Acta Informatica 2, 335—355 (1973)

An Axiomatic Definition of the Programming Language PASCAL

C. A. R. Hoare and N. Wirth

Received December 11, 1972

Summary. The axiomatic definition method proposed in reference [5] is extended and applied to define the meaning of the programming language PASCAL [1]. The whole language is covered with the exception of real arithmetic and go to statements.

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 [3, 4]. 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 [5–7] 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 [5] operates on four levels of discourse:

1. *PASCAL statements*, 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 structures which correspond to the various data types.

3b) Assertions of the form $P\{S\}Q$ which express that, if P is true before the execution of S, then Q is true after the execution of S. This kind of assertion is used to define the meaning of assignment and procedure statements. It is vacuously true, if the execution of S does not terminate.

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 about the components of the structured statements.

In addition, the notation

$$P_y^x$$

is used for the formula which is obtained by systematically substituting y for all free occurrences of x in P. If this introduces conflict between free variables of y and bound variables of P, the conflict is resolved by systematic change of the latter variables.

$$P_{y_1 \dots y_n}^{x_1 \dots x_n}$$

denotes simultaneous substitution for all occurrences of any x_i by the corresponding y_i. Thus occurrences of x_i within any y_i are *not* replaced. The variables $x_1 \dots x_n$ must be distinct; otherwise the simultaneous substitution is not defined.

In proofs of PASCAL programs, it will be necessary to make use of the following two inference rules:

$$\frac{P\{S\}Q, \quad Q \supset R}{P\{S\}R}$$

$$\frac{P \supset Q, \quad Q\{S\}R}{P\{S\}R}$$

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 arithmetic and *go to* statements (jumps). Also the type *alfa* is not treated. It may be defined as

type *alfa* = **array** [1 .. *w*] **of** *char*

where *w* is an implementation-defined integer.

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 referring to [1]. For easy reference, the revised syntax is summarised in the form of diagrams in Appendix 1.

Many of the axioms have been explained in previous papers, e.g. [5–8]. However, the axioms for procedures are novel; an informal explanation and example are given in Appendix 2.

The authors are not wholly satisfied with the axioms presented for classes, and for procedures and functions, particularly with those referring to global variables. This may be due either to inadequacy of the axiomatisation or to genuine logical complexity of these features of the language. The paper is offered as a first attempt at an almost complete axiomatisation of a realistic programming language, rather than as a definitive specification of a language especially constructed to demonstrate the definition method.

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 syntactic 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.

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 $P = \uparrow c$;
var c: **class** n **of** T

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

type $P = \uparrow T$.

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 [8].

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 **powerset** is replaced by the symbols **set of**, 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*.

Declarations of labels are compulsory.

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

type $T = (c_1, c_2 \ldots c_n)$

1.1. $c_1, c_2 \ldots c_n$ are distinct elements of T.

1.2. These are the only elements of T.

1.3. $c_{i+1} = succ\,(c_i)$ for $i = 1 \ldots n-1$

1.4. $c_i = pred\,(c_{i+1})$ for $i = 1 \ldots n-1$

1.5. $\neg\,(x < x)$

1.6. $(x < y) \wedge (y < z) \supset (x < z)$

1.7. $(x \neq c_n) \supset (x < succ\,(x))$

1.8. $x > y \equiv y < x$

1.9. $x \leqq y \equiv \neg\,(x > y)$

1.10. $x \geqq y \equiv \neg\,(x < y)$

1.11. $x \neq y \equiv \neg\,(x = y)$

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

The standard scalar type *Boolean* is defined as

$$\textbf{type } Boolean = (false, true).$$

The standard type *integer* stands for a finite, coherent set of the whole numbers. The logical operators $\vee$, $\wedge$, $\neg$, and the arithmetic operators $+$, $-$, $*$, and **div**, are those of the conventional logical calculus and of whole number arithmetic. The modulus operator **mod** is defined by the equation

$$m \textbf{ mod } n = m - (m \textbf{ div } n) * n$$

whereas **div** denotes division with truncated fraction.

Implementations are permitted to refuse the execution of programs which refer to integers outside the range specified by their definition of the type *integer*.

The Type Char

2.1. The elements of the type *char* are the 26 (capital) letters, the 10 (decimal) digits, and possibly other characters defined by particular implementations. In programs, a constant of type *char* is denoted by enclosing the character in quote marks.

2.2.
$$\begin{array}{ll} \text{'A'} < \text{'B'} & \text{'1'} = succ(\text{'0'}) \\ \text{'B'} < \text{'C'} & \text{'2'} = succ(\text{'1'}) \\ \dots & \dots \\ \text{'Y'} < \text{'Z'} & \text{'9'} = succ(\text{'8'}). \end{array}$$

The sets of letters and digits are ordered, and the digits are coherent.

Axioms (1.5)–(1.11) apply to the *char* type. The functions *ord* and *chr* are defined by the following additional axioms:

2.3. if u is an element of *char*, then $ord(u)$ is a non-negative integer (called the *ordinal number* of u), and

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

2.4.
$$u < v \equiv ord(u) < ord(v).$$

These axioms have been designed to make possible an 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.

Subrange Types

$$\textbf{type } T = m \,..\, n$$

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

$$m \leqq a \leqq n$$

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

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

3.1. $T(a)$ is an element of T.

3.2. These are the only elements of T.

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

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

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

3.5. If $\bigcirc$ is a dyadic operator defined on $T_0 \times T_0$, then

$$\begin{array}{lll} x \bigcirc y & \text{means} & T^{-1}(x) \bigcirc T^{-1}(y) \\ x \bigcirc a & \text{means} & T^{-1}(x) \bigcirc a \\ a \bigcirc x & \text{means} & a \bigcirc T^{-1}(x). \end{array}$$

Array Types

type $T =$ **array** $[I]$ **of** T_0

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

4.1. If x_i is an element of T_0 for all i such that $m \leqq i \leqq n$, then $T(x_m \ldots x_n)$ is an element of T.

4.2. These are the only elements of T.

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

4.4. **array** $[I_1, I_2 \ldots I_k]$ **of** T_0 means **array** $[I_1]$ **of array** $[I_2 \ldots I_k]$ **of** T_0.

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

We introduce the following abbrevation for later use (see 11.1):
$(x, i{:}y)$ stands for
$T(x[m] \ldots x[pred(i)], y, x[succ(i)] \ldots x[n])$.

Record Types

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

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

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

5.2. These are the only elements of T.

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

Variant Records

```
type T = record s₁: T₁; ...; sₘ₋₁: Tₘ₋₁;
              case sₘ: Tₘ of
                   k₁: (s₁′: T₁′);
                   k₂: (s₂′: T₂′);
                   ...
                   kₙ: (sₙ′: Tₙ′)
         end
```

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 5.1 is rewritten as

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

Axioms 5.2 and 5.3 apply to this record type unchanged, and in addition the following axiom is given:

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

We introduce the following abbrevation for later use (see 11.1):

$$(x, s_i : y) \text{ stands for } T(x.s_1 \ldots x.s_{i-1}, y, x.s_{i+1} \ldots x.s_m)$$

and

$$(x, s'_j : y) \text{ stands for } T(x.s_1 \ldots x.s_m, y)$$

The case with a field list containing several fields

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

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

$$\textbf{type } T'_j = \textbf{record } s_{j1} : T_{j1}; \ldots; s_{jk} : T_{jk} \textbf{ end}$$

In this case $x \cdot s_{jt}$ is interpreted as $x \cdot s'_j \cdot s_{jt}$.

Set Types

type $T =$ **set of** T_0

Let x_0, y_0 be elements of T_0.

6.1. $[\,]$ is an element of T.

6.2. If x is an element of T, then $x \vee [x_0]$ is a T.

6.3. These are the only elements of T.

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

$[\,]$ denotes the empty set, and $[x_0]$ denotes the singleton set containing x_0. The operators $\vee$, $\wedge$, and $-$, applied to elements of set type, denote the conventional operations of set union, intersection, and difference.

Note that PASCAL allows implementations to restrict set types to be built only on base types T_0 with a specified maximum number of elements.

File Types

type $T =$ **file of** T_0

Let x_0 be an element of T_0.

7.1. $\langle\,\rangle$ is an element of T.

7.2. If x is an element of T, then $x \& \langle x_0 \rangle$ is an element of T.

7.3. These are the only elements of T.

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

7.5. $x \& \langle x_0 \rangle \neq \langle\,\rangle$.

$\langle\,\rangle$ denotes the empty file (sequence), and $\langle x_0 \rangle$ the singleton sequence containing x_0. 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.

7.6. $\mathit{first}\,(\langle x_0 \rangle \& x) = x_0, \qquad \mathit{rest}\,(\langle x_0 \rangle \& x) = x.$

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

$$\textbf{type}\ T = \uparrow T_0$$

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

$$\textbf{nil},\ \varphi_1, \varphi_2, \varphi_3 \cdots$$

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 $\tau_{\varphi_1}, \tau_{\varphi_2}, \ldots$ which are all of type T_0. These components are the variables to which elements of T (other than **nil**) are "pointing". ξ is used in connection with the "generation" of new elements of T (see 11.7). ξ and τ are not available to the PASCAL programmer.

8.1. $x \neq \textbf{nil} \supset x\uparrow = \tau_x.$

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 *block*, and the following is its rule of inference (expressed in the usual notation for subsidiary deductions):

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

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.

File Variable Declarations

The declaration

$$\textbf{var}\ x{:}T$$

where

$$\textbf{type}\ T = \textbf{file of}\ T_0,$$

introduces the *two* variables x of type T and $x\uparrow$ of type T_0. $x\uparrow$ is called the *buffer variable* of x and is used implicitly by the standard file procedures. A so-called *file position* is associated with x; it splits x into a left part x_L and a right part x_R such that

9.2. x_L and x_R are of type T, and $x \equiv x_L \& x_R$.

x_L and x_R are not explicitly available to the programmer. Assignments to the buffer variable $x\uparrow$ are permitted only if $x_R = \langle\,\rangle$. This condition is denoted in PASCAL by the Boolean function *eof* (end of file):

9.3. $$eof(x) \equiv x_R = \langle\,\rangle.$$

In addition, the following axiom holds:

9.4. $$x_R \neq \langle\,\rangle \supset x\uparrow = first\,(x_R).$$

9.5. The standard type *text*, and the standard variables *input*, and *output* are defined as follows:

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

Function and Procedure Declarations

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

Let **x** be the list of parameters declared in L, and let **y** be the set of global variables occuring within S (implicit parameters). Given the assertion $P\{S\}Q$, where f does not occur free in P, and none of the variables of **x** occurs free in Q, we may deduce the following implication:

10.1. $P \supset Q^f_{f(\mathbf{x},\mathbf{y})}$, for all values of the variables involved in this assertion

Note that the explicit parameter list **x** has been extended by the implicit parameters **y**, that **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 (10.1) 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.

$$\textbf{procedure}\ p(L){:}S.$$

Let **x** be the list of explicit parameters declared in L; let **y** be the set of global variables occuring in S (implicit parameters), let $x_1 \ldots x_m$ be the parameters declared in L as variable parameters, and let $y_1 \ldots y_n$ be those global variables which are changed within S. Given the assertion $P\{S\}Q$ where none of the value parameters of $\mathbf{x}$ occurs free in Q, we may deduce the existence of functions f_i and g_j satisfying the following implication:

10.2. $P \supset Q^{x_1 \;\ldots\; x_m,\; y_1 \;\ldots\; y_n}_{f_1(\mathbf{x},\mathbf{y}) \;\ldots\; f_m(\mathbf{x},\mathbf{y}),\; g_1(\mathbf{x},\mathbf{y}) \;\ldots\; g_n(\mathbf{x},\mathbf{y})}$

for all values of the variables involved in this assertion.

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 $\mathbf{x}$ and $\mathbf{y}$ on entry to the procedure onto the final values of $x_1 \ldots x_m$ and $y_1 \ldots y_n$ 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

11.1. $P^x_y\{x := y\}P$.

We introduce the following conventions

(1) If the type T of x is a subrange of the type of y,

$$P^x_y \text{ means } P^x_{T(y)}.$$

(2) If the type T of y is a subrange of the type of x,

$$P^x_y \text{ means } P^x_{T^{-1}(y)}.$$

(3) If x is an indexed variable

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

(4) If x is a field designator

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

Procedure Statements

11.2. $P^{x_1 \;\ldots\; x_m,\; y_1 \;\ldots\; y_n}_{f_1(\mathbf{x},\mathbf{y}) \;\ldots\; f_m(\mathbf{x},\mathbf{y}),\; g_1(\mathbf{x},\mathbf{y}) \;\ldots\; g_n(\mathbf{x},\mathbf{y})}\ \{p(\mathbf{x})\}\ P$.

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

$f_1 \dots f_m$ and $g_1 \dots g_n$ are the functions introduced and explained in 10.2. Note that $x_1 \dots x_m, y_1 \dots y_n$ must be all distinct (in the sense that none can contain or be a variable which is contained in another); otherwise the effect of the procedure statement is undefined. Rule 11.2 states that the procedure statement $p(\mathbf{x})$ is equivalent with the sequence of assignments (executed "simultaneously")

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

Standard Procedures

The following inference rules specify the properties of the standard procedures *put*, *get*, *reset*, and *rewrite*. The assertion P in 11.3–11.6 must not contain x, x_L, x_R, $x\uparrow$, except in those cases where they occur explicitly in the list of substituends.

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

This axiom specifies that 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, makes $x\uparrow$ undefined, and corresponds to the assignment

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

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

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

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

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

The operation $reset(x)$ corresponds to the three assignments

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

11.6. $P^{x}_{\langle\,\rangle} \{rewrite(x)\}\, P$.

The procedure statement $rewrite(x)$ corresponds to the assignment

$$x := \langle\,\rangle.$$

The following rule specifies the effect of the standard procedure *new*.

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

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

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

Structured Statements

Compound Statements

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

If Statements

12.2.
$$\frac{P\wedge B\{S_1\}Q,\quad P\wedge\neg B\{S_2\}Q}{P\{\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\}Q}$$

12.3.
$$\frac{P\wedge B\{S\}Q,\quad P\wedge\neg B\supset Q}{P\{\textbf{if } B \textbf{ then } S\}Q}$$

Case Statements

12.4.
$$\frac{P\wedge(x=k_i)\{S_i\}Q,\quad\text{for}\quad i=1\ldots n}{(x\in[k_1\ldots k_n])\wedge P\,\{\textbf{case } x \textbf{ of } k_1\!:\!S_1;\ \ldots;\ k_n\!:\!S_n \textbf{ end}\}\,Q}$$

Note: $k_a, k_b, \ldots, k_m\!:\!S$ stands for $k_a\!:\!S$; $k_b\!:\!S$; $\ldots$; $k_m\!:\!S$.

While Statements

12.5.
$$\frac{P\wedge B\{S\}P}{P\{\textbf{while } B \textbf{ do } S\}P\wedge\neg B}$$

Repeat Statements

12.6.
$$\frac{P\{S\}Q,\quad Q\wedge\neg B\supset P}{P\{\textbf{repeat } S \textbf{ until } B\}Q\wedge B}$$

For Statements

12.7.
$$\frac{(a\leqq x\leqq b)\wedge P([a\,..\,x))\{S\}P([a\,..\,x])}{P([\,])\{\textbf{for } x:=a \textbf{ to } b \textbf{ do } S\}P([a\,..\,b])}$$

$[u\,..\,v]$ denotes the closed interval $u\ldots v$, i.e. the set $\{i|u\leqq i\leqq v\}$ and $[u\,..\,v)$ denotes the open interval $u\ldots v$, i.e. the set $\{i|u\leqq i<v\}$. Similarly, $(u\,..\,v]$ denotes the set $\{i|u<i\leqq v\}$. Note that $[u\,..\,u)=(u\,..\,u]$ is the empty set.

$$\frac{(a\leqq x\leqq b)\wedge P((x\,..\,b])\{S\}P([x\,..\,b])}{P([\,])\{\textbf{for } x:=b \textbf{ downto } a \textbf{ do } S\}P([a\,..\,b])}$$

Note that S must not change x, a, or b.

With Statements

12.9.
$$\frac{P^{r\cdot s_1\ldots r\cdot s_m}_{s_1\ldots s_m}\{S\}Q^{r\cdot s_1\ldots r\cdot s_m}_{s_1\ldots s_m}}{P\{\textbf{with } r \textbf{ do } S\}Q}$$

$s_1\ldots s_m$ are the field identifiers of the record variable r. Note that r must not contain any variables subject to change by S, and that

$$\textbf{with } r_1, \ldots, r_n \textbf{ do } S$$

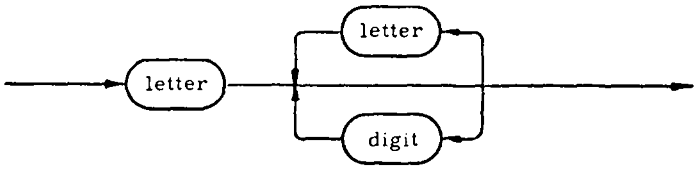
stands for

$$\textbf{with } r_1 \textbf{ do} \ldots \textbf{with } r_n \textbf{ do } S.$$

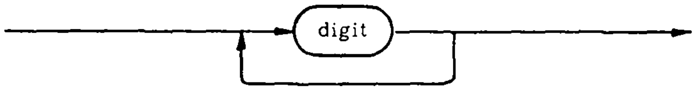
Appendix 1

Syntax Diagrams

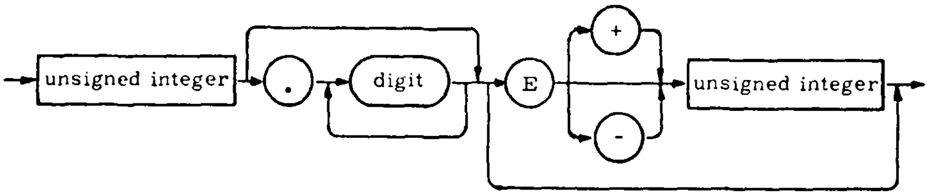
identifier



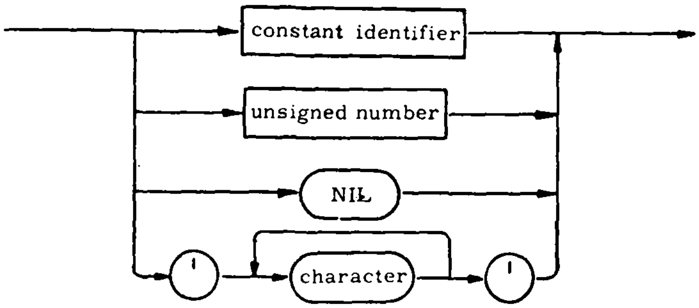
unsigned integer



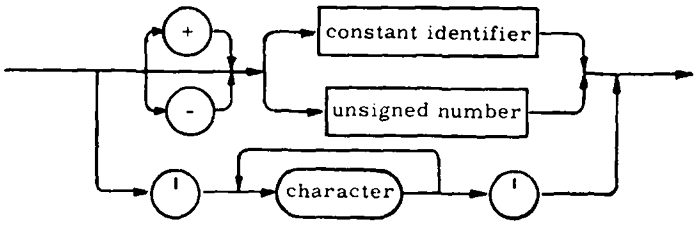
unsigned number



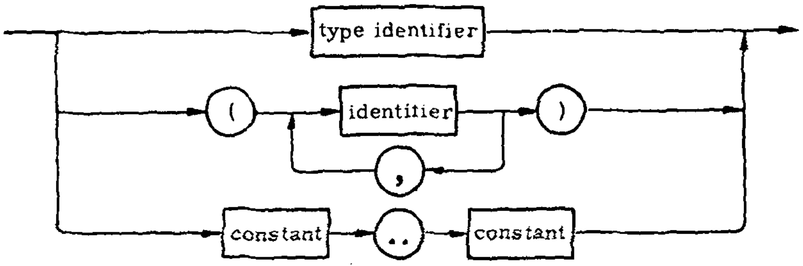
unsigned constant



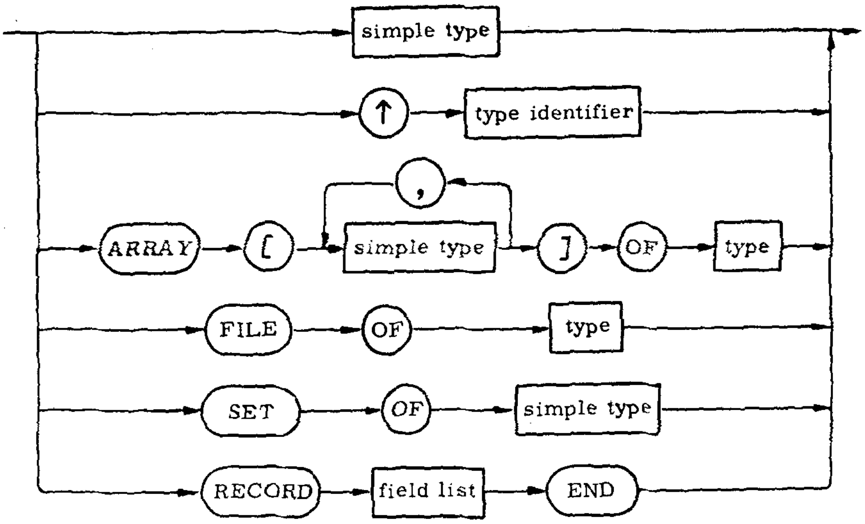
constant



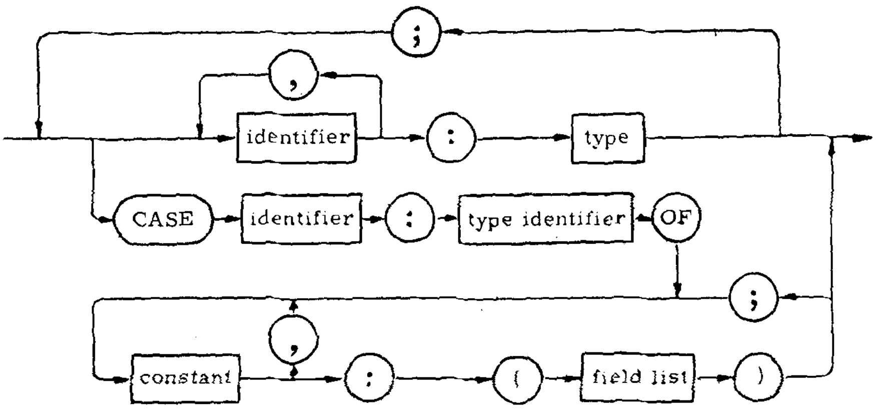
simple type



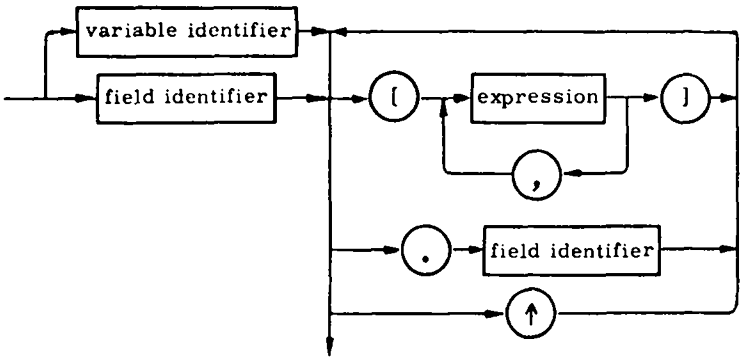
type



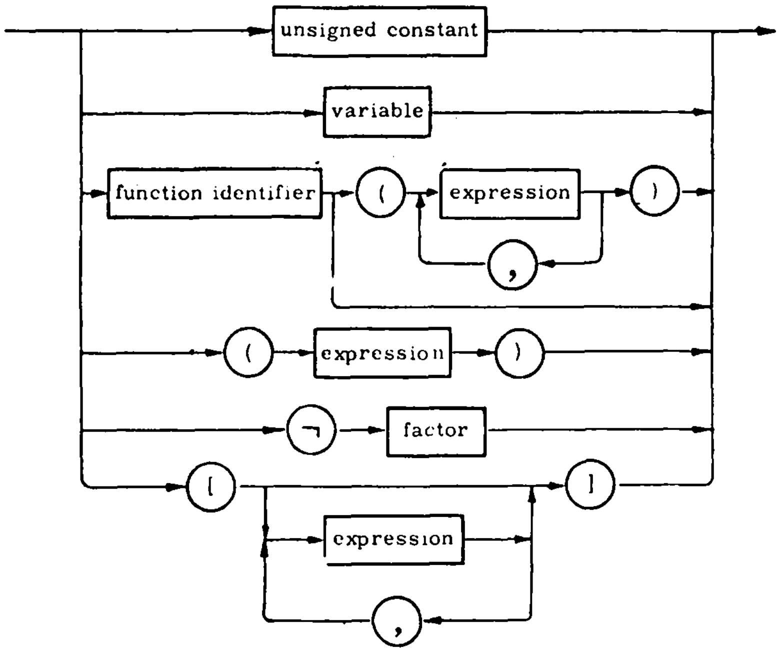
field list



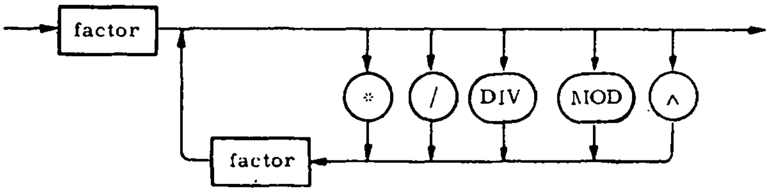
variable



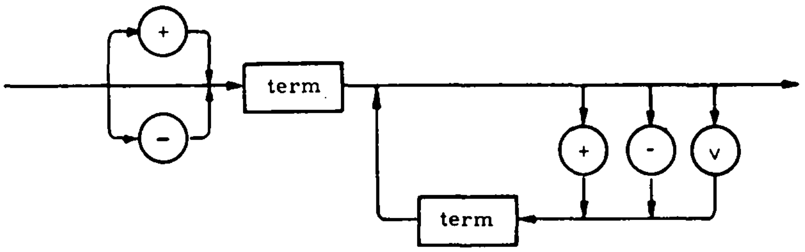
factor



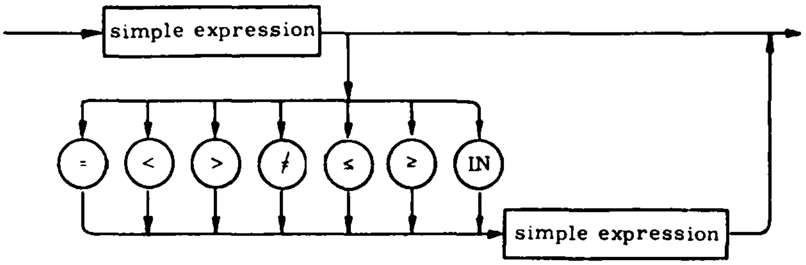
term



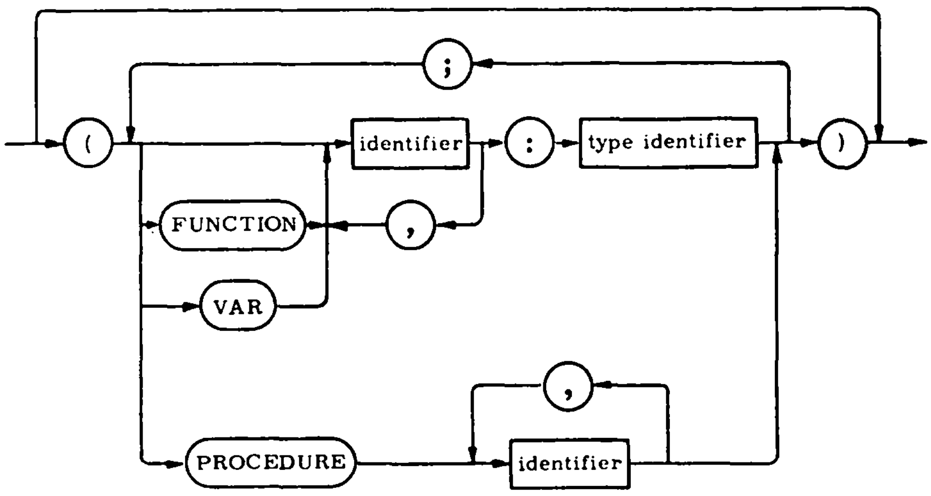
simple expression

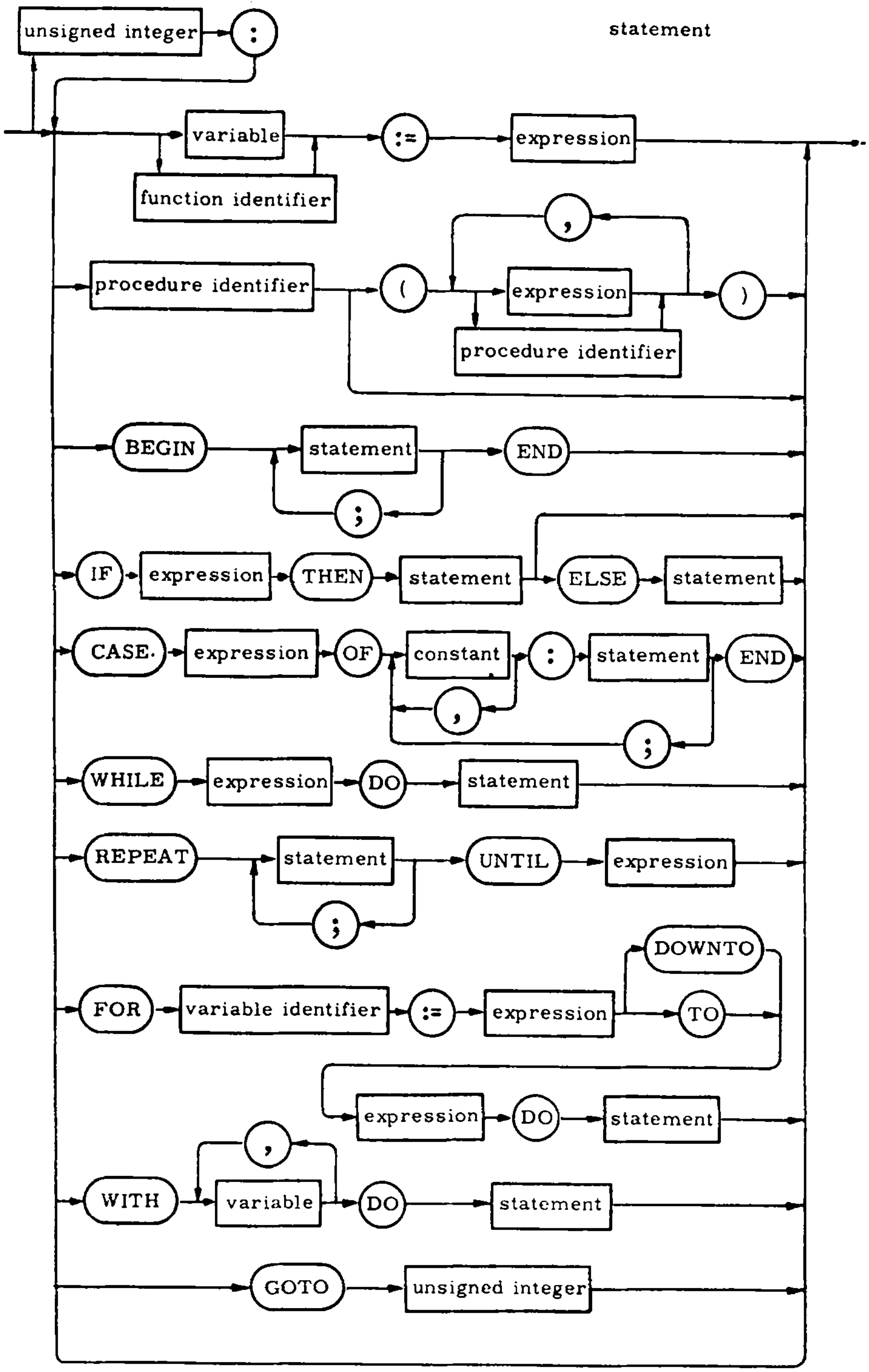


expression

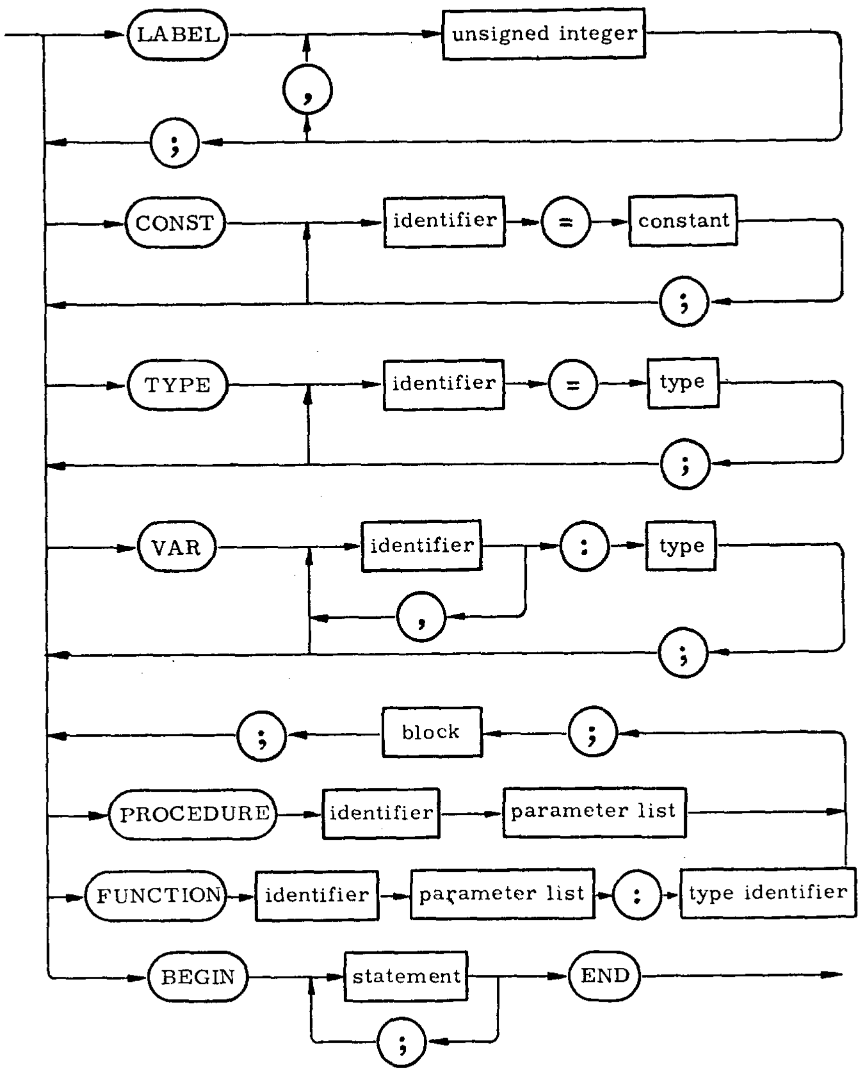


parameter list

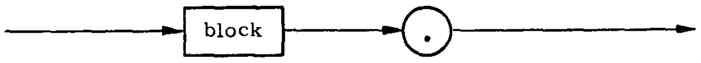



statement
unsigned integer
:
variable
function identifier
:=
expression
procedure identifier
(
,
expression
procedure identifier
)
BEGIN
statement
;
END
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
,
:
statement
;
END
WHILE
expression
DO
statement
REPEAT
statement
;
UNTIL
expression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

block



program



Appendix 2

Procedures

Consider the trivial procedure, with only one variable parameter, one value parameter, and no non-local references.

```
procedure $p$ (var $a$: integer; $b$: integer);
  begin $b := 2*b$; if $a > b$ then $a := b$ end.
```

Letting S stand for the body of this procedure, it is easy to prove

$$(b = b_0) \wedge (b > 0)\{S\}a \leqq 2 * b_0.$$

The invocation of this procedure will (in general) change the value of a in some manner dependent on the initial values of a and b (and initial values of non-local variables, if referenced from within S). Let us introduce a new function symbol f to denote this dependency, so that the final value of a can be denoted as

$$f(a, b).$$

Now the effect of any call of $P(x, y)$ is (by definition of f) equivalent to the simple assignment

$$x := f(x, y),$$

and using the axiom of assignment (11.1) we may validly conclude for any R that

$$R^x_{f(x,y)}\{p(x, y)\}R.$$

But by itself, application of this rule would be useless, since f has been introduced as an arbitrary function symbol. We need therefore to know at least something of the properties of f, and these can only be derived from the properties of the body of the procedure p. Suppose that we have proved

$$P\{S\}Q.$$

Occurrences of a in Q refer to the value of a *after* the execution of S, namely $f(a, b)$. Suppose that Q does not refer to any other program variable (in particular, does not refer to b). Then the values of all free variables of $Q^a_{f(a,b)}$ will be the same as they were in P, since the (a, b) in $f(a, b)$ also refer to the initial values of the parameters. Thus we may validly form the implication

$$P \supset Q^a_{f(a,b)},$$

and this implication will be true for all values of the variables involved in it. Thus, in the case shown above, we have

$$\forall a, b, b_0 ((b = b_0) \wedge (b > 0) \supset f(a, b) \leqq 2 * b_0)$$

or more simply

$$\forall a, b (b > 0 \supset f(a, b) \leqq 2 * b).$$

References

1. Wirth, N.: The programming language PASCAL. Acta Informatica **1**, 35–63 (1971)
2. Naur, P. (Ed.): Revised report on the algorithmic language ALGOL 60. Comm. ACM **6**, 1–17 (1963); Comp. J. **5**, 349–367 (1962/63); Numer. Math. **4**, 420–453 (1963)
3. Wirth, N.: The design of a PASCAL compiler. Software, Practice and Experience **1**, 309–333 (1971)
4. Welsh, J., Quinn, C.: A PASCAL compiler for the ICL 1900 Series Computers. Software, Practice and Experience **2**, 73–77 (1972)
5. Hoare, C. A. R.: An axiomatic basis for computer programming. Comm. ACM **12**, 576–581 (1969)
6. Hoare, C. A. R.: An axiomatic definition of the programming language PASCAL, Second Draft. Proc. Symposium on Theoretical Programming, Novosibirsk, Aug. 1972
7. Hoare, C. A. R.: Notes on data structuring. In: Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R.: Structured programming. London und New York: Academic Press 1972
8. Hoare, C. A. R.: A note on the for statement. BIT **12**, 334–341 (1972)

C. A. R. Hoare
Computer Science Dept.
The Queen's University
Belfast BT7 1NN
Northern Ireland

N. Wirth
Eidgenössische Technische Hochschule
Fachgruppe Computer-Wissenschaften
CH-8006 Zürich
Clausiusstr. 55
Schweiz