Niklaus Wirth

The Programming Language Pascal

Acta Informatica, Vol. 1, Fasc. 1, 1971
pp. 35-63

Acta Informatica 1, 35–63 (1971)

The Programming Language Pascal

N. WIRTH*

Received October 30, 1970

Summary. 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.

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.

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 chosen objectives, such as a systematic structure, flexibility of program and data structuring, and efficient implementability. In that sense ALGOL 60 was used as a basis for Pascal, since it meets most of these demands to a much higher degree than any other standard language [1]. 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 incom-

* Fachgruppe Computer-Wissenschaften, Eidg. Technische Hochschule, Zürich, Schweiz.

patible 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 obey the rules of the IFIP ALGOL Subset [2].

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 help erase 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 [3]. This language is a direct predecessor of Pascal, and was the source of many features such as e.g. the while and case statements and of record structures.

Pascal has been implemented on the CDC 6000 computers. The compiler is written in Pascal itself as a one-pass system which will be the subject of a subsequent report. 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.

The syntax of Pascal has been kept as simple as possible. Most statements and declarations begin with a unique key word. This property facilitates both the understanding of programs by human readers and the processing by computers. In fact, the syntax has been devised so that Pascal texts can be scanned by the simplest techniques of syntactic analysis. This textual simplicity is particularly desirable, if the compiler is required to possess the capability to detect and diagnose errors and to proceed thereafter in a sensible manner.

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*.

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.

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*. 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. 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 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.

A procedure may have a fixed number of parameters, which are classified into constant-, variable-, procedure-, and 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. 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}.

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⟩ ::= +|−|*|/|∨|∧|¬|=|≠|<|>|≦|≧|(|)|[|]|{|}|:=|
$_{10}$|.|,|;|:|'|↑|**div**|**mod**|**nil**|**in**|
if|**then**|**else**|**case**|**of**|**repeat**|**until**|**while**|**do**|
for|**to**|**downto**|**begin**|**end**|**with**|**goto**|
var|**type**|**array**|**record**|**powerset**|**file**|**class**|
function|**procedure**|**const**

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.

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 $_{10}$ preceding the scale factor is pronounced as "times 10 to the power of".

⟨number⟩ ::= ⟨integer⟩ | ⟨real number⟩
⟨integer⟩ ::= ⟨digit⟩⊕
⟨real number⟩ ::= ⟨digit⟩⊕ . ⟨digit⟩⊕ |
⟨digit⟩⊕ . ⟨digit⟩⊕ $_{10}$ ⟨scale factor⟩ | ⟨integer⟩ $_{10}$ ⟨scale factor⟩
⟨scale factor⟩ ::= ⟨digit⟩⊕ | ⟨sign⟩ ⟨digit⟩⊕
⟨sign⟩ ::= + | −

Examples:

1 100 0.1 $5_{10}-3$ $87.35_{10}+8$

5. Constant Definitions

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

⟨unsigned constant⟩ ::= ⟨number⟩ | '⟨character⟩⊕' | ⟨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 α and β 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) refering 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⟩ :**}**$^{\oplus}$ (⟨field list⟩) | **{**⟨case label⟩**}**$^{\oplus}$
⟨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;
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** ⟨type identifier⟩ | **powerset** ⟨subrange type⟩

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 occurrence: 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 ↑ . *eldersibling* ↑ . *youngestchild*

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
$(x+y+z)$
$sin\,(x+y)$
$[red, c, green]$
$\neg p$

Terms: $x*y$
$i/(1-i)$
$p \wedge q$
$(x \leqq y) \wedge (y < z)$

Simple expressions: $x+y$
$-x$
$hue1 \vee hue2$
$i*j+1$

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 must be of identical type (but neither class nor file type), with the following exceptions 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:

$x := y + 2.5$
$p := (1 \leqq i) \wedge (i < 100)$
$i := sqr(k) - (i * j)$
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-, constant-, 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. 9.2.1.).

⟨goto statement⟩ ::= **goto** ⟨label⟩

⟨label⟩ ::= ⟨integer⟩

The following restriction holds concerning the applicability of labels:

The scope (cf. 10) of a label is the procedure declaration within which it is defined. It is therefore not possible to jump into a procedure.

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 $p \neq$ **nil then** $p := p\uparrow.father$

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⟩**:}**$^{\oplus}$ ⟨statement⟩ | **{**⟨case label⟩**:}**$^{\oplus}$

Example:

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

This statement is equivalent to

```
begin
   if date.month = 12 then
   begin date.month := 1; date.year := date.year + 1
   end else date.month := date.month + 1
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⟩
⟨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 constant-, 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 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); {m ≧ 0, n > 0}
var a1, a2, b1, b2, c, d, q, r: integer;
begin{Greatest Common Divisor x of m and n,
      Extended Euclid's Algorithm, cf. [4], p. 14}
  c := m; d := n;
  a1 := 0; a2 := 1; b1 := 1; b2 := 0;
  while d ≠ 0 do
  begin{a1*m + b1*n = d, a2*m + n2*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 $f\uparrow$ 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 $a[i] \ldots a[i+n-1]$ into the alfa variable *z* (for *n* cf. 6.1.1.), and

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

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 parts, any of which, except the first and the last, may be empty (cf. also 10.).

⟨function declaration⟩ ::=
⟨function heading⟩
⟨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
        {x0² − 2*x1*x0 + x = 0}
    until abs(x1 − x0) ≦ eps;
    {(x0 − eps) ≦ x1 ≦ (x0 + eps),
    (x − 2*eps*x0) ≦ x0² ≦ (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 ... ai−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 $abs(x) - 1 < trunc(abs(x)) \leqq abs(x)$

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⟩.

13. 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 **eol** is added to the vocabulary. 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. The symbol $_{10}$ is written as '.

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 the character denoted by **eol**, signifying end-of-line.

The ordered set is:

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*	0	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 ≡ ⌜ ↓ at standard CDC systems.)

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

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 $min(T) \geqq 0$ 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 [5].

10. and 11. Procedure and Function Declarations

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

14. Glossary

The author gratefully acknowledges his indeptedness to C. A. R. Hoare for his many valuable suggestions concerning overall design strategy as well as details, and for his critical scrutiny of this paper.

References

1. Naur, P.: Report on the algorithmic language ALGOL 60. Comm ACM 3, 299–314 (1960).
2. Report on Subset ALGOL 60 (IFIP): Comm. ACM 7, 626–628 (1964).
3. Wirth, N., Hoare, C. A. R.: A contribution to the development of ALGOL. Comm. ACM 9, 413–432 (1966).
4. Knuth, D. E.: The art of computer programming, Vol. 1. Addison-Wesley 1968.
5. Control Data 6000 Computer Systems, SCOPE Reference Manual, Pub. No. 60189400.

Prof. Dr. N. Wirth
Eidgenössische Technische Hochschule
Fachgruppe Computer-Wissenschaften
Clausiusstraße 55
CH-8006 Zürich
Schweiz