Reusable Software
A Collection of C-Modules

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Reusable Software - A Collection of C-Modules

Josef Grosch

Dec. 28, 2000

Document No. 30

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Breslauer Str. 64c
76139 Karlsruhe
Germany

Phone: +49-721-91537544
Fax: +49-721-91537543
Email: grosch@cocolab.com

Abstract

A brief description of a useful collection of reusable modules written in C is given. The modules are oriented towards compiler construction.

1. Overview

The following modules are available:

Module	Task
rMemory	dynamic storage (heap) with free lists
DynArray	dynamic and flexible arrays
StringM	string memory
Idents	identifier table - unambiguous encoding of strings
Sets	sets of scalar values (without run time checks)
Relation	binary relations between scalar values
Position	handling of source positions
Errors	error handler for parsers and compilers
Source	provides input for scanners
General	miscellaneous functions
rSystem	machine dependent code
rTime	access to cpu-time
rString	portable string handling
rGetopt	portable version of Unix getopt
rFsearch	support for searching files
rSrcMem	storage for source code

Almost all modules of this library that store data have functions for initialization and for finalization. The function names uniformly start with Begin or Close:

Module	Initialization	Finalization
Idents	BeginIdents	CloseIdents
StringM	BeginStringMemory	CloseStringMemory
rMemory	BeginrMemory	CloserMemory
Errors	BeginErrors	
Source		CloseSource
rFsearch	Begin_rFsearch	Close_rFsearch
rSrcMem	Begin_rSrcMem	Close_rSrcMem

In simple cases these functions do not have to be called because the modules are initialized automatically, by default. However, if a parser is to be executed repeatedly within a loop then explicit initialization and finalization is advantageous in order to avoid memory leaks. The order of the calls for initialization and finalization is significant and should be as follows:

```
   BeginrMemory         ();
   for (;;) {
      BeginStringMemory ();
      BeginIdents       ();
      BeginErrors       ();
      Parser            ();
      CloseIdents       ();
      CloseStringMemory ();
   }
   CloserMemory         ();
```

2. rMemory: dynamic storage (heap) with free lists

```
extern unsigned long MemoryUsed;
                        /* Holds the total amount of memory managed by  */
                        /* this module.                                 */

extern char *   Alloc           (unsigned long ByteCount);
                        /* Returns a pointer to dynamically allocated   */
                        /* space of size 'ByteCount' bytes.             */

extern void     Free            (unsigned long ByteCount, char * a);
                        /* The dynamically allocated space starting at  */
                        /* address 'a' of size 'ByteCount' bytes is     */
                        /* released.                                    */

extern void     WriterMemory    (void);
                        /* The internal data structure of this module   */
                        /* is printed to stdout. Useful for debugging.  */

extern void     BeginrMemory    (void);
                        /* The memory module is initialized.            */

extern void     CloserMemory    (void);
                        /* All memory managed by this module is         */
                        /* released.                                    */
```

3. DynArray: dynamic and flexible arrays

This module provides dynamic and flexible arrays. The size of a dynamic array is determined at run time. It must be passed to a procedure to create the array. The size of such an array is also flexible, that means it can grow to arbitrary size by repeatedly calling a procedure to extend it.

```
extern void MakeArray    (char * * ArrayPtr, unsigned long * ElmtCount,
                          unsigned long ElmtSize);
                         /* 'ArrayPtr' is set to the start address of a  */
                         /* memory space to hold an array of 'ElmtCount' */
                         /* elements each of size 'ElmtSize' bytes.      */

extern void ResizeArray  (char * * ArrayPtr, unsigned long * OldElmtCount,
                          unsigned long NewElmtCount, unsigned long ElmtSize);
                         /* The memory space for the array is changed    */
                         /* to 'NewElmtCount' elements.                  */

extern void ExtendArray  (char * * ArrayPtr, unsigned long * ElmtCount,
                          unsigned long ElmtSize);
                         /* The memory space for the array is increased  */
                         /* by doubling the number of elements.          */

extern void ShrinkArray  (char * * ArrayPtr, unsigned long * ElmtCount,
                          unsigned long ElmtSize);
                         /* The memory space for the array is reduced    */
                         /* by halving the number of elements.           */

extern void ReleaseArray (char * * ArrayPtr, unsigned long * ElmtCount,
                          unsigned long ElmtSize);
                         /* The memory space for the array is released.  */
```

Example:

```
# define         InitialSize    100
typedef ...      ElmtType;
typedef ElmtType ArrayType [100000];
unsigned long    ActualSize = InitialSize;
ElmtType *       ArrayPtr;

MakeArray (& ArrayPtr, & ActualSize, sizeof (ElmtType));

/* Case 1: continuously growing array */

Index = 0;
for (;;) {
   Index ++;
   if (Index == ActualSize)
      ExtendArray (& ArrayPtr, & ActualSize, sizeof (ElmtType));

   ArrayPtr [Index] = ... ;   /* access an array element */
    ... = ArrayPtr [Index];   /* "      "  "     "       */
}

/* Case 2: non-continuously growing array */

for (;;) {
   Index = ... ;
   while (Index >= ActualSize)
      ExtendArray (& ArrayPtr, & ActualSize, sizeof (ElmtType));

   ArrayPtr [Index] = ... ;   /* access an array element */
    ... = ArrayPtr [Index];   /* "      "  "     "       */
}

ReleaseArray (& ArrayPtr, & ActualSize, sizeof (ElmtType));
```

4. StringM: string memory

This module stores strings of variable length. It supports single byte character strings of type char [] and double byte character strings of type wchar_t [].

```
/* Support for single byte strings: char []                              */

typedef unsigned short * tStringRef;

extern  tStringRef NoString;        /* An empty string                   */

extern  tStringRef PutString1   (const char * s);
                        /* Stores string 's' in the string memory and    */
                        /* returns a handle to the stored string.        */
                        /* The string has to be null-terminated and may  */
                        /* not contain null characters.                  */

extern  tStringRef PutString    (const char * s, unsigned long length);
                        /* Stores string 's' of length 'length' in the   */
                        /* string memory and returns a handle to it.     */
                        /* The string has not to be null-terminated and  */
                        /* may contain null characters.                  */

extern  void    StGetString     (tStringRef r, char * s);
                        /* Returns the string 's' from the string        */
                        /* memory having the handle 'r'.                 */

extern unsigned short StLength  (tStringRef r);
                        /* Returns the length of the string having       */
                        /* the handle 'r'.                               */

extern unsigned short LengthSt  (tStringRef r);
                        /* Returns the length of the string having       */
                        /* the handle 'r'.                               */

extern char *   StGetCStr       (tStringRef r);
                        /* Returns the address of the string having      */
                        /* the handle 'r'.                               */

extern  rbool   IsEqualSt       (tStringRef r, char * s);
                        /* Compares the string having the handle 'r'     */
                        /* and the C string 's'.                         */
                        /* Returns true if both are equal.               */
                        /* Works only, if both strings have the same     */
                        /* length. This has to be checked before.        */

extern  void    WriteString     (FILE * f, tStringRef r);
                        /* The string having the handle 'r' is printed   */
                        /* on the file 'f'.                              */
```

```
/* Support for multi byte strings: wchar_t []                          */

typedef unsigned short * tWStringRef;

extern const tWStringRef NoWString; /* An empty wide character string   */

extern  tWStringRef PutWString1 (const wchar_t * s);
                        /* Stores string ’s’ in the string memory and   */
                        /* returns a handle to the stored string.       */
                        /* The string has to be null-terminated and may */
                        /* not contain null characters.                 */

extern  tWStringRef PutWString  (const wchar_t * s, unsigned long length);
                        /* Stores string ’s’ of length ’length’ in the  */
                        /* string memory and returns a handle to it.    */
                        /* The string has not to be null-terminated and */
                        /* may contain null characters.                 */

extern  void    StGetWString    (tWStringRef r, wchar_t * s);
                        /* Returns the string ’s’ from the string       */
                        /* memory having the handle ’r’.                */

extern wchar_t * StGetWCStr     (tWStringRef r);
                        /* Returns the address of the null-terminated   */
                        /* string having the handle ’r’.                */

extern  rbool   WIsEqualSt      (tWStringRef r, const wchar_t * s);
                        /* Compares the string having the handle ’r’    */
                        /* and the C string ’s’.                        */
                        /* Returns rtrue if both are equal.             */
                        /* Works only, if both strings have the same    */
                        /* length. This has to be checked before.       */

extern  void    WriteWString    (FILE * f, tWStringRef r);
                        /* The string having the handle ’r’ is printed  */
                        /* on the file ’f’.                             */

extern  void    WriteStringMemory (void);
                        /* The contents of the string memory is printed */
                        /* on standard output.                          */

extern  void    BeginStringMemory (void);
                        /* The string memory is initialized.            */

extern  void    CloseStringMemory (void);
                        /* The string memory is finalized.              */
```

5. Idents: identifier table - unambiguous encoding of strings

This module maps strings unambiguously to integers. It supports single byte character strings of type char [] and double byte character strings of type wchar_t [].

```
/* Support for single byte strings: char []                              */

typedef long    tIdent;

extern  tIdent  NoIdent; /* A default identifer (empty string)          */

extern  tIdent  MakeIdent1      (const char * string);
                        /* The string is mapped to a unique             */
                        /* identifier (an integer) which is returned.   */
                        /* The string has to be null-terminated and may */
                        /* not contain null characters.                 */

extern  tIdent  MakeIdent       (const char * string, int length);
                        /* The string (of length) is mapped to a unique */
                        /* identifier (an integer) which is returned.   */
                        /* The string has not to be null-terminated and */
                        /* may contain null characters.                 */

extern  void    GetString       (tIdent ident, char * string);
                        /* Returns the string whose identifier is 'ident'.*/

extern  tStringRef GetStringRef (tIdent ident);
                        /* Returns a handle to the string identified    */
                        /* by 'ident'.                                  */

extern  char *  GetCStr         (tIdent ident);
                        /* Returns the address of the string identified */
                        /* by 'ident'.                                  */

extern  int     GetLength       (tIdent ident);
                        /* Returns the length of the string whose       */
                        /* identifier is 'ident'.                       */

extern  tIdent  MaxIdent        (void);
                        /* Returns the currently maximal identifier.    */

extern  void    WriteIdent      (FILE * file, tIdent ident);
                        /* The string encoded by the identifier 'ident' */
                        /* is printed on the file.                      */

/* Support for multi byte strings: wchar_t []                            */

typedef long    tWIdent;

extern  tWIdent NoWIdent;       /* An empty identifier (empty string)   */

extern  tWIdent MakeWIdent1     (const wchar_t * string);
                        /* The string is mapped to a unique             */
                        /* identifier (an integer) which is returned.   */
                        /* The string has to be null-terminated and may */
                        /* not contain null characters.                 */
```

```
extern  tWIdent MakeWIdent      (const wchar_t * string, int length);
                        /* The string (of length) is mapped to a unique */
                        /* identifier (an integer) which is returned.   */
                        /* The string has not to be null-terminated and */
                        /* may contain null characters.                 */

extern  void    GetWString      (tWIdent ident, wchar_t * string);
                        /* Returns the string whose identifier is 'ident'. */

extern  tWStringRef GetWStringRef (tWIdent ident);
                        /* Returns a reference to the string identified */
                        /* by 'ident'.                                  */

extern  wchar_t * GetWCStr      (tWIdent ident);
                        /* Returns the address of the null-terminated   */
                        /* string identified by 'ident'.                */

extern  int     GetWLength      (tWIdent ident);
                        /* Returns the length of the string whose       */
                        /* identifier is 'ident'.                       */

extern  void    WriteWIdent     (FILE * file, tWIdent ident);
                        /* The string encoded by the identifier 'ident' */
                        /* is printed on the file.                      */

extern  void    WriteIdents     (void);
                        /* The contents of the identifier table is      */
                        /* printed on the standard output (debugging).  */

extern  void    BeginIdents     (void);
                        /* The identifier table is initialized.         */

extern  void    CloseIdents     (void);
                        /* The identifier table is finalized.           */
```

6. Sets: sets for scalar values

The following module provides operations on sets of scalar values. The elements of the sets can be of the types int, unsigned, char, or long. The size of the sets, that is the range the elements must lie in, is not restricted. The elements can range from 0 to 'MaxSize', where space for arbitrary large sets.

The sets are implemented as bit vectors (long []) plus some additional information to improve performance. So don't worry about speed too much because procedures like Select, Extract, or Card are quite efficient. They don't execute a loop over all potentially existing elements always. This happens only in the worst case.

```
# define BitsPerBitset          (sizeof (long) * 8)
# define MaskBitsPerBitset      (BitsPerBitset - 1)

# define IsElement(Elmt, Set)   ((rbool) (((Set)->BitsetPtr \
                [(Elmt) / BitsPerBitset] >> ((Elmt) & MaskBitsPerBitset)) & 1))
# define Size(Set)              ((Set)->MaxElmt)
# define Select(Set)            Minimum (Set)
# define IsNotEqual(Set1, Set2) (! IsEqual (Set1, Set2))
# define IsStrictSubset(Set1, Set2) (IsSubset (Set1, Set2) && \
                                IsNotEqual (Set1, Set2))

typedef struct  {
      int       MaxElmt         ;
      int       LastBitset      ;
      long *    BitsetPtr       ;
      int       Card            ;
      int       FirstElmt       ;
      int       LastElmt        ;
   } tSet;

extern  void    MakeSet         (tSet * Set, int MaxSize);
extern  void    ResizeSet       (tSet * Set, int MaxSize);
extern  void    ReleaseSet      (tSet * Set);
extern  void    Union           (tSet * Set1, tSet * Set2);
extern  void    Difference      (tSet * Set1, tSet * Set2);
extern  void    Intersection    (tSet * Set1, tSet * Set2);
extern  void    SymDiff         (tSet * Set1, tSet * Set2);
extern  void    Complement      (tSet * Set);
extern  void    Include         (tSet * Set, int Elmt);
extern  void    Exclude         (tSet * Set, int Elmt);
extern  int     Card            (tSet * Set);
/* extern int   Size            (tSet * Set); */
extern  int     Minimum         (tSet * Set);
extern  int     Maximum         (tSet * Set);
/* extern int   Select          (tSet * Set); */
extern  int     Extract         (tSet * Set);
extern  rbool   IsSubset        (tSet * Set1, tSet * Set2);
/* extern rbool IsStrictSubset  (tSet * Set1, tSet * Set2); */
extern  rbool   IsEqual         (tSet * Set1, tSet * Set2);
/* extern rbool IsNotEqual      (tSet * Set1, tSet * Set2); */
/* extern rbool IsElement       (int Elmt, tSet * Set); */
extern  rbool   IsEmpty         (tSet * Set);
extern  rbool   Forall          (tSet * Set, rbool (* Proc) (int));
extern  rbool   Exists          (tSet * Set, rbool (* Proc) (int));
extern  rbool   Exists1         (tSet * Set, rbool (* Proc) (int));
extern  void    Assign          (tSet * Set1, tSet * Set2);
extern  void    AssignElmt      (tSet * Set, int Elmt);
extern  void    AssignEmpty     (tSet * Set);
extern  void    ForallDo        (tSet * Set, void (* Proc) (int));
extern  void    ReadSet         (FILE * File, tSet * Set);
extern  void    WriteSet        (FILE * File, tSet * Set);
extern  void    BeginSets       (void);
```

Two parameters of type ’tSet’ passed to one of the above procedures must have the same size, that is they must have been created by passing the same value ’MaxSize’ to the procedure ’MakeSet’. A parameter representing an element (of type unsigned or equivalent) passed to one of the above procedures must have a value between 0 and ’MaxSize’ of the involved set which is the other

parameter passed. If the two conditions above, which can be verified at programming time, don't hold then strange things will happen, because there are no checks at run time, of course.

The following table explains the semantics of the set operations:

Procedure	Semantics
MakeSet	allocates space for a set to hold elements ranging from 0 to 'MaxSize'.
ResizeSet	resizes the allocated space for a set to hold elements ranging from 0 to 'MaxSize'.
ReleaseSet	releases the space taken by a set.
Union	Set1 = Set1 ∪ Set2
Difference	Set1 = Set1 - Set2
Intersection	Set1 = Set1 ∩ Set2
SymDiff	Set1 = Set1 ⊕ Set2 /* corresponds to exclusive or */
Complement	Set = { 0 .. MaxSize } - Set
Include	Set = Set ∪ { Elmt }
Exclude	Set = Set - { Elmt }
Card	returns number of elements in Set (| Set |)
Size	returns 'MaxSize' given at creation time
Minimum	returns smallest element x from Set
Maximum	returns largest element x from Set
Select	returns arbitrary element x from Set
Extract	returns arbitrary element x from Set and removes it from Set
IsSubset	Set1 ⊆ Set2
IsStrictSubset	Set1 ⊂ Set2
IsEqual	Set1 == Set2
IsNotEqual	Set1 ≠ Set2
IsElement	Elmt ∈ Set
IsEmpty	Set = ∅
Forall	∀ e ∈ Set : Proc (e) /* predicate Proc must hold for all elements */
Exists	∃ e ∈ Set : Proc (e) /* predicate Proc must hold for at least 1 element */
Exists1	| { e ∈ Set : Proc (e) } | == 1
Assign	Set1 = Set2
AssignElmt	Set1 = { Elmt }
AssignEmpty	Set1 = ∅
ForallDo	for (e = 0; e <= MaxSize; e ++) if (e ∈ Set) Proc (e);
ReadSet	read external representation of a set from file 'tFile'.
WriteSet	write external representation of a set to file 'tFile'. Example output: { 0 5 6 123}

7. Relation: binary relations between scalar values

This module provides operations on binary relations between scalar values. The elements of the relations can be pairs of the types int, long, short, char, or of an enumeration type. The size of the relations is not restricted. The size is a parameter to the procedure MakeRelation which dynamically allocates space for arbitrary large relations.

The relations are implemented as bit matrices or to be more precise as arrays of sets. Relations are viewed as sets of pairs. Therefore the set operations as defined above are also applicable for relations. The additional operations have the meaning known from theory.

```
typedef struct { tSet * ArrayPtr; int Size1, Size2; } tRelation;

extern  void    rMakeRelation   (tRelation * Rel, int Size1, int Size2);
extern  void    rReleaseRelation(tRelation * Rel);
extern  void    rInclude        (tRelation * Rel, int e1, int e2);
extern  void    rExclude        (tRelation * Rel, int e1, int e2);
extern  rbool   rIsElement      (int e1, int e2, tRelation Rel);
extern  rbool   rIsRelated      (int e1, int e2, tRelation Rel);
extern  rbool   rIsReflexive1   (int e1, tRelation Rel);
extern  rbool   rIsSymmetric1   (int e1, int e2, tRelation Rel);
extern  rbool   rIsTransitive1  (int e1, int e2, int e3, tRelation Rel);
extern  rbool   rIsReflexive    (tRelation Rel);
extern  rbool   rIsSymmetric    (tRelation Rel);
extern  rbool   rIsTransitive   (tRelation Rel);
extern  rbool   rIsEquivalence  (tRelation Rel);
extern  rbool   rHasReflexive   (tRelation Rel);
extern  rbool   rIsCyclic       (tRelation Rel);
extern  void    rGetCyclics     (tRelation Rel, tSet * Set);
extern  void    rClosure        (tRelation * Rel);
extern  void    rAssignEmpty    (tRelation * Rel);
extern  void    rAssignElmt     (tRelation * Rel, int e1, int e2);
extern  void    rAssign         (tRelation * Rel1, tRelation Rel2);
extern  void    rUnion          (tRelation * Rel1, tRelation Rel2);
extern  void    rDifference     (tRelation * Rel1, tRelation Rel2);
extern  void    rIntersection   (tRelation * Rel1, tRelation Rel2);
extern  void    rSymDiff        (tRelation * Rel1, tRelation Rel2);
extern  void    rComplement     (tRelation * Rel);
extern  rbool   rIsSubset       (tRelation Rel1, tRelation Rel2);
extern  rbool   rIsStrictSubset (tRelation Rel1, tRelation Rel2);
extern  rbool   rIsEqual        (tRelation * Rel1, tRelation * Rel2);
extern  rbool   rIsNotEqual     (tRelation Rel1, tRelation Rel2);
extern  rbool   rIsEmpty        (tRelation Rel);
extern  int     rCard           (tRelation * Rel);
extern  void    rSelect         (tRelation * Rel, int * e1, int * e2);
extern  void    rExtract        (tRelation * Rel, int * e1, int * e2);
extern  rbool   rForall         (tRelation Rel, rbool (* Proc) (int, int));
extern  rbool   rExists         (tRelation Rel, rbool (* Proc) (int, int));
extern  rbool   rExists1        (tRelation Rel, rbool (* Proc) (int, int));
extern  void    rForallDo       (tRelation Rel, void (* Proc) (int, int));
extern  void    rReadRelation   (FILE * f, tRelation * Rel);
extern  void    rWriteRelation  (FILE * f, tRelation Rel);
extern  void    rProject1       (tRelation Rel, int e1, tSet * Set);
extern  void    rProject2       (tRelation Rel, int e1, tSet * Set);
```

8. Position: handling of source positions

A simple representation of the position of tokens in a source file consisting of fields for line, column, and filename. This module can be copied and tailored to the user's needs, if necessary. Modifications may be necessary for example if the type 'unsigned short' is too small to count the columns.

```
# include "Idents.h"

typedef struct {
   unsigned long  Line    ;
   unsigned short Column  ;
   tIdent         FileName; } tPosition;

extern tPosition NoPosition;
                        /* A default position (0, 0, "").              */

extern int      Compare         (tPosition Position1, tPosition Position2);
                        /* Returns -1 if Position1 < Position2.        */
                        /* Returns  0 if Position1 = Position2.        */
                        /* Returns  1 if Position1 > Position2.        */

extern void     WritePosition   (FILE * File, tPosition Position);
                        /* The 'Position' is printed on the 'File'.    */

extern void     ReadPosition    (FILE * File, tPosition * Position);
                        /* The 'Position' is read from the 'File'.     */

extern char *   FormatPosition  (char * String, tPosition Position);
                /* The 'Position' is formatted and stored as a string. */
                /* If 'String' is NULL a static string is returned.    */
                /* Otherwise 'String' is used and returned.            */
```

9. Errors: error handler for parsers and compilers

This module is needed by parsers generated with the parser generators *lark* or *ell*. It can also be used to report error messages found during scanning or semantic analysis. Note: If the module *Position* is copied (and maybe modified) then the module *Errors* has to be copied, too, because the modules depend on each other.

```
                                        /* error codes          */
# define xxNoText                   0
# define xxTooManyErrors            1
# define xxTclTkError               2
                                        /* reuse library        */
# define xxAllocOutOfMemory         10
# define xxStringUndefined          11
# define xxIdentOutOfBounds         12
                                        /* scanner              */
# define xxScannerInternalError     20
# define xxScannerOutOfMemory       21
# define xxFileStackUnderflow       22
# define xxCannotOpenInputFile      23
# define xxStartStackUnderflow      24
                                        /* parser               */
# define xxSyntaxError              30
# define xxExpectedTokens           31
# define xxRestartPoint             32
# define xxTokenInserted            33
# define xxTokenFound               34
# define xxFoundExpected            35
# define xxInvalidCallOfReParse     36
                                        /* ast                  */
# define xxTreeOutOfMemory          40
# define xxTreeIOError              41
# define xxQueryTreeEof             42
                                        /* ast.CheckTree        */
# define xxCheckTreeError           50
# define xxCheckTreeAddrOfParent    51
# define xxCheckTreeTypeOfParent    52
# define xxCheckTreeNameOfChild     53
# define xxCheckTreeValueOfChild    54
# define xxCheckTreeAddrOfChild     55
# define xxCheckTreeTypeOfChild     56
# define xxCheckTreeExpectedType    57
# define xxCheckTreeParentNode      58
# define xxCheckTreeChildNode       59
                                        /* ag                   */
# define xxCyclicDependencies       60
                                        /* puma                 */
# define xxRoutineFailed            70
                                        /* error classes        */
# define xxFatal                    1
# define xxRestriction              2
# define xxError                    3
# define xxWarning                  4
# define xxRepair                   5
# define xxNote                     6
# define xxInformation              7
```

```
                                                /* info classes         */
# define xxNone                         0
# define xxInteger                      1
# define xxShort                        2
# define xxLong                         3
# define xxUSLong                       4
# define xxReal                         5
# define xxBoolean                      6
# define xxCharacter                    7
# define xxWCharacter                   8
# define xxString                       9
# define xxWString                      10
# define xxIdent                        11
# define xxWIdent                       12
# define xxSet                          13
# define xxPosition                     14

extern void (* Errors_Exit) (void);
                        /* Refers to a procedure that specifies         */
                        /* what to do if 'ErrorClass' = Fatal.          */
                        /* Default: terminate program execution.        */

extern void BeginErrors   (void);
                        /* Initialize the 'Errors' module.              */

extern void StoreMessages (rbool Store);
                        /* Messages are stored if 'Store' = TRUE        */
                        /* for printing with the routine 'WriteMessages'*/
                        /* otherwise they are printed immediately.      */
                        /* If 'Store'=TRUE the message store is cleared.*/

extern void ErrorMessage  (int ErrorCode, int ErrorClass, tPosition Position);
                        /* Report a message represented by an integer   */
                        /* 'ErrorCode' and classified by 'ErrorClass'.  */

extern void ErrorMessageI (int ErrorCode, int ErrorClass, tPosition Position,
                           int InfoClass, char * Info);
                        /* Like the previous routine with additional    */
                        /* information of type 'InfoClass' at the       */
                        /* address 'Info'.                              */

extern void Message       (char * ErrorText, int ErrorClass, tPosition Position);
                        /* Report a message represented by a string     */
                        /* 'ErrorText' and classified by 'ErrorClass'.  */

extern void MessageI      (char * ErrorText, int ErrorClass, tPosition Position,
                           int InfoClass, char * Info);
                        /* Like the previous routine with additional    */
                        /* information of type 'InfoClass' at the       */
                        /* address 'Info'.                              */

extern void WriteMessages (FILE * File);
                        /* The stored messages are sorted by their      */
                        /* source position and printed on 'File'.       */

extern char * CodeToText  (long ErrorCode);
                        /* Map 'ErrorCode' to a message text.           */
```

```
extern int  GetCount      (int ErrorClass);
                         /* Return the number of messages of class       */
                         /* 'ErrorClass'.                                */

extern void MessageChar   (char * ErrorText, int ErrorClass, tPosition Position,
                           char * Info);
        /* Should be called similar to: MessageChar
           ("illegal character", xxError, some.Position, TokenPtr);
           Is eqivalent to the call: MessageI
           ("illegal character", xxError, some.Position, xxCharacter, TokenPtr);
           but tries to print the character at TokenPtr [0] either as printable
           character or as an escape sequence.
         */
```

This module can be regarded as a prototype for reporting compiler error messages. It can be copied an modified or even replaced in order to meet the requirements of the user's application. The body of the module contains three C preprocessor variables that control the style of the error messages:

BRIEF	summarize syntax errors in one error message instead of several messages
FIRST	report only the first error message on a line instead of all messages
TRUNCATE	truncate additional information for messages (such as the set of expected symbols) to around 25 characters

Example: The following Pascal program contains two syntax errors:

```
program test (output);
begin
   if (a = b] write (a;
end.
```

If all three preprocessor variables are undefined then the following messages are reported:

```
3, 13: Error       syntax error
3, 13: Information token found    : ]
3, 13: Information expected tokens: ) = + - <> <= >= < > IN OR * / DIV MOD AND
3, 15: Information restart point
3, 15: Repair      token inserted : )
3, 15: Repair      token inserted : THEN
3, 23: Error       syntax error
3, 23: Information token found    : ;
3, 23: Information expected tokens: , ) = + - : <> <= >= < > IN OR * / DIV MOD AND
3, 23: Repair      token inserted : )
```

If BRIEF is defined then this is compressed into two lines:

```
3, 13: Error       found/expected : ])/) = + - <> <= >= < > IN OR * / DIV MOD AND
3, 23: Error       found/expected : ;/, ) = + - : <> <= >= < > IN OR * / DIV MOD AND
```

If BRIEF and FIRST are defined then this results in just one line:

```
3, 13: Error       found/expected : ])/) = + - <> <= >= < > IN OR * / DIV MOD AND
```

If BRIEF, FIRST, and TRUNCATE are defined then this one line becomes even shorter:

```
3, 13: Error       found/expected : ]/) = + - <> <= >= < > IN OR * / ...
```

In all of the abbreviated styles the information about restart points or inserted tokens is suppressed and the messages reporting the found token and the set of expected tokens are combined into one message.

10. Source: provides input for scanners

This module is needed by scanners generated with the scanner generator *rex*.

```
                                /* encodings of the input stream       */
# define CODE_NONE      0
# define CODE_BYTE      1       /* 1 byte         */
# define CODE_WCHAR_T   2       /* 2 or 4 bytes   */
# define CODE_UCS2      3       /* 2 bytes        */
# define CODE_UCS4      4       /* 4 bytes        */
# define CODE_UTF8      5       /* seq of 1 byte  */
# define CODE_UTF16     6       /* seq of 2 bytes */

   /* The above comments give the size of an input stream item in bytes.
      All input stream items (or sequences of input stream items in the cases
      of UTF8 and UTF16) represent Unicode characters.
      The encodings BYTE, UCS2, and UTF16, and possibly WCHAR_T can represent
      subsets of the full Unicode character set, only.
      A Unicode character will be stored in a variable of type wchar_t.
      Note, the size of wchar_t is 2 or 4 bytes, depending on the compiler.
      Therefore, if the size of wchar_t is 2 then characters encoded by UCS4,
      UTF8, and UTF16 will be truncated to two bytes.
   */
                                /* endian property of the input stream  */

# define ENDIAN_NONE    0       /* no endian property specified        */
# define ENDIAN_LITTLE  1       /* little-endian                       */
# define ENDIAN_BIG     2       /* big-endian                          */

extern void SetEncoding (int Encoding, int Endian);

   /*
      Specify the encoding and the endian property of the input stream.
      The arguments have to be values as defined above.
      This function has to be called after the function BeginSource... .
      If neither little-endian nor big-endian is specified then the endian
      property of the current system is assumed to hold for the input.
      The function GetWLine will convert the input stream to a stream
      of type wchar_t.
   */

extern int  BeginSourceFile (char * FileName);

   /*
      BeginSourceFile is called from the scanner function BeginFile
      indicating that input should be read from a file. The file specified
      by the parameter 'FileName' is opened and used as input file.
      If not called input is read from standard input. The function
      should return an integer file descriptor as provided by the system
      call 'open' or any other handle understood by the function GetLine.
   */

extern int  BeginSourceFileW (wchar_t * FileName);

   /*
      Same as BeginSourceFile for type wchar_t instead of type char.
      The Source module has to be extended by the user in order to
      implement this feature.
   */
```

```
extern void BeginSourceMemory (void * InputPtr);

   /*
      BeginSourceMemory is called from the scanner function BeginMemory
      indicating that input should be read from the null terminated string
      of input items at location 'InputPtr'.
      The input string may not contain null characters. The contents of the
      string may not be changed until it has been processed completely.
   */

extern void BeginSourceMemoryN (void * InputPtr, int Length);

   /*
      BeginSourceMemoryN is called from the scanner function BeginMemoryN
      indicating that the input should be 'Length' input items at location
      'InputPtr'. The input may contain null characters. The contents of the
      input may not be changed until it has been processed completely.
   */

extern void BeginSourceGeneric (void * InputPtr);

   /*
      BeginSourceGeneric is called from the scanner function BeginGeneric
      indicating that the input is user-defined at location 'InputPtr'.
      The Source module has to be extended by the user in order to
      implement this feature.
   */

extern int  GetLine (int File, char * Buffer, int Size);

   /*
      GetLine is called from the scanner in order to fill a buffer at
      address 'Buffer' with a block of maximal 'Size' characters. Input
      should be read from a file specified by the integer file descriptor
      'File' if the current input stream comes from a file. Otherwise input
      comes from memory and the parameter 'File' can be ignored.
      Lines are terminated by newline characters (ASCII = 0xa).
      The function returns the number of characters transferred.
      Reasonable block sizes are between 128 and 8192 or the length of a line.
      Smaller block sizes - especially block size 1 - will drastically
      slow down the scanner. The end of file or end of input is indicated
      by a return value <= 0.
   */

extern int  GetWLine (int File, wchar_t * Buffer, int Size);

   /*
      Same as GetLine for type wchar_t instead of type char.
   */

extern void CloseSource (int File);

   /*
      CloseSource is called from the scanner function CloseFile
      at end of file or at end of input, respectively.
      It can be used to close files.
      The functions BeginSource... and CloseSource can be called
      in a nested way, for example in order to handle include files.
      The encoding and the endian property of the input stream are stacked.
      Therefore after a call of CloseSource the properties of the
      previous input stream are restored.
   */
```

11. General: miscellaneous functions

```
# define Min(a,b) (((a) <= (b)) ? (a) : (b))
                          /* Returns the minimum of 'a' and 'b'.          */
# define Max(a,b) (((a) >= (b)) ? (a) : (b))
                          /* Returns the maximum of 'a' and 'b'.          */

extern unsigned long    Log2 (unsigned long x);
                          /* Returns the logarithm to the base 2 of 'x'.  */
extern unsigned long    Exp2 (unsigned long x);
                          /* Returns 2 to the power of 'x'.               */
```

12. rSystem: machine dependent code

This module provides a few machine dependent operations.

```
/* interface for machine dependencies */

# define tFile int

/* binary IO */

extern tFile    OpenInput       (char * FileName);
                        /* Opens the file whose name is given by the    */
                        /* string parameter 'FileName' for input.       */
                        /* Returns an integer file descriptor.          */

extern tFile    OpenOutput      (char * FileName);
                        /* Opens the file whose name is given by the    */
                        /* string parameter 'FileName' for output.      */
                        /* Returns an integer file descriptor.          */

extern int      rRead           (tFile File, char * Buffer, int Size);
                        /* Reads 'Size' bytes from file 'tFile' and     */
                        /* stores them in a buffer starting at address  */
                        /* 'Buffer'.                                    */
                        /* Returns the number of bytes actually read.   */

extern int      rWrite          (tFile File, char * Buffer, int Size);
                        /* Writes 'Size' bytes from a buffer starting   */
                        /* at address 'Buffer' to file 'tFile'.         */
                        /* Returns the number of bytes actually written.*/

extern void     rClose          (tFile File);
                        /* Closes file 'tFile'.                         */

extern void     rDeleteFile     (char * FileName);
                        /* Deletes the file named 'FileName'.           */

extern rbool IsCharacterSpecial (tFile File);
                        /* Returns TRUE when file 'tFile' is connected  */
                        /* to a character device like a terminal.       */

extern long     rFileSize       (char * FileName);
                        /* Returns the total number of bytes in the     */
                        /* file named 'FileName'.                       */

extern char  DirectorySeparator (void);
                        /* Returns '\' under Windows, otherwise '/'.    */

/* calls other than IO */

extern char *   rAlloc          (long ByteCount);
                        /* Returns a pointer to dynamically allocated   */
                        /* memory space of size 'ByteCount' bytes.      */
                        /* Returns NIL if space is exhausted.           */
```

```
extern void     rFree           (char * Ptr);
                        /* The dynamically allocated memory space       */
                        /* pointed to by 'Ptr' is released.             */

extern long     rTime           (void);
                        /* Returns consumed cpu-time in milliseconds.   */

extern int      GetArgCount     (void);
                        /* Returns number of arguments.                 */

extern void     GetArgument     (int ArgNum, char * Argument);
                        /* Stores a string-valued argument whose index  */
                        /* is 'ArgNum' in the memory area 'Argument'.   */

extern char *   GetEnvVar       (char * Name);
                        /* Returns a pointer to the environment         */
                        /* variable named 'Name'.                       */

extern void     PutArgs         (int Argc, char * * Argv);
                        /* Dummy procedure that passes the values       */
                        /* 'argc' and 'argv' from Modula-2 to C.        */

extern int      rErrNo          (void);
                        /* Returns the current system error code.       */

extern int      rSystem         (char * String);
                        /* Executes an operating system command given   */
                        /* as the string 'String'. Returns an exit or   */
                        /* return code.                                 */

extern void     rExit           (int Status);
                        /* Terminates program execution and passes the  */
                        /* value 'Status' to the operating system.      */

extern void  (* Reuse_Exit)     (void);
                        /* Configurable exception handling of the Reuse */
                        /* library. Is called if out of memory.         */

extern void     BEGIN_System    (void);
                        /* Dummy procedure with empty body.             */
```

13. rTime: access to cpu-time

```
extern int      StepTime        (void);
                        /* Returns the sum of user time and system time */
                        /* since the last call to 'StepTime' in milli-  */
                        /* seconds.                                     */

extern void     WriteStepTime   (char * string);
                        /* Writes a line consisting of the string       */
                        /* 'string' and the value obtained from a call  */
                        /* to 'StepTime' on standard output.            */
```

14. rString: portable string handling

This module tries to be a portable replacement for the non-portable header files <string.h> and <strings.h>. The functions strcasecmp, strncasecmp, wcscpy and wcslen are provided if not supplied by the current C library.

```
extern int       strcasecmp     (const char * s1, const char * s2);
extern int       strncasecmp    (const char * s1, const char * s2, size_t n);
extern wchar_t * wcscpy         (wchar_t * s1, const wchar_t * s2)
extern int *     wcslen         (register const wchar_t * s)
```

15. rGetopt: portable version of Unix getopt

This is Cocktail's version of the UNIX getopt routine. Due to many incompatibilities under various versions of UNIX and MS Windows this version is provided by the Cocktail Reuse library. rGetopt behaves more or less like the usual getopt.

The main differences to other getopt-implementations are:

- Error handling:
  - No error message is printed, instead the global variable rOptmsg gets an error string assigned.
  - The variable opterr (rOpterr) is ignored.
- Handling of the optstring argument of rGetopt(): optstring is preprocessed on the first call and stored internally. This preprocessing is done by rGetopt() when rOptind has the value 0, or if rOptreset() is called. (Other getopt-implementations rescan optstring each time getopt() is called, allowing to change the allowed options each time getopt() is called.)
- No environment variables are considered (others use e.g. POSIX_CORRECT).

```
extern int rGetopt (int argc, char *const *argv, const char *optstring);

/* The rGetopt() function parses the command line arguments. Its arguments
 * argc and argv are the argument count and array as passed to the main()
 * function on program invocation. An element of argv that starts with '-'
 * (and is not exactly "-" or "--") is an option element. The characters of
 * this element (aside from the initial '-') are option characters. If
 * rGetopt() is called repeatedly, it returns successively each of the option
 * characters from each of the option elements.
 *
 * If rGetopt() finds another option character, it returns that character,
 * updating the external variable rOptind so that the next call to rGetopt()
 * can resume the scan with the following option character or argv-element.
 *
 * If there are no more option characters, rGetopt() returns EOF. Then rOptind
 * is the index in argv of the first argv-element that is not an option.
 *
 * optstring is a string containing the legitimate option characters. If such
 * a character is followed by a colon, the option requires an argument, so
 * rGetopt places a pointer to the following text in the same argv-element, or
 * the text of the following argv-element, in rOptarg. Two colons mean an
 * option takes an optional arg; if there is text in the current argv-element,
 * it is returned in rOptarg, otherwise rOptarg is set to zero.
 *
```

```
 * If the first character of optstring is ‘+’, then option processing stops as
 * soon as a non-option argument is encountered. If the first character of
 * optstring is ‘-’, then each non-option argv-element is handled as if it were
 * the argument of an option with character code 1 (i.e. ’\1’).
 * (This is used by programs that were written to expect options and other
 * argv-elements in any order and that care about the ordering of the two.)
 * The special argument ‘--’ forces an end of option-scanning regardless of
 * the scanning mode.
 * Default (if  the first character of optstring is not ‘+’ or ‘-’) is the same
 * as if ‘+’ is given.
 *
 * If rGetopt() does not recognize an option character or if a required option
 * argument is missing, rGetopt stores the (option) character in rOptopt,
 * and returns ‘?’.
 *
 * In case of an error rGetopt returns ’?’ and stores an error message
 * number string in rOptmsg. No error message is printed.
 */

/* Error messages stored in rOptmsg, this message may be used like
 * printf ("%s %c\n", rOptmsg, rOptopt), or
 * (strcmp (rOptmsg,rGETOPT_NO_ARG) == 0)
 * If no error occurred rOptmsg == 0.
 */

extern char *rOptmsg;
# define rGETOPT_ILLEGAL  "illegal option character"
# define rGETOPT_NO_ARG   "required argument missing for option"
# define rGETOPT_TOO_MANY "too many option characters passed to rGetopt()"
  /* treat rGETOPT_TOO_MANY as a fatal error, options are not processed */

extern char *rOptarg;
/* For communication from ‘rGetopt’ to the caller.
 * When ‘rGetopt’ finds an option that takes an argument,
 * the argument value is returned here. Otherwise NULL. */

extern int rOptind;
/* Index in ARGV of the next element to be scanned.
 * This is used for communication to and from the caller
 * and for communication between successive calls to ‘rGetopt’.
 *
 * On entry to ‘rGetopt’, zero means this is the first call; initialize.
 *
 * When ‘rGetopt’ returns EOF, this is the index of the first of the
 * non-option elements that the caller should itself scan.
 *
 * Otherwise, ‘rOptind’ communicates from one call to the next
 * how much of ARGV has been scanned so far.
*/

extern int rOptopt;
/* Set to the option character which caused the error */

extern int rOpterr;
/* Just for compatibility with other getopt implementations. The value of this
 * variable is ignored by rGetopt.
 */
```

```
extern void rOptreset (void);
/* Recomputes the internal representation of optstring, next time rGetopt
 * is called. rOptreset() does not change the value of rOptind or other
 * internal values. Hence it may be called to switch optstring between two
 * calls of rGetopt and continue with processing the commandline.
 */
```

16. rFsearch: support for searching files

This module allows to search a file with a given name in a list of directories. The search may be done case sensitive or case insensitive. This module may be used under UNIX as well as under Microsoft Windows operating systems.

```
/*
 * rAddDir
 * -------
 *  The directories are specified using the ‘rAddDir’ function.
 *  Each directory may be given a ‘mark’. The directories with the same
 *  ‘mark’ may be removed or cached, while the others remain untouched.
 *  This is useful, e.g. if some of the directories are specified
 *  by command line options, while others are specified in the analyzed
 *  program. When parsing lots of sources in one program run, one wishes
 *  to delete the directories added by the source after processing this source
 *  while the command line directories should persist for all processed
 *  sources.
 *  The order in which the directories are considered when searching a file
 *  is determined by the ‘at_end’ parameter: If true, that directory
 *  is appended to the already existing list of directories. If ‘at_end’
 *  is false, the directory will be the new first element.
 *  If the search list is empty, the current directory "./" is used by default.
 *  If the search list is not empty and the current directory should be
 *  searched, it has to be added explicitly.
 *
 * rReadDirs, rReadAllDirs
 * -----------------------
 *  To speed up ‘rFindFile’, the contents of a directory may be read in
 *  advance using ‘rReadDirs’ and ‘rReadAllDirs’. ‘rFindFile’ will
 *  check only that cached directory contents. If the directories are not
 *  cached, for each file name a ‘stat’ or ‘access’ system call is used.
 *  Caching is ~4 (case sensitive) and ~3 (case insensitive) times faster as
 *  non cached access.
 *  "Mixed usage" (i.e. some directories cached others not) is possible.
 *
 * rAddExtension, rDeleteExtensions
 * --------------------------------
 *  Besides searching a file with a fixed name, the user may specify a set
 *  of extensions which are used to construct the file name.
 *  To add such an extension use ‘rAddExtension’.
 *  If the list of extensions is empty, the filename remains unchanged
 *  (obviously).
 *  If the list of extensions is not empty and the "empty" extension should
 *  be considered too, one has to add it by calling ‘rAddExtension ("")’.
 *  The order in which the extensions are tried is the reverse order of the
 *  calls of ‘rAddExtension’.
 *
 * rFindFile
```

```
* ---------
*  To search a file in the list of directories, use ‘rFindFile’.
*  The entire path is returned, if the file is found and is readable.
*  The lookup is: visit all directories D and all extensions E and
*  check for the given name N whether a file "D/NE" exists.
*  More precisely:
*
*  FOR ALL directories D given by rAddDir in the specified order DO
*     IF rFileIsReadable (D/N) THEN RETURN D/N FI
*     FOR ALL extensions E given by rAddExtension DO
*       IF rFileIsReadable (D/NE) THEN RETURN D/NE FI
*     END;
*  END;
*  RETURN NULL;
*
*  If the ‘name’ contains a "/" and caching is enabled, the directory part
*  of ‘name’ is appended to a search directory and if it exists, it is cached
*  too.
*  rFindFile removes leading a ’./’ in its output.
*  Used under UNIX: leading ’//’ is replaced by one ’/’.
*  Used under Microsoft: a leading ‘\\’ is not replaced (since it denotes
*  remote file access.
*  Note: the memory holding the resulting string is allocated internally, so
*        that a second call of rFindFile will invalidate the result of the
*        first call.
*
* rDeleteDirs, rDeleteAllDirs, rDeleteExtensions
* ----------------------------------------------
*  Delete these things and release the corresponding memory.
*
* Begin_rFsearch, Close_rFsearch
* ------------------------------
*  Initialize and finalize the module. Must be done once per program run.
*  ‘Close_rFsearch’ releases most of the allocated memory, allocated by
*  this module.
*
*  Before calling any of the functions of this module, it must be initialzed.
*  After finialization, none of the modules functions or strings returned
*  by them may be used.
*
*  Begin_rFsearch must be called _always_ _after_  BeginrMemory and
*  Close_rFsearch  must be called _always_ _before_ CloserMemory
*
* For convenience there are the following routines:
* rDirOrFileExists
* ----------------
*  Returns true, iff a file or directory (or anything else) with this name
*  exists in the file system.
*
* rFileIsReadable, rDirIsReadable
* -------------------------------
*  Return true, iff the file / directory can be read.
*  (The implementation uses ‘OpenInput’ of the ‘rSystem’ module, and
*   a check whether it’s a file or a directory)
*
* rDirName, rBaseName, rDirBaseName
* ---------------------------------
```

```
*  Split a filename into its directory and basename parts.
*
* rOpenFileSystemDir, rNextFileSystemEntry, rCloseFileSystemDir
* -------------------------------------------------------------
*  Reading the content of a directory in the file system.
*  These routines access the file system directly, without using the
*  caching mechanism described above. Use them as follows:
*   tFileSystemDir dir = rOpenFileSystemDir (name);
*   if (dir != NULL) {
*     const char *entry;
*     while ((entry = rNextFileSystemEntry(dir)) != NULL) {
*          do_something_with_entry (entry);
*     }
*   }
*
* rDirSeparator
* -------------
*  A read-only variable, containing the directory separator. Under
*  UNIX: that's a '/', under Microsoft it is a '\'.
*
* rReplaceDirSeparator
* --------------------
*  Replaces '/' or '\' by the content of rDirSeparator
*
* General remarks:
* ----------------
*  Memory allocation
*  -----------------
*  We are using 'rMemory' to allocate the required memory. Hence
*  'BeginrMemory' should be called before 'Begin_rFsearch', and
*  'CloserMemory' should be called only after 'Close_rFsearch'.
*
*  Directory separators '/' and '\'
*  --------------------------------
*  Depending on the operating system under which the module is used,
*  the variable rDirSeparator is set accordingly.
*  If one of the routines above generate a directory separator,
*  the content of rDirSeparator is used.
*  All '/' or a '\' characters contained in a file or directory name are
*  replaced by the value of rDirSeparator by all routines of this module
*  (i.e. rReplaceDirSeparator is called automatically).
*  Hence one can e.g. specify filenames under Microsoft as "/foo/bar"
*  or under UNIX as "\foo\bar" without any problems.
*
*  Case sensitivity
*  ----------------
*  The adding of directories ('rAddDir') as well as searching files
*  ('rFindFile') may handle the case of the given names in different ways:
*  the module may be used in "case sensitive" or "case insensitive" mode.
*  (see 'Begin_rFsearch').
*  In case-sensitive mode:
*      a file is found if and only if the directory, file names and
*      extensions are given as stored in the file system.
*  In case insensitive mode:
*      the behavior depends on whether caching is enabled or not
*      ('rReadDirs', 'rReadAllDirs'):
*      If caching is enabled:
```

```
 *         ‘rReadDirs’ / ’rReadAllDirs’ modifies the directory names
 *         given by ‘rAddDir’ (in the internal data structures) so that the
 *         directories are found even if their names were given with the
 *         wrong case  (if it exists at all).
 *         ‘rFindFile’ tries to find the file in the (so modified) search
 *         directories without considering the case (inclusive the
 *         extensions).
 *      If caching is disabled:
 *         A simple heuristics is used for the name given by ‘rFindFile’:
 *           1. try filename (and extension) as given
 *           2. try filename (and extension) in uppercase letters
 *           3. try filename (and extension) in lowercase letters
 *           4. try first letter of the filename in uppercase, the rest in
 *              lowercase.
 *         There is no such heuristics for the name of the search directories.
 *         It must be given with the correct case of the letters.
 *      If the file is found the result of ‘rFindFile’ contains the name with
 *      the letters in the "correct" case, i.e. one can use the result
 *      in a call to open the file (e.g. ‘open’ or OpenInput).
 *      Note: When used unter a Microsoft operating system, everything
 *            is done automatically case-_in_sensitive.
 */

extern void Begin_rFsearch (rbool CaseSensitive);
/* Initialize the rFsearch module.
 * Use case sensitive (rtrue) or case insensitive (false) handling of
 * file and directory names.
 */

extern void Close_rFsearch (void);
/* Terminate the module (deallocates all allocated memory for this module) */

extern rbool rGetCaseSensitive (void);
/* Returns the value of ‘CaseSensitive’ given to ‘Begin_rFsearch’ */

extern void Print_rFsearch (FILE *file);
/* Print the list of searched directories and extensions to ‘file’.
 * The entries are listed in the search order.
 */

extern const char rDirSeparator;
/* Read-only variable holding the directory separator character.
 * - UNIX:      ’/’
 * - Microsoft: ’\’
 */

extern void rReplaceDirSeparator (char *name);
/*  Replaces in ‘name’ all ’/’ or ’\’ by the content of rDirSeparator */

extern rbool rDirOrFileExists (const char *name);
/* Returns true, iff ‘name’ exists in the file system. */

extern rbool rDirIsReadable (const char *name);
/* Returns true, iff directory ‘name’ can be opened for reading.
 * In case of errors ‘errno’ is set appropriately.
 */
```

```
extern rbool rFileIsReadable (const char *name);
/* Returns true, iff file ‘name’ can be opened for reading.
 * In case of errors ‘errno’ is set appropriately.
 */

extern char *rDirName (char *name, char *buffer);
/* Returns the directory part of the ‘name’.
 * The resulting string is not terminated by "/" (one exception: if
 * the root directory / is the result, see below)
 * If ‘name’ contains no directory part, "." is returned.
 * Trailing "/" are ignored.
 * e.g.   "/"           ==> "/"
 *        "/foo/"       ==> "/"
 *        "/foo//"      ==> "/"
 *        "/foo"        ==> "/"
 *        ""            ==> "."
 *        "foo"         ==> "."
 *        "foo/"        ==> "."
 *        "foo/bar"     ==> "foo"
 *        "foo/bar/"    ==> "foo"
 *        "foo/bar//"   ==> "foo"
 * In a Microsoft environment we have also:
 *        "C:"          ==> "C:"
 *        "C:foo"       ==> "C:"
 *        "C:foo\"      ==> "C:"
 *        "C:foo\bar"   ==> "C:foo"
 *        "C:\"         ==> "C:"
 *        "C:\foo"      ==> "C:"
 *        "C:\foo\"     ==> "C:"
 *        "C:\foo\bar"  ==> "C:\foo"
 *        "C:\foo\bar\" ==> "C:\foo"
 *
 * If ‘buffer’ == NULL the result is copied to a static buffer
 *     and that address is returned.
 * If ‘buffer’ == ‘name’, ‘name’ is modified and ‘name’ is returned.
 * If ‘buffer’ != NULL and ‘buffer’ != ‘name’ the result is copied to buffer
 *     and ‘buffer’ is returned."
 */

extern char *rBaseName (char *name, char *buffer);
/* Returns the basename part of ‘name’.
 * Trailing "/" are ignored.
 * e.g.  "/"           ==> ""
 *       "/foo"        ==> "foo"
 *       "/foo/"       ==> "foo"
 *       "/foo//"      ==> "foo"
 *       "foo/bar"     ==> "bar"
 *       "foo/bar.ext" ==> "bar.ext"
 * In a Microsoft environment we have also:
 *       "C:               ==> ""
 *       "C:foo"           ==> "foo"
 *       "C:foo\"          ==> "foo"
 *       "C:foo\bar"       ==> "bar"
 *       "C:\              ==> ""
 *       "C:\foo"          ==> "foo"
 *       "C:\foo\"         ==> "foo"
 *       "C:\foo\bar"      ==> "bar"
```

```
 *       "C:\foo\bar.ext"  ==> "bar.ext"
 *
 * If ‘buffer’ == NULL the result is copied to a static buffer
 *     and that address is returned.
 * If ‘buffer’ == ‘name’, ‘name’ is modified and ‘name’ is returned.
 * If ‘buffer’ != NULL and ‘buffer’ != ‘name’ the result is copied to buffer
 *     and ‘buffer’ is returned."
 */

extern void rDirBaseName (const char *name, char *dir_name, char *base_name);
/* Splits ‘name’ into its dirname and basename part as described above.
 * ‘dir_name’, and ‘base_name’ must be allocated already and be large
 * enough to hold that information.
 */

typedef struct sFileSystemDir *tFileSystemDir;
extern tFileSystemDir rOpenFileSystemDir   (const char *name);
/* Opens the directory ‘name’ so that its content can be read with
 * successive calls to ’rNextFileSystemEntry’.
 * Returns NULL, if the directory can’t be opened.
 */

extern const char    *rNextFileSystemEntry (tFileSystemDir dir);
/* Reads one entry in the directory ‘dir’. If there is no more entry,
 * returns NULL.
 */
extern void           rCloseFileSystemDir  (tFileSystemDir dir);
/* Closes the directory. */

extern void rAddDir (const char*  dir,
                     unsigned int mark,
                     rbool        at_end);
/* Add the directory ‘dir’ to the search list. A terminating ’/’ is
 * added automatically if needed.
 */

extern void rDeleteDirs (unsigned int mark);
/* Remove all directories with the given mark from the search list */

extern void rDeleteAllDirs (void);
/* Remove all directories from the search list */

extern void rReadDirs    (unsigned int mark);
/* Read the contents (name of files, subdirectories etc.) of all directories
 * with the given mark and store the directory entries in an internal cache.
 * Use the cached directory entries from now on.
 */

extern void rReadAllDirs (void);
/* Read the contents (name of files, subdirectories etc.) of all directories
 * and store the directory entries in an internal cache.
 * Use the cached directory entries from now on.
 */

extern void rAddExtension (const char *ext);
/* Add a file extension to the list of extensions */
```

```
extern void  rDeleteExtensions (void);
/* Delete all extensions from the extensions list */

extern char *rFindFile (const char* name);
/* If file is found and is readable return static pointer to the complete
 * name, otherwise return NULL.
 * If the file is found in the current directory (‘./’), no leading ‘./’
 * is added.
 * NOTE: - do not modify the resulting string
 *       - in most cases, a pointer to a static variable is returned.
 */

extern rbool rFsearchShowDebug;
/* If debugging of the rFsearch module is enabled (c.f. ‘rFsearch.c’)
 * then if ‘rFsearchShowDebug’ is true,
 *      then debugging output is enabled,
 *      else debugging output is suppressed (default)
 * else this variable has no meaning.
 */
```

17. rSrcMem: storage for source code

This module allows to store the source to be scanned / parsed in memory and to retrieve parts of the source based on a start and end position (i.e. line and column information). The retrieval expands tab-chars. The source is stored on a line-by-line base. Adding text into a store must not be done line-by-line but an entire buffer may be added. The splitting into lines is done by this module.

```
/*
 * Begin_rSrcMem, Close_rSrcMem
 * ----------------------------
 *  Initialization and termination of the module.
 *  NOTE: We are using ‘rMemory’ to allocate the required memory. Hence
 *       ‘BeginrMemory’ should be called before ‘Begin_rSrcMem’, and
 *       ‘CloserMemory’ should be called only after ‘Close_rSrcMem’.
 *
 * rNewSrcMem, rDeleteSrcMem, rDeleteAllSrcMem
 * -------------------------------------------
 *  Before a source file can be stored, a new storage has to be created.
 *  Several independent storages may be created and deleted separately.
 *
 * rPrintSrcMem
 * ------------
 *  Writes the given store to a file.
 *
 * rAddSrcMem
 * ----------
 *  Adds an entire buffer into the given storage.
 *  Split the content of the buffer into \n-terminated lines and stores
 *  them. If this routine is called several times, and the last line of the
 *  last call was not terminated by a \n, the first line of current buffer is
 *  concatenated to the last line of the last call.
 *
 * rSetPositionForSrcMem
 * ---------------------
 *  Some times the first call of ‘rAddSrcMem’ should not be stored as line 1,
 *  column 1, but as some other line and column number (e.g. calling a
```

```
 *  separate scanner/parser for some embedded language).
 *  This routine can be used to set the initial line/column information.
 *  That first line must contain a '%line digits,digits' directive.
 *  This directive is removed and replaced by 'column-1' spaces.
 *
 * rSetTabStop
 * -----------
 *  Sets the tab stop, default: 8
 *
 * rGetSrcMem
 * ----------
 *  Fills a buffer with the source contained between the start and end
 *  position. Tabulator characters are expanded. \n are added if several
 *  lines are retrieved.
 *  If the result consists of only one line, and the start column is > 1,
 *  no leading blanks are inserted. If the result consists of more than
 *  one line, and the start column is > 1 leading blanks are inserted.
 */

/* You may use these data structures, but only in a read-only fashion! */
/* a single line of source */
typedef struct {
  char   *line;
  size_t  size; /* of allocated memory for the line, this must not be
                 * equal to 'strlen (line)'.
                 */
  rbool   nl;   /* true, if the line is terminated with a \n.
                 * Note: \n char's are not stored.
                 */
} trSrcMem_line;

/* all source lines */
typedef struct {
  long first_nr;
        /* The "real" source line number of array element 0.
         * In the "standalone" system this is always 1. If the scanner
         * is called for an embedded language, the first line contains a
         * a %-Position directive, setting the "real" source line number,
         * which is stored here. Hence array element 'i' stores the
         * physical line with line number 'i + first_nr'.
         * The line containing the %-Position directive
         * is not stored in this structure.
         */
  long last_nr;
        /* The array elements 0 .. last_nr are used. */
  unsigned long max_lines;
        /* The array contains 0 .. max_lines elements. */
  trSrcMem_line *lines;
        /* Dynamic array of lines, c.f. DynArray.h */
} trSrcMem_elem, *trSrcMem;


extern void Begin_rSrcMem (void);
/* Initialize the rSrcMem module. */

extern void Close_rSrcMem (void);
/* Terminate the module (deallocates all allocated memory for this module) */
```

```
extern trSrcMem rNewSrcMem (void);
/* Create a new source memory data structure. Return a handle to it. */

extern void rDeleteSrcMem (trSrcMem store);
/* Delete the ‘store’. */

extern void rDeleteAllSrcMem (void);
/* Delete all stores created before. */

extern void rPrintSrcMem (FILE *file, trSrcMem store);
/* write ‘store’ to ‘file’ */

extern void  rAddSrcMem (trSrcMem store, const char *buffer, int size);
/* Split the content of ‘buffer’ into \n-terminated lines and store
 * them ‘store’. ‘size’ is the number of char’s contained in ‘buffer’.
 */

extern void rSetPositionForSrcMem (trSrcMem store,
                                   int      line,
                                   int      column);
/* Set store->first_nr.
 * The %-Directive-stuff is removed from the store->lines[0].line,
 * ‘column-1’ blanks are inserted instead of the directive.
 */

extern void rSetTabStop (int tab);
/* Sets the tab stop to ‘tab’. default: 8 */

extern char *rGetSrcMem (trSrcMem       store,
                         tPosition      from,
                         tPosition      to,
                         char          *buffer,
                         unsigned long *size);
/* Returns the source stored within the given position range.
 * If ‘buffer’ == NULL
 * then an global internal buffer is used, large enough to hold all characters
 *      of of the given range. This buffer is reused the next time this routine
 *      is called with ‘buffer’ == NULL. The address of that buffer is
 *      returned.
 *      If ’size’ != NULL *size = strlen (buffer)
 * If ‘buffer’ != NULL
 * then the buffer can store at most ‘size’ characters. If the result is
 *      larger, the result its truncated. ‘buffer’ is returned.
 *      *size = strlen (buffer)
 * The resulting string is always \0 terminated.
 */
```

Contents