[Top] [Contents] [Index] [?]

The Programming Language Modula-2

Niklaus Wirth

Report on the Programming Language Modula-2

Fourth Edition

1988 Springer Verlag

1. Introduction
2. Syntax
3. Vocabulary and representation
4. Declarations and scope rules
5. Constant declarations
6. Type declarations
7. Variable declarations
8. Expressions
9. Statements
10. Procedure declarations
11. Modules
12. System-dependent facilities
13. Processes
14. Compilation units

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

1. Introduction

Modula-2 grew out of a practical need for a general, efficiently implementable systems programming language for minicomputers. Its ancestors are Pascal and Modula. From the latter it is inherited the name, the important module concept, and a systematic, modern syntax, from Pascal most of the rest. This includes in particular the data structures, i.e. arrays, records, variant records, sets, and pointers. Structured statements include the familiar if, case, repeat, while, for, and with statements. Their syntax is such that every structure ends with an explicit termination symbol.

The language is essentially machine-independent, with the exception of limitations due to wordsize. This appears to be in contradiction to the notion of a system-programming language, in which it must be possible to express all operations inherent in the underlying computer. The dilemma is resolved with the aid of the module concept. Machine-dependent items can be introduced in specific modules, and their use can thereby effectively be confined and isolated. In particular, the language provides the possibility to relax rules about data type compatibility in these cases. In a capable system-programming language it is possible to express input/output conversion procedures, file handling routines, storage allocators, process schedulers etc. Such facilities must therefore not be included as elements of the language itself, but appear as (so-called low-level) modules which are components of most programs written. Such a collection of standard modules is therefore an essential part of a Module-2 implementation.

The concept of processes and their synchronization with signals as included in Modula is replaced by the lower-level notion of coroutines in Modula-2. It is, however, possible to formulate a (standard) module that implements such processes and signals. The advantage of not including them in the language itself is that the programmer may select a process scheduling algorithm tailored to his particular needs by programming that module on his own. Such a scheduler can even be entirely omitted in simple (but frequent) cases, e.g. when concurrent processes occur as device drivers only.

A modern system programming language should in particular also facilitate the construction of large programs, possibly designed by several people. The modules written by individuals should have well-specified interfaces that can be declared independently of their actual implementations. Modula-2 supports this idea by providing separate definition and implementation modules. The former define all objects exported from the corresponding implementation module; in some cases, such as procedures and types, the definition module specifies only those parts that are relevant to the interface, i.e. to the user or client of the module. This report is not intended as a programmer's tutorial. It is intentionally kept concise, and (we hope) clear. Its function is to serve as a reference for programmers, implementors, and manual writers, and as an arbiter, should they find disagreement.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

2. Syntax

A language is an infinite set of sentences, namely the sentences well formed according to its syntax. In Modula-2 these sentences are called compilation units. Each unit is a finite sequence of symbols from a finite vocabulary. The vocabulary of Module-2 consists of identifiers, numbers, strings, operators, and delimiters. They are called lexical symbols and are composed of sequences of characters. (Note the distinction between symbols and characters.)

To describe the syntax, an extended Backus-Naur Formalism called EBNF is used. Angular brackets [] denote optionality of the enclosed sentential form, and curly brackets {} denote its repetition (possibly 0 times). Syntactic entities (not-terminal symbols) are denoted by English word expressing their intuitive meaning. Symbols of the language vocabulary (terminal symbols) are string enclosing in quote marks or words written in capital letters, so-called reserved words. Syntactic rules (productions) are designated by a $ sign at the left Margin of the line.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

3. Vocabulary and representation

The representation of symbols in terms of characters depends on the underlying character set. The ASCII set is used in this paper, and the following lexical rules must be observed. Blanks must not occur within symbols (except in strings). Blanks and line breaks are ignored unless they are essential to separate two consecutive symbols.

1. Identifiers are sequences of letters and digits. The first character must be a letter.

   ```
   $ ident = letter{letter|digit}.
   ```

   Examples:

   ```
   x scan Modula ETH GetSymbol firstLetter
   ```

2. Numbers are (unsigned) integers or real numbers. Integers are sequences of digits. If the number is followed by the letter B, it is taken as an octal number; if it is followed by the letter H, it is taken as a hexadecimal number; if it is followed by the letter C, it denotes the character with given (octal) ordinal number and is of type CHAR (see section 6.1 Basic types), i.e. is a character constant.

   An integer i in the range 0<=i<=MaxInt can be considered as either of type INTEGER or CARDINAL; if it is in the range MaxInt<=i<=MaxCard, it is of type CARDINAL. For 16-bit computers: MaxInt=32767, MaxCard=65535.

   A real number always contains a decimal point. Optionally it may also contain a decimal scale factor. The letter E is pronounced as "ten to the power of". A real number is of type REAL.

   ```
   $ number = integer | real.
   $ integer = digit{digit} |
   $    octalDigit{octalDigit}("B"|"C")|
   $    digit{hexDigit}"H".
   $ real = digit{digit}"."{digit}[ScaleFactor].
   $ ScaleFactor = "E"["+"|"-"]digit{digit}.
   $ hexDigit = digit|"A"|"B"|"C"|"D"|"E"|"F".
   $ digit = octalDigit|"8"|"9".
   $ octalDigit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7".
   ```

   Examples:

   ```
   1980 37648 7BCH 33C 12.3 45.67E-8
   ```

3. Strings are sequences of characters enclosed in quote marks. Both double quotes and single quotes (apostrophes) may be used as quote marks. However, the opening and closing marks must be the same character, and this character cannot occur within the string. A string must not extend over the end of a line.

   ```
   $ string = "'"{character}"'"|'"'{character}'"'
   ```

   A string consisting of n characters is of type (see section 6.4 Array types)

   ```
   ARRAY [0..n] OF CHAR
   ```

   Examples:

   ```
   "MODULA" "Don't worry!" 'codeword "Barbarossa"'
   ```

4. Operators and delimiters are the special characters, character pairs, or reserved words listed below. These reserved words consist exclusively of capital letters and must not be used in the role of identifiers. The symbols # and <> are synonyms, and so are &, AND, and ~, NOT.

   ```
   +       =       AND         FOR             QUALIFIED
   -       #       ARRAY       FROM            RECORD
   *       <       BEGIN       IF              REPEAT
   /       >       BY          IMPLEMENTATION  RETURN
   :=      <>      CASE        IMPORT          SET
   &       <=      CONST       IN              THEN
   .       >=      DEFINITION  LOOP            TO
   ,       ..      DIV         MOD             TYPE
   ;       :       DO          MODULE          UNTIL
   (       )       ELSE        NOT             VAR
   [       ]       ELSIF       OF              WHILE
   {       }       END         OR              WITH
   ^       |       EXIT        POINTER
   ~               EXPORT      PROCEDURE
   ```

5. Comments may be inserted between any two symbols in a program. They are arbitrary character sequences opened by the bracket (* and closed by *). Comments may be nested, and they do not affect the meaning of a program.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

4. Declarations and scope rules

Every identifier occurring in a program must be introduced by a declaration, unless it is a standard identifier. The latter are considered to be predeclared, and they are valid in all parts of a program. For this reason they are called pervasive. Declarations also serve to specify certain permanent properties of an object, such as whether it is a constant, a type, a variable, a procedure, or a module.

The identifier is then used to refer to the associated object. This is possible in those parts of a program only which are within the so-called scope of the declaration. In general, the scope extends over the entire block (procedure or module declaration) to which the declaration belongs and to which the object is local. The scope rule is augmented by the following cases:

1. If an identifier x defined by a declaration D1 is used in another declaration (not statement) D2, then D1 must textually precede D2.
2. A type T1 can be used in a declaration of a pointer type T (see section 6.7 Pointer types) which textually precedes the declaration of T1, if both T and T1 are declared in the same block. This is a relaxation of rule 1.
3. If an identifier defined in a module M1 is exported, the scope expands over the block which contains M1. If M1 is a compilation unit (see section 14. Compilation units), it extends to all those units which import M1.
4. Field identifiers of a record declaration (see section 6.5 Record types) are valid only in field designators and in with statements referring to a variable of that record type.

An identifier may be qualified. In this case it is prefixed by another identifier which designates the module (see section 11. Modules) in which the qualified identifier is defined. The prefix and the identifier are separated by a period. Standard identifiers appear below.

```
$ qualident = ident{"."ident}.
```

```
ABS         (see section 10.2 Standard procedures)
BITSET      (see section 6.6 Set types)
BOOLEAN     (see section 6.1 Basic types)
CAP         (see section 10.2 Standard procedures)
CARDINAL    (see section 6.1 Basic types)
CHAR        (see section 6.1 Basic types)
CHR         (see section 10.2 Standard procedures)
DEC         (see section 10.2 Standard procedures)
EXCL        (see section 10.2 Standard procedures)
FALSE       (see section 6.1 Basic types)
FLOAT       (see section 10.2 Standard procedures)
HALT        (see section 10.2 Standard procedures)
HIGH        (see section 10.2 Standard procedures)
INC         (see section 10.2 Standard procedures)
INCL        (see section 10.2 Standard procedures)
INTEGER     (see section 6.1 Basic types)
LONGINT     (see section 6.1 Basic types)
LONGREAL    (see section 6.1 Basic types)
MAX         (see section 10.2 Standard procedures)
MIN         (see section 10.2 Standard procedures)
NIL         (see section 6.7 Pointer types)
ODD         (see section 10.2 Standard procedures)
ORD         (see section 10.2 Standard procedures)
PROC        (see section 6.8 Procedure types)
REAL        (see section 6.1 Basic types)
SIZE        (see section 10.2 Standard procedures)
TRUE        (see section 6.1 Basic types)
TRUNC       (see section 10.2 Standard procedures)
VAL         (see section 10.2 Standard procedures)
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

5. Constant declarations

A constant declaration associates an identifier with a constant value.

```
$ ConstantDeclaration = ident"="ConstExpression.
$ ConstExpression = expression.
```

A constant expression is an expression which can be evaluated by a mere textual scan without actually executing the program. Its operands are constants. (see section 8. Expressions).

Examples of constant declarations are

```
N     =100
limit =2*N-1
all   = {0..WordSize}
bound = MAX(INTEGER)-N
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

6. Type declarations

A data type determines a set of values which variables of that type may assume, and it associates an identifier with the type. In the case of structured types, it also defines the structure of variables of this type. There are three different structures, namely arrays, records, and sets.

```
$ TypeDeclaration = ident"="type.
$ type = SimpleType | ArrayType | RecordType |
$        SetType| PointerType | ProcedureType.
$ SimpleType = qualident | enumeration |
$              SubrangeType.
```

Examples:

```
Color       = (red,green,blue)
Index       = [1..80]
Card        = ARRAY Index OF CHAR
Node        = RECORD key: CARDINAL;
                left, right: TreePtr
                END
Tint        = SET OF Color
TreePtr     = POINTER TO Node
Function    = PROCEDURE(CARDINAL): CARDINAL
```

6.1 Basic types
6.2 Enumerations
6.3 Subrange types
6.4 Array types
6.5 Record types
6.6 Set types
6.7 Pointer types
6.8 Procedure types

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

6.1 Basic types

The following basic types are predeclared and denoted by standard identifiers:

1. INTEGER comprises the integers between MIN(INTEGER) and MAX(INTEGER).
2. CARDINAL comprises the integers between 0 and MAX(CARDINAL).
3. BOOLEAN comprises the truth values TRUE or FALSE.
4. CHAR denotes the character set provided by the used computer system.
5. REAL (and LONGREAL) denote finite sets of real numbers.
6. LONGINT comprises the integers between MIN(LONGINT) and MAX(LONGINT).

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

6.2 Enumerations

An enumeration is a list of identifiers that denote the values which constitute a data type. These identifiers are used as constants in the program. They, and no other values, belong to this type. The values are ordered, and the ordering relation is defined by their sequence in the enumeration. The ordinal number of the first value is 0.

```
$ enumeration = "("IdentList")".
$ IdentList = ident{","ident}.
```

Examples of enumerations:

```
(red,green,blue)
(club,diamond,heart,spade)
(Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday)
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

6.3 Subrange types

A type T may be defined as a subrange of another, basic or enumeration type T1 (except REAL) by specification of the least and the highest value in the subrange.

```
$ SubrangeType = [ident]
$   "["ConstExpression".."ConstExpression"]".
```

The first constant specifies the lower bound, and must not be greater than the upper bound. The type T1 of the bounds is called the base type of T, and all operators applicable to operands of type T1 are also applicable to operands of type T. However, a value to be assigned to a variable of a subrange type must lie within the specified interval. The base type can be specified by an identifier preceding the bounds. If it is omitted, and if the lower bound is a non-negative integer , the base type of the subrange is taken to be CARDINAL; if it is a negative integer, it is INTEGER.

A type T1 is said to be compatible with a type T0, if it is declared either as T1=T0 or as a subrange of T0, or if T0 is subrange of T1, or if T0 and T1 are both subranges of the same (base) type.

Examples of subranges types:

```
[0..N-1]
["A".."Z"]
[Monday..Friday]
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

6.4 Array types

An array is a structure consisting of a fixed number of components which are all of the same type, called the component type. The elements of the array are designated by indices, values belonging to the index type. The array type declaration specifies the component type as well as the index type. The latter must be an enumeration, a subrange type, or one of the basic types BOOLEAN or CHAR.

```
$ ArrayType = ARRAY SimpleType
$     {","SimpleType} OF type.
```

A declaration of the form

```
ARRAY T1,T2,...,Tn OF T
```

with n index types T1...Tn must be understood as an abbreviation for the declaration

```
ARRAY T1 OF
  ARRAY T2 OF
    ...
      ARRAY Tn OF T
```

Examples of array types:

```
ARRAY [0..N-1] OF CARDINAL
ARRAY [1..10],[1..20] OF [0..99]
ARRAY [-10..+10] OF BOOLEAN
ARRAY WeekDay OF Color
ARRAY Color OF WeekDay
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

6.5 Record types

A record type is a structure consisting of a fixed number of components of possibly different types. The record type declaration specifies for each component, called field, its type and an identifier which denotes the field. The scope of these field identifiers is the record definition itself, and they are also accessible within field designators (see section 8.1 Operands) referring to components of record variables, and within with statements.

A record type may have several variant sections, in which case the first field of the section is called the tag field. Its value indicates which variant is assumed by the section. Individual variant structures are identified by case labels. These labels are constants of the type indicated by the tag field.

```
$ RecordType = RECORD FieldListSequence END.
$ FieldListSequence = FieldList{";"FieldList}.
$ FieldList = [IdentList":"type |
$     CASE [ident]":"qualident OF variant
$     {"|"variant}[ELSE FieldListSequence] END].
$ variant = [CaseLabelList":"FieldListSequence].
$ CasaLabelList = CaseLabels {","CaseLabels}.
$ CaseLabels = ConstExpression
      [".."ConstExpression].
```

Examples of record types:

```
RECORD day:[1..31];
  month: [1..12];
  year: [0..2000]
END

RECORD
  name,firstname: ARRAY [0..9] OF CHAR;
  age: [0..99];
  salary: REAL
END

RECORD x,y: T0;
  CASE tag0: Color OF
    red:  a:Tr1; b:Tr2 |
    green:  c:Tg1; d:Tg2 |
    blue:  e:Tb1; f:Tb2
  END
  z:T0;
  CASE Tag1: BOOLEAN OF
    TRUE: u,v: INTEGER |
    FALSE: r,s: CARDINAL
  END
END
```

The example above contains two variant sections. The variant of the first section is indicated by the value of the tag field tag0, the one of the second section by the tag field tag1.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

6.6 Set types

A set type defined as SET OF T comprises all sets of values of its base type T. This must be a subrange of the integers between 0 and N-1, or a (subrange of an) enumeration type with at most N values, where N is a small constant determined by implementation, usually the computer's wordsize or a small multiple thereof.

```
$ SetType = SET OF SimpleType.
```

The standard type BITSET is defined as follows, where W is a constant defined by implementation, usually the wordsize of the computer.

```
BITSET = SET OF [0..W-1]
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

6.7 Pointer types

Variables of a pointer type P assume as values pointers to variables of another type T. The pointer type P is said to be bound to T. A pointer value is generated by a call to an allocation procedure in a storage management module.

```
$ PointerType = POINTER TO type.
```

Besides such pointer values, a pointer variable may assume the value NIL, which can be thought as pointing to no variable at all.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

6.8 Procedure types

Variables of a procedure type T may assume as their value a procedure P. The (types of the) formal parameters of P must be the same as those indicated in the formal type list of T. The same holds for the result type in the case of a function procedure.

Restriction: P must not be declared local to another procedure, and neither can it be a standard procedure.

```
$ ProcedureType = PROCEDURE [FormalTypeList].
$ FormalTypeList = "("[[VAR] FormalType
$     {","[VAR] FormalType}]")"[":"qualident].
```

The standard type PROC denotes a parameterless procedure:

```
PROC = PROCEDURE
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

7. Variable declarations

Variable declarations serve to introduce variables and associate them with a unique identifier and a fixed data type and structure. Variables whose identifiers appear in the same list all obtain the same type.

```
$ VariableDeclaration = IdentList":"type.
```

The data type determines the set of values that a variable may assume and the operators that are applicable; it also defines the structure of the variable.

Examples of variable declarations (see section 6. Type declarations):

```
i,j:    CARDINAL
k:      INTEGER
p,q:    BOOLEAN
s:      BITSET
F:      Function
a:      ARRAY Index Of CARDINAL
w:      ARRAY [0..7] OF
             RECORD ch: CHAR;
               count: CARDINAL
             END
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

8. Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands.

8.1 Operands
8.2 Operators

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

8.1 Operands

With the exception of literal constants, i.e. numbers, character strings, and sets (see section 5. Constant declarations), operands are denoted by designators. A designator consists of an identifier referring to the constant, variable, or procedure to be designated. This identifier may possibly be qualified by module identifiers (see section 4. Declarations and scope rules), (see section 11. Modules), and it may be followed by selectors, if the designated object is an element of a structure. If the structure is an array A, then the designator A[E] denotes that component of A whose index is the current value of the expression E. The index type of A must be assignment compatible with the type of E (see section 9.1 Assignments). A designator of the form

A[E1,E2,...,En] stands for A[E1][E2]...[En].

If the structure is a record R, then the designator R.f denotes the record field f of R.

The designator P^ denotes the variable which is referenced by the pointer P.

```
$ designator = qualident
$    {"."ident | "["ExpList"]" | "^"}.
$ ExpList = expression{","expression}.
```

If the designated object is a variable, then the designator refers to the variable's current value. If the object is a function procedure, a designator without parameter list refers to that procedure. If it is followed by a (possibly empty) parameter list, the designator implies an activation of the procedure and stands for the value resulting from its execution, i.e. for the "returned" value. The (types of these) actual parameters must correspond to the formal parameters as specified in the procedure's declaration (see section 10. Procedure declarations).

Examples of designators (see section 7. Variable declarations):

```
k                 (INTEGER)
a[i]              (CARDINAL)
w[3].ch           (CHAR)
t^.key            (CARDINAL)
t^.left^.right    (TreePtr)
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

8.2 Operators

The syntax of expressions specifies operator precedences according to four classes of operators. The operator NOT has the highest prcits execution , i.e. for the "returned" value. The (types of these) actual parameters must correspond to the formal parameters as specified in the procedure's declaration (see operators of the same precedence are executed from left to right.

```
$ expression = SimpleExpression
$     [relation SimpleExpression].
$ relation = "=" | "#" | "<" |
$          "<=" | ">" | ">=" | IN.
$ SimpleExpression = ["+"|"-"] term
$            { AddOperator term}.
$ AddOperator = "+" | "-" | OR.
$ term = factor {MulOperator factor}.
$ MulOperator = "*" | "/" | DIV | MOD | AND.
$ factor = number | string | set |
$      designator[ActualParameters] |
$      "("expression")" | NOT factor.
$ set = [qualident] "{" [ element
$       { "," element }] "}".
$ element = expression [ ".." expression].
$ ActualParameters = "(" [ExpList] ")".
```

The available operators are listed in the following tables. In some instances, several different operations are designated by the same operator symbol. In these cases, the actual operation is identified by the types of the operands.

8.2.1 Arithmetic operators
8.2.2 Logical operators
8.2.3 Set operators
8.2.4 Relations

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

8.2.1 Arithmetic operators

```
  symbol       operation
 --------------------------
    +           addition
    -           subtraction
    *           multiplication
    /           real division
   DIV          integer division
   MOD          modulus
```

These operators (except /) apply to operands of type INTEGER, CARDINAL, or subranges thereof. Both operands must be either of type CARDINAL or a subrange with base type CARDINAL, in which case the result is of the type CARDINAL, or they must both be of type INTEGER or a subrange with base type INTEGER, in which case the result is of type INTEGER.

The operators +, -, and * also apply to operands of type REAL. In this case, both operands must be of type REAL, and the result is then also of type REAL. The division operator / applies to REAL operands only. When used as operators with a single operands only, - denotes sign inversion and + denotes the identity operation. Sign inversion applies to operands of type INTEGER or REAL. The operations DIV and MOD are defined by the following rules:

```
x DIV y is equal to the truncated quotient of x/y
x MOD y is equal to the remainder of the division x DIV y
x = (x DIV y)*y + (x MOD y),  0 <= (x MOD y) < y
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

8.2.2 Logical operators

```
 symbol        operation
------------------------
   OR          logical conjunction
   AND         logical disjunction
   NOT         negation
```

These operators apply to BOOLEAN operands and yield a BOOLEAN result.

```
p OR q    means  "if p then TRUE, otherwise q"
p AND q   means  "if p then q, otherwise FALSE"
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

8.2.3 Set operators

```
  symbol        operation
-------------------------
    +           set union
    -           set difference
    *           set intersection
    /           symmetric set difference
```

These operations apply to operands of any set type and yield a result of the same type.

```
x IN (s1+s2)   iff   (x IN s1) OR (x IN s2)
x IN (s1-s2)   iff   (x IN s1) AND NOT (x IN s2)
x IN (s1*s2)   iff   (x IN s1) AND (x IN s2)
x IN (s1/s2)   iff   (x IN s1) # (x IN s2)
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

8.2.4 Relations

Relations yield a Boolean result. The ordering relations apply to the basic types INTEGER, CARDINAL, BOOLEAN, CHAR, REAL, to enumerations, and to subrange types.

```
 symbol         relation
------------------------
   =            equal
   #            unequal
   <            less
   <=           less or equal  (set inclusion)
   >            greater
   >=           greater or equal (set inclusion)
   IN           contained in   (set membership)
```

The relations = and # also apply to sets and pointers. If applied to sets, <= and >= denote (improper) inclusion. The relation IN denotes set membership. In an expression of the form x IN s, the expression s must be of type SET OF T, where t is (compatible with) the type of x.

Examples of expressions (see section 7. Variable declarations):

```
980                         (CARDINAL)
k DIV 3                     (INTEGER)
NOT p OR q                  (BOOLEAN)
(i+j)*(i-j)                 (CARDINAL)
s-{8,9,13}                  (BITSET)
a[i]+a[j]                   (CARDINAL)
a[i+j]*a[i-j]               (CARDINAL)
(0<=k)&(k<100)              (BOOLEAN)
t^.key=0                    (BOOLEAN)
{13..15}<=s                 (BOOLEAN)
i IN {0,5..8,15}            (BOOLEAN)
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9. Statements

Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that themselves statements. They are the assignment, the procedure call, and the return and exit statements. Structured statements are composed of parts that are themselves statements. These are used to express sequencing, and conditional, selective, and repetitive execution.

```
$ statement = [ assignment | ProcedureCall |
$      IfStatement | CaseStatement |
$      WhileStatement | RepeatStatement |
$      LoopStatement | ForStatement |
$      WithStatement | EXIT |
$      RETURN [ expression ] ].
```

A statement may also be empty, in which case it denotes no action. The empty statement is included in order to relax punctuation rules in statement sequences.

9.1 Assignments
9.2 Procedure calls
9.3 Statement sequences
9.4 If statements
9.5 Case statements
9.6 While statements
9.7 Repeat statements
9.8 For statements
9.9 Loop statements
9.10 With statements
9.11 Return and exit statements

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.1 Assignments

The assignments serves to replace the current value of a variable by a new value indicated by an expression. The assignment operator is written as ":=" and pronounced as "becomes".

```
$ assignment = designator ":=" expression.
```

The designator to the left of the assignment operator denotes a variable. After an assignment is executed, the variable has the value obtained by evaluating the expression. The old value is lost (overwritten). The type of the variable must be assignment compatible with the type of the expression. Operand types are said to be assignment compatible, if either they are compatible or both are INTEGER or CARDINAL or subranges with base types INTEGER or CARDINAL.

A string of length n1 can be assigned to an array variable with n2>n1 elements of type CHAR. In this case, the string value is extended with a null character (0C). A string of length 1 is compatible with the type CHAR.

Examples of assignments:

```
i:=k
p:=i=j
j:=log2(i+j)
F:=log2
s:={2,3,5,7,11,13}
a[i]:=(i+j)*(i-j)
t^.key:=i
w[i+1].ch:="A"
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.2 Procedure calls

A procedure call serves to activate a procedure. The procedure call may contain a list of actual parameters which are substituted in place of their corresponding formal parameters defined in the procedure declaration (see section 10. Procedure declarations). The correspondence is established by the positions of the parameters in the list of actual and formal parameters respectively. There exist two kinds of parameters: variable and value parameters.

In the case of variable parameters, the actual parameter must be a designator denoting a variable. If it designates a component of structured variable, the selector is evaluated when the formal/actual parameter substitution takes place, i.e. before the execution of the procedure. If the parameter is a value parameter, the corresponding actual parameter must be an expression. This expression is evaluated prior to the procedure activation, and the resulting value is assigned to the formal parameter which now constitute a local variable. The types of corresponding actual and formal parameters must be compatible in the case of variable parameters and assignment compatible in the case of value parameters.

```
$ ProcedureCall = designator [ActualParameters].
```

Examples of procedure calls:

```
   Read(i)    (@xref{Procedure Declarations})
   Write(j*2+1,6)
   INC(a[i])
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.3 Statement sequences

Statement sequences denote the sequence of actions specified by the component statements which are separated by semicolons.

```
$ StatementSequence = statement { ";" statement }.
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.4 If statements

```
$ IfStatement = IF expression THEN StatementSequence
$       { ELSIF expression THEN StatementSequence }
$       [ ELSE StatementSequence ] END.
```

The expressions following the symbols IF and ELSIF are of type BOOLEAN. They are evaluated in the sequence of their occurrence, until one yields the value TRUE. Then its associated statement sequence is executed. If an ELSE clause is present, its associated statement sequence is executed if and only if all Boolean expressions yielded the value FALSE.

Example:

```
   IF(ch>="A")&(ch<="Z") THEN ReadIdentifier
   ELSIF(ch>="0")&(ch<="9") THEN ReadNumber
   ELSIF ch='"' THEN ReadString('"')
   ELSIF ch="'" THEN ReadString("'")
   END
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.5 Case statements

Case statements specify the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then the statement sequence is executed whose case label list contains the obtained value. The type of the case expression must be a basic type (except REAL), an enumeration type, or a subrange type, and all labels must be compatible with this type, Case labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any case, the statement sequence following the symbol ELSE is selected.

```
$ CaseStatement = CASE expression OF case
$    { "|" case }[ ELSE StatementSequence ] END.
$ case = [ CaseLabelList ":" StatementSequence ].
```

Example:

```
   CASE i OF
     0: p:=p OR q; x:=x+y|
     1: p:=p OR q; x:=x-y|
     2: p:=p AND q; x:=x*y
   END
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.6 While statements

While statement specify the repeated execution of a statement sequence depending on the value of a Boolean expression. The expression is evaluated before each subsequent execution of the statement sequence. The repetition stops as soon as this evaluation yields the value FALSE.

```
$ WhileStatement = WHILE expression DO
$        StatementSequence END.
```

Examples:

```
   WHILE j>0 DO
     j:=j DIV 2; i:=i+1
   END

   WHILE i#j DO
     IF i>j THEN i:=i-j
     ELSE j:=j-i
     END
   END

   WHILE (t#NIL)&(t^.key#i) DO
     t:=t^.left
   END
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.7 Repeat statements

Repeat statements specify the repeated execution of statement sequence depending on the value of a Boolean expression. The expression is evaluated after each execution of the statement sequence , and the repetition stops as soon as it yields the value TRUE. Hence, the statement sequence is executed at least once.

```
$ RepeatStatement = REPEAT StatementSequence
$       UNTIL expression.
```

Example:

```
   REPEAT k:=i MOD j; i:=j; j:=k
   UNTIL j=0
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.8 For statements

The for statement indicates that a statement sequence is to be repeatedly executed while a progression of values is assigned to a variable. This variable is called the control variable of the for statement. It cannot be a component of a structured variable, it cannot be imported, nor can it be a parameter. Its value should not be changed by the statement sequence.

```
$ ForStatement = FOR ident ":=" expression
$          TO expression [ BY ConstExpression ]
$          DO StatementSequence END.
```

The for statement

```
FOR v:=A TO B BY C DO SS END
```

expresses repeated execution of the statement sequence SS with successively assuming the values A, A+C, A+2C, ..., A+nC, where A+nC is the last term not exceeding B. v is called the control variable, A the starting value, B the limit, and C the increment. A and B must be compatible with v; C must be a constant of type INTEGER or CARDINAL. If no increment is specified, it is assumed to be 1.

Examples:

```
   FOR i:=1 TO 80 DO j:=j+a[i] END
   FOR i:=80 TO 2 BY -1 DO a[i]:=a[i-1] END
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.9 Loop statements

A loop statement specifies the repeated execution of a statement sequence. It is terminated by the execution of any exit statement within that sequence.

```
$ LoopStatement = LOOP StatementSequence END
```

Example:

```
   LOOP
     IF t1^.key>x THEN t2:=t1^.left; p:=TRUE
     ELSE t2:=t1^.right; p:=FALSE
     END;
     IF t2=NIL THEN
        EXIT
     END;
     t1:=t2
   END
```

While, repeat, and for statements can be expressed by loop statements containing a single exit statement. Their use is recommended as they characterize the most frequently occurring situations where termination depends either on a single condition at either the beginning or end of the repeated statement sequence, or on reaching the limit of an arithmetic progression. The loop statement is, however, necessary to express the continuous repetition of cyclic processes, where no termination is specified. It is also useful to express situation exemplified above. Exit statements are contextually, although not syntactically bound to the loop statement which contains them.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.10 With statements

The with statement specifies a record variable and a statement sequence. In these statements the qualification of field identifiers may be omitted, if they are no refer to the variable specified in with clause. If the designator denotes a component of a structured variable, the selector is evaluated once (before the statement sequence). The with statement opens a new scope.

```
$ WithStatement = WITH designator DO
$        StatementSequence END.
```

Example:

```
   WITH t^ DO
     key:=0; left:=NIL; right:=NIL
   END
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

9.11 Return and exit statements

A return statement consists of the symbol RETURN, possibly followed by an expression. It indicates the termination of a procedure (or a module body), and the expression specifies the value returned as result of a function procedure. Its type must be assignment compatible with the result type specified in the procedure heading (see section 10. Procedure declarations).

Function procedures require the presence of a return statement indicating the result value. There may be several, although only one will be executed. In proper procedures, a return statement is implied by the end of the procedure body. An explicit return statement therefore appears as an additional, probably exceptional termination point.

An exit statement consists of the symbol EXIT, and it specifies termination of the enclosing loop statement and continuation with the statement following that loop statement (see section 9.9 Loop statements).

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

10. Procedure declarations

Procedure declarations consists of a procedure heading and a block which is said to be the procedure body. The heading specifiers the procedure identifier and the formal parameters. The block contains declarations and statements. The procedure identifier is repeated at the end of the procedure declaration.

There are two kinds of procedures, namely proper procedures and function procedures. The latter are activated by a function designator as a constituent of an expression, and yield a result that is an operand in the expression. Proper procedures are activated by a procedure call. The function procedure is distinguished in the declaration by indication of the type of its result following the parameter list. Its body must contain a RETURN statement which defines the result of the function procedure.

All constants, variables, types, modules and procedures declared within the block that constitutes the procedure body are local to the procedure. The values of the local variables, including those defined within local module, are undefined upon entry to the procedure. Since procedures may be declared as local objects too, procedure declarations may be nested. Every object is said to be declared at a certain level of nesting. If it is declared local to a procedure at level k, it has itself level k+1. Objects declared in the module that constitutes a compilation unit (see section 14. Compilation units) are defined to be at level 0.

In addition to its formal parameters and local objects, also the objects declared in the environment of the procedure are known and accessible in the procedure (with the exception of those objects that have the same name as objects declared locally).

The use of the procedure identifier in a call within its declaration implies recursive activation of the procedure.

```
$ ProcedureDeclaration = ProcedureHeading ";"
$               block ident.
$ ProcedureHeading = PROCEDURE ident
$          [ FormalParameters ].
$ block = { declaration }
$    [ BEGIN StatementSequence ] END.
$ declaration = CONST { ConstantDeclaration ";" } |
$          TYPE { TypeDeclaration ";"} |
$          VAR { VariableDeclaration ";"} |
$          ProcedureDeclaration ";" |
$          ModuleDeclaration ";".
```

- 10.1 Formal parameters
- 10.2 Standard procedures

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

10.1 Formal parameters

Formal parameters are identifiers which denote actual parameters specified in the procedure call. The correspondence between formal and actual parameters is established when the procedure is called. There are two kinds of parameters, namely value and variable parameters. The kind is indicated in the formal parameter list. Value parameters stand for local variables to which the result of the evaluation of the corresponding actual parameter is assigned as initial value. Variable parameters correspond to actual parameters that are variables, and they stand for these variables. Variable parameters are indicated by the symbol VAR, value parameters by the absence of the symbol VAR. Formal parameters are local to the procedure, i.e. their scope is the program text which constitutes the procedure declaration.

```
$ FormalParameters = "("[FPSection{";"FPSection}]")"
$            [":"qualident].
$ FPSection = [ VAR ] IdentList ":" FormalType.
$ FormalType = [ ARRAY OF ] qualident.
```

The type of each formal parameter is specified in the parameter list. In the case of variable parameters it must be identical with its corresponding actual parameter (see section 9.2 Procedure calls, for exceptions), (see section 12. System-dependent facilities, for exceptions); in the case of value parameters the formal type must be assignment compatible with the actual type (see section 9.1 Assignments). If the parameter is an array, the form

```
ARRAY OF T
```

may be used, where the specification of the actual index bounds is omitted. The parameter is then said to be an open array parameter. T must be the same as the element type of the actual array, and the index range is mapped onto the integers 0 to N-1, where N is the number of elements. The formal array can be accessed elementwise only, or it may occur as actual parameter whose formal parameter is without specified index bounds. A function procedure without parameters has an empty parameter list. It must be called by a function designator whose actual parameter list is empty too.

Restriction: if a formal parameter specifies a procedure type, then the corresponding actual parameter must be either a procedure declared at level 0 or a variable (or parameter) of that procedure type. It cannot be a standard procedure.

Examples of procedure declarations:

```
   PROCEDURE Read(VAR x:CARDINAL);
     VAR i:CARDINAL; ch:CHAR;
   BEGIN i:=0;
     REPEAT ReadChar(ch)
     UNTIL(ch>="0")&(ch<="9");
     REPEAT i:=10*i+(ORD(ch)-ORD("0"));
       ReadChar(ch)
     UNTIL(ch<"0") OR (ch>"9");
     x:=i
   END Read

   PROCEDURE Write(x,n:CARDINAL);
     VAR i:CARDINAL;
         buf:ARRAY [1..10] OF CARDINAL;
   BEGIN i:=0;
     REPEAT INC(i); buf[i]:=x MOD 10; x:=x DIV 10
     UNTIL x=0
     WHILE n>i DO
       WriteChar(" "); DEC(n)
     END;
     REPEAT WriteChar(CHR(buf[i]+ORD("0")));
       DEC(i)
     UNTIL i=0;
   END Write

   PROCEDURE log2(x:CARDINAL):CARDINAL;
     VAR y:CARDINAL; (* assume x>0 *)
   BEGIN x:=x-1; y:=0;
     WHILE x>0 DO
       x:=x DIV 2; y:=y+1
     END;
     RETURN y
   END log2
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

10.2 Standard procedures

Standard procedures are predefined. Some are generic procedures that cannot be explicitly declared, i.e. they apply to classes of operand types or have several possible parameter list forms. Standard procedures are

```
ABS(x)       absolute value;
             result type = argument type.

CAP(ch)      if ch is a lower case letter,
             the corresponding capital letter;
             if ch is a capital letter, the same letter.

CHR(x)       the character with ordinal number x.
             CHR(x)=VAL(CHAR,x)

FLOAT(x)     x of type INTEGER represented
             as a value of type REAL.

HIGH(a)      high index bound of array a.

MAX(T)       the maximum value of type T.

MIN(T)       the minimum value of type T.

ODD(x)       x MOD 2 # 0

ORD(x)       ordinal number (of type CARDINAL)
             of x in the set of values defined
             by type T of x. T is any enumeration
             type, CHAR, INTEGER, or CARDINAL.

SIZE(T)      the number of storage units required
             by a variable of type T.

TRUNC(x)     real number x truncated to its
             integral part (of type INTEGER).

VAL(T,x)     the value with ordinal number x
             and with type T. T is any enumeration
type,
             or CHAR, INTEGER, or CARDINAL.
             VAL(T,ORD(x))=x, if x of type T.

DEC(x)       x:=x-1

DEC(x,n)     x:=x-n

EXCL(s,i)    s:=s-{i}

HALT         terminate program execution

INC(x)       x:=x+1

INC(x,n)     x:=x+n

INCL(s,i)    s:=s+{i}
```

The procedures INC and DEC also apply to operands x of enumeration types and of type CHAR. In these cases they replace x by its (n-th) successor or predecessor.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

11. Modules

A module constitutes a collection of declarations and a sequence of statements. They are enclosed in the bracket MODULE and END. The module heading contains the module identifier, and possibly a number of import lists and an export list. The former specify all identifiers of objects that are declared outside but used within the module and therefore have to be imported. The export list specifies all identifiers of objects declared within the module and used outside. Hence, a module constitutes a wall around its local objects whose transparency is strictly under control of the programmer.

Objects local to a module are said to be at same scope level as the module. They can be considered as being local to the procedure enclosing the module but residing within a more restricted scope. The module identifier is repeated at the end of the declaration.

```
$ ModuleDeclaration = MODULE ident [priority]
$     ";" {import} [export] block ident.
$ priority = "[" ConstExpression "]".
$ export = EXPORT [QUALIFIED] IdentList ";".
$ import = [FROM ident] IMPORT IdentList ";".
```

The statement sequence that constitutes the module body is executed when the procedure to which the module is local is called. If several modules are declared, then these bodies are executed in sequence in which the modules occur. These bodies serve to initialize local variables and must be considered as prefixes to the enclosing procedure's statement part.

If an identifier occurs in the import (export) list, then the denoted object may be used inside (outside) the module as if the module brackets did not exist. If, however, the symbol EXPORT is followed by the symbol QUALIFIED, then the listed identifiers must be prefixed with the module's identifier when used outside the module. This case is called qualified export, and is used when modules are designed which are to be used in coexistence with other modules not known a priori. Qualified export serves to avoid clashes of identical identifiers exported from different modules (and presumably denoting different objects).

A module may feature several import lists which may be prefixed with the symbol FROM and a module identifier. The FROM clause has the effect of unqualifying the imported identifiers. Hence they may be used within the module as if they had been exported in normal, i.e. non-qualified mode.

If a record type is exported, all its field identifiers are exported too. The same holds for the constant identifiers in the case of an enumeration type.

Examples of module declarations:

The following module serves to scan a text and to copy it into an output character sequence. Input is obtained characterwise by a procedure inchr and delivered by a procedure outchr. The characters are given in the ASCII code; control characters are ignored, with the exception of LF (line feed) and FS (file separator). They are both translated into a blank and cause the Boolean variables eoln (end of line) and eof (end of file) to be set respectively. FS is assumed to be preceded by LF.

```
MODULE LineInput;
  IMPORT inchr,outchr;
  EXPORT read, NewLine, NewFile, eoln, eof, lno;
  CONST LF=12C; CR=15C; FS=34C;

  VAR lno:CARDINAL;(*line number *)
    ch: CHAR;   (*last character read *)
    eof,eoln: BOOLEAN;

  PROCEDURE NewFile;
  BEGIN
      If NOT eof THEN
         REPEAT inchr(ch) UNTIL ch=FS;
      END;
      eof:=FALSE; eoln:=FALSE; lno:=0
  END NewFile;

  PROCEDURE NewLine;
  BEGIN
      IF NOT eoln THEN
      REPEAT inchr(ch) UNTIL ch=LF;
        outchr(CR); outchr(LF)
      END;
      eoln:=FALSE;
      INC(lno)
  END NewLine;

  PROCEDURE read(VAR x:CHAR);
  BEGIN (* assume NOT eoln AND NOT eof *)
    LOOP inchr(ch); outchr(ch);
      IF ch>=" " THEN
         x:=ch; EXIT
      ELSIF ch=LF THEN
        x:=" "; eoln:=TRUE; EXIT
      ELSIF ch=FS THEN
        x:=" "; eoln:=TRUE; eof:=TRUE; EXIT
      END
    END
  END read;
BEGIN eof:=TRUE; eoln:=TRUE
END LineInput
```

The next example is a module which operates a disk track reservation table, and protects it from unauthorized access. A function procedure NewTrack yields the number of a free track which is becoming reserved. Tracks can be released by calling procedure ReturnTrack.

```
MODULE TrackReservation;

  EXPORT NewTrack, ReturnTrack;

  CONST ntr=1024; (* no. of tracks *)
    w=16;        (* word size *)
    m=ntr DIV w;

  VAR i: INTEGER;
    free: ARRAY [0..m-1] OF BITSET;

  PROCEDURE NewTrack(): INTEGER;
    (* reserves a new track and yields its index as result,
       if a free track is found, and -1 otherwise *)
    VAR i,j: INTEGER; found: BOOLEAN;
  BEGIN found:=FALSE; i:=m;
    REPEAT DEC(i); j:=w;
      REPEAT DEC(j);
        IF j IN free[i] THEN found:=TRUE; END
      UNTIL found OR (j=0)
    UNTIL found OR (i=0);
    IF found THEN EXCL(free[i],j); RETURN i*w+j
    ELSE RETURN -1
    END
  END NewTrack;

  PROCEDURE ReturnTrack(k: INTEGER);
  BEGIN (* assume 0<=k<ntr *)
    INCL(free[k DIV w],k MOD w)
  END ReturnTrack;

BEGIN (* mark all tracks free *)
  FOR i:=0 TO m-1 DO free[i]:={0..w-1} END
END TrackReservation
```

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

12. System-dependent facilities

Modula-2 offers certain facilities that are necessary to program low-level operations referring directly to objects particular of a given computer and/or implementation. These include for example facilities for accessing devices that are controlled by the computer, and facilities to break the data type compatibility rules otherwise imposed by the language definition. Such facilities are to be used with utmost care, and it is strongly recommended to restrict their use to specific modules(called low-level modules). Most of them appear in the form of data types and procedures imported from the standard module SYSTEM. A low-level module is therefore explicitly characterized by the identifier SYSTEM appearing in its import list.

Note: Because the objects imported from the module SYSTEM obey special rules, this module must be known to the compiler. It is therefore called a pseudo module and need not be supplied as a separate definition module (see section 14. Compilation units).

The facilities exported from the module SYSTEM are specified by individual implementations. Normally, the types WORD and ADDRESS, and the procedures ADR, TSIZE, NEWPROCESS, TRANSFER, are among them (see section 13. Processes).

The type WORD represents an individually accessible storage unit. No operation except assignment is defined on this type. However, if a formal parameter of a procedure is of type WORD, the corresponding actual parameter may be of any type that uses one storage word in the given implementation. If a formal parameter has the type ARRAY OF WORD, its corresponding actual parameter may be of any type; in particular it may be a record type to be interpreted as an array of words.

The type ADDRESS is defined as

```
ADDRESS = POINTER TO WORD
```

It is compatible with all pointer types, and also with the type CARDINAL, Therefore, all operators for integer arithmetic apply to operands of this type. Hence, the type ADDRESS can be used to perform address computations and to export the results as pointers. If a formal parameter is of type ADDRESS, the corresponding actual parameter may be of any pointer type, even if the formal parameter is a VAR parameter. The following example of a primitive storage allocator demonstrates a typical usage of the type ADDRESS.

```
MODULE Storage;
  FROM SYSTEM IMPORT ADDRESS;
  EXPORT Allocate;

  VAR lastused: ADDRESS;

  PROCEDURE Allocate(VAR a: ADDRESS; n: CARDINAL);
  BEGIN a:= lastused; lastused:=lastused+n
  END Allocate;

BEGIN lastused:=0
END Storage
```

The function ADR(x) denotes the storage address of variable x and is of type ADDRESS. TSIZE(T) is the number of storage units assigned to any variable of type T. TSIZE is of an arithmetic type depending on implementation.

Besides those exported from the pseudo-module SYSTEM, there are two other facilities whose characteristics are system-dependent. The first is the possibility to use a type identifier T as a name denoting the type transfer function from the type of the operand to the type T. Evidently, such functions are data representation dependent, and they involve no explicit conversion instructions.

The second, non-standard facility may be provided in variable declarations. It allows to specify the absolute address of a variable and to override the allocation scheme of a compiler. This facility is intended for access to storage locations with specific purpose and fixed address, such as e.g. device registers on computers with "memory-mapped I/O". This address is specified as a constant integer expression enclosed in brackets immediately following the identifier in the variable declaration. The choice of an appropriate data type is left to the programmer. For examples, refer to 13.2.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

13. Processes

Modula-2 is designed primarily for implementation on a conventional single-processor computer. For multiprogramming it offers only some basic facilities which allow the specification of quasi-concurrent processes and of genuine concurrency for peripheral devices. The word process is here used with the meaning of coroutine. Coroutines are processes that are executed by a (single) processor one at a time.

- 13.1 Creating a process and transfer of control
- 13.2 Device processes and interrupts

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

13.1 Creating a process and transfer of control

A new process is created by a call to

```
PROCEDURE NEWPROCESS(P: PROC; A: ADDRESS;
                    n: CARDINAL; VAR p1: ADDRESS)

P   denotes the procedure which constitutes the process,
A   is a base address of the process' workspace,
n   is the size of this workspace,
p1  is the result parameter.
```

A new process with P as program and A as workspace of size n is assigned to p1. This process is allocated, but not activated. P must be a parameterless procedure declared at level 0.

A transfer of control between two processes is specified by a call to

```
PROCEDURE TRANSFER(VAR p1,p2: ADDRESS)
```

This call suspends the current process, assigns it to p1, and resumes the process designated by p2. Evidently, p2 must have been assigned a process by an earlier call to either NEWPROCESS or TRANSFER. Both procedures must be imported from the module SYSTEM. A program terminates, when control reaches the end of a procedure which is the body of a process.

Note: assignment to p1 occurs after identification of the new process p2; hence, the actual parameters may be identical.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

13.2 Device processes and interrupts

If a process contains an operation of a peripheral device, then the processor may be transferred to another process after the operation of the device has been initiated, thereby leading to a concurrent execution of that other process with the device process. Usually, termination of the device's operation is signalled by an interrupt of the main processor. In terms of Modula-2, an interrupt is a transfer operation. This interrupt transfer is (in Modula-2 implemented on the PDP-11) preprogrammed by and combined with the transfer after device initiation. This combination is expressed by a call to

```
PROCEDURE IOTRANSFER(VAR p1,p2: ADDRESS; va: CARDINAL)
```

In analogy to TRANSFER, this call suspends the calling device process, assigns it to p1, resumes (transfers to) the suspended process p2, and in addition causes the interrupt transfer occurring upon device completion to assign the interrupted process to p2 and to resume the device process p1. va is the interrupt vector address assigned to the device. The procedure IOTRANSFER must be imported from the module SYSTEM, and should be considered as PDP-11 implementation-specific.

It is necessary that interrupts can be postponed (disabled) at certain times, e.g. when variables common to the cooperating processes are accessed, or when other, possibly time-critical operations have priority. Therefore, every module is given a certain priority level, and every device capable of interrupting is given a priority level. Execution of a program can be interrupted, if and only if the interrupting device has a priority that is greater than the priority level of the module containing the statement currently being executed. Whereas the device priority is defined by the hardware, the priority level of each module is specified by its heading. If an explicit specification is absent, the level in any procedure is that of the calling program. IOTRANSFER must be used within modules with a specified priority only.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

14. Compilation units

A text which is accepted by the compiler as a unit is called a compilation unit. There are three kinds of compilation units: main modules, definition modules, and implementation modules. A main module constitutes a main program and consists of a so-called program module. In particular, it has no export list. Imported objects are defined in other (separately compiled) program parts which themselves are subdivided into two units, called definition module and implementation module.

The definition module specifies the names and properties of objects that are relevant to clients, i.e. other modules which imported from it. The implementation module contains local objects and statements that need not be known to a client. In particular the definition module contains constant, type, and variable declarations, and specifications of procedure headings. The corresponding implementation module contains the complete procedure declarations, and possibly further declarations of objects not exported. Definition and implementation modules exist in pairs. Both may contain import lists, and all objects declared in the definition module are available in corresponding implementation module without explicit import.

```
$ DefinitionModule = DEFINITION MODULE ident ";"
$         {import} {definition} END ident ".".
$ definition = CONST {ConstantDeclaration ";"} |
$              TYPE {ident ["="type] ";"} |
$              VAR {VariableDeclaration ";"} |
$              ProcedureHeading ";".
$ ProgramModule = MODULE ident [priority] ";"
$          {import} block ident ".".
$ CompilationUnit = DefinitionModule |
$          [IMPLEMENTATION] ProgramModule.
```

The definition module evidently represents the interface between the implementation module on one side and its clients on the other side. The definition module contains those declarations which are relevant to the client modules, and presumably no other ones. Hence, the definition module acts as the implementation module's (extended) export list, and all its declared objects are exported.

Definition modules imply the use of qualified export. Type definitions may consist of the full specification of the type (in this case its export is said to be transparent), or they may consist of the type identifier only. In this case the full specification must appear in the corresponding implementation module , and its export is said to be opaque. The type is known in the importing client modules by its name only , and all its properties are hidden. Therefore, procedures operating on operands of this type, and in particular operating on its components, must be defined in the same implementation module which hides the type's properties. Opaque export is restricted to pointers and to subranges of standard types. Assignment and test for equality are applicable to all opaque types.

As in local modules, the body of an implementation module acts as an initialization facility for its local objects. Before its execution, the imported modules are initialized in the order in which they are listed. If circular references occur among modules, their order of initialization is not defined.

[Top] [Contents] [Index] [?]

Table of Contents

- 1. Introduction
- 2. Syntax
- 3. Vocabulary and representation
- 4. Declarations and scope rules
- 5. Constant declarations
- 6. Type declarations
  - 6.1 Basic types
  - 6.2 Enumerations
  - 6.3 Subrange types
  - 6.4 Array types
  - 6.5 Record types
  - 6.6 Set types
  - 6.7 Pointer types
  - 6.8 Procedure types
- 7. Variable declarations
- 8. Expressions
  - 8.1 Operands
  - 8.2 Operators
    - 8.2.1 Arithmetic operators
    - 8.2.2 Logical operators
    - 8.2.3 Set operators
    - 8.2.4 Relations
- 9. Statements
  - 9.1 Assignments
  - 9.2 Procedure calls
  - 9.3 Statement sequences
  - 9.4 If statements
  - 9.5 Case statements
  - 9.6 While statements
  - 9.7 Repeat statements
  - 9.8 For statements
  - 9.9 Loop statements
  - 9.10 With statements
  - 9.11 Return and exit statements
- 10. Procedure declarations
  - 10.1 Formal parameters
  - 10.2 Standard procedures
- 11. Modules
- 12. System-dependent facilities
- 13. Processes
  - 13.1 Creating a process and transfer of control
  - 13.2 Device processes and interrupts
- 14. Compilation units

[Top] [Contents] [Index] [?]

Short Table of Contents

- 1. Introduction
- 2. Syntax
- 3. Vocabulary and representation
- 4. Declarations and scope rules
- 5. Constant declarations
- 6. Type declarations
- 7. Variable declarations
- 8. Expressions
- 9. Statements
- 10. Procedure declarations
- 11. Modules
- 12. System-dependent facilities
- 13. Processes
- 14. Compilation units

[Top] [Contents] [Index] [?]

About this document

This document was generated using *texi2html*

The buttons in the navigation panels have the following meaning:

Button	Name	Go to	From 1.2.3 go to
[<]	Back	previous section in reading order	1.2.2
[>]	Forward	next section in reading order	1.2.4
[<<]	FastBack	previous or up-and-previous section	1.1
[Up]	Up	up section	1.2
[>>]	FastForward	next or up-and-next section	1.3
[Top]	Top	cover (top) of document	
[Contents]	Contents	table of contents	
[Index]	Index	concept index	
[?]	About	this page	

where the **Example** assumes that the current position is at **Subsubsection One-Two-Three** of a document of the following structure:

- 1. Section One
    - 1.1 Subsection One-One
        - ...
    - 1.2 Subsection One-Two
        - 1.2.1 Subsubsection One-Two-One
        - 1.2.2 Subsubsection One-Two-Two
        - 1.2.3 Subsubsection One-Two-Three **<== Current Position**
        - 1.2.4 Subsubsection One-Two-Four
    - 1.3 Subsection One-Three
        - ...
    - 1.4 Subsection One-Four

This document was generated by *Anthony Fok* on *September, 26 2002* using *texi2html*