Algol W
Language Description

by
Henry Bauer
Sheldon Becker
Susan L. Graham
Edwin Satterthwaite
Richard L. Sites

June 1972

Contents

1 Terminology, Notation and Basic Definitions **4**
1.1 Notation 4
1.2 Definitions 5

2 Sets of Basic Symbols and Syntactic Entities **6**
2.1 Basic Symbols 6
2.2 Syntactic entities 7

3 Identifiers **7**

4 Values and Types **9**
4.1 Numbers 10
4.2 Logical Values 11
4.3 Bit Sequences 11
4.4 Strings 12
4.5 References 12

5 Declarations **13**
5.1 Simple Variable Declarations 13
5.2 Array Declarations 14
5.3 Procedure Declarations 16
5.4 Record Class Declarations 19

6 Expressions **20**
6.1 Variables 22
6.2 Function Designators 23
6.3 Arithmetic Expressions 24
6.4 Logical Expressions 26
6.5 Bit Expressions 28
6.6 String Expressions 29
6.7 Reference Expressions 29
6.8 Conditional Expressions 30

7 Statements **31**
7.1 Blocks 32
7.2 Assignment Statements 33
7.3 Procedure Statements 34
7.4 Goto Statements 36
7.5 If Statements 36
7.6 Assert Statements 37

7.7 Case Statements 38
7.8 Iterative Statements 39
7.9 Standard Procedures 41
7.9.1 The Input/Output System 41
7.9.2 Read Statements 42
7.9.3 Write Statements 43
7.9.4 Control Statements 45

8 Standard Functions and Predeclared Identifiers 47
8.1 Standard Transfer Functions 47
8.2 Standard Functions of Analysis 49
8.3 Time Function 50
8.4 Predeclared Variables 51
8.5 Exceptional Conditions 52

A Character Encodings 56

Index of Syntactic Entities 57

Words with Special Meanings in Algol W 59

Index 61

1 Terminology, Notation and Basic Definitions

The Reference Language is a phrase structure language, defined by a formal metalanguage. This metalanguage makes use of the notation and definitions explained below. The structure of the language ALGOL W is determined by:

1. V_T, the set of basic (or *terminal*) constituents of the language[1],
2. V_N, the set of syntactic entities (or *nonterminal symbols*), and
3. P, the set of syntactic rules (or productions)

1.1 Notation

A syntactic entity is denoted by its name (a sequence of letters) closed in the brackets $\langle$ and $\rangle$. A syntactic rule has the form

$$\langle A \rangle ::= \text{x}$$

where $\langle A \rangle$ is a member of V_N, x is any possible sequence of basic constituents and syntactic entities, simply to be called a "sequence". In ALGOL W, the set P contains the syntactic rule

$$\langle bar \rangle ::= \text{`|'}$$

implying that '|' is a basic symbol of the language. Adopting the convention that all references to this basic symbol in other syntactic rules shall be replaced by $\langle bar \rangle$ permits the unambiguous[1] use subsequently of the notation

$$\langle A \rangle ::= \text{x} \mid \text{y} \mid \ldots \mid \text{z}$$

is used as an abbreviation for the set of syntactic rules

$$\begin{aligned} \langle A \rangle &::= \text{x} \\ \langle A \rangle &::= \text{y} \\ &\ldots \\ \langle A \rangle &::= \text{z} \end{aligned}$$

In the syntactic rule

$$\langle empty \rangle ::=$$

the sequence contains zero symbols, i.e. the empty sequence.

[1] In this LaTeX document, terminal symbols are represented by characters between 'single quotes' or words in **boldface**. The use of symbol '|' in syntactic rules has become unambiguous.

1.2 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 either (i) for some $\langle A \rangle$ in V_N, $x = u\langle A \rangle w$, $y = uvw$, and $\langle A \rangle ::= v$ is a rule in P; or (ii) $x = uw$, $y = uvw$ and v is a "comment" (see below).

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 W program if and only if its constituents are members of the set V_T, and x can be produced from the syntactic entity $\langle program \rangle$.

The sets V_T and $V_N - \{\text{`|'}\}$ are defined through enumeration of their members given throughout the sequel of the Report. To provide explanations for the meaning of ALGOL W 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 refer to the corresponding syntactic definition. Along with these letter sequences the symbol τ may occur. It is understood that this symbol must be replaced by any one of a finite set of English words (or word pairs). Unless otherwise specified in the particular section, all occurrences of the symbol τ within one syntactic rule must be replaced consistently, and the replacing words are

integer	*logical*
real	*bit*
long real	*string*
complex	*reference*
long complex	

For example, the production

$$\langle \tau \ term \rangle ::= \langle \tau \ factor \rangle$$

corresponds to

$$\begin{aligned}
\langle integer\ term \rangle &::= \langle integer\ factor \rangle \\
\langle real\ term \rangle &::= \langle real\ factor \rangle \\
\langle long\ real\ term \rangle &::= \langle long\ real\ factor \rangle \\
\langle complex\ term \rangle &::= \langle complex\ factor \rangle \\
\langle long\ complex\ term \rangle &::= \langle long\ complex\ factor \rangle
\end{aligned}$$

The production

$$\langle \tau_0\ primary \rangle ::= \textbf{long}\ \langle \tau_1\ primary \rangle$$

corresponds to

$$\langle long\ real\ primary \rangle ::= \textbf{long}\ \langle real\ primary \rangle$$
$$\langle long\ integer\ primary \rangle ::= \textbf{long}\ \langle integer\ primary \rangle$$
$$\langle long\ complex\ primary \rangle ::= \textbf{long}\ \langle complex\ primary \rangle$$

It is recognized that typographical entities exist of lower order than basic symbols, called characters. The accepted characters are those of the IBM System 360 EBCDIC code.

The symbol **comment** followed by any sequence of characters not containing semicolons, followed by a semicolon, is called a comment. A comment has no effect on the meaning of a program, and is ignored during execution of the program. An identifier (cf. 3) immediately following the basic symbol **end** is also regarded as a comment.

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. In the definition of the implemented language the evaluation or execution of certain constructs is either (1) defined by System 360 operations, e.g., real arithmetic, or (2) left undefined, e.g., the order of evaluation of arithmetic primaries in expressions, or (3) said to be *not valid* or *not defined*.

2 Sets of Basic Symbols and Syntactic Entities

2.1 Basic Symbols

‘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’ | ‘_’ |

‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ |

true | **false** | ‘"’ | **null** | ‘#’ | ‘’’ |
integer | **real** | **complex** | **logical** | **bits** | **string** | **reference** |
long real | **long complex** | **array** | **procedure** | **record** |
‘,’ | ‘;’ | ‘:’ | ‘.’ | ‘(’ | ‘)’ | **begin** | **end** | **if** | **then** | **else** | **case**
| **of** | ‘+’ | ‘−’ | ‘*’ | ‘/’ | ‘**’ | **div** | **rem** | **shr** | **shl** | **is** | **abs** |

long | **short** | **and** | **or** | ‘¬’ | ⟨*bar*⟩ | ‘<=’ | ‘¬=’ | ‘<’ | ‘=’ | ‘>’ | ‘>=’ | ‘: :’ | ‘:=’ | **goto** | **go to** | **for** | **step** | **until** | **do** | **while** | **comment** | **value** | **result** | **assert** | **algol** | **fortran**

All **bold lowercase** words, which we call “reserved words”, are represented by the same words in capital letters in an actual program, with no intervening blanks.

Adjacent reserved words, identifiers (cf. 3) and numbers must include no blanks and must be separated by at least one blank space. Otherwise blanks have no meaning and can be used freely to improve the readability of the program.

2.2 Syntactic entities

(See the *Syntactic Entities* index on page 56.)

3 Identifiers

⟨*identifier*⟩ ::= ⟨*letter*⟩
| ⟨*identifier*⟩ ⟨*letter*⟩
| ⟨*identifier*⟩ ⟨*digit*⟩
| ⟨*identifier*⟩ ‘_’

⟨τ *variable identifier*⟩ ::= ⟨*identifier*⟩

⟨τ *array identifier*⟩ ::= ⟨*identifier*⟩

⟨*procedure identifier*⟩ ::= ⟨*identifier*⟩

⟨τ *function identifier*⟩ ::= ⟨*identifier*⟩

⟨*record class identifier*⟩ ::= ⟨*identifier*⟩

⟨τ *field identifier*⟩ ::= ⟨*identifier*⟩

⟨*label identifier*⟩ ::= ⟨*identifier*⟩

⟨*control identifier*⟩ ::= ⟨*identifier*⟩

⟨*letter*⟩ ::= ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’
| ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’
| ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’
| ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’

⟨*digit*⟩ ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

⟨*identifier list*⟩ ::= ⟨*identifier*⟩
| ⟨*identifier list*⟩ ‘,’ ⟨*identifier*⟩

3.0.1. Semantics

Variables, arrays, procedures, record classes and record fields are said to be quantities. Identifiers serve to identify quantities, or they stand as labels, formal parameters or Control identifiers. Identifiers have no inherent meaning, and can be chosen freely in the reference language. In an actual program a reserved word cannot be used as an identifier.

Every identifier used in a program must be defined. This is achieved through

(a) declaration (cf. Section 5), if the identifier identifies a quantity. It is then said to denote that quantity and to be a τ variable identifier, τ array identifier, τ procedure identifier, τ function identifier, record class identifier or τ field identifier, where the symbol τ stands for the appropriate word reflecting the type of the declared quantity;

(b) a label definition (cf. 7.1), if the identifier stands as a label. It is then said to be a label identifier;

(c) its occurrence in a formal parameter list (cf. 5.3). It is then said to be a formal parameter;

(d) its occurrence following the symbol **for** in a for clause (cf. 7.8). It is then said to be a control identifier

(e) its implicit declaration in the language. Standard procedures, standard functions, and predefined variables (cf. 7.9 and 8) may be considered to be declared in a block containing the program.

The recognition of the definition of a given identifier is determined by the following rules:

Step 1. If the identifier is defined by a declaration of a quantity or by its standing as a label within the smallest block (cf. 7.1) embracing a given occurrence of that identifier, then it denotes that quantity or label. A statement following a procedure heading (cf. 5.3) or a **for** clause (cf. 7.8) is considered to be a block.

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 stands as that formal parameter.

Step 3. Otherwise, if that block is preceded by a **for** clause and the identifier is identical to the control identifier of that **for** clause, then it stands as that control identifier. Otherwise, these rules are applied 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 definition, then the identification is undefined.

The scope of a quantity, a label, a formal parameter, or a control identifier is the set of statements in which occurrences of an identifier may refer by the above rules to the definition of that quantity, label, formal parameter or control identifier.

3.0.2. Examples

```
I
PERSON
ELDERSIBLING
X15, X20, X25
NEW_PAGE
```

4 Values and Types

Constants and variables (cf. 6.1) are said to possess a value. The value of a constant is determined by the denotation of the constant. In the language, all constants (except references) have a reference denotation (cf. 4.1 4.4). The value of a variable is the one most recently assigned to that variable. A value is (recursively) defined as either a simple value or a structured value (an ordered set of 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 a 32 bit integer,
real	the value is a 32 bit floating point number,
long real	the value is a 64 bit floating point number,
complex	the value is a complex number composed of two numbers of type *real*,
long complex	the value is a complex number composed of two long real numbers,
logical	the value is a logical value,
bits	the value is a linear sequence of 32 bits,
string	the value is a linear sequence of at most 256 characters,
reference	the value is a reference to a record.

The following types of structured values are distinguished:

array	the value is an ordered set of values, all of identical simple type,
record	the value is an ordered set of simple values.

A procedure may yield a value, in which case it is said to be a function procedure, or it may not yield 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. 6.2).

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, except, of course, in the case of strings.

4.1 Numbers

4.1.1. Syntax

⟨*long complex constant*⟩ ::= ⟨*complex constant*⟩ 'L'

⟨*complex constant*⟩ ::= ⟨*imaginary constant*⟩

⟨*imaginary constant*⟩ ::= ⟨*real constant*⟩ 'I'
| ⟨*integer constant*⟩ 'I'

⟨*long real constant*⟩ ::= ⟨*real constant*⟩ 'L'
| ⟨*integer constant*⟩ 'L'

⟨*real constant*⟩ ::= ⟨*unscaled real*⟩
| ⟨*unscaled real*⟩ ⟨*scale factor*⟩

| ⟨*integer constant*⟩ ⟨*scale factor*⟩
| ⟨*scale factor*⟩

⟨*unscaled real*⟩ ::= ⟨*integer constant*⟩ '.' ⟨*integer constant*⟩
| '.' ⟨*integer constant*⟩
| ⟨*integer constant*⟩ '.'

⟨*scale factor*⟩ ::= ''' ⟨*integer constant*⟩
| ''' ⟨*sign*⟩ ⟨*integer constant*⟩

⟨*integer constant*⟩ ::= ⟨*digit*⟩
| ⟨*integer constant*⟩ ⟨*digit*⟩

⟨*sign*⟩ ::= '+' | '-'

(Note: a long complex constant may have the `I` and `L` in either order in a program, but they must be in the order `IL` on data cards.)

4.1.2. Semantics

Numbers are interpreted according to the conventional decimal notation. A scale factor denotes an integral power of 10 which is multiplied by the unscaled real or integer number preceding it. Each number has a uniquely defined type. (Note that all ⟨τ *constant*⟩s are unsigned.)

4.1.3. Examples

```
1                   05                  11
0100                1'3                 0.671
3.1416              6.02486'+23         1IL
2.7182818284590452353602874L            2.3'-6
```

4.2 Logical Values

4.2.1. Syntax

⟨*logical constant*⟩ ::= **true** | **false**

4.3 Bit Sequences

4.3.1. Syntax

⟨*bit constant*⟩ ::= '#' ⟨*hex digit*⟩
| ⟨*bit constant*⟩ ⟨*hex digit*⟩

⟨*hex digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
| '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'

Note that '`A`' | ... | '`F`' corresponds to 10_{10}|...|15_{10}.

4.3.2. Semantics

The number of bits in a bit sequence is 32, or 8 hex digits. The bit sequence is always represented by a 32 bit word with the specified bit sequence right justified in the word and zeros filled in on the left.

4.3.3. Examples

`#4F` = 0000 0000 0000 0000 0000 0000 0100 1111
`#9` = 0000 0000 0000 0000 0000 0000 0000 1001

4.4 Strings

4.4.1. Syntax

⟨*string constant*⟩ ::= ⟨*string*⟩

⟨*string*⟩ ::= '`"`' ⟨*open string*⟩ '`"`'

⟨*open string*⟩ ::= ⟨*character*⟩ | ⟨*open string*⟩ ⟨*character*⟩

4.4.2. Semantics

Strings consist of any sequence of (a t least one and at most at most 256) characters accepted by the System 360 enclosed by `"`, the string quote. If the string quote appears in the sequence of characters it must be immediately followed by a second string quote which is then ignored. The number of characters in a string is said to be the length of the string. The characters accepted by the IBM system 360 are listed in Appendix I.

4.4.3. Example

`"JOHN"`
`""""` This is the string of length 1 consisting of the string quote.

4.5 References

4.5.1. Syntax

⟨*reference constant*⟩ ::= **null**

4.5.2. Semantics

The reference value **null** fails to designate a record; if a reference expression occurring in a field designator (cf. 6.1) has this value, then the field designator is undefined.

5 Declarations

Declarations serve to associate identifiers with the quantities used in the program, to attribute certain permanent properties to these quantities (i.e:. type, structure), and to determine their Scope. The quantities declared by declarations are simple variables, arrays, procedures and record classes.

Upon exit from a block, all quantities declared or defined within that block lose their value and significance (cf. 7.1 and 7.4).

5.0.3. Syntax

⟨*declaration*⟩ ::= ⟨*simple variable declaration*⟩
| ⟨τ *array declaration*⟩
| ⟨*procedure declaration*⟩
| ⟨*record class 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** | **bits** '`(`' '`32`' '`)`'
| **string** | **string** '`(`' ⟨*integer constant*⟩ '`)`'
| **reference** '`(`' ⟨*record class identifier list*⟩ '`)`'

⟨*record class identifier list*⟩ ::= ⟨*record class identifier*⟩
| ⟨*record class identifier list*⟩'`,`' ⟨*record class identifier*⟩

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 5). 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) can be assigned to it. It is understood that the value of a variable is equal to the value of the expression most recently assigned to it.

A variable of type *bits* is always of length 32 whether or not the declaration specification is included.

A variable of type *string* has a length equal to the unsigned integer in the declaration specification. If the simple type is given only as **string**, the length of the variable is 16 characters.

A variable of type *reference* may refer only to records of the record classes whose identifiers appear in the record class identifier list of the reference declaration specification.

5.1.3. Example

```
integer I, J, K, M, N
real X, Y, Z
long complex C
logical L
bits G, H
string (10) S, T
reference (PERSON) JACK, JILL
```

5.2 Array Declarations

5.2.1. Syntax

⟨*τ array declaration*⟩ ::= ⟨*simple type*⟩ **array** ⟨*identifier list*⟩
'(' ⟨*bound pair list*⟩ ')'

⟨*bound pair list*⟩ ::= ⟨*bound pair*⟩
| ⟨*bound pair list*⟩ ',' ⟨*bound pair*⟩

⟨*bound pair*⟩ ::= ⟨*lower bound*⟩ ':' ⟨*upper bound*⟩

⟨*lower bound*⟩ ::= ⟨*integer expression*⟩

⟨*upper bound*⟩ ::= ⟨*integer expression*⟩

5.2.2. Semantics

Each identifier of the identifier list of an array declaration is associated with a variable which is declared to be of type *array*. A variable of type *array* is an ordered set of variables whose type is the simple type preceding the symbol array. The dimension of the array is the number of entries in the bound pair list.

Every element of an array is identified by a list of indices. The indices are the integers between and including the values of the lower bound and the upper bound. Every expression in the bound pair list is evaluated exactly once upon entry to the block in which the declaration occurs. The bound pair expressions can depend only on variables and procedures global to the block in which the declaration occurs. In order to be valid, for every bound pair, the value of the upper bound must not be less than the value of the lower bound.

5.2.3. Example

```
integer array H(1::100)
real array A, B(1::M, 1::N)
string (12) array STREET, TOWN, CITY (J::K + 1)
```

5.3 Procedure Declarations

5.3.1. Syntax

⟨*procedure declaration*⟩ ::= ⟨*proper procedure declaration*⟩
| ⟨*τ function procedure declaration*⟩

⟨*proper procedure declaration*⟩ ::= **procedure** ⟨*procedure heading*⟩ ';'
⟨*proper procedure body*⟩

⟨*τ function procedure declaration*⟩ ::= ⟨*simple type*⟩
procedure ⟨*procedure heading*⟩ ';'
⟨*τ function procedure body*⟩

⟨*proper procedure body*⟩ ::= ⟨*statement*⟩
| ⟨*external reference*⟩

⟨*τ function procedure body*⟩ ::= ⟨*τ expression*⟩
| ⟨*block body*⟩ ⟨*τ expression*⟩ **end**
| ⟨*external reference*⟩

⟨*procedure heading*⟩ ::= ⟨*identifier*⟩
| ⟨*identifier*⟩
'(' ⟨*formal parameter list*⟩ ')'

⟨*formal parameter list*⟩ ::= ⟨*formal parameter segment*⟩
| ⟨*formal parameter list*⟩ ';'
⟨*formal parameter segment*⟩

⟨*formal parameter segment*⟩ ::= ⟨*formal type*⟩ ⟨*identifier list*⟩
| ⟨*formal array parameter*⟩

⟨*formal type*⟩ ::= ⟨*simple type*⟩
| ⟨*simple type*⟩ **value**
| ⟨*simple type*⟩ **result**
| ⟨*simple type*⟩ **value result**
| ⟨*simple type*⟩ **procedure**
| **procedure**

⟨*formal array parameter*⟩ ::= ⟨*simple type*⟩ **array** ⟨*identifier list*⟩
'(' ⟨*dimension specification*⟩ ')'

⟨*dimension specification*⟩ ::= '*'
| ⟨*dimension specification*⟩ ',' '*'

⟨*external reference*⟩ ::= **fortran** ⟨*string constant*⟩
| **algol** ⟨*string constant*⟩

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. Other parts of the block in whose heading the procedure is declared can then cause this procedure body to be executed or evaluated. 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.

5.3.2.1 Type specification of formal parameters. All formal parameters of a formal parameter segment are of the same indicated type. The type must be such that the replacement of the formal parameter by the actual parameter of this specified type leads to correct ALGOL W expressions and statements (cf. 7.3).

5.3.2.2 The effect of the symbols **value** and **result** appearing in a formal type is explained by the following rule, which is applied to the procedure body before the procedure is invoked:

1. The procedure body is enclosed by the symbols **begin** and **end**;
2. For every formal parameter whose formal type contains the symbol **value** or **result** (or both),
   (a) a declaration followed by a semicolon is inserted after the first **begin** of the procedure body, with a simple type as indicated in the formal type, and with an identifier different from any identifier valid at the place of the declaration.
   (b) throughout the procedure body, every occurrence of the formal parameter identifier is replaced by the identifier defined in step 2(a);
3. If the formal type contains the symbol **value**, an assignment statement (cf. 7.2) followed by a semicolon is inserted after the declarations in the outermost block of the procedure body. Its left part contains the identifier defined in step 2a, and its expression consists of the formal parameter identifier. The symbol **value** is then deleted;

4. If the formal type contains the symbol **result**, an assignment statement preceded by a semicolon is inserted before the symbol **end** which terminates the procedure body. Its left part contains the formal parameter identifier, and its expression consists of the identifier defined in step 2a. The symbol **result** is then deleted.

5.3.2.3 Specification of array dimensions. The number of *'s appearing in the formal array specification is the dimension of the array parameter.

5.3.2.4 External references. Use of an external reference as a procedure body indicates that the actual procedure body is specified by the environment in which the program is to be executed. The information in the external reference is used to locate and interpret that procedure body. The details of such use depend on the specific environment.

5.3.3. Examples

```
procedure INCREMENT; X := X+1

real procedure MAX (real value X, Y);
    if X < Y then Y else X

procedure COPY (real array U, V (*,*); integer value A, B);
    for I := 1 until A do
        for J := 1 until B do U(I,J) := V(I,J)

real procedure HORNER (real array A (*); integer value N;
    real value X);
    begin real S; S := 0;
        for I := 0 until N do S := S * X + A(I);
        S
    end

reference (PERSON) procedure YOUNGESTUNCLE (reference (PERSON) R);
    begin reference (PERSON) P, M;
        P := YOUNGESTOFFPRING (FATHER (FATHER (R)));
        while (P ¬= null) and (¬ MALE (P)) or (P = FATHER(R)) do
            P := ELDERSIBLING (P);
        M := YOUNGESTOFFSPRING (MOTHER (MOTHER (R)));
        while (M ¬= null) and (¬ MALE (M)) do
            M := ELDERSIBLING (M);
        if P = null then M else
```

```
        if M = null then P else
        if AGE(P) < AGE(M) then P else M
    end

procedure PLOTSUBROUTINE (integer value I); fortran "PLOTSUB"
```

5.4 Record Class Declarations

5.4.1. Syntax

⟨*record class declaration*⟩ ::= **record** ⟨*identifier*⟩ '(' ⟨*field list*⟩ ')'

⟨*field list*⟩ ::= ⟨*simple variable declaration*⟩
| ⟨*field list*⟩ ';' ⟨*simple variable declaration*⟩

5.4.2. Semantics

A record class declaration serves to define the structural properties of records belonging to the class. The principal constituent of a record-class declaration is a sequence of simple variable declarations which define the fields and their simple types for the records of this class and associate identifiers with the individual fields. A record class identifier can be used in a record designator (cf. 6.3) to construct a new record of the given class.

5.4.3. Examples

```
record NODE (reference (NODE) LEFT, RIGHT)

record PERSON (
    string NAME;
    integer AGE;
    logical MALE;
    reference (PERSON) FATHER, MOTHER, YOUNGESTOFFSPRING, ELDERSIBLING
);
```

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. The operands are either constants, variables or function designators, or other expressions, enclosed by parentheses if necessary. The evaluation of operands other than constants 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 has two operands, then these operands may be evaluated in any order with the exception of the logical operators discussed in 6.4.2.1.

Expressions are distinguished by a type and precedence level, the former depending on the types of the operands and the latter resulting from the precedence hierarchy imposed upon operators in the syntactic rules which follow. The syntactic entities naming different kinds of expression in these rules display these attributes, the word "expression" prefixed by the a type and, usually, postfixed by an integer indicating the precedence level. (higher precedence is implied by increasing magnitude of this integer.) The operators and their precedence levels are:

Level	**Operators**
1	**or**
2	**and**
3	¬
4	<, <=, =, ¬=, >=, >, **is**
5	+, -
6	*, /, **div**, **rem**
7	**shl**, **shr**, **
8	**long**, **short**, **abs**

When the types allow an operator at level i to be applied to operands, the resulting expression, which which belongs to the syntactic class $\langle \tau \ \textit{expression} \ i \rangle$ has the intuitive meaning given in the second column of the following table.

Syntactic Entity	Meaning	Definitions
$\langle \tau$ *expression 1*$\rangle$	disjunction	6, 6.4, 6.5
$\langle \tau$ *expression 2*$\rangle$	conjunction	6, 6.4, 6.5
$\langle \tau$ *expression 3*$\rangle$	negation	6, 6.4, 6.5
$\langle \tau$ *expression 4*$\rangle$	relation	6, 6.4
$\langle \tau$ *expression 5*$\rangle$	sum	6, 6.3
$\langle \tau$ *expression 6*$\rangle$	term	6, 6.3
$\langle \tau$ *expression 7*$\rangle$	factor	6, 6.3, 6.5
$\langle \tau$ *expression 8*$\rangle$	primary	6, 6.3, 6.6

The third column of the table indicates sections where definitions of these syntactic entities occur.

Throughout section 6 and its subsections the symbol τ has to be replaced consistently as described in Section 1, and where the triplets τ_0, τ_1, τ_2 have to be either all three replaced by the same one of the words

logical
bit
string
reference

or (subject to specification to the contrary) in accordance with the following "triplet rules".

1. Given the qualities (*integer*, *real* or *complex*) of τ_1 and τ_2, the corresponding quality of τ_0 is given in the table

τ_1 \ τ_2	*integer*	*real*	*complex*
integer	*integer*	*real*	*complex*
real	*real*	*real*	*complex*
complex	*complex*	*complex*	*complex*

2. τ_0 has the quality *long* either if both τ_1 and τ_2 have that quality, or if one has the quality *long* and the other is *integer*.

Syntax

$\langle \tau$ *expression*$\rangle$ ::= $\langle \tau$ *expression 1*$\rangle$ | $\langle$*conditional* τ *expression*$\rangle$

$\langle \tau$ *expression 1*$\rangle$ ::= $\langle \tau$ *expression 2*$\rangle$

$\langle \tau$ *expression 2*$\rangle$ ::= $\langle \tau$ *expression 3*$\rangle$

⟨τ *expression 3*⟩ ::= ⟨τ *expression 4*⟩

⟨τ *expression 4*⟩ ::= ⟨τ *expression 5*⟩

⟨τ *expression 5*⟩ ::= ⟨τ *expression 6*⟩

⟨τ *expression 6*⟩ ::= ⟨τ *expression 7*⟩

⟨τ *expression 7*⟩ ::= ⟨τ *expression 8*⟩

⟨τ *expression 8*⟩ ::= ⟨τ *constant*⟩ | '(' ⟨τ *expression*⟩ ')' | ⟨τ *block expression*⟩

⟨τ *block expression*⟩ ::= ⟨*block body*⟩ ⟨τ *expression*⟩ **end**

Semantics

There are 8 levels of precedence; an expression at one level of precedence is a valid expression at a lower level of precedence.

A block expression introduces a new level of nomenclature and specifies the execution of statements in the block body as described for blocks (cf.7.1). After execution of of the block body, the final expression is evaluated and the value of that expression becomes the value of the entire block expression.

6.1 Variables

6.1.1. Syntax

⟨*simple τ variable*⟩ ::= ⟨τ *variable identifier*⟩
| ⟨τ *field designator*⟩
| ⟨τ *array designator*⟩

⟨τ *variable*⟩ ::= ⟨*simple τ variable*⟩

⟨*string variable*⟩ ::= ⟨*substring designator*⟩

⟨τ *field designator*⟩ ::= ⟨τ *field identifier*⟩ '(' ⟨*reference expression*⟩ ')'

⟨τ *array designator*⟩ ::= ⟨τ *array identifier*⟩ '(' ⟨*subscript list*⟩ ')'

⟨*subscript list*⟩ ::= ⟨*subscript*⟩
| ⟨*subscript list*⟩ ',' ⟨*subscript*⟩

⟨*subscript*⟩ ::= ⟨*integer expression*⟩

6.1.2. Semantics

An array designator denotes the variable whose indices are the current values of the expressions in the subscript list. The value of each subscript must lie within the declared bounds for that subscript position.

A field designator designates a field in the record referred to by its reference expression. The simple type of the field designator is defined by the declaration of that field identifier in the record class designated by the reference expression of the field designator (cf.5.4).

6.1.3. Examples

```
X       A(I)     M(I+J, I-J)
FATHER(JACK)     MOTHER(FATHER(JILL))
```

6.2 Function Designators

6.2.1. Syntax

⟨*τ function designator*⟩ ::= ⟨*τ function identifier*⟩
| ⟨*τ function identifier*⟩
'(' ⟨*actual parameter list*⟩ ')'

6.2.2. Semantics

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

Step 1. A copy is made of the body of the function procedure whose procedure identifier is given by the function designator and of the actual parameters of the latter.

Steps 2, 3, 4. As specified in 7.3.

Step 5. The copy of the function procedure body, modified as indicated in steps 2–4, is executed. Execution of the expression which constitutes or is part of the modified procedure body consists of evaluation of that expression, and the resulting value is the value of the function designator. The simple type of the function designator is the simple type in the corresponding function procedure declaration.

6.2.3. Examples

```
MAX (X ** 2, Y ** 2)
SUM (I, 100, H(1))
SUM (I, M, SUM (J, N, A(I,J)))
YOUNGESTUNCLE (JILL)
SUM (I, 10, X(1) * Y(1))
HORNER (X, 10, 2.7)
```

6.3 Arithmetic Expressions

6.3.1. Syntax

⟨τ_3 *expression 5*⟩ ::= '+' ⟨τ_3 *expression 6*⟩
| '-' ⟨τ_3 *expression 6*⟩

⟨τ_0 *expression 5*⟩ ::= ⟨τ_1 *expression 5*⟩ '+' ⟨τ_2 *expression 6*⟩
| ⟨τ_1 *expression 5*⟩ '-' ⟨τ_2 *expression 6*⟩

⟨τ_0 *expression 6*⟩ ::= ⟨τ_1 *expression 6*⟩ '*' ⟨τ_2 *expression 7*⟩
| ⟨τ_1 *expression 6*⟩ '/' ⟨τ_2 *expression 7*⟩

⟨*integer expression 6*⟩ ::= ⟨*integer expression 6*⟩ **div** ⟨*integer expression 7*⟩
| ⟨*integer expression 6*⟩ **rem** ⟨*integer expression 7*⟩

⟨τ_4 *expression 7*⟩ ::= ⟨τ_5 *expression 7*⟩ '**' ⟨*integer expression 8*⟩

⟨τ_4 *expression 8*⟩ ::= **abs** ⟨τ_5 *expression 8*⟩
| **long** ⟨τ_5 *expression 8*⟩
| **short** ⟨τ_5 *expression 8*⟩

⟨*integer expression 8*⟩ ::= ⟨*control identifier*⟩

6.3.2. Semantics

An arithmetic expression is a rule for computing a number. According to its simple type it is called an integer expression, real expression, long real expression, complex expression, or long complex expression.

6.3.2.1 The operators +, -, *, and / have the conventional meanings of addition, subtraction, multiplication and division.

For the operator *, the second "triplet rule" is modified so that τ_0 has the quality *long* unless both τ_1 and τ_2 are *integer*.

For the operator /, the "triplet rules" except when both τ_1 τ_2 are *integer*, then τ_0 is *long real*.

6.3.2.2 The operator - standing as the first symbol of a expression at priorty level 5 denotes the monadic operation of sign inversion. The type of the result is the type of the operand. The operator + standing as the first symbol of a simple expression denotes the monadic operation of identity.

In the relevant syntactic rules of 6.3, every occurrence of the symbol τ_3 must be systematically replaced by one of the following words (or word pairs):

integer
real

long real
complex
long complex

6.3.2.3 The operator **div** is mathematically defined (for `B` $\neq 0$) as

```
A div B = SGN(A × B) × D(abs A, abs B)
```

(cf. 6.3.2.5) `A` and `B` both must be integer expressions.
For the purpose of the definition above, `SGN` and `D` mean

```
integer procedure SGN (integer value A);
    if A < 0 then -1 else 1;

integer procedure D (integer value A, B);
    if A < B then 0 else D(A - B, B) + 1
```

6.3.2.4 The operator **rem** (remainder) is mathematically defined as

```
A rem B = A − (A div B) × B
```

6.3.2.5 The operator `**` denotes exponentiation of the first operand to the power of the second operand. In the relevant syntactic rule of 6.3. the symbols τ_4, τ_5 are to be replaced by any of the following combinations of words:

τ_4	τ_5
long real	*integer*
long real	*real*
long complex	*complex*

τ_4 has the quality *long* whether or not τ_5 has.

6.3.2.6 The monadic operator **abs** yields the absolute value or modulus of the operand. In the relevant syntactic rule of 6.3. the symbols τ_4 and τ_5 have to be replaced by any of the following combinations of words:

τ_4	τ_5
integer	*integer*
real	*real*
real	*complex*

If τ_5 has the quality *long* then so does τ_5.

6.3.2.7 Precision of arithmetic. If the result of an arithmetic operation is of simple type *real*, *complex*, *long real*, or *long complex* then it value is defined by System/360 arithmetic and is the mathematically understood result of the operation performed on operands which may deviate from actual operands.

In the relevant syntactic rules of 6.3 the symbols τ_4, τ_5 must be replaced by the following combinations of words (or word pairs):

Operator **long**

τ_4	τ_5
long real	*real*
long real	*integer*
long complex	*complex*

Operator **short**

τ_4	τ_5
real	*long real*
complex	*long complex*

6.3.3. Examples

```
C + A(1) * B(1)
EXP(-X/(2 * SIGMA)) / SQRT (2 * SIGMA)
```

6.4 Logical Expressions

6.4.1. Syntax

In the following rules for ⟨*relation*⟩ the symbols τ_6 and τ_7 must either be identically replaced by any one of the following words:

bit
string
reference

or by any of the words from:

complex
long complex
real
long real
integer

and the symbols τ_8 and τ_9 must be identically replaced by *string* or must be replaced by any of *real*, *long real*, *integer*.

⟨*logical expression 1*⟩ ::= ⟨*logical expression 1*⟩
| ⟨*logical expression 1*⟩ **or** ⟨*logical expression 2*⟩

⟨*logical expression 2*⟩ ::= ⟨*logical expression 2*⟩
| ⟨*logical expression 2*⟩ **and** ⟨*logical expression 3*⟩

⟨*logical expression 3*⟩ ::= '¬' ⟨*logical expression 4*⟩

⟨*relation*⟩ ::= ⟨τ_6 *expression 5*⟩ ⟨*equality operator*⟩ ⟨τ_7 *expression 5*⟩
| ⟨τ_8 *expression 5*⟩ ⟨*inequality operator*⟩ ⟨τ_9 *expression 5*⟩
| ⟨*reference expression 5*⟩ **is** ⟨*record class identifier*⟩

⟨*equality operator*⟩ ::= '=' | '¬='

⟨*inequality operator*⟩ ::= '<' | '<=' | '>=' | '>'

6.4.2. Semantics

A logical expression is a rule for computing a logical value.

6.4.2.1 The relational operators represent algebraic ordering for arithmetic arguments and EBCDIC ordering for string arguments. If two strings of unequal length are compared, the shorter string is considered to be extended to the length of the longer (for comparision only) by appending blanks to the right. The relational operators yield the logical value **true** if the relation is satisfied for the values of the two operands; **false** otherwise. Two references are equal if and only if they are both **null** or both refer to the same record. The operator **is** yields the *logical* value **true** if the reference expression designates a record of the indicated record class; **false** otherwise. The reference value **null** fails to designate a record of any record class.

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 **and** Y	**if** X **then** Y **else false**
X **or** Y	**if** X **then true else** Y

6.4.3. Examples

```
P or Q
(X < Y) and (Y < Z)
YOUNGESTOFFSPRING (JACK) 1 = null
FATHER (JILL) is PERSON
```

6.5 Bit Expressions

6.5.1. Syntax

⟨*bit expression 1*⟩ ::= ⟨*bit expression 1*⟩ **or** ⟨*bit expression 2*⟩

⟨*bit expression 2*⟩ ::= ⟨*bit expression 1*⟩ **and** ⟨*bit expression 3*⟩

⟨*bit expression 3*⟩ ::= '¬' ⟨*bit expression 4*⟩

⟨*bit expression 7*⟩ ::= ⟨*bit expression 7*⟩ **shl** ⟨*integer expression 8*⟩
| ⟨*bit expression 7*⟩ **shr** ⟨*integer expression 8*⟩

6.5.2. Semantics

A bit expression is a rule for computing a bit sequence.

The operators **and**, **or**, 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 and Y	X or Y
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

The operators **shl** and **shr** denote the shifting operation to the left and to the right respectively by the number of bit positions indicated by the absolute value of the integer primary. Vacated bit positions to the right or left respectively are assigned the bit value 0.

6.5.3. Examples

```
G and H or #38
G and ¬(H or G) shr 8
```

6.6 String Expressions

6.6.1. Syntax

⟨*substring designator*⟩ ::= ⟨*string variable*⟩
‘(’ ⟨*integer expression*⟩ ⟨*bar*⟩ ⟨*integer constant*⟩
‘)’

6.6.2. Semantics

A string expression is a rule for computing a string (sequence of characters).

6.6.2.1 A substring designator denotes a sequence of characters of the string designated by the string variable. The integer expression preceding the ⟨*bar*⟩ selects the starting character of the sequence. The value of the expression indicates the position in the string variable. The value must be greater than or equal to 0 and less than the declared length of the string variable. The first character of the string has position 0. The integer number following the ⟨*bar*⟩ indicates the length of the selected sequence and is the length of the string expression. The sum of the integer expression and the integer number must be less than or equal to the declared length of the string variable.

6.6.3. Examples

```
string (10) S;
S(4|3)
S(I+J|1)
string (10) array T (1::M,2::N);
T(4,6)(3|5)
```

6.7 Reference Expressions

6.7.1. Syntax

⟨*reference expression 8*⟩ ::= ⟨*record designator*⟩

⟨*record designator*⟩ ::= ⟨*record class identifier*⟩
| ⟨*record class identifier*⟩ ‘(’ ⟨*expression list*⟩ ‘)’

⟨*expression list*⟩ ::= ⟨*empty*⟩
| ⟨τ *expression*⟩
| ⟨*expression list*⟩ ‘,’ ⟨τ *expression*⟩

6.7.2. Semantics

A reference expression is a rule for computing a reference to a record.

The value of a record designator is the reference to a newly created record belonging to the designated record class. If the record designator contains an expression list, then the length of the list must equal the number of fields specified in the class declaration. Values of nonempty expressions in the expression list are assigned to the corresponding fields of the new record, and the simple types of the expressions must be assignment compatible with the simple types of the record fields (cf.7.2).

6.7.3. Example

```
PERSON ("CAROL", 0, false, JACK, JILL, null, YOUNGESTOFFSPRING(JACK))
NODE ( , null)
```

6.8 Conditional Expressions

6.8.1. Syntax

⟨*conditional* τ *expression*⟩ ::= ⟨*case clause*⟩ '(' ⟨τ *expression list*⟩ ')'

⟨*conditional* τ_0 *expression*⟩ ::= ⟨*if clause*⟩ ⟨τ_1 *expression*⟩ **else** ⟨τ_2 *expression*⟩

⟨τ *expression list*⟩ ::= ⟨τ *expression*⟩

⟨τ_0 *expression list*⟩ ::= ⟨τ_1 *expression list*⟩ ',' ⟨τ_2 *expression*⟩

⟨*if clause*⟩ ::= **if** ⟨*logical expression*⟩ **then**

⟨*case clause*⟩ ::= **case** ⟨*integer expression*⟩ **of**

6.8.2. Semantics

The construction

⟨*if clause*⟩ ⟨τ_1 *expression*⟩ **else** ⟨τ_2 *expression*⟩

causes the selection and evaluation 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 expression following the **if** clause is selected; if the value is **false**, the expression following **else** is selected. If τ_1 and τ_2 are type *string*, the length of the resulting expression is the maximum of the lengths of the component expressions; if necessary, blanks are appended on the right right of the shorter string. The construction

⟨*case clause*⟩ '(' ⟨τ *expression list*⟩ ')'

causes the selection of the expression whose ordinal number in the expression list is equal to the current value of the integer expression contained in the **case** clause. In order that the **case** expression be defined, the current value of this expression must be the ordinal number of some expression in the expression list. type If τ is type *string*, the length of the resulting expression is the maximum of the lengths of the stings in the expression list. If necessary, the length of any shorter element is increased by appending blanks on the right.

6.8.3. Examples

```
X   -1   A*B   COLUMN rem 5  (X+Y)*3   long abs BALANCE
if X=3 then Y+37 else Z*2.1
case I of (3.14, 2.78, 448.9)
case DECODE(C) - 128 of ("A", "B", "C", "D", "E", "F")
```

7 Statements

A statement denotes 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 action such as the evaluation of expressions or the execution of other statements.

7.0.4. Syntax

⟨*program*⟩ ::= ⟨*statement*⟩ '.'
| ⟨*proper procedure declaration*⟩ '.'
| ⟨τ *function procedure declaration*⟩ '.'

⟨*statement*⟩ ::= ⟨*simple statement*⟩
| ⟨*iterative statement*⟩
| ⟨*if statement*⟩
| ⟨*case statement*⟩

⟨*simple statement*⟩ ::= ⟨*block*⟩
| ⟨τ *assignment statement*⟩
| ⟨*empty*⟩
| ⟨*procedure statement*⟩
| ⟨*goto statement*⟩
| ⟨*standard procedure statement*⟩
| ⟨*assert statement*⟩
| ⟨*empty*⟩

Programs which are procedure declarations cannot be executed directly, but the corrsponding procedure bodies can form part of the environment in which other ALGOL W programs are executed (cf.5.3).

7.1 Blocks

7.1.1. Syntax

⟨*block*⟩ ::= ⟨*block body*⟩ ⟨*statement*⟩ **end**

⟨*block body*⟩ ::= ⟨*block head*⟩
| ⟨*block body*⟩ ⟨*statement*⟩ ';'
| ⟨*block body*⟩ ⟨*label definition*⟩

⟨*block head*⟩ ::= **begin**
| ⟨*block head*⟩ ⟨*declaration*⟩ ';'

⟨*label definition*⟩ ::= ⟨*identifier*⟩ ':'

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, say `A`, defined in the block head or in a label definition of the block body is already defined at the place from which the block is entered, then every occurrence of that identifier, A, within the block except for occurrence in array bound expressions is systematically replaced by another identifier, say `A'`, which is defined neither within the block nor at the place from which 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 block body begins with the execution of the first statement following the block head.

After execution of the last statement of the block body (unless it is a goto statement) a block exit occurs, and the statement following the entire block is executed.

7.1.3. Example

```
begin real U;
      U := X; X := Y; Y := Z; Z := U
end
```

7.2 Assignment Statements

7.2.1. Syntax

In the following rules the symbols τ_0 and τ_1 must be replaced by words as indicated in Section 1, subject to the restriction that the type τ_1 is assignment compatible with the type τ_0 as defined in 7.2.

$$\begin{array}{lll} \langle \tau_0 \textit{ assignment statement} \rangle & ::= & \langle \tau_0 \textit{ left part} \rangle \ \langle \tau_1 \textit{ expression} \rangle \\ & | & \langle \tau_0 \textit{ left part} \rangle \ \langle \tau_1 \textit{ assignment statement} \rangle \\ \langle \tau \textit{ left part} \rangle & ::= & \langle \tau \textit{ variable} \rangle \ \text{`:='} \end{array}$$

7.2.2. Semantics

The execution of a simple assignment statement

$$\langle \textit{assignment statement} \rangle ::= \langle \tau_0 \textit{ left part} \rangle \ \langle \tau_1 \textit{ expression} \rangle$$

causes the assignment of the value of the expression to the variable. If a shorter string is to be assigned to a longer one, the shorter string is first extended to the right with blanks until the lengths are equal. In a multiple assignment statement

$$\langle \textit{assignment statement} \rangle ::= \langle \tau_0 \textit{ left part} \rangle \ \langle \tau_1 \textit{ assignment statement} \rangle$$

the assignments are performed from right to left. For each left part variable, the simple type of the expression or assignment variable immediately to the right must be assignment compatible with the simple type of that variable.

A simple type τ_0 is said to be *assignment compatible* with a simple type τ_1 if either

(1) the two types are identical (except that if τ_0 and τ_1 are *string*, the length of the τ_0 variable must be greater than or equal to the length of the τ_1 expression or assignment), or

(2) τ_0 is *real* or *long real*, and τ_1 is *integer*, *real* or *long real*, or

(3) τ_0 is *complex* or *long complex*, and τ_1 is *integer*, *real*, *long real*, *complex* or *long complex*.

In the case of a reference, the reference to be assigned must refer to a record of one of the classes specified by the record class identifiers associated with the reference variable in its declaration.

7.2.3. Examples

```
z := AGE(JACK) := 28
X := Y + abs Z
C := I + X + C
P := X1 ¬= Y
```

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*⟩ ::= ⟨τ *expression*⟩
| ⟨*statement*⟩
| ⟨τ *subarray designator*⟩
| ⟨*procedure identifier*⟩
| ⟨τ *function identifier*⟩

⟨τ *subarray designator*⟩ ::= ⟨τ *array identifier*⟩
| ⟨τ *array identifier*⟩ '(' ⟨*subarray designator list*⟩ ')'

⟨*subarray designator list*⟩ ::= ⟨*subscript*⟩
| '*'
| ⟨*subarray designator list*⟩ ',' ⟨*subscript*⟩
| ⟨*subarray designator list*⟩ ',' '*'

7.3.2. Semantics

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

Step 1. A copy is made of the body of the proper procedure whose procedure identifier is given by the procedure statement, and of the actual parameters of the latter. The procedure statement is replaced by the copy of the procedure body.

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.

Step 3. The copies of the actual parameters are treated in an undefined order as follows: If the copy is an expression different from a variable, then it is enclosed by a pair of parentheses, or if it is a statement it is enclosed by the symbols **begin** and **end**.

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 W expressions and statements.

Step 5. The copy of the procedure body, modified as indicated in steps 2-4, is executed.

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 The following table summarises the forms of actual parameters which may be substituted for each kind of formal parameter's specification.

Formal type	**Actual parameter**
⟨τ *type*⟩	⟨τ *expression*⟩
⟨τ_0 *type*⟩ **value**	⟨τ_1 *expression*⟩
⟨τ_1 *type*⟩ **result**	⟨τ_0 *variable*⟩
⟨τ_1 *type*⟩ **value result**	⟨τ_2 *variable*⟩
⟨τ *type*⟩ **procedure**	⟨τ *function identifier*⟩ | ⟨τ *expression*⟩
procedure	⟨*procedure identifier*⟩ | ⟨*statement*⟩
⟨τ *type*⟩ **array**	⟨τ *subarray designator*⟩

The type τ_1 must be assignment compatible with type τ_0. τ_1 and τ_2 must be mutually assignment compatible.

7.3.2.3 Subarray designators. A complete array may be passed to a procedure by specifying the name of the array if the number of subscripts of the actual parameter equals the number of subscripts of the corresponding formal parameter. If the actual array parameter has more subscripts than the corresponding formal parameter, enough subscripts must be specified by

integer expressions so that the number of ∗'s appearing in the subarray designator equals the number of subscripts of the corresponding formal parameter. The subscript positions of the formal array designator are matched with the positions with ∗'s in the subarray designator in the order they appear.

7.3.3. Examples

```
INCREMENT
COPY (A, B, M, N)
INNERPRODUCT (IP, N, A(I,*), B(*,J))
```

7.4 Goto Statements

⟨*goto statement*⟩ ::= **goto** ⟨*label identifier*⟩
| **go to** ⟨*label identifier*⟩

7.4.1. Semantics

An identifier is called a label identifier if it stands as a label.

A **goto** statement determines that execution of the text be continued after the label definition of the label identifier. The identification of that label definition is accomplished in the following steps:

Step 1. If some label definition within the most recently activated but not yet terminated block contains the label identifier, then this is the designated label definition. 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

⟨*if statement*⟩ ::= ⟨*if clause*⟩ ⟨*statement*⟩
| ⟨*if clause*⟩ ⟨*simple statement*⟩ **else** ⟨*statement*⟩

⟨*if clause*⟩ ::= **if** ⟨*logical expression*⟩ **then**

7.5.2. Semantics

The execution of **if** statements causes certain statements to be executed or skipped depending on the values of specified logical expressions. An **if** statement of the form

⟨*if clause*⟩ ⟨*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 statement following the **if** clause is executed. Otherwise Step 2 causes no action to be taken at all.

An **if** statement of the form

⟨*if clause*⟩ ⟨*simple statement*⟩ **else** ⟨*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 simple statement following the **if** clause is executed. Otherwise the statement following **else** is executed.

7.5.3. Examples

```
if X = Y then goto L
if X < Y then U := X else if Y < Z then U := Y else V := Z
```

7.6 Assert Statements

7.6.1. Syntax

⟨*assert statement*⟩ ::= **assert** ⟨*logical expression*⟩

7.6.2. Semantics

The **assert** statement is equivalent to the **if** statement:

if (⟨*logical expression*⟩) **then** *endexecution*

where *endexecution* signifies a procedure which terminates the execution of an ALGOL W program. The assert statement can be used both as a debugging aid (asserting conditions which should be true, but may not be if a bug exists), and as a program documentation aid.

7.7 Case Statements

7.7.1. Syntax

⟨*case statement*⟩ ::= ⟨*case clause*⟩ **begin** ⟨*statement list*⟩ **end**

⟨*statement list*⟩ ::= ⟨*statement*⟩
| ⟨*statement list*⟩ ';' ⟨*statement*⟩

⟨*case clause*⟩ ::= **case** ⟨*integer expression*⟩ **of**

7.7.2. 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 statement list is equal to the value obtained in Step 1 is executed. In order that the case statement be defined, the current value of the expression in the case clause must be the ordinal number of some statement of the statement list.

7.7.3. Examples

```
case I of
begin X := X + Y;
      Y := Y + Z;
      Z := Z + X
end

case J of
begin H(1) := -H(I);
      begin H(I-1) := H(I-1) + H(I); I := I - 1 end;
      begin H(I-1) := H(I-1) * H(I); I := I - 1 end;
      begin H(H(I-1)) := H(I); I := I - 2 end
end
```

7.8 Iterative Statements

⟨*iterative statement*⟩ ::= ⟨*for clause*⟩ ⟨*statement*⟩
| ⟨*while clause*⟩ ⟨*statement*⟩

⟨*for clause*⟩ ::= **for** ⟨*identifier*⟩ ':=' ⟨*initial value*⟩ **step** ⟨*increment*⟩ **until** ⟨*limit*⟩ **do**
| **for** ⟨*identifier*⟩ ':=' ⟨*initial value*⟩ **until** ⟨*limit*⟩ **do**
| **for** ⟨*identifier*⟩ ':=' ⟨*for list*⟩ **do**

⟨*for list*⟩ ::= ⟨*integer expression*⟩
| ⟨*for list*⟩ ',' ⟨*integer expression*⟩

⟨*initial value*⟩ ::= ⟨*integer expression*⟩

⟨*increment*⟩ ::= ⟨*integer expression*⟩

⟨*limit*⟩ ::= ⟨*integer expression*⟩

⟨*while clause*⟩ ::= **while** ⟨*logical expression*⟩ **do**

7.8.1. Semantics

The iterative statement serves to express that a statement be executed repeatedly depending on certain conditions specified by a **for** clause or a **while** clause. The statement following the **for** clause or the **while** clause always acts as a block, whether it has the form of a block or not. The value of the control identifier (the identifier following **for**) cannot be changed by assignment within the controlled statement.

(a) An iterative statement of the form

for ⟨*identifier*⟩ := E_1 **step** E_2 **until** E_3 **do** ⟨*statement*⟩

is exactly equivalent to the block

begin ⟨*statement-0*⟩; ⟨*statement-1*⟩; ...
⟨*statement-I*⟩; ...
⟨*statement-N*⟩
end

when in the I^{th} statement every occurrence of the control identifier is replaced by the value of the expression $(E_1 + I \times E_2)$.

The index N of the last statement is determined by $N < (E_3 - E_1)/E2 < N + 1$. If $N < 0$, then it is understood that the sequence is empty. The expressions E_1, E_2, and E_3 are evaluated exactly once, namely before

execution of ⟨*statement-0*⟩. Therefore they can not depend on the control identifier.

(b) An iterative statement of the form

for ⟨*identifier*⟩ := E_1 **until** E_3 **do** ⟨*statement*⟩

is exactly equivalent to the iterative statement

for ⟨*identifier*⟩ := E_1 **step** 1 **until** E_3 **do** ⟨*statement*⟩

(c) An iterative statement of the form

for ⟨*identifier*⟩ := E_1, E_2, ..., E_N **do** ⟨*statement*⟩

is exactly equivalent to the block

begin ⟨*statement-1*⟩ ; ⟨*statement-2*⟩; ...
⟨*statement-I*⟩ ; ...
⟨*statement-N*⟩
end

when in the I^{th} statement every occurrence of the control identifier is replaced by the value of the expression E_I.

(d) An iterative statement of the form

while E **do** ⟨*statement*⟩

is exactly equivalent to the block

begin
L: **if** E **then**
begin ⟨*statement*⟩ ; **goto** L **end**
end

where it is understood the L represents an identifier which is not defined at the place from which the **while** statement is entered.

7.9 Standard Procedures

Standard procedures are provided in ALGOL W for the purpose of communication with the input/output system. A standard procedure differs from an explicitly declared procedure in that the number and type of its actual parameters need not be identical in every statement which invokes the standard procedure identifier appears.

Syntax:

⟨*standard procedure statement*⟩ ::= 'READ' '(' ⟨*input parameter list*⟩ ')'
| 'READON' '(' ⟨*input parameter list*⟩ ')'
| 'READCARD' '(' ⟨*input parameter list*⟩ ')'
| 'WRITE' '(' ⟨*transput parameter list*⟩ ')'
| 'WRITEON' '(' ⟨*transput parameter list*⟩ ')'
| 'IOCONTROL' '(' ⟨*transput parameter list*⟩ ')'

⟨*input parameter list*⟩ ::= ⟨τ *variable*⟩
| ⟨*simple statement*⟩
| ⟨*input parameter list*⟩ ',' ⟨τ *variable*⟩
| ⟨*input parameter list*⟩ ',' ⟨*simple statement*⟩

⟨*transput parameter list*⟩ ::= ⟨τ *expression*⟩
| ⟨*simple statement*⟩
| ⟨*transput parameter list*⟩ ',' ⟨τ *variable*⟩
| ⟨*transput parameter list*⟩ ',' ⟨*simple statement*⟩

7.9.1 The Input/Output System

ALGOL W provides a single legible input stream and a single legible output stream. These streams are conceived as sequences of records, each record consisting of a character sequence of fixed length. The input stream has the logical properties of a sequence of cards in a card reader; records consist of 80 characters. The output stream has the logical properties of a sequence of lines on a line printer; records consist of 132 characters, and the records are grouped into logical pages. Each page consists of not less than one nor more than 60 lines.

Input records may be transmitted as strings without analysis. Alternatively, it is possible to invoke a procedure which will scan the sequence of records for data items to be interpreted as numbers, bit sequences, strings, or logical values. If such analysis is specified, data items may be reference denotations of the corresponding constants (cf. Section 4). In addition, the following forms of arithmetic expressions are acceptable data items, and the

corresponding simple types are those determined by the rules for expressions (cf. 6.3):

(1) ⟨*sign*⟩ ⟨τ *constant*⟩

where τ is one of: *integer*, *real*, *long real*, *complex*, *long complex*

(2) ⟨τ_0 *constant*⟩ ⟨*sign*⟩ ⟨τ_1 *constant*⟩ | ⟨*sign*⟩ ⟨τ_0 *constant*⟩ ⟨*sign*⟩ ⟨τ_1 *constant*⟩

where τ_0 is one of: *integer*, *real*, *long real*,
and τ_1 is one of: *complex*, *long complex*

Data items are separated by one or more blanks. Scanning for data items initially begins with the first character of the input stream; after the initial scan, it normally begins with the character following the one which terminated the most recent previous scan. Leading blanks are ignored. The scan is terminated by the first blank following the data item. In the process, new records are fetched as necessary; character position 80 of one record is considered to be immediately followed by character position 1 of the next record. There exist procedures to cause the scanning process to begin with the first character of a record; if scanning would not otherwise start there, a new record is fetched.

Output items are assembled into records by an editing procedure. Items are automatically converted to character sequences and placed in fields according to the simple type of each item, as described below. The first field transmitted begins the output stream; thereafter, each field is normally placed immediately following the most recent previously transmitted field. If, however, the field corresponding to an item cannot be placed entirely within a non-empty record, that item is made the first field of the next record. In addition, there exist procedures to cause the field corresponding to an item to begin a new record. Each page group is automatically terminated after 60 records; procedures are provided for causing earlier termination.

7.9.2 Read Statements

Both `READ` and `READON` designate free field input procedures. Input records are scanned as described in 7.9.1. Values on input records are read, matched with the variables of the actual parameter list in order of appearance, and assigned to the corresponding variables. The simple type of each data item must be assignment compatible with the simple type of the corresponding variable. For each `READ` statement, scanning for the first data item is caused to begin with the first character of a record; for a `READON` statement, scanning continues from the previous point of termination as determined by prior use of `READ`, `READON`, or `IOCONTROL` (cf. 7.9.1).

READCARD designates a procedure transmitting 80 character input records without analysis. For each variable of the actual parameter list, the scanning process is set to begin at the first character of a record (by fetching a new record if necessary), all 80 characters of that record are assigned to the corresponding string variable, and subsequent input scanning is set to begin at the first character of the next sequential record.

7.9.3 Write Statements

WRITE and WRITEON designate output procedures with automatic format conversion. Values of expressions of the actual parameter list are converted to character fields which are assembled into output records in order of appearance (cf. 7.9.1). For each WRITE statement, the field corresponding to the first value is caused to begin an output record; for a WRITEON statement, assembly continues from the previous point of termination.

The values of a set of predeclared *editing variables* controls the field widths and the formats of numerical quantities printed by the standard ALGOL W output routines. These variables are initialized to appropriate default settings; their values can be inspected and modified in the course of the execution of the ALGOL W program. Their attributes are given in the following table:

Identifier	Type	Initial Value	Interpretation
I_W	*integer*	14	Width of *integer* fields.
R_FORMAT	*string(1)*	"F"	Format of *real*, *long real*, *complex* and *long complex* fields.
R_W	*integer*	14	Width of real and long real fields; width of complex and long complex fields ($2 \times$ R_W $+ 2$)
R_D	*integer*	0	Places following the decimal point in *real*, *long real*, *complex* and *long complex* fields.
S_W	*integer*	2	Width of the fields of blanks appended to the end of each fields (excluding *string* fields).

Values of I_W and R_W control the output field widths used for numerical quantities, in conjunction with the values of of S_W they determine the layout of each line of numerical output. Integer quantities are converted according to a standard format, but three different formats for the representation of *real*, *long real*, *complex* and *long complex* values (strictly, rounded approximations

of these values) are available. For a particular output value, the actual format is determined by the interrogation of the the variable `R_FORMAT`, which must specify one of the following:

(1) *scaled* format (`R_FORMAT = "S"`), in which the legible representation takes the form of a normalized mantissa followed by an explict scale factor;

(2) *aligned* format (`R_FORMAT = "A"`), in which the representation includes an integeral part, a frctional part with the specified number of digits, but no scale factor;

(3) *free-point* format (`R_FORMAT = "F"`), in which the representation is chosen to use a specified number of significate digits, with the diecimal point suitably positioned and with a scale factor only if necessary.

Scaled and aligned representations are sometimes said to use "scientific" and " fixed-point" notation respectively. If scaled or free-point format is specified, the number of significant digits printed is given by `R_W - 7`. If (but only if) aligned format is specified, the number of digits following the decimal point is controlled by the value of `R_D`, and the magnitude of the numerical quantity determines then number of significant digits printed.

The field in which an output item is placed depends on the type of the item, as follows:

Simple Type	Field Description
integer	Right justified in a field of `I_W` characters and followed by `S_W` blanks.
real	Right justified in a field of `R_W` characters and followed by `S_W` blanks.
long real	Right justified in a field of `R_W` characters and followed by `S_W` blanks.
complex	Right justified in a field of (2×`R_W`+2) characters and followed by `S_W` blanks.
long complex	Right justified in a field of (2×`R_W`+2) characters and followed by `S_W` blanks.
logical	Right justified in a field of 6 characters followed by `S_W` blanks.
string	Field length is exactly the length of the string.
bits	Right justified in a field of 14 characters and followed by `S_W` blanks.

Parameters corresponding to the syntactic class ⟨*simple statement*⟩ are executed as they are encountered in the corresponding output lists; they

cause no values to be transmitted but they can (and normally should) serve to change the values of the editing variables or the state of the input/output system. Furthermore, the values of the five predeclared editing variables `I_W`, `R_W`, `R_D`, `R_FORMAT` and `S_W` are automatically saved at the beginning of execution of `WRITE` or `WRITEON` statements and restored at the end. Thus changes to the values of these variables within a output statement are localised and can affect only the editing of the remaining elements of the list, but assignments outside of such a list can affect all subsequent editing.

7.9.4 Control Statements

`IOCONTROL` designates a procedure which affects the state of the input/output system. Argument values with defined effect are listed below; other values currently have no effect but are explicitly made available for local use or future expansion.

Value	**Action** (cf. 7.9.1)
1	Subsequent input scanning begins with the first character of a record.
2	Subsequent output assembly begins with the first field of a record.
3	Subsequent output assembly begins with the first field of a record which, in turn, begins a new output page.
4	Subsequent output assembly has no provision for automatic page skips.
5	Subsequent output assembly contains control characters providing automatic page skips. (Initial option.)

7.9.5. Examples

```
READCARD(S, LINE(10|180))
WRITE("AVERAGE␣=", SUM / N)
WRITEON(X(1,J))
IOCONTROL(2)
```

Execution of the program,

```
begin
  procedure SCALED (integer value N);
    begin R_FORMAT := "S"; R_W := N+7
    end;
  procedure ALIGNED (integer value N,D);
    begin R_FORMAT := "A"; R_W := N+D+1; R_D := D
    end;
  procedure FREE_POINT (integer value N);
    begin R_FORMAT := "F"; R_W := N+7
    end;
  procedure NEW_LINE; IOCONTROL(2);

  FREE_POINT(5); I_W := 2; S_W := 1;

  for I := -1, 0, 32 do
    begin WRITE(S_W := 0, I, ":", NEW_LINE, I/3);
    WRITEON("I␣␣", ALIGNED(3,2), I/3, "*", SCALED(12), I/3, "*")
    end
end.
```

will produce the following lines:

```
-1:
␣␣␣␣-0.33333I␣␣␣-0.33␣*␣-3.33333333333'-01␣*
␣0:
␣␣␣␣␣␣␣␣␣␣␣0I␣␣␣␣0.00␣*␣␣␣␣␣␣␣␣␣␣␣␣␣␣0␣␣␣␣␣*
32:
␣␣␣␣␣␣10.667I␣␣␣10.67␣*␣␣1.06666666667'+01␣*
```

Note that the setting of S_W when the corresponding quantity is transmitted determines the number of trailing blanks; also, edited valued are always rounded.

Any values assigned to I_W, R_W, S_W in excess of 132 are treated as 132. In the event that the values of I_W, R_W, R_D, S_W, or R_FORMAT are erroneous or inconsistent with the magnitude or precision of the number to be transmitted then alternative values are used. These values ensure that an

X	TRUNCATE(X)	ENTIER(X)	ROUND(X)
2.3	2	2	2
2.5	2	2	3
2.7	2	2	3
-2.3	-2	-3	-2
-2.5	-2	-3	-3
-2.7	-2	-3	-3

Table 3: Values for TRUNCATE, ENTIER, and ROUND

approximation to the number is always transmitted and that not more digits than are warranted by the precision of then number are transmitted.

8 Standard Functions and Predeclared Identifiers

The ALGOL W environment includes declarations and initialization of certain procedures and variables which supplement the language facilities previously described. Such declarations and initialization are considered to be included in a block which encloses each ALGOL W program (with terminating period eliminated).

8.1 Standard Transfer Functions

Certain functions for conversion of values from one simple type to another are provided. These functions are predeclared; the corresponding implicit declaration headings are listed below:

integer procedure TRUNCATE **(real value** X**);**
comment the integer i such that $i \leq |\mathtt{X}| < |i|$ and $i \times \mathtt{X} \geq 0$;

integer procedure ENTIER **(real value** X**);**
comment the integer i such that $i \leq \mathtt{X} < i + 1$;

integer procedure ROUND **(real value** X**);**
comment the value of the expression**;**
if X < 0 **then** TRUNCATE(X-0.5) **else** TRUNCATE(X+0.5)**;**

integer procedure EXPONENT **(real value** X**);**
comment 0 if $\mathtt{X} = 0$, otherwise the largest integer i such that $i \leq log_{16}(|\mathtt{X}|)+1$. This function obtains the exponent used in the S/360 representation of the real number ;

real procedure ROUNDTOREAL **(long real value** X**);**
comment the properly rounded value of X ;

real procedure REALPART **(complex value** Z**);**
comment the real component of Z ;

long real procedure LONGREALPART **(long complex value** Z**);**

real procedure IMAGPART **(complex value** Z**);**
comment the imaginary component of Z ;

long real procedure LONGIMAGPART **(long complex value** Z**);**

complex procedure IMAG **(real value** X**);**
comment the complex number $0 + \mathtt{X}i$;

long complex procedure LONGIMAG **(long real value** X**);**

logical procedure ODD **(integer value** N**);**
comment the logical value; N **rem** 2 = 1;

bits procedure BITSTRING **(integer value** N**);**
comment two's complement representation of N ;

integer procedure NUMBER **(bits value** X**);**
comment integer with two's complement representation X ;

integer procedure DECODE **(string**(1) **value** S**);**
comment numeric code for the character S (cf. Appendix A) ;

string(1) **procedure** CODE **(integer value** N**);**
comment character with numeric code given by **abs** (N **rem** 256) (cf. Appendix A) ;

In the following comments, the significance of characters in the prototype formats is as follows:

D	decimal digit in a mantissa or integer
E	decimal digit in an exponent
A	hexadecimal digit in a mantissa or integer
B	hexadecimal digit in an exponent
±	sign (blank for positive mantissa or integer)
␣	blank

Each exponent is unbiased. Decimal exponents represent powers of 10; hexadecimal exponents represent powers of 16. Each mantissa (except 0) represents a normalized fraction less than one. Leading zeroes are not suppressed.

string(12) **procedure** BASE10 (**real value** X);
comment string encoding of X with format ␣±EE±DDDDDDD ;

string(12) **procedure** BASE16 (**real value** X);
comment string encoding of X with format ␣␣±BB±AAAAAA ;

string(20) **procedure** LONGBASE10 (**long real value** X);
comment string encoding of X with format ␣±EE±DDDDDDDDDDDDDD ;

string(20) **procedure** LONGBASE16 (**long real value** X);
comment string encoding of X with format ␣␣±BB±AAAAAAAAAAAAAA ;

string(12) **procedure** INTBASE10 (**integer value** N);
comment string encoding of N with format ␣±DDDDDDDDD ;

string(12) **procedure** INTBASE16 (**integer value** N);
comment unsigned, two's complement string encoding of N with format ␣␣␣␣AAAAAAAA ;

8.2 Standard Functions of Analysis

The following functions of analysis are provided in the system environment. In some cases, they are partial functions; action for arguments outside of the allowed domain is described in 8.5. These functions are predeclared; the corresponding implicit declaration headings are listed below:

real procedure SQRT (**real value** X);
comment the positive square root of X, domain : $X \geq 0$;

long real procedure LONGSQRT (**long real value** X);
comment the positive square root of X, domain : $X \geq 0$;

real procedure EXP (**real value** X);
comment e^X, domain : $X < 174.67$;

long real procedure LONGEXP (**long real value** X);
comment e^X, domain : $X < 174.67$;

real procedure LN (**real value** X);
comment logarithm of X to the base e, domain : $X \geq 0$;

long real procedure LONGLN (**long real value** X);
comment logarithm of X to the base e, domain : $X \geq 0$;

real procedure `LOG (real value X);`
comment logarithm of `X` to the base 10, domain : `X` ≥ 0 ;

long real procedure `LONGLOG (long real value X);`
comment logarithm of `X` to the base 10, domain : `X` ≥ 0 ;

real procedure `SIN (real value X);`
comment sine of `X` (radians), domain : $-823550 <$ `X` < 823550 ;

long real procedure `LONGSIN (long real value X);`
comment sine of `X` (radians), domain : $-3.537\times10^{15} <$ `X` $< 3.537\times10^{15}$;

real procedure `COS (real value X);`
comment cosine of `X` (radians), domain : $-823550 <$ `X` < 823550 ;

long real procedure `LONGCOS (long real value X);`
comment cosine of `X` (radians), domain : $-3.537\times10^{15} <$ `X` $< 3.537\times10^{15}$;

real procedure `ARCTAN (real value X);`
comment arctangent (radians) of `X`, range : $-\pi/2 <$ `ARCTAN(X)` $< \pi/2$;

long real procedure `LONGARCTAN (long real value X);`
comment arctangent (radians) of `X`, range : $-\pi/2 <$ `LONGARCTAN(X)` $< \pi/2$;

8.3 Time Function

The Algol W environment includes a clock which measures elapsed time since the beginning of program execution. The resolution of that clock is 1/60 second. A predeclared function is provided for reading the clock.

integer procedure `TIME (integer value N);`

Argument `N`	Result	Units
-1	time of day	seconds/60
0	elapsed execution time	minutes/100
1	elapsed execution time	seconds/60
2	elapsed execution time	seconds/38400

The result for any other argument is not defined.

8.4 Predeclared Variables

The following variables are to be considered declared and initialized by assignment in the conceptual block enclosing the entire ALGOL W program.

The values indicated for *real* and *long real* quantities are to be understood as decimal approximations to the actual machine-format values provided.

integer `I_W`;
comment initialized to 14, controls output field size for integers (cf. 7.9.1) ;

integer `R_W`;
comment initialized to 14, controls output field size for *real*, *long real*, *complex* and *long complex* quantities (cf. 7.9.1) ;

integer `R_D`;
comment initialized to 0, specifies the number of fraction digits in aligned formats (cf. 7.9.1) ;

integer `R_FORMAT`;
comment initialized to `"F"`, controls output format for *real*, *long real*, *complex* and *long complex* quantities (cf. 7.9.1) ;

integer `S_W`;
comment initialized to 2, specifies the number of blanks append to to the end of an ouput numberic field (cf. 7.9.1) ;

integer `MAXINTEGER`;
comment initialized to 2147483647, the maximum positive *integer* allowed by the implementation ;

real `EPSILON`;
comment initialized to 9.536743×10^{-7}, the largest positive *real* number ϵ provided by the implementation such that $1 + \epsilon = 1$;

long real `LONGEPSILON`;
comment initialized to $2.22044604925031 \times 10^{-16}$, the largest positive *long real* number ϵ provided by the implementation such that $1 + \epsilon = 1$;

long real `MAXREAL`;
comment initialized to $7.23700557733226 \times 10^{75}$, the largest positive *long real* number provided by the implementation ;

`low` **real** `PI`;
comment initialized to 3.14159265358979 ;

8.5 Exceptional Conditions

The facilities described below are provided in ALGOL W to allow detection and control of certain exceptional conditions arising in the evaluation of arithmetic expressions and standard functions.

Implicit declarations:

```
record EXCEPTION (
    logical XCPNOTED;
    integer XCPLIMIT, XCPACTION;
    logical XCPMARK;
    string(64) XCPMSG );

reference(EXCEPTION)
    OVFL, UNFL, DIVZERO,
    INTOVFL, INTDIVZERO,
    SQRTERR, EXPERR, LNLOGERR, SINCOSERR;
```

Associated with each exceptional condition which can be processed is a predeclared reference variable to which references to records of the class `EXCEPTION` can be assigned. Fields of such records control the processing of exceptions. The association between conditions and reference variables is as follows:

Reference Variable	**Conditions**
`ENDFILE`	end of file detected in input
`OVFL`	*real*, *long real*, *complex*, *long complex* (exponent) overflow
`UNFL`	*real*, *long real*, *complex*, *long complex* (exponent) underflow
`DIVZERO`	*real*, *long rea*l, *complex*, *long complex* division by zero
`INTOVFL`	*integer* overflow
`INTDIVZERO`	*integer* division by zero
`SQRTERR`	negative argument for `SQRT`, `LONGSQRT`
`EXPERR`	argument of `EXP`, `LONGEXP` out of domain (cf. 8.2)
`LNLOGERR`	argument of `LN`, `LOG`, `LONGLN`, `LONGLOG` out of domain (cf. 8.2)
`SINCOSERR`	argument of `SIN`, `COS`, `LONGSIN`, `LONGCOS` out of domain (cf. 8.2)

When one of the conditions listed above is detected, the corresponding reference variable is interrogated, and one of the alternatives described below is chosen.

Table 5: Results for Exceptional Conditions

condition	XCPACTION ≠ 1 or 2	XCPACTION = 1	XCPACTION = 2	Reference = **null**
ENDFILE[1]	0	0	0	0
OVFL	exponent 128 too small	±MAXREAL	0	exponent 128 too small
UNFL	exponent 128 too large	0	0	0
DIVZERO	dividend	±MAXREAL	0	dividend
INTOVFL	true result $\pm 2^{32}$	true result $\pm 2^{32}$	true result $\pm 2^{32}$	true result $\pm 2^{32}$
INTDIVZERO	dividend	dividend	dividend	dividend
SQRTERR	0	SQRT(**abs** X)	0	0
EXPERR	0	MAXREAL	0	0
LNLOGERR	0	-MAXREAL	0	0
SINCOSERR	0	0	0	0

[1] When an ENDFILE condition occurs on attempting to read a *string*, a string of blanks is supplied; for a logical value **false** is supplied.

If the value of the reference variable interrogated is **null**, the condition is ignored and execution of the Algol W program continues. In such situations, a value of 0 is returned as the value of a standard function. For other conditions the result is that provided by the underlying IBM System/360 hardware. In determining such a result, it is to be noted that in those cases in which the detection of exceptional conditions can be inhibited at the hardware level, namely integer overflow and exponent underflow, detection is so inhibited when the corresponding reference is **null**.

If the value of the reference variable interrogated is not **null**, the fields of the record designated by that reference are interrogated, and processing action is that described by the algorithm given below in the form of an extended Algol W procedure. Identifiers in *lower case* represent quantities which transcend the Algol W language; they are explained subsequently.

```
procedure PROCESSEXCEPTION (reference(EXCEPTION) value CONDITION);
   begin
      XCPNOTED(CONDITION) := true;
      XCPLIMIT(CONDITION) := XCPLIMIT(CONDITION) - 1;
      if (XCPLIMIT(CONDITION) < 0) or XCPMARK(CONDITION) then
         WRITE( "*****␣ERROR␣NEAR␣COORDINATE␣nnnn - ",
                XCPMSG(CONDITION) );
      if XCPLIMIT(CONDITION) < 0 then endexecution else
      if specialcondition then
      resultant := default else
      resultant := if XCPACTION(CONDITION) = 1 then adjustment
                   else if XCPACTION(CONDITION) = 2 then 0L
                   else default
   end PROCESSEXCEPTION
```

This procedure is invoked with the value of the reference variable appropriate to the condition as actual parameter. The significance of the special identifiers used is as follows:

nnnn approximate coordinate of the source code which was being executed when the exceptional condition was detected

endexecution procedure to terminate execution of the ALGOL W program

specialcondition logical value which is true if, and only if, the condition being processed is one of those listed below

default result of the operation or function provided by the ALGOL W system prior to invocation of the exception processing procedure; this is defined by the hardware for arithmetic operations and is the value 0 for standard functions

resultant value to be returned as the result of the arithmetic evaluation or standard function invocation

adjustment adjusted result of the operation according to the following table:

Special Condition	Adjustment
Exponent overflow, division by zero	**if** *default* < 0 **then** -MAXREAL **else** MAXREAL
Exponent underflow	0L
Argument X out of domain for:	
SQRT, LONGSQRT	SQRT(**abs** X), LONGSQRT(**abs** X)
EXP, LONGEXP	MAXREAL
LN, LONGLN	-MAXREAL
LOG, LONGLOG	-MAXREAL
SIN, LONGSIN	0L
COS, LONGCOS	0L
ENDFILE on input; according to type:	
numerical	0
logical	**false**
string	"␣"
bits	#0

The reference variable UNFL is initialized by the system to **null**. All other reference variables listed above are initialized to references to a special record which is accessible only by the system. Interrogation of this record by the procedure described above has the effect of causing the ALGOL W program to be terminated with a message indicating the type of exception. Any other attempt to access any field of this record will result in a reference error.

8.5.1. Example

It is desired to allow up to ten overflows, but to each time replace the result with MAXREAL and to print a warning message.

The values needed for this are:

XCPNOTED	**false**	this will be changed to **true** if an overflow occurs.
XCPLIMIT	10	allow up to ten overflows before being cut off.
XCPACTION	1	replace the result with +MAXREAL.
XCPMARK	**true**	print a message each time an overflow occurs.
XCPMSG	"..."	message to be printed.

The following assignment statement will establish the proper environment:

```
OVFL := EXCEPTION(false, 10, 1, true, "OVERFLOW␣FIXED␣UP");
```

A Character Encodings

The following table presents the correspondence between printable string characters and their (EBCDIC) integer encodings. This encoding establishes the ordering relation on characters and thus on strings. Those characters in parentheses are not available on the line printer. Integer codes not listed below do not correspond to any established character. (Also see `CODE`, `DECODE` on page 48.)

64	*space*	129	(a)	193	A	240	0
		130	(b)	194	B	241	1
74	(¢)	131	(c)	195	C	242	2
75	.	132	(d)	196	D	243	3
76	<	133	(e)	197	E	244	4
77	(	134	(f)	198	F	245	5
78	+	135	(g)	199	G	246	6
79	|	136	(h)	200	H	247	7
80	&	137	(i)	201	I	248	8
						249	9
90	(!)	145	(j)	209	J		
91	$	146	(k)	210	K		
92	*	147	(l)	211	L		
93	)	148	(m)	212	M		
94	;	149	(n)	213	N		
95	¬	150	(o)	214	O		
96	-	151	(p)	215	P		
97	/	152	(q)	216	Q		
		153	(r)	217	R		
107	,						
108	%	162	(s)	226	S		
109	_	163	(t)	227	T		
110	>	164	(u)	228	U		
111	?	165	(v)	229	V		
		166	(w)	230	W		
122	:	167	(x)	231	X		
123	#	168	(y)	232	Y		
124	@	169	(z)	233	Z		
125	’						
126	=						
127	"						

Index of Syntactic Entities

⟨τ *array declaration*⟩, 13, **14**
⟨τ *array designator*⟩, **22**
⟨τ *array identifier*⟩, **7**, 22, 34
⟨τ *assignment statement*⟩, 31, **33**
⟨τ *block expression*⟩, **21**
⟨τ *constant*⟩, **10**–**12**, 21
⟨τ *expression 1*⟩, **21**, **27, 28**
⟨τ *expression 2*⟩, **21**, **27, 28**
⟨τ *expression 3*⟩, **21**, **27, 28**
⟨τ *expression 4*⟩, **21**, 27, 28
⟨τ *expression 5*⟩, **21**, **24**, 27
⟨τ *expression 6*⟩, **21**, **24**
⟨τ *expression 7*⟩, **21**, **24**, **28**
⟨τ *expression 8*⟩, **21**, **24**, 28, **29**
⟨τ *expression*⟩, 14, 16, **21**, 22, 29, 30, 33–39, 41
⟨τ *field designator*⟩, **22**
⟨τ *field identifier*⟩, **7**, 22
⟨τ *function designator*⟩, **23**
⟨τ *function identifier*⟩, **7**, 23, 34, 35
⟨τ *function procedure declaration*⟩, **16**, 31
⟨τ *subarray designator*⟩, **34**, 35
⟨τ *variable identifier*⟩, **7**, 22
⟨τ *variable*⟩, **22**, 29, 33, 35, 41
⟨*actual parameter list*⟩, 23, **34**
⟨*actual parameter*⟩, **34**
⟨*assert statement*⟩, 31, **37**
⟨*bar*⟩, **4**, 6, 29
⟨*bit constant*⟩, **11**
⟨*block body*⟩, 16, 21, **32**
⟨*block*⟩, 31, **32**
⟨*case clause*⟩, **30**, **38**
⟨*case statement*⟩, 31, **38**
⟨*complex constant*⟩, **10**
⟨*conditional τ expression*⟩, 21, **30**
⟨*control identifier*⟩, **7**, 24
⟨*declaration*⟩, **13**, 32
⟨*empty*⟩, 4, 29, 31
⟨*equality operator*⟩, **27**
⟨*external reference*⟩, 16, **16**
⟨*for clause*⟩, **39**
⟨*formal array parameter*⟩, 16
⟨*formal parameter list*⟩, **16**
⟨*formal type*⟩, 16, **16**
⟨*goto statement*⟩, 31, **36**
⟨*identifier list*⟩, **7**, 13, 14, 16
⟨*identifier*⟩, **7**, 16, 19, 32, 39
⟨*if clause*⟩, **30**, **36**
⟨*if statement*⟩, 31, **36**
⟨*inequality operator*⟩, **27**
⟨*integer constant*⟩, **10**, 13, 29
⟨*iterative statement*⟩, 31, **39**
⟨*label definition*⟩, **32**
⟨*label identifier*⟩, **7**, 36
⟨*logical constant*⟩, **11**
⟨*long complex constant*⟩, **10**
⟨*long real constant*⟩, **10**
⟨*procedure declaration*⟩, 13
⟨*procedure heading*⟩, **16**
⟨*procedure identifier*⟩, **7**, 34, 35
⟨*procedure statement*⟩, 31, **34**
⟨*program*⟩, **31**
⟨*proper procedure declaration*⟩, **16**, 31
⟨*real constant*⟩, **10**
⟨*record class declaration*⟩, 13, **19**
⟨*record class identifier*⟩, **7**, 13, 27, 29
⟨*record designator*⟩, **29**
⟨*reference constant*⟩, **12**
⟨*relation*⟩, **27**
⟨*simple τ variable*⟩, **22**

⟨*simple statement*⟩, **31**, 36, 41
⟨*simple type*⟩, **13**, 14, 16
⟨*simple variable declaration*⟩, 13, **13**, 19
⟨*standard procedure statement*⟩, 31, **41**
⟨*statement*⟩, 16, **31**, 32, 34–36, 38, 39
⟨*string constant*⟩, **12**, 16
⟨*subscript*⟩, **22**, 34
⟨*substring designator*⟩, 22, **29**
⟨*while clause*⟩, **39**

Words with Special Meanings in Algol W

¬, 27, 28
*, 24
**, 25
+, 24
-, 24
/, 24

abs, 24, 25
algol, 16
and, 27, 28
ARCTAN, 50
array, 14
assert, 37

BASE10, 48
BASE16, 49
bits, 13
BITSTRING, 48

case, 30, 38
CODE, 48
comment, 6
complex, 13
COS, 50

DECODE, 48
div, 24, 25
DIVZERO, 52

else, 30, 37
end, 6
ENDFILE, 52
ENTIER, 47
EPSILON, 51
EXCEPTION, 52
EXP, 49
EXPERR, 52
EXPONENT, 47

false, 11, 27

for, 8, 39
fortran, 16

goto, 36

I_W, 43, 46, 51
if, 30, 36
IMAG, 48
IMAGPART, 48
INTBASE10, 49
INTBASE16, 49
INTDIVZERO, 52
integer, 13
INTOVFL, 52
IOCONTROL, 42, 45, 46
is, 27

LN, 49
LNLOGERR, 52
LOG, 49
logical, 13
long, 24, 26
long complex, 13
long real, 13
LONGARCTAN, 50
LONGBASE10, 49
LONGBASE16, 49
LONGCOS, 50
LONGEPSILON, 51
LONGEXP, 49
LONGIMAG, 48
LONGIMAGPART, 48
LONGLN, 49
LONGLOG, 50
LONGREALPART, 48
LONGSIN, 50
LONGSQRT, 49

MAXINTEGER, 51

MAXREAL, 51

null, 12, 13, 27
NUMBER, 48

ODD, 48
of, 30
or, 27, 28
OVFL, 52

PI, 51
procedure, 16, 17

R_D, 43, 46, 51
R_FORMAT, 43, 46, 51
R_W, 43, 46, 51
READ, 42
READON, 42
real, 13
REALPART, 48
record, 19
reference, 13
rem, 24, 25
result, 16, 18
ROUND, 47
ROUNDTOREAL, 48

S_W, 43, 46, 51
shl, 24, 28
short, 24
shr, 24, 28
SIN, 50
SINCOSERR, 52
SQRT, 49
SQRTERR, 52
step, 39
string, 13

then, 30
TIME, 50
true, 11, 27
TRUNCATE, 47

UNFL, 52
until, 39

value, 16, 17

while, 39, 40
WRITE, 43, 46
WRITEON, 43, 46

XCPACTION, 52
XCPLIMIT, 52
XCPMARK, 52
XCPMSG, 52
XCPNOTED, 52

Index

Arithmetic expression, **24**
Array
    Bound pair, **15**, 22, 32
    Declaration, **14**
    Dimension, **14**, 36
    Element, **15**
    Indices, **15**
assert statement, **37**
Assignment
    Assignment statement, **33**
    Compatibility, 30, 33, **33**, 35

Bit expression, **28**
Block, **32**
Boolean expression, **26**
Built-in functions, **47**

Call, *see* Procedure
case expression, **30**
case statement, **38**
Characters, *see* System 360, EBCDIC, **12**
Comments, **6**
Conditional expression, **30**
Constants, **9, 10**
Control identifier, **7**, 8, *see* **for** Statement, **39**
Conversions, **25**

Data types, 9
Declaration, **8**, **13**

EBCDIC
    String comparison, **27**
Exceptional conditions, **52**
    Overflow, **53**
    Underflow, **53**
Execution, **6**
Expression, **20**
External References, **18**

Field, *see* Record
for statement, **39**
Fortran, *see* External References
Function, *see* Procedure
    Declaration, **16**
    Designator, **23**
    Procedure, **10**, 17, 23

goto statement, **36**

Identifier
    Binding, **8**
if expression, **30**
if statement, **36**
Input/Output System, **41**, **45**
    Constants for input, **41**
    Editing variables, **43**
    Page eject, **45**
Iterative statements, **39**

Keywords, **6**

Label, **32**
    Identifier, **8**, **36**
Logical expression, **26**

Not defined, 6, **6**, 13, 31
Not valid, **6**, 15, 17

Operators, **6**
    Precedence, **20**, 22

Parameter
    Actual, 35
    Actual-formal correspondence, **35**
    Formal, **8**, **17**, 35
Predeclared identifiers, **51**

Predefined variables, **8**
Procedure, **10**
    Body, **17**, 18, 23, 34
    Call, 23, **34**
    Copy rule, **34**
    Declaration, **16**
    Name parameter, **34**
    Parameter, **34**
    Proper, **10**, 17, 18, 34
Programs, syntax of, **5**

Quantities, **8**

Record
    Class Declaration, **19**
    Creation, **30**
    Field designator, **19**, 22
Reference
    Declaration, **14**
    Denotation, **9, 10**, 41
    Expression, **29**
Reserved words, **7**

Scope, **9**, 13, 32
Simple value, **9**
Simple variable, 13
Standard functions, **8**, **47**
Standard procedures, **8**, **41**
Statement, **31**
String
    default length, **14**
String expression, **29**
Structured value, 9, 13
Subarray, **35**
Substring, **29**
Syntactic entities, **7**
System 360, **47**
    EBCDIC, 6, 12, 27, **56**
    floating point, 47
    Input/Output, **41**
    Operations, **6**, **26**

Transfer functions, **47**
Triplet rules, **21**, 24
Type, **9**, 13
    array, 10
    bits, 10, 14, 28
    complex, 10, 25
    integer, 10, 24
    logical, 10, 11, 27, 37, 40
    long complex, 10, 25
    long real, 10, 25
    real, 10, 24
    record, 10
    reference, 10, 14
    string, 10, 14, 30, 31

Unit of action, **6**
Units of action, 20, 31

Value, **9**, 14
Variable, **9**, 10, 13, 14, **22**
    Simple, **13**

while statement, **40**