[Unique["agn"]]]; $$$74 = p; Map[Remove, Flatten[{g, v}]];
      If[{y} ≠ {}, y = {h, t}, Null]; If[DeleteDuplicates[h] == {True}, True, False]]]
End[]

Begin["`ProcBMQ`"]
ProcBMQ[x_ /; BlockModQ[x], y___] := Module[{a, b, c = " = ", d, p},
    If[! SingleDefQ[x], "Object <" <> ToString[x] <> "> has multiple definitions",
      If[ModuleQ[x], True, {a, b} = {PureDefinition[x], Locals1[x]};
        d = Map[#[[1]] – 1 &, StringPosition[a, c]];
        p = Map[ExprOfStr[a, #, –1, {" ", "{", "["}] &, d];
        p = DeleteDuplicates[Flatten[Map[StrToList, p]]];
        If[{y} ≠ {}, y = MinusList[b, p], Null]; If[p == b, True, False]]]]
End[]

Begin["`ProcFuncTypeQ`"]
ProcFuncTypeQ[x_ /; StringQ[x]] := Module[{a, b, d = {}, k = 1, p},
    If[ShortPureFuncQ[x], {True, "ShortPureFunction"}, a = Definition2[x][[If[Options[x] == {}, 1 ;; –2, 1 ;; –3]]];
      If[SuffPref[x, "Function[{", 1], {True, "PureFunction"},
        If[a === Definition2[], {False, x, "Expression"}, If[SysFuncQ[x], {False, x, "System"},
            For[k, k ≤ Length[a], k ++, b = Map[StringSplit[#, {" := ", " = "}] &, a]; p = b[[k]][[2]];
              If[SuffPref[p, {"Compile[{", "CompiledFunction[{"}, 1], AppendTo[d, "CompiledFunction"], If[SuffPref[p, "Block[{", 1],
                  AppendTo[d, "Block"], If[SuffPref[p, "Module[", 1], AppendTo[d, "Module"], If[SuffPref[p, "Function[{", 1],
                    AppendTo[d, "PureFunction"], If[ShortPureFuncQ[p], AppendTo[d, "ShortPureFunction"],
                      If[PureFuncQ[ToExpression[x]], AppendTo[d, "PureFunction"], AppendTo[d, "Function"]]]]]]];
            {True, d}]]]]]
End[]

Begin["`$Line1`"]
$Line1 := Module[{a = "", c = "Out[", k = 1}, For[k, k < Infinity, k ++, a = ToString[Out[k]];
      If[a == c <> ToString[k] <> "]", Return[k]]]; k]
End[]

Begin["`PackNames`"]
PackNames[x_ /; FileExistsQ[x] && MemberQ[{"m", "nb"}, FileExtension[x]]] :=
  Module[{a = StringReplace[ReadFullFile[x], {" " –> "", "\n" –> ""}], b, c = {}, d = "", k = 1, j, h},
    If[FileExtension[x] == "m", Sort[Map[StringTake[#, {9, –4}] &, StringCases[a, Shortest["Begin[\"`" ~~ __ ~~ "`\"]"]]]],
      b = Quiet[SubsString1[a, {"RowBox[{RowBox[{RowBox[{", "\"::\"", "\"usage\"}]"}, StringQ[#] &, 0]];
      b = Map[StringTake[#, {2, –6}] &, b]; Select[Sort[(If[SymbolQ[#1], #1] &) /@ b], ! SameQ[#, Null] &]]]
End[]

Begin["`PackNames1`"]
PackNames1[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{a = ReadFullFile[x], b = ContextFromMx[x]}, a = StringCases[a, Shortest[b ~~ __ ~~ "`"]];
    a = Select[a, StringFreeQ[#, {"𝒹", "[", "{"}] &]; Sort[Map[StringReplace[#, {b –> "", "`" –> ""}] &, a]]]
End[]

Begin["`PackNames2`"]
PackNames2[x_ /; FileExistsQ[x] && FileExtension[x] == "nb"] :=
  Module[{a = StringReplace[ReadFullFile[x], {" " –> "", "\n" –> ""}], b, c},
    a = StringCases[a, Shortest["RowBox[{\"Begin\",\"[\",\"\\\"\\<`" ~~ __ ~~ "`\\>\\\"\",\"]\"}]"]];
    a = Select[a, StringCount[#, "RowBox"] == 1 &];
    Sort[DeleteDuplicates[
        Map[{c = DeleteDuplicates[Flatten[StringPosition[#, "`"]]], StringTake[#, {c[[1]] + 1, c[[2]] – 1}]}[[2]] &, a]]]
```

```
End[]

Begin["`PackageQ`"]
PackageQ[x_ /; ContextQ[x]] := If[CNames[x] ≠ {}, True, False]
End[]

Begin["`Packages`"]
Packages[] := Select[$Packages, Quiet[PackageQ[#]] &]
End[]
Begin["`ListStringQ`"]
ListStringQ[x_] := ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True}
End[]

Begin["`SaveCurrentSession`"]
SaveCurrentSession[x___String] :=
  Module[{a = Names["*"], b = If[{x} == {}, "SaveCS.m", If[SuffPref[x, ".m", 2], x, x <> ".m"]]}, Save1[b, a];
    b]
End[]

Begin["`RestoreCS`"]
RestoreCS[x___String] := Module[
    {a = If[{x} == {}, "SaveCS.m", If[FileExistsQ[x] && FileExtension[x] == "m", x, $Failed]]}, If[a === $Failed, $Failed, On[General];
      Quiet[Get[a]];
      Off[General]]]
End[]

Begin["`DumpSaveP`"]
DumpSaveP[f_ /; StringQ[f], x_ /; ContextQ[x]] := If[PackageQ[x], DumpSave[f, x], $Failed]
End[]

Begin["`DumpSave1`"]
DumpSave1[x_, y_] := Module[{a, b, c}, If[StringQ[x], If[FileExtension[x] == "mx", c = x, c = x <> ".mx"];  a = Flatten[{y}];
      b = Select[a, (ContextQ[#] && MemberQ[$ContextPath, #]) || ! MemberQ[{"", "Null"}, Quiet[ToString[Definition[#]]]] &];
      If[b ≠ {}, {c, Flatten[DumpSave[c, b]]}, $Failed], $Failed]]
End[]

Begin["`DumpSave2`"]
DumpSave2[x_ /; FileExtension[x] == "mx", y_ /; SymbolQ[y] || ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True},
    z_ /; ContextQ[z]] := Module[{b, c, a = Select[Flatten[{y}], ! SameQ[PureDefinition[#], $Failed] &]},
    If[a == {}, $Failed, a = Map[ToString, a];
      Map[ToExpression["ContextToSymbol[" <> ToString[#] <> "," <> ToString1[z] <> "]"] &, a];  DumpSave[x, z]; ]]
End[]

Begin["`SaveInMx`"]
SaveInMx[x_ /; FileExtension[x] == "mx", y_ /; SymbolQ[y] || ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True},
    z_ /; ContextQ[z]] := Module[{a = Flatten[Select[Map[PureDefinition[#] &, Flatten[{y}]], ! SameQ[#, $Failed] &]], b},
    Map[ToExpression[z <> #] &, a];  $ContextPath = AppendTo[$ContextPath, z];
    DumpSave[x, z]]
End[]

Begin["`HS`"]
HS[LTF_ /; ListQ[LTF], CF_ /; StringQ[CF], P_ /; IntegerQ[P]] :=
  Module[{n = 0, k, C1, Co, CFo, CFf, H, t = 0, Z}, n = StringLength[First[First[LTF]]];
    C1 = "";
    Co = "0";
    Do[C1 = C1 <> Co, {k, 1, n − 1}];
    CFo = C1 <> CF <> C1; CFf = ""; Label[St]; T1 = TimeUsed[];
    Do[CFf = CFf <> StringReplace[StringTake[CFo, {k, k + n − 1}], LTF], {k, 1, StringLength[CFo] + 1 − n}];
    t++;
    T2 = TimeUsed[];
    If[T2 − T1 ≥ 15, Goto[Q], 7]; Goto[C]; Label[Q];
    Print["Steps: " <> ToString[t] <> "; CFfin: " <> CFf]; Z = Input["Continue(y/n)?"];
    If[Z == y, Goto[C], 7]; Return["Job is canceled in step: " <> ToString[t]]; Label[C];
    If[H = Length[StringPosition[CFf, CF]]; H ≥ P, Goto[Fin], CFo = C1 <> CFf <> C1; CFf = ""];
    Goto[St]; Label[Fin]; Print["Initial configuration: " <> CF <> " is " <> ToString[H] <> "−
reproductable!"]; CFfin = CFf; Print["Steps: " <> ToString[t] ]]
End[]

Begin["`Predecessors`"]
```

```
Predecessors[Ltf_ /; ListQ[Ltf], Co_ /; StringQ[Co], n_ /; IntegerQ[n]] :=
  Module[{L, a, b, c, h = {}, i, j, k, d = StringLength[Co]}, a = Gather[Ltf, StringTake[#1, -1] == StringTake[#2, -1] &];
    For[k = 1, k ≤ Length[a], k++, L[StringTake[First[a[[k]]], -1]] = Map2[StringDrop, Map[ToString1, a[[k]]], {-1}]];
    b = L[StringTake[Co, 1]];
    For[k = 2, k ≤ d, k++, c = L[StringTake[Co, {k, k}]];
      For[i = 1, i ≤ Length[b], i++, For[j = 1, j ≤ Length[c], j++,
          If[SuffPref[b[[i]], StringTake[c[[j]], n - 1], 2], h = Append[h, b[[i]] <> StringTake[c[[j]], -1]], Null]]];
      b =
        h;
      h = {}];
    If[Length[b] ≠ (n - 1)^Length[a], Print["Structure possesses the nonconstructability of NCF-type"], Null]; b]
End[]

Begin["`PredecessorsL`"]
PredecessorsL[Ltf_ /; ListQ[Ltf], Co_ /; StringQ[Co], n_ /; IntegerQ[n], Cf_ /; SymbolQ[Cf]] :=
  Module[{L, a, b, c, h = {}, i, j, k, d = StringLength[Co]}, a = Gather[Ltf, StringTake[#1, -1] == StringTake[#2, -1] &];
    For[k = 1, k ≤ Length[a], k++, L[StringTake[First[a[[k]]], -1]] = Map2[StringDrop, Map[ToString1, a[[k]]], {-1}]];
    b = L[StringTake[Co, -1]];
    For[k = d - 1, k ≥ 1, k--, c = L[StringTake[Co, {k, k}]];
      For[i = 1, i ≤ Length[b], i++, For[j = 1, j ≤ Length[c], j++,
          If[StringTake[b[[i]], n - 1] === StringTake[c[[j]], {2, -1}], h = Append[h, StringTake[c[[j]], 1] <> b[[i]]], Null]]];
      b = h; h = {}]; Cf = b; Length[b]]
End[]

Begin["`PredecessorsR`"]
PredecessorsR[Ltf_ /; ListQ[Ltf], Co_ /; StringQ[Co], n_ /; IntegerQ[n], Cf_ /; SymbolQ[Cf]] :=
  Module[{L, a, b, c, h = {}, i, j, k, d = StringLength[Co]}, a = Gather[Ltf, StringTake[#1, -1] == StringTake[#2, -1] &];
    For[k = 1, k ≤ Length[a], k++, L[StringTake[First[a[[k]]], -1]] = Map2[StringDrop, Map[ToString1, a[[k]]], {-1}]];
    b = L[StringTake[Co, 1]];
    For[k = 2, k ≤ d, k++, c = L[StringTake[Co, {k, k}]];
      For[i = 1, i ≤ Length[b], i++, For[j = 1, j ≤ Length[c], j++,
          If[SuffPref[b[[i]], StringTake[c[[j]], n - 1], 2], h = Append[h, b[[i]] <> StringTake[c[[j]], -1]], Null]]];
      b = h; h = {}]; Cf = b; Length[b]]
End[]

Begin["`NcfQ`"]
NcfQ[Ltf_ /; ListQ[Ltf], S_ /; StringQ[S], t_ /; IntegerQ[t] && MemberQ[Range[0, 9], t]] :=
  Module[{n = StringLength[Part[Ltf[[1]], 1]], a, b, c = "", d = {}, k = 1, j, p}, a = Tuples[Range[0, t], p = StringLength[S] + n - 1];
    b = Map[Map[ToString, #] &, a];
    b = Map[StringJoin, b];
    For[k, k ≤ Length[b], k++, For[j = p - n + 1, j ≥ 1, j--, c = StringReplace[StringTake[b[[k]], {j, j + n - 1}], Ltf] <> c];
      d = DeleteDuplicates[AppendTo[d, c]];
      c = ""];
    If[! MemberQ[d, S], True, False]]
End[]

Begin["`MinNCF`"]
MinNCF[Ltf_ /; ListQ[Ltf], n_ /; IntegerQ[n]] := Module[{a, k = 1},
    a = Map[StringJoin, Map[Map[ToString, #] &, Flatten[Map[Tuples[{0, 1}, #] &, Range[1, n]], 1]]];
    For[k, k ≤ Length[a], k++, If[NcfQ[Ltf, a[[k]], 1], Return[a[[k]]], Continue[]]]]
End[]

Begin["`NfToLtf`"]
NfToLtf[m_ /; IntegerQ[m], n_ /; IntegerQ[n]] := Module[{a = IntegerDigits[n, 2], b, c, d = {}},
    If[n ≥ 2^2^m, Defer[NfToLtf[m, n]], b = Map[ToString, Join[Table[0, {2^m - Length[a]}], a]];
      c = Map[StringJoin, Map[Map[ToString, #] &, Flatten[Map[Tuples[{0, 1}, #] &, Range[m, m]], 1]]];
      Do[AppendTo[d, Rule[c[[k]], b[[k]]]], {k, 1, Length[b]}];
      d]]
End[]

Begin["`ComposeGTF`"]
ComposeGTF[g__] := Module[{j = 1, cgtf, v = {g}, s = Length[{g}], r, u, x},
    If[
      s < 4 || DeleteDuplicates[Flatten[Map[{IntegerQ[v[[2*# - 1]]], IntegerQ[v[[2*#]]] && 0 ≤ v[[2*#]] ≤ 2^2^v[[2*# - 1]] - 1} &,
              Range[1, Floor[s/2]]]]] ≠ {True}, Return[Defer[ComposeGTF[g]]],
      cgtf[n_Integer, m_ /; IntegerQ[m], n1_Integer, m1_ /; IntegerQ[m1]] :=
        Module[{a = NfToLtf[n, m], b = NfToLtf[n1, m1], c = n + n1 - 1, d = {}, h, k = 1, p, t},
          h = Map[StringJoin, Map[Map[ToString, #] &, Flatten[Map[Tuples[{0, 1}, #] &, Range[c, c]], 1]]];
          For[k, k ≤ Length[h], k++, p = h[[k]];
```

```
            t = StringTake[p, Map[{# + 1, n + #} &, Range[0, n1 − 1]]]; t = StringJoin[Map[StringReplace[#, a] &, t]];
            d = AppendTo[d, Rule[p, StringReplace[t, b]]]];
          {c, FromDigits[ToExpression[Map[Part[#, 2] &, d]], 2]}];
      r = cgtf[v[[1]], v[[2]], v[[3]], v[[4]]];
      For[j = 3, j ≤ Floor[s / 2], j ++, r = cgtf[r[[1]], r[[2]], v[[2 * j − 1]], v[[2 * j]]]];
      If[EvenQ[s], r, u = Map[StringJoin, Map[Map[ToString, #] &, Flatten[Map[Tuples[{0, 1}, #] &, Range[r[[1]], r[[1]]]], 1]]];
        x = Map[ToString, IntegerDigits[r[[2]], 2, 2 ^ r[[1]]]]; Table[Rule[u[[j]], x[[j]]], {j, 1, 2 ^ r[[1]]}]]]]
End[]

Begin["`GtfMod2`"]
GtfMod2[n_ /; IntegerQ[n]] :=
  Module[{a = Flatten[Map[Tuples[{0, 1}, #] &, Range[n, n]], 1], b, c}, FromDigits[Map[Mod[#, 2] &, Map[Total[#] &, a]], 2]]
End[]

Begin["`CFsequences`"]
CFsequences[Co_ /; StringQ[Co] && Co != "", A_ /; ListQ[A] && MemberQ[Map[Range[0, #] &, Range[9]], A],
    Ltf_ /; ListQ[Ltf] && DeleteDuplicates[Map[RuleQ[#] &, Ltf]] == {True} || FunctionQ[Ltf], n_ /; IntegerQ[n] && n >= 0] :=
  Module[{a = StringTrim2[Co, "0", 3], b, c, t1 = {}, t2 = {}, t3 = {}, t4 = {}, tf, p = n}, If[! MemberQ3[Map[ToString, A], Characters[Co]],
      Print["Initial configuration <" <> Co <> "> is incorrect"]; $Failed, If[FunctionQ[Ltf], b = Arity[Ltf],
        Map[{{AppendTo[t1, StringQ[#[[1]]]], AppendTo[t2, StringLength[#[[1]]]]}, {AppendTo[t3, StringQ[#[[2]]]],
              AppendTo[t4, StringLength[#[[2]]]]}} &, Ltf]; b = Map[DeleteDuplicates[#] &, {t1, t2, t3, t4}];
        If[! (MemberQ3[{True}, {b[[1]], b[[3]]}] && Map[Length, {b[[2]], b[[4]]}] == {1, 1} &&
              Length[t2] == Length[A] ^ (b = b[[2]][[1]])), Print["Local transition function is incorrect"];
          Return[$Failed],
          tf = Map[ToExpression[Characters[#[[1]]]] → ToExpression[#[[2]]] &, Ltf]]; c = StringMultiple1["0", b];
      Print[a];
      While[p > 0, p −−;
        a = c <> a <> c;
        a = Partition[ToExpression[Characters[a]], b, 1]; a = If[FunctionQ[Ltf], Map[Ltf @@ # &, a], ReplaceAll[a, tf]];
        a = StringJoin[Map[ToString, a]]; Print[StringTrim2[a, "0", 3]];]]
End[]

Begin["`GtfMod2Q`"]
GtfMod2Q[n_ /; IntegerQ[n], m_ /; IntegerQ[m]] := Module[{a}, a = ComposeGTF[n, GtfMod2[n], m, GtfMod2[m]];
    If[a[[2]] == GtfMod2[n + m − 1], True, False]]
End[]

Begin["`FunctionToRules`"]
FunctionToRules[x_ /; SymbolQ[x], A_ /; ListQ[A]] :=
  Map[StringJoin[Map[ToString, #]] → ToString[x @@ #] &, Tuples[A, Arity[x]]]
End[]

Begin["`XOR1`"]
XOR1[n_ /; ListQ[n] && Length[n] ≥ 1 && DeleteDuplicates[Map[IntegerQ[#] &, n]] == {True}] :=
  Module[{a, b, c, d}, If[Length[n] == 1, {n}[[1]], a = Map[IntegerDigits[#, 2] &, n];
      b = Sort[Map[Length[#] &, a]][[−1]]; b = Map[PadLeft[#, b] &, a]]; b = Map[Mod[#, 2] &, Total[b]]; FromDigits[b, 2]]
End[]

Begin["`ReprodXOR1`"]
ReprodXOR1[S_ /; StringQ[S], n_ /; IntegerQ[n], m_ /; IntegerQ[m], g___] :=
  Module[{a, b, c = "", h, k, d = 0, t}, a = StringJoin[Map[ToString, NestList[Sin, 0, n − 2]]];
    b = a <> S <> a;
    Label[AGN]; For[k = StringLength[b] − n + 1, k ≥ 1, k −−, h = ToExpression[Characters[StringTake[b, {k, k + n − 1}]]];
      c = ToString[XOR1[h]] <> c]; If[t = StringCount[c, S]; t ≥ m, If[Length[{g}] ≠ 0 && ! ValueQ[g], g = c];
      {StringLength[c], t}, b = a <> c <> a; c = ""; d = d + 1; Goto[AGN]]]
End[]

Begin["`ReprodXOR11`"]
ReprodXOR11[S_ /; ListQ[S], n_ /; IntegerQ[n], m_ /; IntegerQ[m], g___] :=
  Module[{b, c = {}, k, j = 1, d = 0, t, h = Length[S]}, b = PadLeft[PadRight[S, h + n − 1, 0], h + 2 * (n − 1), 0];
    Label[ArtKr]; For[k = Length[b] − n + 1, k ≥ 1, k −−, c = Prepend[c, XOR1[b[[k ;; −j ++]]]]];
    If[t = ListCount[c, S];
      t ≥ m, If[Length[{g}] ≠ 0 && ! ValueQ[g], g = c];
      {Length[c], t}, b = PadLeft[PadRight[c, Length[c] + n − 1, 0], Length[c] + 2 * (n − 1), 0];
      c = {};
      d = d + 1;
      j = 1;
      Goto[ArtKr]]]
End[]
```

```
Begin["`CodeEncode`"]
CodeEncode[x_] := Module[{a = ToString[x], b = Map[{#, Prime[#]} &, Range5[2 ;; 126]], c = ""}, d = ToCharacterCode[a];
    If[Max[d] ≤ 126, Do[c = c <> FromCharacterCode[Select[b, #[[If[Max[d] ≤ 126, 1, 2]]] == d[[k]] &][[1]][[2]]], {k, 1, Length[d]}],
      Do[c = c <> FromCharacterCode[Select[b, #[[2]] == d[[k]] &][[1]][[1]]], {k, 1, Length[d]}]]; c]
End[]

Begin["`CodeEncode1`"]
CodeEncode1[x_ /; FileExistsQ[x] || StringQ[x]] := Module[{a, b, c}, If[FileExistsQ[x],
      a = ReadString[x, "\n" | "\n\n" | "\r\n"];
      If[Quiet[Check[StringCases[a, "Attributes[a$" ~~ Shortest[X__] ~~ "] = {Temporary}"][[1]], $Failed]] === $Failed,
        a = CodeEncode[ReadString[x]]; DeleteFile[x]; Save[x, a], a = Get[x]; a = CodeEncode[a];
        a = If[SuffPref[a, "\"", 1] && SuffPref[a, "\r\n", 2], ToExpression[StringTake[a, {1, -3}]], a];
        WriteString[x, a]; Close[x]; CodeEncode[CodeEncode[a]]], CodeEncode[x]]]
End[]

Begin["`SubConf`"]
SubConf[Ltf_ /; ListQ[Ltf], Cf_ /; StringQ[Cf], p_ /; IntegerQ[p], h_ /; StringQ[h]] :=
  Module[{n = StringLength[Part[Ltf[[1]], 1]], a, b, c = "", k, d = 0, t}, a = StringJoin[Map[ToString, NestList[Sin, 0, n - 2]]];
    b = a <> h <> a; Label[AVZ]; For[k = StringLength[b] - n + 1, k ≥ 1, k--,
      c = StringReplace[StringTake[b, {k, k + n - 1}], Ltf] <> c]; b = a <> c <> a; c = "";
    d = d + 1; If[t = StringCount[b, Cf]; t ≥ p, {d, t}, Goto[AVZ]]]
End[]

Begin["`ToLTF`"]
ToLTF[A_ /; ListQ[A], n_ /; IntegerQ[n], F_ /; SymbolQ[F], p_ /; IntegerQ[p]] := Module[{a = Tuples[A, n], b = {}, k = 1},
    For[k, k ≤ Length[a], k++, b = AppendTo[b, Rule[StringJoin[Map[ToString, a[[k]]]], ToString[F[a[[k]], p]]]]];
    b]
End[]

Begin["`HSD`"]
HSD[A_ /; ListQ[A], delays_ /; ListQ[delays], Cf_ /; ListQ[Cf], ltf_ /; ListQ[ltf], n_ /; IntegerQ[n], p_ /; IntegerQ[p]] :=
  Module[{a = Partition[Riffle[A, delays], 2], b = Table[{0, 0}, {n - 1}], c, d = {}, k, F, t, h, g},
    F[x_List] := Part[Select[ltf, Part[#, 1] == x &][[1]], 2]; g = Map[{#, delays[[Flatten[Position[a, #]][[1]]]]} &, Cf];
    c = Join[b, g, b];
    Do[ c = Map[{#[[1]], If[#[[1]] == 0 && #[[2]] == 0, 0, #[[2]] - 1]} &, c]; For[k = 1, k ≤ Length[c] - n + 1,
        k++, t = c[[k]]; If[t[[2]] ≠ 0, d = AppendTo[d, t], h = c[[k ;; k + n - 1]]; h = F[Map[#[[1]] &, h]];
          d = AppendTo[d, {h, delays[[Flatten[Position[A, 1]][[1]]]]}]]];
      c = Join[b, d, b];
      d = {}, {p}];
    Map[#[[1]] &, c[[n ;; -n]]]]
End[]

Begin["`ReprodHSD`"]
ReprodHSD[A_ /; ListQ[A], delays_ /; ListQ[delays], Cf_ /; ListQ[Cf], ltf_ /; ListQ[ltf], n_ /; IntegerQ[n], p_ /; IntegerQ[p],
    v_ /; IntegerQ[v]] := Module[{a = Partition[Riffle[A, delays], 2], b = Table[{0, 0}, {n - 1}], c, d = {}, k, F, t, h, g, s},
    F[x_List] := Part[Select[ltf, Part[#, 1] == x &][[1]], 2]; g = Map[{#, delays[[Flatten[Position[a, #]][[1]]]]} &, Cf];
    c = Join[b, g, b];
    Do[ c = Map[{#[[1]], If[#[[1]] == 0 && #[[2]] == 0, 0, #[[2]] - 1]} &, c]; For[k = 1, k ≤ Length[c] - n + 1,
        k++, t = c[[k]]; If[t[[2]] ≠ 0, d = AppendTo[d, t], h = c[[k ;; k + n - 1]]; h = F[Map[#[[1]] &, h]];
          d = AppendTo[d, {h, delays[[Flatten[Position[A, 1]][[1]]]]}]]];
      t = Map[#[[1]] &, d];
      If[s = ListCount[t, Cf];
        s ≥ p, Return[{p, s}]];
      c = Join[b, d, b];
      d = {}, {v}]]
End[]

Begin["`ReprodHSM`"]
ReprodHSM[Cf_ /; ListQ[Cf], n_ /; IntegerQ[n],
    p_ /; IntegerQ[p] && p > 0, m_ /; IntegerQ[m], g_ /; IntegerQ[g], v_ /; IntegerQ[v]] :=
  Module[{a = Flatten[Table[Table[0, {p + 1}], {n - 1}]], b, c, d = {}, k, G, h, t, s}, b = Join[{a}, Map[Flatten[{#, a[[1 ;; p]]}] &, Cf], {a}];
    G[x_List] := Mod[Total[x], m]; Do[For[k = 1, k ≤ Length[b] - n + 1, k++, h = b[[k ;; k + n - 1]];
        c = G[Flatten[{Map[Part[#, 1] &, h], b[[k]][[2 ;; -1]]}]]; d = AppendTo[d, Flatten[{c, b[[k]][[3 ;; -1]], c}]]];
      t = Map[#[[1]] &, d]; If[s = ListCount[t, Cf]; s ≥ g, Return[{g, s}]]; b = Join[{a}, d, {a}]; d = {}, {v}]]
End[]

Begin["`ConvertMtoMx`"]
ConvertMtoMx[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y_ /; Head[y] == Symbol, z___] :=
```

```
  Module[{a, b, c, d, h = {}, f = StringTake[x, {1, -2}] <> "mx", p = "\""},
    a = StringReplace[StringJoin[Map[FromCharacterCode, BinaryReadList[x]]],
        {"(**)" -> "", "*)\r\n(*" -> "\r", "*)\r\n\r\n(*" -> "\r", "*)\r\n\r\n(\r\n*" -> "\r", "(* ::" -> "::", "End[]" -> "" }];
    a = StringTake[a, {Flatten[StringPosition[a, "BeginPackage["]][[1]] - 1, -4}];
    b = StringCases[a, "Begin[\"`" ~~ Shortest[X__] ~~ "`\"]"];
    c = StringTake[StringCases[a, "BeginPackage[\"" ~~ Shortest[X__] ~~ "\"]"][[1]], {14, -2}];
     Quiet[ToExpression[StringTake[StringReplace[a, Map[Rule[#, ""] &, b]], {1, -2}]]];
    ToExpression["DumpSave[" <> p <> f <> p <> "," <> c <> "]"];
    y = Sort[Map[StringTake[#, {9, -4}] &, b]];  If[{z} ≠ {}, Map[ClearAll, y];
      d = "ClearAttributes[{$Packages,Contexts},Protected]; $Packages=Select[$Packages,StringFreeQ[#,g]&];
           Contexts[]=Select[Contexts[],StringFreeQ[#,g]&]; SetAttributes[{$Packages,Contexts},Protected];
           $ContextPath=Select[$ContextPath,StringFreeQ[#,g]&]";
      ToExpression[StringReplace[d, ",g" → "," <> c]];, Null]]
End[]

Begin["`ContUsageMfile`"]
ContUsageMfile[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y___] :=
  If[{y} ≠ {} && Head[y] == Symbol, StringReplace[StringTake[StringCases[StringJoin[Map[FromCharacterCode, BinaryReadList[x]]],
           ToString[y] <> "::usage="  ~~ Shortest[X__] ~~ "(**)"][[1]], {1, -9}], "::usage=" -> "::", 1],
    Sort[Map[StringTake[#, {9, -4}] &, StringCases[StringJoin[Map[FromCharacterCode, BinaryReadList[x]]],
          "Begin[\"`" ~~ Shortest[X__] ~~ "`\"]"]]]
End[]

Begin["`CodeObjMfile`"]
CodeObjMfile[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y_ /; Head[y] == Symbol] := StringTake[
    StringReplace[StringCases[StringJoin[Map[FromCharacterCode, BinaryReadList[x]]], "Begin[\"`" <> ToString[y] <> "`\"]" ~~
          Shortest[W__] ~~ "End[]"][[1]], "*)\r\n(*" -> "\r"], {13 + StringLength[ToString[y]], -7}]
End[]

Begin["`ContCodeUsageM`"]
ContCodeUsageM[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y___] :=
  Module[{a = StringJoin[Map[FromCharacterCode, BinaryReadList[x]]], b = {y}}, If[Length[b] > 1 && Head[b[[1]]] == Symbol,
      StringTake[StringReplace[StringCases[a, "Begin[\"`" <> ToString[b[[1]]] <> "`\"]" ~~ Shortest[W__] ~~ "End[]"][[1]],
          "*)\r\n(*" → "\r"], {13 + StringLength[ToString[b[[1]]]], -7}], If[b ≠ {} && Head[b[[1]]] == Symbol, StringReplace[
        StringTake[StringCases[a, ToString[b[[1]]] <> "::usage=" ~~ Shortest[W__] ~~ "(**)"][[1]], {1, -9}], "::usage=" → "::", 1],
       Sort[DeleteDuplicates[Map[StringTake[#, {9, -4}] &, StringCases[a, "Begin[\"`" ~~ Shortest[W__] ~~ "`\"]"]]]]], $Failed]]
End[]

Begin["`ReprodHSM1`"]
ReprodHSM1[Cf_ /; ListQ[Cf], n_ /; IntegerQ[n],
    p_ /; IntegerQ[p] && p > 0, m_ /; IntegerQ[m], g_ /; IntegerQ[g], v_ /; IntegerQ[v]] := Module[
    {a = Flatten[Table[Table[0, {p + 1}], {n - 1}]], b, c, d = {}, k, G, h, t, s, x = 0}, b = Join[{a}, Map[Flatten[{#, a[[1 ;; p]]}] &, Cf], {a}];
    G[x_List] := Mod[Total[x], m];  Do[For[k = 1, k ≤ Length[b] - n + 1, k++, h = b[[k ;; k + n - 1]];
        c = G[Flatten[h]];
        d = AppendTo[d, Flatten[{c, b[[k]][[3 ;; -1]], c}]]];  t = Map[#[[1]] &, d];
      x++;  If[s = ListCount[t, Cf];
        s ≥ g, Return[{x, g, s}]];  b = Join[{a}, d, {a}];
      d = {}, {v}]]
End[]

Begin["`ReprodHSwVni`"]
ReprodHSwVni[A_ /; ListQ[A], Cf_ /; ListQ[Cf], m_ /; IntegerQ[m], g_ /; IntegerQ[g], v_ /; IntegerQ[v]] :=
  Module[{a = Table[0, {Length[A]}], b, c, d = {}, k, G, h, t, s, x = 0},
    G[x_ /; ListQ[x]] := Mod[Total[x], m]; b = Join[{0}, Cf, a];
     Do[For[k = 1, k < Length[b], k++, h = b[[k]]; h = b[[k ;; k + h + 1]]; c = G[Flatten[h]];
        d = AppendTo[d, c]];  x++;  If[s = ListCount[d, Cf];
        s ≥ g, Return[{x, g, s}]];  d = ListTrim[d, 0];
      b = Join[{0}, d, a];
      d = {}, {v}]]
End[]
ToExpression[PureDefinition[TestHeadingQ]]
ToExpression[PureDefinition[FileOpenQ]]
EndPackage[]
```