ETH

Eidgenössische Technische Hochschule
Zürich
Institut für Informatik

Niklaus Wirth

MODULA-2

March 1980 36

MODULA-2

N. Wirth

Abstract

Modula-2 is a general purpose programming language primarily designed for systems implementation. This report constitutes its definition in a concise, although informal style. It also describes the use of an implementation for the PDP-11 computer.

Institut fur Informatik
ETH
CH-8092 Zurich

March 1980

*

Contents

1. Introduction 2
2. Notation for syntactic description 3
3. Vocabulary and representation 4
4. Declarations and scope rules 6
5. Constant declarations 7
6. Type declarations 7
   1. Basic types 7
   2. Enumerations 8
   3. Subrange types 8
   4. Array types 9
   5. Record types 9
   6. Set types 10
   7. Pointer types 11
   8. Procedure types 11
7. Variable declarations 11
8. Expressions 12
   1. Operands 12
   2. Operators 13
9. Statements 15
   1. Assignments 15
   2. Procedure calls 16
   3. Statement sequences 16
   4. If statements 17
   5. Case statements 17
   6. While statements 17
   7. Repeat statements 18
   8. For statements 18
   9. Loop statements 19
   10. With statements 19
   11. Return and exit statements 19
10. Procedure declarations 20
   1. Formal parameters 21
   2. Standard procedures 22
11. Modules 23
12. System-dependent facilities 26
13. Processes 27
   1. Creating a process and transfer of control 27
   2. Device processes and interrupts 28
14. Compilation units 29
15. Implementation and use of Modula-2 30
16. Standard utility modules 32
   1. Input and output 32
   2. Streams 33
   3. Files 34
   4. Terminal input and output 37
   5. Storage management 37
   6. The Loader 38
   7. Process Scheduler 38
17. Syntax summary and index 42

1. Introduction

Modula-2 grew out of a practical need for a general, efficiently implementable systems programming language for minicomputers. Its ancestors are PASCAL [1] and MODULA [2]. From the latter it has 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 Modula-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.

Chapter 15 of this report describes the use of an implementation of Modula-2 on the PDP-11 computer. This programming system consists of a multi-pass compiler, a linker, and a loader. The compiler allows to translate individual modules which can be combined by the linker. The resulting linker output is loaded for execution by the loader. Compatibility checks (e.g. for type consistency) between the separately compiled modules are performed by the compiler.

Chapter 16 is a collection of a few widely usable utility modules, particularly for input and output handling. Listed are the respective definition modules augmented by explanations of the meaning of the exported procedures.

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.

I should like to acknowledge the inspiring influence which the language MESA [3] has exerted on the design of Modula-2. An extended opportunity to use the sophisticated MESA system has taught me how to tackle problems on many occasions, and on a few that it is wiser to avoid them altogether. Acknowledgment is also due to the implementors of Modula-2, L. Geissmann, A. Gorrengourt, Ch. Jacobi, and S.E. Knudsen, who have carefully proofread the manuscript, and whose invaluable feedback has helped to keep the language designer's fancies on firm ground.

References:

1. N.Wirth. The programming language PASCAL. Acta Informatica 1, 35-63 (1971).

2. N.Wirth. Modula: a language for modular multiprogramming. Software - Practice and Experience, 7, 3-35 (1977).

3. J.G.Mitchell, W. Maybury, R.Sweet. Mesa Language Manual. Xerox PARC, CSL-78-1, (1978).

2. Notation for syntactic description

To describe the syntax, an Extended Backus-Naur Formalism called EBNF is used. Syntactic entities (non-terminal symbols) are denoted by English words expressing their intuitive meaning. Symbols of the language (terminal symbols) either are words written in capital letters, or they are strings enclosed in quote marks. Each syntactic rule (production) has the form

```
S = E.
```

where S is a syntactic entity and E is a syntax expression denoting the set of sentential forms (sequences of symbols) for which S stands. An expression E has the form

```
T1 | T2 | ... | Tn        (n > 0)
```

where the Ti are the terms of E. Each Ti stands for a set of sentential forms, and E denotes their union. Each term T has the form

```
F1 F2 ... Fn        (n > 0)
```

where the Fi are the factors of T. Each Fi stands for a set of

sentential forms, and T denotes their concatenation. The concatenation of two sets of sentences is the set of sentences consisting of all possible concatenations of a sentence from the first factor followed by a sentence from the second factor. Each factor F is either a (terminal or non-terminal) symbol, or it is of the form [E] denoting the union of the set E and the empty sentence, or { E } denoting the union of the empty sequence and E, EE, EEE, ... Parentheses may be used for grouping terms and factors.

EBNF is capable of describing its own syntax. We use it here as an example:

```
syntax     = {production}.
production = NTSym "=" expression "." .
expression = term {"|" term}.
term       = factor {factor}.
factor     = TSym | NTSym | "(" expression ")"  |
             "[" expression "]" | "{" expression "}" .
```

3. Vocabulary and representation

A language is an infinite set of sentences (programs), namely the sentences well formed according to its syntax. Each sentence (program) is a finite sequence of symbols from a finite vocabulary. The vocabulary of Modula-2 consists of identifiers, numbers, strings, operators, and delimiters. They are called lexical symbols or tokens, and in turn are composed of sequences of characters. (Note the distinction between symbols and characters.) 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:

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 the given (octal) ordinal number (and is of type CHAR, see 6.1). 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 = "Ø"|"1"|"2"|"3"|"4"|"5"|"6"|"7".
```

Examples:

1980 3764B 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 single-character string is of type CHAR, a string consisting of n>1 chararcters is of type (see 6.4)

```
ARRAY [Ø..n-1] 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.

+	=	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. Blanks must not occur within symbols (except in strings). Blanks and line breaks are ignored unless they are essential to separate two consecutive symbols.

6. 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.

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 6.7) 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 Ch. 14), it extends to all those units which import M1.

4. Field identifiers of a record declaration (see 6.5) 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 Ch. 11) in which the qualified identifier is defined. The prefix and the identifier are separated by a dot.

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

The following are standard identifiers:

ABS	(10.2)	INCL	(10.2)
ADR	(10.2)	INTEGER	(6.1)
ASH	(10.2)	HALT	(10.2)
BITSET	(6.6)	HIGH	(10.2)
BOOLEAN	(6.1)	NEW	(10.2)
CAP	(10.2)	NIL	(6.7)
CARDINAL	(6.1)	ODD	(10.2)
CHAR	(6.1)	PROC	(6.8)
DEC	(10.2)	REAL	(6.1)
DISPOSE	(10.2)	ROUND	(10.2)
EXCL	(10.2)	SIZE	(10.2)
FALSE	(6.1)	TRUE	(6.1)
FLOAT	(10.2)	TSIZE	(10.2)
INC	(10.2)		

5. Constant declarations

A constant declaration associates an identifier with a constant value.

```
$   ConstantDeclaration = ident "=" ConstExpression.
$   ConstExpression = SimpleConstExpr [relation SimpleConstExpr].
$   relation = "=" | "#" | "<>" | "<" | "<=" | ">" | ">=" | IN .
$   SimpleConstExpr = ["+"|"-"] ConstTerm {AddOperator ConstTerm}.
$   AddOperator = "+" | "-" | OR .
$   ConstTerm = ConstFactor {MulOperator ConstFactor}.
$   MulOperator = "*" | "/" | DIV | MOD | AND | "&" .
$   ConstFactor = qualident | number | string | set |
$       "(" ConstExpression ")" | NOT ConstFactor.
$   set = [qualident] "{" [element {"," element}] "}".
$   element = ConstExpression [".." ConstExpression].
```

The meaning of operators is explained in Chapter 8. The identifier preceding the left brace of a set specifies the type of the set. If it is omitted, the standard type BITSET is assumed (see 6.6).

Examples of constant declarations:

```
N     = 100
limit = 2*N -1
all   = {0..WordSize-1}
```

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

The following basic types are predeclared and denoted by standard

identifiers:

INTEGER	A variable of type INTEGER assumes as values the integers between MinInt and MaxInt.
CARDINAL	A variable of type CARDINAL assumes as values the integers between Ø and MaxCard.
BOOLEAN	A variable of this type assumes the truth values TRUE or FALSE. These are the only values of this type which is predeclared by the enumeration BOOLEAN = (FALSE, TRUE).
CHAR	A variable of this type assumes as values characters of the ASCII (ISO) character set.
REAL	A variable of this type assumes as values real numbers.

For implementations on 16-bit Computers, MinInt = -32768, MaxInt = 32767, and MaxCard = 65535.

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.

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

Examples of enumerations:

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

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 = "[" 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. 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 TØ, if either T1 = TØ, or T1 is a subrange of TØ, or TØ is a subrange of T1, or if

T0 and T1 are both subranges of the same (base) type.

Examples of subrange types:

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

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 so-called 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
```

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 so-called field identifiers is the record definition itself, and they are also accessible within field designators (see 8.1) refering to components of record variables.

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 so-called 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.
$   CaseLabelList = 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.

```
RECORD
   CASE BOOLEAN OF
      TRUE:  i: INTEGER  (*signed*) |
      FALSE: r: CARDINAL (*unsigned*)
   END
END
```

This example shows a record structure without fixed part and with a variant part with missing tag field. In this case the actual variant assumed by the variable cannot be derived from the variable's value itself. This situation is sometimes appropriate, but must be programmed with utmost care.

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

WordSize-1, or a (subrange of an) enumeration type with at most WordSize values.

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

The standard type BITSET is defined as

```
BITSET = SET OF [Ø .. WordSize-1]
```

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 the standard procedure NEW (see 10.2).

```
$   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.

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 correspond to those indicated in the formal type list of T. 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
```

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 (refer to examples in Ch. 6):

```
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
t:   TreePtr
```

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

With the exception of literal constants, i.e. numbers, character strings, and sets (see Ch. 5), operands are denoted by so-called 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 Ch. 4 and 11), 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 9.1). A designator of the form A[E1, E2, ... , En] stands as an abbreviation 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 so-called "returned" value. The (types of these) actual parameters must correspond to the formal parameters as specified in the procedure's declaration (see Ch. 10).

Examples of designators (see examples in Ch. 7):

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

8.2. Operators

The syntax of expressions specifies operator precedences according to four classes of operators. The operator NOT has the highest precedence, followed by the so-called multiplying operators, then the so-called adding operators, and finally, with the lowest precedence, the relational operators. Sequences of operators of the same precedence are executed from left to right.

```
$   expression = SimpleExpression [relation SimpleExpression].
$   SimpleExpression = ["+"|"-"] term {AddOperator term}.
$   term = factor {MulOperator factor}.
$   factor = number | string | set | designator [ActualParameters] |
$       "(" expression ")" | NOT factor.
$   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

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 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 operand 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)

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"
```

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)
```

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 (refer to examples in Ch. 7):

```
        1980                        (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)
```

9. Statements

Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that are 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

The assignment 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. Operands are said to be assignment compatible, if either they are compatible, or of type INTEGER or CARDINAL or subranges with base types INTEGER or CARDINAL.

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

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"
```

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 Ch. 10). The correspondence is established by the positions of the parameters in the lists 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 a 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 constitues 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)                          (see Ch. 10)
Write(j*2+1,6)
INC(a[i])
```

9.3. Statement sequences

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

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

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("'")
ELSE SpecialCharacter
END
```

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 that type. No value must occur more than once as a case label. If the value 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
```

9.6. While statements

While statements 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
```

9.7. Repeat statements

Repeat statements specify the repeated execution of a 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
```

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 v 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 assignment 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
```

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 situations exemplified above. Exit statements are contextually, although not syntactically bound to the loop statement which contains them.

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 to refer to the variable specified in the with clause. If the designator denotes a component of a structured variable, the selector is evaluated once (before the statement sequence).

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

Example:

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

9.11. Return and exit statements

A return statement consists of the symbol RETURN, possibly followed by an expression E. It indicates the termination of a procedure (or

a module body). E specifies the value returned as result of a function procedure, and its type must be assignment compatible with the result type specified in the procedure heading (see Ch. 10).

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 a loop statement and continuation with the statement following the loop statement (see 9.9).

10. Procedure declarations

Procedure declarations consist of a procedure heading and a block which is said to be the procedure body. The heading specifies 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 call 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 local variables, including those defined within a local module, are not defined 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 Ch. 14) 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

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 compatible with its corresponding actual parameter (see 9.2), in the case of value parameters the formal type must be assignment compatible with the actual type (see 9.1). If the parameter is an array, the form

```
ARRAY OF T
```

may be used, where the specification of the actual index bounds is omitted. T must be compatible with 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 + (CARDINAL(ch)-CARDINAL("0"));
    ReadChar(ch)
```

```
  UNTIL (ch < "Ø") OR (ch > "9");
  x := i
END Read

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

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

10.2. Standard procedures

Standard procedures are predefined. Some are so-called 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
ADR(v)	address of variable v (of type CARDINAL)
ASH(x,n)	x * (2**n)
CAP(ch)	if ch is a lower case letter, the corresponding capital letter; if ch is a capital letter, the same letter
FLOAT(i)	integer i represented as a REAL value
HIGH(a)	high index bound of array a
ODD(x)	x MOD 2 # Ø
ROUND(x)	x rounded to the nearest integer; of type INTEGER
TSIZE(T)	size of variables of type T
SIZE(x)	size of variable x
DEC(x)	x := x-1
DEC(x,n)	x := x-n
EXCL(s,i)	s := s - {i}
INC(x)	x := x+1
INC(x,n)	x := x+n
INCL(s,i)	s := s + {i}
HALT	terminate program execution

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.

NEW and DISPOSE are translated into calls to ALLOCATE and DEALLOCATE, procedures that are either explicitly programmed or imported from another module.

```
NEW(p)                    =  ALLOCATE(p,TSIZE(T))
DISPOSE(p)                =  DEALLOCATE(p,TSIZE(T))
NEW(p,t1,t2, ... )        =  ALLOCATE(p,TSIZE(T,t1,t2,...))
DISPOSE(p,t1,t2, ... )    =  DEALLOCATE(p,TSIZE(T,t1,t2,...))
```

where p is declared as "VAR p: POINTER TO T".

These procedures must be compatible with the type

```
PROCEDURE (VAR ADDRESS, CARDINAL)
```

11. Modules

A module constitutes a collection of declarations and a sequence of statements. They are enclosed in the brackets MODULE and END. The module heading contains the module identifier, and possibly a number of so-called import-lists and a so-called 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 the 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.

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

The module identifier is repeated at the end of the declaration.

The statement sequence that constitutes the module body (block) is executed when the procedure to which the module is local is called. If several modules are declared, then these bodies are executed in the 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 without appearing in the export list. The same holds for the constant identifiers in the case of an enumeration type.

Standard identifiers are always imported automatically. As a consequence, standard identifiers can be redeclared in procedures only, but not in modules, including the compilation unit (see Ch. 14).

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: CARDINAL;
        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: CARDINAL; 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: CARDINAL);
    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
```

12. System-dependent facilities

Modula-2 offers certain facilities that are necessary to program so-called 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 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 Ch. 14).

The module SYSTEM exports the types WORD, ADDRESS, PROCESS, and the procedures NEWPROCESS, TRANSFER, IOTRANSFER, and possibly others depending on the operating system being used (see Ch. 13).

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. This includes the types CARDINAL, INTEGER, BITSET and all pointers. 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. 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;  INC(lastused, n)
  END Allocate;

BEGIN lastused := 0
END Storage
```

Besides those exported from the pseudo-module SYSTEM, there are two other, generally available 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 facility is used 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 primarily used to access storage locations with specific purpose and fixed address, such as e.g. device registers on a PDP-11 computer. 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.

Examples (see also 13.2):

```
VAR TWS [777564B]:  BITSET;  (*typewriter status*)
    TWB [777566B]:  CHAR;    (*typewriter buffer*)
```

This facility should be considered as specific for the PDP-11 implementation.

13. Processes

Modula-2 is designed primarily for implementation on a conventional single-processor computer. For multiprogramming it offers only some very 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

A new process is created by a call to

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

P denotes the procedure which constitutes the process,
A is the base address of the process' workspace,
n is the size of this workspace.

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 Ø.

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

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

The effect of this call is to suspend the current process, assign it to p1, and to resume the process designated by p2. (Note: assignment

to p1 occurs after identification of the new process p2; hence, the actual parameters may be identical). Evidently, p2 must have been assigned a process by an earlier call to either NEWPROCESS or TRANSFER. Both procedures, as well as the type PROCESS, 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.

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 so-called 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: PROCESS; 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 so-called 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 (Ø..7) is specified by its heading. If an explicit specification is absent, the level in any procedure is that of the calling program.

The following example shows a module with a process that acts as a driver for a typewriter. The module contains a buffer B for N characters.

```
MODULE Typewriter [4];  (*typewriter interrupt priority = 4*)
  FROM SYSTEM IMPORT
    WORD, PROCESS, NEWPROCESS, TRANSFER, IOTRANSFER, LISTEN;
  EXPORT typeout;

  CONST N = 32;
```

```
VAR n: INTEGER;      (*no. of chars in buffer*)
    in, out: [1..N];
    B: ARRAY [1..N] OF CHAR;
    PRO: PROCESS; (*producer*)
    CON: PROCESS; (*consumer = typewriter driver*)
    wsp: ARRAY [0..20] OF WORD;
    TWS [177564B]: BITSET;     (*status register*)
    TWB [177566B]: CHAR;       (*buffer register*)

PROCEDURE typeout(ch: CHAR);
BEGIN INC(n);
  WHILE n > N DO LISTEN END ;
  B[in] := ch; in := in MOD N + 1;
  IF n = 0 THEN TRANSFER(PRO,CON) END
END typeout;

PROCEDURE driver;
BEGIN
  LOOP DEC(n);
    IF n < 0 THEN TRANSFER(CON,PRO) END ;
    TWB := B[out]; out := out MOD N + 1;
    TWS := {6}; IOTRANSFER(CON,PRO,64B); TWS := {}
  END
END driver;

BEGIN n := 0; in := 1; out := 1;
  NEWPROCESS(driver, ADR(wsp), SIZE(wsp), CON);
  TRANSFER(PRO,CON)
END Typewriter
```

LISTEN is a procedure that lowers the processor's priority level so that pending interrupts may be accepted.

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 consist 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 import from it. The implementation module contains local objects and statements that need not be known to a client. In particular the definition module contains the export list, 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 the corresponding implementation module without explicit import.

```
$   DefinitionModule = DEFINITION MODULE ident ";" {import}
$       [export] {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 definition/implementation module pair on one side and its clients on the other side.

Definition modules require 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 paricular 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.

Examples of compilation units follow in chapter 16.

15. Implementation and use of Modula-2

This chapter describes the use of Modula-2 in terms of its implementation for the PDP-11 computer developed at ETH. This system consits of a compiler, a linker, a debugger, and a basic executive including a loader. The units of data which are accepted and generated by these system components are files as defined by the available file handler.

The basic executive is a resident program named MODULA. It accepts a file name (default extensions is LOD) and, using its loader, loads the named file. This file must have been generated by the linker and carry the file name extension LOD. After loading, control is transferred to that program, and upon termination of the program, control returns to the basic executive. It indicates its readyness to accept the file name of the next program by displaying an asterisk ("*").

The compiler is itself a Modula-2 program and carries the file name COMP. After loading, it requests the file name of the compilation unit to be compiled. The assumed default extension is MOD. (The extension MOD is recommended for program modules, DEF for definition modules). The compiler generates a listing with the same file name as the source and with extension LST. In the case of successful compilation, it also generates a linkable object code file with extension LNK in the case of a program module, or a symbol table with file name extension SYM in the case of a definition module.

Additionally, a file to be used by the debugger is generated and carries the extension REF (reference). Besides the source text, the compiler requires as input the symbol table files (SYM) of all modules specified in the import list of the compiled source program.

After compilation, the object code must be linked with the object code of imported modules by the linker, which is itself a Modula-2 program named LINK. It requests the file name of the main program (called the master file) which is to be linked (and assumes the default extension LNK). It requires as further inputs the code files of all modules to be linked, which must have the name extension LNK, and it generates the loadable file with the name of the main program module and extension LOD.

The linking process results in a so-called core-image of the code, and it determines the locations into which the loader will place this code. The allocation strategy of the linker is based on a stack; the linker accepts a specification of an environment, i.e. of a code file (called the base file) assumed to be already present (previously loaded). The default environment is the basic executive. This scheme allows for so-called overlays, i.e. the loading of program parts onto locations occupied by other, presently unused, program parts, because the loader can be called directly from programs themselves (see 16.6).

Compilation units of a program are compiled separately, but not independently. The dependence is established by the symbol table files, which allow the compiler to perform the necessary type consistency checks. This scheme has the consequence that the definition parts of imported modules must have been compiled prior to compilation of the importing module. This chronological ordering applies, however, to the respective definition modules only. It is particularly noteworthy, that an importing module must be recompiled whenever an imported definition module has been recompiled, but that recompilation of an implementation module does not have any such consequence, as long as its definition module is not altered (recompiled). In order to aid in maintaining consistency among the linked versions of compilation units, the compiler supplies each code and symbol table file with a date/time stamp, and the linker refuses to link inconsistently generated files.

If a program terminates with an error, the state of the computation can be inspected with the aid of the debugger. It is itself a Modula-2 program called DEBUG. It requests the name of the terminated program and requires the corresponding files with extensions REF and LST. The state of computation is retained by a file (core dump) generated upon error termination.

Most Modula-2 programs will make use of previously compiled utility modules by importing them. Hence, an implementation will not only consist of the basic executive, the compiler, etc., but also of a collection of frequently used, utilities. They typically include procedures for creating, reading, writing, and closing files, for conventional conversions in input and output, for operating peripheral devices, and for other purposes. Some such utility modules are described in the following chapter. It must be kept in mind, however, that these modules, although essential to most programs, are not part of the language definition. In principle, any

implementation, in fact even every programmer, may supply its (his) own versions of such modules, thereby creating its (his) own environment of utilities that in most conventional computers belong to its fixed operating system, or are even included in the language definition. This additional flexibility is a direct consequence of the module concept in conjunction with the facility for separate compilation.

16. Standard utility modules

This chapter presents a collection of modules that are widely useful and - for the sake of program transferability - suggested to be available in every implementation of Modula-2. They are, however, not part of the language. The collection contains high-level procedures for input and output, and presents a concept for quasi-concurrent processes with signals for their synchronization.

16.1. Input and output

The procedures contained in the utility module InOut are used to perform input from and output to standard input/output devices or to sequential files. In particular, they perform the necessary data representation conversions between the standard data types CARDINAL and INTEGER, and sequences of characters. The input procedures read from the system's standard input device (keyboard). The output procedures write on the system's standard output device (display). However, both input and output can instead be assigned to files by calling OpenIO. This procedure requests the specification of the names of the files (to be read and/or written). Input and output is reset to the default devices (keyboard, display) by calling CloseIO, which must be called in order to close the associated files.

Several output procedures contain a parameter n specifying the number of characters to be written. If n is greater than the number m of characters actually needed to represent the value x, then n-m blanks precede that value. If n is less than m, the specified n is ignored.

```
DEFINITION MODULE InOut;     (*N.Wirth 6.2.80*)
  FROM SYSTEM IMPORT WORD;
  EXPORT QUALIFIED
    StrLeng, String, OpenIO, CloseIO, Read, ReadInt,
    Write, WriteInt, WriteCard, WriteOct, WriteHex,
    Writeln, WriteString, ShowString;

  CONST StrLeng = 80;
  TYPE String = ARRAY [0..StrLeng-1] OF CHAR;
  PROCEDURE OpenIO;
  PROCEDURE CloseIO;

  PROCEDURE Read(VAR ch: CHAR);
```

```
  PROCEDURE ReadInt(VAR x: INTEGER;
                    VAR nextChar: CHAR; VAR IsNum: BOOLEAN);
    (*skip blanks and control characters; if a sequence of
      digits (possibly preceded by a sign) follows, read
      it as a decimal integer. IsNum indicates whether a number
      was read, nextChar gives the last character read, i.e. the
      one following the sequence of digits. 0C signals the end
      of the input stream. No tests for overflow are made.*)

  PROCEDURE Write(ch: CHAR);
  PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
  PROCEDURE WriteCard(x, n: CARDINAL);
  PROCEDURE WriteOct(w: WORD; n: CARDINAL);
  PROCEDURE WriteHex(w: WORD; n: CARDINAL);
  PROCEDURE WriteLn;
  PROCEDURE WriteString(s: String);
  PROCEDURE ShowString(s: String; cr: BOOLEAN);
     (* show s on standard output; if cr, terminate line *)

END InOut.
```

16.2. Streams

This module defines the abstract notion of a sequence. It implements sequences of characters and words called streams in terms of files as presented by the utility module Files.

A stream must first be connected to a file, which itself must first have been opened. When a stream is connected, it can be read or written as a sequence of characters or words. The type of the stream's elements is defined when connecting the stream with a file, and the stream must be read (written) accordingly with the procedures ReadChar or ReadWord (WriteChar or WriteWord). Each call sequentially reads (writes) the next element.

The writing of a stream must be terminated by a call to EndWrite. Thereafter the stream may be disconnected from its file and the file may be closed. After reading an element, the Boolean value obtained from procedure EOS (end of stream) indicates whether the next element had actually been read or the end of the stream was reached. In the latter case, the value obtained by ReadChar is a null character (0C). The procedure Reset is used to reset the reading or writing position to the beginning of the stream.

```
DEFINITION MODULE Streams;     (* N.Wirth 22.1.80*)

  FROM SYSTEM IMPORT WORD;
  FROM Files IMPORT FILE;

  EXPORT QUALIFIED
    STREAM, Connect, Disconnect, Reset,
    WriteWord, WriteChar, EndWrite,
    ReadWord, ReadChar, EOS;

  TYPE STREAM;
```

```
  PROCEDURE Connect(VAR s: STREAM; f: FILE; ws: BOOLEAN);
    (* Connect stream s with (open) file f.
       f = RT-11 channnel number, 0 <= f < 16.
       ws = "s is a word (not a character) stream" *)

  PROCEDURE Disconnect(VAR s: STREAM; closefile: BOOLEAN);
  PROCEDURE Reset(s: STREAM);
  PROCEDURE WriteWord(s: STREAM; w: WORD);
  PROCEDURE WriteChar(s: STREAM; ch: CHAR);
  PROCEDURE EndWrite(s: STREAM);
  PROCEDURE ReadWord(s: STREAM; VAR w: WORD);
  PROCEDURE ReadChar(s: STREAM; VAR ch: CHAR);
  PROCEDURE EOS(s: STREAM): BOOLEAN;
END Streams.
```

The following two schemata demonstrate the typical use of streams. The first generates a stream "out" connected with file 1, the second reads a stream "in" connected to file 2.

```
VAR in, out: STREAM;  ch: CHAR;

Connect(out, 1, FALSE);
REPEAT ... WriteChar(out, ch) ... UNTIL ... ;
EndWrite(out); Disconnect(out, FALSE);

Connect(in, 2, FALSE); ReadChar(in, ch);
WHILE NOT EOS(in) DO
  ...  ReadChar(in, ch)
END ;
Disconnect(in, TRUE)
```

16.3. Files

This module allows the programmer to access files stored on disk with Modula-2 procedure calls. Evidently, these procedures depend to some degree on the underlying file system which is usually an integral part of the given operating system and cannot be determined by the Modula-2 implementation. The module shown here is designed for the RT-11 file system. RT-11 files are essentially sequences of blocks of data corresponding to disk sectors. Each block consists of 512 characters or 256 words. Blocks are numbered 0, 1, 2, ...

Each file is identified by a value of type FILE, which in RT-11 terminology is called a channel number. A file is assigned a channel number by calling either Lookup (when fetching an already existing file listed in the dictionnary) or Create (when establishing a new file). Both procedures require the specification of a file name. An existing file can either be read or overwritten. A new file is entered in the dictionnary when calling the procedure Close (not by Create) after writing has been completed. If this process registers a file name identical to one listed already, then the old file is deleted. An assigned channel number is freed either by Close or, if the assigned file is not to be registered in the dictionnary, by Release.

A file name consists of 12 characters (RT-11 convention). The first

three specify the device (default = "DK "), the next six are the actual name, and the last three are the so-called name extension. For further details, the reader is referred to the RT-11 manual.

```
DEFINITION MODULE Files;        (* Ch. Jacobi 17.9.78 for RT-11 *)

  FROM SYSTEM IMPORT  ADDRESS, WORD;
  IMPORT SystemTypes;

  EXPORT QUALIFIED
    FILE, FileName, Lookup, Create, Delete,
    Release, Close, WriteBlock, ReadBlock, Rename,

    SetBlock, TransmitBlock, Rad50name, Radix50, Errcode;

  (* Procedure     RT-11       function
     name          request

     Lookup        .LOOKUP     lookup file in dictionnary
     Create        .ENTER      create a new file
     Delete        .DELETE     delete file and entry from dictionnary
     Release       .PURGE      release file, no dictionnary entry
     Close         .CLOSE      close file and register in dictionnary
     WriteBlock    .WRITEW     write
     ReadBlock     .READW      read
     Rename        .RENAME     rename a file *)

  TYPE FILE = [0..15];
       FileName = SystemTypes.FileName;  (* ARRAY [0..11] OF CHAR *)

  PROCEDURE Lookup(f: FILE; fn: FileName; VAR reply: INTEGER);
    (* lookup file f in dictionnary
       reply: >=0 = done, file length
              <0 = error
              -1 = channel used
              -2 = file not found *)

  PROCEDURE Create(f: FILE; fn: FileName; VAR reply: INTEGER);
    (* create a new file f
       reply: >=0 = done, file length
              <0 = error
              -1 = channel used
              -2 = no space *)

  PROCEDURE Delete(f: FILE; fn: FileName; VAR reply: INTEGER);
    (* delete file f and entry from dictionnary
       reply: >=0 = done, file length
              <0 = error
              -1 = channel used
              -2 = file not found *)

  PROCEDURE Close(f: FILE);
    (* close file f and register in dictionnary *)

  PROCEDURE Release(f: FILE);
    (* release file f, no entry in dictionnary *)
```

```
  PROCEDURE ReadBlock(f: FILE; p: ADDRESS; blknr, wcount: CARDINAL;
                      VAR reply: INTEGER);
    (* read from file f
       p:      address of buffer
       blknr:  blocknumber of first block to read
       wcount: number of words to read
       reply:  >=0 = number of words transferred
                <0 = error
                -1 = hard error
                -2 = channel not open  *)

  PROCEDURE WriteBlock(f: FILE; p: ADDRESS; blknr, wcount: CARDINAL;
                       VAR reply: INTEGER);
    (* write to file f
       p:      address of buffer
       blknr:  blocknumber of first block to write
       wcount: number of words to write
       reply:  >=0 = number of words transferred
                <0 = error
                -1 = hard error
                -2 = channel not open  *)

  PROCEDURE Rename(f: FILE; new, old: FileName;
                   VAR reply: INTEGER);
    (* renames file f which must not be open
       reply:  0 = done
              <0 = error
              -1 = channel used
              -2 = file not found; *)

  (*-----------------------------------------------------------------*)

  TYPE Rad50name = ARRAY [0..3] OF INTEGER;

  VAR Errcode[52B]: CHAR; (* RT-11 error location,
                             for detection of further errors *)

  PROCEDURE Radix50(VAR name: FileName; VAR name50: Rad50name);

  PROCEDURE SetBlock(f: FILE; VAR fn: FileName; func, l: CARDINAL;
                     VAR reply: INTEGER);
    (* func:  function code: 0 = Delete; 1 = Lookup; 2 = Create
       l:     file length code for Create
       reply: >=0 = done, file length
               <0 = error
               -1 = channel used
               -2 = file not found/no space *)

  PROCEDURE TransmitBlock(f: FILE; func: CARDINAL;
                          PtrToBuf: ADDRESS; blknr, wcount: WORD;
                          VAR reply: INTEGER);
    (* func:   10B = Read; 11B = Write
       reply:  >=0 = number of words transferred
                <0 = error
                -1 = hard error
                -2 = channel not open  *)

END Files.
```

16.4. Terminal input and output

This module exports a procedure that reads a character from the keyboard and one that writes a character on the display (or typewriter). A procedure SetMode allows to read according to several different modes of operation; it is RT-11 specific.

```
DEFINITION MODULE TTIO; (* Ch. Jacobi 15.12.79 *)

  EXPORT QUALIFIED Read, ReadAgain, Write, SetMode;

  PROCEDURE Read(VAR ch: CHAR);

  PROCEDURE ReadAgain;
    (* put the last character back into the buffer,
       such that it can be read again, only once *)

  PROCEDURE Write(ch: CHAR);

  PROCEDURE SetMode(m: CARDINAL; cc: BOOLEAN);
    (* m = 0: read does:  No echo, give control back after any key
                          has been pressed, no special handling
                          of control characters. (default)
          1:               as m=0, but if neither a key is nor had
                           been pressed, a 0C character is returned
                           immediately.
          2:               echo, buffer and wait until CR is pressed,
                           allow indirect command file input,
                           handle automatic backspaces.
       cc = "interpret <ctrl>C as abort to RT-11"  *)
END TTIO.
```

16.5. Storage management

The exported procedures ALLOCATE and DEALLOCATE serve to obtain and return storage space for dynamically allocated variables. The allocated space is identified by a pointer p; its size in terms of storage units is specified by the parameter size.

Assuming the declarations

```
TYPE T =  ...  ;
VAR  p: POINTER TO T;
```

the statements NEW(p) and DISPOSE(p) are translated by the compiler into

```
ALLOCATE(p, TSIZE(T))  and
DEALLOCATE(p, TSIZE(T))
```

Therefore, whenever NEW or DISPOSE occur in a program, procedures ALLOCATE and DEALLOCATE must either be imported or explicitly declared. Usually, they will be imported from the standard module Storage; however, it is also possible to import them from another module supplied by the programmer.

```
DEFINITION MODULE Storage; (* Ch. Jacobi 5.12.79  *)

  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED ALLOCATE, DEALLOCATE, SetMode;

  PROCEDURE ALLOCATE(VAR p: ADDRESS; size: CARDINAL);

  PROCEDURE DEALLOCATE(VAR p: ADDRESS; size: CARDINAL);

  PROCEDURE SetMode(m: CARDINAL);
    (* m= 1:  ALLOCATE aborts when not enough free space (default)
          2:  ALLOCATE gives NIL when not enough free space *)

END Storage.
```

16.6. The Loader

This module exports the procedure Call, which loads a so-called load-module generated by the linker. It uses a stack allocation scheme. The base address of a load module is determined by the base file specified when calling the linker (default is the basic operating system). The scheme allows programs to load other programs (which are called overlays). The variable FirstFree specifies the address of the top of the code stack.

```
DEFINITION MODULE Loader;
   (* Ch. Jacobi, H.H. Naegeli  5.12.79 *)

  FROM SystemTypes IMPORT FileName, LoadResultType, ErrorType;

  EXPORT QUALIFIED Call, FirstFree;

  VAR FirstFree: CARDINAL;  (* address of first free location after
                               top overlay layer. Read-Only *)

  PROCEDURE Call(fn: FileName; VAR LoadRes: LoadResultType;
                 VAR ExecutionRes: ErrorType);

     (* Load and execute a program.

        fn:           Name of code file to be loaded
        LoadRes:      result of loading
        ExecutionRes: result of execution *)

END Loader.
```

16.7. Process Scheduler

Several programming languages of recent years have postulated concepts for expressing concurrent activities, notably Concurrent Pascal, Modula, Pearl, and Portal. They all center around the notion of the sequential process, augmented by a facility for the synchronization of processes. Modula-2 instead features only the

lower-level coroutine concept with an explicit transfer of control from one coroutine to another. The module described here shows how processes and synchronization signals can be expressed in terms of coroutines and transfer statements.

The module ProcessScheduler exports a procedure for starting a process. Its parameters specify a procedure that is to represent the process' actions, and a variable (array) that represents its workspace. The module also exports a data type SIGNAL and its associated operators SEND and WAIT. Each call SEND(s) wakes up exactly one process waiting for s, if there is any.

```
DEFINITION MODULE ProcessScheduler;  (*N.Wirth, 29.1.80*)
  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED
    SIGNAL, StartProcess, SEND, WAIT, Awaited, InitSignal;

  TYPE SIGNAL;

  PROCEDURE StartProcess(P: PROC; A: ADDRESS; n: CARDINAL);
    (* start P with workspace A of length n  *)

  PROCEDURE SEND(VAR s: SIGNAL);
    (* resume first process waiting for s *)

  PROCEDURE WAIT(VAR s: SIGNAL);
    (* insert at end of queue waiting for s and resume any
       process that is ready *)

  PROCEDURE Awaited(s: SIGNAL): BOOLEAN;

  PROCEDURE InitSignal(VAR s: SIGNAL);

END ProcessScheduler.
```

The essential difference between the above mentioned languages and Modula-2 is that the former present processes as a distinct language feature and therefore require a built-in, resident mechanism for scheduling processes (at least for implementations with fewer processors than processes), whereas Modula-2 allows (and requires) the explicit programming of such an algorithm. It can be simple or sophisticated depending on application and circumstances.

Subsequently we present one possible solution in the form of a corresponding implementation module. Its characteristic is simplicity, not sophistication. It corresponds very closely to the nucleus of implementations of Modula. Processes with unscheduled transfers, i.e. interrupts, represented by DOIO statements in Modula, are not treated here.

The implementation module reveals that signals are represented by queues (linked lists) of process descriptors. A signal variable holds a pointer to the queue's head. SEND unlinks the first element of a queue; WAIT inserts a descriptor at the queue's end, hence establishing a first-in first-out strategy. All process descriptors are also linked together in a ring. This ring serves to find a next ready process after suspending the current process by a WAIT.

```
IMPLEMENTATION MODULE ProcessScheduler;
  (* N. Wirth. 29.1.80 *)
  FROM SYSTEM IMPORT PROCESS, ADDRESS, NEWPROCESS, TRANSFER;
  FROM Storage IMPORT ALLOCATE;

  TYPE SIGNAL = POINTER TO ProcessDescriptor;
    ProcessDescriptor =
      RECORD ready: BOOLEAN;
        pr:    PROCESS;
        next:  SIGNAL; (* ring *)
        queue: SIGNAL; (* waiting queue *)
      END ;

  VAR cp: SIGNAL; (* current process *)

  PROCEDURE StartProcess(P: PROC; A: ADDRESS; n: CARDINAL);
    (* start P with workspace A of length n  *)
    VAR t: SIGNAL;
  BEGIN t := cp; NEW(cp);
    WITH cp↑ DO
      next := t↑.next; ready := TRUE; queue := NIL; t↑.next := cp
    END ;
    NEWPROCESS(P, A, n, cp↑.pr); TRANSFER(t↑.pr, cp↑.pr)
  END StartProcess;

  PROCEDURE SEND(VAR s: SIGNAL);
    (* resume first process waiting for s *)
    VAR t: SIGNAL;
  BEGIN
    IF s # NIL THEN
      t := cp; cp := s;
      WITH cp↑ DO
        s := queue; ready := TRUE; queue := NIL
      END ;
      TRANSFER(t↑.pr, cp↑.pr)
    END
  END SEND;

  PROCEDURE WAIT(VAR s: SIGNAL);
    VAR t0, t1: SIGNAL;
  BEGIN (* insert current process at end of queue s *)
    IF s = NIL THEN s := cp ELSE
      t0 := s;
      LOOP t1 := t0↑.queue;
        IF t1 = NIL THEN t0↑.queue := cp; EXIT END ;
        t0 := t1
      END
    END ;
    cp↑.queue := NIL; cp↑.ready := FALSE;
    t0 := cp; (*now find next ready process*)
    LOOP cp := cp↑.next;
      IF cp↑.ready THEN EXIT END ;
      IF cp = t0 THEN HALT (*deadlock*) END
    END ;
    TRANSFER(t0↑.pr, cp↑.pr)
  END WAIT;
```

```
  PROCEDURE Awaited(s: SIGNAL): BOOLEAN;
  BEGIN RETURN s # NIL
  END Awaited;

  PROCEDURE InitSignal(VAR s: SIGNAL);
  BEGIN s := NIL
  END InitSignal;

BEGIN NEW(cp);
  WITH cp↑ DO
    next := cp; ready := TRUE; queue := NIL
  END
END ProcessScheduler.
```

17. Syntax summary and index

```
 1  ident = letter {letter | digit}.
 2  number = integer | real.
 3  integer = digit {digit} | octalDigit {octalDigit} ("B"|"C")|
 4      digit {hexDigit} "H".
 5  real = digit {digit} "." {digit} [ScaleFactor].
 6  ScaleFactor = "E" ["+"|"-"] digit {digit}.
 7  hexDigit = digit |"A"|"B"|"C"|"D"|"E"|"F".
 8  digit = octalDigit | "8"|"9".
 9  octalDigit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7".
10  string = "'" {character} "'" | '"' {character} '"' .
11  qualident = ident {"." ident}.
12  ConstantDeclaration = ident "=" ConstExpression.
13  ConstExpression = SimpleConstExpr [relation SimpleConstExpr].
14  relation = "=" | "#" | "<>" | "<" | "<=" | ">" | ">=" | IN .
15  SimpleConstExpr = ["+"|"-"] ConstTerm {AddOperator ConstTerm}.
16  AddOperator = "+" | "-" | OR .
17  ConstTerm = ConstFactor {MulOperator ConstFactor}.
18  MulOperator = "*" | "/" | DIV | MOD | AND | "&" .
19  ConstFactor = qualident | number | string | set |
20      "(" ConstExpression ")" | NOT ConstFactor.
21  set = [qualident] "{" [element {"," element}] "}".
22  element = ConstExpression [".." ConstExpression].
23  TypeDeclaration = ident "=" type.
24  type = SimpleType | ArrayType | RecordType | SetType |
25      PointerType | ProcedureType.
26  SimpleType = qualident | enumeration | SubrangeType.
27  enumeration = "(" IdentList ")".
28  IdentList = ident {"," ident}.
29  SubrangeType = "[" ConstExpression ".." ConstExpression "]".
30  ArrayType = ARRAY SimpleType {"," SimpleType} OF type.
31  RecordType = RECORD FieldListSequence END.
32  FieldListSequence = FieldList {";" FieldList}.
33  FieldList = [IdentList ":" type |
34      CASE [ident ":"] qualident OF variant {"|" variant}
35      [ELSE FieldListSequence] END].
36  variant = CaseLabelList ":" FieldListSequence.
37  CaseLabelList = CaseLabels {"," CaseLabels}.
38  CaseLabels = ConstExpression [".." ConstExpression].
39  SetType = SET OF SimpleType.
40  PointerType = POINTER TO type.
41  ProcedureType = PROCEDURE [FormalTypeList].
42  FormalTypeList = "(" [[VAR] FormalType
43      {"," [VAR] FormalType}] ")" [":" qualident].
44  VariableDeclaration = IdentList ":" type.
45  designator = qualident {"." ident | "[" ExpList "]" | "↑"}.
46  ExpList = expression {"," expression}.
47  expression = SimpleExpression [relation SimpleExpression].
48  SimpleExpression = ["+"|"-"] term {AddOperator term}.
49  term = factor {MulOperator factor}.
50  factor = number | string | set | designator [ActualParameters] |
51      "(" expression ")" | NOT factor.
52  ActualParameters = "(" [ExpList] ")" .
53  statement = [assignment | ProcedureCall |
54      IfStatement | CaseStatement | WhileStatement |
55      RepeatStatement | LoopStatement | ForStatement |
56      WithStatement | EXIT | RETURN [expression] ].
```

```
57  assignment = designator ":=" expression.
58  ProcedureCall = designator [ActualParameters].
59  StatementSequence = statement {";" statement}.
60  IfStatement = IF expression THEN StatementSequence
61      {ELSIF expression THEN StatementSequence}
62      [ELSE StatementSequence] END.
63  CaseStatement = CASE expression OF case {"|" case}
64      [ELSE StatementSequence] END.
65  case = CaseLabelList ":" StatementSequence.
66  WhileStatement = WHILE expression DO StatementSequence END.
67  RepeatStatement = REPEAT StatementSequence UNTIL expression.
68  ForStatement = FOR ident ":=" expression TO expression
69      [BY ConstExpression] DO StatementSequence END.
70  LoopStatement = LOOP StatementSequence END.
71  WithStatement = WITH designator DO StatementSequence END .
72  ProcedureDeclaration = ProcedureHeading ";" block ident.
73  ProcedureHeading = PROCEDURE ident [FormalParameters].
74  block = {declaration} [BEGIN StatementSequence] END.
75  declaration = CONST {ConstantDeclaration ";"} |
76      TYPE {TypeDeclaration ";"} |
77      VAR {VariableDeclaration ";"} |
78      ProcedureDeclaration ";" | ModuleDeclaration ";".
79  FormalParameters =
80      "(" [FPSection {";" FPSection}] ")" [":" qualident].
81  FPSection = [VAR] IdentList ":" FormalType.
82  FormalType = [ARRAY OF] qualident.
83  ModuleDeclaration =
84      MODULE ident [priority] ";" {import} [export] block ident.
85  priority = "[" integer "]".
86  export = EXPORT [QUALIFIED] IdentList ";".
87  import = [FROM ident] IMPORT IdentList ";".
88  DefinitionModule = DEFINITION MODULE ident ";" {import}
89      [export] {definition} END ident ".".
90  definition = CONST {ConstantDeclaration ";"} |
91      TYPE {ident ["=" type] ";"} |
92      VAR {VariableDeclaration ";"} |
93      ProcedureHeading ";".
94  ProgramModule =
95      MODULE ident [priority] ";" {import} block ident ".".
96  CompilationUnit = DefinitionModule |
97      [IMPLEMENTATION] ProgramModule ".".

assignment              -57    53
ActualParameters         58   -52    50
AddOperator              48   -16    15
ArrayType               -30    24
block                    95    84   -74    72
case                    -65    63    63
CaseLabelList            65   -37    36
CaseLabels              -38    37    37
CaseStatement           -63    54
CompilationUnit         -96
ConstantDeclaration      90    75   -12
ConstExpression          69    38    38    29    29    22    22    20
                        -13    12
ConstFactor              20   -19    17    17
ConstTerm               -17    15    15
character                10    10
```

DefinitionModule 96 -88
declaration -75 74
definition -90 89
designator 71 58 57 50 -45
digit -8 7 6 6 5 5 5 4
3 3 1
ExpList 52 -46 45
element -22 21 21
enumeration -27 26
export 89 -86 84
expression 68 68 67 66 63 61 60 57
56 51 -47 46 46
factor 51 -50 49 49
FieldList -33 32 32
FieldListSequence 36 35 -32 31
FormalParameters -79 73
FormalType -82 81 43 42
FormalTypeList -42 41
ForStatement -68 55
FPSection -81 80 80
hexDigit -7 4
IdentList 87 86 81 44 33 -28 27
IfStatement -60 54
ident 95 95 91 89 88 87 84 84
73 72 68 45 34 28 28 23
12 11 11 -1
import 95 88 -87 84
integer 85 -3 2
LoopStatement -70 55
letter 1 1
ModuleDeclaration -83 78
MulOperator 49 -18 17
number 50 19 -2
octalDigit -9 8 3 3
priority 95 -85 84
PointerType -40 25
ProcedureCall -58 53
ProcedureDeclaration 78 -72
ProcedureHeading 93 -73 72
ProcedureType -41 25
ProgramModule 97 -94
qualident 82 80 45 43 34 26 21 19
-11
RecordType -31 24
RepeatStatement -67 55
real -5 2
relation 47 -14 13
set 50 -21 19
statement 59 59 -53
string 50 19 -10
ScaleFactor -6 5
SetType -39 24
SimpleConstExpr -15 13 13
SimpleExpression -48 47 47
SimpleType 39 30 30 -26 24
StatementSequence 74 71 70 69 67 66 65 64
62 61 60 -59
SubrangeType -29 26

```
term                  -49    48    48
type                   91    44    40    33    30   -24    23
TypeDeclaration        76   -23
VariableDeclaration    92    77   -44
variant               -36    34    34
WhileStatement        -66    54
WithStatement         -71    56
```