CS 110

ALGOL W (REVISED)

COMPUTER SCIENCE DEPARTMENT

STANFORD UNIVERSITY

SEPTEMBER 1969

ALGOL W

LANGUAGE DESCRIPTION

by

Henry R. Bauer
Sheldon Becker
Susan L. Graham
Edwin Satterthwaite

"A Contribution to the Development of ALGOL" by Niklaus Wirth and C. A. R. Hoare[1] was the basis for a compiler developed for the IBM 360 at Stanford University. This report is a description of the implemented language, ALGOL W. Historical background and the goals of the language may be found in the Wirth and Hoare paper.

1) Wirth, Niklaus and Hoare, C. A. R., "A Contribution to the Development of ALGOL", Comm. ACM 9, 6(June 1966), pp. 413-431.

CONTENTS

CONTENTS (cont.)

APPENDIX

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, the set of basic constituents of the language,
(2) U, the set of syntactic entities,- 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) enclosed in the brackets < and >. A syntactic rule has the form

<A> ::= x

where <A> is a member of U, x is any possible sequence of basic constituents and syntactic entities, simply to be called a "sequence". The form

<A> ::= x | y | ... | z

is used as an abbreviation for the set of syntactic rules

<A> ::= x
<A> ::= y
. . . . . .
<A> ::= z

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 <A> in u, x = u<A>w, y = uvw, and <A> ::= v is a rule in ρ; 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 τ, and x can be produced from the syntactic entity <program>.

The sets ν and u are defined through enumeration of their members in Section 2 of this Report (cf. also 4.4.). The syntactic rules are 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

<T term> ::= <T factor> (cf. 6.3.1.)

corresponds to

<integer term> ::= <integer factor>
<real term> ::= <real factor>
<long real term> ::= <long real factor:,
<complex term> ::= <complex factor>
<long complex term> ::= <long complex factor>

The production

$<T_0$ primary> ::= long $<T_1$ primary> (cf. 6.3.1. and table for long 6.3.2.7.)

corresponds to

<long real primary> ::= long <real primary2
<long real primary> ::= long <integer primary>
<long complex primary> ::= long <complex primary>

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. 3.1.) 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 | ¬ | | | = | ¬ = | < |
< = | > | > = | :: |
:= | goto | go to | for | step | until | do | while |
comment | value | result

All underlined 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. 3.1.) 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

(with corresponding section numbers)

3. IDENTIFIERS

3.1. Syntax

<identifier> : : = <letter> | <identifier> <letter> | <identifier> <digit>

<T variable identifier> ::= <identifier3

<T array identifier3 ::= <identifier>
<procedure identifier> ::= <identifier>
<T function identifier> ::= <identifier>
<record class identifier> ::= <identifier>
<T 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>

3.2. 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) a declaration (cf. Section 5), if the identifier identifies a quantity. It is then said to denote that quantity and to be a T variable identifier, T array identifier, T procedure identifier, T function identifier, record class identifier or T field identifier, where the symbol T 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. 7.7.). It is then said to be a control identifier;

(e) its implicit declaration in the language. Standard procedures, standard functions, and predefined variables (cf. 7.8 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. 7.7.) 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.

3.3. Examples

I

PERSON

ELDERSIBLING

x15, X20, x25

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 pot 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. 6.2.2.).

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 number> ::= <complex number>L

<complex number> ::= <imaginary number>

<imaginary number> ::= <real number>I | <integer number>I

<long real number> : : = <real number>L | <integer number>L

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

<unscaled real> ::= <integer number> . <integer number> |
*<integer number> | <integer number>.

<scale factor> : := '<integer number> | '<sign> <integer number>

<integer number> : := <digit> | <integer number> <digit>

<sign> ::= + | -

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 <unsigned number>s are unsigned.)

4.1.3. Examples

1	.5	11
0100	1'3	0.671
3.1416	6.02486'+23	1IL
2.718281828459045235360287L	2.3'-6	

4.2. Logical Values

4.2.1. syntax

<logical value> : := true | false

4.3. Bit Sequences

4.3.1. syntax

<bit sequence> : := # <hex digit> | <bit sequence> <hex digit>

<hex digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B |
C | D | E | F

Note that 2 | . . . | F corresponds to 2_{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> ::= "<sequence of characters>"

4.42. Semantics

Strings consist of any sequence of (at moat 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.

4.4.3. Examples

"JOHN"

"""" is the string of length 1 censiating of the string quote.

4.5. References

4.5.1. Syntax

<null reference> ::= 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 (e.g. 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. 7.1.2. and 7.4.2.).

Syntax:

<declaration> ::= <simple variable declaration> | <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 number>) | reference
(<record class identifier list>)

<record class identifier list> ::= <record class identifer> |
<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 4). If a variable is declared to be of a certain type, then this implies that only values which are assignment compatible with this type (cf. 7.2.2.) can be assigned to it. It is understood that the value of a variable 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. Examples

```
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

<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. Examples

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> |
                             <T function procedure declaration>
<proper procedure declaration> ::= procedure <procedure heading> ;
                                   <proper procedure body>
<T function procedure declaration> ::= <simple type> procedure
                                       <procedure heading>;
                                       <T function procedure body>
<proper procedure body>  ::= <statement>
<T function procedure body>  ::= <T expression> | <block body>
                                 <T expression> end
<procedure heading>  ::= <identifier> | <identifier> (<formal
                         parameter' list>)
<formal parameter list>  ::= <formal parameter segment> |
                             <formal parameter list> ; <formal
                             parameter segment9
<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
                              lists (<dimension specification>)
<dimension specification> ::= * | <dimension specification> , *
```

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. 7.3.2.).

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 if it is not already enclosed by these symbols;

(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 2a;

(3) If the formal type contains the symbol value, an assignment statement (cf. 7.2.) followed by a semicolon is inserted after the declarations 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 a proper procedure body. In the case of a function procedure, an assignment statement preceded by a semicolon is inserted after the final expression of the function 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.

5.3.3. 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(*); value N;
    real value X);
    begin real S; S := 0;
        for I := 0 until N do S := S * X + A(1);
        S
    end

long real procedure SUM (integer K, N; long real X);
    begin long real Y; Y := 0; K := N;
        while K >= 1 do
        begin Y := Y + X; K := K - 1
        end;
        Y
    end

reference (PERSON) procedure YOUNGESTUNCLE (reference (PERSON) R);
    begin reference (PERSON) P, M;
        P := YOUNGESTOFFSPRING (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 null P then M else
        if null M then P else
        if AGE(P) < AGE(M) then P else M
    end
```

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. 6.7.) 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,
EIDERSIBLING)

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 6.4.2.2. Several simple types of expressions are distinguished. Their structure is defined by the following rules, in which the symbol $\mathcal{T}$ has to be replaced consistently as described in Section 1, and where the triplets $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2$ have to be either all three replaced by the same one of the words

logical

bit

string

reference

or by any combination of words as indicated by the following table, which yields $\mathcal{T}_0$ given $\mathcal{T}_1$ and $\mathcal{T}_2$:

$\mathcal{T}_1$ \ $\mathcal{T}_2$	integer	real	complex
integer	integer	real	complex
real	real	real	complex
-complex	complex	complex	complex

$\mathcal{T}_0$ has the quality "long" if either both $\mathcal{T}_1$ and $\mathcal{T}_2$ have that quality, or if one has the quality and the other is "integer".

Syntax:

<$\mathcal{T}$ expression> ::= <simple $\mathcal{T}$ expression> | <case clause>
(<$\mathcal{T}$ expression list>)

<$\mathcal{T}_0$ expression> ::= <if clause> <$\mathcal{T}_1$ expression> else
<$\mathcal{T}_2$ expression>

<$\mathcal{T}$ expression list> ::= <$\mathcal{T}$ expression>

<$\mathcal{T}_0$ expression list> ::= <$\mathcal{T}_1$ expression list> , <$\mathcal{T}_2$ expression>

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

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

The construction

<if clause> <$\mathcal{T}_1$ expression> else <$\mathcal{T}_2$ 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 $\mathcal{T}_1$ and $\mathcal{T}_2$ are simple type string, both string expressions must have the same length. The construction

<case clause> (<$\mathcal{T}$ 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. If $\mathcal{T}$ is simple type string, all the string expressions must have the same length.

6.1. Variables

6.1.1. Syntax

<simple $\mathcal{T}$ variable> ::= <$\mathcal{T}$ variable identifier> | <$\mathcal{T}$ field designator> | <$\mathcal{T}$ array designator>

<$\mathcal{T}$ variable> ::= <simple $\mathcal{T}$ variable>

<string variable> ::= <substring designator>

<$\mathcal{T}$ field designator> ::= <$\mathcal{T}$field identifier> (<reference expression>)

<$\mathcal{T}$ array designator> ::= <$\mathcal{T}$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

<J function designator> ::= <J function identifier> | <J 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 7.3.2.

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

In any of the following rules, every occurrence of the symbol $\mathcal{T}$ must be systematically replaced by one of the following words (or word pairs):

integer
real
long real
complex
long complex

The rules governing the replacement of the symbols $\mathcal{T}_0$, $\mathcal{T}_1$ and $\mathcal{T}_2$ are given in 6.3.2.

<simple $\mathcal{T}$ expression> ::= <$\mathcal{T}$ term> | + <$\mathcal{T}$ term> | - <$\mathcal{T}$ term>

<simple $\mathcal{T}_0$ expression> ::= <simple $\mathcal{T}_1$ expression> + $\mathcal{T}_2$ term> |
<simple $\mathcal{T}_1$ expression> - $\mathcal{T}_2$ term>

$\mathcal{T}$ term> ::= $\mathcal{T}$ factor>

$\mathcal{T}_0$ term> ::= $\mathcal{T}_1$ term> * $\mathcal{T}_2$ factor>

$\mathcal{T}_0$ term> ::= $\mathcal{T}_1$ term> / $\mathcal{T}_2$ factor>

<integer term> ::= <integer term> div <integer factor> |
<integer term> rem <integer factor>

$\mathcal{T}_0$ factor> ::= $\mathcal{T}_0$ primary> | $\mathcal{T}_1$ factor> ** <integer primary>

$\mathcal{T}_0$ primary> ::= abs $\mathcal{T}_1$ primary>

$\mathcal{T}_0$ primary> ::= long $\mathcal{T}_1$ primary>

$\mathcal{T}_0$ primary> ::= short $\mathcal{T}_1$ primary>

$\mathcal{T}$ primary> ::= $\mathcal{T}$ variable> | $\mathcal{T}$ function designator> |
($\mathcal{T}$ expression>) | $\mathcal{T}$ number>

<integer primary> ::= <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. In the relevant syntactic rules of 6.3.1. the symbols $\mathcal{T}_0$, $\mathcal{T}_1$ and $\mathcal{T}_2$ have to be replaced by any combination of words according to the following table which indicates $\mathcal{T}_0$ for any combination of $\mathcal{T}_1$ and $\mathcal{T}_2$.

Operators + | -

$\mathcal{T}_1$ \ $\mathcal{T}_2$	integer	real	complex
integer	integer	real	complex
real	real	real	complex
complex	complex	complex	complex

$\mathcal{T}_0$ has the quality "long" if both $\mathcal{T}_1$ and $\mathcal{T}_2$ have the quality "long", or if one has the quality "long" and the other is "integer".

Operator *

$\mathcal{T}_1$ \ $\mathcal{T}_2$	integer	real	complex
integer	integer	long real	long complex
real	long real	long real	long complex
complex	long complex	long complex	long complex

$\mathcal{T}_1$ or $\mathcal{T}_2$ having the quality "long" does not affect the type of the result.

Operator /

$\mathcal{T}_1$ \ $\mathcal{T}_2$	integer	real	complex
integer	long real	real	complex
real	real	real	complex
complex	complex	complex	complex

$\mathcal{T}_0$ has the quality "long" if both $\mathcal{T}_1$ and $\mathcal{T}_2$ have the quality "long", or if one has the quality "long" and the other is "integer", or if both are "integer".

6.3.2.2. The operator "-" standing as the first symbol of a simple expression 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.

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. 6.3.2.6.)

where the function procedures SGN and D are declared as

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

integer procedure D (integer value A, B);
    if A C 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) x 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 6.3.1. the symbols $\mathcal{T}_0$ and $\mathcal{T}_1$ are to be replaced by any of the following combinations of words:

$\mathcal{T}_0$	$\mathcal{T}_1$
long real	integer
real	real
complex	complex

$\mathcal{T}_0$ has the quality "long" if $\mathcal{T}_1$ does or if $\mathcal{T}_1$ is "integer".

6.3.2.6. The monadic operator abs yields the absolute value or modulus of the operand. In the relevant syntactic rule of 6.3.1. the symbols $\mathcal{T}_0$ and $\mathcal{T}_1$ have to be replaced by any of the following combinations of words:

$\mathcal{T}_0$	$\mathcal{T}_1$
integer	integer
real	real
real	complex

If $\mathcal{T}_1$ has the quality "long", then so does $\mathcal{T}_0$.

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 is the mathematically understood result of the operation performed on operands which may deviate from actual operands.

In the relevant syntactic rules of 6.3.1. the symbols $\mathcal{T}_0$ and $\mathcal{T}_1$ must be replaced by any of the following combinations of words (or word pairs) :

Operator long

$\mathcal{T}_0$	$\mathcal{T}_1$
long real	real
long real	integer
long complex	complex

Operator short

$\mathcal{T}_0$	$\mathcal{T}_1$
real	long real
complex	long complex

6.3.3. Examples

-C + A(1) * B(I)

EXP (-X/(2 * SIGMA)) / SQRT (2 * SIGMA)

6.4. Logical Expressions

6.4.1. Syntax

In the following rules for <relation> the symbols $\mathcal{T}_0$ and $\mathcal{T}_1$ 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 $\mathcal{T}_2$ or $\mathcal{T}_3$ must be identically replaced by string or must be replaced by any of real, long real, integer.

<simple logical expression> ::= <logical element> | <relation>
<logical element> ::= <logical term> | <logical element> or <logical term>
<logical term> ::= <logical factor> | <logical term> and <logical factor>
<logical factor> ::= <logical primary> | ¬ <logical primary>
<logical primary> ::= <logical value> | <logical variable | <logical function designator> | (<logical expression>)
<relation> ::= <simple $\mathcal{T}_0$ expression> <equality operator> <simple $\mathcal{T}_1$ expression> | <logical element3 <equality operator> <logical element> | <simple reference expression> is <record class identifier> | <simple 3, expression> <relational operator> <simple $\mathcal{T}_3$ expression>
<relational operator> ::= < | < = | > = | >
<equality 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 extended to the right by characters less than any possible string character. 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. Two strings are equal if and only if they have the same length and the same ordered sequence of characters. 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) ¬ = null

FATHER (JILL) is PERSON

6.5. Bit Expressions

6.5.1. Syntax

<simple bit expression> ::= <bit term> | <simple bit expression> or <bit term>

<bit term> ::= <bit factor> | <bit term> and <bit factor>

<bit factor> ::= <bit secondary> | ¬ <bit secondary>

<bit secondary> ::= <bit primary> | <bit secondary> shl <integer primary> | <bit secondary> shr <integer primary>

<bit primary> ::= <bit sequence> | <bit variable> | <bit function designator> | (<bit expression>)

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

<simple string expression> ::= <string primary>

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

<substring designator> ::= <simple string variable> (<integer expression> | <integer number>)

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 ▮ 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 ▮ 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. Example

```
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

<simple reference expression) ::= <null reference | <reference variable> | <reference function designator> | <record designator> | (<reference expression>)

<record designator> ::= <record class identifier> | <record class identifier> (<expression list>)

<expression list> ::= ◁ expression> | <expression list>, ◁ 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 values of the expressions are assigned to the fields of the new record, The entries in the expression list are taken in the same order as the fields in the record class declaration, and the simple types of the expressions must be assignment compatible with the simple types of the record fields (cf. 7.2.2.).

6.7.3. Example

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

6.8. Precedence of Operators

The syntax of 6.3.1., 6.4.1., and 6.5.1. implies the following hierarchy of operator precedences:

long, short, abs

shl, shr, **

¬

*, /, div, rem, and

+, -, or

<, < =, =, ¬ =, > =, >, is

Example

A = B and C is equivalent to A = (B and C)

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.

Syntax:

<program> ::= <block> .

<statement> ::= <simple statement> | <iterative statement> | <if statement> | <case statement>

<simple statement> ::= <block> | <assignment statement> | <empty> | <procedure statement> | <goto statement>

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 **APRIME**, 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 $\mathcal{T}_0$ and $\mathcal{T}_1$ must be replaced by words as indicated in Section 1, subject to the restriction that the type $\mathcal{T}_1$ is assignment compatible with the type $\mathcal{T}_0$ as defined in 7.2.2.

$\langle \mathcal{T}_0$ assignment statement> ::= $\langle \mathcal{T}_0$ left part> $\langle \mathcal{T}_1$ expression> |
$\langle \mathcal{T}_0$ left part> $\langle \mathcal{T}_1$ assignment statement>

$\langle \mathcal{T}$ left part> ::= $\langle \mathcal{T}$ variable> :=

7.2.2. Semantics

The execution of a simple assignment statement

$\langle \mathcal{T}_0$ assignment statement> ::= $\langle \mathcal{T}_0$ left part> $\langle \mathcal{T}_1$ expression>:

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

(<$\mathcal{T}_0$ assignment statement> ::= $\langle \mathcal{T}_0$ left part> $\langle \mathcal{T}_1$ assignment statement>)

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 $\mathcal{T}_1$ is said to be assignment compatible with a simple type $\mathcal{T}_0$ if either

(1) the two types are identical (except that if $\mathcal{T}_0$ and $\mathcal{T}_1$ are string, the length of the $\mathcal{T}_0$ variable must be greater than or equal to the length of the $\mathcal{T}_1$ expression or assignment), or

(2) $\mathcal{T}_0$ is real or long real, and $\mathcal{T}_1$ is integer, real or long real or

(3) $\mathcal{T}_0$ is complex or long complex, and $\mathcal{T}_1$ 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 := X ¬= 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> ::= ◁ expression> | <statement> | ◁ subarray designator> | <procedure identifier> | ◁ function identifier>

◁ subarray designator> ::= ◁ array identifier> | ◁ 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 7.1.2.

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. Formal specifications. If a formal parameter is specified by value, then the simple type of the actual parameter must be assignment compatible with the formal type. If it is specified as result, then the formal type must be assignment compatible with the simple type of the actual parameter. If it is specified by value result, both the above

conditions must be satisfied. In all other cases, the types must be identical. If an actual parameter is a statement, then the specification of its corresponding formal parameter must be procedure.

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 Statments

7.4.1. Syntax

<goto statement> ::= goto <label identifier> | go to (label identifier>

7.4.2. 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. Case statements

7.6.1. Syntax

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

7.6.2. Semantics

The execution of a case statement proceeds in the following steps:

Step 1. The expression of the case clause is evaluated.

Step 2. The statement whose ordinal number in the 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.

7.6.3. Examples

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

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

7.7. Iterative Statements

7.7.1. Syntax

<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

7.7.2. 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> := E1 step E2 until E3 do <statement>

. is exactly equivalent to the block

begin <statement-0>; <statement-1> . . . ; <statement-I>;
... ; <statement-N> end

in the I^{th} statement every occurrence of the control identifier is replaced by the value of the expression (E1 + I x E2).

The index N of the last statement is determined by $N < (E3-E1) / E2 < N+1$. If $N < 0$, then it is understood that the sequence is empty. The expressions E1, E2, and E3 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> := E1 until E3 do <statement>

is exactly equivalent to the iterative statement

for <identifier> := E1 step 1 until E3 do <statement>

(c) An iterative statement of the form

for <identifier> := E1, E2, . . . , EN 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 EI.

(d) An iterative statement of the form

while E do <statement>

is exactly equivalent to

```
     begin
L:       if E then
         begin <statement> ; goto L end
     end
```

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

7.7.3. Examples

for V := 1 step 1 until N-1 do S := S + A(U,V)

while (J > 0) and (CITY(J) ¬= S) do J := J-1

for I := X, X + 1, X + 3, X + 7 do P(1)

7.8. Standard Procedures

Standard procedures are provided in ALGOL W for the purpose of communication with-the input/output system. These standard procedures differ from explicitly declared procedures in that the number and type of actual parameters need not be identical in every procedure statement in which the standard procedure identifier appears. In the following descriptions, each T_i is to be replaced by any one of

integer
real
long real
complex
long complex

string (<integer number>)
logical
bits

7.8.1. 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> <T number>

where : T is one of integer, real, long real, complex, long complex;

(2) <$\mathcal{T}_0$ number> <sign> <$\mathcal{T}_1$ number>

<sign> <$\mathcal{T}_0$ number> <sign> <$\mathcal{T}_1$ number>

where : $\mathcal{T}_0$ is one of integer, real, long real, and

$\mathcal{T}_1$ 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:

Simple Type	Field Description
integer	right justified in a field containing the number of characters specified by the current value of INTFIELDSIZE (initialized to 14, cf. 8.5.) and followed by 2 blanks
real	right justified in a field of 14 characters and followed by 2 blanks

long real	right justified in a field of 22 characters and followed by 2 blanks
complex	two adjacent real fields
long complex	two adjacent long real fields
logical	right justified in a field of 6 characters followed by 2 blanks
string	placed in a field exactly the length of the string
bits	same as real

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.

7.8.2. Read Statements

Implicit declaration headings:

procedure READ (T_1 result X_1; . . . ; T_n result X_n);

procedure READON (T_1 result X_1; . . . ; T_n result X_n);

(where n > = 1)

Both READ and READON designate free field input procedures. Input records are scanned as described in 7.8.1. 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. 7.8.1.).

Implicit declaration heading:

procedure **READCARD** (string(80) result X_1, ... X_n);

(where n > = 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.

7.8.3. Write Statements

Implicit declaration headings:

procedure WRITE (T_1 value X_1; . . . ; T_n value X_n);

procedure **WRITEON** (T_1 value X1; . . . ; T_n value X_n);

(where n > = 1)

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. 7.8.1.). 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.

7.8.4. Control Statements

Implicit declaration heading:

procedure IOCONTROL (integer value X_1, . . . , X_n);

(where n > = 1)

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. 7.8.1.)
1	Subsequent input scanning is set to begin with the first character of a record.
2	Subsequent output assembly is set to begin with the first field of a record.
3	Subsequent output assembly is set to begin with the first field of a record which, in turn, is caused to begin a new output page.

7.8.5. Examples

```
READ ( X, A(1) )
READCARD ( S, LINE(10|80) )
WRITE ( "AVERAGE =", SUM/N )
WRITEON ( X(1,J) )
IOCONTROL (2)
```

8. STANDARD FUNCTIONS AND PREDECURED 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). The corresponding identifiers are said to be predeclared.

8.1. Standard Transfer F-unctions

Certain functions for conversion of vaiues 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| < = |X| < |i| + 1 and i*X > = 0
integer procedure ENTIER (real value X);
     comment the integer i such that
          i < = X < i + 1 ;
integer procedure ROUND (real value X);
     comment the value of the integer expression
          if X < 0 then TRUNCATE(X-0.5) else TRUNCATE(X+0.5) ;
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 + Xi ;
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 1) ;
string(1) procedure CODE (integer value N);
     comment character with numeric code (cf. Appendix 1) given by
          abs (N rem 256) ;
```

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+DDDDDDDDDDDDDDD ;
string(20) procedure LONGBASE16 (long real value X);
     comment string encoding of X with format
                  ␣␣+BB+AAAAAAAAAAAAAAA ;
string(12) procedure INTBASE10 (integer value N);
     comment string encoding of N with format
                  ␣+DDDDDDDDDD ;
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>= 0 ;
long real procedure LONGSQRT (long real value X);
     comment the positive square root of X,
       domain : X > = 0 ;
```

```
complex procedure COMPLEXSQRT (complex value Z);
     comment principal square root of Z ;
long complex procedure LONGCOMPLEXSQRT (long complex value Z);
     comment principal square root of Z ;
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 > 0 ;
long real procedure LONGLN (real value X);
     comment logarithm of X to the base e,
       domain : X > 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'+15 < x < 3.537'+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'+15 < x < 3.537'+15 ;
```

```
real procedure ARCTAN (real value X);
     comment arctangent (radians) of X,
       range : -π/2 < ARCTAN(X) < π/2 ;
long real procedure LONGARCTAN (long real value X);
     comment arctangent (radians) of X,
       range : -π/2 < LONGARCTAN(X) < π/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);
     comment returns elapsed time, in hundredths of a minute if N=0,
       in sixtieths of a second otherwise;
```

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 INTFIELDSIZE; .
     comment initialized to 14 ,
       controls output field size for integers (cf. 7.8.1.);
integer MAXINTEGER;
     comment initialized to 2147483647 ,
       the maximum positive integer allowed by the implementation;
```

```
real EPSILON;
     comment initialized to 9.536743 '-07 ,
       the largest positive real number ε provided by the
       implementation such that
                 1 + ε = 1 ;
long real LONGEPSILON;
     comment initialized to 2.220446049250 31'-16L ,
       the largest positive long real number ε provided by
       the implementation such that
                 1 + ε = 1 ;
long real MAXREAL;
     comment initialized to 7.23700557733226'+75L ,
       the largest positive long'real number provided by the
       implementation;
long real PI;
     comment initialized to 3.14159265358979L ;
```

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
OVFL	real, long real,' complex, long complex (exponent) overflow
UNFL	real, long real, complex, long complex (exponent) underflow
DIVZERO	real, long real, 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.

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[2/]. 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("***** EXCEPTION NEAR CARD nnnn - ", XCPMSG(CONDITION));
     if XCPLIMIT(CONDITION) < 0 then endexecution else
     if integercondition then
     resultant := default else
     resultant := if XCPACTION(CONDITION) = 1 then adjustment else
                  if XCPACTION(CONDITION) = 2 then OL 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:

[2/] IBM System/360 Principles of Operation, IBM Systems Library, Form A22-6821

nnnn	approximate line number of the source code which was being executed when the exceptional condition was detected
endexecution	procedure to terminate execution of the ALGOL W program
integercondition	logical value which is true if, and only if, the condition being processed is integer overflow or integer division by zero
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[3] 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

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

[3] IBM System/360 Principles of Operation, IBM Systems Library, Form A22-6821

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.

APPENDIX 1 - 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.

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

ALGOL W

ERROR MESSAGES

by

Henry R. Bauer
Sheldon Becker
Susan L. Graham

ALGOL W ERROR MESSAGES

I. PASS ONE MESSAGES

All Pass One messages appear on the first page following the program listing. The message format is

CARD NO, (number) -- (message)

The (number) corresponds to the card number on which the error was found. The (message) is one of those listed below.

INCORRECT SPECIFTN	syntactic entity of a declaration is incorrect, e.g. variable string length.
INCORRECT CONSTANT	syntax error in number or bitstring.
MISSING END	an END needed to close block.
MISSING BEGIN	an attempt to close outer block before end of code.
MISSING)	) is needed.
ILLEGAL CHARACTER	a character, not in a string, is unrecognizable.
MISSING FINAL .	program must be terminated by a period.
STRING LNGTH ERROR	string is of 0 length or length greater than 256.
BITS LENGTH ERROR	bits constant denotes no bits or more than 32 bits.
MISSING (	(is needed.
TABLE OVERFLOW	terminating error - a compile time table has exceeded its bounds.

TOO MANY ERRORS	the maximum number of errors for Pass One records has been reached. Compilation continues but messages for succeeding errors detected by Pass One are suppressed.
ID LENGTH > 256	more than 256 characters in' identifier.

See alsc discussion of PROGRAM CHECK in IV.

II. PASS TWO MESSAGES

The format of Pass Two error messages is

(message), CARD NUMBER IS (number). CURRENT SYMBOL IS (incoming symbol)

If a $STACK card is included anywhere in the source deck, the SYNTAX ERROR message is followed by

STACK CONTAINS:

(beginning of file)

<symbol-1>

<symbol-n> (top of stack)

The symbol names may differ somewhat from the metasymbols of the syntax.

If any Pass One or Pass Two errors occur, compilation is terminated-at the end of Pass Two.

INCORRECT SIMPLE TYPE <number>	<simple type> of entity is improper as used. Number indicates explanation on list of simple type errors.

ARRAY USED INCORRECTLY	a variable must be used here.
IDENTIFIER MUST BE RECORD CLASS ID	reference declaration is incorrect,
MISMATCHED PARAMTER	formal parameter does not correspond to actual parameter.
MULTIPLY-DEFINED SYMBOL <identifier>	symbol defined more than once in a block
UNDEFINED SYMBOL <identifier>	symbol is not declared or defined.
INCORRECT NUMBER OF ACTUAL PARAMETERS	the number of actual parameters to a procedure does not equal the number of formal parameters declared for the procedure.
INCORRECT DIMENSION	the array has appeared previously with a different number of dimensions.
DATA AREA EXCEEDED	too many declarations in the block.
INCORRECT NUMBER OF FIELDS	the number of fields specified in a record designator does not equal the number of fields the declaration of the record indicates.
INCOMPATIBLE STRING LENGTH	length of assigned string is greater than length of string assigned to.
INCOMPATIBLE REFERENCES	record class bindings are inconsistent.
BLOCKS NESTED TOO DEEP	blocks are nested more than 7 levels.
REFERENCE MUST REFER TO RECORD CLASS	reference must be bound to a record class.
EXPRESSION MISSING IN PROCEDURE BODY	body of typed procedure must end with an expression.

RESULT PARAMETER MUST BE <T VAR>	the actual parameter corresponding to a result formal parameter must be a <T VARIABLE>.
PROCEDURE READ LACKS SIMPLE TYPE	proper procedure ends with an expression
<SYMBOL-1> UNRELATED TO <SYMBOL-2>	the symbol at the top of the stack (<SYMBOL-1>) should not be followed by the incoming symbol (<SYMBOL-2>).
SYNTAX ERROR	construction violates the rules of the grammar. The input string is skipped until the next END, ";", BEGIN, or the end of the program. More than one error message may be generated for a single syntax error.

Simple Type Errors

25. Upper and lower bounds must be integer.
29. Upper and lower bounds must be integer.
32. Simple type of procedure and simple type of expression in procedure body do not agree.
71. Substring index must be integer.
73. Simple variable preceding '(' must be string.
74. Substring length must be integer.
76. Field index must be reference or record class identifier.
77. Array subscript must be integer.
81. Array subscript must be integer.
84. Actual parameters and formal parameters do not agree.
88. Actual parameters and formal parameters do not agree.
93. Expressions in if expression do not agree.
94. Expressions in case expression do not agree.
95. Expression in if clause must be logical.

98. Expressions in case expression do not agree.

99. Expression in case clause must be integer.

101. Arguments of = or ¬= do not agree.

102. Arguments of relational operators must be integer, real, or long real.

103. Argument before is must be reference.

106. Argument of unary + must be arithmetic.

107. Argument of unary - must be arithmetic.

108. Arguments of + must be arithmetic.

109. Arguments of - must be arithmetic.

110. Arguments of or must be both logical or both bits.

112. Record field must be assignment compatible with declaration.

117. Arguments of * must be arithmetic.

118. Arguments of / must be arithmetic.

119. Arguments of div must be integer.

120. Arguments of rem must be integer.

121. Arguments of and must be both logical or both bits.

123. Argument of ¬ must be logical or bits.

125. Exponent or shift quantity must be integer; eqression to be shifted must be bits.

126. Shift quantity must be integer; expression to be shifted must be bits.

130. Actual parameter of standard function has incorrect simple type.

134. Argument of long must be integer, real, or complex.

135. Argument of short must be long real or long complex.

136. Argument of abs must be arithmetic.

148. Record field must be assignment compatible with declaration.

181. Expression is not assignment compatible with variable.

182. Result of assignment cannot be assigned to variable.

188. Limit expression in for clause must be integer.

190. Expression in for list must be integer.

191. Assignment to for variable must be integer.

193. Expression in for list must be integer.

195. Step element must be integer.

197. Expression in while clause must be logical.

III. PASS THREE ERROR MESSAGES

The form of Pass Three error messages is

***** (message)
***** NEAR CARD (number)

The number indicates the number of the card near which the error occurred. The message may be

PROGRAM SEGMENT OVERFLOW	the amount of code generated for a procedure exceeds 8192 bytes.
COMPILER STACK OVERFLOW	constructs nested too deeply.
CONSTANT POINTER TABLE TOO LARGE	too many literals appear in a procedure.
BLOCKS NESTED TOO DEEPLY	parameters in procedure call are nested too deeply; procedure calls in block nested too deeply.
DATA SEGMENT OVERFLOW	too many variables declared in the block.
TOO MANY PROCEDURES	the program contains too many procedure declarations; the number of procedures allowed depends on the size of each procedure and cannot exceed 52.
CARD TABLE OVERFLOW	density of information on (non-blank and non-comment) source cards is too low.

IV. RUN TIME ERROR MESSAGES

The form of run error messages is

RUN ERROR NEAR CARD (number) - (message)

SUBSTRING INDEXING	substring selected not within named string.
CASE SELECTION INDEXING	index of case statement or case expression is less than 1 or greater than number of cases.
ARRAY SUBSCRIPTING	array subscript not within declared bounds.

LOWER BOUND> UPPERBOUND — lower bound is greater than upper bound in array declaration.

ARRAY TOO LARGE — The (n-1) dimensional array obtained by deleting the right-most bound-pair of the array being declared has too many elements The maximum number of elements allowed in this (n-1) dimensional array is given below, according to the declared type of the array.

type	maximum # of elements in first (n-1) dimensions
logical, string	32767
integer, real	8191
bits, reference	8191
long real, complex	4095
long complex	2047

ASSIGNMENT TO NAME PARAMETER — assignment to a formal name parameter whose corresponding actual parameter is an expression, a literal, control identifier., or procedure name.

DATA AREA OVERFLOW — storage available for program execution has been exceeded.

ACTUAL-FORMAL PARAMETER MISMATCH IN FORMAL PROCEDURE CALL — the number of actual parameters in a formal procedure call is different from the number of formal parameters in the called procedure, or the parameters are not assignment compatible.

RECORD STORAGE AREA OVERFLOW — no more storage exists for records.

LENGTH OF STRING INPUT	string read is not assignment compatible with corresponding declared string.
LOGICAL INPUT	quantity corresponding to logical quantity is not true or false.
'NUMERICAL INPUT	numerical input not assignment compatible with specified quantity.
REFERENCE INPUT	reference quantities cannot be read.
READER EOF	a system control card has been encountered during a read request.
REFERENCE	the null reference has been used to address a record, or a reference bound to two or more record classes was used to address a record class to which it was not currently pointing.
LINE ESTIMATE EXCEEDED	line estimate on %ALGOL card is exceeded.
TIME ESTIMATE EXCEEDED	time estimate on %ALGOL card is exceeded.
I/O ERROR	see consultant.
PROGRAM CHECK #nn	see consultant.

Counts of certain exceptional conditions detected during program compilation or execution are maintained.' If any of these are non-zero, they are listed after the post-compilation or post-execution elapsed time message in the following format:

nnnn PROGRAM CHECK NO xx

The number of times the condition was detected (modulo 10000) is given by nnnn; the nature of the condition is indicated by xx according to the following table:

08	integer overflow
09	integer division by zero
12	real exponent overflow
13	real exponent underflow
15	real division by zero

This counting is inhibited for integer overflow and exponent underflow whenever the value of the corresponding reference variable is null (cf. LANGUAGE DESCRIPTION, Section 8.5.).

v. OTHER

PRG PSW	see consultant.
COMPILER ERROR	see consultant.
INSUFFICIENT STORAGE	insufficient memory available to complete compilation.

NOTES ON NUMBER REPRESENTATION

ON SYSTEM/360

AND RELATIONS TO ALGOL W

by

George E. Forsythe

The following notes are intended to give the student of Computer Science 136 some orientation into how numbers are represented in the IBM System/360 computers. Because we are using Algol W, some references are made to that language. However, very little of what is said here depends on the peculiarities of Algol W, and this exposition is mostly applicable to Fortran or Algol 60 with slight changes in wording. It will also do for the floating-point numbers and full-word integers of PL/1. Users of shorter or longer integers or decimal arithmetic in PL/1 will need more orientation.

On IBM's system 360, the following units of information storage are used:

a) the bit., a single 0 or 1

b) the byte, a group of eight consecutive bits

c) the (short) word, a group of four consecutive bytes--i.e., 32 consecutive bits

d) the long word, a group of two consecutive short words--i.e., eight bytes or 64 bits.

For number representation in Algol W the words and long words are the main units of interest,

INTEGERS.

Integers are stored in (short) words, Of the 32 bits of a short word, one is reserved for the sign (0 for + and 1 for -), leaving 31 bits to represent the magnitude, A positive or zero integer is stored in a binary (base 2) representation, Thus 21_{10} (the subscript means base 10) is stored as

0000 0000 0000 0000 0000 0000 0001 0101 .

sign bit

To confirm this, note that

$21 = 0 \times 2^{30} + \ldots + 0 \times 2^5 + \underline{1} \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + \underline{1} \times 2^0.$

The largest integer that can be stored in a word is

$$2^{30} + 229 + \ldots + 210 + 2 = 2^{31} - 1 = (2147483647)_{10} .$$

Any attempt to create or store an integer larger than $2^{31} - 1$ will produce erroneous results, and (unfortunately) the user will not always be warned of the error (See below,)

To save space in writing words on paper, each group of four bits in a word is frequently converted to a single base-16 (hexadecimal) digit, according to the following code:

base 2	base 16	base 2	base 16
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Thus A, B, C, D, E, F are used as base-16 representations of the decimal numbers 10, 11, 12, 13, 14, 15 respectively. Nevertheless, integers are stored as base-2 numbers.

Using hexadecimal notation, the decimal number 21 is represented by

$$00000015_{16} .$$

Note that 15_{16} is the base-16 representation of 21_{10}.

Negative integers are stored in what is called the "two's complement form". For example, -1 is stored as

1111 1111 1111 1111 1111 1111 1111 1111 ,

$$= FFFFFFFF_{16} .$$

Also, -21 is stored as

1111 1111 1111 1111 1111 1111 1110 1011

$$= FFFFFFEB_{16} ,$$

The representation for -21 is obtained from that for +21 by changing every 0 to 1 and every 1 to 0, and then adding + 1 in base-2 arithmetic to the result. Similarly for any negative integers. Every negative integer has 1 as its sign bit. The smallest integer storable in System/360 is $-2^{31} = -2147483648$, and is represented by 80000000_{16}. Another way to think of the representation of negative numbers is to consider a 32-place binary accumulating register (the base-2 equivalent of the-decimal accumulating register in a desk calculating machine), If one starts with all zeros in this register, one gets the representation for -1 by subtracting 1. The process requires a "borrow" to propagate to the left all the way across the register, leaving all ones, just as on a decimal accumulator this would leave all nines. Continued subtraction will give the representations for -2, -3, ...

From the point of view of an accumulator we can also see what happens when we create a positive number larger than $2^{31}-1$. For example, if we add 1 to $2^{31}-1$, the resulting carry will go all the way into the sign bit, leaving a sign bit of 1 with all other digits zero. But this is the representation of -2^{31}. Thus the attempt to produce positive numbers in the range from 2^{31} to approximately 2^{32} will yield a negative sign bit. Consequently, positive integers that "overflow" into this range are sensed as negative by System/360. The mechanisms of ALGOL W for detecting integer overflow (not described in this document) can be used to detect additions, subtractions, or multiplications that produce integers outside the range from -2^{31} to $2^{31}-1$ (so-called integer overflow). Attempts to divide an integer by 0 will yield an error message and an irrelevant quotient and remainder.

The behavior of System/360 on integer overflow is quite different from the Burroughs B5500. In the latter machine, any integer that overflows is replaced by a rounded floating-point number. There are advantages to either approach to integer overflow, depending on the application.

If the user suspects that integers in his program are getting anywhere near 10^9, he should convert them to double-precision floating-point numbers by use of the Algol W operator LONG. Conversion to single-precision floating-point numbers may lose some precision.

The most important thing for a scientific user to remember is that integers in the range -2^{31} to $2^{31}-1$ are stored without any approximation. Moreover, operations on integers (adding, subtracting, multiplying) are done without any error, so long as all intermediate and final results are integers between -2^{31} and $2^{31}-1$. It is perhaps easier to remember as safe the interval from -2×10^9 to 2×10^9, obtained from the useful approximation $2^{10} \doteq 103$.

The operations of division without remainder (called DIV in Algol W) and taking the remainder on division (called REM in Algol W) always give integer answers, If the divisor is 0, an error message is given,

In Algol W two operations on integers give results that are not stored as integers--namely / and ** .

FLOATING-POINT NUMBERS

Numbers in many scientific computations will grow in magnitude well beyond the range of integers described above, To provide for this, System/360 and most scientific computers have a second way to represent numbers--the so-called floating-point representation, The significance of the name "floating-point" is that the radix point --for example, the decimal point in base-10 numbers--is permitted to float to the right or left, thus permitting scaling of numbers by various powers of the radix, Although a decimal point that has floated off to the left will produce a number written like 0.001345, the nuuibers are actually represented in a form closer to what is often called scientific notation, here 1.345×10^{-3} .

In System/360, floating-point numbers are always represented in base-16 notation; i.e., the radix or number base is 16. This permits us to write numbers in abbreviated form (as we did with integers earlier). More important, the use of base-16 conforms with the hardware arithmetic processes in which shifting is done four bits at a time to speed up the operations. The speed-up is achieved at a slight cost in precision, as is learned frcxn detailed error analyses which we cannot go into here,

We first consider the floating-point representation of numbers by a single word of 32 bits. This is the so-called single-precision or short real number, the number of type REAL in Algol W. The 32 bits of a word are numbered frcxn 0 to 31, from left to right, just to identify them, In floating-point representation the left-hand eight bits (bits 0 to 7, equivalent to two hexadecimal digits) are devoted to the sign of the number and the exponent of 16 associated with the number, The right-hand 24 bits (bits 8 to 31, equivalent to six hexadecimal digits)

represent six significant hexadecimal digits (the significand) of the number.

As with integers, the sign of the number is denoted by bit 0, with 0 representing + and 1 representing - .

Bits 1 to 7 give the binary (base-2) representation of a non-negative integer in the range 0_{10} to 127_{10}, inclusive, This Integer is called the biased exponent, for reasons now to be explained, If this integer were taken directly a8 the exponent, we would have no negative exponent s, and our range of floating-point number6 could not include such number6 as 16^{-25}. It is desirable to have an exponent range that is approximately symmetric about zero. In System/360 one obtain8 the true exponent of the floating-point number by subtracting 64 from the biased exponent represented by bits 1 to 7. A8 a result, the actual exponent6 range from -64 to 63.

The 24 bit6 8 to 31 of a number are regarded a6 six hexadecimal -digits with a hexadecimal point at the left-hand end. If the floating-point number zero is being represented, all the hexadecimal digit8 are zero, as are all the other bits, Otherwise, at least one of the hexadecimal digits must be nonzero. A floating-point number is said to be normalized if the left-hand hexadecimal digit (the most significant digit) of the significand is nonzero. In System/360 the floating-point numbers are ordinarily normalized, and we will not consider any other forms.

We now give the floating-point representations of saae sample numbers. As we said before, the number zero is represented by 32 zero bits, Le., by eight 0 hexadecimal digits. Thus zero is represented by the same trords in floating-point or integer form. No other number ha6 this property.

The number 1.0 is represented by the word

sign bit

0, 100 0001, 0001 0000 0000 0000 0000 0000,

biased exponent

significand

To check this, note that the sign is 0 (representing +). The biased exponent is 1000001_2 or 65_{10}. Subtracting 64_{10} yields 1 as the true exponent. The hexadecimal significand is 100000_{16}. Putting a hexadecimal point at the left end gives the hexadecimal fraction $.100000_{16}$, which equals 1/16. Thus the above word represents $+1/16$ times 16^1, or 1.0 .

To save writing, the above word is ordinarily written in the hexadecimal form 41100000 . While one gradually learns to recognize some floating-point numbers in this form, the author knows no easy way to convert such a hexadecimal word into a real number. One just has to take the right-hand six hexadecimal digits, and prefix a hexadecimal point. Then one examines the left-hand two-hexadecimal-digit number (here 41). If this is less than 80_{16}, the floating-point number is positive and one gets the true exponent by subtracting $40_{16} = 64_{10}$. If the left-hand two-hexadecimal-digit number is 80_{16} or larger, the floating-point number is negative, and one gets the true exponent by subtracting $C0_{16} = 80_{16} + 40_{16} = 192_{10}$ and affixing a minus sign. Some facility with hexadecimal arithmetic is required, if one has to deal with such numbers.

In this presentation, we have considered the radical point to be at the left of the six significant hexadecimal digits, and regarded the exponent as biased high by 64_{10}. As an alternative, the reader may prefer to place the radix point just to the right of the most significant digit of the significand, and regard the exponent as biased high by -65_{10}. This brings the significand closer to usual scientific notation but, of course, requires a trickier conversion to get the true exponent, The fact that either interpretation (and many others) are possible shows that really the radical. point is just in the eye of the beholder, and not in the computer!

Several examples of floating-point numbers are now given in hexadecimal notation, with the confirmation left to the reader.

decimal		floating-point
0.0	=	00000000
1.0	=	41100000
0.0625	=	40100000
16.0	=	42100000
256.0	=	43100000
-1.0	=	C1100000
-16.0	=	C2100000
3.5	=	41380000

The largest floating-point number is 7FFFFFFF, representing .FFFFFF x 16^{3F} or $(1 - 16^{-6}) \times 16^{63} \doteq 7.23 \times 10^{75}$. (Here 10 and 16 denote decimal numbers.)

The smallest positive normalized floating-point number is 00100000, representing

$$\frac{1}{16} \times 16^{-64} \doteq 5.40 \times 10^{-79}$$

Negatives of these two numbers can also be represented, and are the extremes in magnitude of representable negative numbers.

Very few numbers can be exactly represented with six significant decimal digits. (Exercise: Which ones can?) For example, $1/3 = .333333_{10}$ only approximately. In the same way, very few numbers can be exactly represented with six significant hexadecimal digits, (Exercise: Which ones can?) For example, $1/3 = .555555_{16}$ only approximately. Moreover, some numbers that are exactly representable in decimal are only approximately representable in hexadecimal; for example,

$1/10 = .100000_{10}$ exactly; but

$1/10 = .19999A_{16}$ only approximately.,

Thus round-off error enters into the representation of most floating-point numbers on System/360, and the round off differs from that with decimal numbers,, This can easily give rise to unexpected results. For example, if the above number $.19999A_{16}$ ($\doteq 0.1_{10}$) is multiplied by the integer $100_{10} = 64_{16}$, one gets not $A.00000_{16} = 10.0_{10}$, but instead $A.00003_{16}$, as a cumulative effect of the slightly high approximation to 0.1_{10}. And $A.00003_{16}$ rounds to 10.00002_{10} on conversion to decimal.

The precision of a single-precision hexadecimal number is roughly 10^{-7}. One can think of this as being crudely equivalent to seven sig-

niflcant decimal digits,

Not only do errors appear in the representation of numbers inside System/360 (or any computer), but they arise from arithmetic operations perf ormed on numbers, For example, the product of two floating-point numbers may have up to 12 significant hexadecimal digits. When the product is stored as a single-precision floating-point number, it must be rounded to six hexadecimal digits, This introduces an error, even though the factors might have been exact,

The story of round off and its effect on arithmetic is a complex. and Anteresting one, Only within the current decade have there begun to appear even partly satisfactory methods to analyze round off, and we cannot go into the matter now, Some idea of this is obtained in Computer Science 137.

When an Algol W program assigns decimal numbers or integer values to variables of type REAL, these are immediately converted to hexadecimal floating-point numbers, with (usually) a round-off error. When one outputs numbers from the computer in Algol W, they are converted to decimal. Both conversions are done as well as possible, but introduce changes in the numbers that the programmer must be aware of, And, of course, all intermediate operations introduce further round offs and possible errors. It is unthinkable to do the anaiysis necessary to counteract these errors and get the true answer to the problem, If the user wishes answers uncontaminated by round off, he should use integers and integer arithmetic, and be prepared to guard against overflow,

Fortunately most users can accept an indeterminate amount of round off in their numbers, provided they have some assurance that round off is not growing out of control, It is the business of numerical analysts to provide algorithms whose round-off properties are reasonably under control. This has been well accomplished in some areas, and hardly at all in others.

DOUBLE PRECISION

The precision of single-precision floating-point numbers seems

very adequate for most scientific and engineering purposes, being at the level of seven decimals, However, a considerable number of computations require still more precision in the middle sanewhere, just in order to cane out with ordinary accuracy at the end. As a result, System/360 has provided an easy mechanism for getting a great deal more precision in the computations. For this purpose a double word of 64 bits is used to store a floating-point number of so-called double precision or long precision. In this representation, the sign and biased exponent are found in the first word of the double-word, with precisely the same interpretation as with single-precision floating-point numbers, The second word of the double-word consists of eight hexadecimal digits immediately following the six found in the first word., There is no sign or exponent in the second word. Thus a double-word represents a signed floating hexadecimal. number with 14 significant hexadecimal digits, As before, nonzero numbers are normalized so that the most significant digit of the 14 is nonzero.

Examples:

		long significand	
1.0L	= 41'	100000	00000000
0.1L	= 40	199999	9999999A

There is a full set of arithmetic operations for both single and double-precision operations. Very crudely, for an example, single-precision multiplication of single-precision factors takes around 4 micro-seconds, while that for double-precision factors takes around 7 micro-seconds. For modest problems the extra time is completely lost in the several. seconds of time lost to systems and compilers, and the use of double-precision is strongly recommended for all scientific computation. Normally the only possible disadvantage of using long precision is the doubling in the amount of storage needed. If one has arrays with tens of thousands of elements, the extra storage may be very costly. Otherwise, it should not matter,

Since $16^{-14} \doteq 10^{-17}$, the double-precision numbers are crudely equivalent in precision to 17 significant decimal digits.

For a machine with the speed of the 360/67, a number precision of

six hexadecimal digits (roughly seven decimals) is considered 'very low, while a precision of 14 hexadecimal digits (roughly 17 decimals) is very adequate .

The floating-point arithmetic hardware of System/360 provides the possibility of detecting when numbers have gone outside the exponent range stated above. The reader may think that a range from roughly 10^{-79} to 10^{75} should cover all reasonable computations. While exponent overflow and exponent underflow are not very common, they can be the cause of very elusive errors, The evaluation of a determinant is a common computation, and for a matrix of order 40 is quite rapidly done (if you know how). If the matrix elements are of the quite reasonable magnitude 10^{-3}, the magnitude of the determinant will be no larger than roughly 10^{-90} (and probably much smaller), well below the range of representable floating-point numbers, Such problems are a frequent source of exponent underflow.

We shall not discuss here the mechanisms of Algol W for detecting exponent overflow and underflow, for these should be written up in another place, Even without these, we see that floating-point numbers behave well for numbers that are at least 10^{66} times as large as the largest integer in the system: Hence use of floating-point numbers meets almost all the problems raised by integer overflow. And, of course, it permits the use of a large set of rational numbers, which do not even enter the integer system,

ALGOL W REALS AND LCINGREALS

The Algol W manual tells how to represent real variables and numbers to take advantage of both single-and double-precision, The purpose of this section is to bring this information into rapport with the hardware representation of numbers. If a variable X is declared REAL, one word is set aside for its values, and it will be stored in single-precision floating-point form, If a variable is declared to be LONG REAL, a double-word is set aside to hold its values, and it will be stored in double-precision form,,

If a number is written in one of the decimal forms without an L at the end, it will be chopped to single-precision, no matter how many digits are set down. Thus 3.1415926535891932 will be immediately chopped to single-precision in the program, and all the superfluous digits are lost at once. Thus the assignment statement

```
XX := 3.1415926535897932
```

will result in the double-word XX receiving an approximation to π in the more significant half, and all zeros in the less significant half! Thus one gets a precision of only approximately seven decimals for the pain of writing 17, and this may well contaminate all the rest of the computation.

If one wants XX to be precise to approximately full double precision, one must write the statement in the form

```
XX := 3.1415926535897932L .
```

With the declaration REAL X, the statement

```
X := 3.1415926535897932L
```

will result in X having a single-precision approximation to π, as the long representation of π is chopped upon assignment to X.

The reader should now go back and examine the specifications of the types of various arithmetic expressions, as stated on pages 9, 10, 11 of the Algol W Notes, and on pp. 25, 26 of the Language Definition. Some of the less expected effects are the following: Suppose we have declarations

```
REAL x, Y, z;
LONG REAL XX, YY, ZZ;
INTEGER I, J, K;
```

Then X*Y, I**J, and I*X are all-long real.

The assignment statement

```
xx := x := Y*Z
```

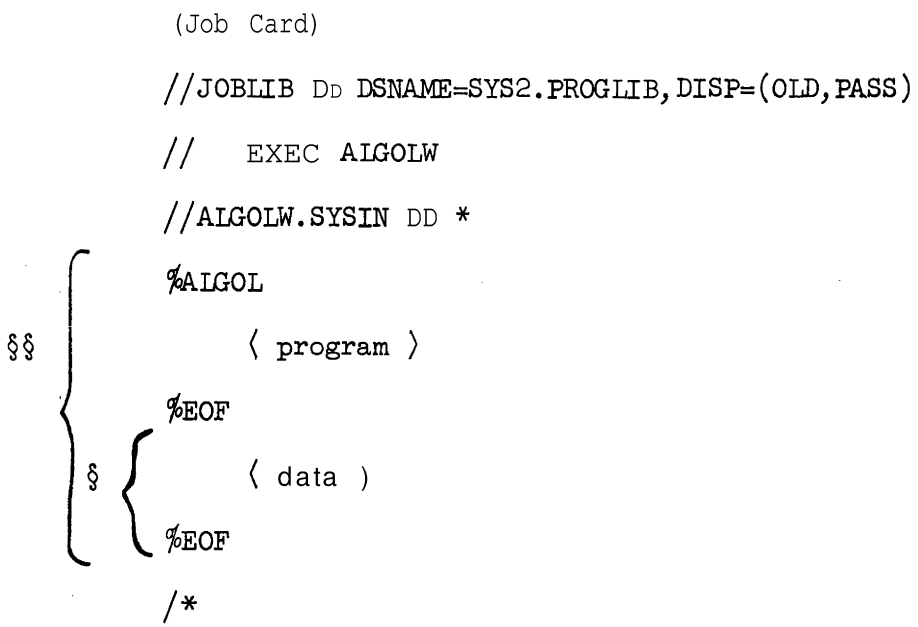
will result in XX having a single-precision chopped version of Y*Z in the more significant half, and zeros in the less significant word.

Moreover, I*I is INTEGER, but I**2 is LONG REAL.

If the reader understands the language Algol W and the preceding pages on number representation, he should have a good basis for understanding the effects of mathematical algorithms. But he should always remain wary of what a computer is actually doing to his numbers!

APPENDIX

Algol W Deck Set-Up



§ Optional

§§ May be repeated

Note: The Stanford ALGOL W system monitors execution time and number of lines of output for each job. The default limits on these quantities are 10 seconds execution time and 500 lines of printed output. Alternately, the programmer may explicitly specify limits on the %ALGOL card. Columns 10-29 of that card are scanned for such specification according to the following syntax:

```
(limit specification)     ::= (time limit) | (time limit), (line limit)
(time limit)              ::= (minutes specification) |
                              (minutes specification) : (seconds specification)
(minutes specification)   ::= (unsigned integer) | (empty)
(seconds specification)   ::= (unsigned integer) | (empty)
(line limit)              ::= (unsigned integer) | (empty)
```

An empty field is given the corresponding default value. The program is automatically terminated if necessary at the end of the indicated time. Similarly, the program is automatically terminated if necessary after the indicated number of lines have been printed.

GRAMMATICAL DESCRIPTION OF ALGOL W

by

R. Floyd

In the grammatical description of ALGOL W on the following pages, Roman capital letters, such as A B C D, stand for themselves. A script letter, possibly accented, stands for a defined infinite class of symbol strings; for example, $\mathcal{I}$, as defined, stands for the class which includes the symbols A, B, C, ... Z, AA, AB, . . .,A9, BA,...,B9,...Z9, AAA, ... Z99, AAAA, ... A Greek letter, such as λ , stands for a given finite set of characters.

The symbol | means "or"; if $\mathcal{A}$ is defined as $\mathcal{B}|C$, this means that a particular inscription is an $\mathcal{A}$ if it is a $\mathcal{B}$ or if it is a C .

The notation $\mathcal{A}^*$, or equivalentiy $\{\mathcal{A}\}^*$, means any number (including zero) of inscriptions, one after another, each of which is an $\mathcal{A}$. For example, $\{A|B\}^*$ means A or B or AA or AB or BA or BB or AAA or . . . . or Λ , where Λ means no inscription at all.

The notation $\mathcal{A}^+$ means any number (but at least one) of inscriptions, one after another, each of which is an $\mathcal{A}$. It abbreviates $\mathcal{A}\mathcal{A}^*$. For example, $\{A|B\}^+$ means A or B or AA or . . . or BB or AAA , etc.

The notation $[\mathcal{A}]$ means an optional occurrence of $\mathcal{A}$; it abbreviates $\{\mathcal{A}|\Lambda\}$.

Thenotation $\overline{\mathcal{A}+\mathcal{B}}$ means $\mathcal{A}$ or $\mathcal{A}\mathcal{B}\mathcal{A}$ or $\mathcal{A}\mathcal{B}\mathcal{A}\mathcal{B}\mathcal{A}$, etc; it abbreviates $\mathcal{A}\{\mathcal{B}\mathcal{A}\}^*$.

The notation $\mathcal{A} \perp \mathcal{B}$ means $\mathcal{A}$ and/or $\mathcal{B}$; it abbreviates $\mathcal{A}|\mathcal{B}|\mathcal{A}\mathcal{B}$.

The curly brackets { } are used simply as parentheses to show the scope of the above operators.

All other characters, such as / - , () / < etc., stand for themselves, including * and + when they are not raised.

The Grammar of a Simple Subset of **ALGOL** W

Descriptive Name	Symbol	Definition
letter	λ	A|B|C|D|E|...|X|Y|Z
digit	δ	0|1|2|3|...|8|9
identifier	$\mathcal{I}$	$\lambda\ \{\lambda\|\delta\}^*$
symbol	σ	Any symbol on the keypunch, except the double quote
constant	C	$\delta^+[\ .\ \delta^*]\ \|\ "\sigma^+"$
function value	$\mathcal{F}$	$4\ (\overline{\mathcal{E}^+,})]$
expression	$\mathcal{E}$	$C\text{-}1\ \overline{\{\mathcal{I}\|C\|\mathcal{F}\|(\mathcal{E})\}^{**}\{*\|/\}}^{+}\ \{+\|-\}\ \{<\|<=\|=\|>=\|>\|\neg=\}$
simple statement	S′	$\mathcal{I}:=\mathcal{E}\|\mathcal{I}[(\overline{\mathcal{E}^+,})]\|$GO TO $\mathcal{I}\ \|\mathcal{B}$
statement	S	S′|IF $\mathcal{E}$ THEN S|IF $\mathcal{E}$ THEN S′ ELSE S|FOR $\mathcal{I}:=\mathcal{E}$ UNTIL $\mathcal{E}$ DO S
block	$\mathcal{B}$	BEGIN $\{\mathcal{D};\}^*\{S;\|\mathcal{I}:\}^*$S END
declaration	$\mathcal{D}$	$\mathcal{T}\ \overline{\mathcal{I}^+,}\|\mathcal{T}$ PROCEDURE $\mathcal{N};\{\mathcal{E}\|$BEGIN$\{\mathcal{D};\}^*\{S;\|\mathcal{I}:\}^*\mathcal{E}$ END]
type	$\mathcal{T}$	INTEGER|REAL|LOGICAL|STRING(C)
procedure heading	$\mathcal{N}$	$\mathcal{I}(\mathcal{T}\{$VALUE|PROCEDURE$\}\overline{\mathcal{I}^+,};)$
program	$\mathcal{P}$	$\mathcal{B}.$

Descriptive Name	Symbol	Definition
letter	λ	A|B|C|D|E|$\cdots$|X|Y|Z
digit	δ	0|1|2|3|$\cdots$|8|9
identifier	ϑ	$\lambda\{\lambda\|\delta\}^*$
variable	v	$\{\vartheta\|\vartheta(\varepsilon)\|\vartheta(\overline{\varepsilon,}^+)\}[(\varepsilon$ C$)]$
symbol	σ	Any character on the keypunch, except the double quote.
constant	C	$\{\{\delta^+[\ .\delta^*]\ \|\ .\delta^+\}\ \curlywedge\ \{'[+\|-]\ \delta^+\}\}$[I][L] |TRUE|FALSE |# $\{\delta$ |A|B|C|D|E|F$\}^+$|"$\{\sigma$|""$\}^+$"|NULL
function value	$\mathcal{F}$	$\vartheta[(\overline{a^+,})]$
simple expression	ε''	$[+\|-][\neg]\{$ABS|LONG|SHORT$\}^*\{v\|$C$\|\mathcal{F}\|(\varepsilon)\}$ $\{**$|SHL|SHR$\}\{*\|/$|DIV|REM|AND$\}\{+\|-$|OR$\}$
simple expression or relation	ε'	$\varepsilon''\ \|\varepsilon''\{<\|<=\|=\|>=\|>\|\neg=\}\varepsilon''\ \|\varepsilon''$ IS ϑ
expression	ε	ε'|IF ε THEN ε ELSE ε|CASE ε OF $(\overline{\varepsilon^+,})$
argument	a	ε|S|$\vartheta[(\{\overline{\varepsilon\|*\}^+,})$ I
simple statement	S'	$\{v:=\}^+\varepsilon$| GO TO ϑ|$\vartheta[(\overline{a^+,})]$|$\Lambda$|$\mathcal{B}$
empty	Λ	The empty statement; no character at all, **or** a space.
statement	s	S'|IF ε THEN **S**|IF ε THEN S' ELSE **S**|CASE ε OF **BEGIN** $\overline{S^+;}$ END |WHILE ε DO S|FOR $\vartheta:=\varepsilon$ $\{$[STEP ε] UNTIL $\varepsilon\|\{,\varepsilon\}^*\}$DO s

Descriptive Name	Symbol	Definition
block	$\mathcal{B}$	BEGIN $\{\mathcal{D};\}^*$ $\{S;|\mathcal{S}:\}^*S$ END
declaration	$\mathcal{D}$	$\mathcal{T}\ \overline{\mathcal{I}^+,}|\mathcal{T}$ ARRAY $\mathcal{I}^+,(\overline{\mathcal{E}::\mathcal{E}^+,})|$PROCEDURE $\mathcal{N}$; s $|\mathcal{T}$ PROCEDURE $\mathcal{N}$;$\{\mathcal{E}|$BEGIN $\{\mathcal{D};\}^*\{S;|\mathcal{S}:\}^*\mathcal{E}$ END$\}|$RECORD $\overline{\mathcal{S}(\mathcal{T}\ \overline{\mathcal{I}^+,}\ ;)}$
type	$\mathcal{T}$	INTEGER$|$[LONG]$\{$REAL$|$COMPLEX$\}|$LOGICAL$|$BITS[(32)]$|$STRING[(C)] $|$REFERENCE$(\overline{\mathcal{I}^+,})$
procedure heading	$\mathcal{N}$	$\mathcal{S}[(\{\mathcal{T}$ [VALUE][RESULT]$|$[$\mathcal{T}$] PROCEDURE$\}\overline{\mathcal{I}^+,}|\mathcal{T}$ ARRAY $\overline{\mathcal{I}^+,(*^+,)};)]$
program	$\mathcal{P}$	$\mathcal{B}$

The Operators and Functions of ALGOL W, Their Formats, Meanings and Type Constraints

Use of Symbols

$\mathcal{E}_i$ = any ALGOL W expression.

α_1 = value of expression $\mathcal{E}_1$.

k_i = kind of data represented by α_i corresponding to expression $\mathcal{E}_1$.

The kinds of data are:

1. N = numeric
2. L = logical
3. S = string
4. B = bits
5. R= reference

d_i = domain of α_i when k_i = N.

The domains are:

1. I = integer
2. R = real
3. C = complex

They are ordered as follows: $I \subset R \subset C$.

p_i = precision of α_i when k_i = N.

They are ordered as follows: S < L.

If d_i = I, then p_i = L.

Format	Meaning	Kinds of Arguments and Results	Domains of Numeric Arguments and Results	Precision of Numeric Arguments and Results
$\mathcal{E}_1 + \mathcal{E}_2$	$\alpha_1 + \alpha_2$	N+ N → N	$d_1+d_2 \rightarrow \max(d_1,d_2)$	$p_1+p_2 \rightarrow \min(p_1,p_2)$
$\mathcal{E}_1 - \mathcal{E}_2$	$\alpha_1 - \alpha_2$	N - N → N	$d_1-d_2 \rightarrow \max(d_1,d_2)$	$p_1-p_2 \rightarrow \min(p_1,p_2)$
$\mathcal{E}_1 * \mathcal{E}_2$	$\alpha_1 \times \alpha_2$	N * N → M	$d_1*d_2 \rightarrow \max(d_1,d_2)$	$p_1*p_2 \rightarrow L$
$\mathcal{E}_1 / \mathcal{E}_2$	α_1 / α_2	N/N → N	$d_1/d_2 \rightarrow \max(d_1,d_2, R)$	$p_1/p_2 \rightarrow \min(p_1,p_2)$
$\mathcal{E}_1 ** \mathcal{E}_2$	$\alpha_1^{\alpha_2}$	N**N → N	$d_1**I \rightarrow \max(d_1,R)$	$p_1**L \rightarrow p_1$
$+ \mathcal{E}_1$	α_1	+N → N	$+d_1 \rightarrow d_1$	$+p_1 \rightarrow p_1$
$- \mathcal{E}_1$	$-\alpha_1$	-N → N	$-d_1 \rightarrow d_1$	$-p_1 \rightarrow p_1$
$\mathcal{E}_1$ DIV $\mathcal{E}_2$	TRUNCATE(α_1/α_2)	I DIV I → I		
$\mathcal{E}_1$ REM $\mathcal{E}_2$	$\alpha_1 - (\alpha_1$ DIV $\alpha_2) * \alpha_2$, the remainder of $\mathcal{E}_1$ DIV $\mathcal{E}_2$	I REM I → I		
ABS $\mathcal{E}_1$	$\lvert\alpha_1\rvert$	ABS N → N	ABS $d_1 \rightarrow \min(d_1,R)$	ABS $p_1 \rightarrow p_1$
LONG El	α_1	LONG N → N	LONG $d_1 \rightarrow \max(d_1,R)$	LONG pl → L where p_1=s or d_1=I
SHORT $\mathcal{E}_1$	α_1	SHORT N → N	SHORT $d_1 \rightarrow d_1$	SHORT $p_1 \rightarrow$ S where p_1=L and $d_1 \neq$ I

Format	Meaning	Kinds of Arguments and Results	Domains of Numeric Arguments and Results	Precision of Numeric Arguments and Results
$\mathcal{E}_1$ OR $\mathcal{E}_2$	$\alpha_1 \vee \alpha_2$	L OR L → L B OR B → B		
$\mathcal{E}_1$ AND $\mathcal{E}_2$	$\alpha_1 \wedge \alpha_2$	L AND L → L B AND B → B		
$\neg\ \mathcal{E}_1$	NOT α_1	$\neg$ L → L $\neg$ B → B		
$\mathcal{E}_1 = \mathcal{E}_2$	$\alpha_1 = \alpha_2$	$k_1 = k_2$ → L(where k_1=k_2)	any	any
$\mathcal{E}_1 \neg= \mathcal{E}_2$	$\alpha_1 \neq \alpha_2$	$k_1 \neg= k_2$ → L(where k_1=k_2)	any	any
$\mathcal{E}_1 < \mathcal{E}_2$	$\alpha_1 < \alpha_2$	N < N → L S < S → L	$d_1, d_2 \subseteq$ R	any
$\mathcal{E}_1$ <= $\mathcal{E}_2$	$\alpha_1 \leq \alpha_2$	N <= N → L S <= S → L	$d_1, d_2 \subseteq$ R	any
$\mathcal{E}_1$ >= $\mathcal{E}_2$	$\alpha_1 \geq \alpha_2$	N >= N → L N >= S → L	$d_1, d_2 \subseteq$ R	any
$\mathcal{E}_1 > \mathcal{E}_2$	$\alpha_1 > \alpha_2$	N > N → L S > S → L	$d_1, d_2 \subseteq$ R	any
$\mathcal{E}_1$ IS ϑ_2	α_1 belongs to the record class ϑ_2	R IS ϑ_2 → L		
$\mathcal{E}_1$ SHL $\mathcal{E}_2$	α_1 shifted left α_2 places	B SHL N → B	d_2 = I	
$\mathcal{E}_1$ SHR $\mathcal{E}_2$	α_1 shifted right α_2 places	B SHR N → B	d_2 = I	
$\mathcal{V}_1(\mathcal{E}_2 \mid \mathcal{E}_3)$	characters α_2 through $\alpha_2 + \alpha_3 - 1$ of α_1	S(N|N) → S	$d_2 = d_3$ = I	

Format	Meaning	Kinds of Arguments and Results	Domains of **Numeric** Arguments and Results .	**Precision of** Numeric **Arguments and** Results
IF $\mathcal{E}_1$ THEN $\mathcal{E}_2$ ELSE $\mathcal{E}_3$	if α_1 then α_2, otherwise α_3	IF L THEN k_2 ELSE $k_3 \rightarrow k$ where $k_2 = k_3 = k$	IFLTHEN d_1 **ELSE** d_2 $\rightarrow$ **max**(d_1, d_2)	**IF L THEN** p_1 ELSE p_2 $\rightarrow$ **min**(p_1, p_2)
CASE $\mathcal{E}_o$ of $(\mathcal{E}_1, \ldots, \mathcal{E}_n)$	α_{α_0} $(1 \leq \alpha_o \leq n)$	CASE N OF $(k_1, k_2, \ldots, k_n)$ $\rightarrow$ k where $k_1 = k_2 = \ldots = k_n = k$	CASE L **OF** $(d_1, d_2, \ldots, d_n)$ $\rightarrow$ **max**($d_1, d_2, \ldots, d_n$)	**CASE L OF** $(p_1, \ldots, p_n)$ $\rightarrow$ **min**($p_1, \ldots, p_n$)

All the following functions have the format $F(e_1)$, where F is the function name.

We shall omit reference to the format, accordingly.

Function	Meaning		Kinds	Domains	Precision
TRUNCATE	The integer i, with the same sign as α_1, such that $\lvert\alpha_1\rvert - 1 < \lvert i\rvert \leq \lvert\alpha_1\rvert$				
ENTIER	The integer i such that $\alpha_1 - 1 < i \leq \alpha_1$	I	N → N	R → I	Any
ROUND	The integer i, with the same sign α_1, such that $\lvert\alpha_1\rvert - 1/2 < \lvert i\rvert \leq \lvert\alpha_1\rvert + 1/2$				
ROUNDTOREAL	α_1		N → N	R → R	L → S
REALPART	The real part of α_1		N → N	C → R	Any → S*
IMAGPART	The imaginary part of α_1				
IMAG	$\alpha_1 * \sqrt{-1}$		N → N	$d_1 \rightarrow C$ ($d_1 \subseteq R$)	Any → S*

*Note : A asterisk on a short precision-result means that prefixing the letters LONG to the function name yields a long precision result.

Function	Meaning	Kinds	Domains	Precision
SQRT	$\sqrt{\alpha_1}$, for $\alpha_1 \geq 0$	N → N	$d_1 \rightarrow R$ ($d_1 \subseteq R$)	**Any → s***
COMPLEXSQRT	$\sqrt{\alpha_1}$	N → N	Any → C	Any → S*
EXP	e^{α_1}, for $\alpha_1 < 174.67$	N → N	$d_1 \rightarrow R$ ($d_1 \subseteq R$)	Any → S*
LN	$\log_e(\alpha_1)$, for $\alpha_1 > 0$			
LOG	$\log_{10}(\alpha_1)$ for $\alpha_1 > 0$			
SIN	$\sin(\alpha_1)$, for $\lvert\alpha_1\rvert < 823550$			
COS	$\cos(\alpha_1)$, for $\lvert\alpha_1\rvert < 823550$			
ARCTAN	$\tan^{-1}(\alpha_1)$, in the range $(-\pi/2, \pi/2)$			
TIME	elapsed time, in units of 1/100 minute if $\alpha_1 = 0$, otherwise in units of 1/60 second.	I → I		
ODD	α_1 is an odd number	I → L		
BITSTRING	The sequence of bits which represents α_1 in binary. See manuals for details.	I → B		

Function	Meaning	Kinds	Domains	Precision
NUMBER	The integer which α_1 represents in binary.	B → I		
DECODE	The number which is used as a code for the character α_1.	s(1) 3 I		
CODE	The character for which α_1 is used as a code.	I → s(1)		
BASE10	A string of the form b±12±1234567 representing α_1 as a power of ten times a. fraction. (b represents a blank space).	N → S(12)	$d_1 \subseteq R$	Any
LONGBASE10	As above, for b±12±123456789012345	N → S(20)	$d_1 \subseteq R$	Any
BASE16	A string of the form bb±12±123456 representing α_1 as a power of sixteen times a fraction, both in hexadecimal.	N → S(12)	$d_1 \subseteq R$	Any
LONGBASE16	As above, for bb±12±12345678901234	N → S(20)	$d_1 \subseteq R$	Any
INTBASE10	A string of the form b±1234567890 representing α_1 in decimal.	I → s(12)		
INTBAsE16	A string of the form bbbb12345678 representing α_1 in hexadecimal, using two's complement notation.	I → s(12)		