REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 60*

P. NAUER, EDITOR

Dedicated to the Memory of WILLIAM TURANSKI

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.

CONTENTS

*Reprinted from *Comm ACM*, 6, 1, 1963, 1-17, copyright 1963.

INTRODUCTION

Background

After the publication of a preliminary report on the algorithmic language ALGOL,[1,2] 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.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the *Communications of the ACM*, 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,[3] 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.

April 1962 Conference [Edited by M. Woodger]

A meeting of some of the authors of ALGOL 60 was held on April 2–3, 1962 in Rome, Italy, through the facilities and courtesy of the International Computation Centre. The following were present:

Authors	*Advisers*	*Observer*
F. L. Bauer	M. Paul	W. L. van der Poel (Chairman, IFIP TC 2.1 Working Group ALGOL)
J. Green	R. Franciotti	
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 pro-

* [EDITOR'S NOTE. 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.]

[1] Preliminary report—International Algebraic Language. *Comm. ACM 1*, 12 (1958), 8.

[2] 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. *Num. Math. 1* (1959), 41–60.

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

gramming 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.

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 bracket [ ]	Lowering of the line between the brackets and removal of the brackets
Exponentiation ↑	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 läßt, läßt 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 formulae—called assignment statements.

To show the flow of computational processes, certain nonarithmetic 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 refer to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets **begin** and **end** to form a compound statement.

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. 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.

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.

In the sequel the syntax and semantics of the language will be given.[4]

1.1. Formalism for Syntactic Description

The syntax will be described with the aid of metalinguistic formulae.[5] Their interpretation is best explained by an example

⟨ab⟩ ::= (| [| ⟨ab⟩ (| ⟨ab⟩⟨d⟩

Sequences of characters enclosed in the brackets ⟨ ⟩ 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 ⟨ab⟩. It indicates that ⟨ab⟩ may have the value (or [or that given some legitimate value of ⟨ab⟩, another may be formed by following it with the character (or by following it with some value of the variable ⟨d⟩. If the values of ⟨d⟩ are the decimal digits, some values of ⟨ab⟩ are:

[(((1(37(
(12345(
(((
[86

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 formulae have been given in more than one place.

Definition:

⟨empty⟩ ::=
(i.e. the null string of symbols).

[4] Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is 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.

[5] Cf. J. W. Backus, The syntax and semantics of the proposed international algebraic language of the Zürich ACM–GAMM conference. Proc. Internat. Conf. Inf. Proc., UNESCO, Paris, June 1959.

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[6] (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**[7]
⟨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.

For the purpose of including text among the symbols of

[6] It should be particularly noted that throughout the reference language underlining [in typewritten copy; boldface type in printed copy—Ed.] 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. Within the present report [not including headings—Ed.], boldface will be used for no other purpose.

[7] **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.

a program the following "comment" conventions hold:

The sequence of basic symbols:	*is equivalent to*
; **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 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.

2.4. 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).

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 and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

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⟩

3.1.2. Examples

epsilon
detA
*a*17
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

⟨adding operator⟩ ::= +|−
⟨multiplying operator⟩ ::= ×|/|÷
⟨primary⟩ ::= ⟨unsigned number⟩|⟨variable⟩|⟨function designator⟩|(⟨arithmetic expression⟩)
⟨factor⟩ ::= ⟨primary⟩|⟨factor⟩↑⟨primary⟩
⟨term⟩ ::= ⟨factor⟩|⟨term⟩⟨multiplying operator⟩⟨factor⟩
⟨simple arithmetic expression⟩ ::= ⟨term⟩|⟨adding operator⟩⟨term⟩|⟨simple arithmetic expression⟩⟨adding operator⟩⟨term⟩
⟨if clause⟩ ::= **if** ⟨Boolean expression⟩**then**
⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩|⟨if clause⟩⟨simple arithmetic expression⟩**else**⟨arithmetic expression⟩

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** z

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 5.4.4. Values of Function Designators) 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**, mathematically defined as follows:

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

(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,

$2\uparrow n\uparrow k$ 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), 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

⟨relational operator⟩ ::= $<|\leqq|=|\geqq|>|\neq$
⟨relation⟩ ::= ⟨simple arithmetic expression⟩
⟨relational operator⟩⟨simple arithmetic expression⟩
⟨Boolean primary⟩ ::= ⟨logical value⟩|⟨variable⟩|
⟨function designator⟩|⟨relation⟩|(⟨Boolean expression⟩)
⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩|$\neg$⟨Boolean primary⟩
⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩|
⟨Boolean factor⟩$\wedge$⟨Boolean secondary⟩
⟨Boolean term⟩ ::= ⟨Boolean factor⟩|⟨Boolean term⟩
$\vee$⟨Boolean factor⟩
⟨implication⟩ ::= ⟨Boolean term⟩|⟨implication⟩$\supset$⟨Boolean term⟩
⟨simple Boolean⟩ ::= ⟨implication⟩|
⟨simple Boolean⟩$\equiv$⟨implication⟩
⟨Boolean expression⟩ ::= ⟨simple Boolean⟩|
⟨if clause⟩⟨simple Boolean⟩ **else** ⟨Boolean expression⟩

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 $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\supset$ (implies), and $\equiv$ (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 zeros 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

⟨unlabelled basic statement⟩ ::= ⟨assignment statement⟩|⟨go to statement⟩|⟨dummy statement⟩|⟨procedure statement⟩
⟨basic statement⟩ ::= ⟨unlabelled basic statement⟩|⟨label⟩:⟨basic statement⟩
⟨unconditional statement⟩ ::= ⟨basic statement⟩|⟨compound statement⟩|⟨block⟩
⟨statement⟩ ::= ⟨unconditional statement⟩|⟨conditional statement⟩|⟨for statement⟩
⟨compound tail⟩ ::= ⟨statement⟩ **end** |⟨statement⟩ ;⟨compound tail⟩
⟨block head⟩ ::= **begin** ⟨declaration⟩|⟨block head⟩ ;⟨declaration⟩
⟨unlabelled compound⟩ ::= **begin** ⟨compound tail⟩
⟨unlabelled block⟩ ::= ⟨block head⟩ ; ⟨compound tail⟩
⟨compound statement⟩ ::= ⟨unlabelled compound⟩|⟨label⟩:⟨compound statement⟩
⟨block⟩ ::= ⟨unlabelled block⟩|⟨label⟩:⟨block⟩
⟨program⟩ ::= ⟨block⟩|⟨compound statement⟩

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**

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 nonlocal to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling 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.

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

4.2. Assignment Statements

4.2.1. Syntax

⟨left part⟩ ::= ⟨variable⟩ := |⟨procedure identifier⟩ :=
⟨left part list⟩ ::= ⟨left part⟩|⟨left part list⟩⟨left part⟩
⟨assignment statement⟩ ::= ⟨left part list⟩⟨arithmetic expression⟩|⟨left part list⟩⟨Boolean expression⟩