STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM
REKENAFDELING

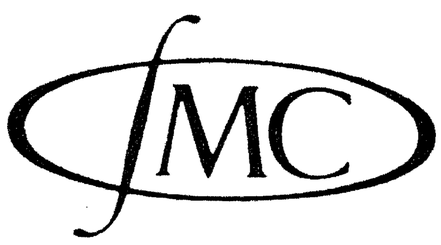
MR 75

A Proposal for a Report on a Successor of ALGOL 60

by

Niklaus Wirth

IFIP WG 2.1
Working Document
Limited Circulation
Copy No. 63



August 1965

The Mathematical Centre at Amsterdam, founded the 11th of February, 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications, and is sponsored by the Netherlands Government through the Netherlands Organization for Pure Research (Z.W.O.) and the Central National Council for Applied Scientific Research in the Netherlands (T.N.O.), by the Municipality of Amsterdam and by several industries.

Introductory Comments

This paper contains in the sequel a proposal for a report on a successor of ALGOL 60. It was prepared upon the suggestion of the IFIP Working Group 2.1 on ALGOL, and is intended to be a Working Document of WG 2.1. As far as possible, i.e. without infringing on the consistency of the proposed language, opinions and suggestions expressed at the WG 2.1 meeting at Princeton in May 1965 were used as guide lines. The following are a few comments on important issues in the design of the language.

1. Data types.

To the value types integer, real, and Boolean of ALGOL 60 have been added the types complex, bits (bit sequence), and string (character sequence). Values of these six types are to be considered as basic units of data. In general, a value may be a simple value (i.e. it is of one of these six types), or a structured value, defined to be an ordered set of values.

Variables are classified according to the same principles: they are either simple variables whose value is a simple value ("simple" is here not used in the ALGOL 60 sence of "non-subscripted"), or structured variables, consisting of an ordered set of variables. A structured variable can be either of type tree, in which case the number and type of variables in the set is determined by the execution of the program, or of type array, in which case the number, the subscripts, and the type of the elements of the set are determined by the declaration. In the case of a tree, its elements may vary in type among each other (necessarily some elements will be sub-trees, other ones will be simple variables), and also may vary their own type during execution of the program, (allowing the structure of the tree to change dynamically). In the case of an array, all elements have the same type. Subscripts of tree variables must be positive integers.

Procedure declarations require parameter specifications. Thus, dynamic type-checking can always be avoided, except in the known cases when tree variables are involved.

Greater precision of real and complex variables can be specified by preceding their declaration with the symbol long.

2. Side-effects.

There is the nasty problem of side-effects of function procedures. To me, they are desirable as long as they do not influence the value of any variable in the expression of which the function designator is a part. Thus there are desirable (read: useful) and undesirable side-effects, and the only way to keep the first and eliminate the latter kind, is to explicitly declare that the order of evaluation of primaries in an expression is not specified. Then, a program using a side-effect of the latter kind is an undefined program. The difficulty is, of course, to detect this undefinedness.

An additional benefit from leaving the order of evaluation of primaries undefined is that in certain cases the efficiency of processing systems may be increased. This may not be very relevant in cases of simple values, but it becomes crucial in the case of tree variables. Fixing the order of evaluation of primaries would amount to requiring that upon evaluation of a primary a copy of that primary (tree) would have to be made in order to preserve its value from side-effects which might alter the original primary.

Following the principle of consistency, also the order of executing an assignment statement or an actual parameter list is left undefined. Indeed, if the definition of the assignment statement of ALGOL 60 were adopted, a side-effect might delete the variable to which an assignment was to be made.

3. Name- and Value Parameters.

As in ALGOL 60, two different kinds of parameters are distinguished. Their properties as well as their denotation, however, differ from their counterparts in ALGOL 60. Name- and value parameters are distinguished at the call side rather than at the declaration side. An actual parameter which is a name parameter is enclosed between a pair of apostrophes, and it is substituted for the corresponding formal parameter literally. A name parameter may be an expression, a statement, an array identifier or a procedure identifier. By contrast, a value parameter can only be an expression. This expression is evaluated, and the corresponding formal parameter is replaced by a possible reference denotation of the obtained result. As a consequence, no assignment can be made to a value parameter.

The idea of introducing a third kind of parameter, namely an adress parameter requiring the calculation of possible subscripts but not the evalu-

ation of the adressed quantity, has been rejected after long considerations. Instead, only the two extreme solutions, complete evaluation and no evaluation at all, have been retained.

Specifications are required of all formal parameters, except if a formal parameter is used as actual name parameter only, i.e. passed on only. If a formal is used in an expression (input parameter), then the specified type must be assignment compatible with the type of the actual parameter; if the formal is used in a left part of an assignment statement (output parameter), then the type of the actual must be assignment compatible with the specified type of the formal parameter. Assignment compatibility is defined such that the following assignments are possible:

<type-1> := <type-1>

<real> := <integer>

<complex> := <integer>

<complex> := <real>

4. Labels.

The label is not defined to be a quantity upon which operations can be performed. The label can only occur in two contexts, namely a) preceding a colon and thereby labeling the subsequent statement, and b) following the symbol goto, thereby forming a goto statement.

Thus, designational expressions are eliminated. The removal of the switch designator (and the switch declaration) are compensated by the introduction of the case statement. The possibility to pass on a label as a parameter is also excluded, but instead compensated by the possibility that an entire statement, and therefore in particular a goto statement, can be passed on as a name-parameter.

5. Environment and Standard Procedures.

It is admitted that an ALGOL program is understood to be executed in an environment where quantities are declared which are accessible to the program. They are either standard procedures (permanently required in every environment) or other quantities (e.g. so-called code procedures) which are introduced into the environment by unspecified means. It is also possible that such procedures may cause the insertion of actions which are invisible

in the program. Thereby the possibility of specifying interrupts (and possibly means which are sufficient to indicate parallelisms) is given.

6. Iterative statements.

The ALGOL 60 for statement has been replaced by the so-called iterative statement, which is simpler and free from undesirable complexity. The iterated statement is prefixed with one of two kinds of clauses, which are of the form

i) while b do and

ii) over v do n times

In i) the statement is repeatedly executed as long as b is true. In ii) the statement is executed n times with the variable v assuming the values 1, 2,..., n. Here, the arithmetic expression whose value determines n is evaluated only once.

It has been considered to let the variable v be implicitly declared by the over clause with a scope ranging over the iterative statement. This would seem to be attractive with respect to optimization possibilities, if in addition assignments to v would be prohibited. It was felt, however, that the implicit declarations would be an undesirable and foreign feature in ALGOL, while the concept of "frozen variables" should be introduced in greater generality, if at all.

A considered proposal was the following, illustrated on an example:

```
begin integer i, j, k; array [1:2] real u;
procedure p; i:=1;
i:=0; j:=2;
begin freeze i,j; comment the values of i,j may not be altered within this
block. the statements labelled "b" and "c" are thus illegal;
a: u[i,j]:= u[i,j] + 1;
b: i:=2;
c: p
end
end
```

Statements like the one labelled "a" could easily be optimized (avoiding the identification of the array element more than once), since it is guaranteed that the subscripts do not change. The difficulty of detecting illegal assign-

ments, however, (statement c) seemed to be a serious objection to this approach. It was finally felt that no features should be introduced which provide additional facilities for the programmer to contradict himself, if those features merely serve to facilitate efficient compilation.
Admittedly, the iterative statement adds no power to the language, but only notational convenience. It should also be an effective tool against prolific use of goto statements. It is believed that it even achieves a gain on notational convenience compared to the ALGOL 60 for statement, because it is conceptually much simpler.

7. Precision of real arithmetic, size of integers, bit- and character sequences.
The specification of the above mentioned quantities along with the declaration of variables, as it is done in PL/I, was first thought to be an attractive solution. Upon closer consideration, however, the consequences of such specifications upon the language (not to speak of their implementation) seemed to be controversial, unclear, and manifold, and their usefulness questionable (even vociferously declared as nonexistent by Peter Naur). Thus, these quantities are left unspecified, but it is recognized that implementations may impose fixed limits on them.
In order to express that in a program certain computations should preferably be performed with higher precision than other computations, variable declarations of type real and complex may be preceded by the symbol long. This would then express the programmers intent that variables declared in this way would represent values with greater precision.
This document will be distributed among the members of WG 2.1 well in advance of the next meeting in the end of October. I hope to receive comments and suggestions based on this proposal from the WG 2.1 members as soon as possible, and intend to prepare a survey of them for that meeting. This proposal is supposed to be a consistent and complete description of the language, and incorporating major revisions or supplements without violating this consistency is an intricate affair. Contributions which are thought to be of rather important and extensive nature should therefore preferably be edited in a form which is compatible with the form of this report, i.e. they should contain references to all the places where altera-

tions are necessary. This will not only facilitate my work, but also force the contributor to consider carefully all the consequences of his amendments. Please address your comments to

Niklaus Wirth
Computer Science Department
Stanford University
Stanford, California

This document was prepared and edited at the Mathematical Centre in Amsterdam. I would like to express my sincere appreciation and gratitude to Prof. A. van Wijngaarden for his many invaluable critizisms and suggestions and for his support. I also express my thanks to the members of the staff at MC for their helpful discussions and the editing of the manuscript. Gratefully acknowledged is also the support of the U.S. National Science Foundation (GP 4053), which made my stay at Amsterdam possible.

Niklaus Wirth

C O N T E N T S

1. Introduction

The Reference Language is a phrase structure language, defined by a formal system. This formal system makes use of the notation and the definitions explained below.

The structure of the language ALGOL is determined by the four quantities
(1) T, called the set of basic constituents of the language,
(2) U, called the set of syntactic entities,
(3) P, called the set of syntactic rules, and
(4) <program>, a specific member of the set U.

Notation

A syntactic entity is denoted by its name (a sequence of letters) enclosed in the brackets < and >. A syntactic rule has the form
<A> ::= x
where <A> is a member of U, x is a member of the set V, and <A> ≠ x.
V shall be the set of all possible sequences of basic constituents and syntactic entities, simply to be called "sequences" below.
The form
<A> ::= x |y |...|z
is used as an abbreviation for the set of syntactic rules
<A> ::= x
<A> ::= y
.....
<A> ::= z

Definitions

1. A sequence x is said to directly produce a sequence y, if and only if there exist (possibly empty) sequences u and w, so that for some <A> in U x = u<A>w, y = uvw, and <A> ::= v is a rule in P.
2. A sequence x is said to produce a sequence y, if and only if there exists an ordered set of sequences s[0], s[1], ..., s[n], so that x = s[0], s[n] = y, and s[i-1] directly produces s[i] for all i = 1, ..., n.

3. A sequence x is said to be an ALGOL program, if and only if its constituents are members of the set T, and x can be produced from the syntactic entity <program>.

The sets T and U are defined through enumeration of their members in section 2 of this Report (cf. also 4.6.). The members of the set P of syntactic rules are given throughout the sequel of the Report.
In order to provide explanations for the meaning of ALGOL programs, the letter sequences denoting syntactic entities have been chosen to be English words describing approximately the nature of that syntactic entity or construct. Where words which have appeared in this manner are used elsewhere in the text, they will refer to the corresponding syntactic definition.

It is recognized that typographical entities of lower order than basic symbols, called characters, exist. Some basic symbols are identical with characters, other ones, so-called word-delimiters, are composed of one or more characters. Neither the set of available characters nor the decomposition of basic symbols into them will be defined here. It is understood that there may exist characters which are neither basic symbols nor constituents of them; these characters may however, enter the program as constituents of strings, i.e. character sequences delimited by so-called string quotes. The basic constituents of the language are thus the basic symbols (cf. 2. 1.) and these strings (cf. 4.6.).
All quantities referred to in a program must be defined. Their definition is achieved either within the ALGOL program by so-called declarations and label definitions, or is thought to be done in a text, written in another language, in which the ALGOL program is embedded. A program which contains references to quantities defined in the latter way can only be executed in an environment where these quantities are known.
Some examples of such quantities, which are assumed to be known in every environment, are listed and described in section 8 of this Report.

The execution of a program can be considered as a sequence of units of action. The sequence of these units of action is defined as the evaluation of expressions and the execution of statements as denoted by the program.

As an after effect of the execution of certain procedures, units of action not mentioned in the text of the program may be inserted between the ones denoted by the text. These so-called interrupts may be used by the system which executes the program to act upon the occurrence of certain events which may arise during computation, e.g. an arithmetic operation resulting in a value exceeding a given range. The procedures mentioned above belong to the quantities not defined within the ALGOL text.

In the definition of the language the evaluation or execution of certain constructs is (1) either not precisely defined, e.g. real arithmetic, or (2) is left undefined, e.g. the order of evaluation of primaries in expressions, or (3) is even said to be undefined or not valid. This is to be interpreted in the sense that a program, which uses constructs of the first two categories, only fully defines a computational process, if accompanying information specifies what is not given in the definition of the language. No meaning can be attributed to a program using constructs of the third category.

For notational convenience, the following abbreviations will be permitted:

1. Any occurrence of the sequence of the two symbols "] [" may be replaced by ",".

2. Any occurrence of the sequence of the three symbols "] array [" may be replaced by ",".

Moreover, insertion of a sequence of symbols

comment <any sequence of characters not containing ; > ;

shall be permitted between any two basic constituents of the ALGOL program, without having any effect on the meaning of that program.

2. The sets of basic symbols and syntactic entities

2.1. Basic Symbols

Note : This is left to the reader of this draft proposal as an exercise.
Hints:

1. Symbols not inherited from ALGOL 60:

÷ | ≡ | ⊃ | ‘ | ’ | own | Boolean | switch | label | for | step | until | value

2. Symbols not present in ALGOL 60, but used in this proposal:

abs | div | rem | case | of | " | ' | conc | ⊥ | b | _ |
complex | bits | tree | logical | over | times | long

2.2. Syntactic entities

Note: Left to the conscientious reader of this proposal as an exercise.

3. Identifiers

3.1. Syntax

<identifier> ::= <letter> |<identifier><letter> |<identifier><digit>
<variable identifier> ::= <identifier>
<procedure identifier> ::= <identifier>
<label identifier> ::= <identifier>
<array identifier> ::= <identifier>
<letter> ::= a |b |...|z
<digit> ::= 0 |1 |2 |3 |4 |5 |6 |7 |8 |9
<identifier list> ::= <identifier> |<identifier list>, <identifier>

3.2. Semantics

Identifiers have no inherent meaning, but serve for the identification of variables, procedures, labels and formal parameters. They may be chosen freely.
The set of letters may be extented with suitably chosen characters.
Every identifier used in the program must be defined. If the identifier identifies a variable or a procedure, this is achieved through a declaration (cf. section 5). The identifier is then said to denote that variable or that procedure, and it is said to be a variable identifier, or a procedure identifier. If the identifier identifies a label, then definition occurs through the labelling of a statement with the identifier, (cf. section 7) and the identifier is then said to stand as a label, and to be a label identifier.
If the identifier identifies a formal parameter, definition is achieved through the occurrence of the identifier in a formal parameter list (cf. 5.3.).

The identification of the quantities, labels or formal parameters which are denoted by a given identifier, is determined by the following process:
Step 1: If the identifier is defined within the smallest block embracing the given occurrence of that identifier by a declaration of a quantity or by its standing as a label, then it denotes that quantity or that label.
Step 2: Otherwise, if that block is a procedure body, and if the given identifier is identical with a formal parameter in the associated procedure heading, then it denotes that formal parameter.

Otherwise, this process is repeated considering the smallest block embracing the block which has previously been considered.
If either step 1 or step 2 could lead to more than one quantity, label, or formal parameter denoted by the identifier, then the identification is undefined.
The scope of a quantity, a label, or a formal parameter is the set of occurrences of the identifier which by the given process are shown to denote that quantity, label, or formal parameter.

4. Quantities, values, and types

The following kinds of quantities are distinguished:
Constants, variables, and procedures.
Constants and variables are said to possess a value.
The value of a constant is determined by the denotation of the constant.
In the language, every constant has a reference denotation (cf. 4.1.-4.6.).
The value of a variable is the one most recently assigned to that variable.
A value is (recursively) defined as either being a simple value, or a structured value i.e. an ordered set of zero, one, or more values. Every value is said to be of a certain type. The following types of simple values are distinguished:
integer: the value is an integer,
real: the value is a real number,
complex: the value is a complex number,
logical: the value is a logical value,
bits: the value is a linear sequence of bits, and,
string: the value is a linear sequence of characters.

The following types of structured values are distinguished:
array: the value is an ordered set of values, all of equal type,
tree: the value is an ordered set of values.

A procedure may have a value, in which case it is said to be a function procedure, or it may not have a value, in which case it is called a proper procedure. The value of a function procedure is defined as the value which results from the execution of the procedure body (cf. 7.3.2.).

Subsequently, the reference denotation of constants is defined. The reference denotation of any constant consists of a sequence of characters. This, however, does not imply that the value of the denoted constant is a sequence of characters, nor that it has the properties of a sequence of characters.

4.1. Numbers

4.1.1. Syntax

<number> ::= <complex number> |<real number> |<integer>

<complex number> ::= <real part> ⊥ <imaginary part>

<real part> ::= <real number>

<imaginary part> ::= <real number>

<real number> ::= <unsigned real number> |
_ <unsigned real number>

<unsigned real number> ::= <unscaled real> |
<unscaled real><scale factor>

<unscaled real> ::= <unsigned integer> |
<unsigned integer><fraction> | <fraction>

<fraction> ::= .<unsigned integer>

<scale factor> ::= $_{10}$<integer>

<integer> ::= <unsigned integer> | _ <unsigned integer>

<unsigned integer> ::= <digit> | <unsigned integer><digit>

4.1.2. Semantics

Unsigned integers have the conventional decimal notation. _ denotes a monadic minus sign.

Unsigned real numbers have the conventional decimal notation. _ denotes a monadic minus sign. The symbol $_{10}$ followed by an integer denotes a scale factor expressed as an integral power of 10.

4.2. Logical values

4.2.1. Syntax

<logical value> ::= true |false

4.3. Bit sequences

4.3.1. Syntax

<bit sequence> ::= b<bit> |<bit sequence><bit>

<bit> ::= 0 |1

4.4. Strings

4.4.1. Syntax

<string> ::= "<sequence of characters>"

4.4.2. Semantics

Strings consist of any sequence of characters enclosed by but not containing the character ", called string quote. They are considered to be basic constituents of the language (cf. section 1).

5. Declarations

Declarations serve to associate identifiers with the quantities used in the program, to attribute certain permanent properties to these quantities, and to determine their scope. The quantities declared by declarations are simple variables, trees and arrays,and procedures.
Upon exit from a block, all quantities declared within that block lose their value and significance.

Syntax:
<declaration> ::= <variable declaration> |<procedure declaration>
<variable declaration> ::= <simple variable declaration> |<tree declaration>|
<array declaration>

5.1. Simple variable declarations

5.1.1. Syntax

<simple variable declaration> ::= <simple type><identifier list>
<simple type> ::= integer |real |long real |complex |
long complex |logical |bits |string

5.1.2. Semantics

Each identifier of the identifier list is associated with a variable which is declared to be of the indicated type. A variable is called a simple variable, if its value is simple (cf. section 4). If a variable is declared to be of a certain type, then this implies that only values which are assignment compatible with this type (cf. 7.2.2.) can be assigned to it.
It is understood that the value of a variable of type integer is only equal to the value of the expression most recently assigned to it, if this value lies within certain unspecified limits. It is also understood that if a value of a variable of type real is not an integer, then it is available only with a possible, unspecified deviation from the value of the expression most recently assigned to it. If in a declaration the symbol real is preceded by the symbol long, then this deviation is intended to be smaller than when the symbol long is missing. In the case of a variable of type complex this holds

separately for the real and imaginary parts of the complex number.
It is furthermore understood that the value of a variable of type bits is only equal to the value of the expression most recently assigned to it, if the number of bits in this value does not exceed a fixed but unspecified limit.

5.2. Tree and array declarations

5.2.1. Syntax

<tree declaration> ::= tree <identifier list>
<array declaration> ::= array <bound pair><variable declaration>
<bound pair> ::= [<lower bound> : <upper bound>]
<lower bound> ::= <arithmetic expression>
<upper bound> ::= <arithmetic expression>

5.2.2. Semantics

Each identifier of the identifier list of a tree declaration, or an array declaration, is associated with a variable which is declared to be of type tree or array respectively.
For a variable to be of type tree means that only structured values can be assigned to it. A variable of type tree is understood to be the ordered set of zero or more variables whose number and types are dynamically determined by the operations performed on the variable of type tree.
A variable of type array is an ordered set of variables whose number and type are determined by the declaration and remain constant as long as the array variable is valid. The type of all elements of that ordered set is given by the variable declaration contained in the array declaration.
The number of elements is determined by the bound pair of the array declaration, which moreover determines the subscript bounds for the array. Since the definition of the array declaration is recursive, also arrays of arrays may be declared.
The number of bound pairs which precede a variable declaration is said to be the order of the array(s) denoted by the identifier(s) in that variable declaration. In order to be valid, all arithmetic expressions acting as bounds must be of type integer, and for every pair, the current value of the lower bound must not exceed the current value of the upper bound. Every expressi-

on is evaluated exactly once upon the entry of the block in which the declaration appears, and all array identifiers of a variable declaration are declared with the same number of elements, as indicated by the bound pair preceding that variable declaration.

5.3. Procedure declarations

5.3.1. Syntax

<procedure declaration> ::= <proper procedure declaration> |
<function procedure declaration>

<proper procedure declaration> ::=
procedure <procedure heading> ; <proper procedure body>

<function procedure declaration> ::= <simple type or tree> procedure
<procedure heading> ; <function procedure body>

<proper procedure body> ::= <statement>

<function procedure body>::= <expression> |<block body><expression> end

<procedure heading> ::= <procedure identifier> |
<procedure identifier>(<formal parameter list>);<specification part> |
<procedure identifier> (<formal parameter list>)

<formal parameter list> ::= <identifier list>

<specification part> ::= <specification> |
<specification part> ; <specification>

<specification> ::= <type><identifier list>

<type> ::= <variable type> |procedure |<simple type or tree> procedure

<simple type or tree> ::= <simple type> |tree

<variable type> ::= <simple type or tree> |array <variable type>

5.3.2. Semantics

A procedure declaration associates the procedure body with the identifier immediately following the symbol procedure. The principal part of the procedure declaration is the procedure body, a statement which can be caused to be executed, or an expression which can be caused to be evaluated from other parts of the block in whose heading the procedure is declared. A proper procedure is activated by a procedure statement (cf. 7.3.), a function procedure by a function designator (cf. 6.2.).

Associated with the procedure body is a heading, containing the procedure identifier and possibly a list of formal parameters with or without specifications. The procedure body always acts as a block, whether it has the form of a block or not, and the scope of the formal parameters is this block combined with the heading.

5.3.2.1. The type preceding procedure preceding the heading of a function procedure must be possibly assignment compatible with the type of the expression that constitutes or is part of the function procedure body.

5.3.2.2. Specifications of formal parameters

Every formal parameter must be specified, unless in the procedure body it occurs as name parameter only (cf. 7.3.2.). Its specified type must be such that the substitution of the formal by an actual parameter of this specified type leads to correct ALGOL expressions and statements.

6. Expressions

Expressions are rules which specify how new values are computed from existing ones. These new values are obtained by performing the operations indicated by the operators on the values of the operands.

According to the type of their value, several types of expressions are distinguished:

Syntax:

<expression> ::= <arithmetic expression> |<logical expression> |
<bit expression> |<string expression> |<tree expression>

The operands are either constants, variables, or function designators or other expressions between parentheses. The evaluation of the latter three may involve smaller units of action such as the evaluation of other expressions or the execution of statements. The value of an expression between parentheses is obtained by evaluating that expression. If an operator operates on two operands, then the order of evaluation of those operands is undefined with the exception of the case mentioned in 6.4.4.2.

The construct

<if clause><expression> else <expression>

in which the expressions are of the same type, causes the selection of an expression on the basis of the current value of the logical expression contained in the if clause. If this value is true, the value of the expression following the if clause is selected, if the value is false, the value of the expression following else is selected.

The construct

<case clause> (<expression list>)

where the expression list is either an arithmetic, logical, bit, string, or tree expression list, causes the selection of the expression whose ordinal number in the expression list is equal to the current value of the arithmetic expression contained in the case clause. In order that the case expression is defined, this expression must be of type integer, and its current value must be the ordinal number of some expression in the expression list.

6.1. Variables

6.1.1. Syntax

<variable> ::= <unsubscripted variable> |<subscripted variable>
<unsubscripted variable> ::= <variable identifier>
<subscripted variable> ::= <variable><subscript>
<subscript> ::= [<arithmetic expression>]

6.1.2. Semantics

An identifier is a variable identifier, if it is either defined by a variable declaration, or if it is a formal parameter with an analogous specification.
The syntactic unit
<variable> [<expression>]
also designates a variable, namely the one whose ordinal number in the ordered set of variables designated by <variable> is the current value of the expression enclosed in brackets. This designation is defined only if the expression is of type integer. In the case of the variable being a variable of type tree, its value must be positive and not exceed the current width of the variable, and in the case of a variable of type array, its value must lie within the declared bounds.
The value of a variable may be used in expressions for forming other values, and may be changed by assignments to that variable.
An array identifier must always be followed by a number of subscripts equal to the order of the denoted array, except in its declaration and when the array identifier occurs as a name parameter, in which case it may also occur without subscripts at all.

6.2. Function designators

6.2.1. Syntax (cf. also 7.3.1.)

<function designator> ::= <procedure identifier> |
<procedure identifier> (<actual parameter list>)

6.2.2. Semantics

An identifier is a procedure identifier, if it is either defined by a procedure declaration, or if it is a formal parameter specified procedure.

A function designator defines a value which can be obtained by a process performed in the following steps:

Step 1: Copies are taken of the body of the function procedure whose procedure identifier is given by the function designator and of the actual parameters of the latter.

Step 2, 3, 4: As specified in 7.3.2.

Step 5: The copy of the function procedure body, modified as indicated in steps 2-4, is executed as if it were written at the position of the function designator. The value of the function designator is the value of the expression which constitutes or is part of the modified function procedure body. The type of the function designator is the type preceding procedure preceding the heading of the corresponding function procedure declaration. In order for the process to be defined this type should be assignment compatible with the type of the expression.

6.3. Arithmetic expressions

6.3.1. Syntax

<arithmetic expression> ::= <simple arithmetic expression> |
        <if clause><arithmetic expression> else <arithmetic expression> |
        <case clause> (<arithmetic expression list>)

<arithmetic expression list> ::= <arithmetic expression> |
        <arithmetic expression list>, <arithmetic expression>

<if clause> ::= if <logical expression> then

<case clause> ::= case <arithmetic expression> of

<simple arithmetic expression> ::= <term> |+ <term> |- <term> |
        <simple arithmetic expression> + <term> |
        <simple arithmetic expression> - <term>

<term> ::= <factor> |<term> × <factor> |<term> / <factor> |
        <term> div <factor> |<term> rem <factor>

<factor> ::= <secondary> |<factor> ↑ <secondary>

<secondary> ::= <primary> |abs <primary>

<primary> ::= <number> |<variable> |
        <function designator> |(<arithmetic expression>)

6.3.2. Semantics

An arithmetic expression is a rule for computing a number.

The type of an arithmetic expression is the type of the result of the last operation which is necessary to evaluate the expression. Variables and function designators entered as primaries must have been declared of type integer, real, or complex.

6.3.2.1. The operators +, −, and × have the conventional meaning of addition, subtraction and multiplication. The type of the result is

i) integer, if both operands are of type integer, otherwise

ii) real, if neither operands are of type complex, otherwise

iii) complex.

6.3.2.2. The operator − may denote the monadic operation of sign inversion. The type of the result is the type of the operand. The operator + may denote the monadic operation of identity.

6.3.2.3. The operator / denotes division. The type of the result is

i) real, if the operands are of type real or integer, otherwise

ii) complex.

6.3.2.4. The operator div is defined only for two operands of type integer and yields a result of type integer mathematically defined as

a div b = sgn(a×b) × d(abs a, abs b)

where the function procedures sgn and d are declared as

```
integer procedure sgn(a); integer a;
        if a < 0 then -1 else 1;
integer procedure d(a,b); integer a,b;
        if a < b then 0 else d(a-b,b) + 1
```

6.3.2.5. The operator rem (remainder) is defined only for two operands of type integer and yields a result of type integer mathematically defined as

a rem b = a − (a div b) × b

6.3.2.6. The operator ↑ denotes exponentiation and is defined only if the second operand is of type integer, and if its value is ≥ 0. The result is of the same type as the first operand.

6.3.2.7. The monadic operator abs will yield as result the absolute value of the operand. The result is of type integer, if the operand is of type integer, real otherwise.

6.3.2.8. Precedence of operators. The syntax of 6.3.1. implies the following hierarchy of operator precedences:

abs

↑

× / div rem

+ −

Sequences of operations of equal precedence shall be executed in order from left to right.

6.3.2.9. If one of the constituent expressions which can be selected by an if clause or a case clause is of type complex, or otherwise of type real, then the type of the entire arithmetic expression is of type complex, or of type real respectively. Otherwise, it is of type integer.

6.3.2.10. Precision of arithmetic. If the result of an arithmetic operation is of type real or complex, then it is the mathematically understood result of the operation performed on operands which may deviate from the actual operands. In case of the operands being variables, this deviation, as described in 5.1.2., is intended to be relatively small if the declarations of those variables include the symbol long. The same holds again for results of operations performed on operands which themselves all have this property.

6.4. Logical expressions

6.4.1. Syntax

<logical expression> ::= <simple logical expression> |
<if clause><logical expression> else <logical expression> |
<case clause> (<logical expression list>)

<logical expression list> ::= <logical expression> |
<logical expression list>, <logical expression>

<simple logical expression> ::= <logical term> |<relation>

<logical term> ::= <logical factor> |<logical term> ∨ <logical factor>

<logical factor> ::= <logical secondary> |
<logical factor> ∧ <logical secondary>

<logical secondary> ::= <logical primary> |¬ <logical primary>

<logical primary> ::= <logical value> |<variable> |
<function designator> |(<logical expression>)

<relation> ::= <simple arithmetic expression><relational operator>
<simple arithmetic expression> |
<logical term><equality operator><logical term> |
<simple bit expression><equality operator><simple bit expression> |
<simple string expression><equality operator><simple string expression>
<relational operator> ::= <equality operator> |< |≤ |≥ |>
<equality operator> ::= = |≠

6.4.2. Semantics

A logical expression is a rule for computing a logical value. Variables and function designators entered as logical primaries must have been declared of type logical.

6.4.2.1. The relational operators have their conventional meaning, and yield the logical value true, if the relation is satisfied for the values of the two operands, false otherwise.

6.4.2.2. The operators ¬ (not), ∧ (and), and ∨ (or), operating on logical values, are defined by the following equivalences:

¬x	if x then false else true
x ∧ y	if x then y else false
x ∨ y	if x then true else y

6.4.2.3. Precedence of operators: The syntax of 6.4.1. implies the following hierarchy of operator precedences:

¬
∧
∨
= ≠

6.5. Bit expressions

6.5.1. Syntax

<bit expression> ::= <simple bit expression> |
<if clause><bit expression> else <bit expression> |
<case clause> (<bit expression list>)
<bit expression list> ::= <bit expression> |
<bit expression list>, <bit expression>

<simple bit expression> ::= <bit factor> |
        <simple bit expression> ∨ <bit factor>
<bit factor> ::= <bit secondary> |<bit factor> ∧ <bit secondary>
<bit secondary> ::= <bit primary> |¬ <bit primary>
<bit primary> ::= <bit sequence> |<variable> |
        <function designator> |(<bit expression>)

6.5.2. Semantics

A bit expression is a rule for computing a bit sequence.
The operators ∨, ∧ and ¬ produce a result of type bits, every bit being dependent on the corresponding bit(s) in the operand(s) as follows:

x	y	¬x	x ∧ y	x ∨ y
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Variables and function designators entered as bit primaries must have been declared of type bits. If a bit sequence to be operated upon consists of fewer bits than the fixed amount mentioned in 5.1.2., then it is extended by inserting an appropriate number of 0's after the b heading the bit sequence.

6.6. String expressions

6.6.1. Syntax

<string expression> ::= <simple string expression> |
        <if clause><string expression> else <string expression> |
        <case clause> (<string expression list>)
<string expression list> ::= <string expression> |
        <string expression list>, <string expression>
<simple string expression> ::= <string primary> |
        <simple string expression> conc <string primary>
<string primary> ::= <string> |<variable> |
        <function designator> |(<string expression>)

6.6.2. Semantics

A string expression is a rule for computing a string (sequence of characters). Variables and function designators entered as string primaries must have been declared of type string.

6.6.2.1. The operator conc (concatenate) yields the string consisting of the sequence of characters resulting from evaluation of the first operand after omitting the final string quote, followed by the sequence of characters resulting from evaluation of the second operand after omitting the leading string quote, mathematically defined as

"<sequence-1>" conc "<sequence-2>" = "<sequence-1><sequence-2>"

6.7. Tree expressions

6.7.1. Syntax

<tree expression> ::= <simple tree expression> |
        <if clause><tree expression> else <tree expression> |
        <case clause> (<tree expression list>)
<tree expression list> ::= <tree expression> |
        <tree expression list>, <tree expression>
<simple tree expression> ::= <tree primary> |
        <simple tree expression> conc <tree primary>
<tree primary> ::= <tree> |<variable> |<function designator> |
        (<tree expression>)
<tree> ::= [<expression list>] |[ ] |<tree><subscript>
<expression list> ::= <expression> |<expression list>, <expression>

6.7.2. Semantics

A tree expression is a rule for computing a tree. Variables and function designators entered as tree primaries must have been declared of type tree. The value of a tree is obtained by evaluating all expressions of its expression list. The order in which the expressions are evaluated is not specified. The value of the tree is then the ordered set of values (taken in the order of the denotation) of the results of the evaluated expressions.

A tree followed by a subscript denotes the value of the expression of the expression list of that tree, whose ordinal number is the current value of the arithmetic expression in the subscript.

7. Statements

A statement is said to denote a unit of action. By the execution of a statement is meant the performance of this unit of action which may consist of smaller units of actions such as the evaluation of expressions or the execution of other statements.

A statement which contains no symbols denotes no action.

After the execution of a statement (not being a goto statement) has been terminated, the next statement in lexicographical sequence will be executed.

Statements may in certain instances be labelled by preceding them with a label definition. The label identifier of the label definition is said to label that statement and to stand as that label.

Syntax:

<statement> ::= <unlabelled statement> |<label definition><statement>
<label definition> ::= <label identifier> :
<unlabelled statement> ::= <basic statement> |<prefixed statement>
<basic statement> ::= <simple statement> |<closed if statement>
<simple statement> ::= <block> |<assignment statement> | |
<procedure statement> |<goto statement>
<prefixed statement> ::= <open if statement> |<case statement> |
<iterative statement>
<program> ::= <statement>

7.1. Blocks

7.1.1. Syntax

<block> ::= <blockbody><statement> end
<blockbody> ::= <blockhead> |<blockbody><statement> ;
<blockhead> ::= begin |<blockhead><declaration> ;

7.1.2. Semantics

Every block introduces a new level of nomenclature. This is realized by execution of the block in the following steps:

Step 1: If an identifier defined within the block is already defined at the place from where the block is entered, then every occurence of that identifier within the block is systematically replaced by another identifier, which is defined neither within the block nor at the place from where the block is entered.
Step 2: If the declarations of the block contain array bound expressions, then these expressions are evaluated.
Step 3: Execution of the statements contained in the blockbody begins with the execution of the first statement following the blockhead.

7.2. Assignment statements

7.2.1. Syntax

<assignment statement> ::= <left part list><expression>
<left part list> ::= <left part> |<left part><left part list>
<left part> ::= <variable> :=

7.2.2. Semantics

The execution of assignment statements causes the assignment of the value of the expression to one or several variables. The assignment is performed after the evaluation of the expression. The types of all left part variables must be assignment compatible with the type of the expression.
The type of a left part variable is said to be assignment compatible with the type of the expression, if either
i) the two types are identical, or
ii) the left part variable is of type real, and the expression is of type integer, or
iii) the left part variable is of type complex, and the expression is of type real or integer, or
iv) the left part variable is an element of a tree variable, in which case its type becomes equal to the type of the expression.

7.3. Procedure Statements

7.3.1. Syntax

<procedure statement> ::= <procedure identifier> |
<procedure identifier> (<actual parameter list>)
<actual parameter list> ::= <actual parameter> |
<actual parameter list>, <actual parameter>
<actual parameter> ::= <value parameter> |'<name parameter>'
<value parameter> ::= <expression>
<name parameter> ::= <expression> |<statement> |
<array identifier> |<procedure identifier>

7.3.2. Semantics

The execution of a procedure statement is equivalent with a process performed in the following steps:

Step 1: Copies are taken of the body of the proper procedure whose procedure identifier is given by the procedure statement and of the actual parameters of the latter.

Step 2: If the procedure body is a block, then a systematic change of identifiers in its copy is performed as specified by step 1 of 7.1.2.

Step 3: The copies of the actual parameters are treated in an undefined order as follows:

i) if it is a name parameter enclosed in apostrophes then the apostrophes are dropped, and if it is then an expression different from a variable it is enclosed by a pair of parentheses, or if it is then a statement it is enclosed by the brackets begin and end,

ii) if it is a value parameter, it is evaluated and replaced by a possible reference denotation of the result.

Step 4: In the copy of the procedure body every occurrence of an identifier identifying a formal parameter is replaced by the copy of the corresponding actual parameter (cf. 7.3.2.1.).

In order for the process to be defined, these replacements must lead to correct ALGOL expressions and statements. Moreover, if the identifier identifying a formal parameter occurs in an expression, then the specified type of that formal parameter must be assignment compatible with the type of the actual parameter, and if it occurs as a left part variable of an assignment statement, then the type of the actual parameter must be assignment compatible with the specified type of the formal parameter.

Step 5: The copy of the procedure body, modified as indicated in steps 2-4, is executed as if it were written at the position of the procedure statement.

7.3.2.1. Actual-formal correspondence.
The correspondence between the actual parameters and the formal parameters is established as follows: The actual parameter list of the procedure statement (or of the function designator) 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.

7.3.2.2. Formal specifications.
If the specified type of a formal parameter and the type of the substituted actual parameter are not identical, then the operations indicated by the procedure body will be executed as if applying to operands of the type specified for the formal parameter. If an actual parameter is a statement enclosed in quotes, then the specification of its corresponding formal parameter must be procedure.

7.4. Goto Statements

7.4.1. Syntax

<goto statement> ::= goto <label identifier>

7.4.2. Semantics

An identifier is called a label identifier, if it stands as a label.
A goto statement defines its successor in execution explicitly as the statement whose label is represented by the label identifier. The determination of this statement is accomplished in the following steps:

Step 1: If some statement of the most recently activated and not yet terminated block is labelled with the given label identifier, then this is the statement designated as successor.
Otherwise
Step 2: the execution of that block is considered as terminated and Step 1 is taken as specified above.

7.5. If statements

7.5.1. Syntax

<open if statement> ::= <if clause><unlabelled statement>
<closed if statement> ::=
<if clause><basic statement> else <unlabelled statement>
<if clause> ::= if <logical expression> then

7.5.2. Semantics

The if statements are either open or closed if statements and their execution causes certain statements to be executed or skipped depending on the values of specified logical expressions.
The open if statement is executed in the following steps:
Step 1: The logical expression in the clause is evaluated.
Step 2: If the result of step 1 is true, then the statement following the clause is executed. Otherwise step 2 causes no action to be taken at all.
The closed if statement is executed in the following steps:
Step 1: The logical expression in the if clause is evaluated.
Step 2: If the result of step 1 is true, then the basic statement following the if clause is executed. Otherwise the statement following else is executed.

7.6. Case statements

7.6.1. Syntax

<case statement> ::= <case clause> begin <statement list> end
<statement list> ::= <statement> |<statement list> ; <statement>
<case clause> ::= case <arithmetic expression> of

7.6.2. Semantics

The execution of a case statement proceeds in the following steps:
Step 1: The expression of the case clause is evaluated.
Step 2: The statement whose ordinal number in the case statement list is equal to the value obtained in Step 1 is executed.
In order that the case statement is defined, the case clause must contain

an expression of type integer, whose current value is the ordinal number of some statement of the statement list.

7.7. Iterative statements

7.7.1. Syntax

<iterative statement> ::= <over clause><unlabelled statement> |
<while clause><unlabelled statement>

<over clause> ::= over <unsubscripted variable> do
<arithmetic expression> times

<while clause> ::= while <logical expression> do

7.7.2. Semantics

The iterative statement serves to express that an unlabelled statement be executed repeatedly depending on certain conditions specified by an over clause or a while clause. The unsubscripted variable and the arithmetic expression of the over clause must both be of type integer. The meaning of an iterative statement is explained in terms of equivalent ALGOL statements by which the iterative statement can be replaced.

over <var> do <arithmetic expression> times <statement>
is equivalent to
begin integer n; n:= <arithmetic expression> ; <var>:= 1;
1: if <var> ≤ n then
begin <statement> ; <var>:= <var> + 1; goto 1 end
end

while <logical expression> do <statement>
is equivalent to
begin 1: if <logical expression> then
begin <statement> ; goto 1 end
end

It is understood that in these substitutions the auxiliary identifiers, here 1 and n, are chosen different from any other identifier valid at the position of the iterative statement.

8. Standard procedures

As indicated in the Introduction, an ALGOL program may be thought as being embedded in an environment to be specified by the system which executes the ALGOL program, and which may be described in a language other than ALGOL.
Certain quantities referred to by the program may be defined in that environment. Subsequently some examples of procedures are given which are assumed to be defined in every environment. Their body is replaced by a comment explaining the effect of the activation of the procedure.

```
integer procedure round(x); real x; ...
integer procedure truncate(x); real x; ...
integer procedure order(a); array a;
comment yields the order of the array a;
integer procedure lowerbound(a,i); integer i; array a;
comment yields the lower subscript bound in the i-th dimension of the array
a;
integer procedure upperbound(a,i); integer i; array a; ...
integer procedure width(t); tree t; ...
comment yields the number of elements (simple values or subtrees) in the
tree t;
real procedure sin(x); real x; ...
real procedure cos(x); real x; ...
real procedure sqrt(x); real x; ...
real procedure exp(x); real x; ...
real procedure ln(x); real x; ...
real procedure arctan(x); real x; ...
real procedure realpart(x); complex x; ...
real procedure imagpart(x); complex x; ...
complex procedure complex(x,y); real x, y;
comment yields number so that realpart(complex(x,y)) = x and
imagpart(complex(x,y)) = y;
logical procedure even(x); integer x; ...
logical procedure odd(x); integer x; ...
```

logical procedure isinteger(t,i); integer i; tree t; ...
comment yields true, if the i-th element of the tree t is of type integer, false otherwise.
Analogously defined are: isreal, iscomplex, islogical, isbits, isstring, istree, isarray;
bits procedure rshift(b,i); bits b; integer i;
comment shifts the bit sequence i positions to the right and fills the vacated positions at the left with 0's, the rightmost i bits are lost.
Analogously defined is lshift;
bits procedure rdshift(a,b,i); bits a, b; integer i;
comment concatenates the rightmost i bits of a with the leftmost n-i bits of b, where n is the fixed number of bits in a bit sequence.
Analogously defined is ldshift;
string procedure extraxt(s,i,l); integer i, l;
string s; comment yields the string consisting of the characters s[i], ..., s[i+l-1], if i+l-1 $\leq$ n, where n is the number of characters in s. If l $\leq$ 0, the length of the resulting string is 0, and if i+l-1 > n the length of the resulting string is n-i+1;
integer procedure length(s); string s;
comment yields the number of characters in the string s not counting the surrounding string quotes;
string procedure char(s,i); string s; integer s; extract(s,i,1);
tree procedure next(t); tree t;
comment yields the tree consisting of the tree t without its first element;
procedure delete(t,i); tree t; integer i;
comment deletes the i'th element of the tree t;
integer procedure available space; ...
procedure inchar(channel,s); integer channel; string s; ...
procedure outchar(channel,s); integer channel; string s; ...
procedure overflow(p); procedure p;
comment if in subsequent computations an overflow condition occurs, then the computation is interrupted and the procedure p is executed;