MODULA

a language for modular multiprogramming

Eidgenössische
Technische
Hochschule
Zürich

*Institut
für
Informatik*

Niklaus Wirth

MODULA: A language for modular multiprogramming

März 1976

18

Eidgenössische
Technische
Hochschule
Zürich

Institut
für
Informatik

Niklaus Wirth

MODULA: A language for modular multiprogramming

MODULA: A LANGUAGE FOR MODULAR MULTIPROGRAMMING

N.Wirth

Abstract

This paper defines a language called Modula, which is intended primarily for programming dedicated computer systems, including process control systems on smaller machines. The language is largely based on Pascal, but in addition to conventional block structure it introduces a so-called module structure. A module is a set of procedures, data types, and variables, where the programmer has precise control over the names that are imported from and exported to the environment. Modula includes general multiprocessing facilities, namely processes, interface modules, and signals. It also allows the specification of facilities that represent a computer's specific peripheral devices. Those given in this paper pertain to the PDP-11.

Author's address:
Institut für Informatik, ETH, CH-8092 Zürich

Contents

1. INTRODUCTION

The advantages of high-level programming languages over assembly code in the design of complex systems have been widely recognised and commented [3]. The primary benefit of the use of a suitable high-level language lies in the possibility of defining abstract machines in a precise manner that is reasonably independent of characteristics of particular hardware. Assembly code is still used virtually exclusively in those applications whose predominant purpose is not to design a new system based on abstract specifications, but to operate an existing machine with all its particular devices. Good examples are process control systems, computerised laboratory equipment, and input-output device drivers.

A major aim of the research on Modula is to conquer that stronghold of assembly coding, or at least to attack it vigorously. There is strong evidence that the influence of high-level languages may become equally significant for process control programming as it is now for compiler and operating system design. It may become even more important, considering the availability of microprocessors at very low cost.

There are two requirements of such languages that are characteristic for this area of applications, and that have not existed for general purpose languages in the past. They must offer facilities for multiprogramming, i.e. they must allow to express the concurrent execution of several activities, and they must offer facilities to operate a computer's peripheral devices. The principal obstacle is that such facilities are inherently machine- and even configuration dependent, and as such elude a comprehensive abstract definition. A practical solution lies in accepting this situation and introducing a language construct that encapsulates such machine dependent items, i.e. restricts their validity or existence to a specific, usually very small section of a program. We call this construct a module. Its usefulness is quite general and by no means restricted to the domain of device dependent operations. It is indeed of central importance, and has given the language its name (for modular programming language).

Modula relies very strongly on Pascal [6,10]. Some of Pascal's features, however, have been omitted. The reason is not their inadequacy, but rather the necessity and the desire to keep the language reasonably small. This seemed particularly advisable for a language intended primarily for small computers.

Chapter 2 of this paper gives a brief overview of the language, and concentrates on its "novel" features for modular design, multiprogramming, and device operation. Chapters 3 and 4 define the formalism for the syntactic definition of Modula, and of the actual representation of programs. Chapter 5 defines the core of

the language in detail. It constitutes the conventional part restricted to sequential programming. Most of these features have been adopted from Pascal. Chapter 6 describes the facilities added to express concurrency. The typical Modula program consists of several processes that are themselves sequential algorithms, and which are loosely coupled through synchronization operations. Its multiprogramming facilities are designed accordingly.

The language described in Chapters 5 and 6 is defined in an abstract way which does not refer to any specific computer. It is well-suited to be implemented on any existing computer. But the purpose of a system programming language is not only to aid in constructing and specifying abstract systems, but also to operate existing hardware. For this purpose, additional features must be provided that serve to operate particular facilities of a given computer, in particular its peripheral devices. Chapter 7 describes a set of such features that were added to operate the PDP-11. Such machine-dependent objects can only be declared in specially designated device modules.

In this connection the total lack of any kind of input-output facilities must be explained. Even the file structure of Pascal is missing. The reason is that the typical application of Modula is regarded as the design of systems that implement rather than use such a file facility. It should be possible to run Modula programs on a bare machine with a minimal run-time support. Hence, Modula cannot contain "high-level" features such as file operations. Instead, they may be programmed in terms of device operations, neatly encapsulated within modules.

Another concept that is conspicuously absent is the interrupt. The explanation is that the interrupt is a processor-oriented concept. The purpose of interrupts is to let a processor take part in the execution of several processes (tasks). Hence, in a language that focusses on sequential processes as principal constituents of its programs, the interrupt is a technique to implement a multi-process program on a single-processor computer, rather than a language concept.

A most important consideration in the design of Modula was its efficient implementability. In particular, the presented solution allows processor management and "interrupt handling" to be implemented equally efficiently as in assembly coding. Guarantee of efficiency and absence of additional overhead in the use of a high-level language is an absolute prerequisite for a successful campaign to promote the use of such languages in this traditional stronghold of assembly coding.

The current implementation of Modula is experimental. The presence or absence of certain facilities is still subject to controversy. The usefulness of their presence or the effects of their absence will only be known after a considerable amount of

actual experience in the use of Modula for the development of many different systems.

So far, a cross-compiler has been completed. It relies on a very small run-time support package to handle the switching of the processor from one process to another. In modern systems, this nucleus could well be incorporated as a micro-program or better even directly in the hardware. (The current compiler is not available for distribution.)

2. OVERVIEW

By far the largest part of Modula consists of facilities typical for any sequential programming language. This is not surprising, because even large operating and control systems consist of processes that are themselves purely sequential. Brinch Hansen, in describing the design of an entire operating system, reports that only 4% were coded in a language with multiprogramming facilities [2].

Most of the sequential programming facilities of Modula have been adopted from Pascal, notably the concepts of data types and structures. Modula offers the basic types integer, Boolean, and char. In addition, scalar types can be defined by the programmer by enumerations. In the realm of structures, only the array and the record - the latter without variants - have been adopted. The role of sets is partially taken over by a standard type called bits, which constitutes a short Boolean array. The language does not include any pointers.

Modula provides a rich set of program control structures, including if, case, while, and repeat statements. Their syntax slightly differs from that of Pascal, because the principle was followed that every structured statement not only begins but also ends with an explicit bracketing symbol. The for statement has been replaced by a more general loop statement that allows to specify one or several termination points.

Procedures can be used recursively and can have two kinds of parameters, namely constant and variable parameters. In the former case, assignments to the formal parameter are prohibited, and the corresponding actual parameter is an expression. In the latter case, the actual parameter must be a variable, and assignments to the formal parameter are assignments to that actual variable. (In both cases, parameters may - but do not have to - be implemented by passing an address.)

Procedures form a block in the sense of Algol and Pascal. Hence, constants, types, variables, and other procedures can be declared local to a procedure. This implies that their existence is not known outside the procedure, which thereby constitutes the scope of these local objects. Block structure has proven to

be a most valuable facility in systematic program design. However, block structure alone does not provide the possibility to retain local objects after termination of the procedure, nor to let several procedures share retained (hidden) objects. For this purpose, the module has been introduced as an essential supplement to the block concept.

A module is a collection of constant-, type-, variable-, and procedure declarations. They are called objects of the module and come into existence when control enters the procedure to which the module is local. The module should be thought as a fence around its objects. The essential property of the module construct is that it allows the precise determination of this fence's transparency. In its heading, a module contains two lists of identifiers: The define-list mentions all module objects that are to be accessible (visible) outside the module. The use-list mentions all objects declared outside the module that are to be visible inside. This facility provides an effective means to make available selectively those objects that represent an intended abstraction, and to hide those objects that are considered as details of implementation. A module encapsulates those parts that are non-essential to the remainder of a program, or that are even to be protected from inadvertant access [7]. Modules may be nested.

Example: Suppose that an object a is declared in the environment of M1. Then a, b, and c are accessible in this environment:

```
module M1;
   define b,c;
   use a;
   {declare d}

   module M2;
      define c,e;
      use d;
      {declare c,e,f}
      {c,d,e,f are accessible here}
   end M2;

   procedure b;
      {declare f}
      {a,b,c,d,e,f are accessible here}
   end b ;

   {a,b,c,d,e are accessible here}
end M1
```

Identifiers in the define-list are said to be exported, those in the use-list are imported. If a type is exported, then only its identity is exported, but not its structural details. This means that outside the module from which a type is exported we do not know whether a type is a scalar, an array, or a record.

Therefore, variables of this type can be operated by procedures only that are exported from the same module. The module [8] therefore assumes a similar role as the class construct of Concurrent Pascal [1], which was developed from the class structure of Simula [4].

Exported variables cannot be changed except in the module to which they are local, i.e. they appear as read-only variables. It must be emphasized that the module does not determine the "life-time" of its local objects. It merely establishes a new scope. Objects declared within a module are considered local to the procedure in which the module itself is local, i.e. they come into existence when that procedure is called, and they vanish when it is completed.

Only a minimal number of facilities for multiprogramming are added to the language described so far, which we may call sequential Modula. The additional facilities are processes, interface modules, and signals. A process looks like a procedure. But unlike the procedure it is executed concurrently with the program that called, i.e. initiated it. When control reaches the end of a process, the process goes out of existence. Processes cannot create other processes. Process creation is possible in the main program only, which should be regarded as a system initialization phase. However, it is possible to activate several instances of the same process declaration.

Synchronization is achieved by the use of signals. They are declared similar to variables (syntactically, the signal appears as a data type). Signals can be sent, and a process can wait for a signal. Signals correspond to conditions of Hoare [5] and to queues of Brinch Hansen [1]. A central aspect of this concept is that processes, once started, are anonymous. They can be influenced by signals (and shared variables) only. But the environment cannot force a process to notice these signals (or changes of variables), and there is no way to disrupt or terminate a process by outside intervention.

Processes cooperate via common variables. This requires a facility to guarantee mutual exclusion of processes from critical sections of a program. In Modula, such sections are declared as procedures and these procedures are gathered within a specially designated, so-called interface module, which corresponds to Hoare's monitor [5]. The monitor is a set of corresponding critical sections, where simultaneous execution by several processes is excluded. In contrast to the critical section, however, the interface module allows more than one process to be in a critical section, provided that all but one are either waiting for a signal or are sending a signal. This relaxation of the mutual exclusion condition not only simplifies implementation of the signalling and processor switching mechanism, but also corresponds to many practical patterns of usage. A typical program pattern with two cooperating processes

P and Q is shown below, where v stands for the common variables, for instance data buffers in a producer-consumer constellation, s stands for the signals by which P and Q synchronize their activities, and p and q are their critical sections formulated as interface procedures (see Fig. 1).

```
interface module M;
   define p,q;
   {declare v,s}
   procedure p(x);
      {uses v,s,x}
   end p;
   procedure q(y);
      {uses v,s,y}
   end q;
begin {initialise v}
end M;

process P;
   {uses p}
end P;

process Q;
   {uses q}
end Q
```

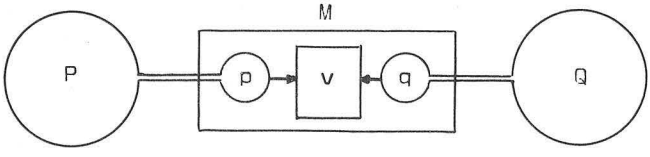


Fig.1. Processes P and Q interfacing via module M.

These general multiprogramming facilities are supplemented by a few computer-dependent features. They are necessary, for instance, to operate a computer's peripheral devices. We describe as an example those which were designed and implemented for the PDP-11. The underlying intention was to keep the number of such facilities minimal, and to express them in strong analogy with machine-independent concepts wherever feasible and economical.

A necessary condition is that the computer's device registers and operators on them are made available. They appear in Modula as variables with a specification of their (hardware-defined) address and (programmer-defined) type. Status registers are usually declared of type bits, which allows the convenient setting and resetting of individual status and function bits.

Moreover, a system implementation language should allow the effective utilization of a computer's interrupt facility, including its interrupt priority system, if one exists. Traditionally, an input/output device is regarded as a process by itself, communicating with a master process by starting signals and completion interrupts. In Modula, the operations performed by the device and those executed by its associated interrupt routine are considered as a single process, the former part being represented by the statement "doio" [9] (see Fig.2). This statement is allowed within so-called device processes (or drivers) only which are declared within a so-called device module. In contrast to regular processes drivers are declared entirely within the device interface module. This is possible, because the doio statement - representing the actions performed by the device - also constitutes a singular point within the interface in the sense that during its execution the mutual exclusion constraint is lifted.

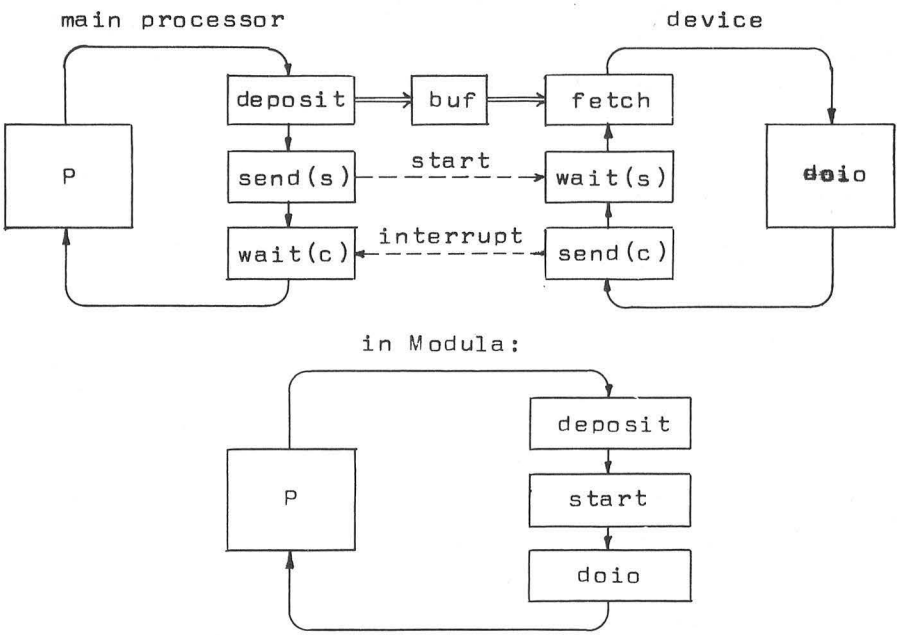


Fig.2. I/O activities viewed as a device process

The explicit designation of device interface modules and drivers facilitates for an implementation the efficient utilization of a given computer system, but also exhibits the close relationship or even identity of these machine oriented parts of a program with the machine-independent concepts of the language.

3. NOTATION FOR SYNTACTIC DESCRIPTION

To describe the syntax an extended Backus-Naur formalism is used. It allows to use syntax expressions as right parts in a production. Syntactic entities are denoted by English words expressing their intuitive meaning. Symbols of the language are enclosed by quote marks (") and appear as so-called literals in the right parts of productions. Each production has the form

S = E .

where S is a syntactic entity and E 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 > Ø)

where the Ti's 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 > Ø)

where the Fi's are the factors of T. Each Fi stands for a set of sentential forms, and T denotes their product. The product 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 has either the form

"x"

(x is a literal, and "x" denotes the singleton set consisting of this single symbol), or

(E)

(denoting the expression E), or

[E]

(denoting the union of the set denoted by E and the empty sentence), or

{E}

(denoting the set consisting of the union of the empty sequence and the sets E, EE, EEE, etc.)

Examples:
The syntax expressions

("a"|"b") ("b"|"c") , "a" {"bc"}, "a" ["b"|"c"] "d"

denote the following sets of sentences respectively:

ab	a	ad
ac	abc	abd
bb	abcbc	acd
bc	abcbcbc	
	...	

4. LANGUAGE VOCABULARY AND REPRESENTATION

The language is an infinite set of sentences (programs), namely the sentences well-formed according to the syntax. Each sentence (program) is a finite sequence of symbols from a finite vocabulary. The vocabulary consists of identifiers, (unsigned)

numbers, literals, operators, and delimiters. They are called lexical symbols or tokens, and in turn are composed of sequences of characters. Their representation therefore depends on the underlying character set. The ASCII set is used in this paper, but the following rules must be observed for any set:

1. Identifiers are sequences of letters and digits. The first character must be a letter. Capital and loger case letters are not distinguished.

   ident = letter {letter | digit}.

2. Numbers (integers) are sequences of digits, possibly followed by the letter B signifying "octal".

   number = integer.
   integer = digit {digit} | octaldigit {octaldigit} "B".

3. Strings are sequences of characters enclosed in quote marks. If a quote mark itself is to occur within that sequence, then it is denoted by two consecutive quote marks. A single character string may also be denoted by its ordinal number (in octal) followed by the letter C. (The ordinal refers to the character set used.)

   string = "'" {character} "'" | octaldigit {octaldigit} "C".

4. Operators and delimiters are special characters, character pairs, or (reserved) words listed in Table 1 below. In this report, they are underlined for clear distinction from identifiers. These (reserved) words must not be used in the role of identifiers.

+	(	div	until	const
-	)	mod	while	var
*	[	or	do	type
/	]	and	loop	array
=	.	not	when	record
<>	,	if	exit	procedure
<	;	then	begin	process
<=	:	elsif	end	module
>	'	else	with	interface
>=		case	value	device
	(*	of	xor	use
:=	*)	repeat		define

Table 1: Operators and Delimiters

5. Blank spaces (and line separation) are ignored unless they are essential to separate two consecutive symbols. Hence, blanks cannot occur within symbols, including identifiers, and numbers.

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

5. FACILITIES FOR SEQUENTIAL PROGRAMMING

Every program contains two essential components: parts where objects of the computation are defined and associated with identifiers, and parts where the algorithmic actions to be performed on (and with) these objects are defined. The former parts are called declarations, the latter statements. A block is a textual unit (usually) consisting of elements of both kinds in a well defined order.

Objects to be declared are constants, data types and structures, variables, procedures, modules, and processes. Procedures and modules consist themselves of a block. Hence blocks are defined recursively and can be nested (see also 5.15).

5.1 Constant declarations

A constant declaration associates an identifier with a constant value.

```
constantdeclaration = ident "=" constant.
constant = unsignedconstant | ("+"|"-") number .
unsignedconstant = ident | number | string | bitconstant.
bitconstant = "[" [bitlist] "]" .
bitlist = bitlistelement {"," bitlistelement} .
bitlistelement  = constant [":" constant] .
```

Numbers are constants of type integer. A constant denoted by a single-character string (or by an ordinal) is of type char (see 5.2.1), a string consisting of n characters is of type (see 5.2.3)

array 1:n of char

A bit constant is a constant of type bits (see 5.2.1). The elements of the bitlist are the indices of those bits that are "true". An element of the form m:n specifies that all bits with indices m through n are "true". All other bits have the value "false".

5.2 Type declarations

Every constant, variable, and expression is of a certain type. In the case of numbers and literals their type is implicitly defined, for variables it is specified by their declaration, and

for expressions it is derivable from the types of their constituent operands and operators. A data type determines the set of values that a variable of that type may assume; it also defines the structure of a variable. There are four standard types, namely integer, Boolean, char, and bits. Enumeration types (enumerations) and the types integer, Boolean, and char are unstructured, i.e. their values are atomic. Structured types (structures) can be declared in terms of these elementary types and of structures.

```
typedeclaration = ident "=" type.
type = ident | enumeration | arraystructure |
       recordstructure .
```

5.2.1 Basic types

integer — The values of type integer are the whole numbers in the range min to max, where min and max are constants dependent on available implementations. (For the PDP-11: min = -32768, max = 32767).

Boolean — The values are the truth values denoted by the predefined identifiers true and false.

char — The values are the characters belonging to the character set determined by each implementation. (For the PDP-11: the ASCII set).

bits — Its values are arrays of w Boolean elements. This type is predeclared as (see 5.2.3)

```
array Ø:w of Boolean
```

The constant w is the wordlength minus 1 of the computer on which Modula is implemented. (For the PDP-11: w=15).

5.2.2 Enumerations

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

```
enumeration = "(" identlist ")".
identlist = ident {"," ident}.
```

5.2.3 Array structures

An array structure consists of a number of components which are all of the same component type. Each component is identified by a number of indices. This number is called the dimensionality of the array. The range of index values of each dimension is specified in the declaration of the array structure. The types of the indices must not be structured.

```
arraystructure = "array" indexrangelist "of" type.
indexrangelist = indexrange {"," indexrange}.
indexrange = constant ":" constant.
```

5.2.4 Record structures

A record structure consists of a number of components, called record fields. Each component is identified by a unique field identifier. Field identifiers are known only within the record structure definition and within field designators, i.e. when they are preceded by a qualifying record variable identifier. The data type of each component is specified in the field list.

```
recordstructure = "record" fieldlist {";" fieldlist} "end".
fieldlist = [identlist ":" type].
```

Examples of type declarations:

```
color = (red,yellow,green,blue)
vector = array 1:100 of color
matrix = array 1:20, 0:10 of integer
account = record x: integer;
                 y: Boolean;
                 z: array 0:9 of char
          end
```

5.3 Variables

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

```
variabledeclaration = identlist ":" type.
```

Examples of variable declarions:

```
i,j,k: integer
p,q: Boolean
ch: char
u: record s: bits;
          a: vector
   end
s,t: bits
r: account
a: vector
m: matrix
w: array 1:10 of account
```

The syntactic construct of a designation of a variable is simply called "variable". It either refers to a variable as a whole, namely when it consists of the identifier of the variable, or to one of its components, when the identifier is followed by a selector. If a variable, say v , has a record structure with a field f , this component variable is designated by v.f . If v has an array structure, its component with index i is designated by v[i] .

```
variable = ident | variable "." ident |
           variable "[" indices "]" .
indices = expression {"," expression}.
```

Examples of variables (see declarations above):

```
i    r.x    a[i]    m[i+1,j-1]    w[i].x    u.a[k]
```

5.4 Expressions

Expressions are composed of operands (constants, variables, and functions), operators, and parentheses. They specify rules of computing values; evaluation of an expression yields a value of the type of the expression.

There are four classes of operators with different precedence (binding strength). Relational operators have the least precedence, then follow the so-called adding operators, the multiplying operators, and the negation operator not with highest precedence. Sequences of operators with equal precedence are executed from left to right.

Denotations of a variable in an expression refer to the current value of the variable. Function calls denote activation of a function procedure declaration (i.e. execution of the statements which constitute its body). The result acts as an operand in the expression. The same rules about parameter evaluation and substitution hold as in the case of a procedure call (see 5.7).

```
expression = simpleexpression [relation simpleexpression].
relation = "=" | "<>" | "<=" | "<" | ">" | ">=" .
simpleexpression = ["+"|"-"] term {addoperator term}.
addoperator = "+" | "-" | "or" | "xor" .
term = factor {muloperator factor}.
muloperator = "*" | "/" | "div" | "mod" | "and" .
factor = unsignedconstant | variable | functioncall |
         "(" expression ")" | "not" factor.
functioncall = ident parameterlist.
```

Arithmetic operators (+ - * / div mod) apply to operands of type integer and yield a result of this type. The operators + - * and / denote addition, subtraction, multiplication and division with truncated fraction. The monadic operators + and - denote identity and sign inversion. The operators div and mod yield a quotient q = x div y and a remainder r = x mod y such that x = q*y+r , $0 \leq r < y$. The divisor (or modulus) y must be strictly positive.

Example: x = -15 , y = 4
x/y = -3 , x div y = -4 , x mod y = 1 .

Boolean operators (or xor and not) apply to Boolean operands and yield a result of type Boolean. The term a and b is evaluated as

"if a then b else false", and the expression a or b is evaluated as "if a then true else b" (Note: conditional expressions are not available in Modula). Boolean operators can also be applied to operands of type bits. The specified operation is then performed on all corresponding elements of the operands.

Relations yield a result of type Boolean. (<> <= >= stand for ≠ ≤ ≥ respectively). They apply to operands of the standard types integer, char, Boolean, and bits (to the latter only = and <>), and of enumeration types.

Examples of factors:

27	i	(i+j+k)	not p	[2,3,5,7,11]

Examples of terms:

i*k	i/(i-1)	ord(ch)	(i<j) and (j<k)	s and t

Examples of simple expressions

i+j	i+5*k	-i	p or q	s or t

Examples of expressions

(i+j)*(j+k)	i	k+5	i=j	t xor [Ø:7]

(Given the variables declared in 5.3, the first three examples in each line are of type integer, the fourth is of type Boolean, and the fifth of type bits.)

5.5 Statements

Statements denote actions. Elementary statements are the assignment statement and the procedure call. Composite statements may be constructed out of elementary statements and other composite statements.

```
statement = assignment | procedurecall | processstatement |
  ifstatement | casestatement | whilestatement |
  repeatstatement | loopstatement | withstatement | .
```

5.6 Assignments

An assignment denotes the action of evaluating an expression and of assigning the resulting value to a variable. The symbol := is called assignment operator (pronounced "becomes").

```
assignment = variable ":=" expression.
```

After an assignment is executed, the variable has the value obtained by evaluating the expression. The old value is lost ("overwritten"). The variable must be of the same type as (the value of) the expression.

```
Examples of assignments:
      i := 100
      p := true
  m[i,j] := 10*i+j
```

5.7 Procedure calls

A procedure call denotes the execution of the specified procedure, i.e. of the statement part of its body. The procedure call must contain the same number of parameters as the corresponding procedure declaration. Those of the call are called actual parameters. An actual parameter corresponding (by its position in the parameter list) to a const-parameter must be an expression. The types of the actual and the formal parameters must be the same, and the formal parameter appears as a read-only parameter, i.e. assignments to this parameter are prohibited. If the actual parameter corresponds to a var-parameter, it must be a variable. That variable is substituted for the formal parameter throughout the procedure body. Types must be identical, and if the actual parameter is an indexed variable, the index expressions are evaluated upon procedure call.

```
procedurecall = identification [parameterlist].
parameterlist = "(" parameter {"," parameter} ")".
parameter = expression | variable.
```

```
Examples of procedure calls:
      inc(i,10)
      sort(a,100)
```

5.8 Statement sequences

A sequence of statements separated by semicolons is called a statement sequence and specifies the sequential execution of the statements in the order of their occurrence.

```
statementsequence = statement {";" statement} .
```

5.9 If statements

If statements specify conditional execution of actions depending on the value of Boolean expressions.

```
ifstatement = "if" expression "then" statementsequence
              {"elsif" expression "then" statementsequence}
              ["else" statementsequence] "end".
```

5.10 Case statements

Case statements specify the selective execution of a statement sequence depending on the value of an expression. First the case expression is evaluated, then the statement sequence with label equal to the resulting value is executed. The type of the case expression must not be structured.

```
casestatement = "case" expression "of" case {";" case} "end".
case = [caselabels ":" "begin" statementsequence "end"] .
caselabels = constant {"," constant}.
```

5.11 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 the first and after each execution of the statement sequence. The repetition stops as soon as this evaluation yields the value false.

```
whilestatement =
    "while" expression "do" statementsequence "end" .
```

5.12 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 .
```

5.13 Loop statements

Loop statements specify the repeated execution of statement sequences. The repetition can be terminated depending on the values of possibly several Boolean expressions, called exit conditions.

```
loopstatement = "loop" statementsequence
    {"when" expression ["do" statementsequence] "exit"
     statementsequence } "end".
```

Hence, the general form is

```
loop S1 when B1 do X1 exit
     S2 when B2 do X2 exit
        ...
```

```
        Sn when Bn do Xn exit
        S
    end
```

First, S1 is executed, then B1 is evaluated. If it yields the value true, X1 is executed and thereupon execution of the loop statement is terminated. Otherwise it continues with S2, etc. After Sn, execution continues unconditionally with S1.

Note: all repetitions can be expressed by loop statements alone, the while and repeat statement merely express simple and frequently occurring cases.

5.14 With statements

The with statement specifies a record variable and a statement sequence to be executed. In these statements field identifiers of that record variable may occur without preceding qualification, and refer to the fields of the variable specified.

```
withstatement =
    "with" variable "do" statementsequence "end" .
```

5.15 Procedures

Procedure declarations consist of a procedure heading and a block which is said to be the procedure body. The heading specifies the procedure identifier by which the procedure is called, and its formal parameters. The block contains declarations and statements.

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 acts as operand in the expression. The former are activated by a procedure call. The function procedure is distinguished in the declaration by the fact that the type of its result is indicated following the parameter list. Its body must contain an assignment to the procedure identifier which defines the value of the function procedure. There are two kinds of parameters, namely constant and variable parameters. The kind is indicated in the formal parameter list. Constant parameters stand for a value obtained through evaluation of the corresponding actual parameter when the procedure is called. Assignments cannot be made to a constant parameter. Variable parameters correspond to actual variables, and assignments to them are permitted (see 5.7).

Formal parameters are local to the procedure, i.e. their scope is the program text which constitutes the procedure declaration.

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 (or process) at level k , it has itself level k+1 . Objects declared in the block that constitutes the main program are defined to be at level Ø .

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, unless the procedure declaration contains a so-called use-list. In this case, only formal parameters, local objects, and the identifiers occurring in the use-list are known inside the procedure (see 5.16). Standard objects are accessible in any case.

```
proceduredeclaration = "procedure" ident
   ["(" formalparameters ")"] [":" ident] ";"
   [ uselist ] block ident .
formalparameters = section {";" section} .
section = ["const"|"var"] ident {"," ident} ":" formaltype.
formaltype = ["array" indextypes "of"] ident.
indextypes = identlist.
uselist = "use" [ident {"," ident}]";" .
block = {declarationpart} [initializationpart]
        [statementpart] "end" .
declarationpart = "const" {constantdeclaration ";"}|
     "type" {typedeclaration ";"}|
     "var" {variabledeclaration ";"} | module ";" |
     proceduredeclaration ";" | processdeclaration ";" .
initializationpart = "value" {ident "=" initialvalue} .
initialvalue = constant | "[" repetition "]" initialvalue |
     "(" initialvalue {"," initialvalue} ")" .
repetition = integer | ident .
statementpart = "begin" statementsequence.
```

The identifier ending the procedure declaration must be the same as the one following the symbol procedure, i.e. the procedure identifier. If the specifier const or var is missing in a section of formal parameters, then its elements are assumed to be constant (read-only) parameters.

An initialization part serves to assign initial values to variables declared in the same block. Parentheses indicate the structure of the assigned value, which must correspond to that of the initialised variable. Initialization parts can only occur in blocks at level Ø, i.e. in the main program and in modules declared in the main program.

The use of the procedure identifier in a call within its

declaration implies recursive activation of the procedure. If a formal type indicates an array structure, then only the types but not the bounds of the indices are specified.

Examples of procedure declarations:

```
procedure readinteger (var x: integer);
   var i: integer; ch: char;
begin i := 0;
      repeat readcharacter(ch)
      until ('0' <= ch) and (ch <= '9');
      repeat i := 10*i+(integer(ch) - integer('0'));
            readcharacter(ch)
      until (ch < '0') or ('9' < ch);
      x := i
end readinteger

procedure writeinteger(x: integer);
   var i,q: integer;     (*assume x >= 0*)
       buf: array 1: 10 of integer;
begin i := 0; q := x; writecharacter(' ');
      repeat i := i+1; buf[i] := q mod 10; q := q div 10
      until q = 0;
      repeat writecharacter(buf[i]); i := i-1
      until i = 0
end writeinteger

procedure gcd(x,y: integer): integer;
   var a,b: integer;     (*assume x,y > 0*)
begin a := x; b := y;
      while a <> b do
         if a < b then b := b-a else a := a-b
         end
      end ;
      gcd := a
end gcd
```

Standard procedures

Standard procedures are predeclared and available throughout every program.

Proper procedures

inc(x,n)	=	x := x+n
dec(x,n)	=	x := x-n
inc(x)	=	x := x+1
dec(x)	=	x := x-1
halt		terminates the entire program

Function procedures

off(b1,b2)	=	b1 and b2 = [] (b1,b2 of type bits)
off(b)	=	b = []
among(i,b)	=	b[i] (b is a bit expression)

```
low(a)      =     low index bound of array a
high(a)     =     high index bound of array a
adr(v)      =     address of variable v
size(v)     =     size of variable v

Type transfer functions
   integer(x)  =     ordinal of x in the set of values
                     defined by the type of x.
   char(x)     =     character with ordinal x.
```

(adr and size are of type integer, and are implementation-dependent).

5.16 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 so-called use-list and a so-called define-list. The former specifies all identifiers of objects that are used within the module and declared outside it. The latter specifies all identifiers of objects declared within the module that are to be used outside it. 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 level as the module.

```
module = moduleheading [definelist] [uselist] block ident .
moduleheading = ["interface"] "module" ident ";"|
    "device" "module" ident priority ";" .
definelist = "define" ident {"," ident} ";" .
```

The identifier at the end of the module must be the same as the one following the symbol module, i.e. the module identifier. (For an explanation of the prefixes interface and device see Sections 6.3. and 7.1) Identifiers which occur in the module's use-list are said to be imported, and those in the define-list are said to be exported.

If a type is defined local to a module and its identifier occurs in the define-list of the module, then only the type's identity, but none of its structural details becomes known outside the module. If it is a record type, the field names remain unknown, if it is an array type, index range and elements type remain unknown outside. Hence, variables declared of a type that was exported in this way from a module can be used only by procedures declared within and exported from that same module. This implies that if a module defines a type, it also has to include the definition of all operators belonging to this type.

If a local variable occurs in the define-list of a module, it cannot be changed outside the module, i.e. it appears as a read-

only variable.

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. The bodies serve to initialize local variables.

Example:

```
procedure P;
   module M1;
      define F1, n1;
      var n1: integer;
      procedure F1(x: integer): integer;
         begin ... inc(n1) ... F1 := ...
         end F1 ;
   begin n1 := Ø
   end M1 ;

   module M2;
      define F2, n2;
      var n2: integer;
      procedure F2(x: integer): integer;
         begin ... inc(n2) ... F2 := ...
         end F2 ;
   begin n2 := Ø
   end M2 ;

begin (*use procedures F1 and F2; n1 and n2 are counters of
        their calls and cannot be changed at this place*)
end P
```

In this example, the two statements n1 := Ø and n2 := Ø can be considered as being prefixed to the body of procedure P. Within this body, assignments to these variables are prohibited.

Examples:
The following sample module serves to scan a text and to copy it onto 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 (linefeed) 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 follow lf immediately.

```
module lineinput;
   define read, newline, newfile, eoln, eof, lno;
   use inchr, outchr;
   const lf = 12C; cr = 15C; fs = 34C;
   var lno: integer; (*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; 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);
            when ch >= ' ' do x := ch exit
            when ch = lf do x := ' '; eoln := true exit
            when ch = fs do x := ' '; eoln := true;
                            eof := true  exit
        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 unauthorised 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;
   define newtrack, returntrack;
   const m = 64; w = 16;    (*there are m*w tracks*)
   var i: integer;
      reserved: array 0:63 of bits;

   procedure newtrack: integer;
      (*reserves a new track, yields its index as function
        result, if a free track is found, and -1 otherwise*)
      var i,j: integer; found: Boolean;
   begin found := false; i := m;
        repeat dec(i); j := w;
              repeat dec(j);
                  if not reserved[i,j] then found := true end
              until found or (j = 0)
```

Fehler
reserved zu true setzen!

```
        until found or (i = 0);
        if found then newtrack := i*w+j
                 else newtrack := -1 end
    end newtrack;

    procedure returntrack(k: integer);
    begin (*assume 0 <= k < m*w *)
          reserved[k div w, k mod w] := false
    end returntrack;

begin i := m; (*mark all tracks free*)
      repeat dec(i); reserved[i] := [ ]
      until i = 0
end trackreservation
```

5.17 Programs

A Modula program is formulated as a module.

```
program = module "." .
```

6. FACILITIES FOR MULTIPROGRAMMING

This chapter defines those facilities that are needed to express the concurrent execution of several program parts. They are already referenced in the syntax of the preceding chapter, and comprise three essential facilities: processes, interface modules, and synchronization primitives.

6.1 Processes

A process declaration describes a sequential algorithm - including its local objects - that is intended to be executed concurrently with other processes. No assumption is made about the speed of execution of processes, except that this speed is greater than zero.

A process declaration has the form of a procedure declaration, and the same rules about locality and accessibility of objects hold.

```
processdeclaration =
  "process" ident ["(" formalparameters ")"] [intvector] ";"
   uselist block ident .
```

The identifier at the end of the declaration must be the same as the one following the symbol process, namely the process identifier. (For an explanation of intvector see 7.1)

Restriction:

Processes must be declared at level Ø, i.e. they cannot be nested or be local to procedures. Objects local to a process are said to be at level 1.

6.2 Process control statements

A process statement expresses the starting of a new process. Syntactically it corresponds to the procedure call. However, in the case of a procedure call, the calling program can be thought to be suspended until the procedure execution has been completed, whereas a program starting a new process is not suspended. Rather the execution of the started process may proceed concurrently with the continuation of the starting program.

```
processstatement = ident [parameterlist].
```

Whereas a process declaration defines a pattern of behaviour, a process statement initiates the execution of actions according to this pattern. This implies that reference to the same process declaration in several process statements initiates the concurrent execution of several processes according to the same pattern (usually according to different parameters).

Restriction:
Process statements are confined to the body of the main program, i.e. they can neither occur within procedures nor processes.

6.3 Interface modules

The interface module is the facility which provides exclusion of simultaneous access from several processes to common objects. Variables that are to establish communication or data transfers between processes are declared local to an interface module. They are accessed via procedures also declared local (so-called interface procedures) and which are exported from the module. If a process has called any such procedure, another process calling the same or another one of these procedures is delayed, until the first process has completed its procedure or starts waiting for a signal (see 6.4).

An interface module is syntactically distinguished from regular modules by the prefix symbol interface. Interface procedures must not call on procedures declared outside the interface module (except standard procedures). Examples of interface modules are given in Section 6.4.

6.4 Signals

In general, processes communicate via common variables, usually

declared within interface modules. However, it is not recommended to achieve synchronization by means of such common, shared variables. A delay of a process could in this way be realised only by a "busy waiting" statement, i.e. by polling. Instead, a facility called a signal should be used.

Signals are introduced in a program (usually within interface modules) like other objects. In particular, the syntactic form of its declaration is like that of a variable, although the signal is not a variable in the sense of having a value and being assignable. There are only two operations and a test that can be applied to signals. They are represented by three standard procedures.

1. The procedure call wait(s,r) delays the process until it receives the signal s. The process is given delay rank r, where r must be a positive valued integer expression. wait(s) is a short form for wait(s,1).

2. The procedure call send(s) sends the signal s to that process which had been waiting for s with least delay rank. If several processes are waiting for s with same delay rank, that process receives s which had been waiting longest. If no process is waiting for s, the statement send(s) has no effect.

3. The Boolean function procedure awaited(s) yields the value true, if there is at least one process waiting for signal s, false otherwise.

If a process executes a wait statement within an interface procedure, then other processes are allowed to execute other such procedures, although the waiting process has not completed his interface procedure. If a send statement is executed within an interface procedure, and if the signal is sent to a process waiting within the same interface module, then the receiving process obtains control over the module and the sending process is delayed until the other process has completed its interface procedure. Hence, both the wait and send operations must be considered as "singular points" or enclaves in the interface module, which are exempted from the mutual exclusion rule.

If a signal variable is exported from a module, then no send operations can be applied to it outside the module.

Examples of interface modules with signal operations [5]:

```
interface module resourcereservation;
   define semaphore,P,V,init;
   type semaphore = record taken: Boolean;
                           free: signal
                    end;
   procedure P(var s: semaphore);
```

```
        begin if s.taken then wait(s.free) end;
              s.taken := true
        end P;

    procedure V(var s: semaphore);
        begin s.taken := false;
              send(s.free)
        end V;

    procedure init(var s: semaphore);
        begin s.taken := false
        end init;
end resourcereservation


interface module bufferhandling;
    define get,put,empty;
    const nmax = 256;
    var n,in,out: integer;
        nonempty, nonfull: signal;
        buf: array 1: nmax of char;

    procedure empty: Boolean;
        begin empty := n = 0
        end empty;

    procedure put(ch: char);
        begin if n = nmax then wait(nonfull) end;
              inc(n);
              buf[in] := ch; in := (in mod nmax)+1;
              send(nonempty)
        end put;

    procedure get(var ch: char);
        begin if n = 0 then wait(nonempty) end;
              dec(n);
              ch := buf[out]; out := (out mod nmax)+1;
              send(nonfull)
        end get;

    begin n := 0; in := 1; out := 1
    end bufferhandling

interface module diskheadscheduler;
    define request,release;
    use cylmax;    (*no. of cylinders*)
    var headpos: integer;
        up,busy: Boolean;
        upsweep, downsweep: signal;

    procedure request(dest: integer);
    begin
        if busy then
```

```
            if headpos < dest then wait(upsweep,dest)
               else wait(downsweep,cylmax-dest) end;
         busy := true; headpos := dest
      end request;

      procedure release;
      begin busy := false;
         if up then
            if awaited(upsweep) then send(upsweep)
               else up := false; send(downsweep)
            end else
            if awaited(downsweep) then send(downsweep)
               else up := true; send(upsweep)
            end
         end
      end release;

begin headpos := 0; up := true; busy := false
end diskheadscheduler
```

7. PDP-11 SPECIFIC FACILITIES

All language facilities described in section 5 and 6 are defined without reference to a specific computer, i.e. they are defined by this report alone. This is not the case for the additional facilities introduced in this chapter, for they refer to features particular to the PDP-11 computer family, and can only be fully understood by referring to a PDP-11 description. They represent that computer's features for communicating with peripheral devices. These language facilities are available to the programmer only within modules specially designed as device modules.

7.1 Device modules and processes

A device module is an interface module that interfaces one or more so-called device processes - also called drivers - with other processes - also called "regular" processes. A device process is a process that contains operations that activate (drive) a peripheral device, and its heading is marked by the prefix symbol device. Whereas regular processes are declared outside interface modules and interact via procedures declared within the interface module, device processes are entirely declared within the interface module (and hence need not be especially distinguished by a mark or symbol). They, and only they, may contain a statement denoted by the identifier doio. While executing this statement, the process relinquishes exclusive access to the module's variables (as in the case of wait and send). The doio statement represents that part of the device process that is executed by the peripheral device. Usually it is preceded by some statement initiating the device

operation by accessing a device register.

The PDP-11 processor operates at a certain priority level. According to this level, interrupts from devices at lower levels are disabled and saved until the processor drops its level and "returns to duties of lower priority". The integer in the module heading specifies that level ($4 \leq L \leq 6$), and signifies that all procedures and processes defined in this module are executed with this processor priority. The programmer is advised to include in a device module only operations on devices that have exactly that interrupt priority.

priority = "[" integer "]" .

If a device process sends a signal to a process of lower priority, then the signalling process continues until it encounters a wait or a doio statement. This is an exception of the rule given in 6.4, which specifies that the signalled process continues. Regular processes have priority 0.

All processes defined within a device module are device processes, and each such process is associated with a so-called interrupt vector, i.e. with all devices that are interrupting to one and the same store location. The address of that location (interrupt vector) is to be specified in the device process heading (also enclosed in brackets). Interrupts must be disabled during the execution of wait statements. Two examples of device modules are given in Section 7.2.

intvector = "[" integer "]" .

Restrictions:
1. Device processes must not send signals to other device processes.
2. Device processes must not call any nonlocal procedures.
3. Only a single instance of a device process can be activated. Device processes are not "reentrant".
4. Wait statements within device processes must not specify a rank.

7.2 Device register declarations

Register declarations serve to introduce interface registers that are needed to communicate with peripheral devices. In the PDP-11 each device is associated with one or several registers. These registers have fixed store addresses which are to be specified in register declarations.

A register appears in a Modula program as a variable of the basic type specified in its declaration. Hence registers are also declared like variables by a variable declaration. Status registers are usually declared to be of type bits, whereas

buffer registers are usually of type integer or char.

The address of a register is prescribed by the hardware and it is specified immediately following the identifier and is enclosed in brackets. Hence, the syntax of variable declarations within a device module is slightly extended as follows:

```
variabledeclaration = ident [address] {"," ident [address]}
      ":" type .
address = "[" integer "]".
```

Examples:
The following module defines two procedures, readch and writech, which input a character from the typewriter keyboard and output a character to its printer. Both routines communicate with the devices via device processes and data buffers.

```
device module typewriter [4];
   define readch,writech
   const n = 64;    (*buffer size*)
   var KBS [177560B]: bits;    (*keyboard status*)
       KBB [177562B]: char;    (*keyboard buffer*)
       PRS [177564B]: bits;    (*printer  status*)
       PRB [177566B]: char;    (*printer  buffer*)
       in1,in2,out1,out2: integer;
       n1,n2: integer;
       nonfull1,nonfull2,nonempty1,nonempty2: signal;
       buf1,buf2: array 1: n of char;

   procedure readch(var ch: char);
   begin
        if n1 = 0 then wait(nonempty1) end;
        ch := buf1[out1]; out1 := (out1 mod n) + 1;
        dec(n1); send(nonfull1)
   end readch;

   procedure writech(ch: char);
   begin
        if n2 = n then wait(nonfull2) end;
        buf2[in2] := ch; in2 := (in2 mod n) + 1;
        inc(n2); send(nonempty2)
   end writech;

   process keyboarddriver [60B];
   begin
        loop
            if n1 = n then wait(nonfull1) end ;
            KBS[6] := true; doio; KBS[6] := false;
            buf1[in1] := KBB; in1 := (in1 mod n) + 1;
            inc(n1); send(nonempty1)
        end
   end keyboarddriver;
```

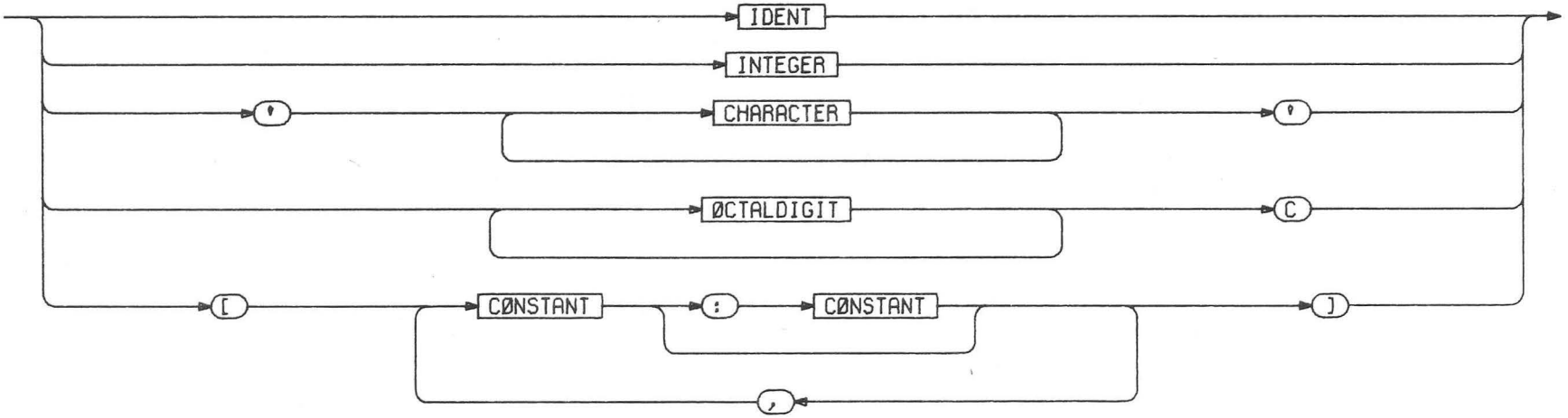
```
   process printerdriver [64B];
   begin
        loop
            if n2 = Ø then wait(nonempty2) end;
            PRB := buf2[out2]; out2 := (out2 mod n) + 1;
            PRS[6] := true; doio; PRS[6] := false;
            dec(n2); send(nonfull2)
        end
   end printerdriver;

begin in1 := 1; in2 := 1; out1 := 1; out2 := 1;
      n1 := Ø; n2 := Ø;
      keyboarddriver; printerdriver
end typewriter
```

The following module defines a variable time that is incremented every 2Ø msec, a signal tick that is sent every 2Ø msec, and a procedure pause(n) which delays the calling process by n*2Ø msec.

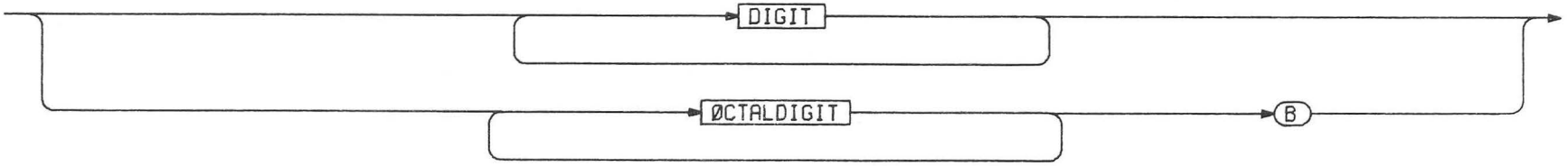
```
device module realtime [6];
   define time, tick, pause;
   var time: integer; tick: signal;
       LCS [177546B]: bits;    (*Line Clock Status*)

   procedure pause(n: integer);
      var delay: integer;
   begin delay := n;
      while delay > Ø do
         wait(tick); dec(delay)
      end
   end pause ;

   process clock [1ØØB];
   begin LCS[6] := true;
      loop doio; inc(time);
         while awaited(tick) do send(tick) end
      end
   end clock ;

begin time := Ø; clock
end realtime
```

According to Restriction 1 (7.1), other device processes can neither wait for the signal tick, nor can they call the procedure pause.

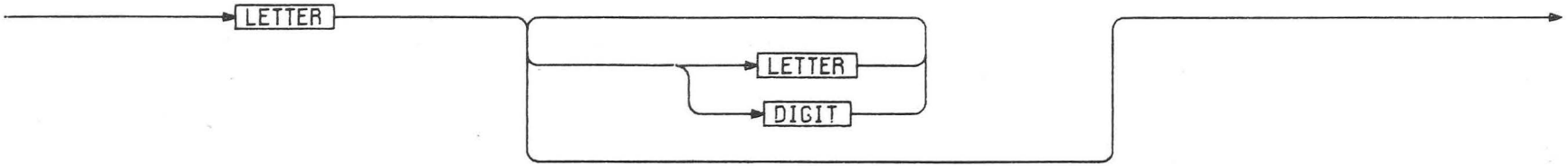
UNSIGNED CØNSTANT



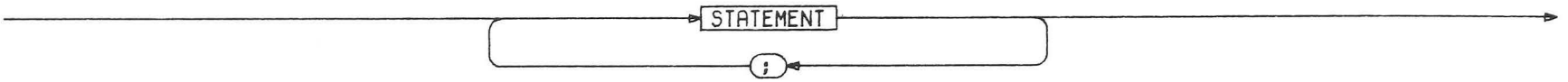
INTEGER



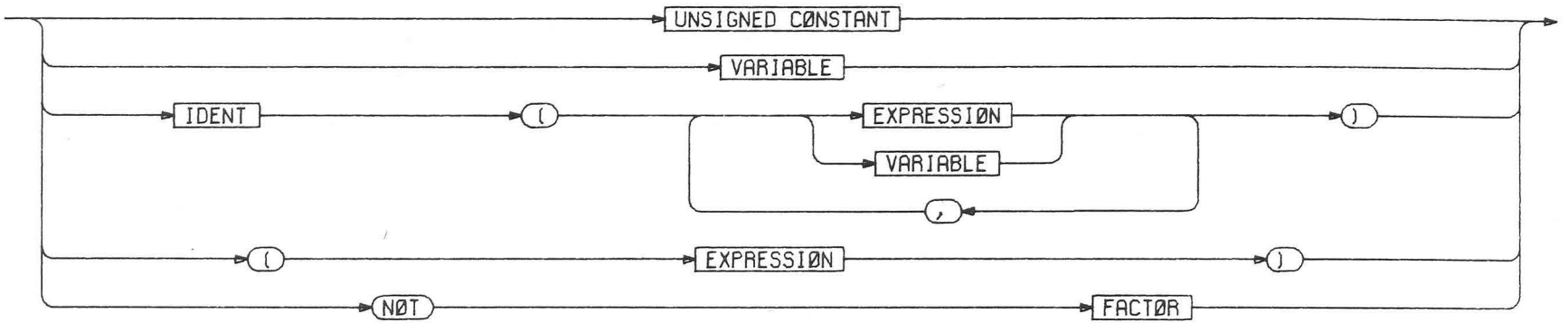
IDENT



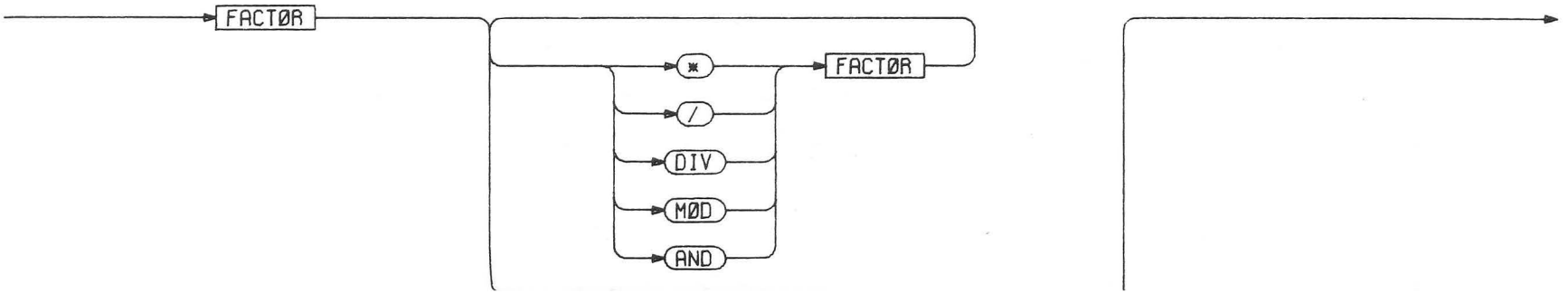
STATEMENTSEQUENCE



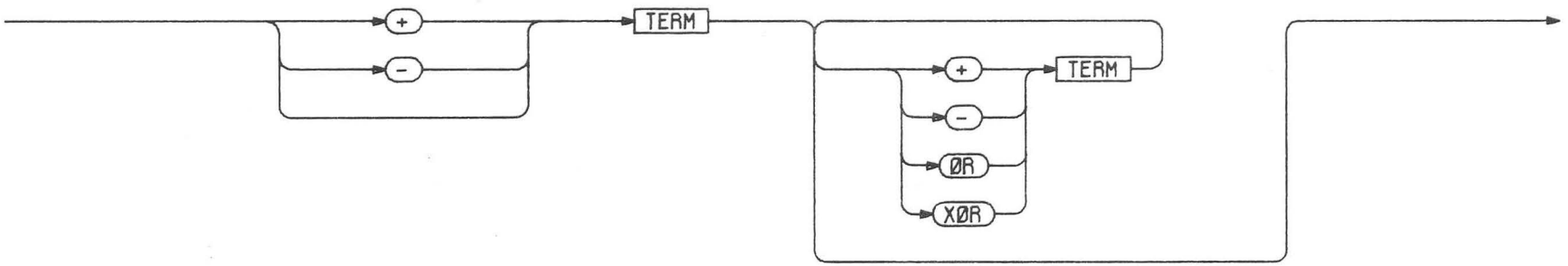
FACTØR



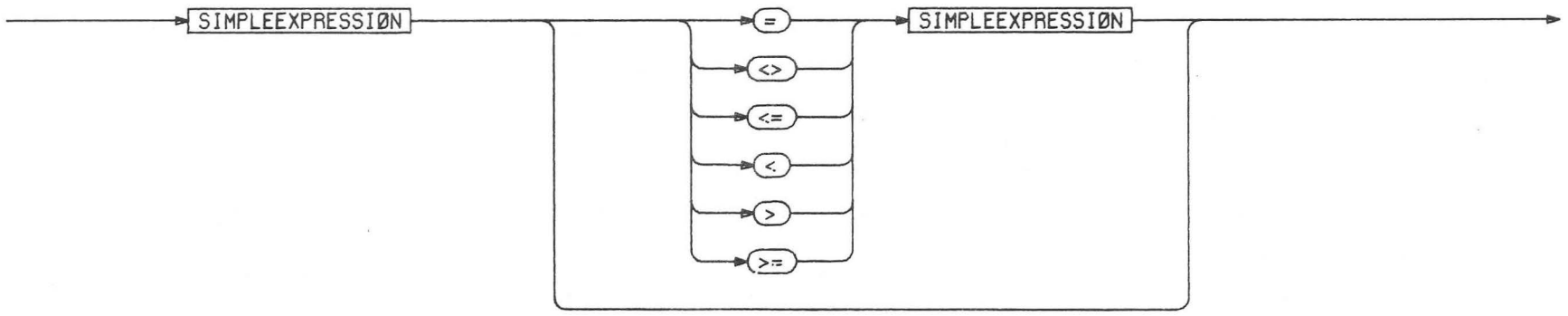
TERM



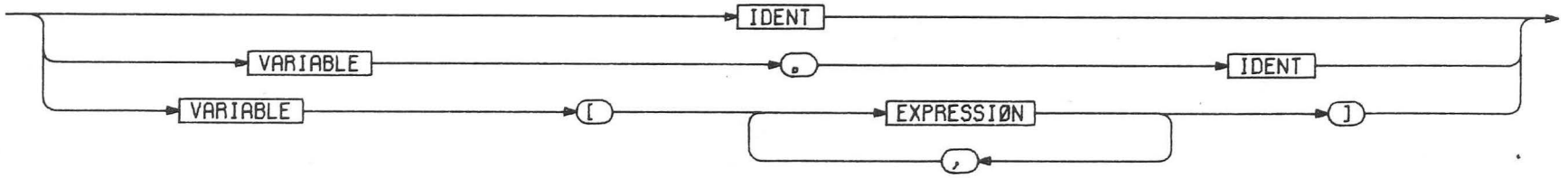
SIMPLEEXPRESSIØN



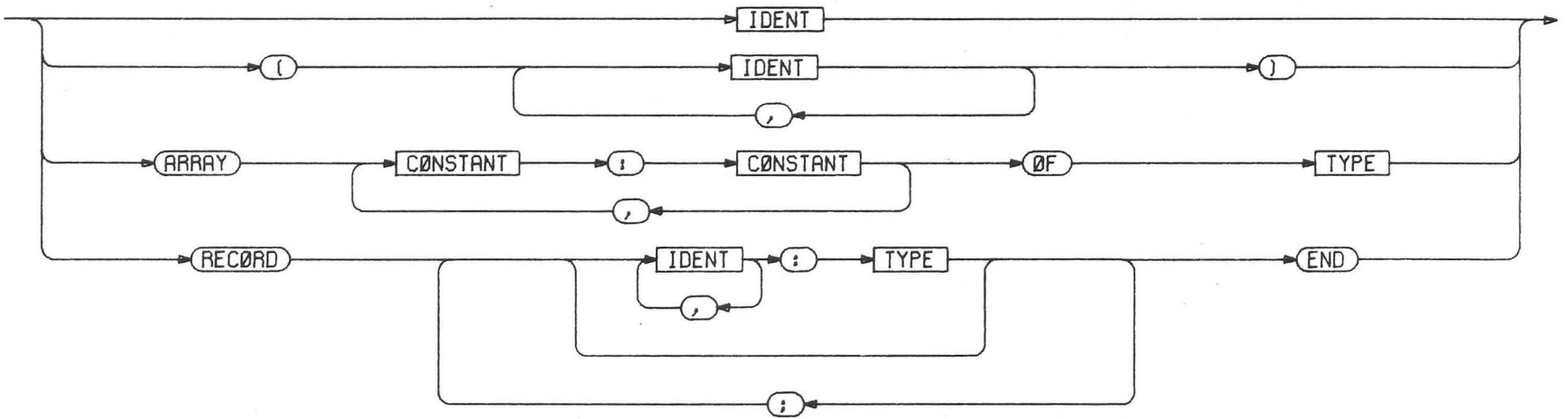
EXPRESSIØN



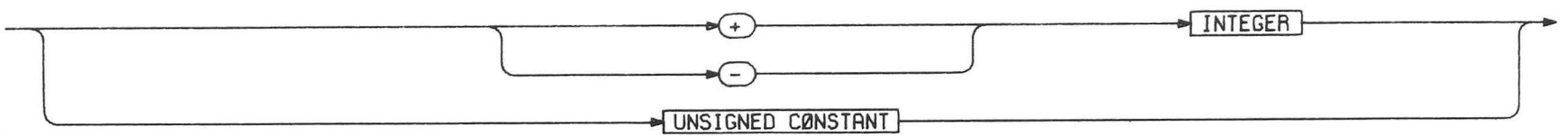
VARIABLE



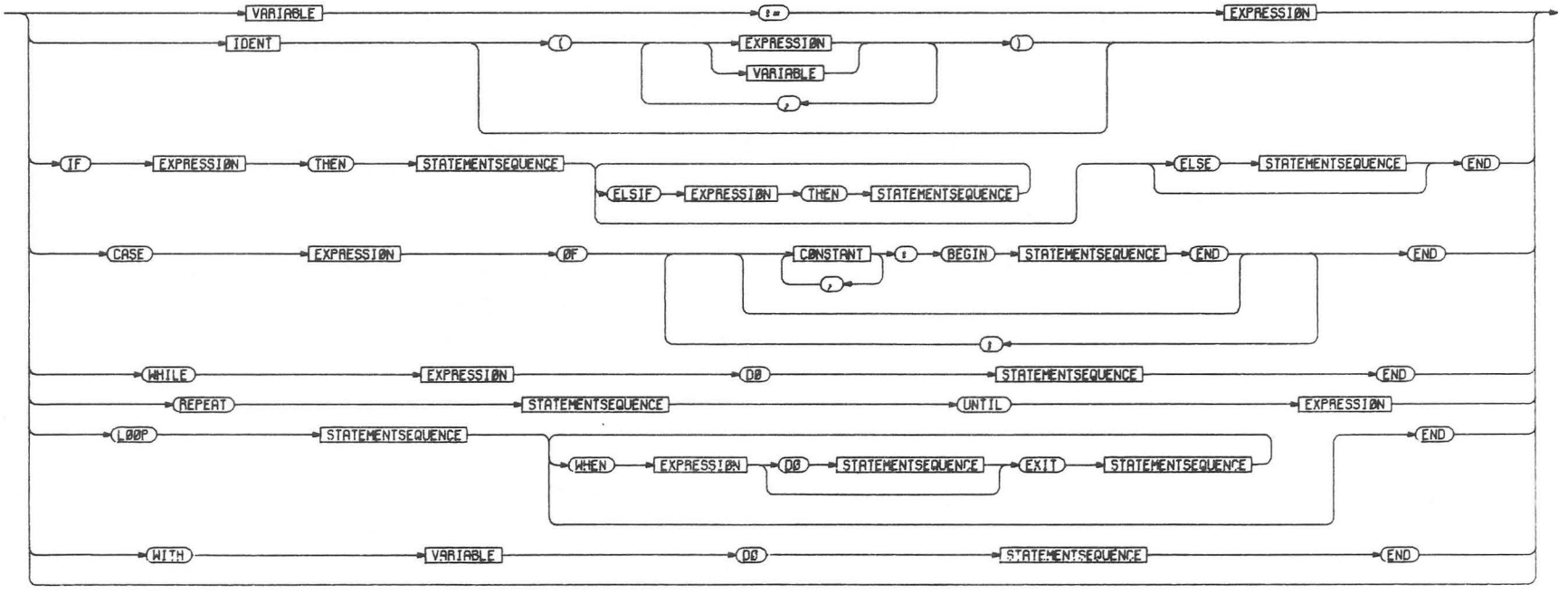
TYPE



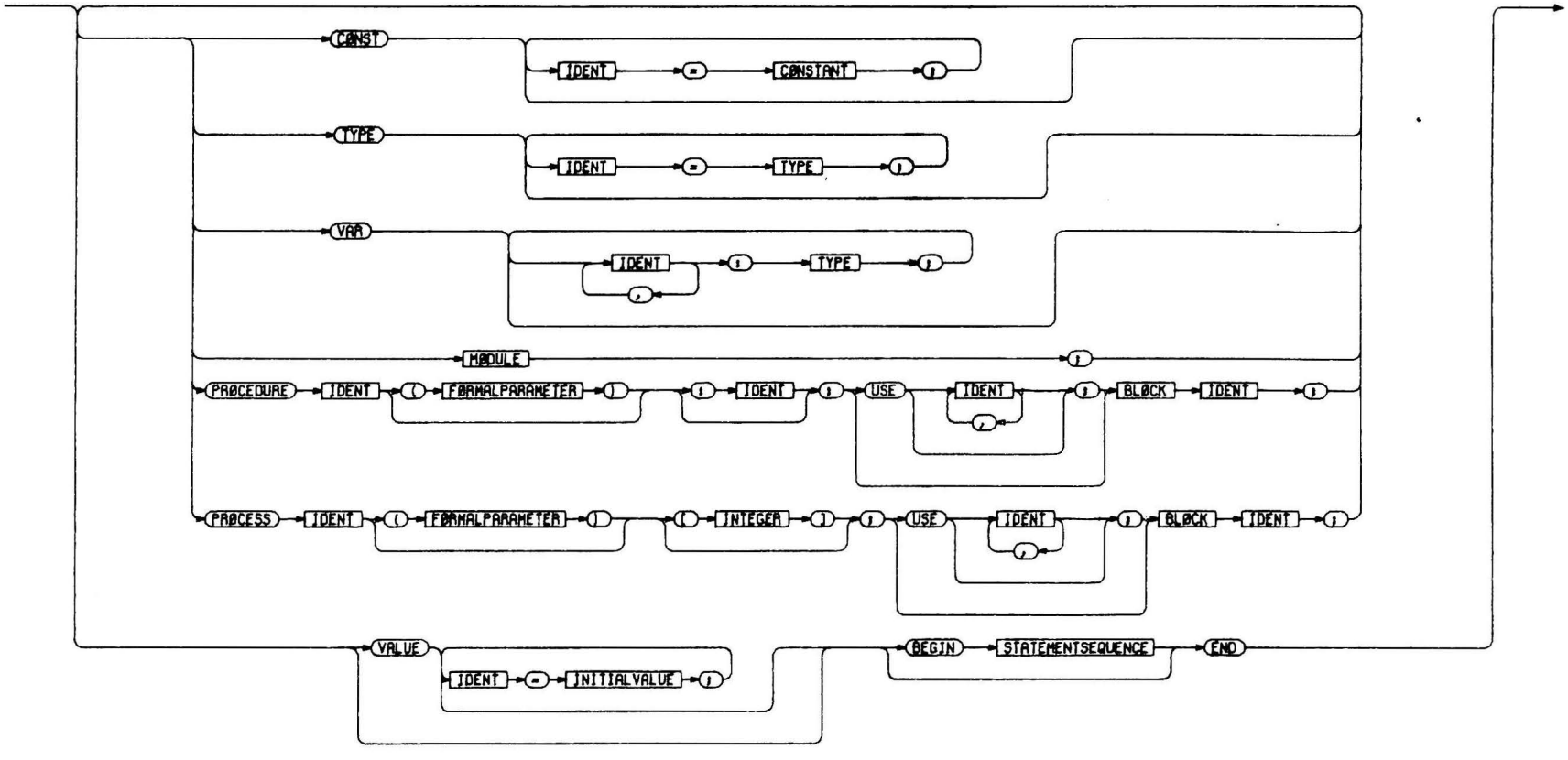
CØNSTANT

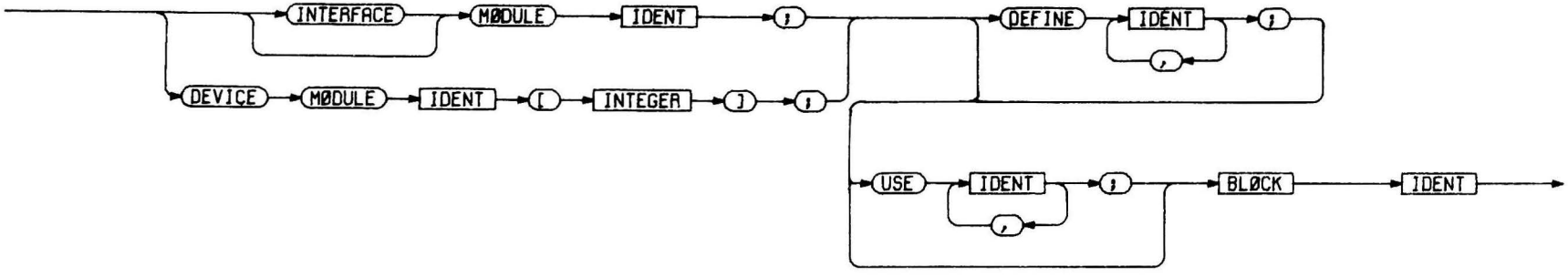


STATEMENT

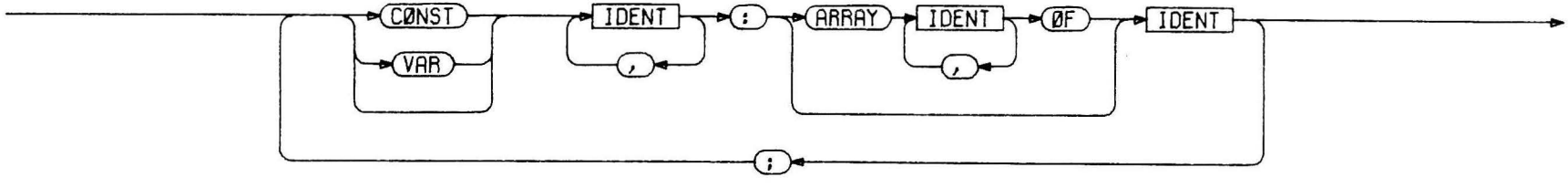

VARIABLE
:=
EXPRESSIØN
IDENT
(
EXPRESSIØN
VARIABLE
,
)
IF
EXPRESSIØN
THEN
STATEMENTSEQUENCE
ELSIF
EXPRESSIØN
THEN
STATEMENTSEQUENCE
ELSE
STATEMENTSEQUENCE
END
CASE
EXPRESSIØN
ØF
CØNSTANT
,
:
BEGIN
STATEMENTSEQUENCE
END
END
|
WHILE
EXPRESSIØN
DØ
STATEMENTSEQUENCE
END
REPEAT
STATEMENTSEQUENCE
UNTIL
EXPRESSIØN
LØØP
STATEMENTSEQUENCE
WHEN
EXPRESSIØN
DØ
STATEMENTSEQUENCE
EXIT
STATEMENTSEQUENCE
END
WITH
VARIABLE
DØ
STATEMENTSEQUENCE
END

BLØCK

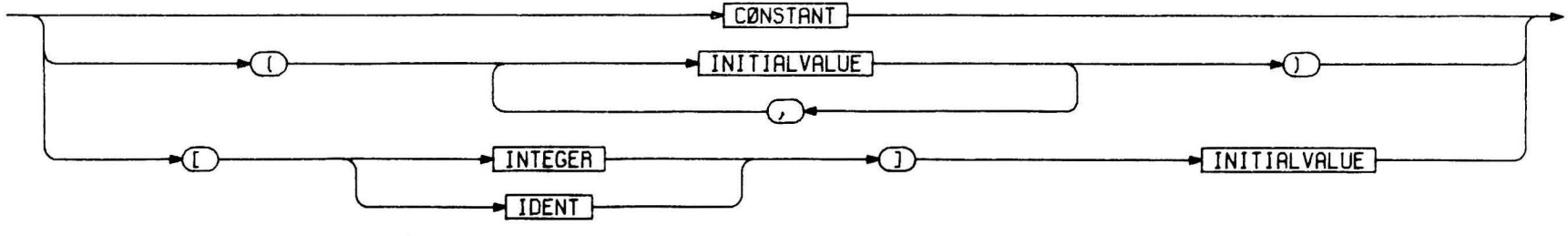

CØNST
IDENT
=
CØNSTANT
;
TYPE
IDENT
=
TYPE
;
VAR
IDENT
,
:
TYPE
;
MØDULE
;
PRØCEDURE
IDENT
(
FØRMALPARAMETER
)
:
IDENT
;
USE
IDENT
,
;
BLØCK
IDENT
;
PRØCESS
IDENT
(
FØRMALPARAMETER
)
(
INTEGER
)
;
USE
IDENT
,
;
BLØCK
IDENT
;
VALUE
IDENT
=
INITIALVALUE
;
BEGIN
STATEMENTSEQUENCE
END



FØRMALPARAMETER



INITIALVALUE



References

1. Brinch Hansen, P., Concurrent Pascal Report, Calif. Inst. of Technology, June 1975
2. ----- The Solo operating system, Calif. Inst. of Technology, July 1975
3. Brooks, F.P. Jr., The mythical man month, Addison-Wesley, Reading, Mass. 1975
4. Dahl, O.-J., Myhrhaug, B., Nygaard, K., The SIMULA 67 common base language. Norwegian Comp. Center, Oslo 1968.
5. Hoare, C.A.R., Monitors: An operating system structuring concept. Comm.ACM 17,10,549-557 (Oct. 1974)
6. Jensen,K., and Wirth, N., PASCAL - User manual and report. Springer-Verlag, 1974/5.
7. Parnas, D.L., Information distribution aspects of design methodology. IFIP Congress 71, Booklet TA-3, pp.26-30.
8. Sandmayr, H., Strukturen und Konzepte zur Multiprogrammierung und ihre Anwendung auf ein System für Datenstationen (Hexapus). ETH-Dissertation 5537.
9. Wirth, N., On multiprogramming, machine coding, and computer organization, Comm.ACM 12,9,489-498 (Sept.1969)
10. ---- The programming language Pascal, Acta Informatica 1, 35 - 63, (1971)

Acknowledgement

I wish to thank U.Ammann, J.Hoppe, V.K.Le, and R.Schoenberger for their contributions to the experimental Modula implementation. Thanks are due to H.Sandmayr for many valuable suggestions, and to J.Spillmann for the preparation of the program to draw syntax diagrams automatically.

Berichte des Instituts für Informatik

Nr. 1	Niklaus Wirth:	The Programming Language Pascal (out of print)
Nr. 2	Niklaus Wirth:	Program development by step-wise refinement (out of print)
Nr. 3	Peter Läuchli:	Reduktion elektrischer Netzwerke und Gauss'sche Elimination
Nr. 4	Walter Gander, Andrea Mazzario:	Numerische Prozeduren I
Nr. 5	Niklaus Wirth:	The Programming Language Pascal (Revised Report) (out of print)
Nr. 6	C.A.R. Hoare, Niklaus Wirth:	An Axiomatic Definition of the Language Pascal (out of print)
Nr. 7	Andrea Mazzario, Luciano Molinari:	Numerische Prozeduren II
Nr. 8	E. Engeler, E. Wiedmer, E. Zachos:	Ein Einblick in die Theorie der Berechnungen
Nr. 9	Hans-Peter Frei:	Computer Aided Instruction: The Author Language and the System THALES (out of print)
Nr.10	K.V. Nori, U. Ammann, K.Jensen, H.H. Nägeli:	The PASCAL 'P' Compiler: Implementation Notes
Nr.11	G.I. Ugron, F.R. Lüthi:	Das Informations-System ELSBETH
Nr.12	Niklaus Wirth:	PASCAL-S: A Subset and its Implementation
Nr.13	U. Ammann:	Code Generation in a PASCAL Compiler
Nr.14	Karl Lieberherr:	Toward Feasible Solutions of NP-Complete Problems
Nr.15	E. Engeler:	Structural Relations between Programs and Problems
Nr.16	W. Bucher:	A contribution to solving large linear systems
Nr.17	Niklaus Wirth:	Programming languages: what to demand and how to assess them and Professor Cleverbyte's visit to heaven
Nr.18	Niklaus Wirth:	MODULA: A language for modular multiprogramming