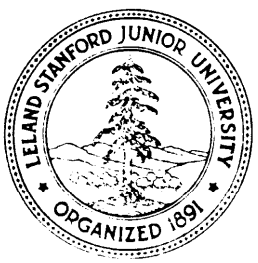
ALGOL. W
REFERENCE MANUAL

BY

RICHARD L. SITES

STAN-CS-71-230
FEBRUARY, 1972

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



ALGOL W
REFERENCE MANUAL

RICHARD L. SITES

FEBRUARY, 1972

This manual refers to the version of the Algol W compiler dated 16 January 1972.

"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.

HISTORICAL NOTE

This document is a major revision of and supersedes CS 110. The revisions were made in order to document a significantly improved version of the ALGOL W compiler. This version was known as X ALGOL W during the spring and summer of 1971. In addition to new debugging facilities documented under Compiler Options, the new version of the canpiler has slightly more meaningful error messages documented in the completely re-written Error Messages section. Various minor corrections and changes have been made throughout the book, and some examples have been added. There is now an index, and a complete list of all words the compiler treats in any special way. 'Below is a quick summary of the changes in the ALGOL W language:

1. Reserved words:

   There are three new reserved words: algol, assert, and fortran.

2. New statements and functions:

   There is now an ASSERT statement (cf. Section 7.5a).

   Procedures can be declared with empty bodies that instead specify that a linkage to an externally-compiled algol or fortran procedure is needed (cf. Section 5.3). A new standard function, TRACE, is added as part of the debugging facility (cf. Section 7.8.6).

---

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

3. Conversions:

   Conversions from integer to real now go to long real.

4. String comparisons:

   In comparing strings of different lengths, the shorter is extended with blanks before the comparison is done.

5. String assignments:

   String assignments are done in a single action, instead of character-by-character left-to-right. This prevents erroneous answers when assigning a string to a substring of itself.

6. Deleted facility:

   The standard functions COMPLEXSQRT and LONGCOMPLEXSQRT are no longer in the ALGOL W library. (cf. Deck Setup and Compiler Options, Section 3, for use of the Fortran library.)

The present author wishes to thank all those who have gone before him, especially Ed Satterthwaite for his extraordinary care in building the debugging facilities.

Table of Contents

LANGUAGE DESCRIPTION

ALGOL W

LANGUAGE DESCRIPTION

by

Henry Bauer

Sheldon Becker

Susan L. Graham

Edwin Satterthwaite

Richard L. Sites

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 P; or (ii) x = uw, y = uvw and v is a "comment" (see below).

2. A sequence x is said to produce a sequence y if and only if there exists an ordered set of sequences s[0], s[1], . . . , s[n], so that x = s[0], s[n] = y, and s[i-1] directly produces s[i] for all i = 1, . . . , n.

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

The sets V and U are defined through enumeration of their members in Section2 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 ref'er to the corresponding syntactic definition. Along with these letter sequences the symbol T 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 T 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 primary>
<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 | + | - | * I / | ** | div | rem | shr | shl | is |
abs | long | short | and | or | ¬ | | = | ¬ = | < |
< = | > | > = | :: |
:= | got0 | go to | for | step | until | do | while |
comment | value | result | assert | algol | fortran

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> <letter3 | <identifier? <digit> | <identifier> _

<T variable identifier> ::= <identifier>

<T array identifier+ ::= <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 not yield a value, in which case it is called a proper procedure. The value of a function procedure is defined as the value which results from the execution of the procedure body (cf. 6.2.2.).

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> ::= + | -

(N0te: a long complex constant may have the I and L in either order in a program, but they must be in the order IL on data cards.)

4.1.2. Semantics

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

4.1.3. Examples

1	.5	11
0100	1'3	0.671
3.1416	6.02486'+23	1IL
2.71828182845904523536028 7L		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>"

4.4.2. Semantics

Strings consist of any sequence of (at most 256) characters accepted by the System 360 enclosed by ", the string quote. If the string quote appears in the sequence of characters it must be immediately followed by a second string quote which is then ignored. The number of characters in a string is said to bc the length of the string.

4.4.3. Examples

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

4.5. References

43.1. 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> iidentifier 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> |
                            <ℑ function procedure declaration>
<proper procedure declaration> ::= procedure <procedure heading>;
                                   <proper procedure body>
<ℑ function procedure declaration> ::= <simple type> procedure
                                       <procedure heading>;
                                       <ℑ function procedure body>
<proper procedure body> ::= <statement> | <external procedure>
<ℑ function procedure body> ::= <ℑ expression> | <block body>
                                <ℑ expression> end | <external procedure>
<procedure heading> ::= <identifier> | <identifier> (<formal
                        parameter list>)
<formal parameter list> ::= <formal parameter segment> |
                            <formal parameter list> ; <formal
                            parameter segment>
<formal parameter segment> ::= <formal type> <identifier list> |
                               <formal array parameter>
<formal type> ::= <simple type> | <simple type> value | <simple
                  type> result | <simple type> value result |
                  <simple type> procedure | procedure
<formal array parameter> ::= <simple type> array <identifier
                             list> (<dimension specification>)
<dimension specification> ::= * | <dimension specification>, *
<external procedure> ::= fortran <string> | algol <string>
```

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.

5.3.2.4. External procedures. The body of a procedure can be just the construct

fortran <string>

or the construct

algol <string> .

In these cases, the actual body of the procedure is specified in a program that is compiled separately (externally). The <string> is a one-to-eight character external name that is used in the separate compilation. Thus, the example on page 27 could be used to refer to a FORTRAN program that begins:

```
SUBROUTINE PLOTSB(N) ...
```

(cf. Deck Setup and Compiler Options, Section 3 for details).

5.3.3. Examples

```
procedure INCREMENT; X := X+1
real procedure MAX (real value X, Y);
     if X < Y then Y else X
procedure COPY (real array U, V (*,*); integer value A, B);
     for I := 1 until A do
     for J := 1 until B do U(I,J) := V(I,J)
real procedure HORNER (real array A (*); integer value N;
     real value X);
     begin real S; S := 0;
          for I := 0 until N do S := S * X + A(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 := EIDERSIBLING (P);
          M := YOUNGESTOFFSPRING (MOTHER (MOTHER (R)));
          while (M ¬ = null) and (¬ MALE (M)) do
               M := ELDERSIBLING (M);
          if P= null then M else
          if M = null then P else
          if AGE(P) <AGE(M) then P else M
     end

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

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,
          ELDERSIBLING)
```

6. EXPRESSIONS

Expressions are rules which specify how new values are computed from existing ones. These new values are obtained by performing the operations indicated by the operators on the values of the operands. The operands are either constants, variables or function designators, or other expressions, enclosed by parentheses if necessary. The evaluation of operands other than constants may involve smaller units of

action such as the evaluation of other expressions or the execution of statements. The value of an expression between parentheses is obtained by evaluating that expression. If an operator has two operands, then these operands may be evaluated in any order with the exception of the logical operators discussed in 6.4.2.2. 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 that 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, the shorter expression will be padded on the right with blanks to make it the length of the longer one. 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, the string expressions will be padded on the right with blanks to make all alternatives the length of the longest one.

Examples of expressions

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

6.1. Variables

6.1.1. syntax

```
<simple T variable> ::= <T variable identifier> | <T field designator> |
                        <T array designator>
<T variable> ::= <simple T variable>
<string variable> ::= <substring designator>
<T field designator> ::= <T field identifier> (<reference expression>)
<T array designator> ::= <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

<ƒ function designator> ::= <ƒ function identifier> | <ƒ 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> ::= tinteger term> div tinteger 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 $\mathcal{T}_2$. (Also see page 134.)

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 ≠ 0) as

A div B = SGN (A x B) x D(abs A, abs B) (cf. 6.3.2.6.)

A and B both must be integer expressions.

For the purpose of the definition above, SGN and D mean

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

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

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

A rem B = A - (A div B) x B

A and B both must be integer eqressions.

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 $\mathcal{T}_0$, $\mathcal{T}_1$, and $\mathcal{T}_2$ are to be replaced by same combination of words from the table below. If the value of the exponent, N, is positive, then the first operand is multiplied by itself N times; if N is negative, the expression is evaluated as 1/(first operand**(-N)); if N is zero, the result is always 1. If the first operand is zero and the second operand is negative, then division by zero will result. Note that -1**N is parsed as -(1**N); use (-1)**N instead. To force something like I**J (where I ≥ 0 and J ≥ 0) to be an integer, use TRUNCATE(I**J).

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 the same types.

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 τ_0, τ_1, and τ_2 must be replaced by any of the combinations of words (or word pairs) in the tables below.

Operators + | -

τ_1 \ τ_2	integer	real	long real	complex	long complex
integer	integer	real	long real	complex	long complex
real	real	real	real	complex	complex
long real --.	long real	real	long real	complex	long complex
complex	canplex	complex	complex	complex	complex
long complex	long complex	complex	long complex	complex	long complex

Operator *

τ_1 \ τ_2	integer	real	complex
integer	integer	long real	long complex
real	long real	long real	long complex
complex	long complex	long complex	long complex

τ_1 or τ_2 having the quality "long" does not affect the type of the result.

Operator /

τ_1 \ τ_2	integer	real	long real	complex	long complex
integer	long real	real	long real	complex	long complex
real	real	real	real	complex	complex
long real	long real	real	long real	complex	long complex
complex	complex	complex	complex	complex	complex
long complex	long complex	complex	long complex	complex	long complex

Table of values for div and rem operators

I	J	I div J	I rem J
10	2	5	0
11	2	5	1
10	-2	-5	0
11	-2	-5	1
-10	2	-5	0
-11	2	-5	-1
-10	-2	5	0
-11	-2	5	-1

Operator **

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

Operator long

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

Operator short

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

6.3.3. Examples.

```
C + A(1) * B(1)

EXP (-x/(2 * SIGMA)) / SQRT (2 * SIGMA)
```

6.4. Logical Expressions

6.4.1. Syntax

In the following rules for <relation> the symbols $\mathcal{T}_0$ 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 primary3 | ¬ <logical primary>
<logical primary> ::= <logical value> | <logical variable> | <logical function designator3 | (<logical expression>)
<relation> ::= <simple $\mathcal{T}_0$ expression> <equality operator> <simple $\mathcal{T}_1$ expression> | <logical element> <equality operator-2 <logical element> | <simple reference expression> is <record class identifier> | <simple $\mathcal{T}_2$ 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 first extended to the right with blanks. The relational operators yield the logical value true if the relation is satisfied for the values of the two operands; false otherwise. Two references are equal if and only if they are both null or both refer to the same record. The operator is yields the logical value true if the reference expression designates a record of the indicated record class; false otherwise. The reference value null fails to designate a record of any record class.

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

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

6.4.3. Examples

P or Q

(X < Y) and (Y < Z)

YOUNGESTOFFSPRING (JACK) ¬ = 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>)

(The | stands for the vertical bar character | .)

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

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

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

(Note: the terminating period is optional.)

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.

<σ_0 assignment statement> ::= <σ_0 left part> <σ_1 expression> |
<σ_0 left part> <σ_1 assignment statement>

<σ left part> ::= <σ variable> :=

7.2.2. Semantics

The execution of a simple assignment statement

<σ_0 assignment statement> ::= <σ_0 left part> <σ_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

(<σ_0 assignment statement> ::= <σ_0 left part> <σ_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 τ_1 is said to be assignment compatible with a simple type τ_0 if either

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

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

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

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

7.2.3. Examples

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

7.5a Assert Statements

7.5a.1 Syntax

<assert statement> ::= assert <logical expression>

7.5a.2 Semantics

The assert statement is equivalent to the if statement:

if ¬(<logical expression>) then endexecution

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

7.6. Case Statements

7.6.1. Syntax

<case statement> ::= <case clause> begin <statement list> 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(1) := -H(I);
      begin H(I-1) := H(I-1) + H(1); I := I-l end;
      begin H(I-1) := H(I-1) *H(I); I := I-l 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> := tinitial 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

Example FOR statement	Values I takes on
for I:=1 step 2 until 10 do	1, 3, 5, 7, 9
for I:=1 step 2 until 1 do	1
for I:=1 step 2 until -10 do	none
for I:=1 step -2 until 10 do	none
for I:=1 step -2 until 1 do	1
for I:=1 step -2 until -10 do	1, -1, -3, -5, -7, -9
for I:=1 step 0 until 10 do	1, 1, 1, 1, 1, 1, ...
for I:=1 step 0 until 1 do	1, 1, 1, 1, 1, 1, ...
for I:=1 step 0 until -10 do	none

Table of results for various FOR statements.

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 $N \leq (E3-E1) / E2 < N+1$. 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-D; <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(I)
```

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	string (<integer number>)
real	logical
long real	bits
complex	
long complex	

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 ($\mathcal{T}_1$ value X_1; . . . ; $\mathcal{T}_n$ value X_n);

procedure WRITEON ($\mathcal{T}_1$ value X_1; . . . ; $\mathcal{T}_n$ 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 $X_1,\ldots,X_n$);

(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. Does nothing if already positioned at the first character of a record.
2	Subsequent output assembly is set to begin with the first character of a record. Does nothing if already positioned at the first character of a record.
3	Like IOCONTROL(2), except that the new record is also caused to begin a new output page. Does nothing if already positioned at the first character at the top of a page.
4	Subsequent automatic page ejects on the printed output are suppressed, thus allowing more than 60 records on a page. This suppresses only the automatic page eject after 60 records; IOCONTROL(3) still works. (Note that some operating systems also have a feature to force page ejects after 60 records.*/
5	Subsequent automatic page ejects on the printed output are allowed; undoes IOCONTROL(4). While the automatic page eject is suppressed, page and line counts are still maintained based on 60 records per page, so a program may still be cut off for exceeding the page estimate. Also, after an IOCONTROL(5), the first automatic page eject may occur after 1 to 60 more records, unless the counters are re-synchronized at that point via IOCONTROL(3).
72	Subsequent use of READ and READON are to use only the first 72 characters of a record; the last eight are ignored. READCARD still reads all 80 characters.
80	Subsequent use of READ and READON are to use all 80 characters of a record.

/ At Stanford, a / PRINT EJECT=NO card must be included next to the

7.8.5. Examples

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

7.8.6. TRACE standard procedure

The number of times each source statement is traced by the debugging facilities (see $DEBUG in the Deck Setup section) can be modified by the standard procedure TRACE.

Implicit declaration heading:

```
procedure TRACE (integer value N);
    comment changes the upper bound for statement tracing:
    if N > 0 then N becomes the bound,
    if N = 0 then tracing is suspended,
    if N < 0 then the $DEBUG card value (m) becomes the bound;
```

TRACE has no effect unless the $DEBUG option digit n is 3 or 4.

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

Table of values for TRUNCATE, ENTIER, and ROUND

8. STANDARD FUNCTIONS AND PREDECLARED IDENTIFIERS

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

8.1. Standard Transfer Functions

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

```
integer procedure TRUNCATE (real value X);
    comment the integer i such that
        |i| < = |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) ;
integer procedure EXPONENT (real value X);
    comment 0 if X = 0, otherwise the largest integer i such that
        i < = log16(|X|) + 1 .
    This function obtains the exponent used in the S/360
    representation of the real number;
real procedure ROUNDTOREAL (long real value X);
    comment the properly rounded value of X ;
real procedure REALPART (complex value Z);
    comment the real component of Z ;
long real procedure LONGREALPART (long complex value Z);
real procedure IMAGPART (complex value Z);
    comment the imaginary component of Z ;
long real procedure LONGIMAGPART (long complex value Z);
```

```
complex procedure IMAG (real value X);
     comment the complex number 0 + 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)
u	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+AAAAAAAAAAAAAA ;
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 ;
```

```
real procedure EXP (real value X);
     comment e ** X ,
       domain : X < 174.67 ;
low 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 (long 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 arc-tangent (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) <n/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=' Argument                          Result Units
                - time of day
              -1                                seconds/60
                - elapsed execution time -
               0                                minutes/100
               1                                seconds/60
               2                                seconds/38400
    The result for any other argument is not defined;
```

8.4. Predeclared Variables

The following variables are to be considered declared and initialized by assignment in the conceptual block enclosing the entire ALGOL W program. The values indicated for real and long real quantities are to be understood as decimal approximations to the actual machine-format values provided.

```
integer 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 2.220446049250313'-16L ,
       the largest positive long real number ε provided by
       the implementation such that
                 1 + ε = 1 ;
long real MAXREAL;
     comment initialized to 7.237005577332263'+75L ,
       the largest positive long real number provided by the
       implementation;
long real PI;
     comment initialized to 3.141592653589793L ;
```

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);
  beain
     XCPNOTED(CONDITION) := true;
     XCPLIMIT(CONDITION) := XCPLIMIT(CONDITION) - 1;
     if (XCPLIMIT(CONDITION) < 0) or XCPMARK(CONDITION) then
       WRITE("***** ERROR NEAR COORDINATE nnnn -");
     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 coordinate of the source code which was being executed when the exceptional condition was detected
endexecution	procedure to terminate execution of the ALGOL W program
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

[2] 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.

condition	XCPACTION≠1 or 2	XCPACTION=1	XCPACTION=2	Reference=NULL
OVFL	exponent 128 too small	± MAXREAL	0	exponent 128 too small
UNFL	exponent 128 too large	0	0	0
DIVZERO	dividend	± MAXREAL	0	dividend
INTOVFL	true result ± 2**32	true result ± 2**32	true result + 2**32	true result ± 2**32
INTDIVZERO	dividend	dividend	dividend	dividend
SQRTERR	0	sqrt(abs x)	0	0
EXPERR	0	MAXREAL	0	0
LNLOGERR	0	-MAXREAL	0	0
SINCOSERR	0	0	0	0

Table of Results for Exceptional Conditions

Example:

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

The values needed for this are:

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

The following assignment statement will establish the proper environment:

```
OVFL := EXCEPTION(FALSE, 10, 1, TRUE, "OVERFLOW FIXED UP");
```

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. (Also see CODE, DECODE on page 139.)

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	(o)	214	0		
97	/	151	(P)	215	P		
107	,	152	(q)	216	Q		
108	%	153	(r)	217	R		
109	_	162	(s)	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

Richard L. Sites

ALGOL W ERROR MESSAGES

The compiler is divided into three passes: pass 1 reads the program, lists it, and saves it in memory-in a compressed (tokenized) form; pass 2 parses the program, examining each statement to see if it is written properly; pass 3 generates the 360 machine code for the program. Each pass is capable of detecting a different set of errors. (There is also a fourth, loader, pass that on rare occasions may generate messages.) Errors may also occur while a compiled program is executing; these are called Run-Time errors.

Pass One Error Messages

All pass 1 error messages are of the form:

ERROR lxxx NEAR COORDINATE yyyy - message

yyyy corresponds to one of the coordinate numbers in the first column on the program listing. If you have many statements on a card, only the coordinate of the first one is on the program listing. Some messages are only warnings, in which case the fixup action taken is indicated below.

The messages are:

1001 INCORRECTLY FORMED DECLARATION

a) STRING(x) or BITS(x), where x is not a number.

b) STRING(0) or STRING(> 256). FIXUP: treated as STRING(l).

c) BITS (not 32).

1002 WARNING: INCORRECT CONSTANT

a) More than 256 digits. FIXUP: treated as 0.

b) A bad exponent. FIXUP: exponent treated as 0.

1003 MISSING "END"

Final "." or /* card or % card encountered before an END matching each BEGIN. The coordinate indicated may be two or three more than the last coordinate on your listing. (Check the block numbers in the second column of your program listing.)

1004 UNMATCHED "END" (DELETED)

An END encountered after what appeared to be the final END. When possible, the innermost END is deleted. (Check the block numbers in the second column of your program listing.)

1005 WARNING: MISSING ")"

STRING(x or BITS(x with no closing ")". FIXUP: supplied.

1006 WARNING: ILLEGAL CHARACTER

A strange character accidently keypunched (or overpunched). It is likely that the character will print as a blank, so look at your card. The characters on a standard keypunch that are illegal except in comments and strings are: ¢ & ! $ % ? @ . FIXUP: treated as a blank.

1007 WARNING: MISSING FINAL "."

May occur if the program ends with an un-terminated string constant or an un-terminated comment.

1008 WARNING: INVALID STRING LENGTH

a) A string constant of length > 256. FIXJP: truncated to 256 characters. (You may have left out a quote.)

b) An empty string constant (""). FIXUP: replaced with "?".

1009 WARNING: INVALID BITS LENGTH

a) "#" not followed by hex digits. FIXUP: replaced with #0.

b) "#" followed by more than 8 hex digits. FIXUP: replaced with #0.

1010 MISSING " ("

REFERENCE not followed by "(".

1011 ERRORTABLE OVERFLOW

More than 50 error messages from pass 1. The rest are lost.

1012 COMPILER TABLE OVERFLOW

The program is too big to fit in memory during compilation. The following is a list of tables which could be full at this point. If you re-compile with more memory, the starred tables will be bigger.

* BCD POINTERS -- if all of your names are short (3, 4 letters) this table may fill up before the id table.

  BLOCK LIST -- 511 entries, one for each BEGIN, PROCEDURE (except for formal parameter specification), and FOR.

  BLOCKSTACK -- this has a fixed size of thirty entries. It will overflow if you have 31 BEGINs nested within each other. (The block numbers in the second column of your program listing show how full this stack is.)

* ID TABLE -- place for the characters in your identifiers.

* NAME TABLE -- table of attributes of all declared identifiers.

* PROGRAM TOKEN SPACE -- the internal text for the program. This is the most likely table to be full.

* REFERENCE LIST -- information about each variable declared of type REFERENCE.

1013 WARNING: ID LENGTH > 256

One of the names in your program is much too long. FIXUP: truncated to 256 characters.

1014 WARNING: UNEXPECTED "."

An apparently final "." not followed by % card or /* card, such as in a constant with an inadvertant space: . 123 . FIXUP: treated as a blank.

1015 TOO MANY RECORD CLASSES

Only 15 are allowed.

1016 WARNING: SEQ FIELD OUT OF ORDER

a) The numeric part of columns 73-80 was not greater than the numeric part of the previous card.

b) The alphabetic part of columns 73-80 was not the same as the alphabetic part on the previous card.

In either case, the offending card(s) is marked with #### on the listing. This message appears only once in any single compilation. The coordinate specified is the coordinate on the first erroneous card.

1017 WARNING: SEQ FIELD CONTAINS TRASH

a) The first card of the deck did not contain a sequence number, but columns 73-80 on this card are not all blank. (A statement may have accidently run past column 72).

b) The first card of the deck has a non-blank sequence field (columns 73-80), but there are no digits in it.

In either case, the offending card(s) is marked with **** on the listing. Like 1016, this message appears at most once, and the coordinate refers to the first instance.

1018 WARNING: ";" DELETED BEFORE "ELSE"

This is a common mistake that the compiler fixes up.

Pass Two Error Messages

AU pass 2 error messages have the format:

ERROR 2xxx NEAR COORDINATE yyyy - message

(FOUND NEAR "...")

yyyy corresponds to one of the coordinate numbers in the first column on the program listing. If you have many statements on a card, only the coordinate of the first one is on the program listing. "..." is the program text being scanned at the time the error is detected (which may be somewhat after the actual point of error). If any pass one or pass two error messages occur (other than warnings), then compilation stops at the end of pass two. Often many error messages are generated for what is essentially a single mistake.

2001MORE THAN ONE DECLARATION OF "XXXX" IN THIS BLOCK

The variable XXXX has been declared more than once in the same block.

2002 "XXXX" IS UNDEFINED

The variable or label XXXX has not been declared in the current block or in one containing it.

2003 SYNTAX ERROR

This is a "catch-all" message that is produced when the compiler cannot find anything more meaningful to say. The current context will point to the part of the program being analyzed when the error was DETECTED, but in general the real error may be much earlier in the program. If the current context is at or near a semi-colon and you cannot find any errors there, try looking at the beginning of the statement which ends at that semi-colon. If the current context is at or near an END, try-looking at the corresponding BEGIN. For example, if ELSE BEGIN . . . END; occurs, but not after an IF, the compiler will not detect the error until it reaches END; .

2004 IDENTIFIER MUST BE RECORD CLASS ID

In a declaration REFERENCE(xyz) , xyz is not the name of a record class.

2005 MISMATCHED PARAMETER

A procedure call is passing an actual parameter which is not of the same type as the formal parameter in the procedure declaration.

2006 INCORRECT NUMBER OF ACTUAL PARAMETERS

The number of actual parameters in a procedure call does not equal the number of formal parameters in the procedure declaration.

2007 INCORRECT DIMENSION

a) The number of dimensions of an actual parameter does not equal the number of dimensions declared for the corresponding formal parameter.

b) The wrong number of subscripts have been used in an array element reference.

2008 DATA AREA EXCEEDED

The data for each PROCEDURE or BEGIN block with declarations is limited to 4096 bytes. Read the suggestions for 3001.

2009 INCORRECT NUMBER OF FIELDS

In creating a record, too many or too few initial values have been specified.

2010 INCOMPATIBLE STRING LENGTHS

a) In STRING1 := STRING2 , STRING2 is longer than STRING1.

b) In STRING3(x|y) , y is larger than the declared size of STRING3.

c) A long string has been passed to a shorter formal string parameter.

2011 INCOMPATIBLE REFERENCES

A reference variable refers to a wrong record class.

2012 BLOCKS NESTED TOO DEEPLY

Non-trivial blocks (i.e., BEGIN blocks with declarations, or the blocks associated with a PROCEDURE) are nested more than eight deep (including the BEGIN at the start of the program). The error is detected early in the ninth block. Also, procedure calls nested too deeply.

2013 WARNING: ";" SHOULD NOT FOLLOW EXPRESSION

In BEGIN . . . expression ; END the semi-colon is incorrect but ignored.

2014 REFERENCE MUST REFER TO RECORD CLASS

In REFERENCE(xyz)... , xyz is not a record class.

2015 EXPRESSION MISSING IN PROCEDURE BODY

A function PROCEDURE must have its final value specified by an expression standing alone immediately before the END.

2016 IMPROPER COMBINATION OF TYPES

Mixing incompatible types as alternatives of a conditional or case expression.

2017 RESULT PARAMETER MUST BE A VARIABLE

In a procedure declaration, a formal parameter is declared ... RESULT xyz , but a call to that procedure has passed an expression which is not a variable.

2018 PROPER PROCEDURE ENDS WITH AN EXPRESSION

A procedure which returns no value nonetheless ends with an expression. (This sometimes happens when a final assignment statement has been mis-punched A = B , instead of A := B .)

2019 "XXXX" CANNOT FOLLOW "YYYY" HERE

There are no legal programs in which XXXX and YYYY can be written together. This is much like 2003. (You may have left out a semi-colon, a comma, or an operator.)

2020 ARRAY USED INCORRECTLY

A simple variable must be used here.

2021 TOO MANY CONSTANTS IN PROCEDURE

No more than 256 different constants are allowed.

2022 INCORRECT STRING LENGTH

In S(x|y) , Y is negative, zero, or greater than 256.

2023 COMPILER TABLE OVERFLOW

The program is too big to fit into memory during compilation -- there is no more room for the parse trees that represent the program at this point. If you re-compile with more memory, there will be more room available for the program.

2024 TOO MANY PROCEDURES

Only 255 different procedures or BEGIN blocks with declarations are allowed by the compiler.

2025 CONSTANT OUT OF RANGE

a) The absolute value of an integer is greater than (2**31)-1 (9+ digits).

b) The absolute value of the adjusted exponent in a real number is greater than 75. The exponent written is first adjusted to include the number of digits written in front of the decimal point.

2026 INDEX OF ARRAY OR STRING MUST BE INTEGER

a) In S(x|y) , x is not an integer expression.

b) In Arrayname(...x...) , x is not an integer expression. (You may have accidently used a REAL variable.)

2027 INCORRECT OPERAND TYPE(S) FOR XXXX

XXXX is a unary operator.

a) LONG is applied to something which is LOGICAL, STRING, BITS, or REFERENCE.

b) SHORT is applied to something which is LOGICAL, STRING, BITS, or REFERENCE.

c) ¬ (not) is applied to something which is neither LOGICAL nor BITS.

d) Prefix + or - is applied to something which is LOGICAL, STRING, BITS, or REFERENCE.

e) ABS is applied to something which is LOGICAL, STRING, BITS, or REFERENCE.

f) In Recordvariable , x is not a REFERENCE.

g) In FOR 1:=x... , x is not an integer expression.

h) In various other contexts, an INTEGER or LOGICAL operand is required.

2028 INCORRECT OPERAND TYPE(S) FOR XXXX

XXXX is a binary operator. Even when the error is in the first operand, the error is detected after both operands are inspected.

a) AND or OR is applied to expressions which are not both BITS or both LOGICAL. This case often happens in an IF statement when necessary parentheses are left out;

```
IF X < Y OR Z = 3 THEN . . .
```

As written, y is to be ORed with z before anything else is calculated. Try instead:

```
IF (X < Y) OR (Z = 3) THEN . . .
```

b) A relational operator (like >) is applied to something which is COMPLEX, LOGICAL, or REFERENCE.

c) SHL or SHR is applied to something which is not BITS, or the shift amount is not INTEGER.

d) In x IS Recordclass , x is not a REFERENCE.

e) In x**y , x is LOGICAL, STRING, BITS, or REFERENCE, or y is not INTEGER.

f) In a FOR statement, the UNTIL expression is not INTEGER.

g) In various other contexts, an INTEGER operand is required.

2029 INCORRECT PARENTHESIZATION OF EXPRESSION

This often occurs in conjunction with 2027 or 2028. Usually, additional parentheses are required in the expression.

2030 ASSIGNMENT INCOMPATIBILITY

An attempt to assign an expression of one type to a variable of a different type (or pass an actual parameter to a formal parameter of a different type). The only automatic conversions allowed are INTEGER to REAL, INTEGER to LONGREAL, REAL to/from LONGREAL, INTEGER/REAL/LONGREAL to COMPLEX/LONGCOMPLEX, COMPLEX to/from LONGCOMPLEX. (You cannot assign REAL to INTEGER without using TRUNCATE, ENTIER, or ROUND.)

2031 WARNING: NAME PARAMETER SPECIFIED

In a PROCEDURE declaration, it is usually intended that each formal parameter have VALUE specified.

2032 SIMPLE VARIABLE USED INCORRECTLY

In " x(", x is a simple variable and not STRING.

2033 75 ERRORS. COMPILATION TERMINATED

Something is drastically wrong with your program. To save time and paper, the rest of the program is ignored.

2999 DEBUG TABLE OVERFLOW

If $DEBUG,x is specified with x equal to 2, 3, or 4, then a table is created with a fixed maximum of 448 entries, where one entry is used for each GROUP of statements that all occur together with no labels, branches or conditional expressions. All the statements in such a group are guaranteed to be executed the same number of times. Also, this message occurs if the compressed form of the program occupies more than 65536 bytes of memory (the compressed form is used to generate the pseudo-listing with the statement counts).

Pass Three Error Messages

Pass 3 error messages are of the form:

ERROR 3xxx NEAR COORDINATE yyyy - message

yyyy corresponds to-one of the coordinate numbers in the first column on the program listing. If you have many statements on a card, only the coordinate of the first one is on the program listing.

All of the pass 3 errors are disastrous, so compilation terminates immediately. After any pass 3 error, a table is listed of (coordinate number, byte offset, byte length) triples, indicating how much code was generated for each statement in the current program segment. The last entry of this table and the last two byte lengths are usually garbage.

3001 PROGRAM SEGMENT OVERFLOW

This error message occurs because of a design constraint of the compiler: the total amount of machine code and constants for any PROCEDURE or other BEGIN block with declarations must be less than 8192 bytes. All of the constants for a block are allocated in front of the first statement. Therefore, if the byte offset of the first statement is very large, constants are taking up too much space. This sometimes happens in programs with'too many string constants (ten 80-character string constants take up 800 bytes). The coordinate indicated may or may not be very accurate. The only solutions are to make your program smaller, or to add some artificial PROCEDUREs or BEGIN blocks with at least one declaration, such that part of the block that was too big is forced into another segment.

3002 COMPILER STACK OVERFLOW

While generating code for a statement, the compiler uses a push-down stack to keep track of where it is in the statement tree. If you are about to get a PROGRAM SEGMENT OVERFLOW (3001), you may get this message instead.

3003 COMPILER LOGIC ERROR

Internal consistency checks performed by the compiler have failed. Take your card deck, exactly as it is, to a consultant.

3004 PROGRAM AREA OVERFLOW

Although the words are similar to 3001, this is entirely different. This message means that there is no more room in memory to put the machine code for your program (like 2023 and 1012). If you re-compile with more memory, there will be more room available for the machine code.

3005 DATA SEGMENT OVERFLOW

The data for each PROCEDURE or BEGIN block with declarations is limited to 4096 bytes. Read the suggestions for 3001.

4006 COORDINATE TABLE OVERFLOW

In order to supply the coordinate number in run-time error messages, a table is built of (coordinate number, address in machine code) pairs. If you re-compile with more memory, this table will be larger.

'3007 TOO MANY PROCEDURE CALLS

References to only 31procedures are allowed within any single procedure.

Loader Error Messages

Loader error messages are all of the form:

*** LOADING ERROR - message

Like pass 3 messages, these are disastrous and terminate processing.

DUPLICATE GLOBAL NAME - XXX	Two procedures with the same name were loaded.
INSUFFICIENT STORAGE	Not enough room to run the program. Re-run with more memory.
INVALID OBJECT RECORDS	A bad object card was presented, often an extra blank card.
NO EXECUTABLE STATEMENTS	No main program was loaded, only external procedures.
TOO MAN-Y PROCEDURES	Only 96 program segments are allowed by the loader.
UNDEFINED GLOBAL NAME - XXX	An external procedure was declared, but not loaded.

Run Time Error Messages

All run error messages are of the form:

RUN ERROR NEAR COORDINATE yyyy IN procedure name - message

After a run error, a post-mortem dump of all of the program's variables is given, unless it is explicitly turned off with a $DEBUG,0 card. To keep the dump reasonably small, at most eight values are dumped from an array. If the same identifier is declared in many blocks (note that the index variable in a FOR loop is considered to be declared in a block around just the FOR statement), then that identifier will be listed many times. Variables which have never been assigned any meaningful value are printed as ".?" .

ACTUAL-FORMAL MISMATCH IN PROCEDURE CALL, PARAMETER #xx

The actual parameter passed is not assignment compatible with the formal parameter.

ARRAY SUBSCRIPTING

An array subscript was not within the declared bounds.

ARRAY TOO LARGE

The first n-l dimensions of an array declaration define too many elements. The product of the size of a single element times the first n-l dimension lengths (upper bound-lower bound+1) must be strictly less than 32768. The element sizes are:

logical	1
integer, real, bits, reference	4
long real, complex	8
long complex	16
string	length of a single string

ASSERTION xxxxxxx FAILED

An assertion was not true. xxxxxxx is a running count of how many assertions were true, to give a feel for how long the program had run.

ASSIGNMENT TO NAME PARAMETER

Attempt to assign to a name parameter whose actual argument is not a variable, but is instead an expression, a constant, or a control identifier.

CASE SELECTION INDEXING

Index in a case statement or case expression is less than 1 or greater than the number of cases.

DATA AREA OVERFLOW

No more storage is left for variables. This will happen if a program gets in a loop calling itself recursively, or if there really is not enough memory.

DIVISION BY ZERO

May also be caused by 0**(-n) .

EXPERROR

The argument to EXP must be less than 174.67 .

INCOMPATIBLE FIELD DESIGNATOR

An attempt has been made to access a field of a record, but the reference does not designate a record of the corresponding class (it might be NULL or undefined).

INCORRECT NUMBER OF PARAMETERS

The number of actual parameters in a procedure call is different from the number of formal parameters declared in the called procedure.

INTEGER DIVISION BY ZERO

An integer operation attempted to divide by zero.

INTEGER OVERFLOW

An integer operation produced a number whose absolute value is bigger than (2**31)-1 . The standard functions ROUND, TRUNCATE, and ENTIER will produce an integer overflow if presented with arguments whose absolute value is bigger than (2**31)-1 .

LENGTH OF STRING INPUT

The string read was longer than the string variable has room for. This sometimes happens if a string ends in exactly column 80 of a card, and another string begins in column 1 of the next card, since the two quote marks (col 80 and col 1) are part of the same string. Put at least one blank in between (or a whole blank card). Also, check for a missing quote.

IN/LOG ERROR

An attempt to take the logarithm of a negative or zero number.

LOGICAL INPUT

The quantity read was not TRUE or FALSE.

NULL ORUNDEFINED REFERENCE

An attempt has been made to access a record field using a null or never initialized reference.

NUMERICAL INPUT

The number read was not assignment compatible with the variable in the READON or READ statement. This sometimes happens when running from a terminal if the line numbers on the data cards are accidently read.

OVERFLOW

A real operation produced a number whose absolute value is bigger then 7.2'+75 . This may occur when dividing by a very small number, such as in 1'+50/1'-50 .

PAGE ESTIMATE EXCEEDED

The page estimate on the %ALGOL card is exceeded. Note that any tracing ($DEBUG,3 or 4) output is included in this page limit. (cf. Deck Setup and Compiler Options, page 103.)

PROGRAM CHECK #M

The compiler or the code it generated was wrong. If this happens, take your card deck, exactly as it is, to a consultant.

READER EOF

No more data cards. A % card or a /* card was read instead. This is a normal way to terminate in many programs.

RECORD STORAGE AREA OVERFLOW

No more storage exists for records.

REFERENCE INPUT

References cannot be read.

SIN/COS ERROR

See the domain restrictions in Section 8.2.

SQRT ERROR

Attempt to take the square root of a negative number.

STRING INPUT

A null string or a string greater than 256 characters was read. See LENGTH OF STRING INPUT above.

SUBSTRING INDEXING

Substring selected extends off one end of the string

TIME ESTIMATE EXCEEDED

The time estimate on the %ALGOL card is exceeded.

UNDERFLOW

A real operation produced a number whose absolute value is less than 5.4'-79 , but not exactly zero. This may occur when dividing by a very large number, such as in 1'-50/1'+50 .

ABEND Messages

You may occasionally get terse messages on the first page of your output of the form:

*** ABNORMAL JOB END *** SYSTEM CODE X xxx

or

COMPLETION CODE - SYSTEM = xxx

where xxx might be:

222 322 722	You ran out of time or lines as specified on your JOB card (not the limits on the %ALGOL card). (cf. page 103.)
0C1 0C4 0C6	The compiler probably made a mistake. After verifying that the deck or catalogued procedure includes both a //SYSPRINT and //SYSIN DD card, take your deck, exactly as it is, to a consultant.

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 105 or 106 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

$$21 = \underline{0} \times 2^{30} + \ldots + \underline{0} \times 25 + \underline{1} \times 2^{4} + 0 \times 2^{3} + 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} .$$

The largest integer that can be stored in a word is

$$2^{30} + 2^{29} + \ldots + 2^{1} + 2^{0} = 2^{31} - 1 = (2147483647)_{10} .$$

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/j60 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 $2^{10} \doteq 10^3$.

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@ 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 numbers 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 conformswith 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 from 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 from 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.

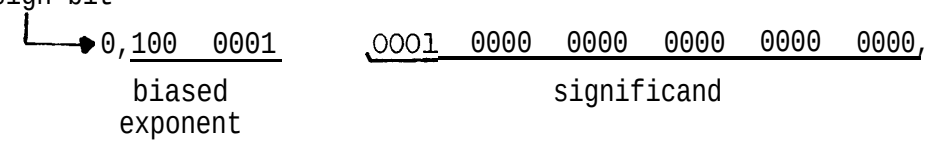
As with integers, the sign of the number is denoted by bit 0, with 0 representing + and 1 representing - .

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 as the exponent, we would have no negative exponents, and our range of floating-point numbers could not include such numbers as 16^{-25}. It is desirable to have an exponent range that is approximately symmetric about zero. In System/360 one obtains the true exponent of the floating-point number by subtracting 64 from the biased exponent represented by bits 1 to 7. As a result, the actual exponents range from -64 to 63.

The 24 bits 8 to 31 of a number are regarded as six hexadecimal digits with a hexadecimal point at the left-hand end. If the floating-point number zero is being represented, all the hexadecimal digits 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 some sample numbers. As we said before, the number zero is represented by 32 zero bits, i.e., by eight 0 hexadecimal digits. Thus zero is represented by the same words in floating-point or integer form. No other number has this property.

The number 1.0 is represented by the word



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} . fitting a hexadecimal point at the left end gives the hexadecimal fraction 0.100000_{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, . 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, . 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 $\times 16^{3F}$ 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

significant decimal digits.

Not only do errors appear in the representation of numbers inside System/360 (or any computer), but they arise from arithmetic operations performed 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 interesting 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 analysis 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 somewhere, just in order to come 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 9999999-11

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 microseconds, while that for double-precision factors takes around 7 microseconds. 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 t-he 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 LONG REALS

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 thehardwarerepresentation 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 3.1415926535897932 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 := 3.1415926535897932~
```

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 in Section 6.3 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?

DECK SETUP
AND
COMPILER OPTIONS

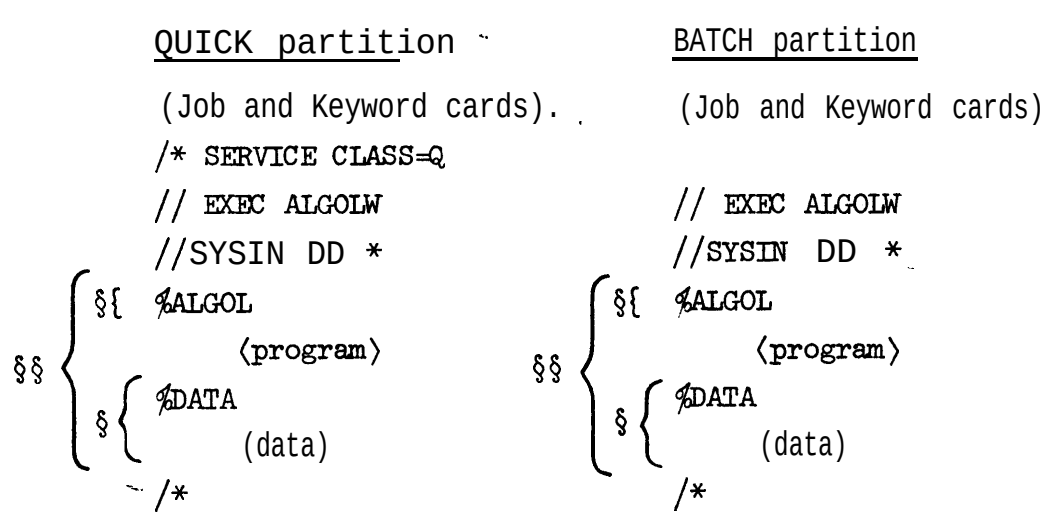
by

Richard L. Sites

1. DECK SETUP

ALGOL W Deck Setup and Compiler Options

1. Simple Deck Setup



§ Optional.

§§ May be repeated -- second and following %ALGOL cards are required.

For simple cases, the above control cards are sufficient. More complicated cases are discussed later under 3. Linkage to Separately-Compiled Procedures.

1.1 Time and Page Limits

To avoid using too much computer time or paper when a program has mistakes in it, both the operating system and the ALGOL W system monitor the amount of time and pages used. The operating system keeps track of the total time used for compiling one or more programs, executing them, printing any post-mortem dumps, loading the compiler into core, interpreting the operating system control cards, etc. The operating system also keeps track of the total amount of printed output from a run -- control card listing, compiler listing, actual execution output, error messages,

post-mortem dump, etc. The limits for these totals are specified on the JOB card in tenths of minutes and thousands of lines; exceeding these JOB card limits results in an ABEND 322 message from the operating system and no other information.

The ALGOL W system monitors the amount of time and pages used by each program just during its execution, not during its compilation or during any post-processing. If these execution limits are exceeded, ALGOL W will print a run-time error message (TIME ESTIMATE EXCEEDED or PAGE ESTIMATE EXCEEDED) with the coordinate of the program statement executing at the time. The subsequent post-mortem dump and optional program listing can be very helpful in determining what went wrong. To make sure that the ALGOL W system is able to get out this information, the JOB card limits always should be sufficiently bigger than the ALGOL W limits.

The normal ALGOL W execution limits are 10 seconds and 9 pages (60 lines/page). These may be changed by specifying different limits on the %ALGOL card in columns 8-29:

```
%ALGOL  TIME=sss,PAGES=ppp
```

where sss is the maximum execution time in seconds; ppp is the maximum number of pages of execution and tracing output. TIME may be abbreviated T; PAGES, P. Time and Pages may be given in either order.

Example: for 2 minutes and 20 pages, use:

```
%ALGOL T=120,P=20
```

(Previous versions of the compiler had slightly different control cards: %EOF instead of %DATA, and min:sec,pages instead of TIME= and PAGES=. These older conventions are also accepted by the present compiler.)

1.2 Other %ALGOL Card Parameters

Two other execution environment options may appear on the %ALGOL card. MARGIN=72 specifies that READ and READON should only scan the first 72 columns of data cards. MARGIN=80 specifies that READ and READON should scan all 80 columns of data cards. The default value is MARGIN=80, unless the program source cards are sequence numbered; in that case, it is assumed that the data cards are also sequence numbered and MARGIN=72 is the default. MARGIN may be abbreviated MARG. (cf. Section 7.8.4. for dynamic control of this margin.) SIZE=xxxK specifies that the maximum amount of dynamic space requested by either the compiler or the execution library is xxx*1024 bytes. This directive is only used in rare cases to prevent the compiler from using all of the core available to it.

TIME, PAGES, MARGIN, and SIZE may be specified in any order.

2. Compiler Options

Any of the following cards can appear in a deck between a %ALGOL and the next %card:

$NOLIST — Do not list subsequent source cards. The compiler normally lists all input cards.

$LIST — List subsequent source cards: this undoes a previous $NOLIST.

$TITLE,"..." — Start the program listing on the next page, and place "..." (up to 30 characters) as a title in the middle of the heading line.

$SYNTAX — Analyze the program for syntax errors, but do not execute.

$STACK — Dump the current parsing stack if a pass 2 syntax error should occur, with the most recent syntactic element listed last.

$DUMP*ab,cc — Dump certain internal tables during a compilation. This option in general is used only by those maintaining the compiler, but is documented here for the sake of completeness. Since its use significantly increases the amount of printed, output for even small compilations, random experimenting is discouraged. See the table at the end of this section.

$NOCHECK — Omit checking subscript ranges and reference compatibility and omit initialization of variables to "undefined".

$DEBUG,n(m) — Activate the tracing, statement counting, and post-mortem dump facilities of the ALGOL W system.

The single digit n specifies:

0 nothing fancy (use this to minimize the space used by the system).
1 a post-mortem dump of all the program's variables if execution terminates abnormally, else nothing.
2 the above plus counts of how often each statement was executed.

3 the above plus a statement-by-statement trace of each value stored.

4 the above plus a trace of each value fetched.

If tracing is specified ($DEBUG,3 or $DEBUG,4) and the standard procedure TRACE (cf. Section 7.8.6.) is not used, then each ALGOL statement will be traced in symbolic form the first m times it is executed. Each time a statement is traced, it produces at least two lines of output (included in the run-time limit), so a 100 statement program with $DEBUG,3(2) will produce at least 400 lines of output (unless it dies an early death).

THE DEFAULT IS $DEBUG,1 -- post-mortem dump, but no counts or traces.

The following abbreviated control cards are acceptable:

$DEBUG for $DEBUG,4(2)
$DEBUG, X for $DEBUG,x(2)
(no DEBUG card) for $DEBUG,1

All variables are initialized to a bit pattern considered to represent an undefined value (printed in the traces and post-mortem dump as "?"). For some data types, all bit patterns can be valid, so valid data can appear to be undefined.

See Section 4, page 111, for a detailed explanation of the debugging facilities.

$DUMP* options

The $DUMP* card specifies two things: what tables to be dumped, and which segments in the program the dumping applies to. 'For example, the 360 machine code for only one of many procedures can be dumped.

General format:

$DUMP*ab,cc

a is a single digit and is ignored.

b is a single digit and asks for some combination of 5 tables to be dumped.

cc is exactly two digits -- a number in the range 0 to 63, or two blanks. If cc is blank, then tables for all segments will be dumped. If cc is a number, then the machine code for only that segment will be dumped. Many $DUMP* cards may be used to specify more than one segment. If the b digits are different, the last one is used.

	tables dumped:				
b digit	pass2 parse tree	pass2 nametable	pass2 editcode (hex)	pass3 360 code w/ some addresses missing	pass3 360 code w/ most addresses inserted
					X
				X	X
		X			
		X			X
5		X		X	X
6	X	X			
7	X	X	X		X
8	X	X	X	X	X
9	X	X	X		X (same as 7)

3. Linkage to Separately-Compiled Procedures

ALGOL W provides a facility for pre-canpiling procedures and linking them back together again. For small programs, it is not worthwhile to use this facility, since re-compiling a procedure may be faster than punching an object deck and reading it back in. A facility is provided for generating standard IBM linkages for calling FORTRAN programs.

3.1 Compiler Organization

As shown in the diagram below, there are actually two versions of the ALGOL W compiler; both versions use exactly the same code for the various phases of-the compiler and for the run-time library, but the monitor phase is slightly different. The compile, load, and go incore version is called ALGOLW; it can handle object decks only in a crude way, but its in-core loader handles the debugging feature information. The compile only version is called ALGOLY; it produces standard OS/360 object decks, but cannot pass any debugging information (so $DEBUG,0 is forced). The output from ALGOLY can be link-edited with other object decks or load modules, including those produced by Fortran G or H. In order to be executable, the object decks from ALGOLY must be link-edited or loaded with the ALGOL library and with the ALGOL run-time monitor (ALGOLX). To facilitate this, all object decks for ALGOL main programs include external references to the monitor and to the library.

The restricted object deck facility for the compile, load and go version only handles:

1) object decks
2) of procedures (not main programs)
3) from ALGOL w
4) run with no debugging features ($DEBUG,0) .

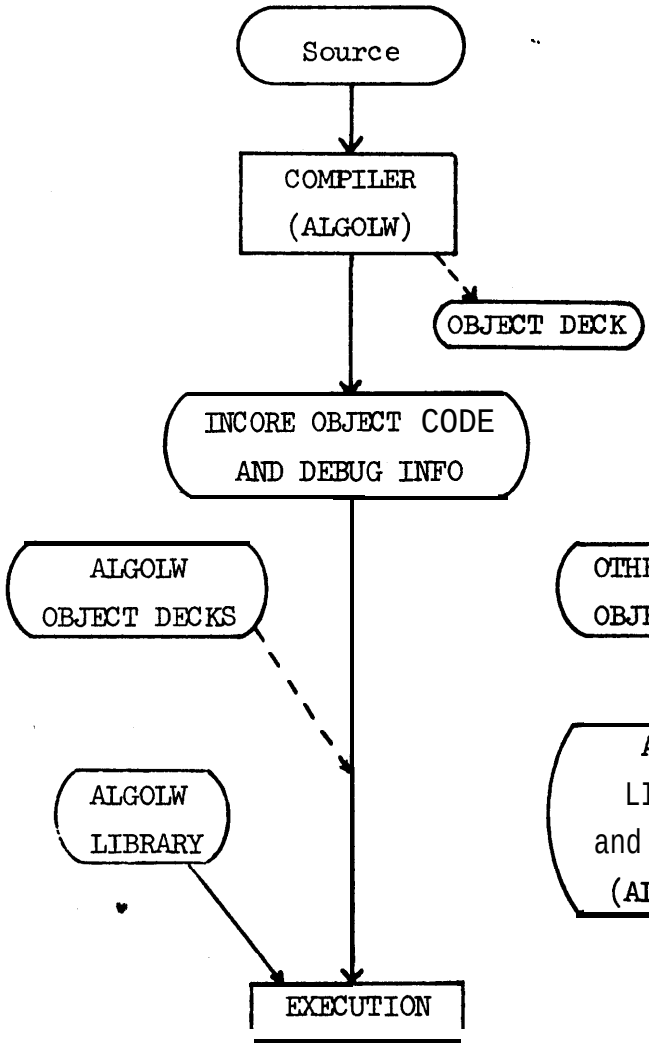
If a procedure declaration is compiled and a //SYSPUNCH DD card is supplied, then an OS/360 object deck for that procedure is produced. This deck can then be used with the link-editor or OS/360 loader as above, or it can be read back into the compile, load, and go system when the main program is compiled. For this purpose, the deck setup is extended to:

```
§§ { §{ %ALGOL
       $DEBUG,O (must be specified)
            (main program)
     § { %OBJECT
            (procedure object deck(s))
     § { %DATA
            (data)
       /*
```

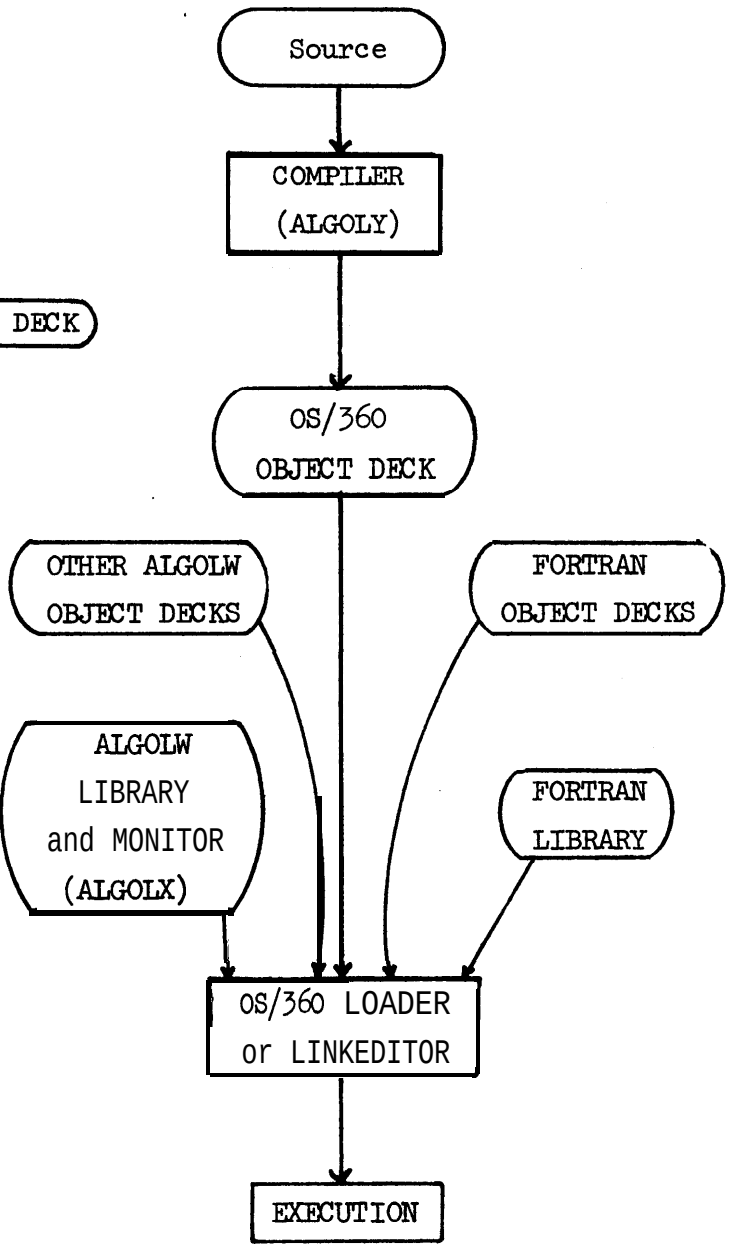
§ Optional.

§§ May be repeated -- second and following %ALGOL cards are required.

COMPILE, LOAD, and GO INCORE



COMPILE and use OS/360
LOADER or LINKEDITOR



3.2 Control Cards for Using OS/360 Loader

Three catalogued procedures are provided: ALGOLCG, ALGOLC, and ALGOLG, for compile and load, compile only, and load only respectively. In all of them, the object decks are passed in the same way that Fortran object decks are passed, so (for instance) ALGOLC and FORTHC can be intermixed and followed by ALGOLG. The stepnames are COMP and GO. Parameters given on a %ALGOL card are not passed to the GO step; instead, the EXEC card parameter field is decoded the same way.

Example:

```
//STEPA EXEC ALGOLCG,PARM.GO='MAP, EP=ALGOLX/TIME=5, PAGES=15 '
```

3.3 Calling External Procedures

In a program which calls an external procedure, a dummy procedure declaration and body are used to establish the proper correspondence (cf. Section 5.3.2.4). The symbols algol and fortran in that dummy body indicate the use of ALGOL W and standard IBM linkages respectively; the associated string is extended (with blanks) or truncated to eight characters and is used as the entry point name of the external procedure. For a FORTRAN external procedure, the entry point name is just the name of the FORTRAN subroutine or function. For an independently compiled ALGOL W procedure, the entry point name is the procedure identifier extended (with " # "s) or truncated to five characters and followed by "001" .

Example:

first compilation

```
INTEGER PROCEDURE MYFUNCTION(REAL VALUE X);
   BEGIN INTEGER I;
      :
   I
   END.
```

second compilation

```
BEGIN
INTEGER K,L,M;
REAL A,B;
INTEGER PROCEDURE YOURFUNCTION(REAL VALUE Y);
     ALGOL "MYFUNOOl";
  :
K := YOURFUNCTION(A);
  :
Em.
```

A FORTRAN subroutine or subprogram can be used as an ALGOL W procedure. The type correspondence between ALGOL W and FORTRAN is given by the following table:

ALGOL W	IBM FORTRAN IV
integer	INTEGER*4
real	REAL*4
long real	REAL*8
complex	COMPLEX*8
long complex	COMPLEX*16
logical	LOGICAL*1
(string)	(LOGICAL*n)
bits	LOGICAL*4
reference	- - -

String functions are not implemented. The following formal parameter types are allowed and are interpreted as indicated:

(1) (simple T type)

The corresponding actual parameter is examined. If that parameter is a variable, the address of that variable is computed (once only) and transmitted. Otherwise, the expression which is the actual parameter is evaluated, the value is assigned to an anonymous local variable, and the address of that variable is transmitted.

(2) (simple T type) value , (simple T type) result ,
(simple T type) value result

As in ALGOL W procedures, a local variable unique to the call is created, and the address of that variable is transmitted.

(3) (simple T type) array

The address of the actual array element with unit indices in each subscript position is computed and transmitted, even if that element lies outside the declared bounds of the ALGOL W array. Arrays with only one dimension and arrays with unit lower subscript bounds will have elements with indices which are identical in ALGOL W and FORTRAN routines. Array cross-sections should not normally be used as actual parameters of FORTRAN subprograms.

If FORTRAN input/output or FORTRAN error handling facilities are to be used, the subroutine package IBCOM, or a suitable substitute, is required.

Example;

Algol W compilation

```
BEGIN
   COMPLEX Z;
   COMPLEX PROCEDURE COMPLEXSQRT(COMPLEX VALUE A);
      FORTRAN "FAKEIT";

   Z := COMPLEXSQRT(Z);
      :
END
```

Fortran compilation

```
FUNCTION FAKEIT(X)
COMPLEX FAKEIT,X
FAKEIT = CSQRT(X)
RETURN
END
```

4. Compiler Output

11.1. Introduction

The printed output of the compiler consists of five general categories :

1) Source card listing
2) Error messages
3) Run-time and tracing output
4) Statement counts
5) Post-mortem dump

The amount of output in some of these categories can be controlled by various compiler options (cf. Compiler Options, page 104).

1) $NOLIST, $LIST, $TITLE.
2) No control.
3) $DEBUG,3 or $DEBUG,4 activates the tracing. The standard procedure TRACE (cf. Section 7.8.6.) dynamically controls the tracing output.
4) $DEBUG,2 , 3 or 4 activates the statement counts.
5) If a program terminates with a run error and $DEBUG,0 was not used, a post-mortem dump is produced.

(In the explanation which follows, circled numbers are keyed to the circled numbers on the sample output.)

4.1.1. Source Card Listing

The source listing consists of four columns of output:

a) Coordinate number ①

This statement count is incremented once for each semi-colon (except end-of-comment), BEGIN, or ELSE in the program. If there are many statements on a card ⑤ , the coordinate listed refers to the first statement on that card. All error messages and tracing information are keyed to the coordinate numbers.

b) Block nesting level$_0$2

The nesting level counter is incremented by one for each BEGIN in the program and decremented by one for each END. The counter is printed only when it changes; then the first character in this column refers to the nesting level of the first BEGIN on the card, and the second character refers to the nesting level of the last END on the card. If you have the proper number of BEGINs and ENDs, the nesting level for the last card should be 1 .

c) Card image$_0$3

Columns l-72 of each card are printed exactly as they were read. $ option cards are not printed.

d) Sequence field$_0$4

Columns 73-80 of each card are printed here, with eight spaces between column 72 (card image) and column 73 (sequence field) (6).

The source card listing is followed by a line giving the options which will be in effect during the execution of the program (6a) . These include the debugging option (specified by a $DEBUG card), the time limit in seconds, the page limit, the word NOCHECK if that option has been specified (cf. Section 2, Compiler Options), and the words MARGIN=72 if the initial right margin for READ and READON is set at column 72 instead of 80. This last option is set if the source deck is sequence numbered, on the assumption that the data cards are also (cf. Section 7.8.4. for more details on margins).

4.1.2. Error Messages (6b)

These are printed immediately after the source card listing and are further explained in the Error Messages section of this manual.

4.1.3. Compile Time and Amount of Code (6c)

The last line of the compilation gives the amount of time spent in the compiler and either the phrase NO CODE GENERATED if fatal error messages occurred, or the phrase (xxxxx, yyyyy) BYTES OF CODE GENERATED if

compilation was successful. xxxxx is the number of bytes of /360 machine language generated. yyyyy is the number of bytes of information generated for the debugging facilities:

$DEBUG,n and above	information included
0 (i.e., always)	Table relating coordinate numbers to program addresses, for creating RUN ERROR messages.
1	Table of names and types of each variable used, for post-mortem dump and tracing.
2	A compressed version of the source code, for the pseudo-listing.
3,4	Additional editing markers in the compressed source code, for breaking the tracing at the proper points, and for more closely correlating the machine code with the source code.

4.1.4. Run-time and Tracing Output

This category includes an optional statement-by-statement trace of the program as it executes 07 (explained in more detail below), any output that the program itself produces in WRITE and WRITEON statements 08, and perhaps a run error message saying why the program terminated 09. If the tracing were turned off, the output would look like that on page 118.

4.1.5. Statement Counts

This optional print-out consists of a pseudo-listing of the program 12 with coordinate numbers 10 and counts of how many times each statement was executed 11. To determine how many times a particular statement was executed, follow the vertical bars straight up and to the

left until a number is encountered. For example, the statement count for the IF at coordinate 0012 is found by following the bars up to coordinate 0005 , then up and left to the 6. on the preceding line; if this path goes through the statement where the program terminated prematurely [13] , then subtract one from the count. Thus, the IF statement at coordinate 0012 was executed 5 times (true 1 time, false 4 times). The pseudo-listing has all the comments removed and is formatted to show the block structure of the program. You are encouraged to make use of the statement counting facility in order to better understand just where your program is spending its time.

4.1.6. Post-Mortem Dump

This error analysis aid [14] shows the names and values of all variables which were active at the time the program stopped. By looking at the values of the variables used in the last statement executed [13] , it is easier to determine what (if anything) went wrong. The exact format of the dump is discussed below.

```
STANFORD ALGOL W (16JAN72)                                    19 JANUARY 1972  a  21:54        PAGE   1
 1    2    3                                                                  4
0001  1=      BEGIN
0001  --      COMMENT PROGRAM TO FIND AVERAGE OF GROUPS OF NUMBERS.  EACH GROUP
0001  --           ENDS WITH THE NUMBER -1;
0001  --
0001  --        INTEGER SUM,COUNT,NUMB;
0002  --        WHILE TRUE DO COMMENT LOOP UNTIL INPUT EXHAUSTED;
0002  2=          BEGIN                                                    6
0003  --          COMMENT THIS CARD HAS A SEQUENCE NUMBER FIELD ---------------------->;        ABCD1234****
0003  --  5       SUM := COUNT := 0;
0004  --          READON(NUMB); WRITEON(NUMB);
0006  --          WHILE NUMB=-1 DO
0006  3=            BEGIN
0007  --            SUM := SUM + NUMB;
0008  --            COUNT := COUNT + 1;
0009  --            READON(NUMB); WRITEON(NUMB)
0010  -3            END;
0011  --          IF COUNT=0 THEN WRITE("EMPTY GROUP") ELSE
0012  --          WRITE("COUNT ",COUNT,"SUM ",SUM,"AVERAGE ",SUM/COUNT);
0013  --          IOCONTROL(2)
0013  -2          END
0013  -1        END

6a EXECUTION OPTIONS: DEBUG,4(2) TIME=10 SECONDS PAGES=9
```



```
STANFORD ALGOL W (16JAN72)                COMPILATION DIAGNOSTICS        19 JANUARY 1972  a  21:54        PAGE   2

6b  ERROR 1017 NEAR COORDINATE 0003 - WARNING: SEQ FIELD CONTAINS TRASH

6c  000.14 SECONDS IN COMPILATION, (00528, 01336) BYTES OF CODE GENERATED
```

Sample Computer Output

```
=> TRACING (MAIN):
0002            1.--|       WHILE  TRUE  DO
                            * = TRUE;
0003            1.--|         SUM := COUNT := 0
                              COUNT := 0;  SUM := 0;
0004            1.--|         READON(NUMB)
INPUT RECORD: "1 2 3 -1                                                    29.    "
                              NUMB := 1;
0005            1.--|         WRITEON(NUMB)
                              NUMB = 1;

0006            1.--|         WHILE  NUMB ¬= -1  DO
                              NUMB = 1;  * = TRUE;
0007            1.--|       SUM := SUM + NUMB
                              SUM = 0;  NUMB = 1;  SUM := 1;
0008            1.--|       COUNT := COUNT + 1
                              COUNT = 0;  COUNT := 1;
0009            1.--|       READON(NUMB)
                              NUMB := 2;
0010            1.--|       WRITEON(NUMB)
                              NUMB = 2;
                (8)           2
0006             |        (WHILE  NUMB ¬= -1)
                            NUMB = 2;  * = TRUE;
0007            2.--|         SUM := SUM + NUMB
                              SUM = 1;  NUMB = 2;  SUM := 3;
0008            2.--|       COUNT := COUNT + 1
                              COUNT = 1;  COUNT := 2;
0009            2.--|       READON(NUMB)
                              NUMB := 3;
0010            2.--|       WRITEON(NUMB)
                              NUMB = 3;
                                          3
0006             |        (WHILE  NUMB ¬= -1)
                            NUMB = 3;  * = TRUE;  ...

                                                  -1

=> TRACING (MAIN):
0011            1.--|       IF  COUNT = 0  THEN
                              COUNT = 3;  * = FALSE;
0011            1.--|       WRITE("COUNT ", COUNT, "SUM ", SUM, "AVERAGE ", SUM/COUNT)
COUNT
                              COUNT = 3;
                (8)     3
                     SUM
                              SUM = 6;
                                    6
                                       AVERAGE
                              SUM = 6;  COUNT = 3;
                                                    2.00000000000000
0013            1.--|       IOCONTROL(2)

0002             |        (WHILE  TRUE)
                            * = TRUE;
0003            2.--|         SUM := COUNT := 0
                              COUNT := 0;  SUM := 0;
0004            2.--|         READON(NUMB)
INPUT RECORD: "57 32 24 88 1 0 2 -1                                        30.    "
                              NUMB := 57;
```

Execution Output for the Preceding Program

```
0005              2.--|       WRITEON(NUMB)
                               NUMB = 57;
            57
0006              2.--|       WHILE  NUMB ¬= -1  DO
                               NUMB = 57;  * = TRUE;  ...

                             32              24              ²8              1                    1

=> TRACING (MAIN):
0011              2.--|       IF  COUNT = 0  THEN
                               COUNT = 7;  * = FALSE;
0011              2.--|       WRITE("COUNT ", COUNT, "SUM ", SUM, "AVERAGE ", SUM/COUNT)
COUNT
                               COUNT = 7;
                     7
                         SUM
                               SUM = 204;
                                      204
                                    AVERAGE
                               SUM = 204;  COUNT = 7;
                                                    29.142857142857
0013              2.--|       IOCONTROL(2)

0002               |        (WHILE  TRUE)
                            * = TRUE;  ...

                  0 -          0              -1
COUNT                2  SUM              0  AVERAGE                    0
                  1

=> TRACING (MAIN):
0011              1.--|       WRITE("EMPTY GROUP")
EMPTY GROUP
                               ...


                  4            5              6              7              -1
COUNT                4  SUM             22  AVERAGE        5.500000000000000
```

RUN ERROR NEAR COORDINATE 0004, IN (MAIN) - READER EOF

(9)

000.15 SECONDS IN EXECUTION

Execution Output continued

```
=> EXECUTION FLOW SUMMARY
(10)      (11)      (12)
0000        1.--|       BEGIN
0001            .  |.    INTEGER SUM, COUNT, NUMB;
0002               |      WHILE  TRUE  DO
0002           6.--|          BEGIN  SUM := COUNT := 0;
--- ERROR ---------- ---------- ----------
0004          (13)  |          READON(NUMB);
--- ERROR ---------- ---------- ----------
0005                |          WRITEON(NUMB);
0006                |          WHILE  NUMB  <> -1  DO
0006             16.--|             BEGIN  SUM := SUM + NUMB;   COUNT := COUNT + 1;   READON(NUMB);   WRITEON(NUMB)
0010                |             END;
0011           5.--|          IF  COUNT  =  0  THEN
0011              1.--|          WRITE("EMPTY GROUP")  ELSE
0012              4.--|          WRITE("COUNT", COUNT, "SUM ",SUM,"AVERAGE", SUM/COUNT);
0013           5.--|          IOCONTROL(2)
0013                |          END
0013        0.--|       END

(14)=> TRACE OF ACTIVE SEGMENTS

=> SEGMENT NAME:  (MAIN)

  VALUES OF LOCAL VARIABLES:
    SUM = 0                        COUNT = 0                        NUMB = -1
```

```
                1                           3            -1
COUNT               3  SJY             6  AVERAGE     2.00000000000000
               57              32         24            88              1                        2           -1
COUNT               7  SUM           204  AVERAGE    29.1429571425571
                0               0         -1
COUNT               2  SUM             0  AVERAGE                    0
               -1
EMPTY GROUP
                4               5          6             7              1
COUNT               4  SUM            22  AVERAGE     5.50000000000000
--------------------------------------------------------------------------------------------------------------
RUN ERROR       NEAR COORDINATE 0004, IN(MAIN)   -READER E O F
                                                                 !
000.03SECONDSIN EXECUTION


=>TRACE OF ACTIVE SEGMENTS

=> SEGMENT NAME:   (MAIN)

  VALUES OF LOCAL VARIABLES:
    SUM = 0                      COUNT = 0                     NUMB = - 1
```

Output of the Preceding Program with no Tracing ($DEBUG,1)

4.2. Details of the Tracing Output

The tracing features of ALGOL W allow the programmer to watch the statement-by-statement execution of his program. The tracing output consists of four kinds of information for each statement:

a) The coordinate of the statement.

b) The number of times that statement has been executed.

c) The source statement itself.

d) A description of the values used in the statement.

There are special notations for procedure calls, for iterations and for showing data cards.

4.2.1. Basic Notations

For each value fetched during the execution of a statement, the fetch and store trace ($DEBUG,4) prints VARIABLE NAME = VALUE (6). The store trace only ($DEBUG,3) suppresses all of these fetch values. For each value stored (assigned), the tracing prints VARIABLENAME := VALUE (7). For each logical expression in an IF or WHILE statement the value of the expression is printed as * = TRUE or * = FALSE . If tracing is suspended because the next statement has already been executed m times (cf. Compiler Options for details of $DEBUG,n(m)) or because the TRACE function is used, then three dots are printed (1) (23). The second and subsequent times through a WHILE or FOR loop are indicated by the WHILE or FOR statement in parentheses (8) (9). Whenever a new card is needed by READ or READON, the complete card image is printed as INPUT RECORD: " 80 characters " 12 . Note that in general string values are printed with quotes on each end, but any quotes within

the string are not doubled. Reference values are printed as Recordclass .# , where # is a unique number (in order of allocation).

4.2.2. Procedure Call Notations

→ XYZ;	Indicates a call to procedure XYZ 13 .
⇒ TRACING XYZ;	Indicates that a new procedure is being traced 14 .
⟨PARAMETER ASSIGNMENT⟩	A dummy statement indicating whatever calculations must be performed in binding the actual parameters to the formal parameters 15 .
((PARAMETER IN xxx AT yyyy: trace))	If the actual parameter is an expression, then this notation gives the name of the calling routine, the coordinate of the call, and a trace of the expression evaluation 16 (21). Note that in the first example given, the expression MAKELONG(I) is actually another procedure call, whose tracing terminates about 25 lines later. There is a second example 24 on the next page.
FPARM :- APARM	Indicates the correspondence between the formal parameter and the actual parameter 17 .
FPARM' := value	In the case of VALUE and VALUE RESULT parameters, this indicates the value assigned to the local copy of the formal parameter 18 . The local copy is then used inside the procedure 19 .
#	Used as the name of an expression which otherwise has no name 22 .

XYZ(..) = value	Indicates the value returned from a function procedure 20 (25). This notation is preceded by a blank line to indicate a return to tracing the calling procedure.

```
=> TRACING (MAIN):
C031        1.--|  TRACE(3)
                   ...                    (1)

*    23     15     345    3 4 5
+    12     45      51    6 1
     45     23      22    2 2
-    45     29      16    1 6
     45     73     -28   -- 2 3
     55     41

=> TRACING LENGTH:
C245        1.--|      RM := COPY(N)
                        -> COPY;

=> TRACING COPY:
(2)C030     (3)1.--|   (4)<PARAMETER ASSIGNMENT>
(2)C031     (3)1.--|   (4)P := NIL; N' = RNODE.32;  IN' := RNODE.32;

                       (5) IN' = RNODE.32;  P := RNODE.32;
C032        1.--|      Q := NULL
                        Q := NULL;
C033        1.--|      WHILE P ¬= NULL  DO
                        P = RNODE.32;  * = TRUE;
C034           1.--|      Q := RNODE(Q, VAL(P))                    (6)
                           Q = NULL;  LINK(RNODE.36) := NULL; P = RNODE.32;  VAL(RNODE.32) = 5;
                           VAL(RNODE.36) := 5; Q := RNODE.36;
C035           1.--|      P := LINK(P)                             (7)
                           P = RNODE.32;  LINK(RNODE.32) = RNODE.33; P := RNODE.33;
C036        (8)    |      (WHILE P ¬= NULL)
                        P = RNODE.33;  * = TRUE;
C034           L.--|      Q := RNODE(Q, VAL(P))
                           Q = RNODE.36;  LINK(RNODE.37) := RNODE.36;  P = RNODE.33;  VAL(RNODE.33) = 5 ;
                           VAL(RNODE.37) := 5; Q := RNODE.37;
C035           2.--|      P := LINK(P)
                           P = RNODE.33;  LINK(RNODE.33) = NULL; P := NULL;
C033        (9)           (WHILE P ¬= NULL)
                        P = NULL; * = FALSE; (10)
C037        1.--|      REVERSE(Q)
                        -> REVERSE;
C038        1.--|      Q
                        Q = RNODE.36;

                        COPY(..) = RNODE.36; RM :=  RNODE.36;
C246        1.--|      Q := ZERO
                        -> ZERO;

=> TRACING ZERO:
C077        1.--|      RNODE(NULL, 0)
                        LINK(RNODE.38) := NULL;  VAL(RNODE.38) := 0;

                        ZERO = RNODE.38; Q := RNODE.36;
C247        1.--|      LN := LENGTH(N)
                        -> LENGTH;  LENGTH(..) = 2;  LN := 2;
C248        1.--|      LD := LENGTH(D)
                        -> LENGTH;  LENGTH(..) = 2;  LD := 2;
C249        1.--|      IF  LN < LD  THEN
                        LN = 2;  LD = 2;  * = FALSE; (11)
C250           1.--|     REVERSE(RM)
                         -> [illegible]
```

```
=> TRACING (MAIN):
C001        1.--|      INTFIELDSIZE := 3
C328        1.--|      BIGM := 10
                        BIGM := 10;
0329        1.--|      HALFM := 5
                        HALFM := 5;
0330        1.--|      WHILE  TRUE  DO
                        * = TRUE;
(12) 0331       1.--|     READDN(I, J)
INPUT RECORD: "99 999
                         I := 99;  J := 999;
C332        1.--|      R := LONGMPY(MAKELONG(I), MAKELONG(J))
                (13)     -> LONGMPY;

(14) => TRACING LONGMPY:
C194        1.--|  (15) <PARAMETER ASSIGNMENT>
                (16)  << PARAMETER IN (MAIN) AT 0332:  -> MAKELONG;

=> TRACING MAKELONG:
0014        1.--|      <PARAMETER ASSIGNMENT>          (18)
                (17)   INT :- I;  I = 99;  INT' := 99;
C016        1.--|      ANSWER := RNODE(NULL, INT REM BIGM)
                        LINK(RNODE.1) := NULL;  INT' = 99;  BIGM = 10;  VAL(RNODE.1) := 9;  ANSWER := RNODE.1;
C017        1.--|      R2 := ANSWER
                        ANSWER = RNODE.1;  R2 := RNODE.1;
C018        1.--|      INT2 := INT DIV BIGM
                (19)    INT' = 99;  BIGM = 10;  INT2 := 9;
0019        1.--|      WHILE  INT2 ¬= 0  DO
                        INT2 = 9;  * = TRUE;
0020           1.--|     R := RNODE(NULL, INT2 REM BIGM)
                          LINK(RNODE.2) := NULL;  INT2 = 9;  BIGM = 10;  VAL(RNODE.2) := 9;  R := RNODE.2;
0021           1.--|     ASSERT  LINK(R2) = NULL
                          R2 = RNODE.1;  LINK(RNODE.1) = NULL;
0022           1.--|     LINK(R2) := R
                          R2 = RNODE.1;  R = RNODE.2;  LINK(RNODE.1) := RNODE.2;
0023           1.--|     R2 := R
                          R = RNODE.2;  R2 := RNODE.2;
0024           1.--|     INT2 := INT2 DIV BIGM
                          INT2 = 9;  BIGM = 10;  INT2 := 0;
0019            |       (WHILE  INT2 ¬= 0)
                        INT2 = 0;  * = FALSE;
C026        1.--|      ANSWER
                (20)    ANSWER = RNODE.1;
                                                  (21)
                (22)    MAKELONG(..) = RNODE.1;  >>
                        N1 :- #;  # = RNODE.1;  N1' := RNODE.1;
                        << PARAMETER IN (MAIN) AT 0332:  -> MAKELONG;

=> TRACING MAKELONG:
C014        2.--|      <PARAMETER ASSIGNMENT>
                        INT :- J;  J = 999;  INT' := 999;
0016        2.--|      ANSWER := RNODE(NULL, INT REM BIGM)
                        LINK(RNODE.3) := NULL;  INT' = 999;  BIGM = 10;  VAL(RNODE.3) := 9;  ANSWER := RNODE.3;
0017        2.--|      R2 := ANSWER
                        ANSWER = RNODE.3;  R2 := RNODE.3;
C018        2.--       INT2 := INT DIV BIGM
                        INT' = 999;  BIGM = 10;  INT2 := 99;
C019        2.--|      WHILE  INT2 ¬= 0  DO
                        INT2 = 99;  * = TRUE;
0020           2.--|     R := RNODE(NULL, INT2 REM BIGM)
```

Tracing Output continued

```
                          LINK(RNODE.4) := NULL;  INT2 = 99;  BIGM = 10;  VAL(RNODE.4) := 9;  R := RNODE.4;
0021          2.--|       ASSERT  LINK(R2) = NULL
                          R2 = RNODE.3;  LINK(RNODE.3) = NULL;
0022          2.--|       LINK(R2) := R
                          R2 = RNODE.3;  R = RNODE.4;  LINK(RNODE.3) := RNODE.4;
0023          2.--|       R2 := R
                          R = RNODE.4;  R2 := RNODE.4;
0024          2.--|       INT2 := INT2 DIV BIGM
                          INT2 = 99;  BIGM = 10;  INT2 := 9;
0019           |          (WHILE  INT2 ¬= 0)
                          INT2 = 9;  * = TRUE;  ...  (23)

=> TRACING MAKELONG:
0026          2.--|       ANSWER
                          ANSWER = RNODE.3;

                          MAKELONG(..) = RNODE.3;  >>
                          N2 :- #;  * = RNODE.3;  N2' := RNODE.3;
0197          1.--|       P := N1
                          N1' = RNODE.1;  P := RNODE.1;
0198          1.--|       Q := N2
                          N2' = RNODE.3;  Q := RNODE.3;
0199          1.--|       R := RNODE(NULL, 0)
                          LINK(RNODE.6) := NULL;  VAL(RNODE.6) := 0;  R := RNODE.6;
0200          1.--|       ANSWER := R
                          R = RNODE.6;  ANSWER := RNODE.6;
0201          1.--|       RIGHTPARTIAL := R
                          R = RNODE.6;  RIGHTPARTIAL := RNODE.6;
0202          1.--|       IF  ((VAL(P) = 0)  AND  (LINK(P) = NULL))  OR  ((VAL(Q) = 0)  AND  (LINK(Q) = NULL))  THEN
                          P = RNODE.1;  VAL(RNODE.1) = 9;  Q = RNODE.3;  VAL(RNODE.3) = 9;  * = FALSE;
0202             1.--|      WHILE  P ¬= NULL  DO
                            P = RNODE.1;  * = TRUE;
0203                1.--|     IF  RIGHTPARTIAL = NULL  THEN
                              RIGHTPARTIAL = RNODE.6;  * = FALSE;
0207                1.--|     R := RIGHTPARTIAL
                              RIGHTPARTIAL = RNODE.6;  R := RNODE.6;
0208                1.--|     C := 0
                              C := 0;
0209                1.--|     Q := N2
                              N2' = RNODE.3;  Q := RNODE.3;
0210                1.--|     WHILE  Q ¬= NULL  DO
                              Q = RNODE.3;  * = TRUE;
0211                   1.--|    A := HIGH(VAL(P)*VAL(Q))
                                -> HIGH;

=> TRACING HIGH:
0075          1.--|       <PARAMETER ASSIGNMENT>
                    (24)  << PARAMETER IN LONGMPY AT 0211:  P = RNODE.1;  VAL(RNODE.1) = 9;  Q = RNODE.3;
                          VAL(RNODE.3) = 9;  >>
                          NUMB :- #;  * = 81;  NUMB' := 81;
0075          1.--        NUMB DIV BIGM
                          NUMB' = 81;  BIGM = 10;

                              (25)  HIGH(..) = 8;  A := 8;
0212                   1.           B := LOW(VAL(P)*VAL(Q))
                                    -> LOW;

=> TRACING LOW:
0073          1.--|       <PARAMETER ASSIGNMENT>
                          << PARAMETER IN LONGMPY AT 0212:  P = RNODE.1;  VAL(RNODE.1) = 9;  Q = RNODE.3;
```

4.3. Details of the Post-mortem Dump

The post-mortem dump begins with ⇒ TRACE OF ACTIVE SEGMENTS (1), then the complete call chain is printed starting with the procedure which was active at the point of termination and working back to its caller, etc. For each procedure, the following information is printed:

a) The name of the procedure (4). The outermost procedure is called "(MAIN)" and a simple BEGIN block is named "⟨BLOCK⟩".

b) The names and values of the local variables in the procedure (5). Uninitialized values print as "?" (7). Local copies of parameters are named with primes (6). Strings are printed with a single quote added on each end, but quotes within the string are--not doubled. At most eight values are printed from an array, usually the first seven and last one (8) (9). Reference values are printed as Recordclass .# , where # is a unique number (in order of allocation). The control variables in FOR statements are all distinct even if they are spelled the same way. so if I is used in many FOR statements, it will be dumped many times (11).

c) The name of the calling routine and the coordinate of the call (10). For NAME parameters, a procedure may be re-entered (environment re-established) to evaluate the corresponding argument (2) (3).

```
0292          16.--|      BEGIN  IN(I) := I;   W(I) := 1/(2*NN);   B(0, I) := 1.0;
0296               |      END;
0297          . . .|    LINPROG(NU + 1, 2*NN, NU + 1, B, BB, C, W, Z, IN, ERR);
0298               |    IF  ERR <> 0  THEN
0298           0.--|      WRITE("ERROR NO. ", ERR)  ELSE
0298           0.--|      FOR  I := 0  STEP  1  UNTIL  2*N - 1  DO
0299              0.--|      WRITE("INDEX ", IN(I), " VALUE ", W(I))
0298          |      END
```

(1) => TRACE OF ACTIVE SEGMENTS

=> SEGMENT NAME: AB

(2) AB WAS REENTERED FROM GMAT, NEAR COORDINATE 0072, TO ACCESS A PARAMETER

=> SEGMENT NAME: GMAT

VALUES OF LOCAL VARIABLES:
RI' = ? CI' = ?
GMAT WAS ACTIVATED FROM AB, NEAR COORDINATE 0242

=> SEGMENT NAME: AB

(3) AB WAS REENTERED FROM TRISOLV, NEAR COORDINATE 0033, TO ACCESS A PARAMETER

=> SEGMENT NAME: TRISOLV

VALUES OF LOCAL VARIABLES:

FIO' = 1	FIE' = -1	TT = ?	PV = ?
QWERTY = ?	I = ?		

TRISOLV WAS ACTIVATED FROM DECOMPOSE, NEAR COORDINATE 0081

=> SEGMENT NAME: DECOMPOSE

VALUES OF LOCAL VARIABLES:

BOTTOM' = 0	TOP' = 0	I = 0	J = ?
J = ?			

DECOMPOSE WAS ACTIVATED FROM AB, NEAR COORDINATE 0242

(4) => SEGMENT NAME: AB

(5) VALUES OF LOCAL VARIABLES:
D1 = ? D2 = ?
AB WAS ACTIVATED FROM LINPROG, NEAR COORDINATE 0249

=> SEGMENT NAME: LINPROG

VALUES OF LOCAL VARIABLES:

(6) M' = 8 N' = 18

Q(0) = ? (7)	Q(1) = ?	Q(2) = ?	Q(3) = ?
Q(4) = ?	Q(5) = ?	Q(6) = ?	(8) . . . Q(8) = ?
H(0) = ?	H(1) = ?	H(2) = ?	H(3) = ?
H(4) = ?	H(5) = ?	H(6) = ?	. . . H(8) = ?
W(0) = ?	W(1) = ?	W(2) = ?	W(3) = ?
W(4) = ?	W(5) = ?	W(6) = ?	. . . W(8) = ?
Y(0) = ?	Y(1) = ?	Y(2) = ?	Y(3) = ?
Y(4) = ?	Y(5) = ?	Y(6) = ?	. . . Y(8) = ?
V(0) = ?	V(1) = ?	V(2) = ?	V(3) = ?

```
V(4) = ?                V(5) = ?                V(6) = ?                ... V(8) = ?
P(3,0) = ?              P(1,0) = ?              P(2,0) = ?                  P(3,0) = ?
P(4,0) = ?              P(5,0) = ?              P(6,0) = ?              ... P(8,8) = ?
IX(0) = 0               IX(1) = 1               IX(2) = 2                   IX(3) = 3
IX(4) = 4               IX(5) = 5               IX(6) = 6               ... IX(26) = ?
RO(0) = 0               RO(1) = 1               RO(2) = 2                   RO(3) = 3
RO(4) = 4               RO(5) = 5               RO(6) = 6               ... RO(8) = 8
NU = 7                  NU = 17                 ALPHA = ?                   BETA = 0
GAMMA = ?               GM1 = ?                 IM1 = -1                    I = ?
J = 17                  K = ?                   L = 0                       T1 = ?
T2 = ?                  INFINITY = 7.237005*+75 PREVZ = -7.237005*+75       ETA = 9.536743*-07
```

(10) LINPROG WAS ACTIVATED FROM (MAIN), NEAR COORDINATE 0297

=> SEGMENT NAME: (MAIN)

VALUES OF LOCAL VARIABLES:

```
NN = 9                  N = 8                   NU = 7                      I = ?
J = ?                   M = 4                   ERR = 0                     NNM1 = 8
BB(0) = 1.000000        BB(1) = 0               BB(2) = 0                   BB(3) = 0
BB(4) = 0               BB(5) = 0               BB(6) = 0               ... BB(20) = ?
W(0) = 0.05555555       W(1) = 0.05555555       W(2) = 0.05555555           W(3) = .05555555
W(4) = 0.05555555       W(5) = 0.05555555       W(6) = 0.05555555       ... W(20) = 0?
C(0) = 0                C(1) = 0                C(2) = 3.051758*-05         C(3) = _3.051758*-05
C(4) = 0.0009765625     C(5) = 0.0009765625     C(6) = 0.007415771      ... C(20) = ?
PSI(0) = 0              PSI(1) = 0.1250000      PSI(2) = 0.2500000          PSI(3) = 0.3750000
PSI(4) = 0.5000000      PSI(5) = 0.6250000      PSI(6) = 0.7500000      ... PSI(20) = ?
B(0,0) = 1.000000       B(1,0) = 0.6666679      B(2,0) = 2.666665           B(3,0) = 0.6666665
B(4,0) = 0              B(5,0) = 0              B(6,0) = 0              ... B(10,20) = ?
U(-3) = -0.7500000      U(-2) = -0.5000000      U(-1) = -0.2500000          U(0) = 0
U(1) = 0.2500000        U(2) = 0.5000000        U(3) = 0.7500000        ... U(10) = ?
IN(0) = 0               IN(1) = 1               IN(2) = 2                   IN(3) = 3
IN(4) = 4               IN(5) = 5               IN(6) = 6               ... IN(20) = ?
Z = ?                   I = 7 *                 I = -3 *                    I = 8 *
I = 16 *                J = 7 *                 I = 17 *                    I = ?
```

(11)

* LAST VALUE OF CONTROL IDENTIFIER PRIOR TO NORMAL EXIT

Post-mortem Dump continued

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; f'or example, $\mathcal{A}$, 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 α is defined as $\beta|\mathcal{C}$, this means that a particular inscription is an α if it is a β or if it is a $\mathcal{C}$.

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

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

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

The notation $\overline{\alpha'\beta}$ means α or $\alpha\beta\alpha$ or $\alpha\beta\alpha\beta\alpha$, etc; it abbreviates $\alpha\{\beta\alpha\}^*$.

The notation $\alpha \perp \beta$ means α and/or β ; it abbreviates $\alpha|\beta|\alpha\beta$.

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	$\mathfrak{I}$	$\lambda\ \{\lambda\|\delta\}^*$
symbol	σ	Any symbol on the keypunch, except the double quote
constant	C	$\delta^+[.\delta^*]$| "σ^+"
function value	$\mathfrak{F}$	$\mathfrak{I}[\ (\overline{\mathcal{E}^+,})]$
expression	$\mathcal{E}$	C-1 $\{\mathfrak{I}\|C\|\mathfrak{F}\|(\mathcal{E})\}$**{*|/} {+|-} {<|<=|=|>=|>|¬=}
simple statement	S'	$\mathfrak{I}$:=$\mathcal{E}$|$\mathfrak{I}[(\overline{\mathcal{E}^+,})]$|GO TO $\mathfrak{I}$ |$\mathcal{B}$
statement	S	S'|IF $\mathcal{E}$ THEN S|IF $\mathcal{E}$ THEN S' ELSE S|FOR $\mathfrak{I}$:=$\mathcal{E}$ UNTIL $\mathcal{E}$ DO S
block	$\mathcal{B}$	BEGIN $\{\mathfrak{D};\}^*\{S;\|\mathfrak{I}:\}^*$S END
declaration	$\mathfrak{D}$	$\mathcal{T}\ \overline{\mathfrak{I}^+,}$|$\mathcal{T}$ PROCEDURE $\mathcal{N}$;{$\mathcal{E}$|BEGIN{$\mathfrak{D}$;}*{S;|$\mathfrak{I}$:}*$\mathcal{E}$ END)
type	$\mathcal{T}$	INTEGER|REAL|LOGICAL|STRING(C)
procedure heading	$\mathcal{N}$	$\mathfrak{I}(\mathcal{T}\{$VALUE|PROCEDURE$\}\overline{\mathfrak{I}^+,}$;)
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 express ion	ε''	[+|-][¬]{ABS|LONG|SHORT}*{v|c|$\mathcal{F}$|(ε)} {**|SHL|SHR}{*|/|DIV|REM|AND}{+|-|OR}
simple expression or relation	ε'	ε'' |$\varepsilon''\{<\|<=\|=\|>=\|>\|¬=\}\varepsilon''$ |ε'' IS ϑ
expression	ε	ε'|IF ε THEN ε ELSE ε|CASE ε OF $(\overline{\varepsilon^+,})$
argument	a	ε|s| $\vartheta[(\overline{\{\varepsilon\|*\}^+,})]$
simple statement	S'	$\{v:=\}^+\varepsilon$|GO TO ϑ|$\vartheta[(\overline{a^+,})]$|$\Lambda$|$\beta$
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 ε|{,$\varepsilon\}^*$}DO S

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

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

Use of Symbols

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

α_i = value of expression $\mathcal{E}_i$.

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

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. I.e., integers are converted to long real.

Format	Meaning	Kinds of Arguments and Results	Domains of Numeric Arguments and Results	Precision of Numeric Arguments and Results
$\varepsilon_1 + \varepsilon_2$	$\alpha_1 + \alpha_2$	N + N → N	$d_1+d_2 \rightarrow \max(d_1,d_2)$	$p_1+p_2 \rightarrow \min(p_1,p_2)$
$\varepsilon_1 - \varepsilon_2$	$\alpha_1 - \alpha_2$	N - N → N	$d_1-d_2 \rightarrow \max(d_1,d_2)$	$p_1-p_2 \rightarrow \min(p_1,p_2)$
$\varepsilon_1 * \varepsilon_2$	$\alpha_1 \times \alpha_2$	N * N → M	$d_1*d_2 \rightarrow \max(d_1,d_2)$	$p_1*p_2 \rightarrow L$
$\varepsilon_1 / \varepsilon_2$	α_1 / α_2	N/N → N	$d_1/d_2 \rightarrow \max(d_1,d_2,R)$	$p_1/p_2 \rightarrow \min(p_1,p_2)$
$\varepsilon_1 ** \varepsilon_2$	$\alpha_1^{\alpha_2}$	N**N → N	$d_1**I \rightarrow \max(d_1,R)$	$p_1**L \rightarrow p_1$
$+ \varepsilon_1$	α_1	+N → N	$+d_1 \rightarrow d_1$	$+p_1 \rightarrow p_1$
$- \varepsilon_1$	$-\alpha_1$	-N → N	$-d_1 \rightarrow d_1$	$-p_1 \rightarrow p_1$
ε_1 DIV ε_2	TRUNCATE(α_1/α_2)	I DIV I → I		
ε_1 REM ε_2	$\alpha_1 - (\alpha_1$ DIV $\alpha_2) * \alpha_2$, the remainder of ε_1 DIV ε_2	I REM I → I		
ABS ε_1	$\lvert\alpha_1\rvert$	ABS N → N	ABS $d_1 \rightarrow \min(d_1,R)$	ABS $p_1 \rightarrow p_1$
LONG ε_1	α_1	LONG N → N	LONG $d_1 \rightarrow \max(d_1,R)$	LONG $p_1 \rightarrow$ L where p_1=s or d_1=I
SHORT ε_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
$\mathcal{E}_1 \neg= \mathcal{E}_2$	$\alpha_1 \neq \alpha_2$	$k_1 \neq k_2$ → 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 ε_1 THEN ε_2 ELSE, ε_3	if α_1 then α_2, otherwise α_3	IF L THEN k_2 ELSE $k_3 \rightarrow k$ where $k_2 = k_3 = k$	IF L THEN 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 ε_0 of $(\varepsilon_1,\ldots,\varepsilon_n)$	α_{α_0} $(1 \leq \alpha_0 \leq n)$	CASE N OF $(k_1,k_2,\ldots,k_n)$ $\rightarrow$ k where $k_1=k_2= \ldots =k = k$	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 $F(\mathfrak{e}_1)$, 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 $\lvert\alpha_1\rvert - 1 < \lvert i\rvert \leq \lvert\alpha_1\rvert$	N → N	R → I	Any
ENTIER	The integer i such that $\alpha_1 - 1 < i \leq \alpha_1$			
ROUND	The integer i, with the same sign α_1, such that $\lvert\alpha_1\rvert - 1/2 < \lvert i\rvert \leq \lvert\alpha_1\rvert + 1/2$			
EXPONENT	The largest integer i such that $i \leq \log_{16}(\lvert\alpha_1\rvert) + 1$ or 0 if $\alpha_1 = 0$	N → N	R → I	Any
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 → C ($d_1 \subseteq$ R)	Any → S*

*Note: An 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	dl → R ($d_1 \subseteq R$)	Any → s*
EXP	e^{α_1}, for $\alpha_1 < 174.67$	N → N	dl → 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. (See page 71.)	s(1) → I		
CODE	The character for which α_1 is used as a code. (See page 71.)	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±12345678901234 5	N → S(20)	$d_1 \subseteq R$	Any
EASE16	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)		

See also pages 56-59 for READ, READON, READCARD, WRITE, WRITEON, IOCONTROL.
See also pages 64-66 for INTFIELDSIZE, MAXINTEGER, EPSILON, MAXREAL, PI.