Eidgenössische
Technische
Hochschule
Zürich

Berichte der Fachgruppe Computer-Wissenschaften

C. A. R. Hoare and
N. Wirth

An Axiomatic Definition of the Programming Language Pascal

November 1972 *6*

C. A. R. Hoare and
N. Wirth

An Axiomatic Definition of the Programming Language Pascal

Summary: The axiomatic definition method proposed in reference [3] is extended and applied to define the meaning of the programming language PASCAL [1]. The whole language is covered with the exception of real (floating-point) arithmetic and go to statements.

* Computer Science Department, The Queen's University, Belfast, Northern Ireland.

Contents

An Axiomatic Definition of the Programming Language PASCAL

INTRODUCTION

The programming language PASCAL was designed as a general purpose language efficiently implementable on many computers and sufficiently flexible to be able to serve in many areas of application. Its defining report [1] was given in the style of the ALGOL 60 report [2]. A formalism was used to define the syntax of the language rigorously. But the meaning of programs was verbally described in terms of the meaning of individual syntactic constructs. This approach has the advantage that the report is easily comprehensible, since the formalism is restricted to syntactic matters and is basically straightforward. Its disadvantage is that many semantic aspects of the language remain sufficiently imprecisely defined to give rise to misunderstanding. In particular, the following motivations must be cited for issuing a more complete and rigorous definition of the language:

1. PASCAL is being implemented at various places on different computers [9, 10]. Since one of the principal aims in designing PASCAL was to construct a basis for truly portable software, it is mandatory to ensure full compatibility among implementations. To this end, implementors must be able to rely on a rigorous definition of the language. The definition must clearly state the rules that are considered as binding, and on the other hand give the implementor enough freedom to achieve efficiency by leaving certain less important aspects undefined.

2. PASCAL is being used by many programmers to formulate algorithms as programs. In order to be safe from possible misunderstandings and misconceptions they need a comprehensive reference manual acting as an ultimate arbiter among possible interpretations of certain language features.

3. In order to prove properties of programs written in a language, the programmer must be able to rely on an appropriate logical foundation provided by the definition of that language.

4. The attempt to construct a set of abstract rules rigorously defining the meaning of a language may reveal irregularities of structure or machine dependent features. Thus the development of a formal definition may assist in better language design.

Among the available methods of language definition the axiomatic approach proposed and elaborated by Hoare [3-5] seems to be best suited to satisfy the different aims mentioned. It is based on the specification of certain axioms and rules of inference. The use of notations and concepts from conventional mathematics and logic should help in making this definition more easily accessible and comprehensible. The authors therefore hope that the axiomatic definition may simultaneously serve as

1. a "contract" between the language designer and implementors (including hardware designers),

2. a reference manual for programmers,

3. an axiomatic basis for formal proofs of properties of programs, and

4. an incentive for systematic and machine independent language design and use.

This axiomatic definition covers exclusively the semantic aspects of the language, and it assumes that the reader is familiar with the syntactic structure of PASCAL as defined in [1]. We also consider such topics as rules about the scope of validity of names and priorities of operators as belonging to the realm of syntax.

The axiomatic method in language definition as introduced in [3] operates on four levels of discourse:

1. <u>PASCAL statements</u>, usually denoted by S

2. Logical formulas describing properties of data, usually denoted by P, Q, R

3. Assertions, usually denoted by H, of which there are two kinds:

   3a. Assertions obtained by quantifying on the free variables in a logical formula. They are used to axiomatise the mathematical structured which corresponds to the various data types.

   3b. Assertions of the form $P \{S\} Q$ which express that, if Q is true on termination of the execution of S, then P was true before the execution of S. This kind of assertion is used to define the meaning of assignment and procedure statements.

4. Rules of inference of the form

$$\frac{H_1, \ldots, H_n}{H}$$

which state that whenever $H_1 \ldots H_n$ are true assertions, then H is also a true assertion, or of the form

$$\frac{H_1, \ldots, H_n \vdash H_{n+1}}{H}$$

which states that if H_{n+1} can be proven from $H_1 \ldots H_n$, then H is a true assertion. Such rules of inference are used to axiomatise the meaning of declarations and of structured statements, where $H_1 \ldots H_n$ are assertions on the components of the structured statements.

In addition, the notation

$$P^x_y$$

is used for the formula which is obtained by systematically substituting y for all free occurences of x in P.

The axioms and rules of inference given in this article explicitly forbid the presence of certain "side-effects" in the evaluation of functions and execution of statements. Thus programs which invoke such side-effects are, from a formal point of view, undefined. The absence of such side-effects can in principle be checked by a textual (compile-time) scan of the program. However, it is not obligatory for a PASCAL implementation to make such checks.

The whole language PASCAL is treated in this article with the exception of real (floating-point) arithmetic and go to statements (jumps). With regard to arithmetic, the interested reader is referred to refernces [7] and [8].

The task of rigorously defining the language in terms of machine independent axioms, as well as experience gained in use and implementation of PASCAL have suggested a number of changes with respect to the original description. These changes are informally described in the subsequent section of this article, and must be taken into account whenever refererring to [1]. For easy reference, the revised syntax is summarised in the form of diagrams in the Appendix.

CHANGES AND EXTENSIONS OF PASCAL

The changes which were made to the language PASCAL since it was defined in 1969 and implemented and reported in 1970 can be divided into semantic and syntactic amendments. To the first group belong the changes which affect the meaning of certain language constructs and can thus be considered as essential changes. The second group was primarily motivated by the desire to simplify text analysis or to coordinate notational conventions which thereby become easier to learn and apply.

File types

The notion of the mode of a file is eliminated. The applicability of the procedures put and get is instead reformulated by antecedent conditions in the respective rules of inference. The procedure reset repositions a file to its beginning for the purpose of reading only. A new standard procedure rewrite is introduced to effectively discard the current value of a file variable and to allow the subsequent generation of a new file.

Packed structured types

In order to allow implementations to offer more than one type of internal representation of structured data the facility of packed data structures is introduced. A packed array, file, record, or set structure is specified by prefixing the symbol array, file, record, or set with the symbol packed. It is generally assumed that a packed structure occupies less storage space than its unpacked equivalent, but that on the other hand access to components of the data structure may expand the code and be more time consuming. Of course, the gain in storage economy and loss in efficiency is implementation dependent; in fact, an implementation may entirely ignore the symbol packed. Since the meaning of a program is defined to be independent of the presence or absence of the symbol packed in type definitions.

The type alfa is removed from the language. It may now be defined by the programmer as

type alfa = packed array [1..alfaleng] of char

where alfaleng is a predefined constant with implementation dependent value (i.e. the number of characters fitting into a single "word"). In addition, constants with packed array structure and components of type char are introduced in the form of sequences of characters delimited by quote marks. These constants are called strings. If c_1, c_2 ... c_n are characters, then

$$'c_1\ c_2\ \dots\ c_n'$$

is a constant of type

packed array [1..n] of char

The standard procedures pack and unpack are generalised such that they are applicable to all packed arrays.

Parameters of procedures

Constant parameters are replaced by so-called value parameters in the sense of ALGOL 60. A formal value parameter represents a variable local to the procedure to which the value of the corresponding actual parameter is initially assigned upon activation of the procedure. Assignments to value parameters from within the procedure are permitted, but do not affect the corresponding actual parameter. The symbol const will not be used in a formal parameter list.

Class and pointer types

The class is eliminated as a data structure, and pointer types are bound to a data type instead of a class variable. For example, the type definition and variable declaration

```
type P = ↑c;
var c: class n of T
```

are replaced and expressed more concisely by the single pointer type definition

```
type P = ↑T
```

This change allows the allocation of all dynamically generated variables in a single pool.

The for statement

In the original report, the meaning of the for statement is defined in terms of an equivalent conditional and repetitive statement. It is felt that this algorithmic definition resulted in some undesirable overspecification which unnecessarily constrains the implementor. In contrast, the axiomatic definition presented in this paper leaves the value of the control variable undefined after termination of the for statement. It also involves the restriction that the repeated statement must not change the initial value [6].

Changes of a syntactic nature

- Commas are used instead of colons to separate (multiple) labels in case statements and variant record definitions.
- Semicolons are used instead of commas to separate constant definitions.
- The symbol <u>powerset</u> is replaced by the symbols <u>set</u> <u>of</u>, and the scale symbol $_{10}$ is replaced by the capital letter E .
- The standard procedure alloc is renamed new, and the standard function int is renamed ord .

DATA TYPES

The axioms presented in this and the following sections display the relationship between a type declaration and the axioms which specify the properties of values of the type and operations defined over them. The treatment is not wholly formal, and the reader must be aware that

1. free variables in axioms are assumed to be universally quantified,
2. the expression of the "induction" axiom is always left informal,
3. the types of variables used have to be deduced either from the chapter heading or from the more immediate context,
4. the name of a type is used as a transfer function constructing a value of the type. Such a use of the type identifier is not available in PASCAL.
5. Axioms for a defined type must be modelled after the definition and be applied only in the scope (block) to which the definition is local.
6. A type name (other than that of a pointer type) may not be used directly or indirectly within its own definition.

Scalar types

$$\underline{\text{type}}\ T = (c_1,\ c_2\ \dots\ c_n)$$

1.1. $c_1,\ c_2\ \dots\ c_n$ are distinct elements of T.
1.2. These are the only elements of T.
1.3. $c_{i+1} = \text{succ}(c_i)$ for $i = 1 \dots n-1$
1.4. $\text{pred}(\text{succ}(u)) = u$, $\text{succ}(\text{pred}(v)) = v$
1.5. $\neg(v<v)$
1.6. $(u<v) \wedge (v\leq w) \supset u<w$
1.7. $v = \text{succ}(u) \vee u = \text{pred}(v) \supset u < v$

1.8. $(u>v) \equiv (v<u)$

1.9. $(u\leq v) \equiv \neg(u>v)$

1.10. $(u\geq v) \equiv \neg(u<v)$

1.11. $(u\neq v) \equiv \neg(u=v)$

We define $min_T = c_1$ and $max_T = c_n$ (not available to the PASCAL programmer).

The Boolean type

<u>type</u> Boolean = (false,true)

Axioms (1.1) - (1.11) apply to the Boolean type with c_1 = false and c_2 = true. The Boolean operators $\neg$, $\wedge$, and $\vee$ are defined by the following additional axioms.

2.1. $\neg$true = false

2.2. $\neg$false = true

2.3. p $\wedge$ false = false $\wedge$ p = false

2.4. true $\wedge$ true = true

2.5. p $\vee$ true = true $\vee$ p = true

2.6. false $\vee$ false = false

The integer type

3.1. 0 is an integer.

3.2. If n is an integer, then succ(n) and pred(n) are integers.

3.3. These are the only integers.

Axioms (1.4) - (1.11) apply to the integer type. The operators + - * <u>div</u>, <u>mod</u>, and the functions abs, sqr, and odd are defined by the following additional axioms.

3.4. n+0 = n

3.5. m+n = pred(m) + succ(n) = succ(m) + pred(n)

3.6. n-0 = n

3.7. m-n = succ(m) - succ(n) = pred(m) - pred(n)

3.8. $n*0 = 0$

3.9. $m*n = (m*succ(n)) - m = (m*pred(n)) + m$

3.10. $(m \geq 0) \wedge (n>0) \supset m-n < (m \underline{div} n)*n \leq m$

3.11. $m \underline{mod} n = m-((m \underline{div} n)*n)$

3.12. $n \geq 0 \supset abs(n) = n$

3.13. $n<0 \supset abs(n) = -n$

3.14. $sqr(n) = n*n$

3.15. $odd(n) = ((n \underline{mod} 2) = 1)$

3.16. 1 means succ(0)

2 means succ(1)

...

9 means succ(8)

3.17. if $d_0, d_1 \ldots d_n$ are digits, then $d_n \ldots d_1 d_0$ means

$$10^n d_n + \ldots + 10^1 d_1 + 10^0 d_0$$

These axioms describe the conventional infinite range of integers. Implementations are permitted to refuse to complete the execution of programs which attempt to refer to integers larger than max_{int} or smaller than min_{int}. The result of division is deliberately left undefined for negative arguments.

<u>The char type</u>

4.1. The elements of the type char are the <u>letters</u>

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

the <u>digits</u>

0 1 2 3 4 5 6 7 8 9

and possibly other characters defined by particular implementations. In programs, a constant of type char is denoted by enclosing the character in quote marks.

4.2. $'A' < 'B'$ $\quad '1' = succ('0')$

$'B' < 'C'$ $\quad '2' = succ('1')$

$\dots$ $\quad \dots$

$'Y' < 'Z'$ $\quad '9' = succ('8')$

The sets of letters and digits are ordered

Axioms (1.4) - (1.11) apply to the char type. The functions ord and chr are defined by the following additional axioms:

4.3. if u is an element of char, then ord(u) is a non-negative integer (called the <u>ordinal number</u> of u), and

$$chr(ord(u)) = u$$

4.4. $u < v \equiv ord(u) < ord(v)$

These axioms have been designed to facilitate interchange of programs between implementations using different character sets. It should be noted that the function ord does not necessarily map the characters onto consecutive integers.

<u>Subrange types</u>

$$\underline{type}\ T = m\ ..\ n$$

Let a,b,m,n, be elements of T_0 such that

$$m \leq a \leq b \leq n$$

and let x,y be elements of T . Then we define

$$min_T = m \quad \text{and} \quad max_T = n$$

5.1. T(a) is an element of T .

5.2. These are the only elements of T .

5.3. $T^{-1}(T(a)) = a$

5.4. If $\ominus$ is a monadic operator defined on T_0 , then

$$\ominus x \quad \text{means} \quad \ominus T^{-1}(x)$$

5.5. If $\odot$ is a dyadic operator defined on $T_0 \times T_0$, then

$x \circledcirc y$ means $T^{-1}(x) \circledcirc T^{-1}(y)$

$x \circledcirc a$ means $T^{-1}(x) \circledcirc a$

$a \circledcirc x$ means $a \circledcirc T^{-1}(x)$

Array types

<u>type</u> T = <u>array</u>[I] <u>of</u> T_o

Let $m = min_I$ and $n = max_I$.

6.1. If x_i is an element of T_o for all i such that $m \leq i \leq n$, then $T(x_m \ldots x_n)$ is an element of T .

6.2. These are the only elements of T .

6.3. $m \leq i \leq n \supset T(x_m \ldots x_n)[i] = x_i$

6.4. <u>array</u> $[I_1, I_2 \ldots I_k]$ <u>of</u> T_o means
<u>array</u> $[I_1]$ <u>of</u> <u>array</u> $[I_2 \ldots I_k]$ <u>of</u> T_o

6.5. $x[i_1, i_2 \ldots i_k]$ means $x[i_1][i_2 \ldots i_k]$

We introduce the following abbrevation for later use:
(x,i:y) stands for
T(x[m] ... x[pred(i)],y,x[succ(i)]...x[n])

If in an array type definition the symbol <u>array</u> is preceded by the symbol <u>packed</u>, this is to be interpreted as a comment to the implementation with no further consequences to the meaning of the program.

If the components of a packed array are of type char, i.e.

<u>type</u> T = <u>packed</u> <u>array</u> [I] <u>of</u> char

then the following additional axioms hold:

6.6. Let $x = T(x_1, x_2 \ldots x_n)$ and $y = T(y_1, y_2 \ldots y_n)$, then $x < y \equiv (x_k < y_k) \wedge (x_i = y_i)$ for $i = 1 \ldots k-1$ for some k

Note that axioms (1.8) to (1.10) also hold for this case.

6.7. If c_1, c_2, ... c_n are characters, then '$c_1 c_2 \ldots c_n$' is called a <u>string</u> of length n and means T('c_1','c_2',...'c_n') where <u>type</u> T = <u>packed</u> <u>array</u> [1..n] <u>of</u> char.

Record types

$$\underline{\text{type}}\ T = \underline{\text{record}}\ s_1 : T_1 ; \ldots ; s_m : T_m\ \underline{\text{end}}$$

Let x_i be an element of T_i for $i = 1 \ldots m$.

7.1. $T(x_1, x_2 \ldots x_m)$ is an element of T .

7.2. These are the only elements of T .

7.3. $T(x_1 \ldots x_m).s_i = x_i$ for $i = 1 \ldots m$

$$\begin{array}{l} \underline{\text{type}}\ T = \underline{\text{record}}\ s_1 : T_1 ; \ldots ; s_{m-1} : T_{m-1} ; \\ \qquad\qquad \underline{\text{case}}\ s_m : T_m\ \underline{\text{of}} \\ \qquad\qquad\quad k_1 : (s'_1 : T'_1); \\ \qquad\qquad\quad k_2 : (s'_2 : T'_2); \\ \qquad\qquad\qquad \ldots\ldots. \\ \qquad\qquad\quad k_n : (s'_n : T'_n) \\ \qquad \underline{\text{end}} \end{array}$$

Let k_j be an element of T_m and let x'_j be an element of T'_j for $j = 1 \ldots n$. Then axiom 7.1 is rewritten as

7.1a $T(x_1 \ldots x_{m-1}, k_j, x'_j)$ is an element of T .

Axioms 7.2 and 7.3 apply to this record type unchanged, and in addition the following axiom is given:

7.4. $T(x_1 \ldots x_{m-1}, k_j, x'_j).s'_j = x'_j$ for $j = 1 \ldots n$

We introduce the following abbrevation for later use:

$$\begin{array}{lll} & (x,\ s_i : y) & \text{stands for} \\ & \quad T(x.s_1 \ldots x.s_{i-1},\ y,\ x.s_{i+1} \ldots x.s_m) & \\ \text{and} & (x,\ s'_j : y) & \text{stands for} \\ & \quad T(x.s_1 \ldots x.s_m,\ y) & \end{array}$$

The case with a field list containing several fields

$$k_j : (s_{j1} : T_{j1} ; \ldots s_{jh} : T_{jh})$$

is to be interpreted as

$$k_j : (s'_j : T'_j)$$

where s'_j is a fresh identifer, and T'_j is a type defined as

$$\underline{\text{type}}\ T'_j = \underline{\text{record}}\ s_{j1}: T_{j1};\ \ldots;\ s_{jh}: T_{jh}\ \underline{\text{end}}$$

and where $x.s_{jt}$ is interpreted as $x.s'_j.s_{jt}$.

Set types

$$\underline{\text{type}}\ T = \underline{\text{set}}\ \underline{\text{of}}\ T_o$$

Let x_o, y_o be elements of T_o .

8.1. $[\ ]$ is a T .

8.2. If x is an element of T , then $x \vee [x_o]$ is a T .

8.3. These are the only elements of T .

$[\ ]$ denotes the empty set and $[x_o]$ the singleton set containing x_o . The following axioms define the operations of set membership, union, intersection, and difference.

8.4. $\neg(x_o\ \underline{in}\ [\ ])$

8.5. $x_o\ \underline{in}\ (x \vee [x_o])$

8.6. $x_o \neq y_o \supset (x_o\ \underline{in}\ (x \vee [y_o]) \equiv x_o\ \underline{in}\ x)$

8.7. $x=y \equiv [(x_o\ \underline{in}\ x) = (x_o\ \underline{in}\ y),\ \text{for all } x_o \text{ in } T_o]$.

8.8. $x_o\ \underline{in}\ (x \vee y) \equiv (x_o\ \underline{in}\ x) \vee (x_o\ \underline{in}\ y)$

8.9. $x_o\ \underline{in}\ (x \wedge y) \equiv (x_o\ \underline{in}\ x) \wedge (x_o\ \underline{in}\ y)$

8.10. $x_o\ \underline{in}\ (x-y) \equiv (x_o\ \underline{in}\ x) \wedge \neg(x_o\ \underline{in}\ y)$

8.11. $[x_1, x_2 \ldots x_n]$ means $(([\ ] \vee [x_1]) \vee [x_2]) \vee \ldots \vee [x_n]$

Note that PASCAL restricts set types to be built only on scalar base types T_o with a maximum number of elements defined by each particular implementation.

File types

$$\underline{\text{type}}\ T = \underline{\text{file}}\ \underline{\text{of}}\ T_o$$

Let x_o be an element of T_o .

9.1. $\langle\ \rangle$ is an element of T .

9.2. If x is an element of T , then $x\ \&\ \langle x_o\rangle$ is an element of T .

9.3. These are the only elements of T .

9.4. $(x\&y)\&z = x\&(y\&z)$

9.5. $x\&\langle x_o\rangle \neq \langle\ \rangle$

$\langle\ \rangle$ denotes the empty file (sequence), and $\langle x_o\rangle$ the singleton sequence containing x_o . The operator & denotes concatenation such that $x\&y = \langle x_1 \ldots x_m, y_1, \ldots y_n\rangle$, if $x = \langle x_1 \ldots x_m\rangle$ and $y = \langle y_1 \ldots y_n\rangle$. Neither the explicit denotation of sequences nor the concatenation operator are available in PASCAL.

9.6. $\mathrm{first}(\langle x_o\rangle\&x) = x_o$, $\mathrm{rest}(\langle x_o\rangle\&x) = x$

The functions first and rest are not explicitly available in PASCAL. They will later be used to define the effect of file handling procedures.

Pointer types

$$\underline{\text{type}}\ T = \uparrow T_o$$

A pointer type consists of an arbitrary, unbounded set of values

$$\underline{\text{nil}}, \varphi_1, \varphi_2, \varphi_3 \cdots$$

over which no operation except test of equality is defined. Associated with a pointer type T are a variable ξ of type integer (and initial value 0) and a variable τ with components τ_{φ_1} , τ_{φ_2}, ... which are all of type T_o . These components are the variables to which elements of T (other than <u>nil</u>) are "pointing". ξ is used in connection with the "generation" of new elements of T (see 3.7). ξ and τ are not available to the PASCAL programmer.

$$x \neq \underline{nil} \supset x\uparrow = \tau_x$$

DECLARATIONS

The purpose of a declaration is to introduce a named object (constant, type, variable, function, or procedure) and to prescribe its properties. These properties may then be assumed in any proof relating to the scope of the declaration.

Constant-, type-, and variable declarations

If D is a sequence of declarations and S is a compound statement, then

D;S

is called a <u>block</u>, and the following is its rule of inference (expressed in the usual notation for subsidiary deductions):

11.1. $$\frac{H \vdash P\{S\}Q}{P\{D;S\}Q}$$

H is the set of assertions describing the properties established by the declarations in D . P and Q may not contain any identifiers declared in D ; if they do, the rule can be applied only after a systematic substitution of fresh identifiers local to the block. In the case of constant declarations the assertions in H are nothing but the list of equations themselves. In the case of type definitions they are the axioms derived from the declaration in the manner described above. In the case of a variable declaration x:T it is the fact that x is an element of T .

Consider the file variable declaration

$\underline{\text{var}}\ x\text{: } T$

where

$\underline{\text{type}}\ T = \underline{\text{file}}\ \underline{\text{of}}\ T_o$

This declaration of x assigns the initial value $\langle\ \rangle$ to x, and in addition introduces variables x_L, x_R, and $x\uparrow$ such that

11.2. $x\uparrow$ is an element of T_o, x_L and x_R are elements of T, and $x = x_L \mathbin{\&} x_R$.

x_L and x_R are not accessible in PASCAL, but serve to denote the parts of the sequential file to the left and right of the read/write head. However, the variable $x\uparrow$ is explicitly available and is called the <u>buffer variable</u> of x . Assignments to $x\uparrow$ are permitted only if $x_R = \langle\ \rangle$. This condition is denoted in PASCAL by the Boolean function eof :

11.3. $\text{eof}(x) \equiv x_R = \langle\ \rangle$

In addition, the following axiom holds:

11.4. $x_R \neq \langle\ \rangle \supset x\uparrow = \text{first}(x_R)$

11.5. The standard objects text, input, and output are defined as follows:

$\underline{\text{type}}\ \text{text} = \underline{\text{file}}\ \underline{\text{of}}\ \text{char}$
$\underline{\text{var}}\ \text{input, output: text}$

<u>Function and procedure declarations</u>

$\underline{\text{function}}\ f(L)\text{:}T;\ S$

Let $\underline{x}$ be the list of parameters declared in L , and let $\underline{y}$ be the set of global variables occuring within S (implicit parameters). Given the assertion $P\{S\}Q$, we may deduce the following implication:

12.1. $P \supset Q^{f}_{f(\underline{x},\underline{y})}$ for all values of $\underline{x}$, $\underline{y}$

Note that the explicit parameter list $\underline{x}$ has been extended by the implicit parameters $\underline{y}$, that $\underline{x}$ may not contain any variable parameters (specified by var), and that no assignments to nonlocal variables may occur within S . It is this property (12.1) that may be assumed in proving assertions about expressions containing calls of the function f , including those occuring within S itself and in other declarations in the same block. In addition, assertions generated by the parameter specifications in L may be used in proving assertions about S .

procedure p(L); S

Let $\underline{x}$ be the list of explicit parameters delcared in L ; let $\underline{y}$ be the set of global variables occuring in S (implicit parameters), let $x_1 \dots x_m$ be the paremeters declared in L as variable parameters, and let $y_1 \dots y_n$ be those global variables which are changed within S . Given the assertion $P\{S\}Q$, we may deduce the existence of functions f_i and g_j satisfying the following implication:

12.2.

$$P \supset Q^{x_1 \ \dots \ x_m, \ y_1 \ \dots \ y_n}_{f_1(\underline{x},\underline{y}) \ \dots \ f_m(\underline{x},\underline{y}), \ g_1(\underline{x},\underline{y}) \ \dots \ g_n(\underline{x},\underline{y})}$$

for all values of the variables involved in this statement.

It is this property that may be assumed in proving assertions about calls of this procedure, including those occuring within S itself and in other declarations in the same block.

The functions f_i and g_j may be regarded as those which map the initial values of $\underline{x}$ and $\underline{y}$ on entry to the procedure onto the final values of $x_1 \dots x_m$ and $y_1 \dots y_n$ on completion of the execution of S .

STATEMENTS

Statements are classified into simple statements and structured statements. The meaning of simple statements is defined by axioms, and the meaning of structured statements is defined in terms of rules of inference permitting deduction of the properties of the structured statement from properties of its constituents. However, the rules of inference are formulated in such a way that the reverse process of deriving necessary properties of the constituents from postulated properties of the composite statement is facilitated. The reason for this orientation is that in deducing proofs of properties of programs it is most convenient to proceed in a "top-down" direction.

Simple statements

Assignment statements:

13.1. $\qquad P^x_y \; \{x := y\} \; P$

In the case where the type T of x is a subrange of the type of y, P^x_y is to be replaced by $P^x_{T(y)}$, and if the type T of y is a subrange of the type of x, then P^x_y is to be replaced by $P^x_{T^{-1}(y)}$ in 13.1.

In the case where x is an indexed variable, we introduce the convention that

$$P^{a[i]}_y \qquad \text{means} \qquad P^a_{(a,i:y)}$$

and if x is a field designator, we introduce the convention that

$$P^{r.s}_y \qquad \text{means} \qquad P^r_{(r,s:y)}$$

Procedure statements:

13.2.

$$P^{x_1 \ \dots \ x_m, \ y_1 \ \dots \ y_n}_{f_1(\underline{x},\underline{y}) \ \dots \ f_m(\underline{x},\underline{y}), \ g_1(\underline{x},\underline{y}) \ \dots \ g_n(\underline{x},\underline{y})} \ \{ p(\underline{x}) \} \ P$$

$\underline{x}$ is the list of actual parameters; $x_1 \dots x_m$ are those elements of $\underline{x}$ which correspond to formal parameters specified as variable parameters, $\underline{y}$ is the set of all variables accessed nonlocally by the procedure p, and $y_1 \dots y_n$ are those elements of y which are subject to assignments by the procedure.

$f_1 \dots f_m$ and $g_1 \dots g_n$ are functions yielding the values assigned by the execution of p to the variables $x_1 \dots x_m$ and $y_1 \dots y_n$, which must all be distinct. (Otherwise the effect of the procedure statement is undefined.) Rule 13.2 states that the procedure statement $p(\underline{x})$ is equivalent with the sequence of assignments (executed "concurrently")

$$x_1 := f_1(\underline{x},\underline{y}); \quad \dots \quad x_m := f_m(\underline{x},\underline{y});$$
$$y_1 := g_1(\underline{x},\underline{y}); \quad \dots \quad y_n := g_n(\underline{x},\underline{y})$$

The following inference rules specify the properties of the standard procedures <u>put</u>, <u>get</u>, <u>reset</u>, and <u>rewrite</u>. The assertion P in 13.3-13.6 must contain x, x_L, x_R, $x\uparrow$ only if they occur explicitly in the list of substituends.

13.3.

$$eof(x) \wedge P^{x}_{x \,\&\, \langle x\uparrow \rangle} \ \{ put(x) \} \ eof(x) \wedge P$$

The variables x, x_L, and x_R must not occur free in P. The procedure $put(x)$ is only applicable, if $eof(x)$ is true, i.e. $x_R = \langle \ \rangle$. It thus leaves $eof(x)$ and $x_L = x$ invariant, leaves $x\uparrow$ undefined, and corresponds to the assignment

$$x := x \ \& \ \langle x\uparrow \rangle$$

13.4.

$$\neg eof(x) \wedge P^{x_L,\ x\uparrow,\ x_R}_{x_L \& \langle x\uparrow\rangle,\ first(rest(x_R)),\ rest(x_R)} \quad \{ get(x) \} \quad P$$

The operation get(x) is only applicable, if $\neg eof(x)$, i.e. $x_R \neq \langle\ \rangle$, and then corresponds to the three assignments performed "concurrently"

$$x_L := x_L \& \langle x\uparrow\rangle; \quad x\uparrow := first(rest(x_R)); \quad x_R := rest(x_R)$$

13.5.

$$P^{x_L\ \ x\uparrow\ \ x_R}_{\langle\rangle,\ first(x),\ x} \quad \{ reset(x) \} \quad P$$

The operation reset(x) corresponds to the three assignments

$$x_L := \langle\ \rangle; \quad x\uparrow := first(x); \quad x_R := x$$

13.6. $\quad P^{x}_{\langle\ \rangle} \quad \{ rewrite(x) \} \quad P$

The procedure statement rewrite(x) corresponds to the assignment

$$x := \langle\ \rangle$$

The following rule specifies the effect of the standard procedure <u>new</u>.

13.7. If t is a pointer variable of type T , then

$$new(t) \quad \text{means} \quad \xi := succ(\xi); \quad t := \varphi_\xi$$

where ξ is the hidden variable associated with the pointer type T

The following rules define the meaning of the standard procedures <u>pack</u> and <u>unpack</u>. Consider the type definitions

<u>type</u> A = <u>array</u> [m..n] <u>of</u> T

and

<u>type</u> B = <u>packed</u> <u>array</u> [u..v] <u>of</u> T

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

13.8. If a is an element of A and b is an element of B, then

pack(a,i,b) means
<u>for</u> j := u <u>to</u> v <u>do</u> b[j] := a[j-u+i]

13.9. unpack(b,a,i) means
<u>for</u> j := u <u>to</u> v <u>do</u> a[j-u+i] := b[j]

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

The following rules specify the meaning of the standard procedures <u>read</u> and <u>write</u>. Let v be a variable and e an expression of type char, then the statement

13.10. read(v)

is equivalent with the statements

v := input↑; get(input)

13.11. write(e)

is equivalent with the statements

output↑ := e; put(output)

<u>Structured statements</u>

<u>Compound statements:</u>

14.1. $$\frac{P_{i-1}\,\{S_i\}\,P_i, \quad \text{for } i = 1 \ldots n}{P_0\{\underline{\text{begin}}\ S_1;S_2 \ldots S_n\ \underline{\text{end}}\}P_n}$$

<u>If statements:</u>

14.2. $$\frac{Q_1\{S_1\}R,\ Q_2\{S_2\}R,\ P \wedge B \supset Q_1,\ P \wedge \neg B \supset Q_2}{P\{\underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\}R}$$

14.3. $$\frac{Q\{S\}R,\ P \wedge B \supset Q,\ P \wedge \neg B \supset R}{P\{\underline{\text{if}}\ B\ \underline{\text{then}}\ S\}\ R}$$

Case statements:

14.4. $$\frac{Q_i\{S_i\}R,\ P\wedge(x=k_i)\supset Q_i,\quad \text{for } i = 1..n}{P\{\underline{\text{case}}\ x\ \underline{\text{of}}\ k_1:S_1;k_2S_2;\ldots k_n:S_n\ \underline{\text{end}}\}R}$$

Note: $k_a, k_b \ldots k_n:S$ stands for $k_a:S;\ k_b:S;\ \ldots\ k_n:S$

While statements:

14.5. $$\frac{Q\wedge B\ \{S\}\ Q}{Q\ \{\underline{\text{while}}\ B\ \underline{\text{do}}\ S\}\ Q\wedge\neg B}$$

Repeat statements:

14.6. $$\frac{P\ \{S\}\ Q,\ Q\wedge\neg B\supset P}{P\ \{\underline{\text{repeat}}\ S\ \underline{\text{until}}\ B\}\ Q\wedge B}$$

Note that PASCAL allows a sequence of statements to occur between the brackets repeat and until. Thus S stands here for a sequence of statements.

For statements:

14.7. $$\frac{(a\leq x\leq b)\wedge P([a..x))\ \{S\}\ P([a..x])}{P([\,])\ \{\underline{\text{for}}\ x := a\ \underline{\text{to}}\ b\ \underline{\text{do}}\ S\}\ P([a..b])}$$

The notation $[u..v]$ is used to denote the closed interval $u...v$, i.e. the set $\{i\,|\,u\leq i\leq v\}$, and $[u..v)$ is used to denote the open interval $u...v$, i.e. the set $\{i\,|\,u\leq i<v\}$. Similarly $(u..v]$ denotes the set $\{i\,|\,u<i\leq v\}$. Note that $[u..u) = (u..u] = [\ ]$ is the empty set.

14.8. $$\frac{(a\leq x\leq b)\wedge P((x..b])\ \{S\}\ P([x..b])}{P([\,])\ \{\underline{\text{for}}\ x := b\ \underline{\text{downto}}\ a\ \underline{\text{do}}\ S\}\ P([a..b])}$$

With statements:

$$14.9. \quad \frac{P^{r.s_1 \ldots r.s_m}_{s_1 \ldots s_m} \{S\} \; Q^{r.s_1 \ldots r.s_m}_{s_1 \ldots s_m}}{P \; \{\underline{\text{with}} \; r \; \underline{\text{do}} \; S\} \; Q}$$

$s_1 \ldots s_m$ are the field identifiers of the record type of r.

Note that r must not contain any variables subject to change by S , and that

$\underline{\text{with}}\ r_1,\ r_2 \ldots r_n\ \underline{\text{do}}\ S$

stands for

$\underline{\text{with}}\ r_1\ \underline{\text{do}}\ \underline{\text{with}}\ r_2\ \underline{\text{do}} \ldots \underline{\text{with}}\ r_n\ \underline{\text{do}}\ S$

STANDARDS 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 it 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. Access to components of packed arrays by indexing is not permitted. (Consequently, there is no need to implement the

complexities of division and taking the remainder involved in extracting or selectively updating an element of a packed array.)

4. A component of a packed structure - in particular of a packed record - may 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. No component of any structuredtype may be of file type. (This avoids a significant complexity of implementation.)

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

12. Implementations based on the (restricted) ASCII or EBCDIC character sets should obey the following transliteration rules concerning the PASCAL symbols not included in the respective character sets:

<table>
<tr><th colspan="3">PASCAL symbol</th><th colspan="3">ASCII characters</th><th colspan="3">EBCDIC characters</th></tr>
<tr><td>∨</td><td>∧</td><td>¬</td><td>OR</td><td>&</td><td>NOT</td><td>|</td><td>&</td><td>¬</td></tr>
<tr><td>≠</td><td>≤</td><td>≥</td><td>#</td><td><=</td><td>>=</td><td>¬=</td><td><=</td><td>>=</td></tr>
<tr><td>{</td><td>}</td><td>↑</td><td>/*</td><td>*/</td><td>^</td><td>/*</td><td>*/</td><td>@</td></tr>
<tr><td>[</td><td>]</td><td></td><td>[</td><td>]</td><td></td><td>(.</td><td>.)</td><td></td></tr>
</table>

These rules are designed such that a simple program may perform a transliteration without consideration of context.

13. The following are standard identifiers defined in every implementation of PASCAL:

Constants:

false, true	(2.1-6)
eol	
alfaleng	

Types:

Boolean	(2.1-6)
integer	(3.1-18)
char	(4.1-4)
real	
text	(11.5)

Variables:

input, output	(11.5)

Functions of real arithmetic:

sqrt, exp, ln,
sin, cos, arctan,
trunc

Functions:

abs	(3.13-14)
sqr	(3.15)
odd	(3.16)
succ, pred	(1.3-4)
ord, chr	(4.3)
eof	(11.3)

Procedures:

put	(13.3)
get	(13.4)
reset	(13.5)
rewrite	(13.6)
new	(13.7)
pack	(13.8)
unpack	(13.9)
read	(13.10)
write	(13.11)

Acknowledgement:

The authors are indebted to Erwin Engeler for his valuable suggestions and comments.

References

1. N. Wirth, "The Programming Language PASCAL", Acta Informatica 1, 35-63 (1971)

2. P. Naur, Ed., "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM 6, 1-17 (1963), Comp. J. 5, 349-367 (1962/63), and Num. Math. 4, 420-453 (1963)

3. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", Comm. ACM 12, 576-581 (Oct. 1969)

4. —— "An axiomatic definition of the programming language PASCAL", Second Draft, Proc. Symposium on Theoretical Programming, Novosibirsk, Aug. 1972

5. —— "Notes on Data Structuring" in Structured Programming by Dahl, Dijkstra, and Hoare, Acad. Press 1972

6. —— "A Note on the For Statement", BIT 12, 334-341 (1972)

7. A. van Wijngaarden, "A Numerical Analysis as an Independent Science", BIT 6, 66-81 (1966)

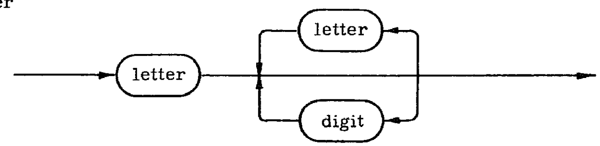
8. H. Rutishauser; Unpublished lecture notes, ETH Zürich

9. N. Wirth, "The Design of a PASCAL Compiler", Software, Practice and Experience 1, 309-333 (1971)

10. J. Welsh and C. Quinn, "A PASCAL Compiler for the ICL 1900 Series Computers", Software, Practice and Experience 2, 73-77 (1972).

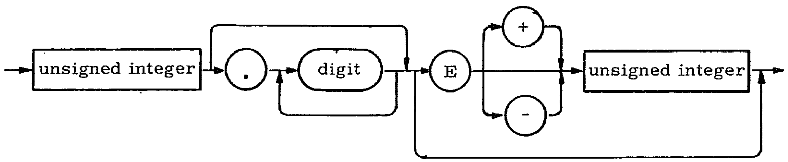
APPENDIX

identifier

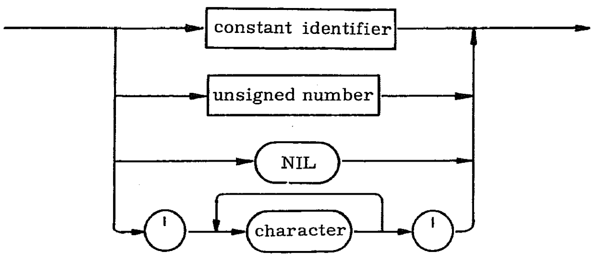


unsigned integer

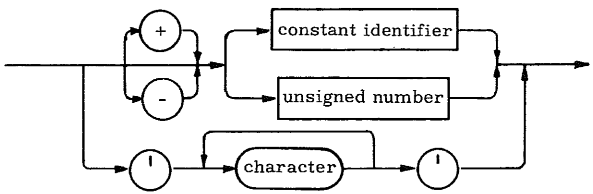
unsigned number



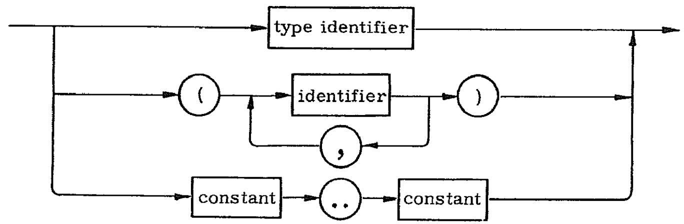
unsigned constant



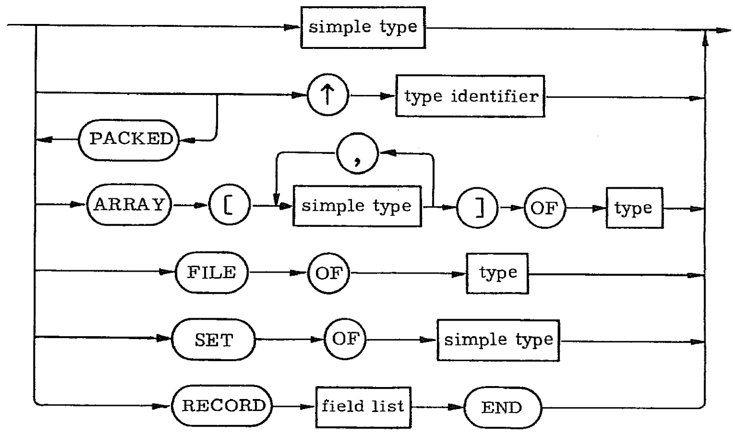
constant



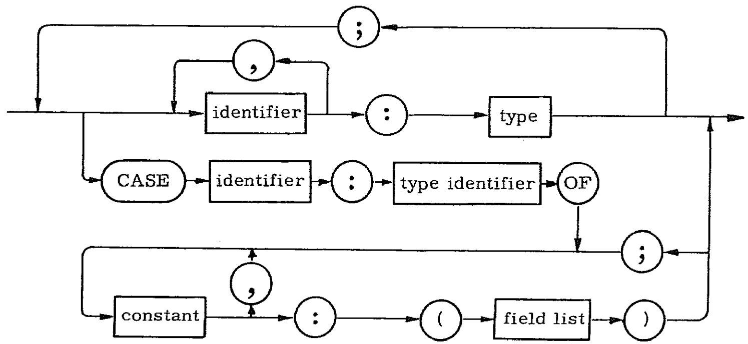
simple type



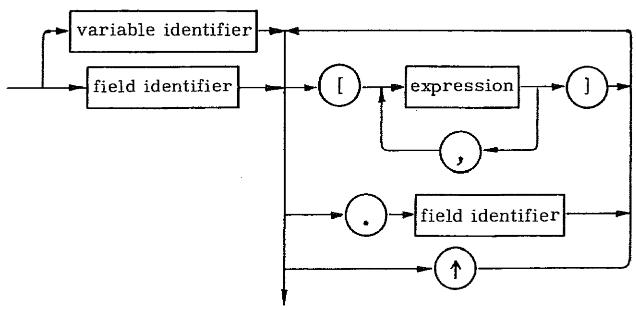
type



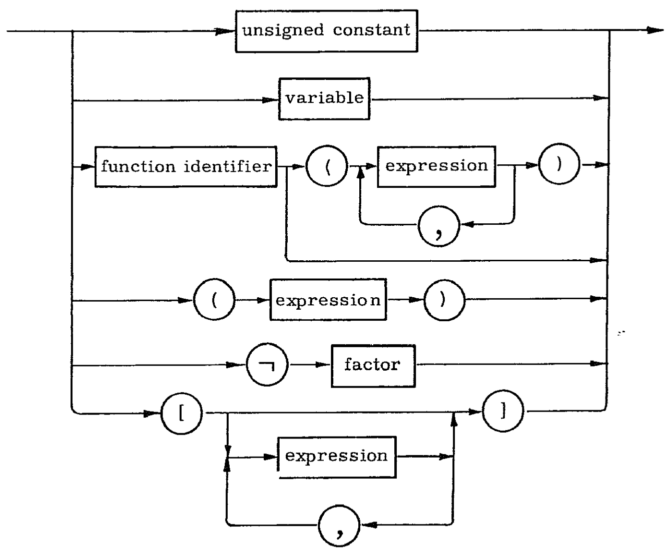
field list



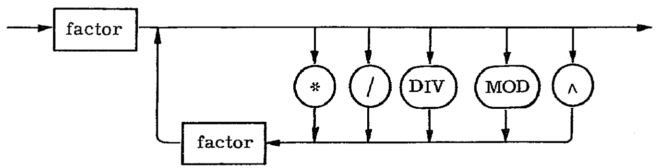
variable



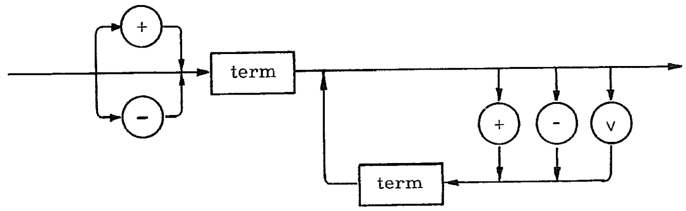
factor



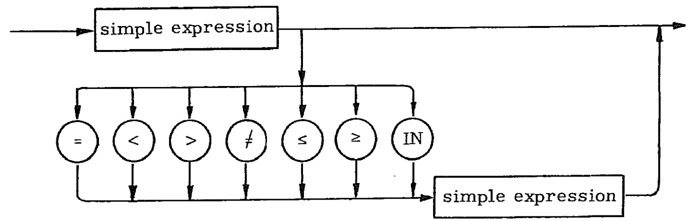
term



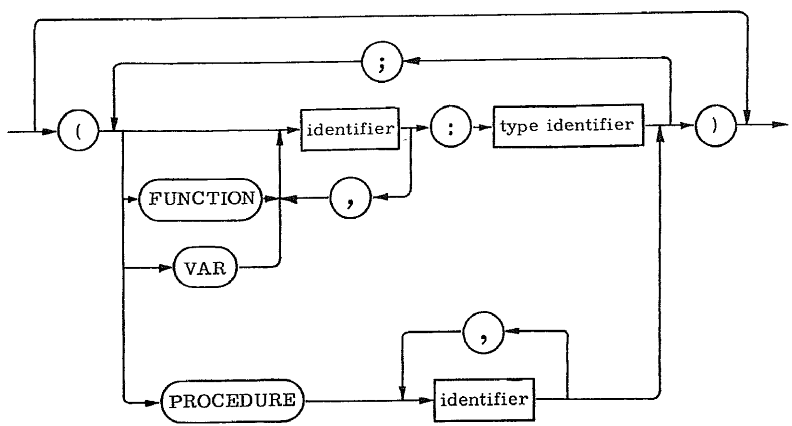
simple expression



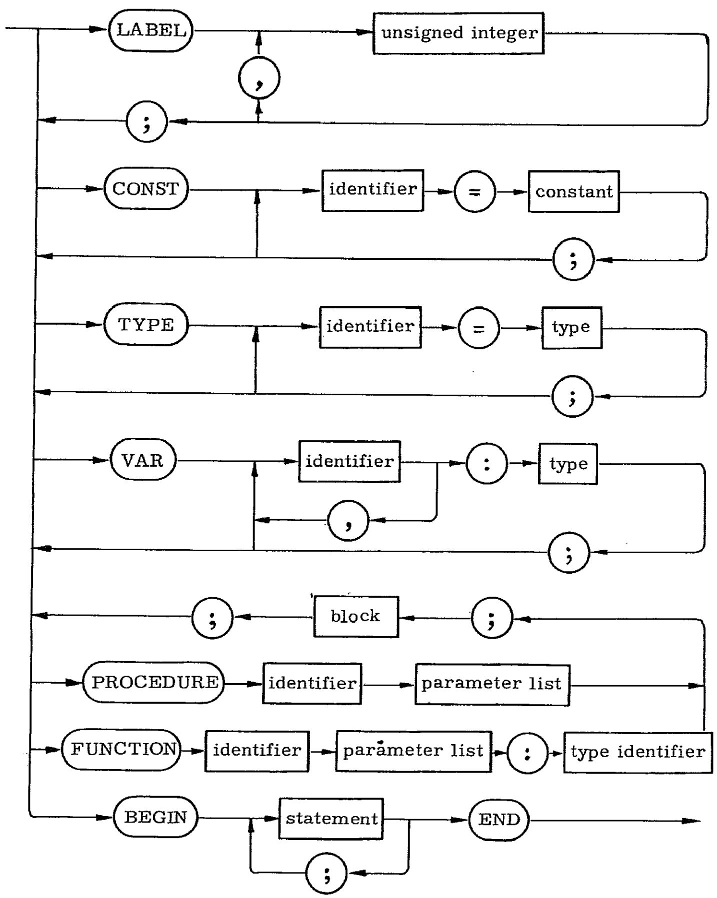
expression



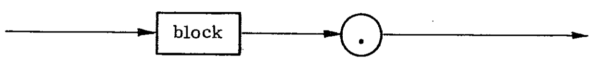
parameter list



block



program



statement