Programming languages — ALGOL 60

Langages de programmation  ALGOL 60

First edition — 1984-10-15
Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Every member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work.

Draft International Standards adopted by the technical committees are circulated to the member bodies for approval before their acceptance as International Standards by the ISO Council. They are approved in accordance with ISO procedures requiring at least 75 % approval by the member bodies voting.

International Standard ISO 1538 was prepared by Technical Committee ISO/TC 97, Information processing systems.

This International Standard replaces ISO/R 1538 (withdrawn in 1977) of which it constitutes a revision.

ISO Recommendation 1538 was a compilation of several source documents. The basic one (developed under the auspices of the International Federation for Information Processing (IFIP), whose contributions are acknowledged) was the Revised Report on the Algorithmic Language ALGOL 60.

The text presented in this International Standard is based on the Modified Report on the Algorithmic Language ALGOL 60, which is a minor technical revision and a textual clarification of the Revised Report, as established by IFIP. For reasons of ISO editorial policy the original introduction which is irrelevant to an International Standard has been deleted and some introductory clauses have been added instead.
Programming Languages — ALGOL 60

0 Introduction
In this International Standard consistent use is made of ALGOL 60 as the name of the language, rather than just ALGOL, in order to avoid confusion with ALGOL 68 which is a completely different language. It is recommended that the language defined in this International Standard be referred to as STANDARD ALGOL 60.

Whenever the name ALGOL is used in this International Standard it is to mean ALGOL 60, not ALGOL 68, unless it is clear from the context that no specific language is indicated.

1 Scope and field of application
This International Standard defines the algorithmic programming language ALGOL 60. Its purpose is to facilitate interchange and promote portability of ALGOL 60 programs between data processing systems.

ALGOL 60 is intended for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

This International Standard specifies:

a) the syntax and semantics of ALGOL 60;

b) characteristics of programs written in ALGOL 60, and of implementations of that language, required for conformance to this International Standard.

This International Standard does not specify:

a) results of processes or other issues, that are, explicitly, left undefined or said to be undefined;

b) questions of hardware representation (these may be the subject of another International Standard), or of implementation;

c) the way non-valid programs are to be rejected, and how this will be reported;

d) requirements and rules for executing programs on an actual data processing system.

2 Reference
ISO/TR 1672, Hardware representation of ALGOL basic symbols in the ISO 7-bit coded character set for information processing interchange.

3 Definitions
For the purpose of this International Standard the following definitions apply:

3.1 valid program: A text written in the ALGOL 60 language that conforms to the rules for a program defined in this International Standard.

3.2 non-valid program: A text that does not conform, but was intended to be a program.

3.3 processor: A compiler, translator or interpreter, in combination with a data processing system, that accepts an intended program, transcribed in a form that can be processed by that data processing system, reports whether the intended program is valid or not, and if valid executes it, if that is being requested.

3.4 implementation: A processor, accompanied with documents that describe

a) its purpose, and the environment (hardware and software) in which it will work;

b) its intended properties, including

— the particular hardware representation of the language, as chosen;

— the actions taken, when results or issues occur that are undefined in this International Standard;

— conventions for issues said to be a question of implementation;
c) with regard to the implemented language, all differences from, restrictions to, or extensions to the language defined in this International Standard;
d) its logical structure;
e) the way to put it into use.

3.5 conforming implementation: An implementation conforming to this International Standard by accepting valid programs as being valid, by rejecting non-valid programs as being non-valid and by executing valid programs in accordance with the given rules.

3.6 implemented language: The version of the language as defined by the implementation.

3.7 conforming language version: A version of the language, defined by a conforming implementation that
a) does not contain any rule conflicting with those defined in this International Standard;
b) does not contain any rule not provided for in this International Standard, except such rules as, either said to be intentionally and explicitly a question of implementation, or otherwise being outside the scope of this International Standard.

3.8 extension: A rule in the implemented language that
a) is not given in this International Standard;
b) does not cause any ambiguity when added to this International Standard (but may serve to remove a restriction);
c) is within the scope of this International Standard.

4 Conformance

4.1 Requirements

Conformance to this International Standard requires
a) for a program, that it shall be a valid program;
b) for an implementation, that it shall be a conforming implementation;
c) for the implemented language, that it shall be a conforming language version.

4.2 Quantitative restrictions

The requirements specified in 4.1 shall allow for quantitative restrictions to rules stated or implied as having no such restriction in this International Standard, but only if they are fully described in the documents with the implementation.

4.3 Extensions

An implementation that allows for extensions in the implemented language is considered to conform to this International Standard, notwithstanding 4.1, if
a) it would conform when the extensions were omitted;
b) the extensions are clearly described with the implementation;
c) while accepting programs that are non-valid according to the rules given in clause 6 of this International Standard, it provides means for indicating which part, or parts, of a program would have led to its rejection, had no extension been allowed.

Valid programs using extensions shall be described as "conforming to ISO 1538 but for the following indicated parts".

4.4 Subsets

Conformance to a subset specified in this International Standard means conformance to the subset rules as if they were the only rules in the language.

5 Tests

Whether an implementation is a conforming implementation or the implemented language is a conforming language version may be decided by a sequence of test programs. If there is any uncertainty or doubt regarding acceptance of these programs then the conclusions drawn from the actual behaviour of the processor will prevail over those derived from its accompanying documents.

6 Description of the reference language

The detailed description of the reference language given herein reproduces, without modification, the text taken from the Modified Report (see the foreword), the contents of which are the following:

1 Structure of the language
   1.1 Formalism for syntactic description
2 Basic symbols, identifiers, numbers, and strings. Basic concepts
   2.1 Letters
   2.2 Digits and logical values
   2.3 Delimiters
   2.4 Identifiers
   2.5 Numbers
   2.6 Strings
2.7 Quantities, kinds and scopes
2.8 Values and types
3 Expressions
3.1 Variables
3.2 Function designators
3.3 Arithmetic expressions
3.4 Boolean expressions
3.5 Designational expressions
4 Statements
4.1 Compound statements and blocks
4.2 Assignment statements
4.3 Go to statements
4.4 Dummy statements
4.5 Conditional statements
4.6 For statements
4.7 Procedure statements
5 Declarations
5.1 Type declarations
5.2 Array declarations
5.3 Switch declarations
5.4 Procedure declarations
Appendix 1 — Subsets
Appendix 2 — The environmental block
Bibliography
Alphabetic index of definitions of concepts and syntactic units

1. Structure of the language
The algorithmic language has two different kinds of representation—reference and hardware—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 other representations may differ. Structure and content must be the same for all representations.

Reference language
1. It is the defining language.
2. The characters are determined by ease of mutual understanding and not by any computer limitations, coder's notation, or pure mathematical notation.
3. It is the basic reference and guide for compiler builders.
4. It is the guide for all hardware representations.

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

It should be particularly noted that throughout the reference language underlining in typescript or manuscript, or boldface type in printed copy, is used to represent certain 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. In the reference language underlining or boldface is used for no other purpose.

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 non-arithmetic statements and statement clauses are added which may describe, e.g. alternatives, or iterative repetitions of computing statements. Since it is sometimes necessary for the function of these statements that one statement refers 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 a compound statement that is contained only within a fictitious block (always assumed to be present and called the environmental block), and that makes no use of statements or declarations not contained within itself, except that it may invoke such procedure identifiers and function designators as may be assumed to be declared in the environmental block.

The environmental block contains procedure declarations of standard functions, input and output operations, and possibly other
operations to be made available without declaration within the program. It also contains the fictitious declaration, and initialisation, of own variables (see Section 5).

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

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.

1.1. Formalism for syntactic description

The syntax will be described with the aid of metalinguistic formulae (Backus, 1959). Their interpretation is best explained by an example:

\[ \langle ab \rangle ::= \langle a \rangle \langle b \rangle \]

Sequences of characters enclosed in the brackets \( \langle \rangle \) represent metalinguistic variables whose values are sequences of symbols. The marks \( ::= \) and \( | \) (the latter with the meaning of 'or') are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable \( \langle ab \rangle \). It indicates that \( \langle ab \rangle \) may have the value (or \( | \) that given some legitimate value of \( \langle ab \rangle \), another may be formed by following it with the character \( ( \) or by following it with some value of the variable \( \langle d \rangle \). If the values of \( \langle d \rangle \) are the decimal digits, some values of \( \langle ab \rangle \) are:

\[ \{(1137)\} \quad (12345) \quad \{( \} \]

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets \( \langle \rangle \) 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:

\( \langle \emptyset \rangle ::= \) (i.e. the null string of symbols).

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

The reference language is built up from the following basic symbols:

\( \langle \text{basic symbol} \rangle ::= \langle \text{letter} \rangle \langle \text{digit} \rangle \langle \text{logical value} \rangle \langle \text{delimiter} \rangle \)

2.1. Letters

\( \langle \text{letter} \rangle ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | w | x | y | z \)

\( A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z \)

This alphabet may be arbitrarily 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 (see Sections 2.4 Identifiers, 2.6 Strings). Within this report the letters (from an extended alphabet) \( I, \delta, \Sigma \) and \( \Omega \) are sometimes used and are understood as not being available to the programmer. If an extended alphabet is in use, that does include any of these letters, then their uses within this report must be systematically changed to other letters that the extended alphabet does not include.

2.2. Digits and logical values

2.2.1. Digits

\( \langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \)

Digits are used for forming numbers, identifiers, and strings.

2.2.2. Logical values

\( \langle \text{logical value} \rangle ::= \text{true} | \text{false} \)

The logical values have a fixed obvious meaning.

2.3. Delimiters

\( \langle \text{delimiter} \rangle ::= \langle \text{operator} \rangle | \langle \text{separator} \rangle | \langle \text{bracket} \rangle | \langle \text{declarator} \rangle | \langle \text{specifier} \rangle \)

\( \langle \text{operator} \rangle ::= \langle \text{arithmetic operator} \rangle | \langle \text{relational operator} \rangle | \langle \text{logical operator} \rangle | \langle \text{sequential operator} \rangle \)

\( \langle \text{arithmetic operator} \rangle ::= + | - | / | * | \frac{1}{2} \)

\( \langle \text{relational operator} \rangle ::= < | = | > | \neq \)

\( \langle \text{logical operator} \rangle ::= \mid, \wedge, \vee, \neg \)

\( \langle \text{sequential operator} \rangle ::= \text{if} | \text{then} | \text{else} | \text{for} | \text{do} \)

\( \langle \text{separator} \rangle ::= , | ; | : | = | \text{step} | \text{until} | \text{while} | \text{comment} \)

\( \langle \text{bracket} \rangle ::= \{ | \} | [ | ] \)

\( \langle \text{declarator} \rangle ::= \text{func} | \text{variable} | \text{array} | \text{switch} | \text{procedure} | \langle \text{specifier} \rangle \)

\( \langle \text{specifier} \rangle ::= \text{string} | \text{label} | \text{value} \)

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 a program the following 'comment' conventions hold:

The sequence

\( ; \text{comment} \langle \text{any sequence of zero or more characters not containing} \rangle ; \)

\( \text{begin} \text{comment} \langle \text{any sequence of zero or more characters not containing} \rangle ; \text{begin} \)

\( \text{end} \langle \text{any sequence of zero or more basic symbols not containing} \rangle \text{end} \)

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

\( \langle \text{identifier} \rangle ::= \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{digit} \rangle \)

2.4.2. Examples

\( q, \) \( Soup, \) \( V17a, \) \( a3kTMN, \) \( MARILYN \)

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. Identifiers also act as formal parameters of procedures, in which capacity they may represent any of the above entities, or a string.

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 (see Section 2.7 Quantities, kinds and scopes and Section 5 Declarations). This rule applies also to the formal parameters of procedures, whether representing a quantity or a string.

2.5. Numbers

2.5.1. Syntax

\( \langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle | \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle \)

\( \langle \text{integer} \rangle ::= \langle \text{unsigned integer} \rangle | + \langle \text{unsigned integer} \rangle | - \langle \text{unsigned integer} \rangle \)

\( \langle \text{decimal fraction} \rangle ::= \langle \text{unsigned integer} \rangle | \langle \text{unsigned integer} \rangle \langle \text{decimal fraction} \rangle \)

\( \langle \text{exponent part} \rangle ::= 10^\langle \text{integer} \rangle \)

\( \langle \text{decimal number} \rangle ::= \langle \text{unsigned integer} \rangle | \langle \text{decimal fraction} \rangle | \langle \text{exponent part} \rangle \)

\( \langle \text{number} \rangle ::= \langle \text{unsigned number} \rangle | + \langle \text{unsigned number} \rangle | - \langle \text{unsigned number} \rangle \)
2.5.2. Examples

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-200.085</td>
<td>-0.08320 - 02</td>
</tr>
<tr>
<td>177</td>
<td>+0.743108</td>
<td>-107</td>
</tr>
<tr>
<td>.5384</td>
<td>9.3410 + 10</td>
<td>10 - 4</td>
</tr>
<tr>
<td>210</td>
<td>4 + 10 + 5</td>
<td></td>
</tr>
</tbody>
</table>

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 integer type. All other numbers are of real type (see Section 5.1 Type declarations).

2.6. Strings

2.6.1. Syntax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(proper string) ::=</td>
<td>(any sequence of characters not containing &quot;or&quot;)</td>
</tr>
<tr>
<td>(open string) ::=</td>
<td>(proper string)</td>
</tr>
<tr>
<td>(closed string) ::=</td>
<td>(open string)</td>
</tr>
<tr>
<td>(string) ::=</td>
<td>(closed string)</td>
</tr>
</tbody>
</table>

2.6.2. Examples

```
5k3 - "["it ["\"= \".2"]"]"
"This_ is _all"
"_one_string"
```

2.6.3. Semantics

In order to enable the language to handle sequences of characters the string quotes " and " are introduced.

The characters available within a string are a question of hardware representation, and further rules are not given in the reference language. However, it is recommended that visible characters, other than " and ", should represent themselves, while invisible characters other than space should not occur within a string. To conform with ISO/TR 1672, a space may stand for itself, although in this document the character \ is used to represent a space.

To allow invisible, or other exceptional characters to be used, they are represented within either matching string quotes or a matched pair of the " symbol. The rules within such an inner string are unspecified, so if such an escape mechanism is used a comment is necessary to explain the meaning of the escape sequence.

A string of the form "closed string" behaves as if it were the string formed by deleting the closing string quote of the closed string and the opening string quote of the following string (together with any layout characters between them).

Strings are used as actual parameters of procedures (see 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 (see 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, labels, switch 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(variable identifier) ::=</td>
<td>(identifier)</td>
</tr>
<tr>
<td>(simple variable) ::=</td>
<td>(variable identifier)</td>
</tr>
<tr>
<td>(subscript expression) ::=</td>
<td>(arithmetic expression)</td>
</tr>
<tr>
<td>(subscript list) ::=</td>
<td>(subscript expression)(subscript list)</td>
</tr>
<tr>
<td>(subscript expression) ::=</td>
<td>(identifier)</td>
</tr>
<tr>
<td>(subscripted variable) ::=</td>
<td>(array identifier)(subscript list)</td>
</tr>
<tr>
<td>(variable) ::=</td>
<td>(simple variable)(subscripted variable)</td>
</tr>
</tbody>
</table>

3.1.2. Examples

```
epsilon
detA
a17
Q(7, 2)
x[sin(A x 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 (see Section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (see Section 5.1 Type declarations) or for the corresponding array identifier (see Section 5.2 Array declarations).

3.1.4. Subscripts

3.1.4.1. Subscripted variables designate values which are components of multidimensional arrays (see 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 (see Section 3.3 Arithmetic expressions).

3.1.4.2. Each subscript position acts like a variable of integer type and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (see 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 (see Section 5.2 Array declarations).

3.1.5. Initial values of variables

The value of a variable, not declared own, is undefined from entry into the block in which it is declared until an assignment is made to it. The value of a variable declared own is zero (if arithmetic) or false (if Boolean) on first entry to the block in which it is declared. On subsequent entries it has the same value as at the preceding exit from the block.

3.2. Function designators

3.2.1. Syntax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(procedure identifier) ::=</td>
<td>(identifier)</td>
</tr>
<tr>
<td>(actual parameter) ::=</td>
<td>(string)(expression)</td>
</tr>
<tr>
<td>(array identifier) ::=</td>
<td>(switch identifier)</td>
</tr>
<tr>
<td>(procedure identifier) ::=</td>
<td>(letter string)(letter string)(letter)</td>
</tr>
<tr>
<td>(letter string) ::=</td>
<td>(identifier)(letter string)</td>
</tr>
<tr>
<td>(parameter delimiter) ::=</td>
<td>(identifier)(letter string)</td>
</tr>
<tr>
<td>(actual parameter list) ::=</td>
<td>(actual parameter)</td>
</tr>
<tr>
<td>(actual parameter) ::=</td>
<td>(actual parameter list)</td>
</tr>
<tr>
<td>(function designator) ::=</td>
<td>(procedure identifier)</td>
</tr>
</tbody>
</table>

```
(actual parameter part) ::= (empty)(actual parameter list)
```

ISO 1538-1984 (E)
3.2.2. Examples
\[ \sin(a - b) \]
\[ J(v + z, a) \]
\[ R \]
\[ S(s - 5) \]
\[ \text{Temperature} : (T) \] Pressure : (P) \]
\[ \text{Compile } (\text{]:= 3) \]
\[ \text{Stack} : (Q) \]

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 (see 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 rules for determining the value of a function designator.

3.2.4. Standard functions and procedures
Certain standard functions and procedures are declared in the environmental block with the following procedure identifiers:

- abs, labs, sign, entier, sqrt, sin, cos, arctan, ln, exp, inchar, outchar, length, outstring, outterminator, stop, fault, integer, outinteger, inreald, outreald, maxreal, minreal, and epsilon.

For details of these functions and procedures, see the specification of the environmental block given as Appendix 2.

3.3. Arithmetic expressions
3.3.1. Syntax

- Adding operator: \( + \)
- Multiplying operator: \( \times \)
- Primary: (unsigned number, variable)

- Factor: (primary, function designator, arithmetic expression)

- Term: (factor, multiplying operator, factor)

- Simple arithmetic expression: (term, adding operator, term)

- If clause: if (Boolean expression) then (simple arithmetic expression) else (arithmetic expression)

3.3.2. Examples

**Primitives:**
\[ 7.394_{10} - 8 \]
\[ \text{sum} \]
\[ w[i + 2, 8] \]
\[ \cos(y + z \times 3) \]
\[ (a - 3)y + w[i + 8] \]

**Factors:**
\[ \text{omega} \]
\[ \text{sum} \cos(y + z \times 3) \]
\[ 7.394_{10} - 8iw[i + 2, 8] \]

**Terms:**
\[ U \]
\[ \text{omega} \times \text{sum} \cos(y + z \times 3) / 7.394_{10} - 8 \]
\[ w[i + 2, 8] (a - 3y + w[i + 8]) \]

Simple arithmetic expression:
\[ U - Yu + \text{omega} \times \text{sum} \cos(y + z \times 3) / 7.394_{10} - 8 \]
\[ w[i + 2, 8] (a - 3y + w[i + 8]) \]

Arithmetic expressions:
\[ w \times u - Q(S + Co) / 2 \]
if \( q > 0 \) then \( S + 3 \times Q / 2 \)
else \( 2 \times S + 3 \times q \)
if \( a < 0 \) then \( U + V \)
else if \( a > b \) then \( U/V \)
else if \( k \neq y \) then \( V/U \)
else 0
\[ a \times \sin(\text{omega} \times t) \]
\[ 0.571012 \times a[N \times (N - 1)/2, 0] \]
\[ (A \times \arctan(y) + Z)/7(7 + Q) \]
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 the 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 (see 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 (see 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 longest arithmetic expression found in this position is understood). If none of the Boolean expressions has the value true, then the value of the arithmetic expression is the value of the expression following the final else.

The order of evaluation of primaries within an expression is not defined. If different orders of evaluation would produce different results, due to the action of side effects of function designators, then the program is undefined.

In evaluating an arithmetic expression, it is understood that all the primaries within that expression are evaluated, except those within any arithmetic expression that is governed by an if clause but not selected by it. In the special case where an exit is made from a function designator by means of a go to statement (see Section 5.4.4), the evaluation of the expression is abandoned, when the go to statement is executed.

3.3.4. Operators and types
Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of real or integer types (see 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 \( \langle \text{term} \rangle \langle \text{factor} \rangle \) and \( \langle \text{term} \rangle \div \langle \text{factor} \rangle \) both denote division. The operations are undefined if the factor has the value zero, but are otherwise to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (see Section 3.3.5). Thus for example:

\[ a/b \times ((p - q) \times v/s) \]

The operator \( \div \) is defined for all four combinations of real and integer types and will yield results of real type in any case. The operator \( \div \) is defined only for two operands both of integer type and will yield a result of integer type. If \( a \) and \( b \) are of integer type, then the value of \( a \div b \) is given by the function:

- integer procedure \( \text{div}(a, b); \text{value} a, b; \)
- integer \( a, b; \)
if \( b = 0 \) then
\[ \text{fault}(\text{div}, \ldots, \ldots, \ldots, \text{zero}^n, a) \]
else
- begin integer \( a, r; \)
\[ q := 0; r := \text{labs}(a); \]
for \( r := r \text{ labs}(b) \) while \( r \geq 0 \) do \( q := q + 1; \)
\[ \text{div} := \text{if} \ a < 0 \text{ or } b > 0 \text{ then } -q \text{ else } q \]
end \( \text{div} \)

3.3.4.3. The operation \( \langle \text{factor} \rangle \langle \text{primary} \rangle \) denotes exponentiation, where the factor is the base and the primary is the exponent. Thus
for example
\[ 2^n \cdot k \text{ means } (2^n)^k \]
while
\[ 2^n \cdot m \text{ means } (2^n)^m. \]
If \( r \) is of real type and \( x \) of either real or integer type, then the value of \( x/r \) is given by the function:
real procedure exp(x, r); value x, r;
real x, r;
if \( x > 0.0 \) then
  expr := expr \times \ln(x)
else if \( x = 0.0 \) then
  expr := 0.0
else
  fault("expr...undefined, x")

The relational operators are: \(<\), \(\leq\), \(\geq\), \(>\), \(\neq\). The syntax given in Section 3.3.6 of using the exponentiation operator may be different from those of using the procedures \text{exp} and \text{expn}.

3.3.4.4. Type of a conditional expression
The type of an arithmetic expression of the form
\[ \text{if } B \text{ then } SAE \text{ else } AE \]
does not depend upon the value of \( B \). The expression is of real type if either \( SAE \) or \( AE \) is real and is of integer type otherwise.

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: \( \neg \)
- second: \( / \), \( \div \)
- 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 real type 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 implementations 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
relations and variables
3.4.5. The operators
3.4.4. Types
3.4.5. The operators
3.4.6. Precedence of operators
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

\[
\begin{align*}
\langle \text{label} \rangle &::= \langle \text{identifier} \rangle \\
\langle \text{switch identifier} \rangle &::= \langle \text{identifier} \rangle \\
\langle \text{switch designator} \rangle &::= \langle \text{switch identifier} \rangle (\langle \text{subscript expression} \rangle) \\
\langle \text{simple designational expression} \rangle &::= \langle \text{label} \rangle \\
\langle \text{designational expression} \rangle &::= \langle \text{simple designational expression} \rangle \\
\langle \text{if clause} \rangle &::= \langle \text{simple designational expression} \rangle \\
\langle \text{compound statement} \rangle &::= \langle \text{unlabelled compound} \rangle \\
\langle \text{block} \rangle &::= \langle \text{unlabelled block} \rangle \langle \text{label} \rangle \langle \text{block} \rangle \\
\langle \text{program} \rangle &::= \langle \text{block} \rangle \langle \text{compound statement} \rangle
\end{align*}
\]

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:**
  \[L: L' \ldots \begin{cases} S; & S; \ldots; S; \text{end} \\
  \text{Block:} & L: L' \ldots \begin{cases} D; & D; \ldots; D; \text{end} \end{cases} \]

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:**

\[
\begin{align*}
\text{begin} &::= \langle \text{integer expression} \rangle \\
\text{go to} &::= \langle \text{identifier} \rangle \\
\text{START.} \text{CONTINUE:} &::= \langle \text{real expression} \rangle
\end{align*}
\]

4.1.3. Semantics

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

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be non-local to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest enclosing block, i.e. the smallest block whose brackets begin and end enclose that statement.

A label is said to be implicitly declared in this block head, as distinct from the explicit declaration of all other local identifiers.

In this context a procedure body, or the statement following a for clause, must be considered as if it were enclosed by \text{begin} and \text{end} and treated as a block, this block being nested within the fictitious block of Section 4.7.3.1, in the case of a procedure with parameters by value. A label that is not within any block of the program (nor within a procedure body, or the statement following a for clause) is implicitly declared in the head of the environmental block.

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

4.2. Assignment statements

4.2.1. Syntax

\[
\begin{align*}
\langle \text{destination} \rangle &::= \langle \text{variable} \rangle \langle \text{procedure identifier} \rangle \\
\langle \text{left part} \rangle &::= \langle \text{destination} \rangle \\
\langle \text{left part list} \rangle &::= \langle \text{left part} \rangle \langle \text{left part list} \rangle \\
\langle \text{assignment statement} \rangle &::= \langle \text{left part list} \rangle \langle \text{arithmetic expression} \rangle \\
\langle \text{left part list} \rangle &::= \langle \text{left part} \rangle \langle \text{Boolean expression} \rangle
\end{align*}
\]
4.4.2. Examples

\[
\begin{align*}
    s &:= p[0] := n := n + 1 + s \\
    n &:= n + 1 \\
    A &:= B/C - \nu - q \times s \\
    S(v, k + 2) &:= 3 - \arctan(s \times \text{zeta}) \\
    V &:= Q > Y \land Z
\end{align*}
\]

4.4.3. Semantics

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

4.5. Conditional statements

4.5.1. Syntax

\[
\begin{align*}
    \langle \text{if clause} \rangle &::= \langle \text{Boolean expression} \rangle \rightarrow \langle \text{unconditional statement} \rangle \\
    \langle \text{Conditional statement} \rangle &::= \langle \text{if statement} \rangle \\
    \langle \text{Block} \rangle &::= \langle \text{Conditional statement} \rangle \\
    \langle \text{Compound statement} \rangle &::= \langle \text{Block} \rangle
\end{align*}
\]

4.5.2. Examples

- If \( x > 0 \) then \( n := n + 1 \)
- If \( y > x \) then \( v := q := n + m \) else go to \( R \)
- If \( s < 0 \) then \( \text{Transfer} \)

4.5.3. Semantics

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

4.5.3.1. If statement

An if statement is of the form

\[
\begin{align*}
    \text{If } B \text{ then } Su
\end{align*}
\]

where \( B \) is a Boolean expression and \( Su \) is an unconditional statement.

4.5.3.2. Conditional Statement

Three forms of unlabelled conditional statement exist, namely:

- If \( B \) then \( Su \)
- If \( B \) then \( S\text{for} \)
- If \( B \) then \( Su \text{ else } S \)

where \( Su \) is an unconditional statement, \( S\text{for} \) is a for statement and \( S \) is a statement.

The meaning of the first form is given in Section 4.5.3.1.
The second form is equivalent to

\[
\begin{align*}
    \text{if } B \text{ then begin } S\text{for end}
\end{align*}
\]

The third form is equivalent to

\[
\begin{align*}
    \text{begin if } B \text{ then begin } Su; \text{ go to } \Gamma \text{ end; } S; \Gamma \text{ end}
\end{align*}
\]

(For the use of \( \Gamma \) see Section 2.1 Letters.) If \( S \) is conditional, and also of this form, a different label must be used instead of \( \Gamma \) in following the same rule.

4.5.4. Go to into a conditional statement

The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the execution of a conditional statement.

4.6. For statements

4.6.1. Syntax

\[
\begin{align*}
    \langle \text{for list element} \rangle &::= \langle \text{arithmetic expression} \rangle \\
    \langle \text{for list} \rangle &::= \langle \text{arithmetic expression} \rangle \rightarrow \langle \text{arithmetic expression} \rangle \\
    \langle \text{while (Boolean expression)} \rangle &::= \langle \text{arithmetic expression} \rangle \\
    \langle \text{for list element} \rangle &::= \langle \text{for list} \rangle \langle \text{for list element} \rangle \\
    \langle \text{for clause} \rangle &::= \langle \text{variable identifier} \rangle \\
    \langle \text{for statement} \rangle &::= \langle \text{for clause} \rangle \langle \text{statement} \rangle \\
    \langle \text{label} \rangle &::= \langle \text{for statement} \rangle
\end{align*}
\]

4.6.2. Examples

\[
\begin{align*}
    L: \begin{align*}
        \text{begin } & \text{statements; } \text{John: end}
    \end{align*}
\end{align*}
\]
4.6.4. The for list elements
If the for list contains more than one element then
for \( V := X, Y \) do \( S \)
where \( X \) is a for list element, and \( Y \) is a for list (which may consist of one element or more), is equivalent to
begin
procedure \( A \); for \( V := X \) do \( A \); end

(For the use of \( \Sigma \) see Section 2.1 Letters.)

4.6.4.1. Arithmetic expression element
If \( X \) is an arithmetic expression then
for \( V := X \) do \( S \)
is equivalent to
begin
\( V := X; S \)
end
where \( S \) is treated as if it were a block (see Section 4.1.3).

4.6.4.2. Step-until element
If \( A, B \) and \( C \) are arithmetic expressions then
for \( V := A \) step \( B \) until \( C \) do \( S \)
is equivalent to
begin
\( \langle \text{type of } B \rangle \theta; \)
\( V := A; \)
\( \theta := B; \)
\( I: \text{if } (V - C) \times \text{sign}(\theta) \leq 0 \text{ then} \)
begin
\( S; \theta := B; V := V + \theta; \)
go to \( I \)
end

end
where \( S \) is treated as if it were a block (see Section 4.1.3).
In the above, \( \langle \text{type of } B \rangle \) must be replaced by \text{real} or \text{integer} according to the type of \( B \). (For the use of \( \theta \) and \( I \) see Section 2.1 Letters.)

4.6.4.3. While element
If \( E \) is an arithmetic expression and \( F \) a Boolean expression then
for \( V := E \) while \( F \) do \( S \)
is equivalent to
begin
\( \Gamma: V := E; \)
if \( F \) then
begin
\( S; \text{go to } \Gamma \)
end
end
where both \( S \) and the outermost compound statement of the above expansion are treated as if they were blocks (see Section 4.1.3). (For the use of \( \Gamma \) see Section 2.1 Letters.)

4.6.5. The value of the controlled variable upon exit
Upon exit from the for statement, either through a go to statement, or by exhaustion of the for list, the controlled variable retains the last value assigned to it.

4.6.6. Go to leading into a for statement
The statement following a for clause always acts like a block, whether it has the form of one or not. Consequently the scope of any label within this statement can never extend beyond the statement.

4.7. Procedure statements
4.7.1. Syntax
\[
\langle \text{actual parameter} \rangle ::= \langle \text{string} \rangle / \langle \text{expression} \rangle / \langle \text{array identifier} \rangle
\]
\[
\langle \text{switch identifier} \rangle / \langle \text{procedure identifier} \rangle
\]
\[
\langle \text{letter string} \rangle ::= \langle \text{letter} \rangle / \langle \text{letter string} \rangle / \langle \text{letter} \rangle
\]
\[
\langle \text{parameter delimiter} \rangle ::= \langle \text{list} \rangle / \langle \text{list} \rangle / \langle \text{letter string} \rangle
\]
\[
\langle \text{actual parameter list} \rangle ::= \langle \text{actual parameter} \rangle / \langle \text{actual parameter list} \rangle / \langle \text{parameter delimiter} \rangle / \langle \text{actual parameter} \rangle
\]
\[
\langle \text{actual parameter part} \rangle ::= \langle \text{empty} \rangle / \langle \text{actual parameter list} \rangle / \langle \text{parameter delimiter} \rangle / \langle \text{actual parameter part} \rangle
\]

4.7.2. Examples
\[
\text{Spur}(A) \text{ Order:}(7) \text{ Result to: } (V)
\]
\[
\text{Transpose}(W, v + 1)
\]
\[
\text{Absmax}(A, N, M, Y, v, K)
\]
\[
\text{Innerproduct}(A[i, P, a], B[P], 10, P, Y)
\]

These examples correspond to examples given in Section 5.4.2.

4.7.3. Semantics
A procedure statement serves to invoke (call for) the execution of a procedure body (see Section 5.4 Procedure declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. Value assignment (call by value)
All formal parameters quoted in the value part of the procedure heading (see Section 5.4) are assigned the values (see Section 2.8 Values and types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types given in the corresponding specifications (see Section 5.4.5). As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block (see Section 5.4.3).

4.7.3.2. Name replacement (call by name)
Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses if it is an expression but not a variable. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

If the actual and formal parameters are of different arithmetic types, then the appropriate type conversion must take place, irrespective of the context of use of the parameter.

4.7.3.3. Body replacement and execution
Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.7.4. Actual-formal correspondence
The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions
For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in Sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.
This imposes the restriction on any procedure statement that the
kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule, and some additional restrictions, are the following:

4.7.5.1. If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, see Section 4.7.8), then this string can only be used within the procedure as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

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

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

Similarly the number, kind and type of any parameters of a formal procedure parameter must be compatible with those of the actual parameter.

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

4.7.5.5. Restrictions imposed by specifications of formal parameters must be observed. The correspondence between actual and formal parameters should be in accordance with the following table.

<table>
<thead>
<tr>
<th>Formal parameter</th>
<th>Mode</th>
<th>Actual parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>value</td>
<td>arithmetic expression</td>
</tr>
<tr>
<td>integer</td>
<td>name</td>
<td>arithmetic expression (see 4.7.5.2)</td>
</tr>
<tr>
<td>Boolean</td>
<td>value</td>
<td>Boolean expression</td>
</tr>
<tr>
<td>Boolean</td>
<td>name</td>
<td>Boolean expression (see 4.7.5.2)</td>
</tr>
<tr>
<td>label</td>
<td>value</td>
<td>designational expression</td>
</tr>
<tr>
<td>integer array</td>
<td>value</td>
<td>arithmetic array</td>
</tr>
<tr>
<td>real array</td>
<td>value</td>
<td>integer array</td>
</tr>
<tr>
<td>Boolean array</td>
<td>value</td>
<td>Boolean array</td>
</tr>
<tr>
<td>typeless procedure</td>
<td>name</td>
<td>arithmetic procedure, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>typeless procedure, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boolean procedure (see 4.7.5.3)</td>
</tr>
</tbody>
</table>

4.7.5.6. Label parameters

A label may be called by value, even though variables of type label do not exist.

4.7.5.7. Parameter delimiters

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

4.7.8. Procedure body expressed in code

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

4.7.9. Standard procedures

Ten standard procedures are defined, which are declared in the environmental block in an identical manner to the standard functions. These procedures are: inchar, outchar, outstring, ininteger, inreal, outinteger, outreal, outterminator, fault and stop. The input/output procedures identify physical devices or files by means of channel numbers which appear as the first parameter. The method by which this identification is achieved is outside the scope of this report. Each channel is regarded as containing a sequence of characters, the basic method of accessing or assigning these characters being via the procedures inchar and outchar.

The procedures inreal and outreal are converses of each other in the sense that a channel containing characters from successive calls of outreal can be re-input by the same number of calls of inreal, but some accuracy may be lost. The procedures ininteger and outinteger are also a pair, but no accuracy can be lost. The procedure outterminator is called at the end of each of the procedures outreal and outinteger. Its action is machine dependent but it must ensure separation between successive output of numeric data.

Possible implementation of these additional procedures are given in Appendix 2 as examples to illustrate the environmental block.

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (see Section 4.1.3). Dynamically this implies the following: at the time of an entry into a block (through the begin since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through end, or by a go to statement) all identifiers which are declared for the block lose their local significance.

A declaration may be marked with the additional declarator own. This has the following effect: upon a reentry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined.

Apart from labels, formal parameters of procedure declarations, and identifiers declared in the environmental block, each identifier appearing in a program must be explicitly declared within the program.

No identifier may be declared either explicitly or implicitly (see Section 4.1.3) more than once in any one block head.

Syntax

(declaration) ::= (type declaration)(array declaration)
                | (switch declaration)(procedure declaration)

5.1. Type declarations

5.1.1. Syntax

(declaration) ::= (simple variable)(simple variable), (type list)

(declaration) ::= real | integer | Boolean |

(declaration) ::= (local or own) own

5.1.2. Examples

integer p, q, s

own Boolean Aeryl, n

5.1.3. Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only...
5.2. Array declarations

5.2.1. Syntax

- Low bound := <arithmetic expression>
- Upper bound := <arithmetic expression>
- Bound pair := <lower bound> :: <upper bound>
- Bound pair list := <bound pair> :: <bound pair list>
- Array segment := <array identifier> [bound pair]
- Array list := <array segment> :: <array list>
- Array declaration := <local or own> array <array declaration> :: <array list>

5.2.2. Examples

```plaintext
array a, b, c[7:n, 2:m], s[-2:10]
own integer array A[2:20]
real array q[-7: if c < 0 then 2 else 1]
```

5.2.3. Semantics

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

5.2.3.1. Subscript bounds

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

5.2.3.2. Dimensions

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

5.2.3.3. Types

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

5.2.4. Lower upper bound expressions

5.2.4.1. The expressions will be evaluated in the same way as subscript expressions (see Section 3.1.4.2).

5.2.4.2. The expressions cannot include any identifier that is declared, either explicitly or implicitly (see Section 4.1.3), in the same block head as the array in question. The bounds of an array declared as own may only be of the syntactic form integer (see Section 2.5.1).

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds. If any lower subscript bound is greater than the corresponding upper bound, the array has no component.

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

5.3. Switch declarations

5.3.1. Syntax

- Switch list := <designational expression>
- Switch list := <switch list>, <designational expression>
- Switch declaration := switch <switch identifier> := <switch list>

5.3.2. Examples

```plaintext
switch S := S1, S2, Q[n], if y > -5 then S3 else S4
switch Q := p1, w
```

5.3.3. Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, ..., obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (see Section 3.5 Designational expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4. Evaluation of expressions in the switch list

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

5.3.5. Influence of scopes

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4. Procedure declarations

5.4.1. Syntax

```plaintext
<identifier> := <parameter delimiter>
<parameter delimiter> := <parameter delimiter> :: <parameter delimiter>
<parameter delimiter> := <formal parameter> :: <parameter delimiter>
<formal parameter> := <identifier> <identifier list> 
<parameter list> := <identifier> <identifier list> 
<identifier list> := <identifier> <identifier list> 
<identifier list> := <identifier> 
<identifier list> := <empty>
<identifier> := <identifier> <identifier list> 
<identifier> := <type> <array declaration> label
<identifier> := <type> <identifier list> 
<identifier list> := <identifier> <identifier list> 
<identifier list> := <identifier> 
<identifier list> := <empty>
```

5.4.2. Examples (see also the examples in Appendix 2)

```plaintext
procedure Transpose(a) Order:(n); value n;
array a; integer n; real s;
begin integer k;
s := 0;
for k := 1 step 1 until n do s := s + a[k, k];
end
```

```plaintext
procedure Absmax(a) size:(n, m) Result:(x); value n, m;
array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a, of size n by m is transferred to y, and the subscripts of this element to i and k;
begin integer p, q;
y := 0; i := k := 1;
for p := 1 step 1 until n do
for q := 1 step 1 until m do
if abs(a[p, q]) > y then
```

5.4.3. Examples

```plaintext
integer procedure Step(a); value u; real u;
Step := if 0 < u and u < 1 then 1 else 0
```

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

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

5.4.3. Semantics
A procedure declaration serves to define the procedure associated
with a procedure identifier. The principal constituent of a procedure
declaration is a statement or a piece of code, the procedure body, which
through the use of procedure statements and/or function designators
may be activated from other parts of the block in the
head of which the procedure declaration appears. Associated with
the body is a heading, which specifies certain identifiers occurring
within the body to represent formal parameters. Formal parameters
in the procedure body will, whenever the procedure is activated (see
Section 4.7 Function designators and Section 4.7 Procedure state-
ments) be assigned the values of or replaced by actual parameters.
Identifiers in the procedure body which are not formal will be either
local or non-local to the body depending on whether they are
declared within the body or not. Those of them which are non-local
to the body may well be local to the block in the head of which the
procedure declaration appears. The procedure body always acts like
a block, whether it has the form of one or not. Consequently the
scope of any label labelling a statement within the body or the body
itself can never extend beyond the procedure body. In addition, if the
identifier of a formal parameter is declared anew within the
procedure body (including the case of its use as a label as in Section
4.1.3), it is thereby given a local significance and actual parameters
which correspond to it are inaccessible throughout the scope of this
inner local quantity.

No identifier may appear more than once in any one formal para-
meter list, nor may a formal parameter list contain the procedure
identifier of the same procedure heading.

5.4.4. Values of function designators
For a procedure declaration to define the value of a function design-
ator there must, within the procedure body, occur one or more uses
of the procedure identifier as a destination; at least one of these must
be executed, and the type associated with the procedure identifier
must be declared through the appearance of a type declarator as the
very first symbol of the procedure declaration. The last value so
assigned is used to continue the evaluation of the expression in which
the function designator occurs. Any occurrence of the procedure
identifier within the body of the procedure other than as a destina-
tion in an assignment statement denotes activation of the procedure.
If a go to statement within the procedure, or within any other
procedure activated by it, leads to an exit from the procedure, other
than through its end, then the execution, of all statements that have
been started but not yet completed and which do not contain the
label to which the go to statement leads, is abandoned. The values of
all variables that still have significance remain as they were immedi-
ately before execution of the go to statement.
If a function designator is used as a procedure statement, then the
resulting value is discarded, but such a statement may be used, if
desired, for the purpose of invoking side effects.

5.4.5. Specifications
The heading includes a specification part, giving information about
the kinds and types of the formal parameters. In this part no formal
parameter may occur more than once.

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

Appendix 1 Subsets
Two subsets of ALGOL 60 are defined, denoted as level 1 and level 2.
The full language is level 0.
The level 1 subset is defined as level 0 with the following additional
restrictions:
1. The own declarator is not included.
2. Additional restrictions are placed upon actual parameters as
given by the following replacement lines to the table in Section
4.7.5.5.

<table>
<thead>
<tr>
<th>Formal parameter</th>
<th>Mode</th>
<th>Actual parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>name</td>
<td>integer expression (see 4.7.5.2)</td>
</tr>
<tr>
<td>real</td>
<td>name</td>
<td>real expression (see 4.7.5.2)</td>
</tr>
<tr>
<td>integer array</td>
<td>name</td>
<td>integer array (see 4.7.5.3)</td>
</tr>
<tr>
<td>real array</td>
<td>name</td>
<td>real array (see 4.7.5.3)</td>
</tr>
<tr>
<td>typeless procedure</td>
<td>name</td>
<td>typeless procedure (see 4.7.5.3)</td>
</tr>
<tr>
<td>integer procedure</td>
<td>name</td>
<td>integer procedure (see 4.7.5.3)</td>
</tr>
<tr>
<td>real procedure</td>
<td>name</td>
<td>real procedure (see 4.7.5.3)</td>
</tr>
</tbody>
</table>
3. Only one alphabet of 26 letters is provided, which is regarded as
being the lower case alphabet of the reference language.
4. If deleting every symbol after the twelfth in every identifier would
change the action of the program, then the program is undefined.
The level 2 subset consists of restrictions 1-3 of level 1 and in addition:
5. Procedures may not be called recursively, either directly or
indirectly.
6. If a parameter is called by name, then the corresponding actual
parameter may only be an identifier or a string.
7. The designational expressions occurring in a switch list may only be
labels.
8. The specifiers switch, procedure and \langle type\rangle procedure are
not included.
9. A left part list may only be a left part.
10. If deleting every symbol after the sixth in every identifier would
change the action of the program, then the program is undefined.
Appendix 2 The environmental block

As an example of the use of ALGOL 60, the structure of the environmental block is given in detail.

begin

comment This description of the standard functions and procedures should be taken as definitive only so far as their effects are concerned. An actual implementation should seek to produce these effects in as efficient a manner as practicable. Furthermore, where arithmetics of real quantities are concerned, even the effects must be regarded as defined with only a finite accuracy (see Section 3.3.6).

Thus, for example, there is no guarantee that the value of \( \sqrt{x} \) is exactly equal to the value of \( |x|^{0.5} \), or that the effects of inreal and outreal will exactly match those given here.

ALGOL coding has been replaced by a metalinguistic variable (see Section 1.1) in places where the requirement is clear, and there is no simple way of specifying the operations needed in ALGOL:

comment Simple functions;

real procedure abs(E);

value E; real E;

abs := if E ≥ 0.0 then E else -E;

integer procedure labs(E);

value E; integer E;

labs := if E > 0 then E else -E;

integer procedure sign(E);

value E; real E;

sign := if E > 0.0 then 1 else if E < 0.0 then -1 else 0;

integer procedure enter(E);

value E; real E;

comment enter := largest integer not greater than E, i.e.

E - 1 < enter ≤ E;

begin

integer j;

j := E;

enter := if j > E then j - 1 else j

end enter;

comment Mathematical functions;

real procedure sqrt(E);

value E; real E;

if E < 0.0 then

fault("negative_sqrt", E)

else

sqrt := E|0.5;

real procedure sin(E);

value E; real E;

comment sin := sine of E radians;

<body>;

real procedure cos(E);

value E; real E;

comment cos := cosine of E radians;

<body>;

real procedure arctan(E);

value E; real E;

comment arctan := principal value, in radians, of arctangent of E,

i.e. \(-\pi/2 < \text{arctan} < \pi/2\);

<body>;

real procedure ln(E);

value E; real E;

comment ln := natural logarithm of E;

if E ≤ 0.0 then

fault("ln_not_positive", E)

else

<statement>;

real procedure exp(E);

value E; real E;

comment exp := exponential function of E;

if E > ln(maxreal) then

fault("overflow_on_exp", E)

else

<statement>;

comment Terminating procedures;

procedure stop;

comment for the use of \( \Omega \), see Section 2.1 Letters;

go to \( \Omega \);

procedure fault(str, r);

value r; string str; real r;

comment \( \Omega \) is assumed to denote a standard output channel. For the use of \( \Omega \) see Section 2.1 Letters. The following calls of fault appear:

integer divide by zero,

definition operation in expr,

definition operation in expn,

definition operation in expi,

and in the environmental block:

sqrt of negative argument,

in of negative or zero argument,

overflow on exp function,

invalid parameter for outchar,

invalid parameter in intinteger(twice),

invalid parameter in intreal(three times);

begin

outstring(\( \Omega \), "fault", r);

outstring(\( \Omega \), str);

outstring(\( \Omega \), "_; "; r);

comment Additional diagnostics may be output here;

stop

end fault;

comment Input/output procedures;

procedure inchar(channel, str, int);

value channel;

integer channel, int; string str;

comment Set int to value corresponding to the first position in str of current character on channel. Set int to zero if character not in str. Move channel pointer to next character;

<body>;

procedure outchar(channel, str, int);

value channel, int;

integer channel, int; string str;

comment Pass to channel the character in str, corresponding to the value of int;

if int < 1 \| int > length(str) then

fault("character_not_in_string", int)

else

<statement>;

integer procedure length(str);

string str;

comment length := number of characters in the open string enclosed by the outermost string quotes, after performing any necessary concatenation as defined in Section 2.6.3. Characters forming an inner string (see Section 2.6.3) are counted in an implementation dependent manner;

<body>;

procedure outstring(channel, str);

value channel;

integer channel; string str;

begin

integer m, n;

n := length(str);

for m := 1 step 1 until n do

outchar(channel, str, m)

end outstring;

procedure outterminator(channel);

value channel; integer channel;

comment outputs a terminator for use after a number. To be
procedure ininteger(channel, int);
value channel; integer channel, int;
comment int takes the value of an integer, as defined in Section 2.5.1, read from channel. The terminator of the integer may be either a space, a newline or a semicolon (if other terminators are to be allowed, they may be added to the end of the string parameter of the call of inchar. No other change is necessary). Any number of spaces or newlines may precede the first character of the integer;
begin
integer k, m;
Boolean b, d;
integer procedure ins;
begins
integer n;
comment read one character, converting newlines to spaces.
The inner string "NL" behaves as a single character representing newline;
inchar(channel, '0123456789 + - ;"NL"', n);
sins := if n = 15 then 13 else n
end ins;
comment pass over initial spaces or newlines;
for k := ins while k = 13 do

;comment fault anything except sign or digit;
if k = 0 or k > 13 then
fault("invalid_character", k);
if k > 10 then
begin
comment sign found, d indicates digit not found yet, b indicates whether + or −, m is value so far;
d := false;
b := k = 12;
m := 0
end
else
begin
comment d indicates digit found, b indicates +, m is value so far;
d := b := true;
m := k − 1
end
for k := ins while k > 0 ∧ k < 11 do
begin
comment deal with further digits;
m := 10 × m + k − 1;
d := true
end k loop;
comment fault if no digit has been found, or the terminator was invalid;
if d = k < 13 then
fault("invalid_character", k);
int := if b then m else −m
end ininteger;

procedure outinteger(channel, int);
value channel, int;
integer channel, int;
comment Passes to channel the characters representing the value of int, followed by a terminator;
begin
procedure digits(int);
value int; integer int;
begins
integer j;
comment use recursion to evaluate digits from right to left, but print them from left to right;
j := int = 10;
int := int − 10 × j;
if j = 0 then
digits(j);
outchar(channel, "0123456789", int + 1)
end digits;
if int < 0 then
begin
outchar(channel, "−", 1);
int := −int
end;
digits(int); outterminator(channel)
end outinteger;

procedure inreal(channel, re);
value channel;
integer channel; real re;
comment re takes the value of a number, as defined in Section 2.5.1, read from channel. Except for the different definitions of a number and an integer the rules are exactly as for ininteger. Spaces or newlines may follow the symbol ";
begin
integer j, k, m;
real r, s;
Boolean b, d;
integer procedure ins;
begins
integer n;
comment read one character, converting newlines to spaces.
The inner string "NL" behaves as a single character representing newline;
inchar(channel, '0123456789 − .10123456789− ;"NL"', n);
sins := if n = 17 then 15 else n
end ins;
comment pass over initial spaces or newlines;
for k := ins while k = 15 do

;comment fault anything except sign, digit, point or ten;
if k = 0 ∨ k > 15 then
fault("invalid_character", k);
comment b indicates whether + or −, d indicates whether further digits can have any effect, m will count places after the point, r is the value so far,
j indicates whether last character read was sign (j = 1), digit before point (j = 2), point (j = 3), digit after point (j = 4), or ten (j = 5);
b := k ≠ 11;
d := true;
m := 1;
j := if k < 11 then 2 else labs(k + k − 23);
r := if k < 11 then k − 1 else 0.0;
if k ≠ 14 then
begin
comment ten not found. Continue until ten or terminator found;
for k := ins while k < 14 do
begin
comment fault for non-numerical character, sign or second point;
if k = 0 ∨ k = 11 ∨ k = 12
$k = 13 ∧ j > 2$ then
fault("invalid_character", k);
comment deal with digit unless it cannot affect value;
if d then
begin
if k = 13 then
begin comment point found;
j := 3
end
else
begin
if j < 3 then
begin
calculate deal with digit before point;
r := 10.0 × r + k − 1
end
else
begin

procedure outheal;
value channel, re;
integer channel; real re;
comment Passes to channel the characters representing the
value of re, followed by a terminator;
begin
integer n;
comment n gives number of digits to print. Could be given as a
constant in any actual implementation;
n := enter(1.0 - ln(epsilon)) / ln(10.0));
if re < 0.0 then
begin
outchar(channel, "-", 1); re := -re
end;
if re < minreal then
begin
outstring(channel, "0.0"); outterminator(channel)
end
else
begin
integer j, k, m, p;
Boolean float, nines;
comment m will hold number of places point must be moved to
standardise value of re to have one digit before point;
m := 0;
comment standardise value of re;
for m := m + 1 while re > 10.0 do
re := re/10.0;
for m := m - 1 while re < 1.0 do
re := 10.0 * re;
if re > 10.0 then
begin
comment this can occur only by rounding error, but is a
necessary safeguard;
re := 1.0;
m := m + 1
end;
if m > n then m := -2 then
begin
comment printing to be in exponent form;
float := true;
p := 1
end
else
begin
comment printing to be in non-exponent form;
float := false;
comment if p = 0 point will not be printed. Otherwise point
will be after p digits;
p := if m = n - 1 then 0 else m + 1;
if m < 0 then
begin
outstring(channel, "0.0");
if m = -2 then
outchar(channel, "0");
end
end
end;
nines := false;
for j := 1 step 1 until n do
begin
comment if nines is true, all remaining digits must be 9. This
can occur only by rounding error, but is a necessary safeguard;
if nines then
k := 9
else
begin
comment find digit to print;
k := enter(re);
if k > 9 then
begin
k := 9;
nines := true
end
else
begin
comment move next digit to before point;
re := 10.0 * (re - k)
end
outchar(channel, "0123456789", k + 1);
if j = p then
outchar(channel, ",", 1)
end j loop;
if float then
begin
comment print exponent part. outinteger includes a call of
outterminator.
outchar(channel, "e", 1);
outinteger(channel, m)
end
else
outterminator(channel)
end
end outheal;
comment Environmental enquiries;
real procedure maxreal;
maxreal := \langle number\rangle;
real procedure minreal;
minreal := \langle number\rangle;
integer procedure maxint;
maxint := \langle integer\rangle;
comment maxreal, minreal, and maxint are, respectively, the
maximum allowable positive real number, the minimum allowable
positive real number, and the maximum allowable positive integer,
such that any valid expression of the form
\langle primary\rangle\langle arithmetic operator\rangle\langle primary\rangle
will be correctly evaluated, provided that each of the primaries
concerned, and the mathematically correct result lies within the open
interval (−maxreal, −minreal) or (minreal, maxreal) or is zero
if of real type, or within the open interval (−maxint, maxint) if of
integer type.
If the result is of real type, the words 'correctly evaluated' must be
understood in the sense of numerical analysis (see Section 3.3.6);
real procedure epsilon;
comment The smallest positive real number such that the
computational result of 1.0 + epsilon is greater than 1.0 and the
computational result of 1.0 − epsilon is less than 1.0;
epsilon := \langle number\rangle;
comment In any particular implementation, further standard
functions and procedures may be added here, but no additional ones
may be regarded as part of the reference language (in particular,
a less rudimentary input/output system is desirable);
\langle fictitious declaration of own variables\rangle;
\langle initialisation of own variables \rangle;
\langle program\rangle;
\Omega;
Alphabetic index of definitions of concepts and syntactic units

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

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

syn Following the abbreviation 'syn', references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text Following the word 'text', the references to definitions given in text are given.

+ see: plus
− see: minus
× see: multiply
/ ÷ see: divide
| see: exponentiation
< ≤ = ≥ > ≠ see: (relational operator)
≡ ≡ ≃ ≈ see: (logical operator)
, see: comma
. see: decimal point
; see: ten
: see: colon
; see: semicolon
= see: colon equal
() see: parenteses
[] see: subscript bracket
“ see: string quote

<actual parameter>, def 3.2.1, 4.7.1
<actual parameter list>, def 3.2.1, 4.7.1
<actual parameter part>, def 3.2.1, 4.7.1
<adding operator>, def 3.3.1
alphabet, text 2.1
arithmetic, text 3.3.6
<arithmetic expression>, def 3.3.1 synt 3, 3.1.1, 3.4.1, 4.2.1, 4.6.1,
5.2.1 text 3.3.3
<arithmetic operator>, def 2.3 text 3.3.4
array, synt 2.3, 5.2.1
text 3.1.4.1
<array declaration>, def 5.2.1 synt 5 text 5.2.3
<array declarator>, def 5.2.1, synt 5.4.1
<array identifier>, def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
<array list>, def 5.2.1
<array segment>, def 5.2.1
<assignment statement>, def 4.2.1 synt 4.1.1 text 1, 4.2.3

<basic statement>, def 4.1.1 synt 4.5.1
<basic symbol>, def 2
begin, synt 2.3, 4.1.1
<block>, def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
<bracket>, def 4.1.1
Boolean, synt 2.3, 5.1.1 text 5.1.3
<Boolean expression>, def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1 text 3.4.3
<Boolean factor>, def 3.4.1
<Boolean primary>, def 3.4.1
<Boolean secondary>, def 3.4.1
<Boolean term>, def 3.4.1
<bound pair>, def 5.2.1
<bound pair list>, def 5.2.1
<bracket>, def 2.3
<closed string>, def 2.6.1

<code>, synt 5.4.1 text 4.7.8, 5.4.6
colon:, synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1
comma, synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1
comment, synt 2.3
comment convention, text 2.3
<compound statement>, def 4.1.1 synt 4.5.1 text 1
<compound tail>, def 4.1.1
<conditional statement>, def 4.5.1 synt 4.1.1 text 4.5.3
<decimal fraction>, def 2.5.1
<decimal number>, def 2.5.1 text 2.5.3
decimal point, synt 2.3, 2.5.1
<declaration>, def 5 synt 4.1.1 text 1, 5
<declarator>, def 2.3
<delimiter>, def 2.3 synt 2
<designational expression>, def 3.5.1 synt 3, 4.3.1, 5.3.1 text 3.5.3
<destination>, def 4.2.1
<digit>, def 2.2.1 synt 2, 2.4.1, 2.5.1
dimension, text 5.2.3.2
divide ÷, synt 2.3, 3.3.1 text 3.3.4.2
do, synt 2.3, 4.6.1
<dummy statement>, def 4.4.1 synt 4.1.1 text 4.4.3
else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.5.1 text 4.5.3.2
<empty>, def 1.1 synt 2.6.1, 3.2.1, 4.4.1, 4.7.1, 5.1.1, 5.4.1
end, synt 2.3, 4.1.1
exponentiation †, synt 2.3, 3.3.1 text 3.3.4.3
<exponent part>, def 2.5.1 text 2.5.3
<expression>, def 3 synt 3.2.1, 4.7.1 text 3
<factor>, def 3.3.1
false, synt 2.2.2
for, synt 2.3, 4.6.1
<for clause>, def 4.6.1 text 4.6.3
<for list>, def 4.6.1 text 4.6.4
<for list element>, def 4.6.1 text 4.6.4.1, 4.6.4.2, 4.6.4.3
<formal parameter>, def 5.4.1, text 5.4.3
<formal parameter list>, def 5.4.1
<formal parameter part>, def 5.4.1
<for statement>, def 4.6.1 synt 4.1.1, 4.5.1 text 4.6
<function designator>, def 3.2.1 synt 3.3.1, 3.4.1 text 3.2.3, 5.4.4
go to, synt 2.3, 4.3.1
<go to statement>, def 4.3.1 synt 4.1.1 text 4.3.3
<identifier>, def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1 text 2.4.3
<identifier list>, def 5.4.1
if, synt 2.3, 3.3.1, 4.5.1
<if clause>, def 3.3.1, 4.5.1 synt 3.4.1, 3.5.1 text 3.3.3, 4.5.3.2
<if statement>, def 4.5.1 text 4.5.3.1
<implication>, def 3.4.1
integer, synt 2.3, 5.1.1 text 5.1.3
<integer>, def 2.5.1 text 2.5.4
label, synt 2.3, 5.4.1
<label>, def 3.5.1 synt 4.1.1, 4.5.1, 4.6.1 text 1, 4.1.3, 4.7.6
<left part>, def 4.2.1
<left part list>, def 4.2.1
<letter>, def 2.1 synt 2, 2.4.1, 3.2.1, 4.7.1
<letter string>, def 3.2.1, 4.7.1
local, text 4.1.3
non-local, text 4.1.3
<number>, def 2.5.1 text 2.5.3, 2.5.4

/open string>, def 2.6.1
/operator>, def 2.3
/own, synt 2.3, 5.1.1 text 5, 5.2.5

/parameter delimiter>, def 3.2.1, 4.7.1 synt 5.4.1 text 4.7.7
/parentheses ( ), synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 4.7.1, 5.4.1, text 3.3.5.2.

/plus + , synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1

/primary>, def 3.3.1
/procedure, synt 2.3, 5.4.1
/procedure body>, def 5.4.1
/procedure declaration>, def 5.4.1 synt 5 text 5.4.3
/procedure heading>, def 5.4.1 text 5.4.3
/procedure identifier>, def 3.2.1 synt 3.2.1, 4.2.1, 4.7.1, 5.4.1 text 4.7.5.4
/procedure statement>, def 4.7.1 synt 4.1.1 text 4.7.3
/program>, def 4.1.1 text 1
/proper string>, def 2.6.1

/quantity, text 2.7

/reall, synt 2.3, 5.1.1 text 5.1.3
/relational operator>, def 2.3, 3.4.1

/scope, text 2.7
/semicolon; synt 2.3, 4.1.1, 5.4.1
/separator>, def 2.3
/sequential operator>, def 2.3
/simple arithmetic expression>, def 3.3.1 synt 3.4.1 text 3.3.3
/simple Boolean>, def 3.4.1
/simple designational expression>, def 3.5.1
/simple variable>, def 3.1.1 synt 5.1.1 text 2.4.3
/specification part>, def 5.4.1 text 5.4.5
/specifier>, def 2.3
/specifier>, def 5.4.1
standard functions and procedures, text 3.2.4
standard procedures, text 4.7.9

/statement>, def 4.1.1 synt 4.5.1, 4.6.1, 5.4.1 text 4
/statement bracket see: begin end

/step, synt 2.3, 4.6.1 text 4.6.4.2
/string, synt 2.3, 5.4.1
/strings>, def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
/string quotes "" synt 2.3, 2.6.1 text 2.6.3
/subscript, text 3.1.4.1
/subscript bound, text 5.2.3.1
/subscript brackets [ ], synt 2.3, 3.1.1, 3.5.1, 5.2.1
/subscripted variable>, def 3.1.1 text 3.1.4.1
/subscript expression>, def 3.1.1 synt 3.5.1
/subscript list>, def 3.1.1

/successor, text 4

/switch, synt 2.3, 5.3.1, 5.4.1
/switch declaration>, def 5.3.1 synt 5 text 5.3.3
/switch designator>, def 3.5.1 text 3.5.3

/switch identifier>, def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
/switch list>, def 5.3.1

/term>, def 3.3.1
ten 10 synt 2.3, 2.5.1
/then, synt 2.3, 3.3.1, 4.5.1

/true, synt 2.2.2
/type>, def 5.1.1 synt 5.4.1 text 2.8
/type declaration>, def 5.1.1 synt 5 text 5.1.3
/type list>, def 5.1.1
/unconditional statement>, def 4.1.1, 4.5.1
/unlabelled basic statement>, def 4.1.1
/unlabelled block>, def 4.1.1
/unlabelled compound>, def 4.1.1
/unsigned integer>, def 2.5.1
/unsigned number>, def 2.5.1 synt 3.3.1
/until, synt 2.3, 4.6.1 text 4.6.4.2
/upper bound>, def 5.2.1 text 5.2.4
/value, synt 2.3, 5.4.1
/value, text 2.8, 3.3.3
/value part>, def 5.4.1 text 4.7.3.1
/variable>, def 3.1.1 synt 3.3.1, 3.4.1, 4.2.1, text 3.1.3
/variable identifier>, def 3.1.1 synt 4.6.1

/while, synt 2.3, 4.6.1 text 4.6.4.3