Eidgenössische
Technische
Hochschule
Zürich

*Berichte der*
*Fachgruppe*
*Computer-*
*Wissenschaften*

Niklaus Wirth

# *The Programming Language Pascal*

*November 1970*

*1*

Niklaus Wirth

# *The Programming Language Pascal*

Abstract

A programming language called Pascal is described which was developed on the basis of Algol 60. Compared to Algol 60, its range of applicability is considerably increased due to a variety of data structuring facilities. In view of its intended usage both as a convenient basis to teach programming and as an efficient tool to write large programs, emphasis was placed on keeping the number of fundamental concepts reasonably small, on a simple and systematic language structure, and on efficient implementability. A one-pass compiler has been constructed for the CDC 6000 computer family; it is expressed entirely in terms of Pascal itself.

Second Edition: July 1971
Price SFr 6.00

(submitted for publication in ACTA INFORMATICA)

Contents

## 1. Introduction

The development of the language Pascal is based on two principal aims. The first is to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language. The second is to develop implementations of this language which are both reliable and efficient on presently available computers, dispelling the commonly accepted notion that useful languages must be either slow to compile or slow to execute, and the belief that any nontrivial system is bound to contain mistakes forever.

The desire for a new language for the purpose of teaching programming is due to my deep dissatisfaction with the presently used major languages whose features and constructs too often cannot be explained logically and convincingly and which too often represent an insult to minds trained in systematic reasoning. Along with this dissatisfaction goes my conviction that the language in which the student is taught to express his ideas profoundly influences his habits of tought and invention, and that the disorder governing these languages directly imposes itself onto the programming style of the students. I am inclined to think that the lack of discipline and structure in professional programming style is the major reason for the present appalling lack of reliability of practically all larger software products.

There is of course plenty of reason to be cautious with the introduction of yet another programming language, and the objection against teaching programming in a language which is not widely used and accepted has undoubtedly some justification - at least based on short-term commercial reasoning. However, the choice of a language for teaching based on its widespread acceptance and availability, together with the fact that the language

most widely taught is thereafter going to be the one most widely used, forms the safest recipe for stagnation in a subject of such profound paedagogical influence. I consider it therefore well worth-while to make an effort to break this vicious circle.

Of course a new language should not be developed just for the sake of novelty; existing languages should be used as a basis for development wherever they meet the criteria mentioned and do not impede a systematic structure. In that sense Algol 60 was used as a basis for Pascal, since it meets the demands with respect to teaching to a much higher degree than any other standard language. Thus the principles of structuring, and in fact the form of expressions, are copied from Algol 60. It was, however, not deemed appropriate to adopt Algol 60 as a subset of Pascal; certain construction principles, particularly those of declarations, would have been incompatible with those allowing a natural and convenient representation of the additional features of Pascal. However, conversion of Algol 60 programs to Pascal can be considered as a negligible effort of transcription, particularly if they do not involve name parameters.

The main extensions relative to Algol 60 lie in the domain of data structuring facilities, since their lack in Algol 60 was considered as the prime cause for its relatively narrow range of applicability. The introduction of record and file structures should make it possible to solve commercial type problems with Pascal, or at least to employ it successfully to demonstrate such problems in a programming course. This should be a help in erasing the mystical belief in the segregation between scientific and commercial programming methods. A first step in extending the data definition facilities of Algol 60 was undertaken in an effort to define a successor to Algol in 1965 [1].

Pascal has been implemented on the CDC 6000 computers. The compiler is written in Pascal itself as an efficient one-pass system. The "dialect" processed by this implementation is described by a few amendments to the general description of Pascal. They are included here as a separate chapter to demonstrate the brevity of a manual necessary to characterise a particular implementation. Moreover, they show how facilities are introduced into this high-level, machine independent programming language, which permit the programmer to take advantage of the characteristics of a particular machine.

## 2. Summary of the language

An algorithm or computer program consists of two essential parts, a description of actions which are to be performed, and a description of the data, which are manipulated by these actions. Actions are described in Pascal by so-called statements, and data are described by so-called declarations and definitions.

The data are represented by values of variables. Every variable occuring in a statement must be introduced by a variable declaration which associates an identifier and a data type with that variable. The data type essentially defines the set of values which may be assumed by that variable. A data type may in Pascal be either directly described in the variable declaration, or it may be referenced by a type identifier, in which case this identifier must be described by an explicit type definition.

The basic data types are the scalar types. Their definition indicates an ordered set of values, i.e. introduces an identifier as a constant standing for each value in the set. Apart from the definable scalar types, there exist in Pascal four standard scalar types, whose values are not denoted by identifiers, but instead by numbers and quotations respectively, which are syntactically distinct from identifiers. These types are: integer, real

char, and alfa.

The set of values of type char is the character set available on the printers of a particular installation. Alfa type values consist of sequences of characters whose length again is implementation dependent, i.e. is the number of characters packed per word. Individual characters are not directly accessible, but alfa quantities can be unpacked into a character array (and vice-versa) by a standard procedure.

A scalar type may also be defined as a subrange of another scalar type by indicating the smallest and the largest value of the subrange.

Structured types are defined by describing the types of their components and by indicating a structuring method. The various structuring methods differ in the selection mechanism serving to select the components of a variable of the structured type. In Pascal, there are five structuring methods available: array structure, record structure, powerset structure, file structure, and class structure.

In an array structure, all components are of the same type. A component is selected by an array selector, or computable index, whose type is indicated in the array type definition and which must be scalar. It is usually a programmer-defined scalar type, or a subrange of the type integer. Given a value of the index type, an array selector yields a value of the component type. Every array variable can therefore be regarded as a mapping of the index type onto the component type. The time needed for a selection does not depend on the value of the selector (index). The array structure is therefore called a random-access structure.

In a record structure, the components (called fields) are not necessarily of the same type. In order that the type of a selected component be evident from the program text (without executing the program), a record selector does not contain a computable value, but instead consists of an identifier uniquely denoting the component to be selected. These component identifiers are defined in the record type definition. Again, the time needed to access a selected component does not depend on the selector, and the record structure is therefore also a random-access structure.

A record type may be specified as consisting of several variants. This implies that different variables, although said to be of the same type, may assume structures which differ in a certain manner. The difference may consist of a different number and different types of components. The variant which is assumed by the current value of a record variable is indicated by a component field which is common to all variants and is called the tag field. Usually, the part common to all variants will consist of several components, including the tag field.

A powerset structure defines a set of values which is the powerset of its base type, i.e. the set of all subsets of values of the base type. The base type must be a scalar type, and will usually be a programmer-defined scalar type or a subrange of the type integer.

A file structure is a sequence of components of the same type. A natural ordering of the components is defined through the sequence. At any instance, only one component is directly accessible. The other components are made accessible through execution of standard file positioning procedures. A file is at any time in one of the three modes called input, output, and neutral.

According to the mode, a file can be read sequentially, or it can be written by appending components to the existing sequence of components. File positioning procedures may influence the mode. The file type definition does not determine the number of components, and this number is variable during execution of the program.

The class structure defines a class of components of the same type whose number may alter during execution of a program. Each declaration of a variable with class structure introduces a set of variables of the component type. The set is initially empty. Every activation of the standard procedure alloc (with the class as implied parameter) will generate (or allocate) a new component variable in the class and yield a value through which this new component variable may be accessed. This value is called a pointer, and may be assigned to variables of type pointer. Every pointer variable, however, is through its declaration bound to a fixed class variable, and because of this binding may only assume values pointing to components of that class. There exists a pointer value nil which points to no component whatsoever, and may be assumed by any pointer variable irrespective of its binding. Through the use of class structures it is possible to construct data corresponding to any finite graph with pointers representing edges and component variables representing nodes.

The most fundamental statement is the assignment statement. It specifies that a newly computed value be assigned to a variable (or component of a variable). The value is obtained by evaluating an expression. Expressions consist of variables, constants, sets, operators and functions operating on the denoted quantities and producing new values. Variables, constants, and functions are either declared in the program or are standard entities. Pascal defines a fixed set of operators, each of which

can be regarded as describing a mapping from the operand type into the result type. The set of operators is subdivided into groups of

1. arithmetic operators of addition, subtraction, sign inversion, multiplication, division, and computing the remainder.

   The operand and result types are the types integer and real, or subrange types of integer.

2. Boolean operators of negation, union (or), and conjunction (and). The operand and result types are Boolean (which is a standard type).

3. set operators of union, intersection, and set difference. The operands and results are of any powerset type.

4. relational operators of equality, inequality, ordering and set membership. The result of relational operations is of type Boolean. Any two operands may be compared for equality as long as they are of the same type. The ordering relations apply only to scalar types.

The assignment statement is a so-called simple statement, since it does not contain any other statement within itself. Another kind of simple statement is the procedure statement, which causes the execution of the designated procedure (see below). Simple statements are the components or building blocks of structured statements, which specify sequential, selective, or repeated execution of their components. Sequential execution of statements is specified by the compound statement, conditional or selective execution by the if statement and the case statement, and repeated execution by the repeat statement, the while statement, and the for statement. The if statement serves to make the execution of a statement dependent on the value of a Boolean

expression, and the case statement allows for the selection among many statements according to the value of a selector. The for statement is used when the number of iterations is known beforehand, and the repeat and while statements are used otherwise.

A statement can be given a name (identifier), and be referenced through that identifier. The statement is then called a procedure, and its declaration a procedure declaration. Such a declaration may additionally contain a set of variable declarations, type definitions and further procedure declarations. The variables, types and procedures thus defined can be referenced only within the procedure itself, and are therefore called local to the procedure. Their identifiers have significance only within the program text which constitutes the procedure declaration and which is called the scope of these identifiers. Since procedures may be declared local to other procedures, scopes may be nested. Entities which are defined or declared in the main program, i.e. not local to some procedure, are called global.

A procedure may have a fixed number of parameters, each of which is within the procedure denoted by an identifier called the formal parameter. Upon an activitation of the procedure statement, an actual quantity has to be indicated for each parameter which can be referenced from within the procedure through the formal parameter. This quantity is called the actual parameter. Parameters can be variable parameters, procedure parameters, or function parameters. In the case of a variable parameter, its type has to be specified in the declaration of the formal parameter. If the actual variable parameter contains a (computable) selector, this selector is evaluated before the procedure is activated in order to designate the selected component variable.

Functions are declared analogously to procedures. The only difference lies in the fact that a function yields a result which is confined to a scalar type and must be specified in the function declaration. Functions may therefore be used as constituents of expressions. In order to eliminate side-effects, assignments to non-local variables are not allowed to occur within the function.

## 3. Notation, terminology, and vocabulary

According to traditional Backus-Naur form, syntactic constructs are denoted by English words enclosed between the angular brackets < and >. These words also describe the nature or meaning of the construct, and are used in the accompanying description of semantics. Possible repetition of a construct is indicated by an asterisk (0 or more repetitions) or a circled plus sign (1 or more repetitions). If a sequence of constructs to be repeated consists of more than one element, it is enclosed by the meta-brackets { and } which imply a repetition of 0 or more times.

The basic vocabulary consists of basic symbols classified into letters, digits, and special symbols.

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<special symbol> ::=

+|-|*|/|∨|∧|¬|=|≠|<|>|≤|≥|(|)|[|]|{|}|:=|
.|,|;|:|'|↑|div|mod|nil|in|
if|then|else|case|of|repeat|until|while|do|
for|to|downto|begin|end|with|goto|
const|var|type|array|record|powerset|file|class|
function|procedure|label

The construct

{ <any sequence of symbols not containing "}"> }

may be inserted between any two identifiers, numbers (cf.4), or special symbols. It is called a comment and may be removed from the program text without altering its meaning. The symbols { and } do not occur otherwise in the language, and when appearing in syntactic descriptions they denote metabrackets implying repetition.

## 4. Identifiers and Numbers

Identifiers serve to denote constants, types, variables, procedures and functions. Their association must be unique within their scope of validity, i.e. within the procedure or function in which they are declared (cf. 10 and 11).

<identifier> ::= <letter><letter or digit>*

<letter or digit> ::= <letter> | <digit>

The decimal notation is used for numbers, which are the constants of the data types integer and real. The symbol ' preceding the scale factor is pronounced as "times 10 to the power of".

<number> ::= <integer> | <real number>
<integer> ::= <digit>$^{\oplus}$
<real number> ::= <digit>$^{\oplus}$.<digit>$^{\oplus}$ |
<digit>$^{\oplus}$.<digit>$^{\oplus}$'<scale factor>|<integer>'<scale factor>
<scale factor> ::= <digit>$^{\oplus}$ | <sign> <digit>$^{\oplus}$
<sign> ::= + | -

Examples:

1 100 0.1 5'-3 87.35'+8

## 5. Constant definitions

A constant definition introduces an identifier as a synonym to a constant.

<unsigned constant> ::= <number> | '<character>$^{\oplus}$' |
<identifier> | nil
<constant> ::= <unsigned constant> | <sign> <number>
<constant definition> ::= <identifier> = <constant>

## 6. Data type definitions

A data type determines the set of values which variables of that type may assume and associates an identifier with the type. In the case of structured types, it also defines their structuring method.

<type> ::= <scalar type> | <subrange type> |
<array type> | <record type> | <powerset type> |
<file type> | <class type> | <pointer type> |
<type identifier>
<type identifier> ::= <identifier>
<type definition> ::= <identifier> = <type>

## 6.1. Scalar types

A scalar type defines an ordered set of values by enumeration of the identifiers which denote these values.

<scalar type> ::= (<identifier> {,<identifier>} )

Examples:

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

Functions applying to all scalar types are:

| | |
|---|---|
| succ | the succeeding value (in the enumeration) |
| pred | the preceding value (in the enumeration) |

### 6.1.1. Standard scalar types

The following types are standard in Pascal, i.e. the identifier denoting them is predefined:

| | |
|---|---|
| integer | the values are the integers within a range depending on the particular implementation. |

The values are denoted by integers (cf.4) and not by identifiers.

real the values are a subset of the real numbers depending on the particular implementation. The values are denoted by real numbers as defined in paragraph 4.

Boolean (false, true)

char the values are a set of characters depending on a particular implementation. They are denoted by the characters themselves enclosed within quotes.

alfa the values are sequences of n characters, where n is an implementation dependent parameter. If $\alpha$ and $\beta$ are values of type alfa

$\alpha = a_1 \ldots a_k \ldots a_n$
$\beta = b_1 \ldots b_k \ldots b_n$ , then

$\alpha = \beta$ , if and only if $a_i = b_i$ for $i = 1 \ldots n$,
$\alpha < \beta$ , if and only if $a_i = b_i$ for $i = 1 \ldots k-1$ and $a_k < b_k$
$\alpha > \beta$ , if and only if $a_i = b_i$ for $i = 1 \ldots k-1$ and $a_k > b_k$

Alfa values are denoted by sequences of (at most) n characters enclosed in quotes. Trailing blanks may be omitted.
Alfa quantities may be regarded as a packed representation of short character arrays (cf. also 10.1.3.).

## 6.1.2. Subrange types

A type may be defined as a subrange of another scalar type by indication of the least and the highest value in the subrange. The first constant specifies the lower bound, and must not be greater than the upper bound.

<subrange type> ::= <constant>..<constant>

Examples: 1..100
-10..+10
Monday..Friday

## 6.2. Structured types

### 6.2.1. Array types

An array type 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 definition specifies the component type as well as the index type.

<array type> ::= array [<index type> {,<index type>}] of <component type>
<index type> ::= <scalar type> | <subrange type> | <type identifier>
<component type> ::= <type>

If n index types are specified, the array type is called n-dimensional, and a component is designated by n indices.

Examples: array [1..100] of real
array [1..10, 1..20] of 0..99
array [-10..+10] of Boolean
array [Boolean] of Color

### 6.2.2. Record types

A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called field, its type and an identifier which denotes it. The scope of these

so-called field identifiers is the record definition itself, and they are also accessible within a field designator (cf.7.2) referring to a record variable of this type.

A record type may have several variants, in which case a certain field is designated as the tag field, whose value indicates which variant is assumed by the record variable at a given time. Each variant structure is identified by a case label which is a constant of the type of the tag field.

```
<record type> ::= record <field list> end
<field list> ::= <fixed part>|<fixed part>;<variant part>|
                 <variant part>
<fixed part> ::= <record section> {;<record section>}
<record section> ::= <field identifier> {,<field identifier>}:<type>
<variant part> ::= case <tag field> : <type identifier> of
                   <variant> {;<variant>}
<variant> ::= {<case label> :}* (<field list>)|{<case label>:}*
<case label> ::= <unsigned constant>
<tag field> ::= <identifier>
```

Examples:

```
record day: 1..31;
       month: 1..12;
       year: 0..2000
end

record name, firstname: alfa;
       age: 0..99;
       married: Boolean
end
```

```
record x,y: real;
       area: real;
case s: Shape of
triangle:  (side: real;
            inclination, angle1, angle2: Angle);
rectangle: (side1, side2: real;
            skew, angle3: Angle);
circle:    (diameter: real)
end
```

### 6.2.3. Powerset types

A powerset type defines a range of values as the powerset of another scalar type, the so-called base type. Operators applicable to all powerset types are:

| | |
|---|---|
| ∨ | union |
| ∧ | intersection |
| - | set difference |
| in | membership |

<powerset type> ::= powerset <subrange type> |
powerset <type identifier>

### 6.2.4. File types

A file type definition specifies a structure consisting of a sequence of components, all of the same type. The number of components, called the length of the file, is not fixed by the file type definition, i.e. each variable of that type may have a value with a different, varying length.

Associated with each variable of file type is a file position or file pointer, denoting a specific element. The file position or the file pointer can be moved by certain standard procedures, some of which are only applicable when the file is in one of the three modes:input (being read), output (being written), or neutral (passive). Initially, a file variable is in the neutral mode.

<file type> ::= file of <type>

### 6.2.5. Class types

A class type definition specifies a structure consisting of a class of components, all of the same type. The number of components is variable; the initial number upon declaration of a variable of class type is zero. Components are created (allocated) during execution of the program through the standard procedure alloc. The maximum number of components which can thus be created, however, is specified in the type definition.

<class type> ::= class <maxnum> of <type>
<maxnum> ::= <integer>

### 6.2.6. Pointer types

A pointer type is associated with every variable of class type. Its values are the potential pointers to the components of that class variable (cf.7.5) and the pointer constant nil designating no component. A pointer type is said to be bound to its class variable.

<pointer type> ::= ↑ <class variable>
<class variable> ::= <variable>

Examples of type definitions:

```
Color     = (red, yellow, green, blue)
Sex       = (male, female)
Charfile  = file of char
Shape     = (triangle, rectangle, circle)
Card      = array [1..80]of char
Complex   = record realpart, imagpart: real end
Person    = record name, firstname: alfa;
                   age: integer;
                   married: Boolean;
                   father, youngestchild, eldersibling: ↑family;
            case s: Sex of
            male: (enlisted, bold: Boolean);
            female: (pregnant: Boolean;
                   size: array [1..3] of integer)
            end
```

## 7. Declarations and denotations of variables

Variable declarations consist of a list of identifiers denoting the new variables, followed by their type.

<variable declaration> ::= <identifier> {,<identifier>} : <type>

Two standard file variables can be assumed to be predeclared as

```
input, output: file of char
```

The file input is restricted to input mode (reading only), and the file output is restricted to output mode (writing only). A Pascal program should be regarded as a procedure with these two variables as formal parameters. The corresponding actual parameters are expected either to be the standard input and output media of the computer installation, or to be specifyable

in the system command activating the Pascal system.

Examples:

```
x,y,z: real
u,v: Complex
i,j: integer
k: 0..9
p,q: Boolean
operator: (plus, times, absval)
a: array [0..63] of real
b: array [Color, Boolean] of
            record occurence: integer;
                   appeal: real
            end
c: Color
f: file of Card
hue1, hue2: powerset Color
family: class 100 of Person
p1,p2: ↑family
```

Denotations of variables either denote an entire variable or a component of a variable.

<variable> ::= <entire variable> | <component variable>

## 7.1. Entire variables

An entire variable is denoted by its identifier.

<entire variable> ::= <variable identifier>
<variable identifier> ::= <identifier>

## 7.2. Component variables

A component of a variable is denoted by the denotation for the variable followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

<component variable> ::= <indexed variable> |
<field designator> | <current file component> |
<referenced component>

### 7.2.1. Indexed variables

A component of an n-dimensional array variable is denoted by the denotation of the variable followed by n index expressions.

<indexed variable> ::=
<array variable> [<expression> {,<expression>} ]
<array variable> ::= <variable>

The types of the index expressions must correspond with the index types declared in the definition of the array type.

Examples:

a[12]
a[i+j]
b[red,true]
b[succ(c), p ∧ q]
f↑[1]

### 7.2.2. Field designators

A component of a record variable is denoted by the denotation of the record variable followed by the field identifier of the component.

<field designator> ::= <record variable>.<field identifier>
<record variable> ::= <variable>
<field identifier> ::= <identifier>

Examples:

u.realpart
v.realpart
b[red,true].appeal
p2↑.size

## 7.2.3. Current file components

At any time, only the one component determined by the current file position (or file pointer) is directly accessible.

<current file component> ::= <file variable>↑
<file variable> ::= <variable>

## 7.2.4. Referenced components

Components of class variables are referenced by pointers.

<referenced component> ::= <pointer variable>↑
<pointer variable> ::= <variable>

Thus, if p1 is a pointer variable which is bound to a class variable v, p1 denotes that variable and its pointer value, whereas p1↑ denotes the component of v referenced by p1.

Examples:

p1↑.father
p1↑.elder sibling↑.youngest child

## 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, i.e. variables and constants, operators, and functions.

The rules of composition specify operator precedences according to four classes of operators. The operator ¬ 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. These rules of precedence are reflected by the following syntax:

```
<factor> ::= <variable> | <unsigned constant> |
             <function designator> | <set> | (<expression>) |
             ¬<factor>
<set> ::= [ <expression> {,<expression>} ] | [ ]
<term> ::= <factor> | <term><multiplying operator><factor>
<simple expression> ::= <term> |
             <simple expression> <adding operator><term> |
             <adding operator><term>
<expression> ::= <simple expression> |
             <simple expression><relational operator>
             <simple expression>
```

Expressions which are members of a set must all be of the same type, which is the base type of the set. [] denotes the empty set.

Examples:

Factors:
```
x
15
(x+y+z)
sin(x+y)
[red,c,green]
¬p
```

Terms: x * y
i/(1-i)
p ∧ q
(x ≤ y) ∧ (y < z)

Simple expressions: x + y
-x
hue1 ∨ hue2
i*j + 1

Expressions: x = 1.5
p ≤ q
(i < j) = (j < k)
c in hue1

## 8.1. Operators

### 8.1.1. The Operator ¬

The operator ¬ applied to a Boolean operand denotes negation.

### 8.1.2. Multiplying operators

<multiplying operator> ::= * | / | div | mod | ∧

| operator | operation | type of operands | type of result |
|---|---|---|---|
| * | multiplication | real<br>integer | integer, if both operands are of type integer, real otherwise |
| / | division | real<br>integer | real |
| div | division with truncation | integer | integer |
| mod | modulus | integer | integer |
| ∧ | logical "and" | Boolean | Boolean |
| | set intersection | any powerset type T | T |

8.1.3. Adding operators

<adding operator> ::= + | - | ∨

| operator | operation | type of operands | type of result |
|---|---|---|---|
| + | addition | real<br>integer | integer, if both operands are of type integer, real otherwise |
| - | subtraction | real<br>integer | |
| - | set difference | | |
| ∨ | logical "or" | Boolean | Boolean |
| | set union | any powerset type T | T |

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

### 8.1.4. Relational operators

<relational operator> ::= =|≠|<|≤|≥|>|in

| operator | type of operands | result |
|---|---|---|
| = ≠ | any type, except file and class types | Boolean |
| < > ≤ ≥ | any scalar or subrange type | Boolean |
| in | any scalar or subrange type and its powerset type respectively | Boolean |

Notice that all scalar types define ordered sets of values. In particular, false < true.

The operators ≤ and ≥ may also be used for comparing values of powerset type, and then denote set inclusion ⊆ and ⊇ respectively.

### 8.2. Function designators

A function designator specifies the activation of a function. It consists of the identifier designating the function and a list of actual parameters. The parameters are variables, expressions, procedures, and functions, and are substituted for the corresponding formal parameters (cf. 9.1.2., 10, and 11).

<function designator> ::=
<function identifier> (<actual parameter>{,<actual parameter>})
<function identifier> ::= <identifier>

Examples:

```
Sum(a,100)
GCD(147,k)
sin(x+y)
eof(f)
```

## 9. Statements

Statements denote algorithmic actions, and are said to be executable.

<statement> ::= <simple statement> | <structured statement>

### 9.1. Simple statements

A simple statement is a statement of which no part constitutes another statement.

<simple statement> ::= <assignment statement> |
<procedure statement> | <goto statement>

#### 9.1.1. Assignment statements

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

<assignment statement> ::= <variable> := <expression> |
<function identifier> := <expression>

The variable (or the function) and the expression must be of identical, but neither file nor class type, with the following exceptions being permitted:

1. the type of the variable is real, and the type of the expression is integer or a subrange thereof.
2. the type of the expression is a subrange of the type of the variable.

Examples:

```
  x := y+z
  p := (1 ≤ i) ∧ (i < 100)
  i := sqr(k) - (i*j)
hue := [blue,succ(c)]
```

## 9.1.2. Procedure statements

A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement 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 four kinds of parameters: variable parameters, constant parameters, procedure parameters (the actual parameter is a procedure identifier), and function parameters (the actual parameter is a function identifier).

In the case of variable parameters, the actual parameter must be a variable. If it is a variable denoting a component of a structured variable, the selector is evaluated when the substitution takes place, i.e. before the execution of the procedure. If the parameter is a constant parameter, then the corresponding actual parameter must be an expression,

<procedure statement> ::= <procedure identifier> |
<procedure identifier> (<actual parameter>
{,<actual parameter>})
<procedure identifier> ::= <identifier>
<actual parameter> ::= <expression> | <variable> |
<procedure identifier> | <function identifier>

Examples:

next
Transpose (a,n,m)
Bisect (sin,-1,+2,x,q)

## 9.1.3. Goto statements

A goto statement serves to indicate that further processing should continue at another part of the program text, namely at the place of the label. Labels can be placed in front of statements being part of a compound statement (cf.2.1.).

<goto statement> ::= goto <label>
<label> ::= <integer>

The following restrictions hold concerning the applicability of labels:

1. The scope (cf.10) of a label is the procedure within which it is defined. It is therefore not possible to jump into a procedure.
2. If a goto statement leads outside of a procedure, then its label must be specified in a label declaration in the heading of the procedure in which the label is defined.

## 9.2. Structured statements

Structured statements are constructs composed of other statements which have to be executed either in sequence (compound statement), conditionally (conditional statements), or repeatedly (repetitive statements).

<structured statement> ::= <compound statement> |
<conditional statement> | <repetitive statement> |
<with statement>

### 9.2.1. Compound statements

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. Each statement may be preceded by a label which can be referenced by a goto statement (cf.9.1.3.).

<compound statement> ::=
begin <component statement> {;<component statement>} end
<component statement> ::=
<statement> | <label definition><statement>
<label definition> ::= <label> :

Example:

begin z := x; x := y; y := z end

### 9.2.2. Conditional statements

A conditional statement selects for execution a single one of its component statements.

<conditional statement> ::=
<if statement> | <case statement>

9.2.2.1. If statements

The if statement specifies that a statement be executed only if a certain condition (Boolean expression) is true.

If it is false, then either no statement is to be executed, or the statement following the symbol else is to be executed.

<if statement> ::= if <expression> then <statement> |
if <expression> then <statement> else <statement>

The expression between the symbols if and then must be of type Boolean.

Note:

The syntactic ambiguity arising from the construct

if <expression-1> then if <expression-2> then <statement-1>
else <statement-2>

is resolved by interpreting the construct as equivalent to

if <expression-1> then
begin if <expression-2> then <statement-1> else <statement-2>
end

Examples:

if x < 1.5 then z := x+y else z := 1.5
if p ≠ nil then p := p↑.father

9.2.2.2. Case statements

The case statement consists of an expression (the selector) and a list of statements, each being labeled by a constant of the type of the selector. It specifies that the one statement be executed whose label is equal to the current value of the selector.

<case statement> ::= case <expression> of
    <case list element> {;<case list element>} end
<case list element> ::= {<case label>:}* <statement> |
    {<case label>:}*

Examples:

```
case operator of
plus:    x := x+y;
times:   x := x*y;
absval:  if x < 0 then x := -x
end
```

### 9.2.3. Repetitive statements

Repetitive statements specify that certain statements are to be executed repeatedly. If the number of repetitions is known beforehand, i.e. before the repetitions are started, the for statement is the appropriate construct to express this situation; otherwise the while or repeat statement should be used.

<repetitive statement> ::= <while statement> |
    <repeat statement> | <for statement>

#### 9.2.3.1. While statements

<while statement> ::= while <expression> do <statement>

The expression controlling repetition must be of type Boolean. The statement is repeatedly executed until the expression becomes false. If its value is false at the beginning, the statement is not executed at all. The while statement

while e do S

is equivalent to

```
if e then
    begin S;
        while e do S
    end
```

Examples:

```
while (a[i] ≠ x) ∧ (i<n) do i := i+1
```

```
while i > 0 do
begin if odd(i) then z := z*x;
      i := i div 2;
      x := sqr(x)
end
```

9.2.3.2. Repeat statements

<repeat statement> ::=

repeat <statement> {;<statement>} until <expression>

The expression controlling repetition must be of type Boolean. The sequence of statements between the symbols repeat and until is repeatedly (and at least once) executed until the expression becomes true. The repeat statement

```
repeat S until e
```

is equivalent to

```
begin S;
    if ¬ e then
        repeat S until e
end
```

Examples:

```
repeat k := i mod j;
       i := j;
       j := k
until j = 0
```

```
repeat get(f)
until (f↑ = a) ∨ eof(f)
```

9.2.3.3. For statements

The for statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the control variable of the for statement.

```
<for statement> ::=
    for <control variable> := <for list> do <statement>
<for list> ::= <initial value> to <final value> |
    <initial value> downto <final value>
<control variable> ::= <identifier>
<initial value> ::= <expression>
<final value> ::= <expression>
```

The control variable, the initial value, and the final value must be of the same scalar type (or subrange thereof).

A for statement of the form

```
for v := e1 to e2 do S
```

is equivalent to the statement

```
if e1 ≤ e2 then
begin v := e1; S;
    for v := succ(v) to e2 do S
end
```

and a for statement of the form

```
for v := e1 downto e2 do S
```

is equivalent to the statement

```
if e1 ≥ e2 then
begin v := e1; S;
    for v := pred(v) downto e2 do S
end
```

Note: The repeated statement S must alter neither the value of the control variable nor the final value.

Examples:

```
for i := 2 to 100 do if a[i] > max then max := a[i]
```

```
for i := 1 to n do
for j := 1 to n do
begin x := 0;
   for k := 1 to n do x := x+a[i,k]*b[k,j];
   c[i,j] := x
end
```

```
for c := red to blue do try(c)
```

### 9.2.4. With statements

<with statement> ::= with <record variable> do <statement>

Within the component statement of the with statement, the components (fields) of the record variable specified by the with clause can be denoted by their field identifier only, i.e. without preceding them with the denotation of the entire record variable. The with clause effectively opens the scope containing the field identifiers of the specified record variable, so that the field identifiers may occur as variable identifiers.

Example:

```
with date do
begin month := month+1;
   if month > 12 then
   begin month := 1; year := year+1
   end
end
```

This statement is equivalent to

```
begin date.month := date.month+1;
   if date.month > 12 then
   begin date.month := 1; date.year := date.year+1
   end
end
```

## 10. Procedure declarations

Procedure declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by procedure statements. A procedure declaration consists of the following parts, any of which, except the first and the last, may be empty:

<procedure declaration> ::=
<procedure heading><label declaration part>
<constant definition part><type definition part>
<variable declaration part>
<procedure and function declaration part><statement part>

The procedure heading specifies the identifier naming the procedure and the formal parameter identifiers (if any).
The parameters are either variable-, procedure-, or function parameters (cf. also 9.1.2.).

<procedure heading> ::= procedure <identifier> ; |
procedure <identifier> (<formal parameter section>
{;<formal parameter section>} ) ;

<formal parameter section> ::=
<parameter group> |
const <parameter group> {;<parameter group>} |
var <parameter group> {;<parameter group>} |
function <parameter group> |
procedure <identifier> {,<identifier>}
<parameter group> ::= <identifier> {,<identifier>}:
<type identifier>

A parameter group without preceding specifier implies constant parameters.

The label declaration part specifies all labels which are defined local to the procedure and occur in goto statements within procedures which are themselves local to the procedure.

<label declaration part> ::= <empty> |
    label <label> {,<label>}

The constant definition part contains all constant synonym definitions local to the procedure.

<constant definition part> ::= <empty>
    const <constant definition> {,<constant definition>};

The type definition part contains all type definitions which are local to the procedure declaration.

<type definition part> ::= <empty> |
    type <type definition> {;<type definition>};

The variable declaration part contains all variable declarations local to the procedure declaration.

<variable declaration part> ::= <empty> |
    var <variable declaration>{;<variable declaration>};

The procedure and function declaration part contains all procedure and function declarations local to the procedure declaration.

<procedure and function declaration part> ::=
  {<procedure or function declaration> ;}
<procedure or function declaration> ::=
    <procedure declaration> | <function declaration>

The statement part specifies the algorithmic actions to be executed upon an activation of the procedure by a procedure statement.

<statement part> ::= <compound statement>

All identifiers introduced in the formal parameter part, the constant definition part, the type definition part, the variable-, procedure or function declaration parts are local to the procedure declaration which is called the scope of these identifiers. They are not known outside their scope. In the case of local variables, their values are undefined at the beginning of the statement part.

The use of the procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure.

Examples of procedure declarations:

```
procedure readinteger (var x: integer);
    var i,j: integer;
begin i := 0;
    while (input↑ ≥ '0') ∧ (input↑ ≤ '9') do
    begin j := int(input↑) - int('0');
          i := i * 10 + j;
          get(input)
    end;
    x := i
end
```

```
procedure Bisect(function f: real; const low, high: real;
    var zero: real;  p: Boolean);
    var a,b,m: real;
begin a := low;  b := high;
    if (f(a) ≥ 0) ∨ (f(b) ≤ 0) then p := false else
    begin p := true;
       while abs(a-b) > eps do
       begin m := (a+b)/2;
          if f(m)  > 0 then b := m else a := m
       end;
       zero := a
    end
end
```

```
procedure GCD(m,n: integer; var x,y,z: integer);
var a1,a2,b1,b2,c,d,q,r: integer;   m ≥ 0, n > 0
begin{Greatest Common Divisor x of m and n,
      Extended Euclid's Algorithm, cf.[2] p.14}
    a1 := 0;  a2 := 1;  b1 := 1;  b2 := 0; c := m;  d := n;
    while d ≠ 0 do
    begin {a1*m + b1*n = d,  a2*m + b2*n = c,
           gcd(c,d) = gcd(m,n)}
           q := c div d; r := c mod d;
           { c = q*d+r,  gcd(d,r) = gcd(m,n)}
           a2 := a2 - q*a1;  b2 := b2 - q*b1;
           { a2*m + b2*n = r, a1*m + b1*n = d}
           c := d;  d := r;
           r := a1; a1 := a2;  a2 := r;
           r := b1;  b1 := b2;  b2 := r;
           { a1*m + b1*n = d, a2*m + b2*n = c,
           gcd(c,d) = gcd(m,n) }
    end;
    { gcd(c,0) = c = gcd(m,n)}
    x := c;  y := a2;  z := b2
    { x = gcd(m,n), y*m + z*n = gcd(m,n) }
end
```

## 10.1. Standard procedures

Standard procedures are supposed to be predeclared in every implementation of Pascal. Any implementation may feature additional predeclared procedures. Since they are, as all standard quantities, assumed as declared in a scope surrounding the Pascal program, no conflict arises form a declaration redefining the same identifier within the program. The standard procedures are listed and explained below.

### 10.1.1. File positioning procedures

put(f) advances the file pointer of file f to the next file component. It is only applicable, if the file is either in the output or in the neutral mode. The file is put into the output mode.

get(f) advances the file pointer of file f to the next file component. It is only applicable, if the file is either in the input or in the neutral mode. If there does not exist a next file component, the end-of-file condition arises, the value of the variable denoted by f↑ becomes undefined, and the file is put into the neutral mode.

reset(f) the file pointer of file f is reset to its beginning, and the file is put into the neutral mode.

### 10.1.2. Class component allocation procedure

alloc(p) allocates a new component in the class to which the pointer variable p is bound, and assigns the pointer designating the new component to p. If the component type is a record type with variants, the form

alloc(p,t) can be used to allocate a component of the variant whose tag field value is t. However, this allocation does not imply an assignment to the tag field. If the class is already compleately allocated, the value nil will be assigned to p.

### 10.1.3. Data transfer procedures

Assuming that a is a character array variable, z is an alfa variable, and i is an integer expression, then

pack(a,i,z) packs the n characters a[i] ... a[i+n-1] into the alfa variable z (for n cf. 6.1.1.), and

unpack(z,a,i) unpacks the alfa value z into the variables a[i] ... a[i+n-1].

## 11. Function declarations

Function declarations serve to define parts of the program which compute a scalar value or a pointer value. Functions are activated by the evaluation of a function designator (cf.8.2) which is a constituent of an expression. A function declaration consists of the following seven parts, any of which, except the first and the last, may be empty (cf. also 10.).

<function declaration> ::=
<function heading><label declaration part>
<constant definition part><type definition part>
<variable declaration part>
<procedure and function declaration part><statement part>

The function heading specifies the identifier naming the function, the formal parameters of the function (note that there must be at least one parameter), and the type of the (result of the) function.

<function heading> ::=
function <identifier> (<formal parameter section>
{;<formal parameter section>}) : <result type> ;
<result type> ::= <type identifier>

The type of the function must be a scalar or a subrange type or a pointer type. Within the function declaration there must be at least one assignment statement assigning a value to the function identifier. This assignment determines the result of the function.

Occurrence of the function identifier in a function designator within its declaration implies recursive execution of the function. Within the statement part no assignment must occur to any variable which is not local to the function. This rule also excludes assignments to parameters.

Examples:

```
function Sqrt(x: real): real;
     var x0,x1: real;
begin x1 := x; { x > 1, Newton's method }
     repeat x0 := x1;  x1 := (x0 + x/x0) * 0.5
          { x0^2 - 2*x1 * x0 + x = 0 }
     until abs (x1 - x0) ≤ eps;
     { (x0-eps) ≤ x1 ≤ (x0 + eps),
     (x - 2*eps*x0) ≤ x0^2 ≤ (x + 2*eps*x0) }
     Sqrt := x0
end
```

```
function Max(a: vector;  n: integer): real;
     var x: real;  i: integer;
begin x := a[1];
     for i := 2 to n do
     begin { x = max(a1...an-1)}
        if x < a[i] then x := a[i]
        { x = max(a1...ai) }
     end;
     { x = max(a1...an)}
     Max := x
end
```

```
function GCD(m,n: integer): integer;
begin if n = 0 then GCD := m else GCD := GCD(n,m mod n)
end
```

```
function Power(x: real; y: integer): real;  { y ≥ 0 }
     var w,z: real;  i: integer;
begin w := x;  z := 1;  i := y;
     while i ≠ 0 do
     begin { z*w^i = x^y }
        if odd(i) then z := z*w;
        i := i div 2; { z*w^2i = x^y }
        w := sqr(w)   { z*w^i = x^y }
     end;
     { i = 0, z = x^y }
     Power := z
end
```

## 11.1. Standard functions

Standard functions are supposed to be predeclared in every implementation of Pascal. Any implementation may feature additional predeclared functions (cf. also 10.1.).

The standard functions are listed and explained below:

### 11.1.1. Arithmetic functions

`abs(x)` computes the absolute value of x. The type of x must be either *real* or *integer*, and the type of the result is the type of x.

`sqr(x)` computes $x^2$. The type of x must be either *real* or *integer*, and the type of the result is the type of x.

`sin(x)`, `cos(x)`, `exp(x)`, `ln(x)`, `sqrt(x)`, `arctan(x)` — the type of x must be either *real* or *integer*, and the type of the result is *real*

### 11.1.2. Predicates

odd(x) the type of x must be integer, and the result is x mod 2 = 1

eof(f) indicates, whether the file f is in the end-of-file status.

### 11.1.3. Transfer functions

trunc(x) x must be of type real, and the result is of type integer, such that abs(x)-1 < trunc(abs(x)) ≤ abs(x)

int(x) x must be of type char, and the result (of type integer) is the ordinal number of the character x in the defined character set.

chr(x) x must be of type integer, and the result (of type char) is the character whose ordinal number is x.

### 11.1.4. Further standard functions

succ(x) x is of any scalar or subrange type, and the result is the successor value of x (if it exists).

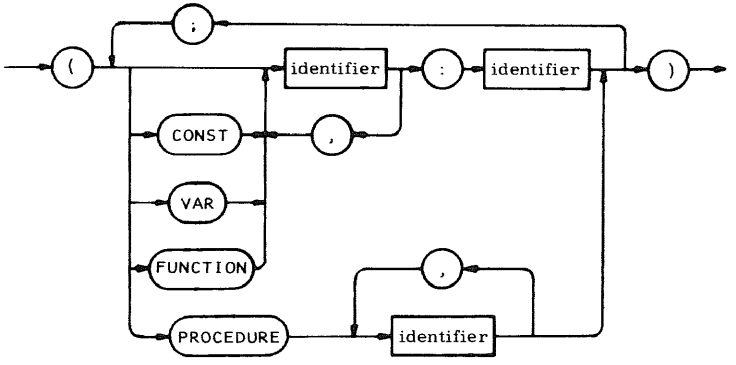
pred(x) x is of any scalar or subrange type, and the result is the predecessor value of x (if it exists).

## 12. Programs

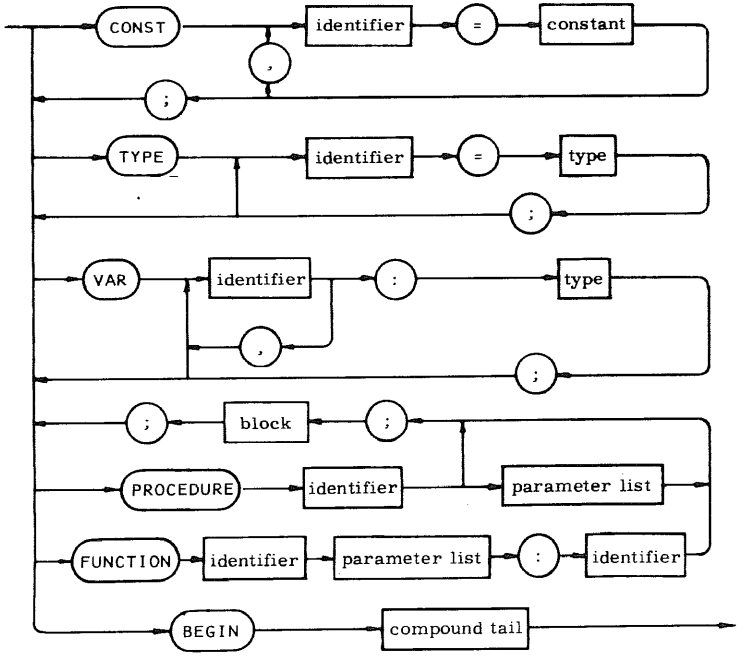
A Pascal program has the form of a procedure declaration without heading (cf. also 7.4.).

<program> ::= <constant definition part><type definition part>
<variable declaration part>
<procedure and function declaration part><statement part>.

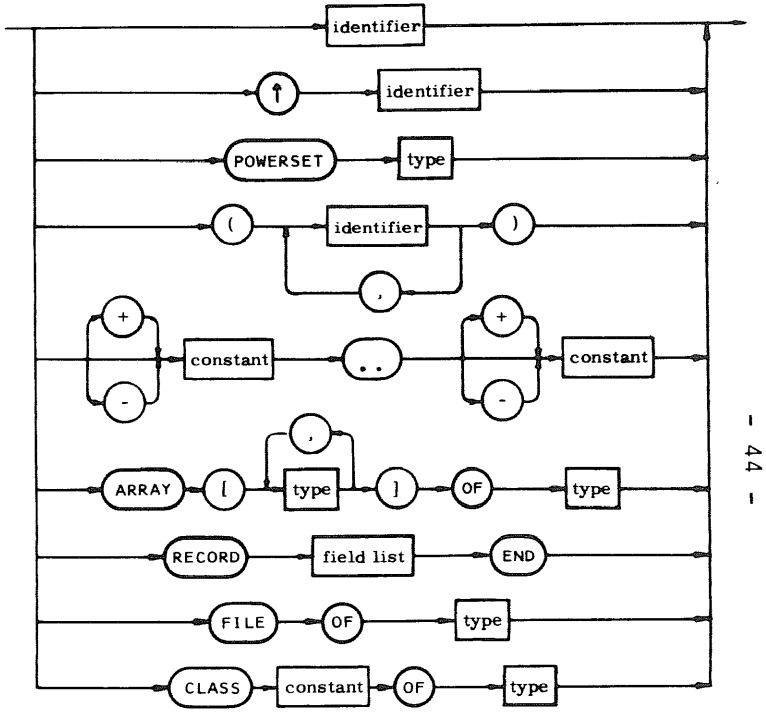
parameter list



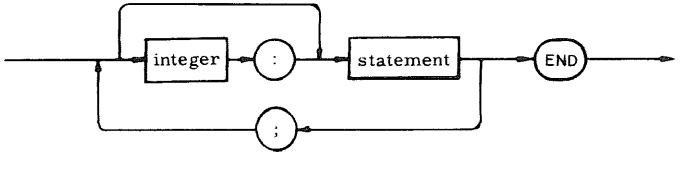

block

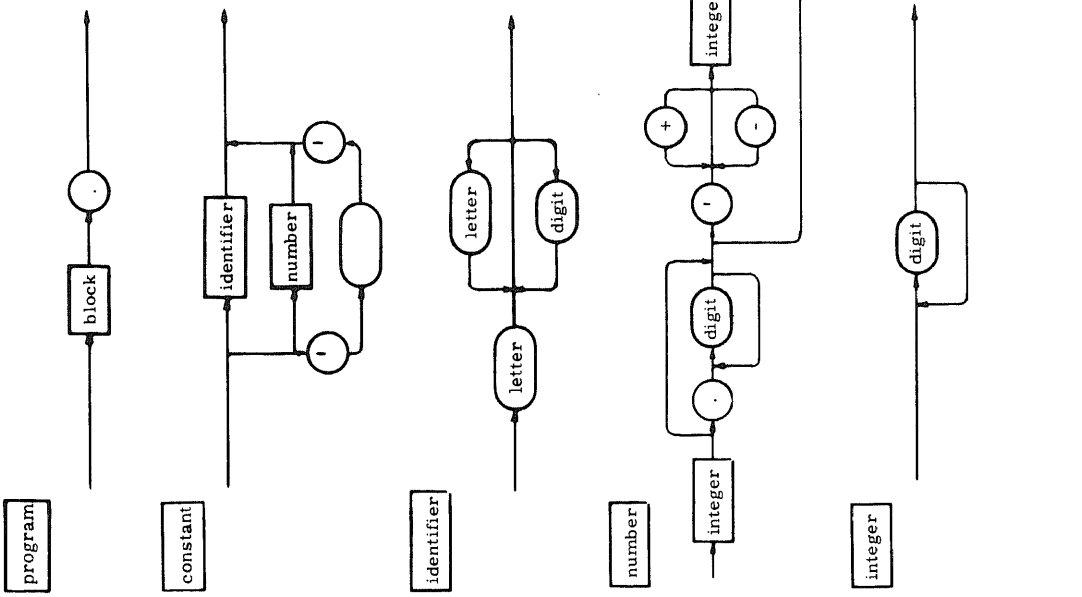


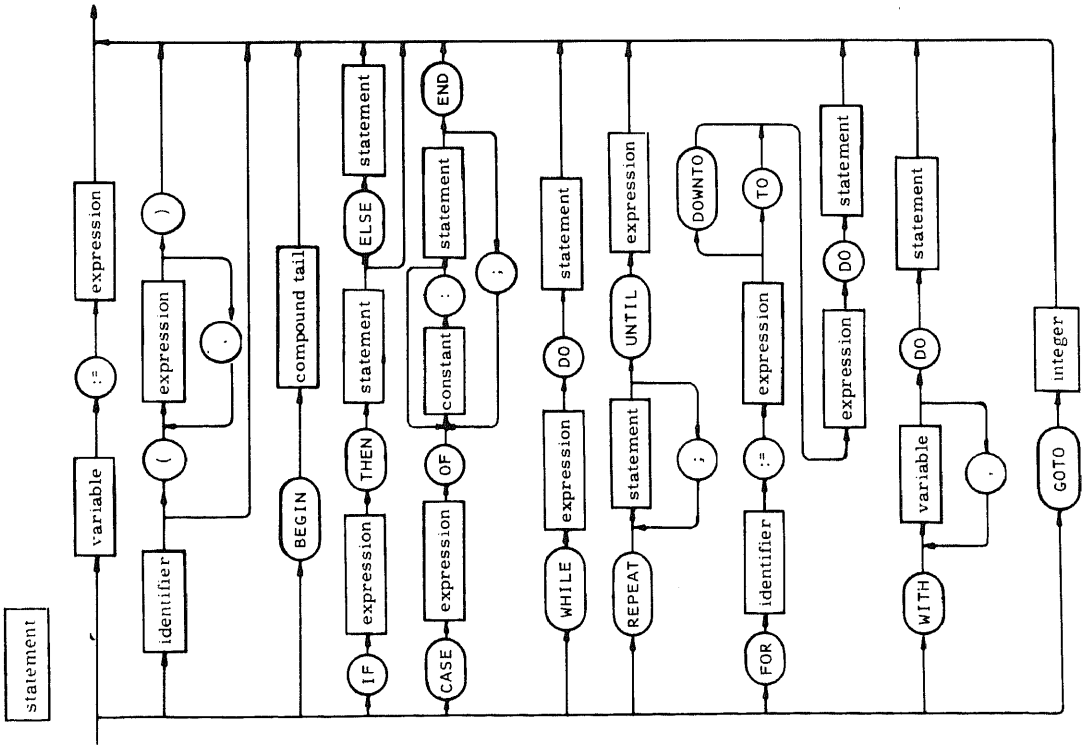

type



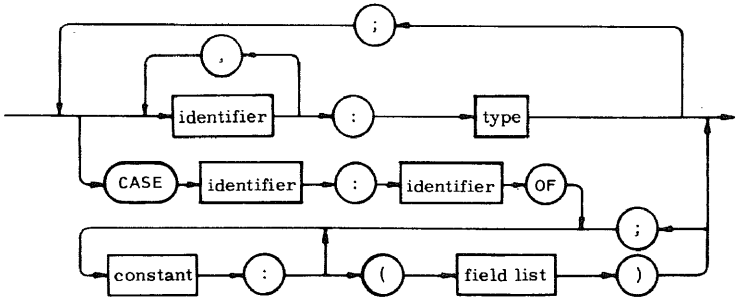

compound tail



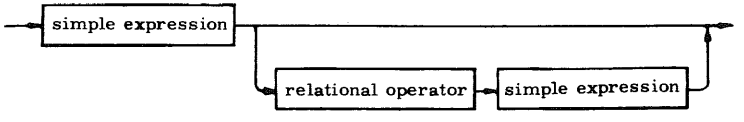

program
block
.
constant
identifier
number
-
identifier
letter
digit
number
integer
.
integer
+
-
integer
digit


statement
variable
:=
expression
identifier
(
expression
)
,
BEGIN
compound tail
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
:
statement
;
END
WHILE
expression
DO
statement
REPEAT
statement
;
UNTIL
expression
FOR
identifier
:=
expression
TO
DOWNTO
expression
DO
statement
WITH
variable
,
DO
statement
GOTO
integer

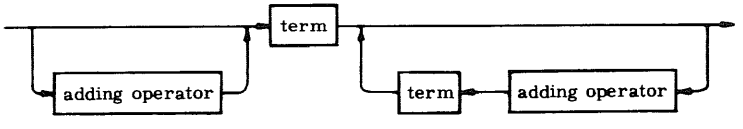
field list



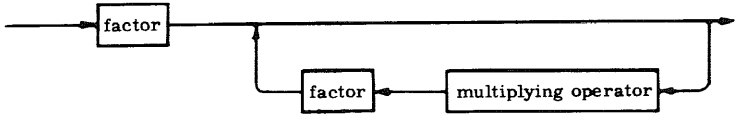

expression



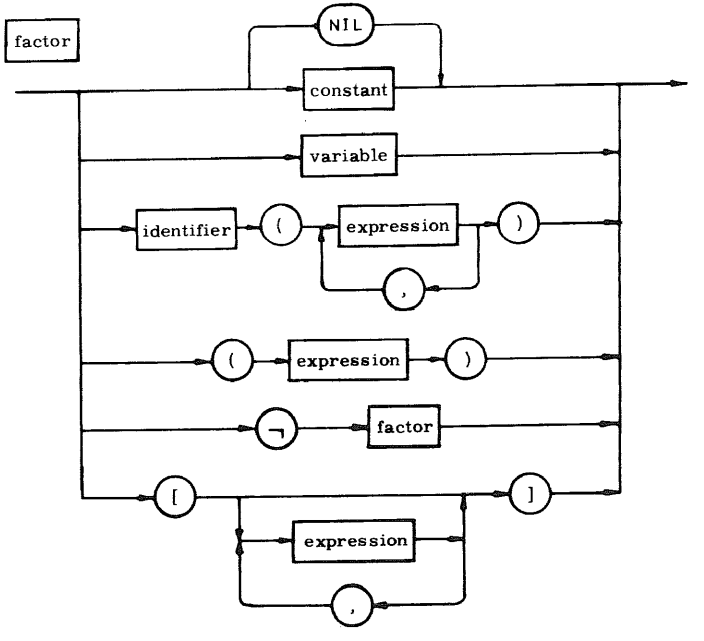

simple expression



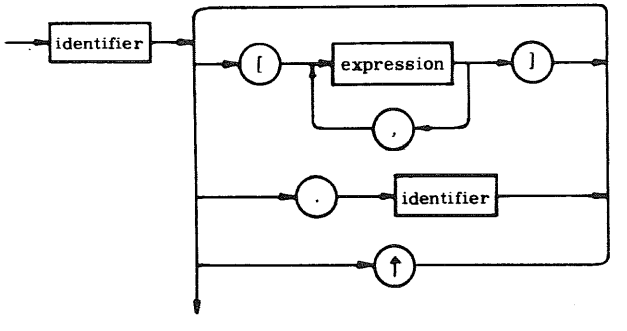

term




factor




variable



## 14. Pascal 6000

The version of the language Pascal which is processed by its implementation on the CDC 6000 series of computers is described by a number of amendments to the preceding Pascal language definition. The amendments specify extensions and restrictions and give precise definitions of certain standard data types. The section numbers used hereafter refer to the corresponding sections of the language definition.

3. Vocabulary

Only capital letters are available in the basic vocabulary of symbols. The symbol packed is added to the vocabulary. (cf.6.2.2.) Symbols which consist of a sequence of underlined letters are called word-delimiters. They are written in Pascal 6000 without underlining and without any surrounding escape characters. Blanks or end-of-lines may be inserted anywhere except within :=, word-delimiters, identifiers, and numbers.

4. Identifiers

Only the 10 first symbols of an identifier are significant. Identifiers not differing in the 10 first symbols are considered as equal.

Word-delimiters must not be used as identifiers. At least one blank space must be inserted between any two word-delimiters or between a word-delimiter and an adjacent identifier.

6. Data types

6.1.1. Standard scalar types

integer — is defined as

type integer = $-2^{48}+1$ .. $2^{48}-1$

real — is defined according to the CDC 6000 floating point format specifications. Arithmetic operations on real type values imply rounding.

char is defined by the CDC 6000 display code character set. This set is incremented by a separator character denoted by the standard identifier eol , meaning "end of line".

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| eol | A | B | C | D | E | F | G | H | I |
| J | K | L | M | N | O | P | Q | R | S |
| T | U | V | W | X | Y | Z | Ø | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 | + | - | * |
| / | ( | ) | $ | = | | , | . | ' | [ |
| ] | : | ≠ | { | ∨ | ∧ | ↑ | } | < | > |
| ≤ | ≥ | ¬ | ; | | | | | | |

(Note that the characters ' { } are special features on the printers of the ETH installation, and correspond to the characters = ↱ ↓ on standard CDC systems.)

alfa the number of characters packed into an alfa value is n=10 (cf. 6.1.1.).

6.2.2. In a record type definition, the symbol record may be preceded by the symbol packed to specify that a compact storage representation is to be used for variables of this type. Fields of packed records must not be used as actual variable parameters, nor as parameters of a formal procedure.

6.2.3. Powerset types

The base type of a powerset type must be either

1. a scalar type with less than 60 values, or
2. a subrange of the type integer, with a minimum element min(T) ≥ 0 and a maximum element max(T) < 59, or
3. a subrange of the type char with the maximum element max(T) < '>'.

6.2.4. and 6.2.5. File and class types

No component of any structured type can be of a file type or of a class type.

7. Variable declarations

File variables declared in the main program may be restricted to either input or output mode by appending the specifiers

[in] or [out]

to the file identifier in its declaration. Files restricted to input mode (input files) are expected to be Permanent Files attached to the job by the SCOPE Attach command, and files restricted to output mode may be catalogued as Permanent Files by the SCOPE Catalog command. In both commands, the file identifier is to be used as the Logical File Name [3].

The specifiers

[print] and [punch]

cause the file to be printed and punched respectively. They are applicable to character files only and restrict the file to the output mode.

Example:

```
f1[in],f2[out],f3[print],f4[punch]; file of char
```

10. and 11. Procedure and function declarations

A procedure or a function which contains local file declarations must not be activated recursively.

10.1.4. Additional Standard Procedures

write(e1,e2 ... en)

e1 ... en are expressions whose values are converted into character sequences which are appended to the standard file OUTPUT. The admissible types of expressions and the lengths of the generated sequences are:

| | | | |
|---|---|---|---|
| char | 1 | | |
| integer | 10 | (or 20) | at least one leading blank |
| real | 20 | | |
| Boolean | 10 | | |
| alfa | 10 | | |

Note that a line may contain at most 136 characters, and that termination of a line must be explicitly indicated by writing a character eol, e.g.

```
write(i,j,x+y,eol)
```

Note further that the first character of each line is not printed but interpreted as a printer control character, and usually should be a blank. '0' causes double spacing, '1' skipping to the top of a new page.

read(v1,v2 ... vn)

for each variable $v_i$, the standard file INPUT is inspected. If $v_i$ is of type char, then the next character is assigned to $v_i$ (i.e. "read(v)" is equivalent to "get(input); v := input↑ "); if $v_i$ is of type integer or real, the file is scanned for a sequence of characters representing a number according to the syntax of Pascal. The number may be preceded by separator characters and a sign; separator characters are any characters except digits, apostrophes, and periods. The first character after the number is also read. The number is then assigned to the variable $v_i$.

text('t')

copies the text t onto the standard file OUTPUT. t may be any sequence of characters not containing apostrophes. This statement must be placed within one line of the program.

## 15. How to use the Pascal 6000 System

### 1. Control statements

In order to activate the Pascal compiler, the following two control statements are required [3]:

```
ATTACH(PASCAL,PASCALSYSTEM)
PASCAL.
```

(PASCALSYSTEM is a Public Permanent File in the RZETH system).
A Pascal job consists of the following parts:

```
Jobcard (minimum field length is 46000_8)
Control statements
EOR (end of record card)
Pascal program
Input data (optional)
EOI (end of information card)
```

The control statement PASCAL may optionally contain any of the following parameters (in any order):

```
PASCAL(P=program,D=data,L=listing,R=results,
                        LL=linelimit,FL=fieldlength)
```

program, data, listing, and results are file names (cf. section 7); linelimit and fieldlength are numbers. The default values are:

| | | |
|---|---|---|
| program | = | INPUT |
| data | = | INPUT |
| listing | = | OUTPUT |
| results | = | OUTPUT |
| linelimit | = | 500 |
| fieldlength | = | current fieldlength (octal) |

## 2. Compiler instructions

The compiler may be instructed to generate code according to certain options; in particular, it may be requested to insert various run-time test instructions into the generated code. Compiler instructions are written as comments and are designated as such by a $-character as the first character of the comment:

{$<instruction sequence> <any comment>}

The instruction sequence is a sequence of instructions separated by commas, and each instruction consists of a letter designating the option followed by a + sign (activation the option) or a - sign (passivating the option).
The following options are available:

A include run-time tests for all assignments to variables of subrange type. Check whether the assigned value lies within the specified range.

D include tests preceding all divisions to check against zero divisors.

I include tests in all automatic integer to real conversions to assure that the converted value satisfies $|i| < 2^{48}$.

R compile real arithmetic operations with rounding.

X include run-time tests to assure that all index values lie within the specified index bounds. This applies to array indices as well as case statements.

C following each procedure, list the compiled instructions in the form of COMPASS assembly code.

The default conventions are

{$A-,D-,I-,R+,X-,C-}

The expansion of code and the degradation in execution speed may be considerable in case of selection of options A, I and X; they are small for option D. The R option involves no additional expense. The C option must be used with great care, since it generates large amounts of output.

## 3. Compiler error messages

The compiler indicates detected errors by an arrow pointing to the relevant place in the text and by a number referring to the following table:

1. scalar type expected
2. integer too large
3. error in constant
4. = expected
5. field name declared twice
6. bad range
7. tagfield type bad
8. name declared twice
9. ) expected
10. : expected
11. identifier expected
12. identifier not declared
13. index must be of scalar type

14. of expected
15. variable type is not class
16. procedure declared twice
17. end expected
18. error in type declaration
19. error in variable declaration
21. error in procedure declaration
23. parameter list ignored
24. error in declaration part
25. lowbound > highbound
26. not a variable identifier
28. symbolic subrange type not allowed
29. parameters missing in function declaration
30. component type is class or file
31. undeclared identifier
32. variable or field identifier expected
33. expression too complicated
34. type of variable should be array
35. type of expression must be scalar
36. conflict of index type with declaration
37. ] expected
38. type of variable should be record
39. no such field in this record
40. type of variable should be pointer or file
41. field name expected
42. illegal symbol in expression
43. undefined label
44. illegal type of parameter in standard function or standard procedure
45. type identifier in statement part
46. procedure used as function
47. type of standard function parameter should be integer
48. ) expected
49. identifier expected
50. illegal type of operand
51. V cannot be used as monadic operator
52. := expected
53. assignment not allowed
54. illegal symbol in statement
55. type or constant identifier
56. then expected
57. type of expression is not Boolean
58. ; expected
59. do expected
60. illegal parameter substitution
61. label expected
62. illegal type of expression
63. constant expected
64. : expected

65. of expected
66. tagfield missing for this variant
67. until expected
68. end expected
69. loop control variable must be simple and local or global
70. to/downto expected
71. too many cases in case statement
72. number of parameters does not agree with declaration
73. mixed types
74. too many labels in this procedure
75. too many (long) constants or yet undefined labels in this procedure
76. depth of procedure nesting too large
77. label defined more than once
78. too many exit labels
79. ( expected
80. , expected
81. assignment to formal function identifier illegal
82. too many nested with-statements
83. standard inline procedure / function used as actual parameter
84. too many (long) constants in this procedure
85. assignment to function identifier must occur in function itself
86. actual parameter must be a variable
87. packed field not allowed here
88. operators < and > are not defined for powersets
89. redundant operation on powersets
90. procedure too long
91. too many exit labels or forward procedures
92. too many class or file variables
93. bad function type
94. = , ≠ not allowed here
95. bad file declaration
96. type declared twice
97. end. encountered
98. [ expected
99. index out of range
100. label too large
101. value is out of range
102. division by zero
103. parameter procedure has more than 17 parameters

## 16. Glossary

## 17. References

1. N. Wirth and C.A.R. Hoare, "A contribution to the development of Algol", Comm. ACM 9, 6, pp. 413-432 (June 1966)

2. D.E. Knuth, "The Art of Computer Programming", Vol. 1, Addison-Wesley (1968)

3. Control Data 6000 Computer Systems, SCOPE Reference Manual, Pub.No. 60189400