Eidgenössische
Technische
Hochschule
Zürich

*Berichte der
Fachgruppe
Computer-
Wissenschaften*

Niklaus Wirth

The Programming Language Pascal (Revised Report)

Juli 1973 5

Niklaus Wirth

The Programming Language Pascal (Revised Report)

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. This Report may serve as a programmers' manual for PASCAL 6000.

Preface to the Revised Report

The language PASCAL has now been in use for several years, during which considerable experience has been gained through its use, its teaching, and its implementation. Although many reasons suggest that a language should be kept unchanged as soon as it has gained a user community, it would be unwise to ignore this experience and to refrain from making good use of it. This Report therefore describes a revised language which includes some changes suggested by the work of the last two years. It is still of the form of the original definition, and in fact the changes are very few and relatively minor. They concern the following subjects:

- Constant parameters are replaced by value parameters (in the sense of ALGOL 60).
- The class structure is eliminated: pointer variables are bound to a data type instead of a class variable.
- The handling of files is changed such that the buffer variable f↑ always has a defined value except when the condition eof(f) is true.
- Packed records and packed arrays are introduced. As a consequence, the type alfa becomes a special case of a packed character array. The generalization has some consequences on the denotation of strings (formerly called alfa constants).
- All labels require a declaration.

Moreover, there are a few minor syntactic changes, such as the renaming of the powerset structure to set structure.

Implementation efforts on various computers have brought the problem of portability and machine independence of software systems to our closer attention. Many of the above mentioned changes, and also some additional restrictions, were adopted and imposed in the interest of

program portability and machine independent definability. They made it possible to define almost the entire language by a set of abstract axioms and rules of inference. Such a rigorous definition is necessary to be able to prove properties of programs. This rigour and machine independence has notably been achieved without sacrifice in the efficiency of program execution.

The chapter on PASCAL for the CDC 6000 computer has been removed from the Report and replaced by a general chapter on suggested standards for implementation and program interchange. This standard specifies ways to represent programs in terms of available character sets, and lists a number of restrictions on the language with the intent of simplifying implementations. Programs to be used on several computers where PASCAL is available should adhere to this standard.

The two procedures read and write have been included in the set of standard procedures and are described in a new Chapter 13. They now constitute a binding standard for legible input and output.

References

N. Wirth, "The Programming Language PASCAL", ACTA INFORMATICA 1, 35-63, (1971) and Berichte der Fachgruppe Computer-Wissenschaften Nr. 1 (Nov. 1970)

—— "Systematisches Programmieren", Teubner Verlag, Stuttgart, 1972

—— "Systematic Programming", Prentice-Hall, Englewood Cliffs, 1973

C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL", Berichte der Fachgruppe Computer-Wissenschaften Nr. 6 (Nov. 1972)

Contents

Preface

1. Introduction 1

2. Summary of the language 2

3. Notation, terminology, and vocabulary 7

4. Identifiers, Numbers, and Strings 8

5. Constant definitions 10

6. Data type definitions 10
   6.1. Simple types 10
   6.2. Structured types 12
   6.3. Pointer types 15

7. Declarations and denotations of variables 16
   7.1. Entire variables 17
   7.2. Component variables 17
   7.3. Referenced variables 19

8. Expressions 19
   8.1. Operators 21
   8.2. Function designators 22

9. Statements 23
   9.1. Simple statements 23
   9.2. Structured statements 25

10. Procedure declarations 31
    10.1. Standard procedures 34

11. Function declarations 36
    11.1. Standard functions 37

12. Programs 39

13. Input and Output 39

14. A Standard for implementation and program interchange 42

15. Glossary 45

Appendix:

Syntax diagrams 47

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.

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 thought and invention, and that the disorder governing these languages directly imposes itself onto the programming style of the students.

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 pedagogical 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.

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. The syntax of Pascal is summarised in graphical form in the Appendix.

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 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 by 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 identifiers standing for each value in the set. Apart from the definable scalar types, there exist four standard scalar types: Boolean, integer, char, and real. Except for the type Boolean, their values are not denoted by identifiers, but instead by numbers and quotations respectively. These are syntactically distinct from identifiers. The set of values of type char is the character set available on a particular installation.

A type may also be defined as a subrange of a 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 four structuring methods available: array structure, record structure, set structure, and file 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 is not a computable value, but instead is an identifier uniquely denoting the component to be selected. These component identifiers are declared in the record type definition. Again, the time needed to access a selected component does not depend on the selector, and the record 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 set structure defines the 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 by progressing sequentially through the file. A file is generated by sequentially appending components at its end. Consequently, the file type definition does not determine the number of components.

Variables declared in explicit declarations are called static. The declaration associates an identifier with the variable which is usedto refer to the variable. In contrast, variables may be generated by an executable statement. Such a dynamic generation yields a so-called pointer (a substitute for an explicit identifier) which subsequently serves to refer to the variable. This pointer may be assigned to other variables, namely variables of type pointer. Every pointer variable may obtain pointers pointing to variables of the same type T only, and it is said to be bound to this type T. It may, however, also obtain the value nil, which points to no variable. Because pointer variables may also occur as components of structured variables, which are themselves dynamically generated, the use of pointers permits the representation of finite graphs in full generality.

The most fundamental statement is the assignment statement. It specifies that a newly computed value be assigned to a variable (or components 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 types 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.

2. Boolean operators of negation, union (OR), and conjunction (AND).

3. set operators of union, intersection, and set difference.

4. relational operators of equality, inequality, ordering, set membership and set inclusion. The results of relational operations are of type Boolean. The ordering relations apply only to scalar types.

The procedure statement causes the execution of the designated procedure (see below). Assignment and procedure 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 declared 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 declared in the main program, i.e. not local to some procedure, are called global.

A procedure has a fixed number of parameters, each of which is denoted within the procedure 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. There are three kinds of parameters: value parameters, variable parameters, and procedure or function parameters. In the first case, the actual parameter is an expression which is evaluated once. The formal parameter represents a local variable to which the result of this evaluation is assigned before the execution of the procedure (or function). In the case of a variable parameter, the actual parameter is a variable and the formal parameter stands for this variable. Possible indices are evaluated before execution of the procedure (or function). In the case of procedure or function parameters, the actual parameter is a procedure or function identifier.

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 should be avoided within function declarations.

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> ::=
+|-|*|/|v|∧|¬|=|≠|<|>|≤|≥|(|)|[|]|{|}|:=|
.|,|;|:|'|↑|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|set|file|
function|procedure|label|packed|program

The construct

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

may be inserted between 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 meta-symbols like | and ::= .

4. Identifiers, Numbers, and Strings

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 usual decimal notation is used for numbers, which are the constants of the data types integer and real (see 6.1.2). The letter E preceding the scale factor is pronounced as "times 10 to the power of".

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

Examples:

1 100 0.1 5E-3 87.35E+8

Sequences of characters enclosed by quote marks are called strings. Strings consisting of a single character are the constants of the standard type char (see 6.1.2). Strings consisting of n (>1) enclosed characters are the constants of the types (see 6.2.1)

packed array [1..n] of char

Note: If the string is to contain a quote mark, then this quote mark is to be written twice.

<string> ::= '<character>$^{\oplus}$'

Examples:

'A' ';' ''''
'PASCAL' 'THIS IS A STRING'

5. Constant definitions

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

<constant identifier> ::= <identifier>
<unsigned constant> ::= <unsigned number> | <string> |
<constant identifier> | nil
<constant> ::= <unsigned number> | <sign><unsigned number> |
<constant identifier> | <sign><constant identifier> | <string>
<constant definition> ::= <identifier> = <constant>

The following are standard constant identifiers defined in every implementation:

eol = control character denoting end of line = 'eol'

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.

<type> ::= <simple type> | <structured type> | <pointer type>
<type definition> ::= <identifier> = <type>

6.1. Simple types

<simple type> ::= <scalar type> | <subrange type> |
<type identifier>
<type identifier> ::= <identifier>

6.1.1. 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 (except real) are:

succ the succeeding value (in the enumeration)

pred the preceding value (in the enumeration)

6.1.2. Standard scalar types

The following types are standard in Pascal:

integer The values are a subset of the whole numbers defined by individual implementations. Its values are the integers (see 4.).

real Its values are a subset of the real numbers depending on the particular implementation. The values are denoted by real numbers (see 4.).

Boolean Its values are the truth values denoted by the identifiers true and false.

char Its values are a set of characters determined by particular implementations. They are denoted by the characters themselves enclosed within quotes.

6.1.3. Subrange types

A type may be defined as a subrange of another scalar type by indication of the least and the largest 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

A structured type is characterised by the type(s) of its components and by its structuring method. Moreover, a structured type definition may contain an indication of the preferred data representation. If a definition is prefixed with the symbol packed, this has no effect on the meaning of a program, but is a hint to the compiler that storage should be economised even at the price of some loss in efficiency of access, and even if this may expand the code necessary for expressing access to components of the structure.

```
<structured type> ::= <unpacked structured type> |
        packed <unpacked structured type>
<unpacked structured type> ::= <array type> |
    <record type> | <set type> | <file type>
```

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> ::= <simple type>
<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 [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 idenrified 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 list> : (<field list>)|<case label list>:
<case label list> ::= <case label> {,<case label>}
<case label> ::= <constant>
<tag field> ::= <identifier>

Examples:

```
record day: 1..31;
       month: 1..12;
       year: integer
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. Set types

A set type defines the range of values which is the powerset of its so-called base type. Base types must not be structured types. Operators applicable to all set types are:

∨	union
∧	intersection
-	set difference
in	membership

<set type> ::= set of <base type>
<base type> ::= <simple type>

6.2.4. File types

A file type definition specifies a structure consisting of a sequence of components which are all of the same type. The number of components, called the length of the file, is not fixed by the file type definition. A file with 0 components is called empty, and files with components of type char are called textfiles.

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

The following is a standard type:

type text = packed file of char

6.3. Pointer types

Variables which are declared in a program (see 7.) are accessible by their identifier. They exist during the entire execution process of the procedure (scope) to which the variable is local, and these variables are therefore called static (or statically allocated). In contrast, variables may also be generated dynamically, i.e. without any correlation to the structure of the program. These dynamic variables are generated by the standard procedure new (see 10.1.2); since they do not occur in an explicit variable declaration, they cannot be referred to by a name. Instead, access is achieved via a so-called pointer value which is provided upon generation of the dynamic variable. A pointer type thus consists of an unbounded set of values pointing to elements of the same type. No operations are defined on pointers except the test for equality.

The pointer value nil belongs to every pointer type; it points to no element at all.

<pointer type> ::= ↑<type identifier>

Examples of type definitions:

```
Color     = (red, yellow, green, blue)
Sex       = (male, female)
Text      = file of char
Shape     = (triangle, rectangle, circle)
Card      = array[1..80] of char
Alfa      = packed array[1..alfaleng] of char
Complex   = record re, im: real end
Person    = record name, firstname: alfa;
                   age: integer;
                   married: Boolean;
                   father, child, sibling: ↑Person;
              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>

Every declaration of a file variable f with components of type T implies the additional declaration of a so-called buffer variable of type T . This buffer variable is denoted by f↑ and serves to append components to the file during generation, and to access the file during inspection (see 7.2.3 and 10.1.1).

The standard file variables input and output are predeclared as textfiles. A Pascal program should be regarded as a procedure with these two variables as formal parameters. The corresponding actual parameters are expected to be either 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, minus, times)
a: array[0..63] of real
b: array[Color,Boolean] of Complex
c: Color
f: file of char
hue1, hue2: set of Color
p1,p2: ↑Person
```

Denotations of variables either designate an entire variable, a component of a variable, or a variable referenced by a pointer (see 6.3). Variables in examples in subsequent chapters are assumed to be declared as indicated above.

<variable> ::= <entire variable> | <component variable> | <referenced 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> | <file buffer>

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]
```

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.re
b[red,true].im
p2↑.size
```

7.2.3. File buffers

At any time, only the one component determined by the current file position (read/write head) is directly accessible. This component is called the current file component and is represented by the file's buffer variable.

<file buffer> ::= <file variable>↑
<file variable> ::= <variable>

7.3. Referenced variables

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

If p is a pointer variable which is bound to a type T , p denotes that variable and its pointer value, whereas p↑ denotes the variable of type T referenced by p .

Examples:

p1↑.father
p1↑.sibling↑.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. The 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 integer	integer, if both operands are of type integer, real otherwise
/	division	real integer	real
div	division with truncation	integer	integer
mod	modulus	integer	integer
∧	logical "and" set intersection	Boolean any set type T	Boolean T

8.1.3. Adding operators

<adding operator> ::= + | - | ∨

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

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 types)	Boolean
< > ≤ ≥	any scalar or subrange type	Boolean
in	any scalar or subrange type and its set 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 set type, and then denote set inclusion ⊆ and ⊇ respectively.

The operators <, ≤, ≥, > may also be applied to packed arrays with components of type char, and then denote alphabetical ordering according to the underlying set of characters.

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

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

9. Statements

Statements denote algorithmic actions, and are said to be executable. They may be provided with a label which can be referenced by goto statements.

<statement> ::= <unlabelled statement> | <label>:<unlabelled statement>
<unlabelled statement> ::= <simple statement> | <structured statement>
<label> ::= <unsigned integer>

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> | <empty statement>

9.1.1. Assignment statements

The assignment statement serves to replace the current value of a variable by a new value specified as an expression.

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

The variable (or the function) and the expression must be of identical 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, or vice-versa.

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: so-called value parameters, variable parameters, procedure parameters (the actual parameter is a procedure identifier), and function parameters (the actual parameter is a function identifier).

In the case of a value parameter, the actual parameter must be an expression (of which a variable is a simple case). The corresponding formal parameter represents a local variable of the called procedure, and the current value of the expression is initially assigned to this variable. In the case of a variable parameter, the actual parameter must be a variable, and the corresponding formal parameter represents this actual variable during the entire execution of the procedure. If this variable is a component of an array, its index is evaluated when the procedure is called. A variable parameter must be used whenever the parameter represents a result of the procedure.

```
<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 (fct,-1.0,+1.0,x)
```

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.

<goto statement> ::= goto <label>

The following restrictions hold concerning the applicability of labels:

1. The scope of a label is the procedure within which it is defined. It is therefore not possible to jump into a procedure.
2. Every label must be specified in a label declaration in the heading of the procedure in which the label marks a statement.

9.1.4. The empty statement

The empty statement consists of no symbols and denotes no actions.

<empty statement> ::=

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. The symbols begin and end act as statement brackets.

<compound statement> ::= begin <statement> {;<statement>} end

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 list> : <statement>
<case label list> ::= <case label> {,<case label>}

Examples:

```
case operator of
   plus:   x := x+y;
   minus:  x := x-y;
   times:  x := x*y
end
```

```
case i of
   1:   x := sin(x);
   2:   x := cos(x);
   3:   x := exp(x);
   4:   x := ln(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 do i := i+1

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


while ¬eof(f) do
begin P(f↑);  get(f)
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 P(f↑); get(f)
until 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), and must not be altered by the repeated statement.

A for statement of the form

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

is equivalent to the sequence of statements

```
v := e1; S; v := succ(v); S; ...; v := e2; S
```

and a for statement of the form

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

is equivalent to the statement

```
v := e1; S; v := pred(S); S; ...; v := e2; S
```

Note: The final value of the control variable is left undefined.

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 Q(c)
```

9.2.4. With statements

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

<record variable list> ::= <record variable> {, <record variable>}

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
if month = 12 then
   begin month := 1;  year := year+1
   end
else month := month+1
```

is equivalent to

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

No assignments may be made by the qualified statement to any constituents of the record variable list.

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 value-, 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> |
var <parameter group> |
function <parameter group> |
procedure <identifier> {,<identifier>}
<parameter group> ::= <identifier>{,<identifier>}:
<type identifier>

A parameter group without preceding specifier implies that its constituents are value parameters.

The label declaration part specifies all labels which mark a statement in the statement part.

<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 definitions 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 := ord(input↑) - ord('0');
          i := 10*i + j;
          get(input)
    end;
    x := i
end
```

```
procedure Bisect(function f: real;  a,b: real;  var z: real);
    var m: real;
begin {assume f(a) < 0 and f(b) > 0}
     while abs(a-b) > 1E-10*abs(a) do
     begin m := (a+b)/2.0;
           if f(m) < 0 then a := m else b := m
     end;
     z := m
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}
    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;
          a2 := a2 - q*a1;  b2 := b2 - q*b1;
          c := d;  d := r;
          r := a1; a1 := a2;  a2 := r;
          r := b1;  b1 := b2;  b2 := r
    end;
    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 program, no conflict arises from a declaration redefining the same identifier within the program. The standard procedures are listed and explained below.

10.1.1. File handling procedures

put(f) appends the value of the buffer variable f↑ to the file f. The effect is defined only if prior to execution the predicate eof(f) is true. eof(f) remains true, and f↑ becomes undefined.

get(f) advances the current file position (read/write head) to the next component, and assigns the value of this component to the buffer variable f↑. If no next component exists, then eof(f) becomes true, and the value of f↑ is not defined. The effect of get(f) is defined only if eof(f) = false prior to its execution. (see 11.1.2)

reset(f) resets the current file position to its beginning and assigns to the buffer variable f↑ the value of the first element of f . eof(f) becomes false, if f is not empty; otherwise f↑ is not defined, and eof(f) remains true.

rewrite(f) discards the current value of f such that a new file may be generated. eof(f) becomes true.

10.1.2. Dynamic allocation procedure

new(p) allocates a new variable v and assigns the pointer to v to the pointer variable p . If the type of v is a record type with variants, the form

new(p,t_1,...,t_n) can be used to allocate a variable of the variant with tag field values t_1,..t_n . The allocation then implies initialization of the tag fields. The values of the tag fields must subsequently remain constant.

10.1.3. Data transfer procedures

Let a be an array variable of type

array[m..n] of T

and let z be a variable of type

packed array[u..v] of T

where n-m $\geq$ v-u . Then

pack(a,i,z) means

```
for j := u to v do  z[j] := a[j-u+i]
```

unpack(z,a,i) means

```
for j := u to v do  a[j-u+i] := z[j]
```

where j denotes an auxiliary variable not occuring elsewhere in the program.

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, and the type of the function.

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

The type of the function must be a scalar, subrange, or 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.

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
    until abs(x1-x0) < eps*x1;
    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...ai-1)}
        if x < a[i] then x := a[i]
     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;
        w := sqr(w)
     end;
     {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 ≠ 0

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

11.1.3. Transfer functions

trunc(x) the real value x is truncated to its integral part.

round(x) the real argument x is rounded.

ord(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).

all(T) is the set of all values of type T .

12. Programs

A Pascal program has the form of a procedure declaration except for its heading.

```
<program> ::= <program heading><label declaration part>
    <constant definition part><type definition part>
    <variable declaration part>
    <procedure and function declaration part><statement part>.
<program heading> ::= <empty>| program <identifier>;|
    program <identifier> (<program parameters>);
```

The form of the program parameters is specified by individual implementations and depends on the available operating system. It is the purpose of these parameters to specify all variables which are not local to the program and upon which the program operates.

The two textfiles input and output are understood to be predefined as parameters and must not be listed explicitly. If they are the only program parameters, the program heading may therefore be omitted. The methods of specifying formal program parameters and corresponding actual parameters upon activation of the program are defined by individual implementations of PASCAL.

13. Input and Output

The basis of legible input and output is established in PASCAL by the two standard text file variables (program parameters) input and output and the standard file procedures get and put (cf. 10.1.1). In order to facilitate the analysis of input text and the formation of output text, the two standard procedures read and write are introduced. They can be used with a variable number of parameters, and with a non-standard syntax for procedure calls. The following rules hold for the procedure read:

1. read(c1, c2 ... cn) means
   begin read(c1); read(c2); ... read(cn) end

2. its parameters may be of type char, integer, or real. In the first case, only the one next character is read; i.e. read(c) stands for

```
c := input↑; get(input)
```

In the latter two cases, a sequence of characters is read which represents an integer or a real number according to the PASCAL syntax (see example on p. 33). (Consecutive numbers must be separated by blanks or end-of-lines.)

Examples:
Read a text, execute P for every character, Q upon end of line:

```
while ¬eof(input) do
begin read(ch);
      while ch ≠ eol do
      begin P(ch); read(ch)
      end;
   Q
end
```

Read a sequence of integers; the last is followed by a period:

```
repeat read(i,ch); P(i)
until ch = '.'
```

The following rules hold for the procedure write:

1. write(p1, p2 ... pn) means

```
begin write(p1); write(p2); ... write(pn) end
```

2. every parameter e may be of type char, integer, real, Boolean, or of any array of characters.

3. if a parameter is of type char, then write(c) stands for

```
output↑ := c; put(output)
```

4. if an argument e is of type integer, then the parameter must be of the form

```
e   or   e:e1
```

where e and e1 are expressions. The value e is converted

into a sequence of e1 characters representing the integer e in decimal form. e1 is called the "field width"; if it is omitted, the default value 10 is assumed.

5. if an argument e is of type real, then the parameter must be of the form

```
e          or
e:e1       or
e:e1:e2
```

where e, e1, e2 are expressions. The value e is converted into a sequence of e1 characters representing the real number e in decimal form. If e1 is omitted, the default value 20 is assumed. If e2 is omitted, then e is represented in floating-point form with a decimal scale factor; otherwise a fixed-point form is produced with e2 digits after the decimal point.

6. if an argument e is a character array (string), then write(e) stands for

```
for i := m to n do write(e[i])
```

where m and n are the lower and upper index bounds of e .

7. if an argument e is of type Boolean, then the parameter must be of the form

```
e          or
e:e1
```

If e1 > 5 , the Boolean value true is represented by the four characters TRUE preceded by e1-4 blanks, and false by the five characters FALSE preceded by e1-5 characters. If e1 $\leq$ 5 , then true and false are represented by the single letter T or F , preceded by e1-1 blanks.

Example:
Let k = 135, n = 4, x = 72.83, b = true, c = 'A', then
write(k+k: n, x: 12, x:6:1, '␣␣A', c, b, eol) appends the character sequence

␣270␣␣7.2830E+01␣␣72.8␣␣AA␣␣␣␣␣␣TRUE eol

to the standard file output. Note that the end of each line must be explicitly indicated by an eol character.

14. A standard for implementation and program interchange

A primary motivation for the development of PASCAL was the need for a powerful and flexible language that could be reasonably efficiently implemented on most computers. Its features were to be defined without reference to any particular machine in order to facilitate the interchange of programs. The following set of proposed restrictions is designed as a guideline for implementors and for programmers who anticipate that their programs be used on different computers. The purpose of these standards is to increase the likelihood that different implementations will be compatible, and that programs are transferable from one installation to another.

1. Identifiers denoting distinct objects must differ over their first 8 characters.

2. Labels consist of at most 4 digits.

3. Procedures and functions which are used as parameters to other procedures and functions must have value parameters only. (Consequently, it is not necessary to test at run time whether a parameter is called by value or by address.)

4. A component of a packed structure must not appear as an actual variable parameter. (Consequently, there is no need to pass addresses of partwords, and to test at run time for the internal representation of the actual variable.)

5. The implementor may set a limit to the size of a base type over which a set can be defined. (Consequently, a bit pattern representation may reasonably be used for all sets.)

6. Packed records cannot be compared directly. (Consequently, it is possible to pack records leaving undefined gaps between components.)

7. The identifiers OR , AND , and NOT are reserved. (Consequently, they may be used as word-symbols in implementations with character sets not including $\vee$, $\wedge$, and $\neg$.)

8. The first character on each line (following eol) in the standard file output is interpreted as a printer control character with the following meanings:

   blank : single spacing
   '0' : double spacing
   '1' : print on top of next page

Representations of PASCAL in terms of available character sets should obey rules 9-12:

9. Word symbols - such as begin, end etc. - are written as a sequence of letters (without surrounding escape characters). They may not be used as identifiers.

10. Blanks, ends of lines, and comments are considered as separators. An arbitrary number of separators may occur between any two consecutive PASCAL symbols with the following exception: no separators must occur within identifiers, numbers, and word symbols.

11. At least one separator must occur between any pair of consecutive identifiers, numbers, or word symbols.

12. Implementations based on the ASCII or EBCDIC character sets should obey the following translation rules for PASCAL symbols not included in the respective sets:

PASCAL symbols			ASCII characters			EBCDIC characters		
∨	∧	¬	OR	AND	NOT	|	&	¬
≠	≤	≥	<>	<=	>=	¬=	<=	>=
{	}	↑	/*	*/	@	/*	*/	@
[	]		[	]		(.	.)	

Table of standard identifiers

Constants:

false, true, eol

Types:

integer, Boolean, real, char, text

Variables:

input, output

Functions:

abs, sqr, odd, succ, pred, ord, chr, trunc, eof,
sin, cos, exp, ln, sqrt, arctan, round

Procedures:

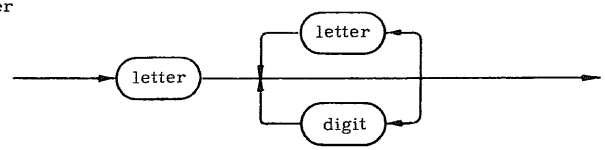
get, put, reset, rewrite, new,
read, write, pack, unpack .

15. Glossary

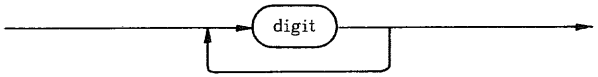
actual parameter	9.1.2
adding operator	8.1.3
array type	6.2.1
array variable	7.2.1
assignment statement	9.1.1
base type	6.2.3
case label	6.2.2
case label list	9.2.2.2 and 6.2.2
case list element	9.2.2.2
case statement	9.2.2.2
component type	6.2.1
component variable	7.2
compound statement	9.2.1
conditional statement	9.2.2
constant	5.
constant definition	5.
constant definition part	10.
constant identifier	5.
control variable	9.2.3.3
digit	3.
empty statement	9.1.4
entire variable	7.1
expression	8.
factor	8.
field designator	7.2.2
field identifier	7.2.2
field list	6.2.2
file buffer	7.2.3
file type	6.2.4
file variable	7.2.3
final value	9.2.3.3
fixed part	6.2.2
for list	9.2.3.3
formal parameter section	10.
for statement	9.2.3.3
function declaration	11.
function designator	8.2
function heading	11.
function identifier	8.2
goto statement	9.1.3
identifier	4.
if statement	9.2.2.1
index type	6.2.1
indexed variable	7.2.1
initial value	9.2.3.3
label	9.
label declaration part	10.
letter	3.
letter or digit	4.
multiplying operator	8.1.2
parameter group	10.

pointer variable 7.3
pointer type 6.3
procedure and function
declaration part 10.
procedure declaration 10.
procedure heading 10.
procedure identifier 9.1.2
procedure or function declaration 10.
procedure statement 9.1.2
program 12.
record section 6.2.2
record type 6.2.2
record variable 7.2.2
record variable list 9.2.4
referenced variable 7.3
relational operator 8.1.4
repeat statement 9.2.3.2
repetitive statement 9.2.3
result type 11.
scale factor 4.
scalar type 6.1.1
set 8.
set type 6.2.3
sign 4.
simple expression 8.
simple statement 9.1
simple type 6.1
special symbol 3.
statement 9.
statement part 10.
string 4.
structured statement 9.2
structured type 6.2
subrange type 6.1.3
tag field 6.2.2
term 8.
type 6.
type definition 6.
type definition part 10.
type identifier 6.1
variable 7.
variable declaration 7.
variable declaration part 10.
variable identifier 7.1
variant 6.2.2
variant part 6.2.2
unlabelled statement 9.
unpacked structured type 6.2
unsigned constant 5.
unsigned integer 4.
unsigned number 4.
unsigned real 4.
with statement 9.2.4
while statement 9.2.3.1

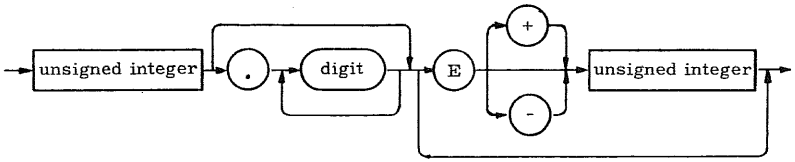
identifier



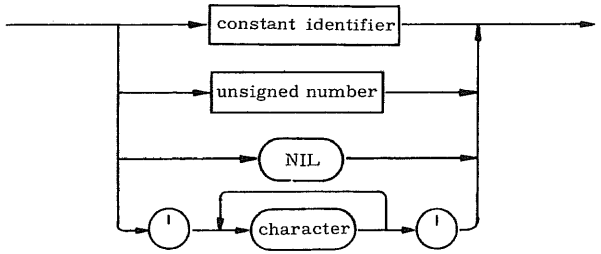
unsigned integer



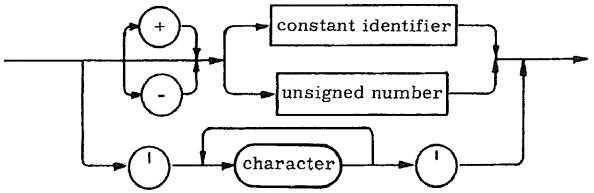
unsigned number



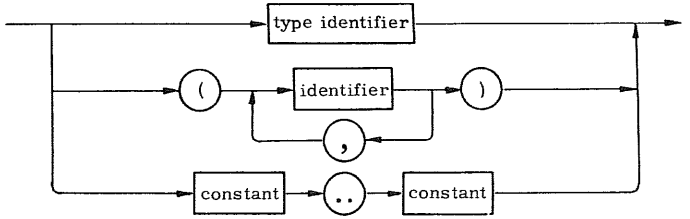
unsigned constant



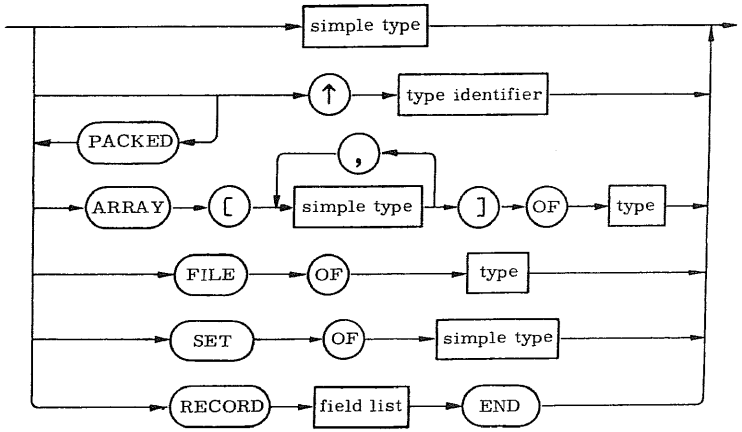
constant



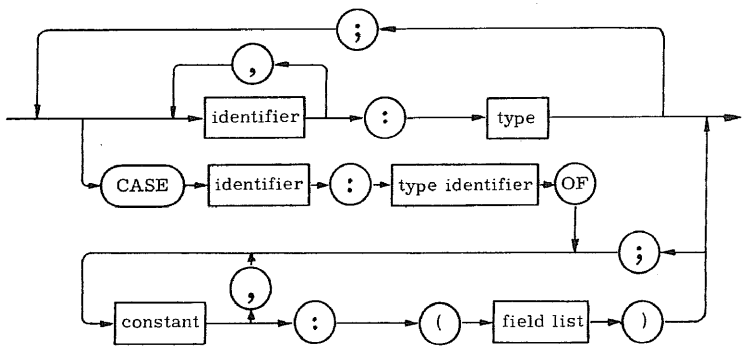
simple type



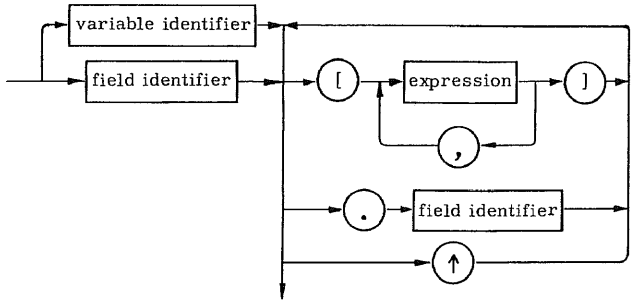
type



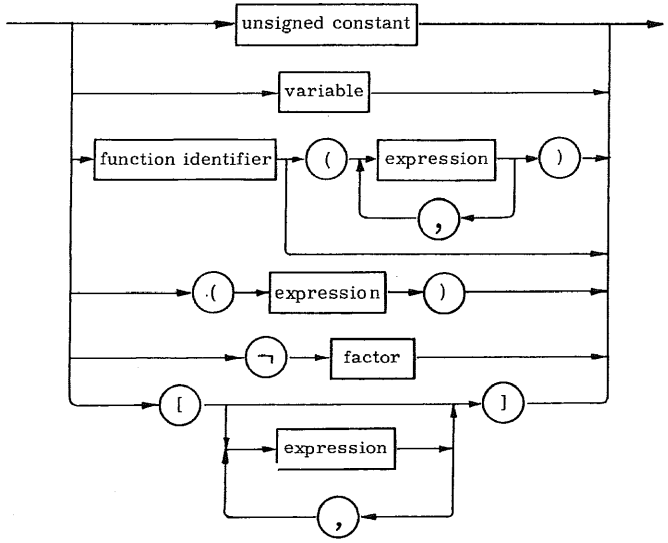
field list



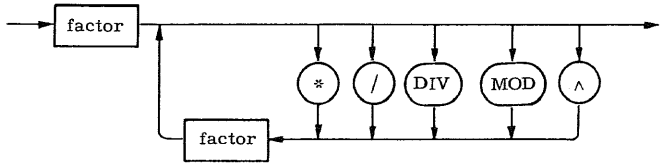
variable



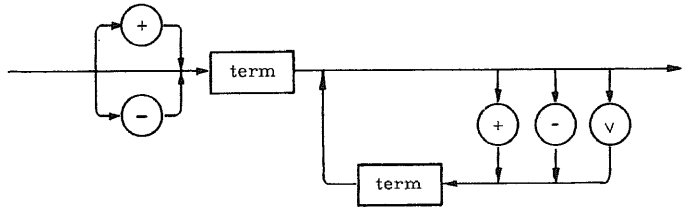
factor



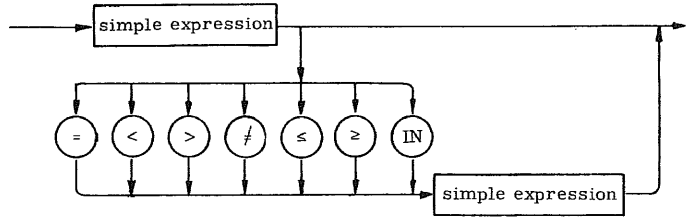
term



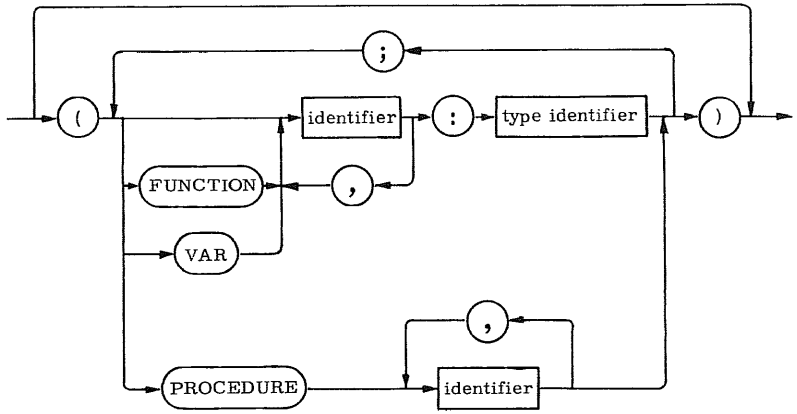
simple expression



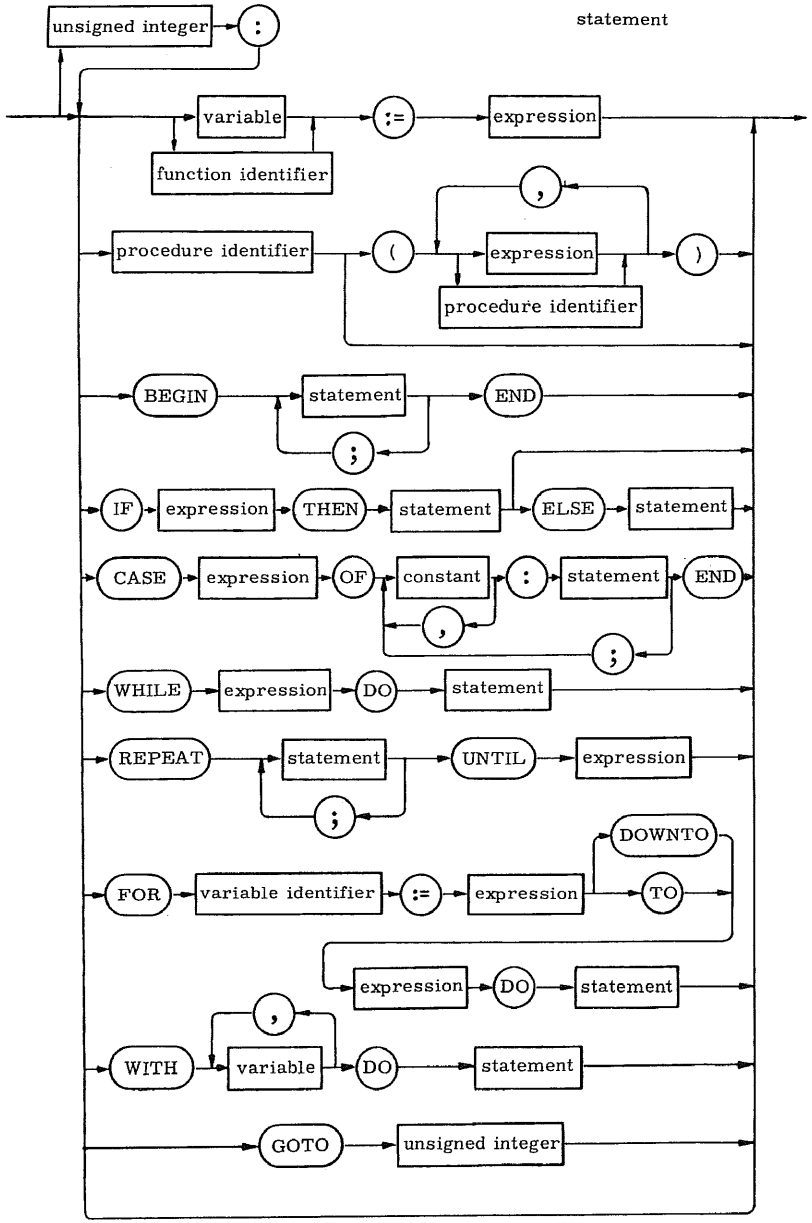
expression



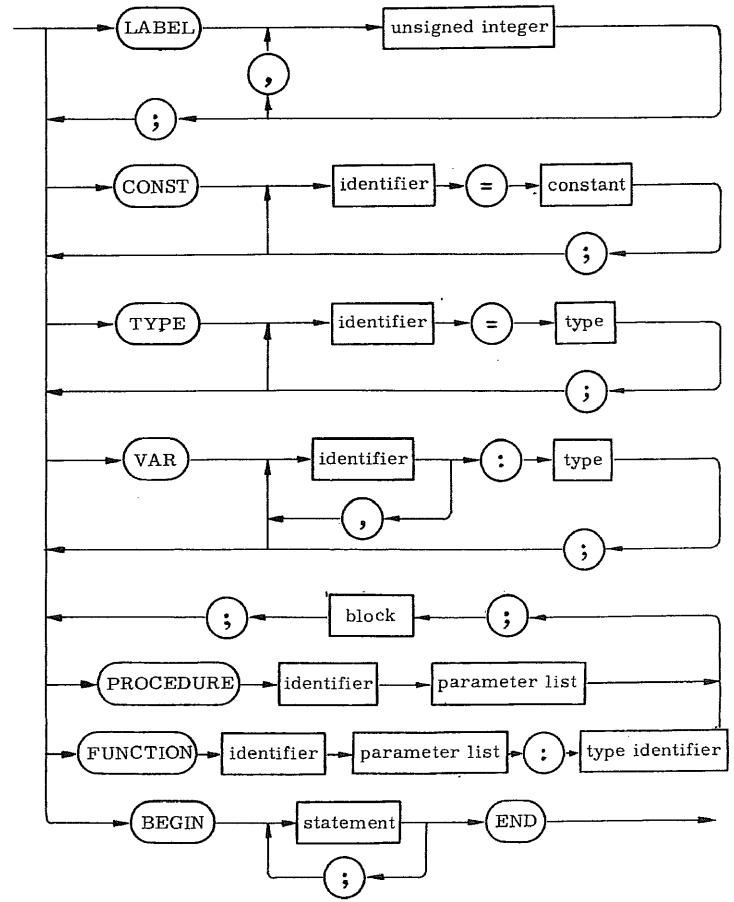
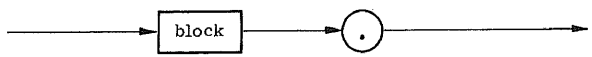
parameter list



statement



block



program