Reusable Software
A Collection of MODULA-Modules

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Reusable Software
A Collection of MODULA-Modules

Josef Grosch

Aug. 4 1992

Report No. 4

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Abstract

A brief description of my personal collection of reusable modules written in MODULA-2 is given. Some modules have been translated to the language C.

1. Overview

The following modules exist currently (Aug. 1992):

Module	Task
Memory	dynamic storage (heap) with free lists
Heap	dynamic storage (heap) without free lists
DynArray	dynamic and flexible arrays
Strings	string handling
StringMem	string memory
Idents	identifier table - unambiguous encoding of strings
Lists	lists of arbitrary objects
Texts	texts are lists of strings (lines)
Sets	sets of scalar values (without run time checks)
SetsC	sets of scalar values (with run time checks)
Relations	binary relations between scalar values
IO	buffered input and output
StdIO	buffered IO for standard files
Layout	more routines for input and output
Positions	handling of source positions
Errors	error handler for parsers and compilers
Source	provides input for scanners
Sort	quicksort for arrays with elements of arbitrary type
General	miscellaneous functions
System	machine dependent code
Checks	checks for system calls
Times	access to cpu-time

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

```
DEFINITION MODULE Memory;

VAR       MemoryUsed    : LONGCARD;
                        (* Holds the total amount of memory managed by  *)
                        (* this module.                                 *)

PROCEDURE Alloc         (ByteCount: LONGINT) : ADDRESS;
                        (* Returns a pointer to dynamically allocated   *)
                        (* memory space of size 'ByteCount' bytes.      *)
                        (* Returns NIL if space is exhausted.           *)

PROCEDURE Free          (ByteCount: LONGINT; a: ADDRESS);
                        (* The dynamically allocated space starting at  *)
                        (* address 'a' of size 'ByteCount' bytes is     *)
                        (* released.                                    *)

END Memory.
```

3. Heap: dynamic storage (heap) without free lists

```
DEFINITION MODULE Heap;

VAR       HeapUsed        : LONGCARD;
                          (* Holds the total amount of memory managed by  *)
                          (* this module.                                 *)

PROCEDURE Alloc           (ByteCount: LONGINT) : ADDRESS;
                          (* Returns a pointer to dynamically allocated   *)
                          (* space of size 'ByteCount' bytes.             *)

PROCEDURE Free            ;
                          (* The complete space allocated for the heap    *)
                          (* is released.                                 *)

END Heap.
```

4. 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.

```
DEFINITION MODULE DynArray;

PROCEDURE MakeArray       (VAR ArrayPtr   : ADDRESS       ;
                           VAR ElmtCount  : LONGINT       ;
                               ElmtSize   : LONGINT)      ;
                          (* 'ArrayPtr' is set to the start address of a  *)
                          (* memory space to hold an array of 'ElmtCount' *)
                          (* elements each of size 'ElmtSize' bytes.      *)

PROCEDURE ExtendArray     (VAR ArrayPtr   : ADDRESS       ;
                           VAR ElmtCount  : LONGINT       ;
                               ElmtSize   : LONGINT)      ;
                          (* The memory space for the array is increased  *)
                          (* by doubling the number of elements.          *)

PROCEDURE ReleaseArray    (VAR ArrayPtr   : ADDRESS       ;
                           VAR ElmtCount  : LONGINT       ;
                               ElmtSize   : LONGINT)      ;
                          (* The memory space for the array is released.  *)

END DynArray.
```

Example:

There is a problem with the index arithmetic generated by the compiler. If TSIZE (ArrayType) < 65536 then index arithmetic will be done with 2 byte values otherwise with 4 byte values. Arithmetic with 2 bytes allows only access to arrays whose size never exceeds 65536 bytes. If larger arrays are desired then ArrayType has to be declared to yield a size greater or equal to 65536 bytes like below.

```
CONST InitialSize       = 100;
TYPE  ElmtType          = ... ;
TYPE  ArrayType         = ARRAY [0 .. 100000] OF ElmtType;
VAR   ActualSize        : LONGINT;
VAR   ArrayPtr          : POINTER TO ArrayType;

BEGIN
   ActualSize := InitialSize;
   MakeArray (ArrayPtr, ActualSize, TSIZE (ElmtType));

   (* Case 1: continuously growing array *)

   Index := 0;
   LOOP
      INC (Index);
      IF Index = ActualSize THEN
         ExtendArray (ArrayPtr, ActualSize, TSIZE (ElmtType));
      END;

      ArrayPtr ^ [Index] := ... ;   (* access an array element *)
       ... := ArrayPtr ^ [Index];   (* "      "  "     "       *)
   END;

   (* Case 2: non-continuously growing array *)

   LOOP
      Index := ... ;
      WHILE Index >= ActualSize DO
         ExtendArray (ArrayPtr, ActualSize, TSIZE (ElmtType));
      END;

      ArrayPtr ^ [Index] := ... ;   (* access an array element *)
       ... := ArrayPtr ^ [Index];   (* "      "  "     "       *)
   END;

   ReleaseArray (ArrayPtr, ActualSize, TSIZE (ElmtType));
END;
```

5. Strings: string handling

This module provides operations on strings whose length is at most 255 characters.

```
DEFINITION MODULE Strings;

CONST   cMaxStrLength   = 255;

TYPE    tString         ;
TYPE    tStringIndex    = [0 .. cMaxStrLength];

PROCEDURE Assign        (VAR s1, s2: tString);
                        (* Assigns the string 's2' to the string 's1'.  *)

PROCEDURE AssignEmpty   (VAR s: tString);
                        (* Returns an empty string 's'.                 *)

PROCEDURE Concatenate   (VAR s1, s2: tString);
                        (* Returns in parameter 's1' the concatenation  *)
                        (* of the strings 's1' and 's2'.                *)

PROCEDURE Append        (VAR s: tString; c: CHAR);
                        (* The character 'c' is concatenated at the end *)
                        (* of the string 's'.                           *)

PROCEDURE Length        (VAR s: tString)                        : CARDINAL;
                        (* Returns the length of the string 's'.        *)

PROCEDURE IsEqual       (VAR s1, s2: tString)                   : BOOLEAN;
                        (* Returns TRUE if the strings 's1' and 's2'    *)
                        (* are equal.                                   *)

PROCEDURE IsInOrder     (VAR s1, s2: tString)                   : BOOLEAN;
                        (* Returns TRUE if the string 's1' is lexico-   *)
                        (* graphically less or equal to the string 's2'.*)

PROCEDURE Exchange      (VAR s1, s2: tString);
                        (* Exchanges the strings 's1' and 's2'.         *)

PROCEDURE SubString     (VAR s1: tString; from, to: tStringIndex; VAR s2: tString);
                        (* Returns in 's2' the substring from 's1' com- *)
                        (* prising the characters between 'from' and 'to'. *)
                        (* PRE  1 <= from <= Length (s1)                *)
                        (* PRE  1 <=  to  <= Length (s1)                *)

PROCEDURE Char          (VAR s: tString; i: tStringIndex)       : CHAR;
                        (* Returns the 'i'-th character of the string 's'. *)
                        (* The characters are counted from 1 to Length (s). *)
                        (* PRE  1 <= index <= Length (s)                *)

PROCEDURE ArrayToString (a: ARRAY OF CHAR; VAR s: tString);
                        (* An array 'a' of characters representing a    *)
                        (* MODULA string is converted to a string 's'   *)
                        (* of type tString.                             *)

PROCEDURE StringToArray (VAR s: tString; VAR a: ARRAY OF CHAR);
                        (* A string 's' of type tString is converted to *)
                        (* an array 'a' of characters representing a    *)
                        (* MODULA string.                               *)

PROCEDURE StringToInt   (VAR s: tString)                        : INTEGER;
```

```
                        (* Returns the integer value represented by 's'. *)

PROCEDURE StringToNumber(VAR s: tString; Base: CARDINAL)        : CARDINAL;
                        (* Returns the integer value represented by 's' *)
                        (* to the base 'Base'.                          *)

PROCEDURE StringToReal  (VAR s: tString)                        : REAL;
                        (* Returns the real value represented by 's'.   *)

PROCEDURE IntToString   (n: INTEGER; VAR s: tString);
                        (* Returns in 's' the string representation of 'n'. *)

PROCEDURE ReadS         (f: tFile; VAR s: tString; FieldWidth: tStringIndex);
                        (* Read 'FieldWidth' characters as string 's'   *)
                        (* from file 'f'.                               *)

PROCEDURE ReadL         (f: tFile; VAR s: tString);
                        (* Read rest of line as string 's' from file    *)
                        (* 'f'. Skip to next line.                      *)

PROCEDURE WriteS        (f: tFile; VAR s: tString);
                        (* Write string 's' to file 'f'.                *)

PROCEDURE WriteL        (f: tFile; VAR s: tString);
                        (* Write string 's' as complete line.           *)

END Strings.
```

6. StringMem: string memory

```
DEFINITION MODULE StringMem;

TYPE tStringRef         ;

PROCEDURE PutString     (VAR s: tString)                         : tStringRef;
                        (* Stores string 's' in the string memory and   *)
                        (* returns a reference to the stored string.    *)

PROCEDURE GetString     (r: tStringRef;                    VAR s: tString);
                        (* Returns the string 's' from the string       *)
                        (* memory which is referenced by 'r'.           *)

PROCEDURE Length        (r: tStringRef)                          : CARDINAL;
                        (* Returns the length of the string 's'         *)
                        (* which is referenced by 'r'.                  *)

PROCEDURE IsEqual       (r: tStringRef; VAR s: tString)          : BOOLEAN;
                        (* Compares the string referenced by 'r' and    *)
                        (* the string 's'.                              *)
                        (* Returns TRUE if both are equal.              *)

PROCEDURE WriteString   (f: tFile; r: tStringRef);
                        (* The string referenced by 'r' is printed on   *)
                        (* the file 'f'.                                *)

PROCEDURE WriteStringMemory;
                        (* The contents of the string memory is printed *)
                        (* on the file 'StdOutput'.                     *)

PROCEDURE InitStringMemory;
                        (* The string memory is initialized.            *)

END StringMem.
```

7. Idents: identifier table - unambiguous encoding of strings

```
DEFINITION MODULE Idents;

TYPE      tIdent        ;

VAR       NoIdent       : tIdent;       (* a default (empty) identifier *)

PROCEDURE MakeIdent     (VAR s: tString)                : tIdent;
                        (* The string 's' is mapped to a unique         *)
                        (* identifier (an integer) which is returned.   *)

PROCEDURE GetString     (i: tIdent; VAR s: tString);
                        (* Returns the string 's' whose identifier is 'i'. *)

PROCEDURE GetStringRef  (i: tIdent)                     : tStringRef;
                        (* Returns a reference to a string whose        *)
                        (* identifier is 'i'.                           *)

PROCEDURE MaxIdent      ()                              : tIdent;
                        (* Returns the current maximal value of the     *)
                        (* type 'tIdent'.                               *)

PROCEDURE WriteIdent    (f: tFile; i: tIdent);
                        (* The string encoded by the identifier 'i' is  *)
                        (* printed on the file 'f'.                     *)

PROCEDURE WriteIdents   ;
                        (* The contents of the identifier table is      *)
                        (* printed on the file 'StdOutput'.             *)

PROCEDURE InitIdents    ;
                        (* The identifier table is initialized.         *)

END Idents.
```

8. Lists: lists of arbitrary objects

This module actually handles lists of untyped pointers. Thus it is possible to have lists of pointers referring to objects of arbitrary types.

```
DEFINITION MODULE Lists;

TYPE tList              ;

PROCEDURE MakeList      (VAR List: tList);
                        (* Create an empty list.                        *)

PROCEDURE Insert        (VAR List: tList; Elmt: ADDRESS);
                        (* Add element at the beginning of the list.    *)

PROCEDURE Append        (VAR List: tList; Elmt: ADDRESS);
                        (* Add element at the end of the list.          *)

PROCEDURE Head          (    List: tList): ADDRESS;
                        (* Return first element of the list.            *)

PROCEDURE Tail          (VAR List: tList);
                        (* Return list except first element.            *)

PROCEDURE Last          (    List: tList): ADDRESS;
                        (* Return last element of the list.             *)

PROCEDURE Front         (VAR List: tList);
                        (* Return list except last element.             *)
                        (* Not implemented.                             *)

PROCEDURE IsEmpty       (    List: tList): BOOLEAN;
                        (* Returns TRUE if the list is empty.           *)

PROCEDURE Length        (    List: tList): CARDINAL;
                        (* Returns the number of elements in the list.  *)

PROCEDURE WriteList     (f: tFile; List: tList; Proc: tProcOfFileAddress);
                        (* Call Proc (f, e) for all elements of a list. *)
                        (* e points to the current list element.        *)
                        (* Can be used e. g. to print a list.           *)

END Lists.
```

9. Texts: texts are lists of strings (lines)

```
DEFINITION MODULE Texts;

TYPE tText              ;

PROCEDURE MakeText      (VAR Text: tText);
                        (* Create an empty text.                        *)

PROCEDURE Append        (VAR Text: tText; VAR String: tString);
                        (* Add a line at the beginning of text 'Text'.  *)

PROCEDURE Insert        (VAR Text: tText; VAR String: tString);
                        (* Add a line at the end of the text 'Text'.    *)

PROCEDURE IsEmpty       (VAR Text: tText): BOOLEAN;
                        (* Test whether a text 'Text' is empty.         *)

PROCEDURE WriteText     (f: tFile; Text: tText);
                        (* Print the text 'Text' on the file 'f'.       *)

END Texts.
```

10. 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 INTEGER, CARDINAL, CHAR, or of an enumeration type. Elements of type CHAR or of enumeration types have to be coerced to the type CARDINAL with the function ORD before use as a parameter of the functions below. 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 (ARRAY OF BITSET) 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.

```
DEFINITION MODULE Sets;

TYPE tSet;
TYPE ProcOfCard          = PROCEDURE (CARDINAL);
TYPE ProcOfCardToBool    = PROCEDURE (CARDINAL): BOOLEAN;

PROCEDURE MakeSet        (VAR Set: tSet; MaxSize: CARDINAL);
PROCEDURE ReleaseSet     (VAR Set: tSet);
PROCEDURE Union          (VAR Set1: tSet; Set2: tSet);
PROCEDURE Difference     (VAR Set1: tSet; Set2: tSet);
PROCEDURE Intersection   (VAR Set1: tSet; Set2: tSet);
PROCEDURE SymDiff        (VAR Set1: tSet; Set2: tSet);
PROCEDURE Complement     (VAR Set: tSet);
PROCEDURE Include        (VAR Set: tSet; Elmt: CARDINAL);
PROCEDURE Exclude        (VAR Set: tSet; Elmt: CARDINAL);
PROCEDURE Card           (VAR Set: tSet): CARDINAL;
PROCEDURE Size           (VAR Set: tSet): CARDINAL;
PROCEDURE Minimum        (VAR Set: tSet): CARDINAL;
PROCEDURE Maximum        (VAR Set: tSet): CARDINAL;
PROCEDURE Select         (VAR Set: tSet): CARDINAL;
PROCEDURE Extract        (VAR Set: tSet): CARDINAL;
PROCEDURE IsSubset       (Set1, Set2: tSet): BOOLEAN;
PROCEDURE IsStrictSubset(Set1, Set2: tSet): BOOLEAN;
PROCEDURE IsEqual        (VAR Set1, Set2: tSet): BOOLEAN;
PROCEDURE IsNotEqual     (Set1, Set2: tSet): BOOLEAN;
PROCEDURE IsElement      (Elmt: CARDINAL; VAR Set: tSet): BOOLEAN;
PROCEDURE IsEmpty        (Set: tSet): BOOLEAN;
PROCEDURE Forall         (Set: tSet; Proc: ProcOfCardToBool): BOOLEAN;
PROCEDURE Exists         (Set: tSet; Proc: ProcOfCardToBool): BOOLEAN;
PROCEDURE Exists1        (Set: tSet; Proc: ProcOfCardToBool): BOOLEAN;
PROCEDURE Assign         (VAR Set1: tSet; Set2: tSet);
PROCEDURE AssignElmt     (VAR Set: tSet; Elmt: CARDINAL);
PROCEDURE AssignEmpty    (VAR Set: tSet);
PROCEDURE ForallDo       (Set: tSet; Proc: ProcOfCard);
PROCEDURE ReadSet        (f: tFile; VAR Set: tSet);
PROCEDURE WriteSet       (f: tFile;     Set: tSet);

END Sets.
```

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 CARDINAL 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'.
ReleaseSet	releases the space taken by a set.
Union	Set1 := Set1 $\cup$ Set2
Difference	Set1 := Set1 - Set2
Intersection	Set1 := Set1 $\cap$ Set2
SymDiff	Set1 := Set1 Set2 (* corresponds to exclusive or *)
Complement	Set := { 0 .. MaxSize } - Set
Include	Set := Set $\cup$ { Elmt }
Exclude	Set := Set - { Elmt }
Card	returns number of elements in 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 $\subseteq$ Set2
IsStrictSubset	Set1 $\subset$ Set2
IsEqual	Set1 = Set2
IsNotEqual	Set1 $\neq$ Set2
IsElement	Elmt $\in$ Set
IsEmpty	Set = Ø
Forall	$\forall$ e $\in$ Set : Proc (e) (* predicate Proc must hold for all elements *)
Exists	$\exists$ e $\in$ Set : Proc (e) (* predicate Proc must hold for at least 1 element *)
Exists1	| { e $\in$ Set : Proc (e) } | = 1
Assign	Set1 := Set2
AssignElmt	Set1 := { Elmt }
AssignEmpty	Set1 := Ø

ForallDo	FOR e := 0 TO MaxSize DO
	IF e $\in$ Set THEN Proc (e); END;
	END;

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}

11. SetsC: sets for scalar values

This module provides the same procedures as the module Sets with the difference that all possible run time checks are performed.

12. Relations: 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 INTEGER, CARDINAL, CHAR, or of an enumeration type. Arguments of type CHAR or of enumeration types have to be coerced to the type CARDINAL with the function ORD before use as a parameter of the functions below. 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 exact 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.

```
DEFINITION MODULE Relations;

TYPE tRelation;
TYPE ProcOfIntInt       = PROCEDURE (INTEGER, INTEGER);
TYPE ProcOfIntIntToBool = PROCEDURE (INTEGER, INTEGER): BOOLEAN;

PROCEDURE MakeRelation  (VAR Rel: tRelation; Size1, Size2: INTEGER);
PROCEDURE ReleaseRelation (VAR Rel: tRelation);
PROCEDURE Include       (VAR Rel: tRelation; e1, e2: INTEGER);
PROCEDURE Exclude       (VAR Rel: tRelation; e1, e2: INTEGER);
PROCEDURE IsElement     (e1, e2: INTEGER; Rel: tRelation): BOOLEAN;
PROCEDURE IsRelated     (e1, e2: INTEGER; Rel: tRelation): BOOLEAN;
PROCEDURE IsReflexive1  (e1: INTEGER; Rel: tRelation): BOOLEAN;
PROCEDURE IsSymmetric1  (e1, e2: INTEGER; Rel: tRelation): BOOLEAN;
PROCEDURE IsTransitive1 (e1, e2, e3: INTEGER; Rel: tRelation): BOOLEAN;
PROCEDURE IsReflexive   (Rel: tRelation): BOOLEAN;
PROCEDURE IsSymmetric   (Rel: tRelation): BOOLEAN;
PROCEDURE IsTransitive  (Rel: tRelation): BOOLEAN;
PROCEDURE IsEquivalence (Rel: tRelation): BOOLEAN;
PROCEDURE HasReflexive  (Rel: tRelation): BOOLEAN;
PROCEDURE IsCyclic      (Rel: tRelation): BOOLEAN;
PROCEDURE Closure       (VAR Rel: tRelation);
PROCEDURE AssignEmpty   (VAR Rel: tRelation);
PROCEDURE AssignElmt    (VAR Rel: tRelation; e1, e2: INTEGER);
PROCEDURE Assign        (VAR Rel1: tRelation; Rel2: tRelation);
PROCEDURE Union         (VAR Rel1: tRelation; Rel2: tRelation);
PROCEDURE Difference    (VAR Rel1: tRelation; Rel2: tRelation);
PROCEDURE Intersection  (VAR Rel1: tRelation; Rel2: tRelation);
PROCEDURE SymDiff       (VAR Rel1: tRelation; Rel2: tRelation);
PROCEDURE Complement    (VAR Rel: tRelation);
PROCEDURE IsSubset      (Rel1, Rel2: tRelation): BOOLEAN;
PROCEDURE IsStrictSubset (Rel1, Rel2: tRelation): BOOLEAN;
PROCEDURE IsEqual       (VAR Rel1, Rel2: tRelation): BOOLEAN;
PROCEDURE IsNotEqual    (Rel1, Rel2: tRelation): BOOLEAN;
PROCEDURE IsEmpty       (Rel: tRelation): BOOLEAN;
PROCEDURE Card          (VAR Rel: tRelation): INTEGER;
PROCEDURE Select        (VAR Rel: tRelation; VAR e1, e2: INTEGER);
PROCEDURE Extract       (VAR Rel: tRelation; VAR e1, e2: INTEGER);
PROCEDURE Forall        (Rel: tRelation; Proc: ProcOfIntIntToBool): BOOLEAN;
PROCEDURE Exists        (Rel: tRelation; Proc: ProcOfIntIntToBool): BOOLEAN;
PROCEDURE Exists1       (Rel: tRelation; Proc: ProcOfIntIntToBool): BOOLEAN;
PROCEDURE ForallDo      (Rel: tRelation; Proc: ProcOfIntInt);
PROCEDURE ReadRelation  (f: tFile; VAR Rel: tRelation);
PROCEDURE WriteRelation (f: tFile;     Rel: tRelation);

END Relations.
```

13. IO: buffered input and output

This module provides procedures for buffered input and output to files. Io to and from terminals is possible too, because terminals are regarded as special cases of files. There are procedures for binary io as well as for text io, that is internal values are converted to their external representation and vice versa.

```
DEFINITION MODULE IO;

CONST   StdInput        ;   (* A pre-opened file for (terminal-)input.  *)
CONST   StdOutput       ;   (* A pre-opened file for (terminal-)output. *)
CONST   StdError        ;   (* A pre-opened file for (terminal-)output  *)
                            (*    of error messages.                    *)
TYPE    tFile           ;

PROCEDURE ReadOpen      (FileName: ARRAY OF CHAR): tFile;
                                               (* open  input file     *)
PROCEDURE ReadClose     (f: tFile);            (* close input file     *)
PROCEDURE Read          (f: tFile; Buffer: ADDRESS; Size: CARDINAL): INTEGER;
                                               (* binary               *)
PROCEDURE ReadC         (f: tFile): CHAR    ;  (* character            *)
PROCEDURE ReadI         (f: tFile): INTEGER ;  (* integer  number      *)
PROCEDURE ReadR         (f: tFile): REAL    ;  (* real     number      *)
PROCEDURE ReadB         (f: tFile): BOOLEAN ;  (* boolean              *)
PROCEDURE ReadN         (f: tFile; Base: INTEGER): INTEGER;
                                               (* number of base 'Base'*)
PROCEDURE ReadS         (f: tFile; VAR s: ARRAY OF CHAR);
                                               (* string               *)
PROCEDURE ReadNl        (f: tFile);            (* new line             *)
PROCEDURE UnRead        (f: tFile);            (* backspace 1 char.    *)

PROCEDURE EndOfLine     (f: tFile): BOOLEAN ;  (* end of line ?        *)
PROCEDURE EndOfFile     (f: tFile): BOOLEAN ;  (* end of file ?        *)

PROCEDURE WriteOpen     (FileName: ARRAY OF CHAR): tFile;
                                               (* open  output file    *)
PROCEDURE WriteClose    (f: tFile);            (* close output file    *)
PROCEDURE WriteFlush    (f: tFile);            (* flush output buffer  *)
PROCEDURE Write         (f: tFile; Buffer: ADDRESS; Size: CARDINAL): INTEGER;
                                               (* binary               *)
PROCEDURE WriteC        (f: tFile; c: CHAR);   (* character            *)
PROCEDURE WriteI        (f: tFile; n: INTEGER ; FieldWidth: CARDINAL);
                                               (* integer  number      *)
PROCEDURE WriteR        (f: tFile; n: REAL; Before, After, Exp: CARDINAL);
                                               (* real     number      *)
PROCEDURE WriteB        (f: tFile; b: BOOLEAN); (* boolean             *)
PROCEDURE WriteN        (f: tFile; n: LONGCARD; FieldWidth, Base: CARDINAL);
                                               (* number of base 'Base'*)
PROCEDURE WriteS        (f: tFile; s: ARRAY OF CHAR);
                                               (* string               *)
PROCEDURE WriteNl       (f: tFile);            (* new line             *)

PROCEDURE CloseIO;                             (* close all files      *)
END IO.
```

Notes:

ReadOpen and WriteOpen
: Open the file whose name is given by the parameter 'FileName' for input resp. output. A file descriptor will be returned.

Read and Write
: Implement binary i/o for byte blocks.

ReadI, ReadR, ReadB, ReadN
: This procedures skip blanks before reading a value.

ReadR and WriteR
: 'ReadR' reads real numbers according to the following syntax:
  [+|-] [digit* . digit*] [E [+|-] digit+]
  'WriteR' writes real numbers similar to the Ada output routine PUT:
  Before . After E Exp
  where Before, After, and Exp give the lengths of the fields. If Exp is 0
  no exponent part is written.

ReadB and WriteB
: The external representation for boolean values are T for TRUE an F for FALSE. On input every character other than T is interpreted as FALSE.

ReadN and WriteN
: Procedures for i/o of octal (Base = 8) or hexadecimal (Base = 16) numbers for example. Base must have a value between 2 and 16.

ReadNl
: Read all characters up to and including the next end of line character.

EndOfLine
: Checks whether end of line has been reached. If this is the case the next character to be read would be an end of line character.

EndOfFile
: Checks whether end of file has been reached. If this is the case none of the input routines can yield any defined result.

WriteFlush
: The actual contents of the buffer for the specified file is written unconditionally to the file.

WriteNl
: Writes an end of line character.

CloseIO
: This procedure has to be called at the end of every program that uses the module IO. All buffers associated with files opened for output are flushed.

14. StdIO: buffered IO for standard files

This module provides the same procedures as the module IO with the difference that the parameter 'f' to specify the file is missing. The input (output) procedures use the file 'StdInput' ('StdOutput').

15. Layout: more routines for input and output

```
DEFINITION MODULE Layout;

PROCEDURE WriteChar     (f: tFile; Ch: CHAR);
                        (* Printable characters are surrounded by       *)
                        (* quotes. Non-printable characters are written *)
                        (* as their internal code followed by a 'C'.    *)

PROCEDURE WriteSpace    (f: tFile);
                        (* Write a blank character to file 'f'.         *)

PROCEDURE WriteSpaces   (f: tFile; Count: INTEGER);
                        (* Write 'Count' blank characters to file 'f'.  *)

PROCEDURE ReadSpace     (f: tFile);
                        (* Skip one character of file 'f'.              *)

PROCEDURE ReadSpaces    (f: tFile; Count: INTEGER);
                        (* Skip 'Count' characters of file 'f'.         *)

PROCEDURE SkipSpaces    (f: tFile);
                        (* Skip as many blank characters as possible.   *)

END Layout.
```

16. Positions: handling of source positions

A simple representation of the position of tokens in a source file consisting of a line and a column field. This module should be copied and tailored to the user's needs, if necessary. Modifications may be necessary if the type SHORTCARD is to small to count the lines or an extra field is needed to describe the source file.

```
DEFINITION MODULE Positions;

TYPE      tPosition     = RECORD Line, Column: SHORTCARD; END;

VAR       NoPosition    : tPosition;
                        (* A default position (0, 0).                   *)

PROCEDURE Compare       (Position1, Position2: tPosition): INTEGER;
                        (* Returns -1 if Position1 < Position2.         *)
                        (* Returns  0 if Position1 = Position2.         *)
                        (* Returns  1 if Position1 > Position2.         *)

PROCEDURE WritePosition (File: tFile; Position: tPosition);
                        (* The 'Position' is printed on the 'File'.     *)

END Positions.
```

17. Errors: error handler for parsers and compilers

This module is needed by parsers generated with the parser generators *lalr* or *ell*. It can also b used to report error messages found during scanning or semantic analysis. Note: This module has to be copied, too, if the module *Positions* is copied and modified because it depends upon this module.

```
DEFINITION MODULE Errors;

CONST
   NoText              = 0     ;
   SyntaxError         = 1     ;        (* error codes          *)
   ExpectedTokens      = 2     ;
   RestartPoint        = 3     ;
   TokenInserted       = 4     ;
   WrongParseTable     = 5     ;
   OpenParseTable      = 6     ;
   ReadParseTable      = 7     ;
   TooManyErrors       = 8     ;

   Fatal               = 1     ;        (* error classes        *)
   Restriction         = 2     ;
   Error               = 3     ;
   Warning             = 4     ;
   Repair              = 5     ;
   Note                = 6     ;
   Information         = 7     ;

   None                = 0     ;
   Integer             = 1     ;        (* info classes         *)
   Short               = 2     ;
   Long                = 3     ;
   Real                = 4     ;
   Boolean             = 5     ;
   Character           = 6     ;
   String              = 7     ;
   Array               = 8     ;
   Set                 = 9     ;
   Ident               = 10    ;
```

```
VAR       Exit           : PROC;
                         (* Refers to a procedure that specifies         *)
                         (* what to do if 'ErrorClass' = Fatal.          *)
                         (* Default: terminate program execution.        *)

PROCEDURE StoreMessages (Store: BOOLEAN);
                         (* 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.*)

PROCEDURE ErrorMessage  (ErrorCode, ErrorClass: CARDINAL; Position: tPosition);
                         (* Report a message represented by an integer   *)
                         (* 'ErrorCode' and classified by 'ErrorClass'.  *)

PROCEDURE ErrorMessageI (ErrorCode, ErrorClass: CARDINAL; Position: tPosition;
                          InfoClass: CARDINAL; Info: ADDRESS);
                         (* Like the previous routine with additional    *)
                         (* information of type 'InfoClass' at the       *)
                         (* address 'Info'.                              *)

PROCEDURE Message       (ErrorText: ARRAY OF CHAR; ErrorClass: CARDINAL;
                          Position: tPosition);
                         (* Report a message represented by a string     *)
                         (* 'ErrorText' and classified by 'ErrorClass'.  *)

PROCEDURE MessageI      (ErrorText: ARRAY OF CHAR; ErrorClass: CARDINAL;
                          Position: tPosition; InfoClass: CARDINAL; Info: ADDRESS);
                         (* Like the previous routine with additional    *)
                         (* information of type 'InfoClass' at the       *)
                         (* address 'Info'.                              *)

PROCEDURE WriteMessages (File: tFile);
                         (* The stored messages are sorted by their      *)
                         (* source position and printed on 'File'.       *)

END Errors.
```

18. Source: provides input for scanners

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

```
DEFINITION MODULE Source;

PROCEDURE BeginSource (FileName: ARRAY OF CHAR): tFile;

   (*
      BeginSource is called from the scanner to open files.
      If not called then input is read form standard input.
   *)

PROCEDURE GetLine (File: tFile; Buffer: ADDRESS; Size: CARDINAL): INTEGER;

   (*
      GetLine is called to fill a buffer starting at address 'Buffer'
      with a block of maximal 'Size' characters. Lines are terminated
      by newline characters (ASCII = 0xa). GetLine returns the number
      of characters transferred. Reasonable block sizes are between 128
      and 2048 or the length of a line. Smaller block sizes -
      especially block size 1 - will drastically slow down the scanner.
   *)

PROCEDURE CloseSource (File: tFile);

   (*
      CloseSource is called from the scanner at end of file or
      at end of input, respectively. It can be used to close files.
   *)

END Source.
```

19. Sort: quicksort for arrays with elements of arbitrary type

```
DEFINITION MODULE Sort;

TYPE tProcIntIntBool    = PROCEDURE (INTEGER, INTEGER): BOOLEAN;
TYPE tProcIntInt        = PROCEDURE (INTEGER, INTEGER);

PROCEDURE Sort (Lwb, Upb: INTEGER; IsLess: tProcIntIntBool; Swap: tProcIntInt);

        (* Sort data from the indices 'Lwb' to 'Upb' using quicksort.   *)
        (* The procedures 'IsLess' and 'Swap' are used to compare and   *)
        (* exchange two data elements.                                  *)

END Sort.
```

20. General: miscellaneous functions

```
DEFINITION MODULE General;

PROCEDURE Min           (a, b: INTEGER)                  : INTEGER;
                        (* Returns the minimum of 'a' and 'b'.          *)

PROCEDURE Max           (a, b: INTEGER)                  : INTEGER;
                        (* Returns the maximum of 'a' and 'b'.          *)

PROCEDURE Log2          (x: LONGINT)                     : CARDINAL;
                        (* Returns the logarithm to the base 2 of 'x'.  *)

PROCEDURE Exp2          (x: CARDINAL)                    : LONGINT;
                        (* Returns 2 to the power of 'x'.               *)

PROCEDURE AntiLog       (x: LONGINT)                     : CARDINAL;
                        (* Returns the number of the lowest bit set in 'x'. *)

PROCEDURE Exp10         (x: INTEGER)                     : REAL;
                        (* Returns 10 to the power of 'x'.              *)

END General.
```

21. System: machine dependent code

This module provides a few machine dependent operations.

```
FOREIGN MODULE System;

CONST   cMaxFile    = 32;
CONST   StdInput        ;   (* A pre-opened file for (terminal-)input.  *)
CONST   StdOutput       ;   (* A pre-opened file for (terminal-)output. *)
CONST   StdError        ;   (* A pre-opened file for (terminal-)output  *)
                            (*    of error messages.                    *)

TYPE    tFile           ;

                        (* binary IO            *)

PROCEDURE OpenInput     (VAR FileName: ARRAY OF CHAR): tFile;
                        (* Opens the file whose name is given by the    *)
                        (* parameter 'FileName' for input.              *)
                        (* Returns a file descriptor.                   *)

PROCEDURE OpenOutput    (VAR FileName: ARRAY OF CHAR): tFile;
                        (* Opens the file whose name is given by the    *)
                        (* parameter 'FileName' for output.             *)
                        (* Returns a file descriptor.                   *)

PROCEDURE Read          (File: tFile; Buffer: ADDRESS; Size: CARDINAL): INTEGER;
                        (* Reads 'Size' bytes from file 'tFile' and     *)
                        (* stores them in a buffer starting at address  *)
                        (* 'Buffer'.                                    *)
                        (* Returns the number of bytes actually read.   *)

PROCEDURE Write         (File: tFile; Buffer: ADDRESS; Size: CARDINAL): INTEGER;
                        (* Writes 'Size' bytes from a buffer starting   *)
                        (* at address 'Buffer' to file 'tFile'.         *)
                        (* Returns the number of bytes actually written.*)

PROCEDURE Close         (File: tFile);
                        (* Closes file 'tFile'.                         *)

PROCEDURE IsCharacterSpecial (File: tFile): BOOLEAN;
                        (* Returns TRUE when file 'tFile' is connected  *)
                        (* to a character device like a terminal.       *)

                        (* calls other than IO  *)

PROCEDURE SysAlloc      (ByteCount: LONGINT): ADDRESS;
                        (* Returns a pointer to dynamically allocated   *)
                        (* memory space of size 'ByteCount' bytes.      *)
                        (* Returns NIL if space is exhausted.           *)

PROCEDURE Time          (): LONGINT;
                        (* Returns consumed cpu-time in milliseconds.   *)

PROCEDURE GetArgCount   (): CARDINAL;
                        (* Returns number of arguments.                 *)

PROCEDURE GetArgument   (ArgNum: CARDINAL; VAR Argument: ARRAY OF CHAR);
                        (* Stores a string-valued argument whose index  *)
                        (* is 'ArgNum' in the memory area 'Argument'.   *)
```

```
PROCEDURE PutArgs       (Argc: CARDINAL; Argv: ADDRESS);
                        (* Dummy procedure that passes the values       *)
                        (* 'argc' and 'argv' from Modula-2 to C.        *)

PROCEDURE ErrNum        (): INTEGER;
                        (* Returns the current system error code.       *)

PROCEDURE System        (VAR String: ARRAY OF CHAR): INTEGER;
                        (* Executes an operating system command given   *)
                        (* as the string 'String'. Returns an exit or   *)
                        (* return code.                                 *)

PROCEDURE Exit          (Status: INTEGER);
                        (* Terminates program execution and passes the  *)
                        (* value 'Status' to the operating system.      *)

END System.
```

22. Checks: checks for system calls

```
DEFINITION MODULE Checks;

PROCEDURE ErrorCheck    (s: ARRAY OF CHAR; n: INTEGER);
                        (* The parameter 'n' should be a value returned *)
                        (* from a system call. If it is negative the    *)
                        (* string 's', the value of 'n', and the value  *)
                        (* returned by 'ErrNum' are printed on the file *)
                        (* 'StdError'.                                  *)

END Checks.
```

Example:

```
   VAR n: INTEGER;

   n := OpenInput ("DataFile");
   ErrorCheck ("error at open of DataFile", n);
```

23. Times: access to cpu-time

```
DEFINITION MODULE Times;

PROCEDURE CpuTime       (): LONGINT;
                        (* Returns the sum of user time and system time *)
                        (* since the start of the program in milli-secs.*)

PROCEDURE StepTime      (): LONGINT;
                        (* Returns the sum of user time and system time *)
                        (* since the last call to 'StepTime' in milli-  *)
                        (* seconds. The first call yields the same      *)
                        (* result as a call to 'CpuTime'.               *)

PROCEDURE WriteStepTime (Text: ARRAY OF CHAR);
                        (* Writes a line consisting of the string       *)
                        (* 'Text' and the value obtained from a call to *)
                        (* 'StepTime' to the file 'StdOutput'.          *)

END Times.
```

24. An Example: Text represented as a List of Strings

```
MODULE Example;

FROM SYSTEM       IMPORT ADDRESS;
FROM IO           IMPORT StdInput, StdOutput, EndOfFile;
FROM Strings      IMPORT tString, ReadL, WriteL;
FROM StringMem    IMPORT tStringRef, PutString, GetString;
FROM Lists        IMPORT tList, MakeList, Append, IsEmpty, Head, Tail;

VAR String      : tString;
VAR Ref         : tStringRef;
VAR Text        : tList;

BEGIN
   MakeList (Text);                      (* start with empty list        *)
   WHILE NOT EndOfFile (StdInput) DO     (* process complete input file  *)
      ReadL (StdInput, String);          (* read one line                *)
      Ref := PutString (String);         (* store it in string memory    *)
      Append (Text, ADDRESS (Ref));      (* append it to current text    *)
   END;

   WHILE NOT IsEmpty (Text) DO           (* process complete text        *)
      Ref := tStringRef (Head (Text));   (* get current first line       *)
      GetString (Ref, String);           (* fetch string from memory     *)
      WriteL (StdOutput, String);        (* print string in one line     *)
      Tail (Text);                       (* continue with remaining text *)
   END;
END Example.
```

Contents