ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

Niklaus Wirth

MODULA-2

Dezember 1978 27

MODULA-2

N. Wirth

Abstract

Modula-2 is a general purpose programming language primarily designed for systems implemenation. This report constitutes its definition in a concise, although informal style.

Institut fur Informatik
ETH
CH-8092 Zurich

December 1978

Note: no compiler is available for distribution at this time.

Please note that the pages 24/25, 30/31 and 34/35 have been changed by mistake.

Contents
---

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.

This report is neither intended as a programmer's manual nor as an implementation tutorial. It is intentionally kept concise, brief, 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.

We reserve the right to extend or even change the language in areas where issues are as yet unresolved and experience in use of the language may provide new insight. This is in particular the case in the domains of definition modules, export of names, and low-level facilities.

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

exclusively of capital letters and MUST NOT be used in the role of identifiers.

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

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. In the case of types, however, it extends from the declaration itself to the end of the block. The scope rules are modified in the following cases:

1. If the object is local to a module and is exported, the scope is extended over that part of the block or module to which the identifier is exported and, for types, which textually follows the exporting module.
2. 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.
3. 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 (module).

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

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

The following are standard identifiers:

ABS	(10.2)	INC	(10.2)
ADR	(10.2)	INCL	(10.2)
ASH	(10.2)	INTEGER	(6.1)
BITSET	(6.6)	HALT	(10.2)
BOOLEAN	(6.1)	HIGH	(10.2)
CAP	(10.2)		
CARDINAL	(6.1)	NEW	(10.2)
CHAR	(6.1)	NIL	(6.7)
DEC	(10.2)	ODD	(10.2)
DISPOSE	(10.2)	SIZE	(10.2)
EXCL	(10.2)	STRING	(10.1)
FALSE	(6.1)	TRUE	(6.1)

5. Constants

A constant declaration associates an identifier with a constant value.

```
$   ConstantDeclaration = ident "=" constant.
$   constant = qualident | ["+"|"-"] number | string | set.
$   set = [qualident] "{" [element {"," element}] "}".
$   element = constant [".." constant].
```

Every constant is said to be of a certain type. Non-negative integers are of type CARDINAL (see 6.1), negative integers of type INTEGER. A single-character string is of type CHAR, a string consisting of n>1 characters is of type (see 6.4)

```
ARRAY 0..N OF CHAR      (where N = n-1).
```

In the case of sets the identifier preceding the left brace specifies the type of the set. If it is omitted, the standard type BITSET is assumed (see 6.6).

6. Types

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 -32768 and +32767.
CARDINAL	A variable of type CARDINAL assumes as values the integers between 0 and 65535.
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 character set.

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, non-structured type T1 by specification of the least and the highest value in the subrange.

```
$   SubrangeType = constant ".." constant.
```

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 T, if either T1 is equal to T, or if T1 or T (or both) are subranges of the same (base) type.

Examples of subrange types:

```
        Ø .. 99
      "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 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 Ø..99 OF CARDINAL
ARRAY 1..1Ø, 1..2Ø OF Ø..99
ARRAY -1Ø..+1Ø 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 (cf. 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 = constant [".." constant].
```

Examples of record types:

```
RECORD day: 1..31;
       month: 1..12;
       year: Ø..2ØØØ
END

RECORD
   name,firstname: ARRAY Ø..9 OF CHAR;
   age: Ø..99;
   sex: (male, female)
END

RECORD x,y: TØ;
   CASE tagØ: Color OF
      red:   a: Tr1; b: Tr2 |
      green: c: Tg1; d: Tg2 |
      blue:  e: Tb1; f: Tb2
   END;
   z: TØ;
   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 tagØ, the one of the second section by the tag field tagl.

```
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 Ø and 15, or a (subrange of an) enumeration type with at most 16 values.

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

The following type is standard:

```
BITSET = SET OF Ø .. 15
```

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 1Ø.1).

```
$   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].
```

7. Variables

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 Sect.6):

```
i,j: INTEGER
p,q: BOOLEAN
s,t: BITSET
F:   Function
a:   ARRAY Index OF CARDINAL
w:   ARRAY Ø..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 Sect.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 Sect. 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 compatible with the type of E. 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 Sect. 10).

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

```
i               (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].
$   relation = "=" | "#" | "<=" | "<" | ">" | ">=" | IN .
$   SimpleExpression = ["+"|"-"] term {AddOperator term}.
$   AddOperator = "+" | "-" | OR .
$   term = factor {MulOperator factor}.
```

```
$   MulOperator = "*" | "/" | DIV | MOD | AND | "&".
$   factor = number | string | set | designator [ActualParameters] |
$        "(" expression ")" | NOT factor.
$   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. Integer operators

symbol	operation
+	addition
-	subtraction
*	multiplication
DIV	division
MOD	modulus

These operators apply to operands of type INTEGER, CARDINAL, or subranges thereof. Both operands must be either of type CARDINAL or a subrange with non-negative lower bound, in which case the result is of type CARDINAL, or they must both be of type INTEGER or a subrange with a negative lower bound, in which case the result is of type INTEGER. If (at least) one operand is a constant in the range Ø <= c < 32768, the type of the constant is taken as that of the other operand.

When used as operators with a single operand only, - denotes sign inversion and + denotes the identity operation.

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. The symbols & and AND are synonyms.

```
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, to enumerations, and to subrange types thereof.

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 Sect.7):

```
1978                        (CARDINAL)
i DIV j                     (INTEGER)
NOT p OR q                  (BOOLEAN)
(i+j) * (i-j)               (INTEGER)
(s*t) - {8,9,13}            (BITSET)
a[i] + a[j]                 (CARDINAL)
a[i+j] * a[i-j]             (CARDINAL)
(Ø<=i) & (i<1ØØ)            (BOOLEAN)
t^.key = Ø                  (BOOLEAN)
{13..15} <= s               (BOOLEAN)
i IN {Ø, 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 compatible with the type of the expression. CARDINAL and INTEGER are considered as compatible under assignment.

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 (ØC).

Examples of assignments:

```
          i := Ø
          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 (cf. 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. 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 equal in the case of variable parameters and compatible in the case of value parameters.

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

Examples of procedure calls:

```
        ReadInteger(i)              (see Sect.10)
        WriteInteger(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 not be structured. 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 x > 0 DO
  x := x DIV 2; i := i+1
END

WHILE a # b DO
  IF a > b THEN a := a-b
           ELSE b := b-a
  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 = Ø
```

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.

```
$   ForStatement = FOR ident ":=" expression TO expression
$       [BY constant] 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 surpassing B. v is called the control variable, A the starting value, B the limit, and C the increment. A, B, and v must be of compatible types; C must be a (possibly signed) integer. If no increment is specified, it is assumed to be 1.

Examples:

```
FOR i := Ø TO 99 DO  s := s+a[i]  END
FOR i := 99 TO 1 BY -1 DO a[i] := a[i-1] END
```

9.9. Loop statements

A loop statements 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.

```
$   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. E specifies the value returned as result of a function procedure, and its type must be the result type specified in the procedure heading (see Sect. 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. 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 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 the program 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

evaluating 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 the same as that of its corresponding actual parameter (see Sect. 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 Ø 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. The standard formal type STRING is defined as

```
STRING = ARRAY OF CHAR
```

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 Ø or a variable (or parameter) of that procedure type. It cannot be a standard procedure.

Examples of procedure declarations:

```
PROCEDURE ReadInteger(VAR x: CARDINAL);
  VAR i: CARDINAL; ch: CHAR;
BEGIN i:=Ø;
  REPEAT ReadChar(ch)
  UNTIL (ch >= "Ø") & (ch <= "9");
  REPEAT i := 1Ø*i + (INTEGER(ch)-INTEGER("Ø"));
    ReadChar(ch)
  UNTIL (ch < "Ø") OR (ch > "9");
  x := i
END ReadInteger

PROCEDURE WriteInteger(x,n: CARDINAL);
  VAR i,q: CARDINAL;
```

```
      buf: ARRAY 1..10 OF CARDINAL;
BEGIN i := 0; q := x;
  REPEAT i := i+1; buf[i] := q MOD 10; q := q DIV 10
  UNTIL q = 0;
  WHILE n > i DO
    WriteChar(" "); DEC(n)
  END ;
  REPEAT WriteChar(buf[i]); DEC(i)
  UNTIL i = 0
END WriteInteger


PROCEDURE log2(x: CARDINAL): CARDINAL;
  VAR y: CARDINAL;
BEGIN x := x-1; y := 0;
  WHILE x > 0 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
ASH(x,n)	n >= 0: x * (2**n), n < 0: x DIV (2**(-n))
CAP(ch)	if ch is a lower case letter, the corresponding capital letter; if ch is a capital letter, the same letter
HIGH(a)	high index bound of array a
ODD(x)	x MOD 2 # 0
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

NEW(p)	=	SYSTEM.ALLOCATE(p,S)
DISPOSE(p)	=	SYSTEM.DEALLOCATE(p,S)
NEW(p,t1,t2, ...)	=	SYSTEM.ALLOCATE(p,S)
DISPOSE(p,t1,t2, ...)	=	SYSTEM.DEALLOCATE(p,S)

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 abbreviations for

Standard identifiers are always imported automatically. As a consequence, standard identifiers can be redeclared in procedures only, but not in modules, including the main program.

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; lno := Ø
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
```

procedures defined in a module called SYSTEM (see also Sect. 13); S denotes the size of the variables referenced by the pointer p, and t1, t2, ... are possible tag field values, if the referenced variable has a variant record structure.

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.

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 m = 64; w = 16;     (*there are m*w tracks*)
  VAR i: CARDINAL;
      free: ARRAY Ø..63 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=Ø)
    UNTIL found OR (i=Ø);
    IF found THEN EXCL(free[i],j); RETURN i*w+j
            ELSE RETURN -1
    END
  END NewTrack;

  PROCEDURE ReturnTrack(k: CARDINAL);
  BEGIN (*assume Ø <= k < m*w *)
    INCL(free[k DIV w], k MOD w)
  END ReturnTrack;

BEGIN (*mark all tracks free*)
  FOR i := Ø TO m-1 DO free[i] := {Ø..15} END
END TrackReservation
```

12. Programs

A unit of program text which is accepted by the compiler is called a program. It has the form of a module declaration and is terminated by a period.

```
$   program = [DEFINITION | ident] ModuleDeclaration "." .
```

It is possible to refer from one program module to other modules according to the import/export rules of modules. Program modules, however, must specify qualified export.

A program may be prefixed by the symbol DEFINITION or by the identifier of a definition module. The former case constitutes a so-called definition module, the latter a so-called implementation module. A definition module contains declarations only. More specifically, it contains constant, type, and variable declarations, and procedure headings. Also, the statement part

must be empty. (Note: these rules are not reflected by the syntax).

A module M1 prefixed with an identifier M of a definition module is said to implement (part of) M. M1 contains declarations of those procedures whose headings appear in the definition module M. All declarations of M are imported into M1, even if not mentioned in the export list of M.

A definition module is likely to be used in those cases where several, (perhaps separately compiled) modules are to be implementing the definition module, and where the definition module serves as a central basis and binding contract between the various implementors.

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 served (executed) by a (single) processor one at a time.

13.1. Creating a process, and transfer of control

A new process is created by the procedure call

```
NEWPROCESS(P,A,n,p1)
```

P denotes the procedure which constitutes the process,
A is the base address of the process' workspace,
n is the size of this workspace, and
p1 a variable of type PROCESS

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

A transfer of control between two processes is specified by the call

```
TRANSFER(p1,p2)
```

where p1 and p2 are variables of type 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 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.

In the following example a procedure called Reply is defined. It is used in a program with two processes in order to transmit "messages". One process is represented by the main program, the other is generated by a call to CallPartner.

```
MODULE Conversation;
  FROM SYSTEM IMPORT
    PROCESS, PROC, NEWPROCESS, TRANSFER;
  EXPORT CreatePartner, Reply;

  VAR spr: PROCESS;  (*suspended process*)
      msg: CARDINAL;
      wsp: ARRAY Ø .. 99 OF WORD; (*workspace*)

  PROCEDURE CallPartner(P: PROC): CARDINAL;
  BEGIN NEWPROCESS(P, ADR(wsp), SIZE(wsp), spr);
    TRANSFER(spr, spr); RETURN msg
  END CallPartner;

  PROCEDURE Reply(x: CARDINAL): CARDINAL;
  BEGIN msg := x; TRANSFER(spr, spr); RETURN msg
  END Reply;
END Conversation
```

13.2. Peripheral devices and interrupts

Control of and communication with peripheral devices differs not only between devices, but in particular between different computer systems. The facilities described in this paragraph are specific for the PDP-11.

Devices are controlled via so-called device registers. They are specified in a program as variables, and their identity is determined by their absolute address. This address is indicated as an integer enclosed in brackets immediately following the identifier in the variable declaration. The choice of an appropriate data type is left to the programmer.

Example:

```
VAR TWB [177566B]: CHAR;  (*typewriter buffer*)
```

If a device is to be operating under interrupt control, then initiation of the device operation is achieved by specific assignments to the appropriate register. This is followed by a call

```
IOTRANSFER(p1,p2,va)
```

where p1 and p2 are variables of type PROCESS and va is the "interrupt vector" address of the device. This procedure must be imported from the module SYSTEM. The effect of the call is (in analogy to TRANSFER) to suspend the calling process, to assign it to p1, to transfer control to the process designated by p2, and additionally to prepare a return transfer to the calling process p1. This return transfer will be initiated by the interrupt signal emitted by the device associated with the interrupt vector at address va. The interrupt signal will suspend the current process, assign it to p2, and resume p1, from which the IO transfer had started.

13.3. Interface modules

Transfer of control between two processes occurs because of an intended, programmed interaction. Such interactions are normally accompanied by the transmittal of a message and/or the transfer of data via variables common to both processes. It is a recommended practice to group the declarations of such common variables (like buffers) together with the procedures operating on these variables in a module, and to program transfers of control within that module only. This module is then called an interface module.

If interrupts are utilized, the interface module has the additional and essential function to suppress all or some of the interrupts in order to prevent interference and to guarantee data integrity. In the case of the PDP-11, the heading of the module declaration may therefore specify a processor priority pp (namely 0 ... 7). This means that procedures declared within this module are executed with priority pp, effectively disabling interrupts from devices with interrupt level less than or equal to pp.

Processes that represent device handlers (interrupt routines) are usually declared fully within an interface module. It is the programmer's responsibility to ensure that its priority is that of the device as specified by the PDP-11 hardware.

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

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

  CONST N = 32;
  VAR n: 0..N;      (*no. of chars in buffer*)
      in, out: 1..N;
      B: ARRAY 1..N OF CHAR;
      PR: PROCESS; (*producer*)
      CO: PROCESS; (*consumer = typewriter driver*)
      wsp: ARRAY 0..20 OF WORD;
      TWS [177564B]: BITSET;    (*status register*)
```

14. Index

14.1 Syntactic terms

```
      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(PR,CO) END
  END typeout;

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

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

14.2 Semantic terms

*

15. Syntax summary

```
ident = letter {letter | digit}.
number = integer.
integer = digit {digit} | octalDigit {octalDigit} ("B"|"C").
string = "'" {character} "'" | '"' {character} '"' .
qualident = ident {"." ident}.
ConstantDeclaration = ident "=" constant.
constant = qualident | ["+"|"-"] number | string | set.
set = [qualident] "{" [element {"," element}] "}".
element = constant [".." constant].
TypeDeclaration = ident "=" type.
type = SimpleType | ArrayType | RecordType | SetType |
       PointerType | ProcedureType.
SimpleType = qualident | enumeration | SubrangeType.
enumeration = "(" IdentList ")".
IdentList = ident {"," ident}.
SubrangeType = constant ".." constant.
ArrayType = ARRAY SimpleType {"," SimpleType} OF type.
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 = constant [".." constant].
SetType = SET OF SimpleType.
PointerType = POINTER TO type.
ProcedureType = PROCEDURE [FormalTypeList].
FormalTypeList = "(" [[VAR] FormalType
                 {"," [VAR] FormalType}] ")" [":" qualident].
VariableDeclaration = IdentList ":" type.
designator = qualident {"." ident | "[" ExpList "]" | "↑"}.
ExpList = expression {"," expression}.
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "#" | "<=" | "<" | ">" | ">=" | IN .
SimpleExpression = ["+"|"-"] term {AddOperator term}.
AddOperator = "+" | "-" | OR .
term = factor {MulOperator factor}.
MulOperator = "*" | "/" | DIV | MOD | AND | "&".
factor = number | string | set | designator [ActualParameters] |
    "(" expression ")" | NOT factor.
ActualParameters = "(" [ExpList] ")" .
statement = [assignment | ProcedureCall |
    IfStatement | CaseStatement | WhileStatement |
    RepeatStatement | LoopStatement | ForStatement |
    WithStatement | EXIT | RETURN [expression] ].
assignment = designator ":=" expression.
ProcedureCall = designator [ActualParameters].
StatementSequence = statement {";" statement}.
IfStatement = IF expression THEN StatementSequence
    {ELSIF expression THEN StatementSequence}
    [ELSE StatementSequence] END.
CaseStatement = CASE expression OF case {"|" case}
                [ELSE StatementSequence] END.
case = CaseLabelList ":" StatementSequence.
WhileStatement = WHILE expression DO StatementSequence END.
RepeatStatement = REPEAT StatementSequence UNTIL expression.
```

	21	15	15	10	6	5	5	-1
import	-77	74						
integer	75	-3	2					
LoopStatement	-60	45						
letter	1	1						
ModuleDeclaration	78	-73	68					
MulOperator	-39	38						
number	40	7	-2					
octalDigit	3	3						
priority	-75	74						
program	-78							
PointerType	-27	12						
ProcedureCall	-48	43						
ProcedureDeclaration	68	-62						
ProcedureHeading	-63	62						
ProcedureType	-28	12						
qualident	72	70	32	30	21	13	8	7
	-5							
relation	-35	34						
RecordType	-18	11						
RepeatStatement	-57	45						
SetType	-26	11						
SimpleExpression	-36	34	34					
SimpleType	26	17	17	-13	11			
StatementSequence	64	61	60	59	57	56	55	54
	52	51	50	-49				
SubrangeType	-16	13						
set	40	-8	7					
statement	49	49	-43					
string	40	7	-4					
term	-38	36	36					
type	31	27	20	17	-11	10		
TypeDeclaration	66	-10						
VariableDeclaration	67	-31						
variant	-23	21	21					
WhileStatement	-56	44						
WithStatement	-61	46						

```
58    ForStatement = FOR ident ":=" expression TO expression
59        [BY constant] DO StatementSequence END.
60    LoopStatement = LOOP StatementSequence END.
61    WithStatement = WITH designator DO StatementSequence END .
62    ProcedureDeclaration = ProcedureHeading ";" block ident.
63    ProcedureHeading = PROCEDURE ident [FormalParameters].
64    block = {declaration} [BEGIN StatementSequence] END.
65    declaration = CONST {ConstantDeclaration ";"} |
66        TYPE {TypeDeclaration ";"} |
67        VAR {VariableDeclaration ";"} |
68        ProcedureDeclaration ";" | ModuleDeclaration ";".
69    FormalParameters =
70              "(" [FPSection {";" FPSection}] ")" [":" qualident].
71    FPSection = [VAR] IdentList ":" FormalType.
72    FormalType = [ARRAY OF] qualident.
73    ModuleDeclaration =
74        MODULE ident [priority] ";" {import} [export] block ident.
75    priority = "[" integer "]".
76    export = EXPORT [QUALIFIED] IdentList ";".
77    import = [FROM ident] IMPORT IdentList ";".
78    program = [DEFINITION | ident] ModuleDeclaration "." .
```

```
assignment              -47    43
ActualParameters         48   -42    40
AddOperator             -37    36
ArrayType               -17    11
block                    74   -64    62
case                    -55    53    53
CaseLabelList            55   -24    23
CaseLabels              -25    24    24
CaseStatement           -53    44
ConstantDeclaration      65    -6
character                 4     4
constant                 59    25    25    16    16     9     9    -7
                          6
declaration             -65    64
designator               61    48    47    40   -32
digit                     3     3     1
ExpList                  42   -33    32
element                  -9     8     8
enumeration             -14    13
export                  -76    74
expression               58    58    57    56    53    51    50    47
                         46    41   -34    33    33
factor                   41   -40    38    38
FieldList               -20    19    19
FieldListSequence        23    22   -19    18
FormalParameters        -69    63
FormalType              -72    71    30    29
FormalTypeList          -29    28
ForStatement            -58    45
FPSection               -71    70    70
IdentList                77    76    71    31    20   -15    14
IfStatement             -50    44
ident                    78    77    74    74    63    62    58    32
```

Berichte des Instituts für Informatik

*	Nr. 1	Niklaus Wirth:	The Programming Language Pascal
*	Nr. 2	Niklaus Wirth:	Program development by step-wise refinement
	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)
*	Nr. 6	C.A.R. Hoare, Niklaus Wirth:	An Axiomatic Definition of the Language Pascal
	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
	Nr. 10	K.V. Nori, U. Ammann, K. Jensen, H.H. Nägeli, Ch. Jacobi:	The Pascal 'P' Compiler: Implementation Notes (Revised Edition)
	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	Urs Ammann:	Code Generation in a PASCAL Compiler
	Nr. 14	Karl Lieberherr:	Toward Feasible Solutions of NP-Complete Problems
*	Nr. 15	Erwin Engeler:	Structural Relations between Programs and Problems
	Nr. 16	Willi 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
*	Nr. 19	Niklaus Wirth:	The use of MODULA and Design and Implementation of MODULA
	Nr. 20	Edwin Wiedmer:	Exaktes Rechnen mit reellen Zahlen
*	Nr. 21	J. Nievergelt, H.P. Frei, et al.:	XS-0, a Self-explanatory School Computer
	Nr. 22	Peter Läuchli:	Ein Problem der ganzzahligen Approximation
	Nr. 23	Karl Bucher:	Automatisches Zeichnen von Diagrammen
	Nr. 24	Erwin Engeler:	Generalized Galois Theory and its Application to Complexity
	Nr. 25	Urs Ammann:	Error Recovery in Recursive Descent Parsers and Run-time Storage Organization
	Nr. 26	Efstathios Zachos:	Kombinatorische Logik und S-Terme
	Nr. 27	Niklaus Wirth:	MODULA-2

* out of print