New Insights from Old Programs

The Structure of The First ALGOL 60 System

van den Hove d'Ertsenryck, G.M.C.J.T.G.

Publication date
2019

Document Version
Other version

License
Other

Link to publication

Citation for published version (APA):
van den Hove d'Ertsenryck, G. M. C. J. T. G. (2019). *New Insights from Old Programs: The Structure of The First ALGOL 60 System*. [Thesis, externally prepared, Universiteit van Amsterdam].

APPENDIX A

ALGOL: THE 1958, 1960 & 1962 REPORTS

§ A.1 The 1958 Preliminary Report

Preliminary Report — International Algebraic Language[1]

by

the ACM Committee on Programming Languages and
the GAMM Committee on Programming

edited by A. J. Perlis and K. Samelson

Note. In the interest of immediate circulation of the results of the ACM-GAMM committee work on an algebraic programming language, this preliminary report is presented. The language described naturally enough represents a compromise, but one based more upon differences of taste than on content or fundamental ideas. Even so, it provides a natural and simple medium for the expression of a large class of algorithms. This report has not been thoroughly examined for errors and inconsistencies. It is anticipated that the committee will prepare a more complete description of the language for later publication.
For all scientific purposes, reproduction of this report is explicitly permitted without any charge.

Acknowledgments. The members of the conference wish to express their appreciation to the Association for Computing Machinery, the Deutsche Forschungsgemeinschaft, and the Eidgenössische Technische Hochschule Zürich, for substantial help in making this conference and resultant report possible.

Part I. Introduction

In 1955, as a result of the Darmstadt meeting on electronic computers, the GAMM (Gesellschaft für Angewandte Mathemathik und Mechanik), Germany, set up a committee on programming (Programmierungsausschuß). Later a subcommittee began to work on formula translation and on the construction of a translator, and a considerable amount of work was done in this direction.
A conference attended by representatives of the USE, SHARE, and DUO organizations and the ACM (Association for Computing Machinery) was held in Los Angeles on May 9 and 10, 1957 for the purpose of examining ways and means for facilitating exchange of all types of computing information. Among other things, these conferees felt that a single universal computer language would be very desirable. Indeed, the successful exchange of programs within various organizations such as USE and SHARE had proved to be very valuable to computer installations. They accordingly recommended that the ACM appoint a committee to study and recommend action toward a universal programming language.
By October 1957 the GAMM group, aware of the existence of many programming languages, concluded that rather than present still another formula language, an effort should be made toward unification. Consequently, on October 19, 1957, a letter was written to Prof. John W. Carr III, president of the ACM. The letter suggested that a joint conference of representatives of the GAMM and ACM be held in order to fix upon a common formula language in the form of a recommendation.
An ACM Ad-Hoc committee was then established by Dr. Carr, which represented computer users, computer manufacturers, and universities. This committee held three meetings starting on January 24, 1958 and discussed many technical details of programming language. The language that evolved from these meetings was oriented more towards problem language than towards computer language and was based on several existing programming systems. On April 18, 1958 the committee appointed a sub-committee to prepare a report giving the technical specifications of a proposed language.
A comparison of the ACM committee proposal with a similar proposal prepared by the GAMM group (presented at the above-mentioned ACM Ad-Hoc committee meeting of April 18, 1958) indicated many common features. Indeed, the GAMM group had planned on its own initiative to

1. [Alternative title:] **Report on the Algorithmic Language ALGOL**

use English words wherever needed. The GAMM proposal represented a great deal of work in its planning and the proposed language was expected to find wide acceptance. On the other hand the ACM proposal was based on experience with several successful, working problem oriented languages.

Both the GAMM and ACM committees felt that because of the similarities of their proposals there was an excellent opportunity for arriving at a unified language. They felt that a joint working session would be very profitable and accordingly arranged for a conference in Switzerland to be attended by four members from the GAMM group and four members from the ACM committee. The meeting was held in Zürich, Switzerland, from May 27 to June 2, 1958 and attended by F. L. Bauer, H. Bottenbruch, H. Rutishauser and K. Samelson from the GAMM committee and by J. W. Backus, C. Katz, A. J. Perlis, and J. H. Wegstein for the ACM committee.[2]

It was agreed that the contents of the two proposals should form the agenda of the meeting, and the following objectives were agreed upon:

I. The new language should be as close as possible to standard mathematical notation and be readable with little further explanation.

II. It should be possible to use it for the description of computing processes in publications.

III. The new language should be mechanically translatable into machine programs.

There are certain differences between the language used in publications and a language directly usable by a computer. Indeed, there are many differences between the sets of characters usable by various computers. Therefore, it was decided to focus attention on three different levels of language, namely a *Reference Language*, a *Publication Language* and several *Hardware Representations.*

Reference Language

1. It is the working language of the committee.
2. It is the defining language.
3. It has only one unique set of characters.
4. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.
5. It is the basic reference and guide for compiler builders.
6. It is the guide for all hardware representations.
7. *It will not normally be used stating problems.*
8. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
9. The main publications of the common language itself will use the reference representation.

Publication Language (see Part IIIc)

1. The description of this language is in the form of permissible variations of the reference language (*e. g.*, subscripts, spaces, exponents, Greek letters) according to usage of printing and handwriting.
2. It is used for stating and communicating problems.
3. The characters to be used may be different in different countries but univocal correspondence with reference representation must be secured.

Hardware Representations

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication language.

2. In addition to the members of the conference, the following people participated in the preliminary work of these committees: GAMM: P. Graeff, P. Läuchli, M. Paul, F. Penzlin; ACM: D. Arden, J. McCarthy, R. Rich, R. Goodman, W. Turanski, S. Rosen, P. Desilets, S. Gorn, H. Huskey, A. Orden, D. C. Evans.

Part II. Description of the reference language

1. Structure of the language

As stated in the introduction, the algorithmic language has three different kinds of representations —reference, hardware, and publication— and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols — and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.
The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language — explicit formulæ— called arithmetic statements.
To show the flow of larger computational processes, certain non-arithmetic statements are added which may describe *e. g.*, alternatives, or recursive repetitions of computing statements.
Statements may be supported by declarations which are not themselves computing rules, but inform the translator of certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers or even the set of rules defining a function.
Sequences of statements and declarations, when appropriately combined, are called programs. However, whereas complete and rigid formal rules for constructing translatable statements are described in the following, no such rules can be given in the case of programs. Consequently, the notion of program must be considered to be informal and intuitive, and the question whether a sequence of statements may be called a program should be decided on the basis of the operational meaning of the sequence.
In the sequel explicit rules —and associated interpretations— will be given describing the syntax of the language. Any sequence of symbols to which these rules do not assign a specific interpretation will be considered to be undefined. Specific translators may give such sequences different interpretations.

2. Basic symbols

The reference language is built up from the basic symbols listed in Part IIIa.
These are:
1. Letters λ (the standard alphabet of small and capital letters)
2. Figures ζ (arabic numerals *0*, …, *9*)
3. Delimiters δ consisting of
   a) operators ω:
      arithmetic operators $+\ -\ \times\ /$
      relational operators $<\ \leqq\ =\ \geqq\ >\ \neq$
      logical operators $\neg\ \vee\ \wedge\ \equiv$
      sequential operators **go to do return stop for if if either or if**
   b) separators σ: $,\ :\ ;\ :=\ =:\ \rightarrow\ _{10}\ .$
   c) brackets β: $(\ )\ [\ ]\ \uparrow\ \downarrow$ **begin end**
   d) declarators φ: **procedure array switch type comment**

Of these symbols, letters do not have individual meaning. Figures and delimiters have a fixed meaning which for the most part is obvious, or else will be given at the appropriate place in the sequel.
Strings of letters and figures enclosed by delimiters represent new entities. However, only two types of such strings are admissible:
1. Strings consisting of figures ζ only represent the *(positive) integers* **G** (including *0*) with the conventional meaning.

2. Strings beginning with a letter λ followed by arbitrary letters λ and/or figures ζ are called *identifiers*.
They have no inherent meaning, but serve for identifying purposes only.

3. Expressions

Arithmetic and logical processes (in the most general sense), which the algorithmic language is primarily intended to describe, are given by arithmetic and logical expressions, respectively. Constituents of these expressions, except for certain delimiters, are numbers, variables, elementary arithmetic operators and relations, and other operators called functions. Since the description of both variables and functions may contain expressions, the definition of expressions, and their constituents, is necessarily recursive.
The following are the units from which expressions are constructed:

i) (Positive) Numbers N

Form: $N \sim G.G_{10}\pm G$
where each G is an integer as defined above.
$G.G$ is a decimal number of conventional form. The scale factor $_{10}\pm G$ is the power of ten given by $\pm G$. The following constituents of a number may be omitted in any occurrence:
the fractional part *.00...0* of integer decimal numbers;
the integer *1* in front of a scale factor;
the $+$ sign in the scale factor;
the scale factor $_{10}\pm 0$.
Examples:
4711
137.06
$2.9997_{10}10$
$_{10}-12$
$3_{10}-12$

ii) Simple variables V

Simple variables V are designations for arbitrary scalar quantities, *e. g.*, numbers as in elementary arithmetic.
Form: $V \sim I$
where I is an identifier as defined above.
Examples:
a
x11
PSI2
ALPHA

iii) Subscripted variables V

Subscripted variables V designate quantities which are components of multidimensional arrays.
Form: $V \sim I[\ell]$
where $\ell \sim E, E, \ldots, E$
is a *list* of arithmetic expressions as defined below. Each expression E occupies one subscript position of the subscripted variable, and is called a *subscript.* The complete list of subscripts is enclosed in the subscript brackets [ ].
The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (*cf. arithmetic expressions*).
Subscripts, however, are intrinsically integer-valued, and whenever the value of a subscript expression is not integral, it is replaced by the nearest integer (in the sense of proper round off).

Variables (both simple and subscripted ones) designate arbitrary real numbers unless otherwise specified. However, certain declarations (*cf. type declarations*) may specify them to be of a special type, *e. g.*, *integral*, or *Boolean*. Boolean (or logical) variables may assume only the two values "true" and "false."

iv) Functions F

Functions F represent single numbers (function values), which result through the application of given sets of rules to fixed sets of parameters.
Form: $F \sim I\,(P,\, P,\, \ldots,\, P)$
where I is an identifier, and $P,\, P,\, \ldots,\, P$ is the ordered list of actual parameters specifying the parameter values for which the function is to be evaluated. A syntactic definition of parameters is given in the sections on *function declarations* and *procedure declarations*. If the function is defined by a function declaration, the parameters employed in any use of the function are expressions compatible with the type of variables contained in the corresponding parameter positions in the function declaration heading (*cf. function declaration*). Admissible parameters for functions defined by *procedure declarations* are the same as admissible input parameters of procedures as listed in the section on *procedure statements*.
Identifiers designating functions, just as in the case of variables, may be chosen according to taste. However, certain identifiers should be reserved for the standard functions of analysis. This reserved list should contain:

abs (E)	for the modulus (absolute value) of the value of the expression E
sign (E)	for the sign of the value of E
entier (E)	for the largest integer not greater than the value of E
sqrt (E)	for the square root of the value of E
sin (E)	for the sine of the value of E

and so on according to common mathematical notation.

v) Arithmetic expressions E

Arithmetic expressions E are defined as follows:
A number, a variable (other than Boolean), or a function is an expression.
Form: $E \sim N$
$\sim V$
$\sim F$
If E_1 and E_2 are expressions, the first symbols of which are neither "+" nor "−", then the following are expressions:
1. $E \sim +\, E_1$
2. $\sim -\, E_2$
3. $\sim E_1 + E_2$
4. $\sim E_1 - E_2$
5. $\sim E_1 \times E_2$
6. $\sim E_1 \;/\; E_2$
7. $\sim E_1 \uparrow E_2 \downarrow$
8. $\sim (E_1)$

The operators $+$, $-$, $\times$, $/$ appearing above have the conventional meaning. The parentheses $\uparrow\ \downarrow$ denote exponentiation, where the leading expression is the base and the expression enclosed in parentheses is the exponent.
Examples:[3]
$2 \uparrow 2 \uparrow n \downarrow \downarrow$ means $2^{(2^n)}$
$2 \uparrow 2 \downarrow \uparrow n \downarrow$ means $(2^2)^n$

3. [Third example:] $a \uparrow 2 \uparrow b \downarrow \downarrow \uparrow 2 \downarrow$ means $(a^{2^b})^2$

The proper interpretation of expressions can always be arranged by appropriate positioning of parentheses.
An arithmetic expression is a rule for computing one real number by executing the indicated arithmetic operations on the actual numerical values of the constituents of the expression. This value is obvious in the case of numbers N. For variables V, it is the current value (assigned last in the dynamic sense), and for functions F it is the value arising from the computing rules defining the function (*cf. function declaration*) when applied to the current values of the function parameters given in the expression.
The sequence of operations within one expression is generally from left to right, with the following additional rules:
a) parentheses are evaluated separately
b) for operators, the conventional rule of precedence applies:
first: $\times$ $/$
second: $+$ $-$
In order to avoid misunderstandings, redundant parentheses should be used to express, for example, $\frac{ab}{c}$ in the form $(a\times b)/c$ or $(a/c)\times b$ rather than by $a\times b/c$, or $a/c\times b$ respectively, and to avoid constructions such as $a/b/c$.
Examples:
A
Alpha
Degree
$A[1, 1]$
$A[j + k - 2, j - k]$
$A[mu[s]]$
$a \times sin\ (omega \times t)$
$0.5 \times a[N \times (N - 1)/2, 0]$

vi) Boolean expressions B

Boolean expressions B are defined analogously to arithmetic expressions:
a) A truth value, a variable (Boolean by declaration), or a function (Boolean by declaration) is an expression.
Form: $B \sim 0$ (the truth value "false")
~ 1 (the truth value "true")
$\sim V$
$\sim F$
b) If E_1 and E_2 are arithmetic expressions, then the following arithmetic relations are expressions:
$B \sim (E_1 < E_2)$
$\sim (E_1 \leqq E_2)$
$\sim (E_1 \neq E_2)$
$\sim (E_1 \geqq E_2)$
$\sim (E_1 > E_2)$
$\sim (E_1 = E_2)$
Such expressions take on the (current) value "true" whenever the corresponding relation is satisfied for the expressions involved, otherwise "false."
c) If B_1 and B_2 are expressions, the following are expressions:
$B \sim \neg B_1$
$\sim B_1 \vee B_2$
$\sim B_1 \wedge B_2$
$\sim B_1 \equiv B_2$
$\sim (B_1)$

The operators $\neg$, $\vee$, $\wedge$, $\equiv$ have the interpretations "not," "or," "and," and "equivalent." Interpretation of the binary operators will be from left to right. The scope of $\neg$ is the first expression to its right. Any other desired precedence must be indicated by the use of parentheses.
Examples:
$(x = 0)$
$(X > 0) \vee (y > 0)$
$(p \wedge q) \vee (x \neq y)$

4. Statements Σ

Closed and self-contained rules of operations are called statements Σ. They are defined recursively in the following way:
a) Basic statements Σ are those described in this section.
b) Strings of one or more statements[4] may be combined into a single (compound) statement by enclosing them within the "statement parentheses" **begin** and **end**. Single statements are separated by the statement separator ";".
Form: $\Sigma \sim$ **begin** Σ; Σ; . . .; Σ **end**
c) A statement may be made identifiable by attaching to it a label L, which is an identifier I, or an integer G (with the meaning of identifier). The label precedes the attached statement being labeled, and is separated from it by the separator colon (:). Label and statement together constitute a statement called "labeled statement."
Form: $\Sigma \sim L$: Σ
A labeled statement may not itself be labeled. In the case of labeled compound statements, the closing parenthesis **end** may be followed by the statement label (followed by the statement separator) in order to indicate the range of the compound statement:
Form: $\Sigma \sim L$: **begin** Σ; Σ; . . .; Σ **end** L;

i) Assignment statements

Assignment statements serve for assigning the value of an expression to a variable.
Form a) $\Sigma \sim V := E$
If the expression on the right hand side of the assignment delimiter := is arithmetical, the variable V on the left hand side must also be numerical, *i. e.*, it must not be Boolean.
Generally, the arithmetic type of the expression E is determined by the constituents and operations of the expression E. However, V may be declared to be of a special type provided this declaration is compatible with the possible values of the expression E.
Form b) $\Sigma \sim V := B$
If the expression on the right hand side of the assignment statement is Boolean, V may be any variable. This means that the truth values "true" and "false" of the Boolean expression may be interpreted arithmetically as integers "*1*" and "*0*", which may then be assigned to a numerical variable.

ii) Go to statements

Normally, the sequence of operations (described by the statement of a program) coincides with the physical sequence of statements. This normal sequence of execution may be interrupted by the use of *go to* statements.
Form: $\Sigma \sim$ **go to** D
D is a *designational expression* specifying the label of the statement to be executed next. It is either a label L or a switch variable $I[E]$ (*cf. switch declaration*), where I is an identifier and E a subscript expression. In the latter case, the numerical value of E (the value of the subscript) is an ordinal which identifies the component of the switch I (named by declaration). This element

4. *Declarations*, which may be interspersed between statements, have no operational (dynamic) meaning. Therefore, they have no significance in the definition of compound statements.

which is again a designational expression specifies the label to be used in the *go to* statement. This label determination is obviously a recursive process, since the elements of the switch may again be switch variables.
Examples:
go to *hell*
go to $exit[(i \uparrow 2 \downarrow - j \uparrow 2 \downarrow + 1) \ / \ 2]$
where *exit* refers to the declaration
switch $exit := (D_1, D_2, \ldots, D_n)$

iii) If statements

The execution of a statement may be made to depend upon a certain condition which is imposed by preceding the statement in question by an *if* statement.
Form: $\Sigma \sim$ **if** B
where B is a Boolean expression.
If the value of B is "true," the statement following the *if* statement will be executed. Otherwise, it will be bypassed, and operation will be resumed with the next statement following.
Example: In the sequence of statements
if $(a > 0)$; $c := a \uparrow 2 \downarrow \times b \uparrow 2 \downarrow$;
if $(a < 0)$; $c := a \uparrow 2 \downarrow + b \uparrow 2 \downarrow$;
if $(a = 0)$; **go to** *bed*
one and only one of the three statements rightmost in each line will be executed.

iv) For statements

Recursive processes may be initiated by use of a *for* statement, which causes the following statement to be executed several times, once for each of a series of values assigned to the recursing variable contained in the *for* statement.
Form: a) $\Sigma \sim$ **for** $V := \ell$
b) $\Sigma \sim$ **for** $V := E_{i_1}(E_{s_1})E_{e_1}, \ldots, E_{i_k}(E_{s_k})E_{e_k}$
where ℓ is a list of k expressions $E_1, E_2, \ldots, E_k$, and E_{i_j}, E_{s_j}, E_{e_j} are expressions.
In form a) the intent is to assign to V in succession the value of each expression of the list (expressions taken in order of listing) and the statement following the *for* statement is executed immediately following each such assignment.
In form b) each group of expressions $E_i(E_s)E_e$ determines an arithmetic progression. The value of E_i is the initial value, E_s gives the value of the increment (step), and E_e determines the end value which is the last term of the progression contained in the interval $[E_i, E_e]$. The intent is to assign to V each value of every progression (these again taken in the order of listing from left to right), and the statement following the *for* statement is executed immediately following each such assignment.
The effect of a *for* statement may be precisely described in terms of "more elementary" statement forms. Thus form (a) is precisely equivalent to:
$V := E_1$; Σ; $V := E_2$; Σ; $\ldots$ $V := E_k$; Σ
where Σ is the statement following the *for* statement.
Form (b) is precisely equivalent to:
$V := E_{i_1}$; L_1: Σ;[5] $V := E_{i_1} + E_{s_1}$; **if** $(V \leqq$[6] $E_{e_1})$; **go to** L_1;
$\vdots$
$V := E_{i_k}$; L_k: Σ; $V := E_{i_k} + E_{s_k}$; **if** $(V \leqq E_{e_k})$; **go to** L_k;
where Σ is the statement following the *for* statement.

5. If Σ is a labeled statement, L_1 is that label. If not, the effect is as though it had a (unique) label L_1. [Additional sentence:] L_k $(k \neq 1)$ are theoretically unique labels.

6. This relational form obtains if the progression is increasing; if decreasing, the relation $\geqq$ is understood to be employed.

Examples:
a) **for** $I := 1(1)n$; $p := p \times y + A[I]$
b) **for** $a := 1,\ 3,\ 5,\ 9.76,\ \ldots,\ -13.75$; **begin** ... **end**

v) Alternative statements

An alternative statement is one which has the effect of selecting for execution one from a set of given statements in accordance with certain conditions which exist when the statement is encountered.
Form: $\Sigma \sim$ **if either** B_1; Σ_1; **or if** B_2; ...; **or if** B_k; Σ_k; **end**
where Σ_i is any statement other than a quantifier, *i. e.*, **if**, **for**, or **or if**, and B_i is any Boolean expression.
The effect of an alternative statement may be precisely described in terms of "more elementary" statement forms. Thus the above form is precisely equivalent to the sequence of statements:
if B_1; **begin** Σ_1; **go to** *next* **end**; **if** B_2; **begin** Σ_2; **go to** *next* **end**; ...; **if** B_k; **begin** Σ_k; **go to** *next* **end**;
where "*next*" is the label of the statement following the alternative statement.
Example:
if either $(a > 0)$; $y := a + 2$; **or if** $(a < 0)$; $y := a/2$; **or if** $(a = 0)$; $y := 0.57$ **end**

vi) Do statements

A statement, or string of statements, once written down, may be entered again (in the sense of copying) in any place of the same program by employing a *do* statement which during copying permits substitution for certain constituents of the statements reused.
Form: $\Sigma \sim$ **do** L_1, L_2 $(S_{\rightarrow} \rightarrow I, \ldots, S_{\rightarrow} \rightarrow I)$
where L_1 and L_2 are labels, the $S_{\rightarrow}$ are strings of symbols not containing the separator $\rightarrow$ and the I are identifiers, or labels, and the list enclosed by parentheses is a substitution list.
The *do* statement operates on the string of statements from, and including, the one labeled L_1 through the one labeled L_2, which statements constitute the range of the *do* statement. If L_1 is equal to L_2, *i. e.*, if the range is just the one statement L_1, the characters ", L_2" may be omitted. The *do* statement causes itself to be replaced by a copy of the string of statements constituting its range. However, in this copy all identifiers or labels, listed on the right-hand side of a separator "$\rightarrow$" in the substitution list of the *do* statement, (and which are utilized in these statements) are replaced by the corresponding strings of symbols $S_{\rightarrow}$ on the left hand side of the separators "$\rightarrow$". These strings $S_{\rightarrow}$ may be chosen freely with the one restriction that the substitutions produce formally correct statements in the copy.[7]
Whenever a *do* statement contains in its range another *do* statement, the copying and substituting process for this second innermost *do* will be executed first. Therefore the (actual) copy induced from a *do* statement never contains a *do* statement.
Declarations within the range of a *do* statement are not reproduced in the copy.
Examples:
do $5, 12\ (x[i] \rightarrow y,$ *black label* $\rightarrow$ *red label*$, \ldots, f\,(x, y) \rightarrow g)$
do $12A, ABC\ (x \uparrow 2 \downarrow + 3/y \rightarrow A, \ldots)$
The range of a *do* statement should contain complete statements only, *i. e.*, if the **begin** (**end**) delimiter of a compound statement lies in the range of the *do*, then so should the matching **end** (**begin**). If this rule is not complied with the result of the *do* statement may not be the one desired.

vii) Stop statements

Stop is a delimiter which indicates an operational (dynamic) end of the program containing it. Operationally, it has no successor statement.
Form: $\Sigma \sim$ **stop**

7. Thus, in the copy produced, any designational expression whose range is a statement within the range of the *do* statement must be transformed so that its range refers to the copy produced.

viii) Return statements

Return is a delimiter which indicates an operational end of a *procedure*. It may appear only in a *procedure declaration* (*cf. procedure declaration*).
Form: $\Sigma \sim$ **return**

ix) Procedure statements

A procedure statement serves to initiate (call for) the execution of a *procedure*, which is a closed and self-contained process with a fixed ordered set of input and output parameters, permanently defined by a *procedure declaration* (*cf. procedure declaration*).
Form: $\Sigma \sim I\,(P_i,\ P_i,\ \ldots,\ P_i) =: (P_o,\ P_o,\ \ldots,\ P_o)$
Here I is an identifier which is the name of some procedure, *i. e.*, it appears in the heading of some procedure declaration (*cf. procedure declaration*). P_i, P_i, …, P_i is the ordered list of actual input parameters specifying the input quantities to be processed by the procedure. P_o, P_o, …, P_o is the ordered list of actual output parameters specifying the variables to which the results of the procedure will be assigned and alternate exits, if any.[8] The procedure declaration defining the called procedure contains, in its heading, a string of symbols identical in form to the procedure statement, and the formal parameters occupying input and output parameter positions there give complete information concerning the admissibility of parameters used in any procedure call shown by the following replacement rules:

formal parameters in procedure declaration	admissible parameters in procedure statement
input parameters:	
single identifier (formal variable)	any expression (compatible with the type of the formal variable)
array, *i. e.*, subscripted variable with k ($\geqq 1$) empty parameter positions	array with n ($\geqq k$) parameter positions k of which are empty
function with k empty parameter positions	function with n ($\geqq k$) parameter positions k of which are empty
procedure with k empty parameter positions	procedure with k empty parameter positions
parameter occurring in a procedure (added as a primitive to the language)[9]	every string of symbols S, which does not contain the symbol "," (comma)
output parameters:	
single identifier (formal variable)	simple or subscripted variable
array (as above for input parameters)	array (as above for input parameters)
(formal) label	label

If a parameter is at the same time an input and output parameter this parameter must obviously meet the requirements of both input and output parameters.
Within a program, a procedure statement causes execution of the procedure called by the statement. The execution, however, is effected as though all formal parameters listed in the procedure declaration heading were replaced, throughout the procedure, by the actual parameters listed, in the corresponding position, in the procedure statement.
This replacement may be considered to be a replacement of every occurrence within the procedure of the symbols, or sets of symbols, listed as formal parameters, by the symbols, or sets of symbols, listed as actual parameters in the corresponding positions of the procedure statement, after

8. [Alternative wording:] The list of actual output parameters P_o, P_o, …, P_o specifies the variables to which the results of the procedure will be assigned, and alternate exits if any.

9. Within a program certain procedures may be called which are themselves not defined by procedure declarations in the program, *e. g.*, input-output procedures. These procedures may require as parameters quantities outside the language, *e. g.*, a string of characters providing input-output format information.

enclosing in parentheses every expression not enclosed completely in parentheses already.
Furthermore, any *return* statement is to be replaced by a *go to* statement referring, by its label, to the statement following the *procedure* statement, which, if originally unlabeled, is treated as having been assigned a (unique) label during the replacement process.
The values assignable to,[10] or computable by, the actual input parameters must be compatible with type declarations concerning the corresponding formal parameters which appear in the procedure.
For actual output parameters, only type declarations duplicating given type declarations for the corresponding formal parameters may be made.
Array declarations concerning actual parameters must duplicate, in corresponding subscript positions, array declarations referring to the corresponding formal parameters.

5. Declarations Δ

Declarations serve to state certain facts about entities referred to within the program. They have no operational meaning, and within a given program their order of appearance is immaterial. They pertain to the entire program (or procedure) in which they occur, and their effect is not alterable by the running history of the program.

i) Type declarations

Type declarations serve to declare certain variables, or functions, to represent quantities of a given class, such as the class of integers, or class of Boolean values.
Form: $\Delta \sim$ **type** $(I, I, \ldots, I, I[\,], \ldots, I[,], \ldots, I[,,], \ldots)$
where **type** is a symbolic representative of some type declarator such as **integer** or **boolean**, and the I are identifiers.
Throughout the program, the variables, or functions named by the identifiers I, are constrained to refer only to quantities of the type indicated by the declaration.[11] On the other hand, all variables, or functions which are to represent other than arbitrary real numbers must be so declared.

ii) Array declarations

Array declarations give the dimensions of multidimensional arrays of quantities.
Form: $\Delta \sim$ **array** $(I, I, \ldots, I[\ell{:}\ell'], I, I, \ldots, I[\ell{:}\ell'], \ldots)$
where **array** is the array declarator, the I are identifiers, and the "ℓ" and "ℓ'" are lists of integers separated by commas.
Within each pair of brackets, the number of positions of ℓ must be the same as the number of positions of ℓ'.
Each pair of lists enclosed in brackets $[\ell{:}\ell']$ indicates that the identifiers contained in the list $I, I, \ldots, I$ immediately preceding it are the names of arrays with the following common properties:
a) the number of positions of ℓ is the number of dimensions of every array.
b) the values of ℓ and ℓ' are the lower and upper bounds of values of the corresponding subscripts of every array.
An array is defined only when all upper subscript bounds are not smaller than the corresponding lower bounds.

iii) Switch declarations

A switch declaration specifies the set of designational expressions represented by a switch variable. If used in a *go to* statement, its value specifies the label of the statement called by the *go to* statement (*cf. go to* statement).
Form: $\Delta \sim$ **switch** $I := (D_1, D_2, \ldots, D_n)$

10. [Alternative wording:] The values assigned to,

11. [Alternative wording:] indicated by the declarator.

where **switch** is the switch declarator, I is an identifier, and the D_i are designational expressions (*cf. go to* statement).
The switch declaration declares the list D_1, D_2, …, D_n to be a symbolic vector (the "switch"), the designational expression D_k being the k^{th} component. Reference is made to the switch by the switch variable $I[E]$, where I is the switch identifier and E is a subscript expression. The switch variable, when used in *go to* statements, selects by the actual value of its subscript that component of the switch determining the label called for by the *go to* statement. A switch variable, being a designational expression, may appear as a component of a switch.

iv) Function declarations

A function declaration declares a given expression to be a function of certain of its variables. Thereby, the declaration gives (for certain simple functions) the computing rule for assigning values to the function (*cf. functions*) whenever this function appears in an expression.
Form: $\Delta \sim I_N\ (I, I, \ldots, I) := E$
where the I are identifiers and E is an expression which, among its constituents, may contain simple variables named by identifiers appearing in the parentheses.
The identifier I_N is the function name. The identifiers in parentheses designate the formal parameters of the function.
Whenever the function $I_N\ (P, P, \ldots, P)$ appears in an expression (a *function call*) the value assigned to the function in actual computation is the computed value of the defining expression E. For the evaluation, every variable V which is listed as a parameter I in the *function declaration*, is assigned the current value of the actual parameter P in the corresponding position of the parameter list of the function in the function call. The (formal) variables V in E which are listed as parameters in the declaration bear no relationship to variables possessing the same identifier, but appearing elsewhere in the program. All variables other than parameters appearing in E have values as currently assigned in the program.
Example:
$I\ (Z) := Z + 3 \times y$
…
$alpha := q + I\ (h + 9 \times mu)$
In the statement assigning a value to alpha the computation is:
$alpha := q + ((h + 9 \times mu) + 3 \times y)$

v) Comment declarations

Comment declarations are used to add to a program informal comments, possibly in a natural language, which have no meaning whatsoever in the algorithmic language, no effect on the program, and are intended only as additional information for the reader.
Form: $\Delta \sim$ **comment** $S_;$
where **comment** is the comment declarator, and $S_;$ is any string of symbols not containing the symbol ";".

vi) Procedure declarations

A procedure declaration declares a program to be a closed unit (a procedure) which may be regarded as a single compound operation (in the sense of a generalized function) depending on a certain fixed set of input parameters, yielding a fixed set of results designated by output parameters, and having a fixed set of possible exits defining possible successors.
Execution of the procedure operation is initiated by a *procedure statement* which furnishes values for the input parameters, assigns the results to certain variables as output parameters, and assigns labels to the exits.
Form: $\Delta \sim$ **procedure** $I\ (P_i) =: (P_o),\ I\ (P_i) =: (P_o),\ \ldots,\ I\ (P_i) =: (P_o)$
$\Delta;\ \Delta;\ \ldots;\ \Delta;$ **begin** $\Sigma;\ \Sigma;\ \ldots;\ \Delta;\ \Delta;\ \ldots;\ \Sigma;\ \Sigma$ **end**

Here, the I are identifiers giving the names of the different procedures contained in the procedure declaration. Each P_i represents an ordered list of formal input parameters, each P_o a list of formal output parameters which include any exits required by the corresponding procedures.

Some of the strings "=: (P_o)" defining outputs and exits may be missing, in which case the corresponding symbols "I (P_i)" define a procedure that may be called within expressions.

The Δ's in front of the delimiter **begin** are declarations concerning only input and output parameters. The entire string of symbols from the declarator **procedure** (inclusive) up to the delimiter **begin** (exclusive) is the procedure heading. Among the statements enclosed by the parentheses **begin** and **end** there must be, for each identifier I listed in the heading as a procedure name, exactly one statement labeled with this identifier, which then serves as the entry to the procedure. For each "single output" procedure I (P_i) listed in the heading, a value must be assigned within the procedure by an assignment statement "$I := E$", where I is the identifier naming that procedure.

To each procedure listed in the heading, at least one **return** statement must correspond within the procedure. Some of these **return** statements may however be identical for different procedures listed in the heading.

Since a procedure is a self-contained program (except for parameters), the defining rules for statements and declarations within procedures are those already given. A formal input parameter may be

a) a single identifier I (formal variable),
b) an array $I[,, \ldots,]$ with k ($k = 1,\ 2,\ \ldots$) empty subscript positions,
c) a function F $(,, \ldots,)$ with k ($k = 1,\ 2,\ \ldots$) empty parameter positions,
d) a procedure P $(,, \ldots,)$ with k ($k = 1,\ 2,\ \ldots$) empty parameter positions,
e) an identifier occurring in a procedure which is added as a primitive to the language.

A formal output parameter may be
a) a single identifier (formal variable),
b) an array with k ($k = 1,\ 2,\ \ldots$) empty subscript positions.

A formal (exit) label may only be a label.

A label is an admissible formal exit label if, within the procedure, it appears in *go to* statements or *switch* declarations.

An array declaration contained in the heading of the procedure declaration, and referring to a formal parameter, may contain expressions in its lists defining subscript ranges. These expressions may contain
a) numbers,
b) formal input variables, arrays, and functions.

All identifiers and all labels contained in the procedure have identity only within the procedure, and have no relationship to identical identifiers or labels outside the procedure, with the exception of the labels identical to the different procedure names contained in the heading.

A procedure declaration, once made, is permanent, and the only identifiable constituents of the declaration are the procedure declaration heading, and the entrance labels. All rules of operations and declarations contained within the procedure may be considered to be in a language different from the algorithmic language. For this reason, a procedure may even initially be composed of statements given in a language other than the algorithmic language, *e. g.*, a machine language may be required for expressing input-output procedures.

A tagging system may be required to identify the language form in which procedures are expressed. The specific nature of such a system is not in the scope of this report.

Thus by using procedure declarations, new primitive elements may be added to the algorithmic language at will.

Part III

a) Basic symbols (α)

Delimiters δ:	
Operators	$\omega \sim + \;\; - \;\; \times \;\; / \;\; \neg \;\; \vee \;\; \wedge \;\; \equiv \;\; = \;\; \neq \;\; > \;\; \geqq \;\; \leqq \;\; <$
	$\sim$ **go to** **do** **return** **stop** **for** **if** **or if** **if either**
Separators	$\sigma \sim$, ; . : := =: $\rightarrow$ $_{10}$
Brackets	$\beta \sim$ ( ) [ ] $\uparrow$ $\downarrow$ **begin** **end**
Declarators	$\varphi \sim$ **procedure** **switch** **array** **type**[12] **comment**
Non-delimiters μ:	
Letters	$\lambda \sim A$ through Z $\sim a$ through z
Digits	$\zeta \sim 0$ through 9

b) Syntactic skeleton

Syllables:	
List	$\ell \sim E, E, \ldots, E$
Simple variable	$V \sim I$
Subscripted variable	$V \sim I[E, E, \ldots, E]$
Function	$F \sim I\ (R)$ where $R \sim P, P, \ldots, P$
Expression	E (see the appropriate sections in Part II for the com-
Boolean expression	B position rules)
Statement label	$L \sim I$
	$\sim G$
Designational expression	$D \sim L$
	$\sim I[E]$
Parameters	P (see the appropriate sections in Part II for the composition rules)
Identifier	$I \sim \lambda\mu\mu\mu\ldots\mu$
Integer	$G \sim \zeta\zeta\zeta\ldots\zeta$
Number	$N \sim G\underset{\uparrow}{.}\overbrace{G}^{\text{may be empty}}\underset{\uparrow}{{}_{10}}\pm G$ may be empty
String of symbols	$S_\delta \sim \alpha\alpha\alpha\ldots\alpha$ where α is not the particular delimiter δ given in the subscript
Statements Σ:	
Assignment statement	$\Sigma \sim V := E$
	$\sim V := B$
Compound statement	$\Sigma \sim$ **begin** $\underbrace{\Sigma; \Sigma; \ldots; \Sigma}_{\text{at least one } \Sigma}$ **end**
Labeled statement	$\Sigma \sim L{:}\ \Sigma$ where Σ is unlabeled
Go to statement	$\Sigma \sim$ **go to** D
Do statement	$\Sigma \sim$ **do** $L, \underbrace{L}_{\text{may be empty}}\ \underbrace{(S_\rightarrow \rightarrow I, S_\rightarrow \rightarrow I, \ldots, S_\rightarrow \rightarrow I)}_{\text{may be empty}}$
Quantifier statements	$\Sigma \sim$ **if** B
	$\sim$ **for** $V := \ell$
	$\sim$ **for** $V := E(E)E, E(E)E, \ldots, E(E)E$
Alternative statement	$\Sigma \sim$ **if either** B_1; Σ_1; **or if** B_2; Σ_2; …; **or if** B_k; Σ_k **end**
Stop and return statements	$\Sigma \sim$ **stop** $\sim$ **return**
Procedure statement	$\Sigma \sim I\ (R)$ =: (R) where $R \sim P, P, \ldots, P$

12. Representant

Declarations Δ:

Function declaration	$\Delta \sim I\ (R) := E$ where $R \sim P, P, \ldots, P$
Procedure declaration	$\Delta \sim$ **procedure** $I\ (R) \overbrace{=: (R)}^{\text{may be empty}}, I\ (R) \overbrace{=: (R)}^{\text{may be empty}}, \ldots, I\ (R) \underbrace{=: (R)}_{\text{may be empty}}$
	$\Delta; \Delta; \ldots; \Delta;$ **begin** $\Sigma; \Sigma; \ldots; \Delta; \Delta; \ldots; \Sigma; \Sigma$ **end** where $R \sim P, P, \ldots, P$
Switch declaration	$\Delta \sim$ **switch** $:= (D, D, \ldots, D)$
Array declaration	$\Delta \sim$ **array** $(I, I, \ldots, I[\ell{:}\ell], I, I, \ldots, I[\ell{:}\ell], \ldots)$
Symbol classification declaration	$\Delta \sim$ **type** $(I, I, \ldots, I)$
Comment declaration	$\Delta \sim$ **comment** $S_;$

c) Publication language

As stated in the introduction, the reference language is a link between hardware languages and handwritten, typed or printed documentation. For transliteration between the reference language and a language suitable for publications (for example, lectures in numerical analysis), the following *transliteration rules* may be used:

Reference Language	*Publication Language*
subscript brackets [ ]	lowering of the line between the brackets
exponentiation parentheses $\uparrow\downarrow$	raising of the line between the arrows
parentheses ( )	any form of parentheses, brackets, braces
basis of ten $_{10}$	raising of the ten and of the following integral number, inserting of the intended multiplication sign
statement separator ;	*line convention*: each statement on a separate line *may be used*

Furthermore, if line convention is used, the following changes may be simultaneously used:

multiplication cross $\times$	multiplication dot .
decimal point .	decimal comma ,
separation mark ,	any common non-ambiguous separation mark

Example

Integration of a function $F(x)$ by Simpson's Rule. The values of $F(x)$ are supplied by an assumed existent function routine. The mesh size is halved until two successive Simpson sums agree to within a prescribed error. During the mesh reduction $F(x)$ is evaluated at most once for any x. A value V greater than the maximum absolute value attained by the function on the interval is required for initializing. (*abs* (absolute value) is the name of a standard procedure always available to the programmer so that it need not be supplied as an input parameter.)

```
procedure Simps (F(), a, b, delta, V)
comment a, b are the min. and max. resp. of the points def. interval of integ. F() is the function
to be integrated. delta is the permissible difference between two successive Simpson sums. V is
greater than the maximum absolute value of F on a, b;
begin
Simps: Ibar := V × (b − a)
       n := 1
       h := (b − a) / 2
       J := h × (F (a) + F (b))
Jl:    S := 0
       for k := 1(1)n
       S := S + F (a + (2 × k − 1) × h)
```

```
        I := J + 4 × h × S
        if (delta < abs (I − Ibar))
        begin
           Ibar := I
           J := (I + J) / 4
           n := 2 × n
           h := h / 2
           go to Jl
        end
        Simps := I / 3
        return
        integer (k, n)
end Simps
```

§ A.2 The 1960 Report & 1962 Revised Report

Report[13] on the Algorithmic Language ALGOL 60

by

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy,
P. Naur (editor), A. J. Perlis, H. Rutishauser, K. Samelson,
B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger

Dedicated to the memory of William Turanski

Introduction[14]

Background. After the publication[15,16] of a preliminary report on the algorithmic language ALGOL, as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.
As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A “hardware group” was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL Bulletin, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959 which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represent the following organizations: Association Française de Calcul, British Computer Society, Gesellschaft für Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

13. [*Revised Report*:] **Revised Report**

14. [In the *Revised Report*, a new section precedes the Introduction:]

Summary

The report gives a complete defining description of the international algorithmic language ALGOL 60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers. The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition the notions reference language, publication language, and hardware representations are explained.
In the first chapter a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.
The second chapter lists all the basic symbols, and the syntactic units known as identifiers, numbers, and strings are defined. Further some important notions such as quantity and value are defined.
The third chapter explains the rules for forming expressions and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical), and designational.
The fourth chapter describes the operational units of the language, known as statements. The basic statements are: assignment statements (evaluation of a formula), go to statements (explicit break of the sequence of execution of statements), dummy statements, and procedure statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: conditional statements, for statements, compound statements, and blocks.
In the fifth chapter the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.
The report ends with two detailed examples of the use of the language and an alphabetic index of definitions.

15. Preliminary Report — International Algebraic Language, Comm. Assoc. Comp. Mach. 1, No. 12 (1958), 8.

16. Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, edited by A. J. Perlis and K. Samelson, Numerische Mathematik Bd. 1, S. 41–60 (1959).

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the ACM Communications where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM Communications. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

January 1960 Conference. The thirteen representatives,[17] from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960.

Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur and the Conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the committee's concepts and the intersection of its agreements.[18]

17. William Turanski of the American group was killed by an automobile just prior to the January 1960 Conference.

18. [In the *Revised Report*, this paragraph is followed by a new section:]

April 1962 Conference (Edited by M. Woodger). A meeting of some of the authors of ALGOL 60 was held on 2nd–3rd April 1962 in Rome, Italy, through the facilities and courtesy of the International Computation Center. The following were present:

Authors	*Advisers*	*Observer*
F. L. Bauer	M. Paul	W. L. van der Poel (Chairman,
J. Green	R. Franciotti	IFIP TC 2.1 Working Group ALGOL)
C. Katz	P. Z. Ingerman	
R. Kogon (representing J. W. Backus)		
P. Naur		
K. Samelson	G. Seegmüller	
J. H. Wegstein	R. E. Utman	
A. van Wijngaarden		
M. Woodger	P. Landin	

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the ALGOL 60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification that were submitted by interested parties in response to the Questionnaire in ALGOL Bulletin No. 14 were used as a guide.

This report* constitutes a supplement to the ALGOL 60 Report which should resolve a number of difficulties therein. Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced programming languages will lead to better resolution:

1. Side effects of functions.
2. The call by name concept.
3. **own**: static or dynamic.
4. For statement: static or dynamic.
5. Conflict between specification and declaration.

The authors of the ALGOL 60 Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification, and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962 and has been approved by the Council of the International Federation for Information Processing.

* The present edition follows the text which was approved by the Council of IFIP. Although it is not clear from the Introduction, the present version is the original report of the January 1960 conference modified according to the agreements reached during the April 1962 conference. Thus the report mentioned here is incorporated in the present version. The modifications touch the original report in the following sections: Changes of text: 1 with footnote; 2.1 footnote; 2.3; 2.7; 3.3.3; 3.3.4.2; 4.1.3; 4.2.3; 4.2.4; 4.3.4; 4.7.3; 4.7.3.1; 4.7.3.3; 4.7.5.1; 4.7.5.4; 4.7.6; 5; 5.3.3; 5.3.5; 5.4.3; 5.4.4; 5.4.5. Changes of syntax: 3.4.1; 4.1.1; 4.2.1; 4.5.1.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

Reference Language

1. It is the working language of the committee.
2. It is the defining language.
3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
7. The main publications of the ALGOL language itself will use the reference representation.

Publication Language

1. The publication language admits variations of the reference language according to usage of printing and handwriting (*e. g.*, subscripts, spaces, exponents, Greek letters).
2. It is used for stating and communicating processes.
3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

Hardware Representations

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from publication or reference language.

For transliteration between the reference language and a language suitable for publications, among others, the following rules are recommended.

Reference Language	*Publication Language*
Subscript brackets [ ]	Lowering of the line between the brackets and removal of the brackets.
Exponentiation $\uparrow$	Raising of the exponent.
Parentheses ( )	Any form of parentheses, brackets, braces.
Basis of ten $_{10}$	Raising of the ten and of the following integral number, inserting of the intended multiplication sign.

Description of the reference language

Was sich überhaupt sagen laßt, laßt sich klar sagen;
und wovon man nicht reden kann, darüber muß man schweigen.
Ludwig Wittgenstein

1. Structure of the language

As stated in the introduction, the algorithmic language has three different kinds of representations —reference, hardware, and publication— and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols — and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language — explicit formulæ— called assignment statements.
To show the flow of computational processes, certain non-arithmetic statements and statement clauses are added which may describe *e. g.*, alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refers to another, statements may be provided with labels. Sequences of statements may be combined into compound statements by insertion of statement brackets.[19]
Statements are supported by declarations which are not themselves computing instructions, but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers or even the set of rules defining a function. Each declaration is attached to and valid for one compound statement. A compound statement which includes declarations is called a block.[20]
A program is a self-contained compound statement, *i. e.* a compound statement which is not contained within another compound statement and which makes no use of other compound statements not contained within it.[21]
In the sequel the syntax and semantics of the language will be given.[22]

1.1. Formalism for syntactic description

The syntax will be described with the aid of metalinguistic formulæ.[23] Their interpretation is best explained by an example:
$\langle ab \rangle$ ::= (| [| $\langle ab \rangle$(| $\langle ab \rangle$ $\langle d \rangle$
Sequences of characters enclosed in the bracket $\langle\rangle$ represent metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter with the meaning of **or**) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable $\langle ab \rangle$. It indicates that $\langle ab \rangle$ may have the value (or [or that given some legitimate value of $\langle ab \rangle$, another may be formed by following it with the character (or by following it with some value of the variable $\langle d \rangle$. If the values of $\langle d \rangle$ are the decimal digits, some values of $\langle ab \rangle$ are:
[(((*1*(*37*(
(*12345*(
(((
[*86*

19. [In the *Revised Report*, this sentence becomes:] A sequence of statements may be enclosed between the statement brackets **begin** and **end** to form a compound statement.

20. [In the *Revised Report*, these two last sentences become:] A sequence of declarations followed by a sequence of statements and enclosed between **begin** and **end** constitutes a block. Every declaration appears in a block in this way and is valid only for that block.

21. [In the *Revised Report*, this sentence becomes:] A program is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it.

22. Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is [*Revised Report*: left undefined or] said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

23. *Cf.* J. W. Backus, The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. ICIP Paris, June 1959.

In order to facilitate the study the symbols used for distinguishing the metalinguistic variables (*i. e.* the sequences of characters appearing within the brackets ⟨⟩ as *ab* in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulæ have been given in more than one place.
Definition:
⟨*empty*⟩ ::=
(*i. e.* the null string of symbols).

2. Basic symbols, identifiers, numbers, and strings. Basic concepts

The reference language is built up from the following basic symbols:
⟨*basic symbol*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩ | ⟨*logical value*⟩ | ⟨*delimiter*⟩

2.1. Letters

⟨*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*

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (*i. e.* character not coinciding with any digit, logical value or delimiter).
Letters do not have individual meaning. They are used for forming identifiers and strings[24] (*cf.* sections 2.4. Identifiers, 2.6. Strings).

2.2.1. Digits

⟨*digit*⟩ ::= *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*
Digits are used for forming numbers, identifiers, and strings.

2.2.2. Logical values

⟨*logical value*⟩ ::= **true** | **false**
The logical values have a fixed obvious meaning.

2.3. Delimiters

⟨*delimiter*⟩ ::= ⟨*operator*⟩ | ⟨*separator*⟩ | ⟨*bracket*⟩ | ⟨*declarator*⟩ | ⟨*specificator*⟩
⟨*operator*⟩ ::= ⟨*arithmetic operator*⟩ | ⟨*relational operator*⟩ | ⟨*logical operator*⟩
| ⟨*sequential operator*⟩
⟨*arithmetic operator*⟩ ::= + | − | × | / | ÷ | ↑
⟨*relational operator*⟩ ::= < | ≦ | = | ≧ | > | ≠
⟨*logical operator*⟩ ::= ≡ | ⊃ | ∨ | ∧ | ¬
⟨*sequential operator*⟩ ::= **go to** | **if** | **then** | **else** | **for** | **do**[25]
⟨*separator*⟩ ::= , | . | $_{10}$ | : | ; | := | ␣ | **step** | **until** | **while** | **comment**
⟨*bracket*⟩ ::= (|) | [|] | ‘ | ’ | **begin** | **end**
⟨*declarator*⟩ ::= **own** | **Boolean** | **integer** | **real** | **array** | **switch** | **procedure**
⟨*specificator*⟩ ::= **string** | **label** | **value**

Delimiters have a fixed meaning which for the most part is obvious, or else will be given at the appropriate place in the sequel.
Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

24. It should be particularly noted that throughout the reference language underlining [boldface] is used for defining independent basic symbols (see sections 2.2.2 and 2.3). These are understood to have no relation to the individual letters of which they are composed. [Added in the *Revised Report*:] Within the present report [not including headings and section numbers] underlining [boldface] will be used for no other purposes.

25. **do** is used in for statements. It has no relation whatsoever to the *do* of the preliminary report, which is not included in ALGOL 60.

For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols:	is equivalent with[26]
; **comment** ⟨*any sequence not containing* ;⟩ ;	;
begin comment ⟨*any sequence not containing* ;⟩ ;	**begin**
end ⟨*any sequence not containing* **end** *or* ; *or* **else**⟩	**end**

By equivalence is here meant that any of the three symbols shown in the right hand column may, in any occurrence outside of strings, be replaced by any sequence of symbols of the structure shown in the same line of the left hand column without any effect on the action of the program.[27]

2.4. Identifiers

2.4.1. Syntax

⟨*identifier*⟩ ::= ⟨*letter*⟩ | ⟨*identifier*⟩ ⟨*letter*⟩ | ⟨*identifier*⟩ ⟨*digit*⟩

2.4.2. Examples

q
Soup
V17a
a34kTMNs
MARILYN

2.4.3. Semantics. Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (*cf.* however section 3.2.4. Standard functions).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (*cf.* section 2.7. Quantities, kinds and scopes and section 5. Declarations).

2.5. Numbers

2.5.1. Syntax

⟨*unsigned integer*⟩ ::= ⟨*digit*⟩ | ⟨*unsigned integer*⟩ ⟨*digit*⟩
⟨*integer*⟩ ::= ⟨*unsigned integer*⟩ | + ⟨*unsigned integer*⟩ | − ⟨*unsigned integer*⟩
⟨*decimal fraction*⟩ ::= . ⟨*unsigned integer*⟩
⟨*exponent part*⟩ ::= $_{10}$ ⟨*integer*⟩
⟨*decimal number*⟩ ::= ⟨*unsigned integer*⟩ | ⟨*decimal fraction*⟩
    | ⟨*unsigned integer*⟩ ⟨*decimal fraction*⟩
⟨*unsigned number*⟩ ::= ⟨*decimal number*⟩ | ⟨*exponent part*⟩
    | ⟨*decimal number*⟩ ⟨*exponent part*⟩
⟨*number*⟩ ::= ⟨*unsigned number*⟩ | + ⟨*unsigned number*⟩ | − ⟨*unsigned number*⟩

2.5.2. Examples

0	*−200.084*	*−.083*$_{10}$*−02*
177	*+07.43*$_{10}$*8*	*−*$_{10}$*7*
.5384	*9.34*$_{10}$*+10*	$_{10}$*−4*
+0.7300	*2*$_{10}$*−4*	*+*$_{10}$*+5*

2.5.3. Semantics. Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4. Types. Integers are of type **integer**. All other numbers are of type **real** (*cf.* section 5.1. Type declarations).

26. [*Revised Report*:] is equivalent to

27. [In the *Revised Report*, this sentence becomes:] By equivalence is here meant that any of the three structures shown in the left hand column may be replaced, in any occurrence outside of strings, by the symbol shown on the same line in the right hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.6. Strings

2.6.1. Syntax

⟨*proper string*⟩ ::= ⟨*any sequence of basic symbols not containing* ‘ *or* ’⟩ | ⟨*empty*⟩
⟨*open string*⟩ ::= ⟨*proper string*⟩ | ‘ ⟨*open string*⟩ ’ | ⟨*open string*⟩ ⟨*open string*⟩
⟨*string*⟩ ::= ‘ ⟨*open string*⟩ ’

2.6.2. Examples

‘*5k*,,−‘[[[‘∧=/:’*Tt*’’
‘..*This*␣*is*␣*a*␣‘*string*’’

2.6.3. Semantics. In order to enable the language to handle arbitrary sequences of basic symbols the string quotes ‘ and ’ are introduced. The symbol ␣ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (*cf.* sections 3.2. Function designators and 4.7. Procedure statements).

2.7. Quantities, kinds and scopes

The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures.

The scope of a quantity is the set of statements in which the declaration for the identifier associated with that quantity is valid, or, for labels, the set of statements which may have the statement in which the label occurs as their successor.[28]

2.8. Values and types

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (*cf.* section 3.1.4.1).

The various “types” (**integer**, **real**, **Boolean**) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

⟨*expression*⟩ ::= ⟨*arithmetic expression*⟩ | ⟨*Boolean expression*⟩ | ⟨*designational expression*⟩

3.1. Variables

3.1.1. Syntax

⟨*variable identifier*⟩ ::= ⟨*identifier*⟩
⟨*simple variable*⟩ ::= ⟨*variable identifier*⟩
⟨*subscript expression*⟩ ::= ⟨*arithmetic expression*⟩
⟨*subscript list*⟩ ::= ⟨*subscript expression*⟩ | ⟨*subscript list*⟩ , ⟨*subscript expression*⟩
⟨*array identifier*⟩ ::= ⟨*identifier*⟩
⟨*subscripted variable*⟩ ::= ⟨*array identifier*⟩ [⟨*subscript list*⟩]
⟨*variable*⟩ ::= ⟨*simple variable*⟩ | ⟨*subscripted variable*⟩

28. [In the *Revised Report*, this sentence becomes:] The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

3.1.2. Examples
epsilon
detA
a17
Q[*7*, *2*]
x[*sin* (*n* × *pi*/*2*), *Q*[*3*, *n*, *4*]]

3.1.3. Semantics. A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (*cf.* section 5.1. Type declarations) or for the corresponding array identifier (*cf.* section 5.2. Array declarations).

3.1.4. Subscripts

3.1.4.1. Subscripted variables designate values which are components of multidimensional arrays (*cf.* section 5.2. Array declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (*cf.* section 3.3. Arithmetic expressions).

3.1.4.2. Each subscript position acts like a variable of type **integer** and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (*cf.* section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (*cf.* section 5.2. Array declarations).

3.2. Function designators

3.2.1. Syntax
⟨*procedure identifier*⟩ ::= ⟨*identifier*⟩
⟨*actual parameter*⟩ ::= ⟨*string*⟩ | ⟨*expression*⟩ | ⟨*array identifier*⟩ | ⟨*switch identifier*⟩
  | ⟨*procedure identifier*⟩
⟨*letter string*⟩ ::= ⟨*letter*⟩ | ⟨*letter string*⟩ ⟨*letter*⟩
⟨*parameter delimiter*⟩ ::= , |) ⟨*letter string*⟩ : (
⟨*actual parameter list*⟩ ::= ⟨*actual parameter*⟩
  | ⟨*actual parameter list*⟩ ⟨*parameter delimiter*⟩ ⟨*actual parameter*⟩
⟨*actual parameter part*⟩ ::= ⟨*empty*⟩ | (⟨*actual parameter list*⟩)
⟨*function designator*⟩ ::= ⟨*procedure identifier*⟩ ⟨*actual parameter part*⟩

3.2.2. Examples
sin (*a* − *b*)
J (*v* + *s*, *n*)
R
S (*s* − *5*) *Temperature*: (*T*) *Pressure*: (*P*)
Compile (':=') *Stack*: (*Q*)

3.2.3. Semantics. Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (*cf.* section 5.4. Procedure declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7. Procedure statements. Not every procedure declaration defines the value of a function designator.

3.2.4. Standard functions. Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. It is recommended that this reserved list should contain:

abs (E)	for the modulus (absolute value) of the value of the expression E
sign (E)	for the sign of the value of E ($+1$ for E > 0, 0 for E $= 0$, -1 for E < 0)
sqrt (E)	for the square root of the value of E
sin (E)	for the sine of the value of E
cos (E)	for the cosine of the value of E
arctan (E)	for the principal value of the arctangent of the value of E
ln (E)	for the natural logarithm of the value of E
exp (E)	for the exponential function of the value of E (e^{E})

These functions are all understood to operate indifferently on arguments both of type **real** and **integer**. They will all yield values of type real, except for *sign* (E) which will have values of type **integer**. In a particular representation these functions may be available without explicit declarations (*cf.* section 5. Declarations).

3.2.5. Transfer functions. It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely

entier (E),

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E.

3.3. Arithmetic expressions

3.3.1. Syntax

$\langle adding\ operator \rangle ::= + \mid -$

$\langle multiplying\ operator \rangle ::= \times \mid / \mid \div$

$\langle primary \rangle ::= \langle unsigned\ number \rangle \mid \langle variable \rangle \mid \langle function\ designator \rangle$
$\quad \mid (\langle arithmetic\ expression \rangle)$

$\langle factor \rangle ::= \langle primary \rangle \mid \langle factor \rangle \uparrow \langle primary \rangle$

$\langle term \rangle ::= \langle factor \rangle \mid \langle term \rangle \langle multiplying\ operator \rangle \langle factor \rangle$

$\langle simple\ arithmetic\ expression \rangle ::= \langle term \rangle \mid \langle adding\ operator \rangle \langle term \rangle$
$\quad \mid \langle simple\ arithmetic\ expression \rangle \langle adding\ operator \rangle \langle term \rangle$

$\langle if\ clause \rangle ::=$ **if** $\langle Boolean\ expression \rangle$ **then**

$\langle arithmetic\ expression \rangle ::= \langle simple\ arithmetic\ expression \rangle$
$\quad \mid \langle if\ clause \rangle \langle simple\ arithmetic\ expression \rangle$ **else** $\langle arithmetic\ expression \rangle$

3.3.2. Examples

Primaries:

$7.394_{10}{-8}$

sum

$w[i + 2, 8]$

$cos\ (y + z \times 3)$

$(a - 3/y + vu \uparrow 8)$

Factors:

$omega$

$sum \uparrow cos\ (y + z \times 3)$

$7.394_{10}{-8} \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu\uparrow 8)$

Terms :

U

$omega \times sum \uparrow cos\ (y + z \times 3)/7.394_{10}{-8} \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu\uparrow 8)$

Simple arithmetic expression:

$U - Yu + omega \times sum \uparrow cos\ (y + z \times 3)/7.394_{10}{-8} \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu\uparrow 8)$

Arithmetic expressions:

$w \times u - Q\ (S + Cu) \uparrow 2$

if $q > 0$ **then** $S + 3 \times Q/A$ **else** $2 \times S + 3 \times q$
if $a < 0$ **then** $U + V$ **else if** $a \times b > 17$ **then** U/V **else if** $k \neq y$ **then** V/U **else** 0
$a \times sin\ (omega \times t)$
$0.57_{10}12 \times a[N \times (N - 1)/2, 0]$
$(A \times arctan\ (y) + Z) \uparrow (7 + Q)$
if q **then** $n - 1$ **else** n
if $a < 0$ **then** A/B **else if** $b = 0$ **then** B/A **else** 2

3.3.3. Semantics. An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (*cf.* section 5.4. Procedure declarations)[29] when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (*cf.* section 3.4. Boolean expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value **true** is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

else ⟨*simple arithmetic expression*⟩

is equivalent to the construction:

else if true then ⟨*simple arithmetic expression*⟩

3.3.4. Operators and types. Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types **real** or **integer** (*cf.* section 5.1. Type declarations). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1. The operators $+$, $-$, and $\times$ have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be **integer** if both of the operands are of **integer** type, otherwise **real**.

3.3.4.2. The operations ⟨*term*⟩/⟨*factor*⟩ and ⟨*term*⟩÷⟨*factor*⟩ both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (*cf.* section 3.3.5). Thus for example

$a/b \times 7/(p - q) \times v/s$

means

$((((a \times (b^{-1})) \times 7) \times ((p - q)^{-1})) \times v) \times (s^{-1})$

The operator / is defined for all four combinations of types **real** and **integer** and will yield results of **real** type in any case. The operator ÷ is defined only for two operands both of type **integer** and will yield a result of type **integer** defined as follows:[30]

$a \div b = sign\ (a/b) \times entier\ (abs\ (a/b))$

(*cf.* sections 3.2.4 and 3.2.5).

3.3.4.3. The operation ⟨*factor*⟩↑⟨*primary*⟩ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus for example

29. [*Revised Report*:] (*cf.* section 5.4.4. Values of function designators)

30. [*Revised Report*:] of type **integer**, mathematically defined as follows:

$2 \uparrow n \uparrow k$ means $(2^n)^k$
while
$2 \uparrow (n \uparrow m)$ means $2^{(n^m)}$
Writing i for a number of **integer** type, r for a number of **real** type, and a for a number of either **integer** or **real** type, the result is given by the following rules:

$a \uparrow i$	If $i > 0$:		$a \times a \times \ldots \times a$ (i times), of the same type as a
	If $i = 0$,	if $a \neq 0$:	1, of the same type as a
		if $a = 0$:	undefined
	If $i < 0$,	if $a \neq 0$:	$1/(a \times a \times \ldots \times a)$ (the denominator has i factors),[31] of type **real**
		if $a = 0$:	undefined
$a \uparrow r$	If $a > 0$:		$exp\ (r \times ln\ (a))$, of type **real**
	If $a = 0$,	if $r > 0$:	0.0, of type **real**
		if $r \leqq 0$:	undefined
	If $a < 0$:		always undefined

3.3.5. Precedence of operators. The sequence of operations within one expression is generally from left to right, with the following additional rules:
3.3.5.1. According to the syntax given in section 3.3.1 the following rules of precedence hold:
first: $\uparrow$
second: $\times$ / $\div$
third: $+$ $-$
3.3.5.2. The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.
3.3.6. Arithmetics of **real** quantities. Numbers and variables of type **real** must be interpreted in the sense of numerical analysis, *i. e.* as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4. Boolean expressions

3.4.1. Syntax
$\langle$*relational operator*$\rangle$ ::= $<$ | $\leqq$ | $=$ | $\geqq$ | $>$ | $\neq$
$\langle$*relation*$\rangle$ ::= $\langle$*arithmetic expression*$\rangle$ $\langle$*relational operator*$\rangle$ $\langle$*arithmetic expression*$\rangle$[32]
$\langle$*Boolean primary*$\rangle$ ::= $\langle$*logical value*$\rangle$ | $\langle$*variable*$\rangle$ | $\langle$*function designator*$\rangle$
  | $\langle$*relation*$\rangle$ | ($\langle$*Boolean expression*$\rangle$)
$\langle$*Boolean secondary*$\rangle$ ::= $\langle$*Boolean primary*$\rangle$ | $\neg$ $\langle$*Boolean primary*$\rangle$
$\langle$*Boolean factor*$\rangle$ ::= $\langle$*Boolean secondary*$\rangle$ | $\langle$*Boolean factor*$\rangle$ $\wedge$ $\langle$*Boolean secondary*$\rangle$
$\langle$*Boolean term*$\rangle$ ::= $\langle$*Boolean factor*$\rangle$ | $\langle$*Boolean term*$\rangle$ $\vee$ $\langle$*Boolean factor*$\rangle$
$\langle$*implication*$\rangle$ ::= $\langle$*Boolean term*$\rangle$ | $\langle$*implication*$\rangle$ $\supset$ $\langle$*Boolean term*$\rangle$
$\langle$*simple Boolean*$\rangle$ ::= $\langle$*implication*$\rangle$ | $\langle$*simple Boolean*$\rangle$ $\equiv$ $\langle$*implication*$\rangle$
$\langle$*Boolean expression*$\rangle$ ::= $\langle$*simple Boolean*$\rangle$
  | $\langle$*if clause*$\rangle$ $\langle$*simple Boolean*$\rangle$ **else** $\langle$*Boolean expression*$\rangle$

31. [*Revised Report*:] (the denominator has $-i$ factors),

32. [*Revised Report*:] $\langle$*relation*$\rangle$ ::= $\langle$*simple arithmetic expression*$\rangle$ $\langle$*relational operator*$\rangle$ $\langle$*simple arithmetic expression*$\rangle$

3.4.2. Examples

$x = -2$

$Y > V \vee z < q$

$a + b > -5 \wedge z - d > q \uparrow 2$

$p \wedge q \vee x \neq y$

$g \equiv \neg a \wedge b \wedge \neg c \vee d \vee e \supset \neg f$

if $k < 1$ **then** $s > w$ **else** $h \leqq c$

if if if a **then** b **else** c **then** d **else** f **then** g **else** $h < k$

3.4.3. Semantics. A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4. Types. Variables and function designators entered as Boolean primaries must be declared **Boolean** (*cf.* section 5.1. Type declarations and section 5.4.4. Values of function designators).

3.4.5. The operators. Relations take on the value **true** whenever the corresponding relation is satisfied for the expressions involved, otherwise **false**.

The meaning of the logical operators ¬ (not), ∧ (and), ∨ (or), ⊃ (implies), and ≡ (equivalent), is given by the following function table.

$b1$	**false**	**false**	**true**	**true**
$b2$	**false**	**true**	**false**	**true**
$\neg b1$	**true**	**true**	**false**	**false**
$b1 \wedge b2$	**false**	**false**	**false**	**true**
$b1 \vee b2$	**false**	**true**	**true**	**true**
$b1 \supset b2$	**true**	**true**	**false**	**true**
$b1 \equiv b2$	**true**	**false**	**false**	**true**

3.4.6. Precedence of operators. The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1. According to the syntax given in section 3.4.1 the following rules of precedence hold:

first: arithmetic expressions according to section 3.3.5.

second: $< \leqq = \geqq > \neq$

third: $\neg$

fourth: $\wedge$

fifth: $\vee$

sixth: $\supset$

seventh: $\equiv$

3.4.6.2. The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.5. Designational expressions

3.5.1. Syntax

⟨*label*⟩ ::= ⟨*identifier*⟩ | ⟨*unsigned integer*⟩

⟨*switch identifier*⟩ ::= ⟨*identifier*⟩

⟨*switch designator*⟩ ::= ⟨*switch identifier*⟩ [⟨*subscript expression*⟩]

⟨*simple designational expression*⟩ ::= ⟨*label*⟩ | ⟨*switch designator*⟩
 | (⟨*designational expression*⟩)

⟨*designational expression*⟩ ::= ⟨*simple designational expression*⟩
 | ⟨*if clause*⟩ ⟨*simple designational expression*⟩ **else** ⟨*designational expression*⟩

3.5.2. Examples

17

p9

Choose[$n - 1$]

Town[**if** $y < 0$ **then** N **else** $N + 1$]

if $Ab < c$ **then** *17* **else** q[**if** $w \leqq 0$ **then** *2* **else** n]

3.5.3. Semantics. A designational expression is a rule for obtaining a label of a statement (*cf.* section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (*cf.* section 5.3. Switch declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.
3.5.4. The subscript expression. The evaluation of the subscript expression is analogous to that of subscripted variables (*cf.* section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values *1*, *2*, *3*, . . ., n, where n is the number of entries in the switch list.
3.5.5. Unsigned integers as labels. Unsigned integers used as labels have the property that leading zeroes do not affect their meaning, *e. g. 00217* denotes the same label as *217*.

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.
In order to make it possible to define a specific dynamic succession, statements may be provided with labels.
Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1. Compound statements and blocks

4.1.1. Syntax
⟨*unlabeled basic statement*⟩ ::= ⟨*assignment statement*⟩ | ⟨*go to statement*⟩
  | ⟨*dummy statement*⟩ | ⟨*procedure statement*⟩
⟨*basic statement*⟩ ::= ⟨*unlabeled basic statement*⟩ | ⟨*label*⟩ : ⟨*basic statement*⟩
⟨*unconditional statement*⟩ ::= ⟨*basic statement*⟩ | ⟨*for statement*⟩
  | ⟨*compound statement*⟩ | ⟨*block*⟩[33]
⟨*statement*⟩ ::= ⟨*unconditional statement*⟩ | ⟨*conditional statement*⟩[34]
⟨*compound tail*⟩ ::= ⟨*statement*⟩ **end** | ⟨*statement*⟩ ; ⟨*compound tail*⟩
⟨*block head*⟩ ::= **begin** ⟨*declaration*⟩ | ⟨*block head*⟩ ; ⟨*declaration*⟩
⟨*unlabeled compound*⟩ ::= **begin** ⟨*compound tail*⟩
⟨*unlabeled block*⟩ ::= ⟨*block head*⟩ ; ⟨*compound tail*⟩
⟨*compound statement*⟩ ::= ⟨*unlabeled compound*⟩ | ⟨*label*⟩ : ⟨*compound statement*⟩
⟨*block*⟩ ::= ⟨*unlabeled block*⟩ | ⟨*label*⟩ : ⟨*block*⟩[35]
This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:
Compound statement:
L: L: . . . **begin** S; S; . . . S; S **end**
Block:
L: L: . . . **begin** D; D; . . . D; S; S; . . . S; S **end**

33. [*Revised Report*:] ⟨*unconditional statement*⟩ ::= ⟨*basic statement*⟩ | ⟨*compound statement*⟩ | ⟨*block*⟩

34. [*Revised Report*:] ⟨*statement*⟩ ::= ⟨*unconditional statement*⟩ | ⟨*conditional statement*⟩ | ⟨*for statement*⟩

35. [Added in the *Revised Report*, after the definition of ⟨*block*⟩:] ⟨*program*⟩ ::= ⟨*block*⟩ | ⟨*compound statement*⟩

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2. Examples

Basic statements:

```
a := p + q
go to Naples
Start: Continue: W := 7.993
```

Compound statement:

```
begin x := 0; for y := 1 step 1 until n do x := x + A[y];
        if x > q then go to STOP else if x > w − 2 then go to S;
        Aw: St: W := x + bob end
```

Block:

```
Q: begin integer i, k; real w;
    for i := 1 step 1 until m do
       for k := i + 1 step 1 until m do
       begin w := A[i,k];
              A[i,k] := A[k,i];
              A[k,i] := w end for i and k
    end block Q
```

4.1.3. Semantics. Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (*cf.* section 5. Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be non-local to it, *i. e.* will represent the same entity inside the block and in the level immediately outside it. The exception to this rule is presented by labels, which are local to the block in which they occur.[36]

Since a statement of a block may again itself be a block the concepts local and non-local to a block must be understood recursively. Thus an identifier, which is non-local to a block A, may or may not be non-local to the block B in which A is one statement.

4.2. Assignment statements

4.2.1. Syntax

⟨*left part*⟩ ::= ⟨*variable*⟩ :=[37]

⟨*left part list*⟩ ::= ⟨*left part*⟩ | ⟨*left part list*⟩ ⟨*left part*⟩

⟨*assignment statement*⟩ ::= ⟨*left part list*⟩ ⟨*arithmetic expression*⟩
    | ⟨*left part list*⟩ ⟨*Boolean expression*⟩

4.2.2. Examples

$s := p[0] := n := n + 1 + s$

$n := n + 1$

$A := B/C - v - q \times S$

$s[v, k + 2] := 3 - \mathit{arctan}\ (s \times \mathit{zeta})$

$V := Q > Y \wedge Z$

36. [In the *Revised Report*, this sentence becomes:] A label separated by a colon from a statement, *i. e.* labeling that statement, behaves as though declared in the head of the smallest embracing block, *i. e.* the smallest block whose brackets **begin** and **end** enclose that statement. In this context a procedure body must be considered as if it were enclosed by **begin** and **end** and treated as a block.

37. [*Revised Report*:] ⟨*left part*⟩ ::= ⟨*variable*⟩ := | ⟨*procedure identifier*⟩ :=

4.2.3. Semantics. Assignment statements serve for assigning the value of an expression to one or several variables.[38] The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4. Types. All variables of a left part list must be of the same declared type. If the variables are **Boolean** the expression must likewise be Boolean. If the variables are of type **real** or **integer** the expression must be arithmetic. If the type of the arithmetic expression differs from that of the variables, appropriate transfer functions are understood to be automatically invoked. For transfer from **real** to **integer** type the transfer function is understood to yield a result equivalent to

entier (E + *0.5*)

where E is the value of the expression.[39]

4.3. Go to statements

4.3.1. Syntax

⟨*go to statement*⟩ ::= **go to** ⟨*designational expression*⟩

4.3.2. Examples

go to *8*

go to *exit*[*n* + *1*]

go to *Town*[**if** *y* < *0* **then** *N* **else** *N* + *1*]

go to if *Ab* < *c* **then** *17* **else** *q*[**if** *w* < *0* **then** *2* **else** *n*]

4.3.3. Semantics. A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4. Restriction. Since labels are inherently local, no go to statements can lead from outside into a block.[40]

4.3.5. Go to an undefined switch designator. A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4. Dummy statements

4.4.1. Syntax

⟨*dummy statement*⟩ ::= ⟨*empty*⟩

4.4.2. Examples

L:

begin . . .; *John*: **end**

4.4.3. Semantics. A dummy statement executes no operation. It may serve to place a label.

38. [In the *Revised Report*, this sentence becomes:] Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (*cf.* section 5.4.4).

39. [In the *Revised Report*, this section becomes:] Types. The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is **Boolean**, the expression must likewise be **Boolean**. If the type is **real** or **integer**, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from **real** to **integer** type the transfer function is understood to yield a result equivalent to *entier* (E + *0.5*) where E is the value of the expression. The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (*cf.* section 5.4.4).

40. [Added in the *Revised Report*:] A go to statement may, however, lead from outside into a compound statement.

4.5. Conditional statements

4.5.1. Syntax

$\langle$*if clause*$\rangle$ ::= **if** $\langle$*Boolean expression*$\rangle$ **then**
$\langle$*unconditional statement*$\rangle$ ::= $\langle$*basic statement*$\rangle$ | $\langle$*for statement*$\rangle$
  | $\langle$*compound statement*$\rangle$ | $\langle$*block*$\rangle$[41]
$\langle$*if statement*$\rangle$ ::= $\langle$*if clause*$\rangle$ $\langle$*unconditional statement*$\rangle$ | $\langle$*label*$\rangle$: $\langle$*if statement*$\rangle$[42]
$\langle$*conditional statement*$\rangle$ ::= $\langle$*if statement*$\rangle$ | $\langle$*if statement*$\rangle$ **else** $\langle$*statement*$\rangle$[43]

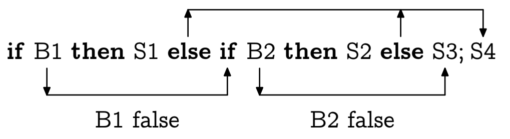
4.5.2. Examples

if $x > 0$ **then** $n := n + 1$
if $v > u$ **then** V: $q := n + m$ **else go to** R
if $s < 0 \vee P \leqq Q$ **then** AA: **begin if** $q < v$ **then** $a := v/s$
  else $y := 2 \times a$ **end else if** $v > s$ **then** $a := v - q$
  else if $v > s - 1$ **then go to** S

4.5.3. Semantics. Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1. If statement. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:
if B1 **then** S1 **else if** B2 **then** S2 **else** S3; S4
and
if B1 **then** S1 **else if** B2 **then** S2 **else if** B3 **then** S3; S4
Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.
The execution of a conditional statement may be described as follows: The Boolean expressions of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value **true** is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, *i. e.* the statement following the complete conditional statement. Thus the effect of the delimiter else may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.
The construction
else $\langle$*unconditional statement*$\rangle$
is equivalent to
else if true then $\langle$*unconditional statement*$\rangle$
If none of the Boolean expressions of the if clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.
For further explanation the following picture may be useful:



4.5.4. Go to into a conditional statement. The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of **else**.

41. [*Revised Report*:] $\langle$*unconditional statement*$\rangle$ ::= $\langle$*basic statement*$\rangle$ | $\langle$*compound statement*$\rangle$ | $\langle$*block*$\rangle$

42. [*Revised Report*:] $\langle$*if statement*$\rangle$ ::= $\langle$*if clause*$\rangle$ $\langle$*unconditional statement*$\rangle$

43. [*Revised Report*:] $\langle$*conditional statement*$\rangle$ ::= $\langle$*if statement*$\rangle$ | $\langle$*if statement*$\rangle$ **else** $\langle$*statement*$\rangle$ | $\langle$*if clause*$\rangle$ $\langle$*for statement*$\rangle$ | $\langle$*label*$\rangle$: $\langle$*conditional statement*$\rangle$

4.6. For statements

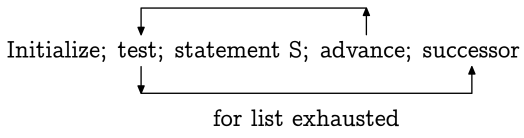
4.6.1. Syntax

⟨*for list element*⟩ ::= ⟨*arithmetic expression*⟩
  | ⟨*arithmetic expression*⟩ **step** ⟨*arithmetic expression*⟩ **until** ⟨*arithmetic expression*⟩
  | ⟨*arithmetic expression*⟩ **while** ⟨*Boolean expression*⟩
⟨*for list*⟩ ::= ⟨*for list element*⟩ | ⟨*for list*⟩ , ⟨*for list element*⟩
⟨*for clause*⟩ ::= **for** ⟨*variable*⟩ := ⟨*for list*⟩ **do**
⟨*for statement*⟩ ::= ⟨*for clause*⟩ ⟨*statement*⟩ | ⟨*label*⟩ : ⟨*for statement*⟩

4.6.2. Examples

```
for q := 1 step s until n do A[q] := B[q]
for k := 1, V1 × 2 while V1 < N do
   for j := I + G, L, 1 step 1 until N, C + D do A[k,j] := B[k,j]
```

4.6.3. Semantics. A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not, the statement following the for clause is executed.

4.6.4. The for list elements. The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1. Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2. Step-until-element. An element of the form A **step** B **until** C, where A, B, and C are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

```
    V := A;
L1: if (V − C) × sign (B) > 0 then go to Element exhausted;
    Statement S;
    V := V + B;
    go to L1;
```

where *V* is the controlled variable of the for clause and *Element exhausted* points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program.

4.6.4.3. While-element. The execution governed by a for list element of the form E **while** F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

```
L3: V := E;
    if ¬F then go to Element exhausted;
    Statement S;
    go to L3;
```

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit. Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will be the same as it was immediately preceding the execution of the go to statement.
If the exit is due to exhaustion of the for list, on the other hand, the value of the controlled variable is undefined after the exit.
4.6.6. Go to leading into a for statement
The effect of a go to statement, outside a for statement, which refers to a label within the for statement, is undefined.

4.7. Procedure statements

4.7.1. Syntax
⟨*actual parameter*⟩ ::= ⟨*string*⟩ | ⟨*expression*⟩ | ⟨*array identifier*⟩ | ⟨*switch identifier*⟩
 | ⟨*procedure identifier*⟩
⟨*letter string*⟩ ::= ⟨*letter*⟩ | ⟨*letter string*⟩ ⟨*letter*⟩
⟨*parameter delimiter*⟩ ::= , |) ⟨*letter string*⟩ : (
⟨*actual parameter list*⟩ ::= ⟨*actual parameter*⟩
 | ⟨*actual parameter list*⟩ ⟨*parameter delimiter*⟩ ⟨*actual parameter*⟩
⟨*actual parameter part*⟩ ::= ⟨*empty*⟩ | (⟨*actual parameter list*⟩)
⟨*procedure statement*⟩ ::= ⟨*procedure identifier*⟩ ⟨*actual parameter part*⟩
4.7.2. Examples
Spur (*A*) *Order*: (*7*) *Result to*: (*V*)
Transpose (*W*, *v* + *1*)
Absmax (*A*, *N*, *M*, *Yy*, *I*, *K*)
Innerproduct ($A[t, P, u]$, $B[P]$, *10*, *P*, *Y*)
These examples correspond to examples given in section 5.4.2.
4.7.3. Semantics. A procedure statement serves to invoke (call for) the execution of a procedure body (*cf.* section 5.4. Procedure declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program:[44]
4.7.3.1. Value assignment (call by value). All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (*cf.* section 2.8. Values and types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. These formal parameters will subsequently be treated as local to the procedure body.[45]
4.7.3.2. Name replacement (call by name). Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.
4.7.3.3. Body replacement and execution. Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed.[46]
4.7.4. Actual-formal correspondence. The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows:

44. [*Revised Report*:] the following operations on the program at the time of execution of the procedure statement:

45. [In the *Revised Report*, this sentence becomes:] The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (*cf.* section 5.4.5). As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block (*cf.* section 5.4.3).

46. [Added in the *Revised Report*:] If the procedure is called from a place outside the scope of any non-local quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions. For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.

This poses[47] the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. Strings cannot occur as actual parameters in procedure statements calling procedure declarations having ALGOL 60 statements as their bodies (*cf.* section 4.7.8).[48]

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4. A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier,[49] because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (*cf.* section 5.4.1) and which defines the value of a function designator (*cf.* section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5. Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6. Non-local quantities of the body. A procedure statement written outside the scope of any non-local quantity of the procedure body is undefined.[50]

4.7.7. Parameter delimiters. All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed in code. The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

5. Declarations

Declarations serve to define certain properties of the identifiers of the program. A declaration for an identifier is valid for one block.[51] Outside this block the particular identifier may be used for other purposes (*cf.* section 4.1.3).

47. [*Revised Report*:] imposes

48. [In the *Revised Report*, this section becomes:] If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, *cf.* section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

49. [Added in the *Revised Report*, before the comma:] or a string

50. [In the *Revised Report*, this section becomes:] Deleted.

51. [In the *Revised Report*, these two sentences become:] Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block.

Dynamically this implies the following: at the time of an entry into a block (through the **begin** since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.
At the time of an exit from a block (through **end**, or by a go to statement) all identifiers which are declared for the block lose their significance again.[52]
A declaration may be marked with the additional declarator **own**. This has the following effect: upon a reentry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (*cf.* sections 3.2.4 and 3.2.5) all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.
Syntax.
⟨*declaration*⟩ ::= ⟨*type declaration*⟩ | ⟨*array declaration*⟩ | ⟨*switch declaration*⟩
  | ⟨*procedure declaration*⟩

5.1. Type declarations

5.1.1. Syntax
⟨*type list*⟩ ::= ⟨*simple variable*⟩ | ⟨*simple variable*⟩ , ⟨*type list*⟩
⟨*type*⟩ ::= **real** | **integer** | **Boolean**
⟨*local or own type*⟩ ::= ⟨*type*⟩ | **own** ⟨*type*⟩
⟨*type declaration*⟩ ::= ⟨*local or own type*⟩ ⟨*type list*⟩
5.1.2. Examples
integer *p*, *q*, *s*
own Boolean *Acryl*, *n*
5.1.3. Semantics. Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values **true** and **false**.
In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.
For the semantics of **own**, see the fourth paragraph of section 5 above.

5.2. Array declarations

5.2.1. Syntax
⟨*lower bound*⟩ ::= ⟨*arithmetic expression*⟩
⟨*upper bound*⟩ ::= ⟨*arithmetic expression*⟩
⟨*bound pair*⟩ ::= ⟨*lower bound*⟩ : ⟨*upper bound*⟩
⟨*bound pair list*⟩ ::= ⟨*bound pair*⟩ | ⟨*bound pair list*⟩ , ⟨*bound pair*⟩
⟨*array segment*⟩ ::= ⟨*array identifier*⟩ [⟨*bound pair list*⟩]
  | ⟨*array identifier*⟩ , ⟨*array segment*⟩
⟨*array list*⟩ ::= ⟨*array segment*⟩ | ⟨*array list*⟩ , ⟨*array segment*⟩
⟨*array declaration*⟩ ::= **array** ⟨*array list*⟩ | ⟨*local or own type*⟩ **array** ⟨*array list*⟩
5.2.2. Examples
array *a*, *b*, *c*[*7*:*n*,*2*:*m*], *s*[−*2*:*10*]
own integer array *A*[**if** *c* < *0* **then** *2* **else** *1*:*20*]
real array *q*[−*7*:−*1*]

52. [*Revised Report*:] lose their local significance.

5.2.3. Semantics. An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts, and the types of the variables.

5.2.3.1. Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : . The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3. Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type **real** is understood.

5.2.4. Lower upper bound expressions.

5.2.4.1. The expressions will be evaluated in the same way as subscript expressions (*cf.* section 3.1.4.2).

5.2.4.2. The expressions can only depend on variables and procedures which are non-local to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4. The expressions will be evaluated once at each entrance into the block.

5.2.5. The identity of subscripted variables. The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3. Switch declarations

5.3.1. Syntax

⟨*switch list*⟩ ::= ⟨*designational expression*⟩ | ⟨*switch list*⟩ , ⟨*designational expression*⟩
⟨*switch declaration*⟩ ::= **switch** ⟨*switch identifier*⟩ := ⟨*switch list*⟩

5.3.2. Examples

switch $S := S1, S2, Q[m]$, **if** $v > -5$ **then** $S3$ **else** $S4$
switch $Q := p1, w$

5.3.3. Semantics. A switch declaration defines the values corresponding to a switch identifier.[53] These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, *1*, *2*, . . ., obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (*cf.* section 3.5. Designational expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4. Evaluation of expressions in the switch list. An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5. Influence of scopes. Any reference to the value of a switch designator from outside the scope of any quantity entering into the designational expression for this particular value is undefined.[54]

53. [In the *Revised Report*, this sentence becomes:] A switch declaration defines the set of values of the corresponding switch designators.

54. [In the *Revised Report*, this section becomes:] Influence of scopes. If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4. Procedure declarations

5.4.1. Syntax

```
⟨formal parameter⟩ ::= ⟨identifier⟩
⟨formal parameter list⟩ ::= ⟨formal parameter⟩
    | ⟨formal parameter list⟩ ⟨parameter delimiter⟩ ⟨formal parameter⟩
⟨formal parameter part⟩ ::= ⟨empty⟩ | ( ⟨formal parameter list⟩ )
⟨identifier list⟩ ::= ⟨identifier⟩ | ⟨identifier list⟩ , ⟨identifier⟩
⟨value part⟩ ::= value ⟨identifier list⟩ ; | ⟨empty⟩
⟨specifier⟩ ::= string | ⟨type⟩ | array | ⟨type⟩ array | label | switch
    | procedure | ⟨type⟩ procedure
⟨specification part⟩ ::= ⟨empty⟩ | ⟨specifier⟩ ⟨identifier list⟩ ;
    | ⟨specification part⟩ ⟨specifier⟩ ⟨identifier list⟩ ;
⟨procedure heading⟩ ::=
      ⟨procedure identifier⟩ ⟨formal parameter part⟩ ; ⟨value part⟩ ⟨specification part⟩
⟨procedure body⟩ ::= ⟨statement⟩ | ⟨code⟩
⟨procedure declaration⟩ ::= procedure ⟨procedure heading⟩ ⟨procedure body⟩
    | ⟨type⟩ procedure ⟨procedure heading⟩ ⟨procedure body⟩
```

5.4.2. Examples (see also the examples at the end of the report).

```
procedure Spur (a) Order: (n) Result: (s); value n; array a; integer n; real s;
begin integer k;
  s := 0; for k := 1 step 1 until n do s := s + a[k,k]
end
procedure Transpose (a) Order: (n); value n; array a; integer n;
begin real w; integer i, k;
for i := 1 step 1 until n do
  for k := 1 + i step 1 until n do
  begin w := a[i,k];
        a[i,k] := a[k,i];
        a[k,i] := w
  end
end Transpose
integer procedure Step (u); real u;
Step := if 0 ≦ u ∧ u ≦ 1 then 1 else 0
procedure Absmax (a) size: (n, m) Result: (y) Subscripts: (i, k);
comment The absolute greatest element of the matrix a, of size n by m is transferred to y,
and the subscripts of this element to i and k;
array a; integer n, m, i, k; real y;
begin integer p, q;
  y := 0;
  for p := 1 step 1 until n do for q := 1 step 1 until m do
    if abs (a[p,q]) > y then begin y := abs (a[p,q]); i := p; k := q end
end Absmax
procedure Innerproduct (a, b) Order: (k, p) Result: (y); value k; integer k, p; real y, a, b;
begin real s;
  s := 0; for p := 1 step 1 until k do s := s + a × b;
  y := s
end Innerproduct
```

5.4.3. Semantics. A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (*cf.* section 3.2. Function designators and section 4.7. Procedure statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are non-local to the body may well be local to the block in the head of which the procedure declaration appears.[55]

5.4.4. Values of function designators. For a procedure declaration to define the value of a function designator there must, within the procedure body, occur an assignment of a value to the procedure identifier, and in addition the type of this value must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration.

Any other occurrence of the procedure identifier within the procedure body denotes activation of the procedure.[56]

5.4.5. Specifications. In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once and formal parameters called by name (*cf.* section 4.7.3.2) may be omitted altogether.[57]

5.4.6. Code as procedure body. It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language.

Examples of procedure declarations

Example 1

```
procedure euler (fct, sum, eps, tim);
value eps, tim; integer tim; real procedure fct; real sum, eps;
comment euler computes the sum of fct(i) for i from zero up to infinity by means of a suitably
refined euler transformation. The summation is stopped as soon as tim times in succession
the absolute value of the terms of the transformed series are found to be less than eps. Hence,
one should provide a function fct with one integer argument, an upper bound eps, and an
integer tim. The output is the sum sum. euler is particularly efficient in the case of a slowly
convergent or divergent alternating series;
begin integer i, k, n, t; array m[0:15]; real mn, mp, ds;
i := n := t := 0; m[0] := fct (0); sum := m[0] / 2;
nextterm: i := i + 1; mn := fct (i);
            for k := 0 step 1 until n do
               begin mp := (mn + m[k]) / 2; m[k] := mn; mn := mp end means;
```

55. [Added in the *Revised Report*:] The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labeling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

56. [In the *Revised Report*, this section becomes:] Values of function designators. For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

57. [In the *Revised Report*, this sentence becomes:] In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (*cf.* section 4.7.3.1) must be supplied and specifications of formal parameters called by name (*cf.* section 4.7.3.2) may be omitted.

```
            if (abs (mn) < abs (m[n])) ∧ (n < 15) then
               begin ds := mn / 2; n := n + 1; m[n] := mn end accept
            else ds := mn;
            sum := sum + ds;
            if abs (ds) < eps then t := t + 1 else t := 0;
            if t < tim then go to nextterm
end euler
```

EXAMPLE 2[58]

```
procedure RK (x, y, n, FKT, eps, eta, xE, yE, fi);
value x, y; integer n; Boolean fi; real x, eps, eta, xE; array y, yE; procedure FKT;
comment: RK integrates the system yk' = fk (x, y1, y2, ..., yn) (k = 1, 2, ... n) of differential
equations with the method of Runge-Kutta with automatic search for appropriate length
of integration step. Parameters are: The initial values x and y[k] for x and the unknown
functions yk (x). The order n of the system. The procedure FKT (x, y, n, z) which represents
the system to be integrated, i. e. the set of functions fk. The tolerance values eps and eta which
govern the accuracy of the numerical integration. The end of the integration interval xE. The
output parameter yE which represents the solution at x = xE. The Boolean variable fi, which
must always be given the value true for an isolated or first entry into RK. If however the
functions y must be available at several meshpoints x0, x1, ..., xn, then the procedure must
be called repeatedly (with x = xk, xE = xk+1, for k = 0, 1, ..., n − 1) and then the later
calls may occur with fi = false which saves computing time. The input parameters of FKT
must be x, y, n, the output parameter z represents the set of derivatives z[k] = fk (x, y[1],
y[2], ..., y[n]) for x and the actual y's. A procedure comp enters as a non-local identifier;
begin
     array z, y1, y2, y3[1:n]; real x1, x2, x3, H; Boolean out;
     integer k, j; own real s, Hs;
     procedure RK1ST (x, y, h, xe, ye); real x, h, xe; array y, ye;
           comment: RK1ST integrates one single Runge-Kutta step with initial values x,
           y[k] which yields the output parameters xe = x + h and ye[k], the latter being
           the solution at xe.
           Important: the parameters n, FKT, z enter RK1ST as non-local entities;
           begin
              array w[1:n], a[1:5]; integer k, j;
              a[1] := a[2] := a[5] := h / 2; a[3] := a[4] := h; xe := x;
              for k := 1 step 1 until n do ye[k] := w[k] := y[k];
              for j := 1 step 1 until 4 do
              begin
                 FKT (xe, w, n, z);
                 xe := x + a[j];
                 for k := 1 step 1 until n do
                 begin
                    w[k] := y[k] + a[j] × z [k];
                    ye[k] := ye[k] + a[j + 1] × z[k] / 3
                 end k
              end j
           end RK1ST;
```

58. This RK-program contains some new ideas which are related to ideas of S. GILL, A process for the step by step integration of differential equations in an automatic computing machine, Proc. Camb. Phil. Soc. Vol. 47 (1951) p. 96, and C.-E. FRÖBERG, On the solution of ordinary differential equations with digital computing machines, Fysiograf. Sällsk. Lund Förhd. 20 Nr. 11 (1950) p. 136–152. It must be clear however that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer [this was not true anymore when the *Revised Report* was published].

```
Begin of program:
      if fi then begin H := xE − x; s := 0 end else H := Hs;
      out := false;
AA: if (x + 2.01 × H − xE > 0) ≡ (H > 0) then
      begin Hs := H; out := true; H := (xE − x) / 2 end if;
      RK1ST (x, y, 2 × H, x1, y1);
BB: RK1ST (x, y, H, x2, y2); RK1ST (x2, y2, H, x3, y3);
      for k := 1 step 1 until n do if comp (y1[k], y3[k], eta) > eps then go to CC;
      comment: comp (a, b, c) is a function designator, the value of which is the absolute value
      of the difference of the mantissæ of a and b, after the exponents of these quantities have
      been made equal to the largest of the exponents of the originally given parameters a, b, c;
      x := x3; if out then go to DD;
      for k := 1 step 1 until n do y[k] := y3[k];
      if s = 5 then begin s := 0; H := 2 × H end if;
      s := s + 1; go to AA;
CC: H := 0.5 × H; out := false; x1 := x2;
      for k := 1 step 1 until n do y1[k] := y2[k];
      go to BB;
DD: for k := 1 step 1 until n do yE[k] := y3[k]
end RK
```

Alphabetic index of definitions of concepts and syntactic units

All references are given through section numbers. The references are given in three groups:

def Following the abbreviation "def" reference to the syntactic definition (if any) is given.

synt Following the abbreviation "synt" references to the occurrences in metalinguistic formulæ are given. References already quoted in the def-group are not repeated.

text Following the word "text" the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined[59] words have been collected at the beginning. The examples have been ignored in compiling the index.

59. [boldfaced]

60. [*Revised Report*:] ⟨*program*⟩, def 4.1.1 text 1