SDS ALGOL 60 REFERENCE MANUAL

90 06 99C

November 1966

SCIENTIFIC DATA SYSTEMS/1649 Seventeenth Street/Santa Monica, California/UP 1-0960

©1965, 1966, Scientific Data Systems, Inc.

Printed in U.S.A.

REVISIONS

This publication, 90 06 99C, augments the previous edition of the SDS ALGOL 60 Reference Manual, 90 06 99B. The additional information is "Non-ALGOL Code", Appendix F.

RELATED PUBLICATIONS

Name of Document	Publication Number
SDS 910 Computer Reference Manual	90 00 08
SDS 920 Computer Reference Manual	90 00 09
SDS 925 Computer Reference Manual	90 00 99
SDS 930 Computer Reference Manual	90 00 64
SDS 9300 Computer Reference Manual	90 00 50
SDS MONARCH Reference Manual	90 05 66
900 Series ALGOL 60-4 Operating Procedures	012029
900 Series ALGOL 60-8 Operating Procedures	042035
9300 ALGOL 60-8 Operating Procedures	642029

CONTENTS

INTRODUCTION 1

SECTION I 2

1. PROGRAM FORM 2
   - 1.1 Lexicon Words 2
   - 1.2 Comments 2
   - 1.3 Declarations 2
   - 1.4 Statements 2
2. DECLARATIONS AND REFERENCES 2
   - 2.1 Type, Value, and Definition 2
   - 2.2 Constants 3
   - 2.3 Strings 3
   - 2.4 Variables 3
   - 2.5 Arrays 3
   - 2.6 Labels 4
   - 2.7 Switches 4
   - 2.8 Procedures 4
3. EXPRESSIONS 5
   - 3.1 Arithmetic Expressions 5
   - 3.2 Boolean Expressions 6
   - 3.3 Designational Expressions 6
4. STATEMENTS 7
   - 4.1 Arithmetic Assignment Statements 7
   - 4.2 Boolean Assignment Statements 7
   - 4.3 GO TO Statements 8
   - 4.4 Procedure Statements 8
   - 4.5 Format Statements 9
   - 4.6 IF Statements 9
   - 4.7 Conditional Statements 9
   - 4.8 FOR Statements 10
   - 4.9 Compound Statements 10
   - 4.10 Blocks 11
   - 4.11 Dummy Statements 11
5. COMPILATIONS 11
6. SAMPLE PROGRAM 12

SECTION II 13

1. STRUCTURE OF THE LANGUAGE 13
2. BASIC CONCEPTS; SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS 13
   - 2.1 Letters and Digits 13
   - 2.2 Logical Values 13
   - 2.3 Delimiters 13
   - 2.4 Identifiers 14
   - 2.5 Numbers 14
   - 2.6 Strings 14
   - 2.7 Quantities, Kinds and Scopes 14
   - 2.8 Values and Types 15
3. EXPRESSIONS 15
   - 3.1 Variables 15
   - 3.2 Function Designators 15
   - 3.3 Arithmetic Expressions 16
   - 3.4 Boolean Expressions 18
   - 3.5 Designational Expressions 19
4. STATEMENTS 19
   - 4.1 Compound Statements and Blocks 19
   - 4.2 Assignment Statements 20
   - 4.3 GO TO Statements 20
   - 4.4 Dummy Statements 21
   - 4.5 Conditional Statements 21
   - 4.6 FOR Statements 22
   - 4.7 Procedure Statements 23
   - 4.8 Format Statements 24
5. DECLARATIONS 27
   - 5.1 Type Declarations 27
   - 5.2 Array Declarations 28
   - 5.3 Switch Declarations 29
   - 5.4 Procedure Declarations 29
   - 5.5 External Procedure Declarations 31

APPENDIXES

- A. DELIMITER CHARACTER SET 32
- B. RESERVED LEXICON WORDS 33
- C. STANDARD FUNCTIONS AND PROCEDURES 33
- D. NOTES TO USERS OF THE ALGOL 60 COMPILERS 34
- E. NOTES TO USERS OF THE ALGOL 60-4 EXECUTORS 34
- F. NON-ALGOL CODE 35

INDEX 37

INTRODUCTION

The SDS ALGOL 60 Compiler is a comprehensive implementation of ALGOL 60, the international algorithmic language. The source language of SDS ALGOL is essentially that specified in the "Revised Report on the Algorithmic Language ALGOL 60", Communications of the ACM, Volume 6, No. 1, January 1963.

SDS ALGOL 60 includes facilities for the following exceptions to the revised report:

FORTRAN-like Input/Output. FORTRAN input/output logic has been adopted and implemented, maintaining the FORMAT declaration of FORTRAN.

Separate Compilation of Procedures. Procedures can be compiled separately, thereby reducing concern for adequate core requirements in program compilations.

Initialization of OWN Variables and Arrays. The ALGOL 60 revised report does not explicitly provide for defining OWN variables or arrays upon initial entry into the block in which they reside. Distinguishing between initial entry and subsequent reentries to the block in which the OWN declaration resides is required. The SDS ALGOL OWN initialization feature permits variables and arrays to be defined with initial values, removing the need to distinguish between initial entry and reentry.

This document is not a primer on ALGOL 60; it is intended for programmers having previous experience with algebraic compilers. However, Section I of this manual contains a brief, general introduction to the ALGOL language and should be studied by those unfamiliar with ALGOL. Section II consists of a precise, technical description of the compiler language; it is directed to experienced ALGOL programmers.

Two SDS ALGOL 60 systems are available — a basic system and an expanded system. The basic system requires one pass that generates code in identical format to that of the expanded system; the expanded system uses magnetic tape for intermediate storage and requires two passes. During the first pass, the source program is encoded into concise intermediate language. As a result, processing time is considerably reduced for the second pass, which is essentially I/O limited.

The basic computer configuration required for ALGOL 60 is an SDS 900 Series Computer with 4K memory (see appendixes of this manual and the 900 Series ALGOL 60-4 Operating Procedures, Catalog No. 012029). The expanded system is either an SDS 900 Series or a 9300 Computer with 8K memory and two magnetic tape units (see either 900 Series ALGOL 60-8 Operating Procedures, Catalog No. 042035, or 9300 ALGOL 60-8 Operating Procedures, Catalog No. 642029).

This manual describes the ALGOL 60-4 and ALGOL 60-8 compiler languages for both systems. The systems include a compiler, loader/executor, and a library consisting of common input/output and mathematical function subroutines. The systems are compatible; most programs may be compiled by either compiler and executed by either executor on the 900 Series and 9300 Computers, except for programs using features available only in the expanded system. The information in the body of this manual applies to the ALGOL 60-8 compiler and executor. The different characteristics of the basic ALGOL 60-4 compiler and executor are described in the appendixes. Interface is provided for programs written and assembled with SYMBOL and META-SYMBOL.

The form of Section II is parallel to that of the previously mentioned ACM revised report. A simple meta-language parallel to the Backus notation in that report is used here to describe the syntax. In this meta-language, uppercase letters, numbers, and most special characters denote themselves. Strings of lower-case letters (possibly separated by hyphens) represent meta-linguistic variables. The first equal sign divides the meta-linguistic variable being described from its description. A vertical line is used to separate entities.

For example:

parameter-delimiter = , |) letter-string:(

may be read, "a parameter delimiter is either a comma or else a right parenthesis followed by a letter string followed by a colon and a left parenthesis."

Examples are given exactly as they would be typed for input to one of the compilers, except that the space character, ordinarily insignificant in both ALGOL and the meta-language, is represented by ⊔.

In case of discrepancies or contradictory statements between Sections I and II of this manual, the reader should accept the descriptions in Section II as valid. Restrictions that affect the writing of ALGOL statements appear in the text. Restrictions concerning table sizes, etc., which affect programs as a whole, appear in the applicable operating procedures.

SECTION I

1. PROGRAM FORM

An ALGOL program consists of a string of letters, numbers, blanks, and special characters arranged in valid comments, declarations, and statements on the input medium. Blanks are generally ignored by the compiler as are carriage return characters, transitions from one card to another, etc. Thus one line of coding may contain many ALGOL statements, or, conversely, one statement may be spread over several lines. This makes it possible to group the characters in easily readable form.

1.1 Lexicon Words

Certain words, like DO and GO TO, have special meanings to the compiler when surrounded by single quotation marks: 'DO', 'GO TO'. These are called lexicon words when within quotation marks; they may be used freely outside quotes without ambiguity.

1.2 Comments

Comments do not change the meaning of the program; they are for the benefit of the human reader only.

Comments may appear in two contexts:

The lexicon word 'COMMENT', followed by any string of characters ending with a semicolon, may appear anywhere in a program except within lexicon words, identifiers, or compound special characters.

The lexicon word 'END', when it appears, may be followed by any string of characters other than apostrophes ending with a semicolon, another 'END' or the lexicon word 'ELSE'.

'COMMENT' BESSELFUNCTION.J.SMITH.10-12-64;

'END' BESSEL FUNCTION

1.3 Declarations

Declarations inform the compiler of the characteristics of the data needed by the program. Some declarations produce code; others produce data words; others produce no output.

Declarations are fully described under the heading "Declarations and References".

1.4 Statements

Statements describe the computations which the program is to perform on the data. Most statements produce computer instructions.

There are several kinds of statements: basic statements; compound statements, containing one or more basic statements; and blocks, which contain statements and declarations of the data to which they refer.

The execution of an ALGOL program consists in executing its statements in order of appearance, except where ordering is changed by the statements themselves.

Statements are classified functionally and are more fully described in paragraph 4.

2. DECLARATIONS AND REFERENCES

Statements may refer to the following kinds of entities:

constants	(constant values)
strings	(groups of alphanumeric characters)
variables	(variable values)
arrays	(ordered sets of variable values)
labels	(names of single statements)
switches	(ordered sets of labels)
procedures	(names of groups of statements)

Except for numbers and strings, each entity is identified by an identifier, consisting of a letter followed optionally by letters or numbers:

A
B12A
POINT TWO

Blanks may appear within identifiers, as elsewhere, without affecting the meaning.

POINT TWO = POINTTWO = P O I N T T W O

However, to refer to the same entity, the same characters must be written in the same order.

END ≠ EDN ≠ DEN ≠ END1 etc.

The same identifier may refer to different entities, even different kinds of entities, at different places in the program, at the discretion of the programmer. See "Blocks," paragraph 4.10.

2.1 Type, Value, and Definition

There are three types of data in ALGOL: BOOLEAN, INTEGER, and REAL. At any time in the running of a program, every variable and each element of every array contains a value of one of these types or is undefined.

The value of an entity may change during the running of a program, but its type cannot change.

The value of a variable may be defined by any of the following means:

1. Input, reading a value from cards, etc.
2. OWN initialization.
3. Assignment statements.
4. FOR statements.

Until a variable is defined by one of these means, it is undefined and computations involving it give unpredictable results.

2.2 Constants

Constants of all three types are written explicitly within statements and declarations. Since their types can be determined by inspection, they do not need declarations.

Boolean constants consist of the lexicon words 'TRUE' and 'FALSE'. Boolean calculations may produce only these two values.

Integer constants consist of whole numbers less than 32768:

```
3
32767
0
```

Integer calculations, however, may produce numbers up to 2,097,151 without exceeding the range of the ALGOL system.

Real constants consist of numbers of the following form (where X and Y represent whole numbers and Z represents an optionally signed whole number):

Form	Example	Value
X.Y	1.5	1.5
X.Y△Z	0.637485△1	6.37485
.Y	.17	0.17
.Y△Z	.5△-5	0.000005
X △Z	4△+9	4000000000.0
△Z	△2	100.0

From these examples it is clear that the value of a real constant is X.Y times 10^Z. When X, Y or Z is absent, zero is assumed; when both X and Y are absent, 1.0 is assumed. X and Y are less than 8,388,608; Z is less than 256 in absolute value.

Real calculations, however, may produce numbers up to 10^{160} without exceeding the range of the ALGOL system. Real calculations do not always yield exact results and it is advisable not to rely on them. However, approximately 12 decimal digits of accuracy are retained.

2.3 Strings

Strings are alphanumeric constants and, like other constants, are declared by being used.

```
''THIS IS A STRING''
```

Strings are begun by two single quotation marks and ended by the next pair of single quotation marks. They may contain any character except the end-of-program character ‡

Blanks are significant in strings, and only in strings.

Strings may be used only in format statements and as arguments of procedures, both described further on.

2.4 Variables

Variables are declared by means of the following declarations:

```
'INTEGER' I
'REAL' A1, A2, END
'BOOLEAN' SWITCH, MARRIED
```

That is, the type is given by a lexicon word; and one or more variables, separated by commas, may be declared in a single declaration.

All variables must be declared except procedure arguments (paragraphs 2.8 and 4.4).

'OWN' variables are declared as follows:

```
'OWN' 'INTEGER' I
'OWN' 'REAL' A1, A2, END
'OWN' 'BOOLEAN' SWITCH, MARRIED
```

The unique properties of OWN variables are discussed in paragraph 4.10.1.

OWN variables may be preset to specific values as follows:

```
'OWN' 'INTEGER' I := 13
'OWN' 'REAL' A1 := 0.9, A2, END := -7△-3
'OWN' 'BOOLEAN' SWITCH, MARRIED := 'TRUE'
```

A2 and SWITCH, which are not explicitly initialized, are preset to .0 and 'FALSE', respectively.

Note that the constant value must be of the same type as the variable to which it corresponds.

Variables are referenced by writing their identifiers. The value of a variable reference is the last value assigned it by the running program.

2.5 Arrays

Arrays are declared by means of the following declarations:

```
'INTEGER' 'ARRAY' BIG (0:999)
'REAL' 'ARRAY' SMALL (-1:1, -1:1)
'BOOLEAN' 'ARRAY' B,C (0:48), D (0:72)
```

The numbers in parentheses are called bound pairs; they specify the number of dimensions of the array and the minimum and maximum subscripts for each dimension position. Each bound may be any arithmetic expression (paragraph 3.1). That is, BIG is used to refer to 1000

integer variables which may be referred to individually as BIG (0), BIG (1), etc.

The numbers in parentheses are subscripts. Subscripts, too, may be any arithmetic expression. Real subscripts are rounded to the nearest integer value before use. SMALL contains 9 real variables; they are referred to as SMALL (n1, n2), where n1 and n2 stand for arithmetic expressions whose values are between -1 and 1 inclusive.

Arrays are referenced by writing the array identifier. References to full arrays may appear only as arguments to procedures. In all other situations a single element is referenced, and the number of subscripts written must be the same as the number of bound pairs declared.

The word 'REAL' is optional when declaring a real array. A list of arrays may be given in a single declaration, as shown in the third example. This example declares B and C to run from 0 through 48, and D to run from 0 through 72. Own arrays are declared by prefixing the declarations like those above with the word 'OWN'.

```
'OWN' 'ARRAY' A (0:1,2:4) := ,,, .5 ,, .4
```

declares A an OWN real array and presets it as follows:

```
A(0,2) 0.0
A(1,2) 0.0
A(0,3) 0.0
A(1,3)  .5
A(0,4)  .5
A(1,4)  .4
```

Note that elements are initialized in a particular order (with the leading subscripts rising fastest), and that unspecified elements are set to the previous value.

2.6 Labels

Labels may be used to name statements. A label is declared by writing its name followed by a colon in front of the statement it names. A statement may have more than one label.

```
L: (statement)
LABEL1: XYZ: HERE: (statement)
```

Labels are referenced by writing their names.

2.7 Switches

Switches are declared by means of switch declarations:

```
'SWITCH' PICK1 := L1, L2, L3
```

Here the name of the switch (PICK1) is associated with one or more labels (L1, L2, and L3) separated by commas. PICK1 may now be referenced by writing

```
PICK1 (1), PICK1 (2), etc.,
```

where the subscript is any arithmetic expression whose value is between 1 and the number of labels in the declaration, inclusive. In this way PICK1 (1) is associated with L1, PICK1 (2) with L2, etc.; and a reference to PICK1 (1) is said to have the value L1. Actually the positions of a switch may themselves be switch references or any other designational expression (paragraph 3.3).

2.8 Procedures

Procedure declarations in ALGOL are very much like subroutines in machine language code, and they exist for the same reasons:

1. To localize the code which performs a certain calculation.
2. To state only one time the code which is to be executed more than once.
3. To permit clear, explicit substitution of variable arguments in an invariable calculation.

Like a subroutine, a procedure is written outside the main body of the code by which it is called, in this case in a declaration.

Like a subroutine, it is entered and executed whenever the running program encounters a call to it; and it normally, but not always, returns to the running program just after the point of call.

The more complicated forms of procedure declarations and references are discussed under "Procedure Statements", paragraph 4.4.

Here is a simple example.

Declaration: 'REAL' 'PROCEDURE' ARCTAN (X, Y); statement

References: ARCTAN (Q, ROOT)... ARCTAN (PI,2.0)

The declaration contains:

- a type word ('INTEGER', 'REAL', or 'BOOLEAN')
- the word 'PROCEDURE'
- the name of the procedure (ARCTAN)
- a list of formal parameters in parentheses (X and Y)
- a semicolon
- a statement

The type word 'REAL' indicates that this procedure calculates a value of type real and returns the value to the point of the call upon it. The word 'PROCEDURE' specifies the kind of entity being declared and tells the compiler that the identifier which follows will be used to call on the procedure.

Because the actual parameters supplied to the procedure usually vary from one call to the next, a list of formal parameters is supplied. Then, in the "statement" part of the procedure, where the formal parameter is referenced, the corresponding actual parameter of the current call is substituted. That is, when ARCTAN (Q, ROOT) is executed, all references to X actually refer to Q; when ARCTAN (PI, 2.0) is executed, they refer to PI. Y refers to ROOT during the first call and to 2.0 during the second. The actual parameters can be quite complicated, but the formal parameters are simply identifiers.

Procedures which do not require actual parameters do not declare formal parameters:

```
'INTEGER' 'PROCEDURE' GET NEXT PRIME;
                statement
```

Some procedures in the ALGOL library may be referenced without the programmer's declaring them. That is, their declarations are built into the compiler.

Other procedure declarations may be compiled separately from the sections of the program which reference them. In this case, the referencing programs must contain 'EXTERNAL' declarations:

```
'REAL' 'EXTERNAL' ARCTAN
'INTEGER' 'EXTERNAL' GET NEXT PRIME
```

Here only the type, the word 'EXTERNAL', and the procedure identifier are written.

References to procedures declared internally and those declared externally are identical in form.

Procedures for which a type is given are called functions and the references to them are called function designators. The value of a function designator is computed by the procedure based on the current values of the variables in the program.

3. EXPRESSIONS

An expression is a rule for computing a single value. ALGOL permits three kinds of expressions:

Symbol	Kind of Expression	Value
A	arithmetic	a real or integer number
B	Boolean	true or false
C	designational	a label

The contexts in which expressions A, B and C may appear will be described further under "Statements", paragraph 4. For the moment, let us concern ourselves with the way in which the entities already described combine with each other, and with expressions to form expressions.

3.1 Arithmetic Expressions

An arithmetic expression is a rule for computing a value of type 'INTEGER' or of type 'REAL'. Wherever the word "arithmetic" appears, the phrase "INTEGER or REAL" may be substituted.

Boolean entities and entities without type may appear in arithmetic expressions, but not as elements of them. This distinction may become clear as we consider the permissible elements of arithmetic expressions.

3.1.1 Arithmetic Elements

1. Arithmetic constants.
2. Arithmetic variable references.
3. Arithmetic array references.
4. Arithmetic function designators.
5. ('IF' B 'THEN' A1 'ELSE' A2)
   Note: The parentheses are unnecessary if this element stands alone.
6. (A), that is, an arithmetic expression in parentheses.

The values of the first four elements have already been discussed.

The value of 'IF' B 'THEN' A1 'ELSE' A2 is the value of A1 if B is true, the value of A2 if B is false.

The value of (A) is the value of A. The parentheses are used to specify grouping to the user and to the compiler.

Any of the above elements, standing alone, constitutes an arithmetic expression.

Expression	Value
2	2
X	last value assigned X
Y(3, Z)	last value assigned the corresponding array element
MOD(Y(3, Z), Y)	resulting value
'IF' 'TRUE' 'THEN' 10 'ELSE' -10	10
((99))	99

3.1.2 Arithmetic Operators

The elements may be combined by the following arithmetic operators:

Operator	Meaning	Type of Result
+	addition	note 1
-	subtraction	note 1
*	multiplication	note 1
/	division	note 2
\	integer division	note 3
**	exponentiation	note 4

Note 1: integer if both elements are integers, otherwise real.

Note 2: real.

Note 3: integer (both elements are of type integer).

Note 4: Depends on current values of operands (see Section II, paragraph 3.3.4.3).

An expression may be preceded by a plus or a minus sign, in which case a leading zero of the appropriate type is implied.

Examples	Values
2+3	5
-2+3	1
4.2-3	1.2
18*.0005	.0090
7/3	2.333...
7\3	2
8\3	3
9.0\3	illegal
2**3	8
0**0	undefined
2**-3	0.125
-3.14159**0	undefined
16**2△0	256.0
0**.1	0.0
0**-.1	undefined

Where more than one arithmetic operator appears in an expression, exponentiation takes precedence over the multiplicative operators (*, /, and \), and multiplicative operators take precedence over additive operators (+ and -). When two operators of the same precedence are adjacent, they are computed from left to right.

Expressions within parentheses have the highest precedence, and parentheses may be used to specify any desired grouping when grouping is important.

Examples	Grouping
A+B*C	A+(B*C)
-A**B/C	0-((A**B)/C)
A\B/C**D	((A\B)/C)**D

These rules permit the reader to construct and comprehend arithmetic expressions as complicated as those in Section II, paragraph 3.3.2.

3.2 Boolean Expressions

3.2.1 Boolean Elements

1. Boolean constants.
2. Boolean variable references.
3. Boolean array references.
4. Boolean function designators.
5. ('IF' B1 'THEN' B2 'ELSE' B3).
   Note: The parentheses are unnecessary if this element stands alone.
6. (B).
7. Relation.

A relation is one of the following:

A1=A2	equal
A1><A2, A1<>A2	not equal
A1<A2	less than
A1>A2	greater than
A1<=A2, A1=<A2	not greater than
A1>=A2, A1=>A2	not less than

That is, it is a comparison between two arithmetic expressions, yielding the result 'TRUE' if the comparison is satisfied, and 'FALSE' if it is not.

3.2.2 Boolean Operators

The Boolean operators are listed below in order of precedence.

Operator	Meaning	Type of Result
'NOT'	negation	Boolean
'AND'	logical intersection	Boolean
'OR'	logical union	Boolean
>>	implication	Boolean
==	equivalence	Boolean
Relations	as described	Boolean

Definitions of the operators appear in Section II, paragraph 3.4.5.

Examples	Grouping
'NOT' A 'AND' B	('NOT' A) 'AND' B
A 'OR' B 'AND' C	A 'OR' (B 'AND' C)
A>>B==C>>D	(A>>B)==(C>>D)

3.3 Designational Expressions

A designational expression is a rule for selecting (not computing) a label.

The following elements may be combined by designational expressions:

1. Labels.
2. Switch references.
3. ('IF' B 'THEN' C1 'ELSE' C2)
   Note: the parentheses are unnecessary if this element stands alone.

Any of these elements by itself consitutes a designational expression.

Examples	Value
P9	P9
CHOOSE (N -1)	n -1st position of switch CHOOSE
'IF' 3>2 'THEN' L1 'ELSE' L2	L1

4. STATEMENTS

Using the symbols A, B, and C as already defined, D to represent a declaration, and S as a statement (not yet fully defined), the following statements are available in ALGOL:

Kind of Statement	Form
1. Arithmetic assignment statement	arithmetic list: = A
2. Boolean assignment statement	Boolean list: = B
3. GO TO statement	'GO TO' C
4. Procedure statement	procedure-name actual-parameter-list
5. Format statement	'FORMAT' string
6. Dummy statement	(empty)
7. IF statement	'IF' B 'THEN' S
8. Conditional statement	'IF' B 'THEN' S 'ELSE' S
9. FOR statement	'FOR' arithmetic-variable:=for-list 'DO' S
10. Compound statement	'BEGIN' S;S...;S 'END'
11. Block	'BEGIN' D;D;...;D; S;S;...;S 'END'

Statements will now be more fully described individually. However, this list should give an idea of the general structure and facilities of ALGOL.

4.1 Arithmetic Assignment Statements

form: arithmetic list := A

Probably the most useful statement in ALGOL is the arithmetic assignment statement, which calls for the evaluation of an arithmetic expression and the subsequent assignment of the resulting value to one or more arithmetic variables or array elements. The operator :=, which resembles an arrow pointing left is sometimes called the replacement operator. It separates each variable to be assigned from the rest of the statement.

The entities to be assigned must all be of the same type. If they are of type integer and the expression is real, the expression is rounded to the nearest integer before being stored.

Example	Meaning
I := 2	assign 2 to I
I := 2.4	if I is real, assign 2.4 to I; if I is integer, assign 2 to I
V(3, 4) := J := 0	assign 0 to V(3, 4) and to J
A := B := 'IF' Q<0 'THEN' X**2 'ELSE' A+1	if Q<0, assign X^2 to A and B; if not, assign A+1 to A and B

It is obvious from the last example that an assignment statement is an algorithm, not an equation. That is, A = A+1 is always false, but A := A+1 is valid as the statement increments A.

4.2 Boolean Assignment Statements

form: Boolean list := B

This statement is identical in form to the arithmetic assignment statement, except that the entities assigned and the expression evaluated are of type Boolean.

Example	Meaning
A := B := 'TRUE'	assign 'TRUE' to A and B
CHANGE := 'NOT' CHANGE	reverse the value of CHANGE
TWO(I*J) := X = 4	if X = 4, assign 'TRUE' to TWO(I*J); if not, assign 'FALSE' to TWO(I*J)

4.3 GO TO Statements

form: 'GO TO' C

The GO TO statement is analogous to a branch in machine language code. It interrupts the order of execution of the statements. The designational expression C is evaluated, yielding a label as a value, and program execution continues with the statement following that label.

Examples	Meaning
'GO TO' ((L))	continue at statement following L
'SWITCH' S := AB, BC ...	
'SWITCH' T := CD, S ('IF' W<0 'THEN' 1 'ELSE' 2) ...	
'GO TO' T (N)	continue at CD if N=1; continue at AB if N≠1, W<0; continue at BC if N≠1, W≥0; continue with next statement if N<1 or N>2

Note: When the designational expression following 'GO TO' is a switch reference whose subscript is out of range of the corresponding switch, no jump takes place.

4.4 Procedure Statements

form: procedure-name actual-parameter-list

We began the discussion of procedure declarations and references under the heading "Declarations and References." There we mentioned function designators only. Function designators are elements of arithmetic and of Boolean expressions of the form

procedure-name actual-parameter-list

Each function designator has a value, like any other arithmetic or Boolean expression, the type of the value being part of the declaration. The value is obtained by executing an assignment statement within the procedure with the procedure name on the left of the replacement operator.

'REAL' 'PROCEDURE' RANDOM;... RANDOM:=**2...

Procedure statements have the same form as function designators, but they do not take part in expressions nor have associated values. For example:

MATRIX MULTIPLY (A, B, C)

Here the procedure has access to arrays A, B, and C and it may alter them or any other variable to which it has access, but no value is returned by the procedure statement.

Neither does the corresponding declaration state a type.

'PROCEDURE' MATRIX MULTIPLY (X, Y, Z);...

If a procedure declaration has a type, it is referenced only by function designators; if not, only by procedure statements.

Unless otherwise stated, the rest of the discussion pertains to declarations/references of either kind.

4.4.1 Calls by Name and Calls by Value

The number of actual parameters in each reference to a procedure must equal the number of formal parameters in a declaration and must be in the same order. During execution of the procedure, the current actual parameters are substituted for the corresponding formal parameters. Two distinct types of substitution are permitted in ALGOL: substitution of the current value of the parameter and substitution of the name of the parameter. The difference may be clarified by a simple example.

Given the declaration

'PROCEDURE' P(X, Y);... Y := 3

we may expect the execution of the statement

P(Z, W)

to set W to 3. It does, if Y is called by name. If Y is called by value, however, the effect is to declare a variable Y inside the procedure, and it is that variable which changes.

Parameters which are to be called by value are indicated by listing them after the word 'VALUE' at the beginning of the procedure declaration.

'PROCEDURE' P(X, Y); 'VALUE' Y, X;... Y := 3

In most cases it does not matter whether the call is by name or by value. Generally, it is only necessary to call by value when one wants to avoid changing the value of the actual parameter.

4.4.2 Specification of Formal Parameters

Since ALGOL permits actual parameters to be any kind of entity in the language, it is clear that there must be some correspondence in kind between actual parameters and formal parameters. It would not make sense, for example, to supply a constant as an actual parameter and

to refer to the corresponding formal parameter as though it were a switch. For this reason, the kinds (and sometimes types) of the formal parameters are specified at the beginning of the procedure. One specification appears for each parameter, with the possible exceptions of arithmetic expressions and strings.

Specifications appear between the 'VALUE' part of the procedure declaration (if any) and the statement which constitutes the procedure body, and are followed by semicolons.

'PROCEDURE' P(X,Y);'VALUE' Y,X;'REAL' X,Y;...

The following specifications are sufficient in specifying parameters called by name:

Specification	Kind of Actual Parameter
'REAL' or 'INTEGER'	arithmetic expression (specification optional)
'BOOLEAN'	Boolean expression
'STRING'	string (specification optional)
'ARRAY'	arithmetic array name
'LABEL'	designational expression
'SWITCH'	switch name
'INTEGER' 'PROCEDURE'	arithmetic procedure name*
'REAL' 'PROCEDURE'	arithmetic procedure name*
'BOOLEAN' 'PROCEDURE'	Boolean procedure name
'PROCEDURE'	untyped procedure name

Programs may assign values to formal parameters only when the corresponding actual parameters are variables or array elements.

The following specifications apply to parameters called by value:

Specification	Kind of Actual Parameter
'INTEGER'	arithmetic expression
'REAL'	arithmetic expression
'BOOLEAN'	Boolean expression
'INTEGER' 'ARRAY'	arithmetic array
'REAL' 'ARRAY'	arithmetic array
'ARRAY'**	arithmetic array
'BOOLEAN' 'ARRAY'	Boolean array

*The type of the actual parameter determines the type of arithmetic performed.

**'ARRAY' implies 'REAL' 'ARRAY'.

If the actual and formal parameters are arithmetic but of different types, the type of the formal parameter determines the type of arithmetic performed within the procedure.

4.5 Format Statements

form: 'FORMAT' string

Executing a format statement does not affect the program. The format statement is intended for reference by a call on the INPUT or the OUTPUT procedure. Formats and their effects are discussed fully in Section II, paragraph 4.8.1.

4.6 IF Statements

form: 'IF' B 'THEN' S

The Boolean expression B is evaluated; if it is true, the statement S is executed, otherwise it is not.

To avoid ambiguity, the statement S may not itself begine with the word 'IF'.

Examples:

```
'IF' DELTA<EPSILON 'THEN' 'GO TO' END
'IF' B1 'AND' B2 'THEN' B3 := 'FALSE'
```

4.7 Conditional Statements

form: 'IF' B 'THEN' S1 'ELSE' S2

The Boolean expression B is evaluated; if it is true, statement S1 is executed; otherwise, statement S2 is executed.

In either case, the statement following the entire conditional statement is executed next, unless of course S1 or S2 changes the flow.

To avoid ambiguities, S1 may not itself begin with the word 'IF'.

Examples:

```
'IF' A 'OR' B 'THEN' X:= 1 'ELSE' X := 2

'IF' 12>Q 'THEN' 'GO TO' TOP 'ELSE' 'IF' 13<Q
                  'THEN' SORT
```

It is interesting to note that the first example above is equivalent to the arithmetic assignment statement X:='IF' A 'OR' B 'THEN' 1 'ELSE' 2. Either statement may be chosen arbitrarily to perform this function.

4.8 FOR Statements

form: 'FOR' arithmetic-variable:=for-list 'DO' S

The FOR statement calls for the repeated execution of the statement S which it encloses. The number of repetitions may actually be zero, one, or more and may depend on the computations done in S or in the FOR list.

The arithmetic variable may be referenced inside S; its value may change depending on the nature of the FOR list and of the computations inside S.

The FOR list consists of elements of any of the following three kinds, separated by commas. The FOR list gives a rule for assigning values to the arithmetic variable.

Kind of Element	Example	Values Assigned the Arithmetic Variable
A	0.001	0.001
A 'STEP' A 'UNTIL' A	10 'STEP'-1 'UNTIL' 0	10, 9, 8, . . . , 0
A 'WHILE' B	X+1 'WHILE' X>0	depends on the loop

The elements may be combined in any order. In practice combinations of like elements are most common.

```
0, 1, 1, 2, 3, 5
0 'STEP'.001 'UNTIL' 1, 1 'STEP'.01 'UNTIL' 10
```

When the element is an arithmetic expression, the expression is evaluated and its value assigned to the arithmetic variable; then the statement which constitutes the body of the FOR statement is executed exactly once. The next FOR-list element is then processed. If there are no more, the statement following the FOR statement is executed. (Of course, the body of the FOR statement may contain an exit which interrupts this sequence.)

When the element is a step-until element, the first arithmetic expression in it is evaluated and its value assigned to the arithmetic-variable. At this point a test is made to determine whether or not the body of the FOR should be executed even once. If it is executed, the value of the arithmetic variable is then increased by the value of the second expression. Testing, execution, incrementing, and testing continue until the test fails or the statement changes the sequence of control.

Denoting the three expressions by A1, A2, and A3, the test is this:

if A2 is positive, stop when the variable exceeds A3

if A2 is negative, stop when the variable is less than A3

In most cases, the value of the variable increases or decreases by A2 each time through the body, and the process stops when it has passed through A3.

Because A2 and A3 are recomputed at each execution of the loop, and the loop may alter either expression, strikingly complicated FOR statements can be constructed.

When the element is a 'WHILE' element, the value of the arithmetic expression is assigned to the variable, but the body is only executed if the value of the Boolean expression is 'TRUE'. Ordinarily, the body of the loop is repeated until it sets the Boolean expression to 'FALSE'.

4.9 Compound Statements

form: 'BEGIN' S;. . . ;S 'END'

Compound statements consist of several statements separated by semicolons and surrounded by the words 'BEGIN' and 'END'. Compound statements are very useful in situations where several simpler statements are to be executed conditionally. They greatly increase the power of the IF, conditional, and FOR statements.

Examples:

```
'IF' LATE 'THEN' 'BEGIN' HOUR := 12 ;
                 'GO TO' BED 'END'

'IF' B>0 'THEN' 'BEGIN' 'IF' V>C 'THEN'
                 'GO TO' END 'END'

'FOR' I := 3, 4, 5 'DO' 'BEGIN' A(I) := B(I) ;
                 B(I) := C(I) 'END'
```

Because a compound statement may appear anywhere a statement may appear, they may be nested.

```
'BEGIN' 'BEGIN'. . . 'END';'BEGIN'. . . 'END' 'END'
        'BEGIN' 'BEGIN'. . . 'END' 'END'
```

(In the second example, one of the begin-end pairs is redundant, but not illegal.)

It is worth noting here that no semicolon is needed before the word 'END', wherever it appears. In general, a semicolon is needed after an 'END', however, and the omission of this semicolon is a common error in ALGOL programming.

4.10 Blocks

form: 'BEGIN' D;...;D;S;...;S 'END'

Blocks differ from compound statements in that they include one or more declarations, by definition. Blocks are also statements and may appear wherever statements may appear, e.g., in a procedure declaration, which is in a block, which is in a procedure declaration, etc.

The primary purpose of the block is to declare entities which may be used only within that block.

```
...'BEGIN' 'REAL' X;...'END';'BEGIN' 'INTEGER'
                    X;...'END'
```

This example shows parallel blocks declaring the same identifier differently. The two X's are totally unrelated; allowing the re-use of identifiers without interference is one of the advantages of the block structure.

```
...'BEGIN' 'REAL' X,Y;...'BEGIN' 'INTEGER'
                    X,Z;...'END' 'END'...
```

This example shows an inner block redeclaring an identifier declared in an outer block; within the inner block, all references to X refer to the "inner" X; the "outer" X cannot be used. Within the outer block, including that part of the outer block which physically follows the inner block, it is the other way around: only the "outer" X can be used.

Y can be used in either block; Z, only in the inner block. In a sense, entities come into existence on entry into the block which declares them and cease to exist when an exit from the block occurs.

Because labels and procedures, too, have existence only in the blocks in which they are declared, it is impossible to enter a block except by way of its 'BEGIN'. It is possible to exit by falling through the end or by going to a label in an outer block.

4.10.1 OWN variables

'OWN' declarations (described earlier) are introduced to permit variables to retain their values, if not their existence, after the declaring block is exited. They permit the programmer to make the action taken in a block depend on the number of times it has been entered, for example.

```
'BEGIN' 'OWN' 'INTEGER' N;...'IF' N>5
                    'THEN'...;'END'
```

4.11 Dummy Statements

form: (empty)

Executing a dummy statement does not affect the program. Dummy statements are most useful in permitting the exits of procedures, compound statements, and blocks to be labeled.

```
...FINISH:'END'...
```

Dummy statements must be labeled.

5. COMPILATIONS

An ALGOL program is defined to be either a block or a compound statement. However, the ALGOL compilers will compile a block, a compound statement, or a procedure declaration as one unit. The executors will run one non-procedure and zero or more procedures as a single unit. The several programs are related as though they were part of a (nonexistent) larger block.

```
('BEGIN')
'PROCEDURE' NUMBER 1;...;‡
'PROCEDURE' NUMBER 2;...;‡

'BEGIN'...'END'‡        -- the single non-procedure
('END')
```

Note that the character ‡ ends every compilation (its only use).

Each of the above compilations which references one of the procedures must contain appropriate 'EXTERNAL' declarations (paragraph 2.8).

5.1 Implicit Declarations of Standard Functions

Standard functions which are implicitly declared may be thought of as being declared in the (nonexistent) outer block of the program structure. They are available to all compilations but may have their identifiers redeclared for local use by any.

6. SAMPLE PROGRAM

```
Program                                                   Remarks

'COMMENT' PRIME FACTORIZATION.                            This is a two-line comment
INPUT = 7 DIGITS, OUTPUT = FACTORS (ASCENDING ORDER);     ending with a semicolon
'BOOLEAN' 'PROCEDURE' PRIME (N);                          First compilation: a typed procedure
   'VALUE' N;'INTEGER' N;                                 Value list and specifications, in that order
   'BEGIN'                                                The body of the procedure is a block
      'EXTERNAL' NEXT;                                       containing 2 declarations
      'INTEGER' Q;                                           . . .
         Q := 2;                                             and 2 statements
         PRIME1:'IF' Q*Q>N 'THEN' PRIME := 'TRUE'            an if statement containing
            'ELSE' 'IF' N\Q*Q = N                            another if statement
               'THEN' PRIME := 'FALSE'                       which in turn contains
               'ELSE' 'BEGIN' NEXT (Q); 'GO TO' PRIME1 'END' a compound statement
   'END' PRIME; ‡                                         End of block, declaration, and compilation

'PROCEDURE' NEXT (X);                                     Second compilation: an untyped procedure
   'BEGIN'                                                   no value or specification:  X is an
      'BOOLEAN' 'EXTERNAL' PRIME;                            arithmetic variable called by name
         A:X := X+1;                                      A labeled arithmetic assignment statement
            'IF' 'NOT' PRIME (X) 'THEN' 'GO TO' A         Indirect recursion: NEXT calls PRIME calls NEXT
   'END' NEXT; ‡                                          End of block, declaration, and compilation

BEGIN:'BEGIN'                                             Third compilation: a labeled block
   'INTEGER' M,P,TYPEWRITER;                                 declaration
      'PROCEDURE' OUT (X);                                   declaration, parameter called by name
         'BEGIN' OUTPUT (TYPEWRITER, I7); IO (X); ENDIO 'END';  the body of OUT is a compound statement
   'EXTERNAL' NEXT;                                          declaration
      TYPEWRITER := 1;                                       . . .
I7:'FORMAT' ''I7'';                                          a format statement (not executed)
   INPUT (TYPEWRITER, I7); IO (M); ENDIO;                    three procedure statements
   P := 2;                                                   . . .
   PRIME3: 'IF' P*P>M                                        an if statement containing
      'THEN' 'BEGIN' OUT (M); 'GO TO' BEGIN 'END'            a compound statement and
      'ELSE' 'IF' M\P*P=M                                    another if statement containing
         'THEN' 'BEGIN' OUT (P); M := M/P 'END'              another compound statement
         'ELSE' NEXT (P);                                    . . .
   'GO TO' PRIME3;                                           . . .
'END' FACTORIZATION ‡                                        comments may follow 'END's
```

SECTION II

1. STRUCTURE OF THE LANGUAGE

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language -- explicit formulae -- called assignment statements.

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe, e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refer to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets 'BEGIN' and 'END' to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. A sequence of declarations followed by a sequence of statements and enclosed between 'BEGIN' and 'END' constitutes a block. Every declaration appears in a block in this way and is valid only for that block.

A program is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it. A program may be compiled in several parts.

The syntax and semantics of the language follow below.

2. BASIC CONCEPTS; SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS

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

basic-symbol = letter|digit|logical-value delimiter

2.1 Letters and Digits

2.1.1 Letters

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

Letters do not have individual meaning. They are used for forming identifiers and strings (cf. paragraphs 2.4 Identifiers, 2.6 Strings).

2.1.2 Digits

digit = 0|1|2|3|4|5|6|7|8|9

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

2.2 Logical Values

logical-value = 'TRUE'|'FALSE'

The logical values have a fixed obvious meaning.

2.3 Delimiters

delimiter = operator|separator|bracket|declarator|specificator

operator = arithmetic-operator|relational-operator|logical-operator|sequential-operator

arithmetic-operator = +|-|*|/|\|**

relational-operator = <|<=|=<|=|>=|=>|>|<>|><

logical-operator = ==|>>|'OR'|'AND'|'NOT'

sequential-operator = 'GO TO'|'IF'|'THEN'|'ELSE'|'FOR'|'DO'

separator = ,|.|△|:|;|:=|␣|'STEP'|'UNTIL'|'WHILE'|'COMMENT'

bracket = (or)|"|'BEGIN'|'END'

declarator = OWN|BOOLEAN|INTEGER|REAL|ARRAY|SWITCH|PROCEDURE

specificator = STRING|LABEL|VALUE

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

For the purpose of including text among the symbols of a program, the following "comment" conventions hold:

The sequence of basic symbols	is equivalent to
'COMMENT' (any-sequence-not-containing-;) ;	␣
'END' (any-sequence-not-containing ' or ;)	'END'

"Equivalence" in this case means that either of the structures shown in the left-hand column may be replaced, in any occurrence outside of strings, by the symbol shown on the same line in the right-hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.4 Identifiers

2.4.1 Syntax

identifier = letter | identifier-letter | identifier-digit

2.4.2 Examples

Q
SOUP
V17A
A34KTMNS
MARGARET

2.4.3 Semantics

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. Identifiers may be of any length; however, identifiers which are external names must be distinct from each other with respect to the first eight characters.

Lexicon words are listed in Appendix B.

The identifiers for standard procedures may be used for other purposes at will. The procedures available are described in Appendix C.

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. paragraphs 2.7 Quantities, Kinds and Scopes, and 5. Declarations).

2.5 Numbers

2.5.1 Syntax

unsigned-integer = digit | unsigned-integer digit

integer = unsigned-integer | + unsigned-integer | - unsigned-integer

decimal-fraction = . unsigned-integer

exponent-part = △ integer

decimal-number = unsigned-integer | decimal-fraction | unsigned-integer decimal-fraction

unsigned-number = decimal-number | exponent-part | decimal-number exponent-part

number = unsigned-number | + unsigned-number | - unsigned-number

2.5.2 Examples

0	-200.084	-.083△-02
177	+07.43△8	-△7
.5384	9.34△+10	△-4
+0.7300	2△-4	+△+5

2.5.3 Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4 Types

Integers are of type 'INTEGER'. All other numbers are of type 'REAL' (cf. paragraph 5.1 Type Declarations).

2.6 Strings

2.6.1 Syntax

string = ''(any-sequence-of-basic-symbols-not-containing-'')''

2.6.2 Examples

''THIS␣IS␣A␣STRING''

2.6.3 Semantics

To enable the language to handle arbitrary sequences of basic symbols, the string quotation marks '' (two single quotation marks) are introduced. The symbol ␣ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. paragraphs 3.2 Function Designators and 4.7 Procedure Statements).

2.7 Quantities, Kinds and Scopes

The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures.

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see paragraph 4.1.3.

2.8 Values and Types

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in paragraph 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. paragraph 3.1.4.1).

The various "types" ('INTEGER', 'REAL', 'BOOLEAN') basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. EXPRESSIONS

In the language, the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

If the interpretation of the syntax of a statement depends on the type of a parameter called by name, the type of the parameter will be assumed to be real.

expression = arithmetic-expression | Boolean-expression |
designational-expression

3.1 Variables

3.1.1 Syntax

variable-identifier = identifier

simple-variable = variable-identifier

subscript-expression = arithmetic-expression

subscript-list = subscript-expression | subscript-list,
subscript-expression

array-identifier = identifier

subscripted-variable = array-identifier (subscript-list)

variable = simple-variable subscripted-variable

3.1.2 Examples

EPSILON
INTEGER
A17
Q(7,2)
X(A+B,5)

3.1.3 Semantics

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (paragraph 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. paragraph 5.1 Type Declarations) or for the corresponding array identifier (cf. paragraph 5.2 Array Declarations).

3.1.4 Subscripts

3.1.4.1 Subscripted variables designate values which are components of multidimensional arrays (cf. paragraph 5.2 Array Declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in parentheses ( ). The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. paragraph 3.3 Arithmetic Expression).

3.1.4.2 Each subscript position acts like a variable of type 'INTEGER', and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. paragraph 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. paragraph 5.2 Array Declarations).

3.2 Function Designators

3.2.1 Syntax

procedure-identifier = identifier

actual-parameter = string | expression | array-identifier |
switch-identifier | procedure-identifier

letter-string = letter | letter-string letter

parameter-delimiter = , |) letter-string : (

actual-parameter-list = actual-parameter | actual-parameter-list parameter-delimiter actual-parameter

actual-parameter-part = empty|(actual-parameter-list)

function-designator = procedure-identifier actual-parameter-part

3.2.2 Examples

```
SIN(A-B)
J(V+S, N)
R
V(S-5)TEMPERATURE:(T)PRESSURE:(P)
COMPILE(":=")STACK:(Q)
```

3.2.3 Semantics

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. paragraph 5.4 Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in paragraph 4.7 Procedure Statements. Not every procedure declaration defines the value of a function designator. For function designators outside the scope of any corresponding procedure declarations, see paragraph 5.5.

3.2.4 Standard Functions

Certain standard functions and procedures, available for use with the compilers, are assumed to be declared in an implicit block outside and including all the statements of each compilation. That is, if any of them is declared in a program, it assumes local significance; if not, it is assumed to be the function or procedure described below, and will be loaded from the library at load time.

The standard functions and procedures are listed in Appendix C.

3.2.5 Transfers of Type

'REAL' variables may be implicitly transferred to type 'INTEGER' by assignment statements, calls by value, and use as arguments of standard functions, array bounds, and subscripts. In transferring to type 'REAL', no significance is lost. In describing transfer to type 'INTEGER', it is convenient to refer to the imaginary function ENTIER, where ENTIER(X) is the largest integer not greater than X.

```
ENTIER(5.5) = 5
ENTIER(-5.5) = -6
```

ENTIER is neither a standard function nor a reserved word.

3.3 Arithmetic Expressions

3.3.1 Syntax

adding-operator = +|-

multiplying-operator = *|/|\

primary = unsigned-number|variable|function-designator|(arithmetic-expression)

factor = primary|factor ** primary

term = factor|term multiplying-operator factor

simple-arithmetic-expression = term|adding-operator term|simple-arithmetic-expression adding-operator term

IF-clause = 'IF' Boolean-expression 'THEN''

arithmetic-expression = simple-arithmetic-expression|IF-clause|simple-arithmetic-expression 'ELSE' arithmetic-expression

3.3.2 Examples

Primaries:

```
7.394△-8
SUM
W(I+2, 8)
COS(Y+Z*3)
(A-3/Y+VU**8)
```

Factors:

```
OMEGA
SUM**COS(Y+Z*3)
7.394△-8**W(I+2, 8)**(A-3/Y+VU**8)
```

Terms:

```
U
OMEGA*SUM
```

Simple Arithmetic Expression:

```
U-OMEGA*SUM+ YU
```

Arithmetic Expressions:

```
W*U-Q(S+CU)**2
'IF' Q>0 'THEN' S+3*Q/A 'ELSE' 2*S
```

3.3.3 Semantics

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions, this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in paragraph 3.3.4. The actual numerical value of a primary is obvious in the case of numbers. For variables, it is the current value (assigned last in the dynamic sense), and for function designators, it is the value arising from the computing rules defining the procedure (cf. paragraph 5.4.4 Values of Function Designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses, the value must through recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include IF clauses, one of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. paragraph 3.4 Boolean Expressions). This selection is made as follows: The Boolean expressions of the IF clauses are evaluated one by one in sequence from left to right until one having the value 'TRUE' is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

'ELSE' (simple arithmetic expression)

is equivalent to the construction:

'ELSE' 'IF' 'TRUE' 'THEN' (simple arithmetic expression)

3.3.4 Operators and Types

Apart from the Boolean expressions of IF clauses, the constituents of simple arithmetic expressions must be of types 'REAL' or 'INTEGER' (cf. paragraph 5.1 Type Declarations). The meaning of the basic operators and the types of the expressions to which they lead are given to the following rules:

3.3.4.1 The operators +, -, and * have the conventional meaning (additions, subtraction, and multiplication). The type of the expression will be 'INTEGER' if both of the operands are of 'INTEGER' type; otherwise, 'REAL'.

3.3.4.2 The operations (term)/(factor) and (term)\(factor) both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. paragraph 3.3.5). Thus, for example

A/B*7/(P-Q)*V/S

means

$((((A*(B^{-1}))*7)*((P-Q)^{-1}))*V)*(S^{-1})$

The operator / is defined for all four combinations of types 'REAL' and 'INTEGER' and will yield results of 'REAL' type in any case. The operator \ is defined only for two operands both of type 'INTEGER' and will yield a result of type 'INTEGER', mathematically defined as follows:

a\b = SIGN(A/B)*ENTIER(ABS(A/B))

(cf. paragraphs 3.2.4 and 3.2.5)

3.3.4.3 The operation factor ** primary denotes exponentiation, where the factor is the base and the primary is the exponent. For example,

2**N**K means $(2^N)^K$

while

2**(N**M) means $2^{(N^M)}$

Writing i for a number of 'INTEGER' type, r for a number of 'REAL' type, and a for a number of either 'INTEGER' or 'REAL' type, the result is given by the following rules:

A**I — If i > 0, a*a*...*a (i times), of the same type as a.
If i = 0, if a ≠ 0, 1, of the same type as a; if a = 0, undefined.
If i < 0, if a ≠ 0, 1/(a*a*...*a) (the denominator has -i factors), of type 'REAL'; if a = 0, undefined.

A**R — If a > 0, EXP(r*LN(a)), of type 'REAL'.
If a = 0, if r > 0, 0.0, of type 'REAL'; if r ≦ 0, undefined.
If a < 0, always undefined.

3.3.5 Precedence of Operators

The sequence of evaluation of primaries within an expression is, in effect, from left to right.

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.3.5.1 According to the syntax given in paragraph 3.3.1, the following rules of precedence hold:

first: **
second: * / \
third: + -

3.3.5.2 The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently, the desired order of execution of operations

within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6 Arithmetics of 'REAL' Quantities

Numbers and variables of type 'REAL' must be interpreted in the sense of numerical analysis, i.e., as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood.

Integer quantities may take on the value zero or any value representable in two's complement form in 22 bits.

Real quantities may take on the value zero or any value representable in two's complement form with a 39-bit fraction and a 9-bit binary exponent.

Programs in which quantities outside the stated ranges occur or are generated are undefined.

3.4 Boolean Expressions

3.4.1 Syntax

```
relational-operator = <|<=|=<|>=|=>|=|>|><|<>
relation = simple-arithmetic-expression relational-
           operator simple-arithmetic-
           expression
Boolean-primary = logical-value|variable|function-
           designator|relation|(Boolean-
           expression)
Boolean-secondary = Boolean-primary|'NOT'
           Boolean-primary
Boolean-factor = Boolean-secondary|Boolean-factor
           'AND' Boolean-secondary
Boolean-term = Boolean-factor|Boolean-term 'OR'
           Boolean-factor
implication = Boolean-term|implication >> Boolean-
           term
simple-Boolean = implication|simple-Boolean ==
           implication
Boolean-expression = simple-Boolean|IF-clause
           simple-Boolean 'ELSE' Boolean-
           expression
```

3.4.2 Examples

```
X = -2
Y>V 'OR' Z<Q
A+B>-5 'AND' Z-D>Q**2
P 'AND' Q 'OR' X <> Y
'IF' K<1 'THEN' S>W 'ELSE' H<=C
'IF' 'IF' 'IF' A 'THEN' B 'ELSE' C 'THEN' D 'ELSE' F
                    'THEN' G 'ELSE' H<K
```

3.4.3 Semantics

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in paragraph 3.3.3.

3.4.4 Types

Variables and function designators entered as Boolean primaries must be declared 'BOOLEAN' (cf. paragraphs 5.1 Type Declarations and 5 4.4 Values of Function Designators).

3.4.5 The Operators

Relations take on the value 'TRUE' whenever the corresponding relation is satisfied for the expression involved; otherwise, 'FALSE'.

The meaning of the logical operators 'NOT', 'AND', 'OR', >> (implies), and == (equivalent), is given by the following function table:

B1 B2	F F	F T	T F	T T
'NOT' B1	T	T	F	F
B1 'AND' B2	F	F	F	T
B1 'OR' B2	F	T	T	T
B1 >> B2	T	T	F	T
B1 == B2	T	F	F	T

where T implies the value 'TRUE' and F implies 'FALSE'.

3.4.6 Precedence of Operators

All primaries in simple Booleans will be evaluated.

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1 According to the syntax given in paragraph 3.4.1, the following rules of precedence hold:

first:	arithmetic expression according to paragraph 3.3.5
second:	relational operators
third:	'NOT'
fourth:	'AND'
fifth:	'OR'
sixth:	>>
seventh:	==

3.4.6.2 The use of parentheses will be interpreted in the sense given in paragraph 3.3.5.2.

3.5 Designational Expressions

3.5.1 Syntax

label = identifier

switch-identifier = identifier

switch-designator = switch-identifier (subscript-expression)

simple-designational-expression = label| switch-designator|(designational-expression)

designational-expression = simple-designational-expression| IF-clause|simple-designational-expression 'ELSE' designational-expression

3.5.2 Examples

```
P9
CHOOSE (N-1)
TOWN ('IF' Y>0 'THEN' N 'ELSE' N+1)
```

3.5.3 Semantics

A designational expression is a rule for obtaining a label of a statement (cf. paragraph 4 Statements). Again, the principle of the evaluation is entirely analogous to that of arithmetic expressions (paragraph 3.3.3). In the general case, the Boolean expressions of the IF clauses will select a simple designational expression. If this is a label, the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. paragraph 5.3 Switch Declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator, this evaluation is obviously a recursive process.

3.5.4 The Subscript Expression

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. paragraph 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1, 2, 3, . . . , n, where n is the number of entries in the switch list.

4. STATEMENTS

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken to GO TO statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

To make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks, the definition of statement must necessarily be recursive. Also, since declarations, described in paragraph 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1 Compound Statements and Blocks

4.1.1 Syntax

unlabeled-basic-statement = assignment-statement| GO-TO-statement|dummy-statement|procedure-statement

basic-statement = unlabeled-basic-statement| label : basic-statement|label : format-statement

unconditional-statement = basic-statement|compound-statement| block

statement = unconditional-statement conditional-statement|FOR-statement

compound-tail = statement 'END'|statement ; compound-tail

block-head = 'BEGIN' declaration|block-head ; declaration

unlabeled-compound = 'BEGIN' compound-tail

unlabeled-block = block-head ; compound-tail

compound-statement = unlabeled-compound|label : compound-statement

block = unlabeled-block| label : block

program = block|compound-statement

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound Statements:

L:L:. . . 'BEGIN' S;S;. . . S;S 'END'

Block:

L:L:. . . 'BEGIN' D;D;. . . D;S;S;. . . S;S 'END'

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2 Examples

Basic Statements:

```
A := P+Q
'GO TO' NAPLES
START:CONTINUE:W := 7.993
```

Compound Statement:

```
'BEGIN' X := 0; Y := A+B**2 'END'
```

Block

```
Q:'BEGIN' 'INTEGER' I,J; 'REAL' W;
                   I := 1; J := I+L**2;
                   W := A(I + J, I*J)
'END' BLOCK Q
```

4.1.3 Semantics

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

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be nonlocal to it; i.e., will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e., labeling that statement, behaves as though declared in the head of the smallest embracing block; i.e., the smallest block whose brackets 'BEGIN' and 'END' enclose that statement. In this context, a procedure body must be considered as if it were enclosed by 'BEGIN' and 'END' and treated as a block.

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

4.2 Assignment Statements

4.2.1 Syntax

left-part = variable := | procedure-identifier :=

left-part-list = left-part | left-part-list left-part

assignment-statement = left-part-list arithmetic-expression | left-part-list Boolean-expression

4.2.2 Examples

```
S := P(0) := N := N+1+S
N := N+1
V := Q>Y 'AND' Z
```

4.2.3 Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may occur only within the body of a procedure defining the value of a function designator (cf. paragraph 5.4.4). The process will in general cases be understood to take place in three steps as follows:

4.2.3.1 Any subscript expressions occurring in the left-part-variables are evaluated in sequence from left to right.

4.2.3.2 The expression of the statement is evaluated.

4.2.3.3 The value of the expression is assigned to all the left-part-variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4 Types

The type associated with all variables and procedure identifiers of a left-part-list must be the same. If this type is 'BOOLEAN', the expression must likewise be 'BOOLEAN'. If the type is 'REAL' or 'INTEGER', the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from 'REAL' to 'INTEGER' type, the transfer function is understood to yield a result equivalent to:

```
ENTIER(E+0.5)
```

where E is the value of the expression. The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. paragraph 5.4.4).

4.3 GO TO Statements

4.3.1 Syntax

GO-TO-statement = 'GO TO' designational-expression

4.3.2 Examples

```
'GO TO' ALPHA
'GO TO' EXIT (N+1)
'GO TO' TOWN ('IF' Y<0 'THEN' N 'ELSE' N+1)
```

4.3.3 Semantics

A GO TO statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus, the next statement to be executed will be the one having this value as its label.

4.3.4 Restriction

Since labels are inherently local, no GO TO statement can lead from outside into a block. A GO TO statement may, however, lead from outside into a compound statement.

4.3.5 GO TO an Undefined Switch Designator

A GO TO statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined because its subscript is out of the range of the corresponding switch. If the designational expression is undefined for any other reason, the GO TO is undefined.

The phrase "equivalent to a dummy statement" means that the flow of the program does not change at this point.

4.4 Dummy Statements

4.4.1 Syntax

dummy-statement = empty

4.4.2 Examples

L:'BEGIN';JOHN:'END'

4.4.3 Semantics

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

4.5 Conditional Statements

4.5.1 Syntax

IF-clause = 'IF' Boolean-expression 'THEN'

unconditional-statement = basic statement | compound-statement | block

IF-statement = IF-clause unconditional-statement

conditional-statement = IF-statement | IF-statement 'ELSE' statement | IF-clause FOR-statement | label : conditional-statement

4.5.2 Examples

'IF' X>0 'THEN' N := N+1
'IF' V>U 'THEN' L:Q:= N+M 'ELSE' 'GO TO' R

4.5.3 Semantics

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

4.5.3.1 IF Statement. The unconditional statement of an IF statement will be executed if the Boolean expression of the IF clause is true. Otherwise, it will be skipped and the operation will be continued with the next statement.

4.5.3.2 Conditional Statement. According to the syntax, two different forms of conditional statements are possible. These may be illustrated as follows:

'IF' B1 'THEN' S1 'ELSE' 'IF' B2 'THEN' S2 'ELSE' S3; S4

and

'IF'B1 'THEN' S1 'ELSE' 'IF' B2 'THEN' S2 'ELSE' 'IF' B3 'THEN' S3; S4

Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expressions of the IF clauses are evaluated one after the other in sequence from left to right until one yielding the value 'TRUE' is found. Then the unconditional statement following this Boolean is executed. Unless this statement defined its successor explicitly, the next statement to be executed will be S4; i.e., the statement following the complete conditional statement. Thus, the effect of the delimiter 'ELSE' may be described by saying that it defines the successor of the statement which it follows to be the statement following the complete conditional statement.

The construction

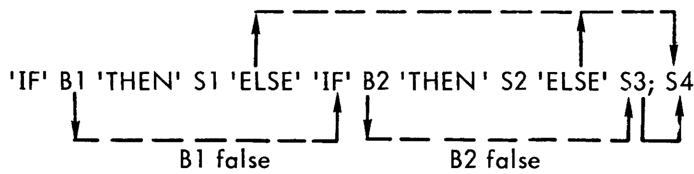
'ELSE' unconditional statement

is equivalent to

'ELSE' 'IF' 'TRUE' 'THEN' unconditional statement

If none of the Boolean expressions of the IF clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation, the following picture may be useful:



4.5.4 GO TO into a Conditional Statement

The effect of a GO TO statement leading into a conditional statement follows directly from the above explanation of the effect of 'ELSE'.

4.6 FOR Statements

4.6.1 Syntax

FOR-list-element = arithmetic-expression| arithmetic-expression 'STEP' arithmetic-expression 'UNTIL' arithmetic-expression| arithmetic-expression 'WHILE' Boolean-expression

FOR-list = FOR-list-element| FOR-list, FOR-list-element

FOR-clause = 'FOR' variable := FOR-list 'DO'

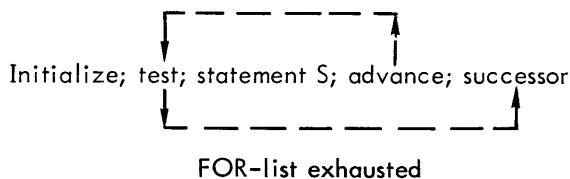
FOR-statement = FOR-clause statement| label : FOR-statement

4.6.2 Examples

```
'FOR' Q:= 1 'STEP' S 'UNTIL' N 'DO' A(Q) := B(Q)
'FOR' K := 1, V1*2 'WHILE' V1<N 'DO' 'FOR'
          J := I+G, L, 1 'STEP' 1 'UNTIL'
          N, C+D 'DO' A(K, J) := B(K, J)
```

4.6.3 Semantics

A FOR clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition, it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



FOR-list exhausted

In this picture, the word initialize means perform the first assignment of the FOR clause. Advance means perform the next assignment of the FOR clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the FOR statement. If not, the statement following the FOR clause is executed.

4.6.4 The FOR List Elements

The FOR list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the FOR list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of FOR list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1 Arithmetic expression. This element gives rise to one value, namely, the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2 Step-until-element. An element of the form A 'STEP' B 'UNTIL' C, where A, B, and C are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

```
V := A;
L1: 'IF'(V - C)*SIGN(B)>0 'THEN' 'GO TO'
                     element exhausted
statement S;
V := V+B;
'GO TO' L1;
```

where V is the controlled variable of the FOR clause and element exhausted points to the evaluation according to the next element in the FOR list, or if the step-until-element is the last of the list, to the next statement in the program.

The arithmetic expressions B and C in "A 'STEP' B 'UNTIL' C" and "B 'WHILE' C" are evaluated dynamically, that is, once per execution of the loop.

If the controlled variable is subscripted, its subscript is evaluated once, at the top of the loop.

4.6.4.3 WHILE-element. The execution governed by a FOR list element of the form E 'WHILE' F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

```
L3:V := E
'IF' 'NOT' F 'THEN' 'GO TO' element exhausted
Statement S;
'GO TO' L3
```

where the notation is the same as in 4.6.4.2 above.

4.6.5 The Value of the Controlled Variable upon Exit

The value of a controlled variable after exhaustion of a FOR list or on exit from the body of the FOR by a GO TO statement is defined as the last value dynamically assigned to the variable in performing the loop.

4.6.6 GO TO Leading into a FOR Statement

The effect of a GO TO statement, outside a FOR statement, which refers to a label within the FOR statement, is undefined.

4.7 Procedure Statements

4.7.1 Syntax

actual-parameter = string|expression|array-identifier|switch identifier|procedure-identifier

letter-string = letter| letter-string letter

parameter-delimiter = ,|)letter-string:(

actual-parameter-list = actual-parameter|actual-parameter-list parameter-delimiter actual-parameter

actual-parameter-part = empty|(actual-parameter-list)

procedure-statement = procedure-identifier actual-parameter-list

4.7.2 Examples

SPUR (A) ORDER:(7) RESULT TO:(V)
TRANSPOSE (W, V+1)

4.7.3 Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. paragraph 5.4 Procedure Declarations). For procedure statements outside the scope of any corresponding procedure declarations, see paragraph 5.5. Where the procedure body is a statement written in ALGOL, the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

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

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

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

4.7.4 Actual-Formal Correspondence

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

4.7.5 Restrictions

For a procedure statement to be defined, it is evidently necessary that the operations on the procedure body defined in paragraphs 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.

This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

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

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

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

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

4.7.5.5 Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement, such restrictions must evidently be observed.

4.7.6 Parameter Delimiters

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

4.8 Format Statements

Values for variables may be transmitted from or to peripheral devices by the program at any point. For this purpose, several standard procedures are supplied and the 'FORMAT' statement is introduced into the language.

4.8.1 Syntax

format-statement = 'FORMAT' ''format-list''

format-list = specification| specification format-list
specification, format-list

specification = repetitions I integer| scale repetitions E integer. integer|scale repetitions F integer. integer|repetitions A integer|$ alphanumeric-string $|integer H alphanumeric-string|repetitions X| / |repetitions (format-list)

scale = sign integer P|empty

sign = +|-|empty

repetitions = integer|empty

4.8.2 Examples

'FORMAT' ''I8''
'FORMAT' ''/2E12.4, 2(A8, 2PF4.0)SDS''

4.8.3 Semantics

Format strings specify the correspondence between the memory representation and the printed (or typed or punch) representation of the values of variables in input/output lists. When the INPUT or the OUTPUT procedure is called, the format which it references is scanned from left to right. During the scan, the sequence of events is as follows:

1. Each / on input indicates that the next record should be read. On output, it signals the scanner to transmit the current record. Any further data will begin a new record. At the end of the format list, a / is implied.
2. ''integer (format-list)'' causes the format-list to be scanned as many times as the integer directs. Repetition within repetitions are not permitted.
3. ''(format-list)'' causes the format-list to be scanned repeatedly until the input record runs out (on input) or until the ENDIO procedure is called (on output). This form normally appears at the end of the format. Parentheses are implied around the entire format-string, so the entire string is rescanned if no parentheses occur in it. The action caused by each specification depends on the specification.

4.8.3.1 Conversion of data: I, E, and F specifications. Conversions of numerical data during input/output may be one of three types:

type-E

internal form - binary floating-point
external form - decimal floating-point

type-F

internal form - binary floating-point
external form - decimal fixed-point

type-I

internal form - binary integer
external form - decimal integer

These types of conversions are specified by the forms:

Ew.d
Fw.d
Iw

where E, F, and I specify the type of conversion required, w is an integer specifying the width of the field, and d is an integer specifying the number of decimal places to the right of the decimal point.

As an example, the statement

    'FORMAT' ''E12.3, F12.3, I12''

may correspond to the line

    10.987E⊔ 00   -16.909   333

Note that the decimal fixed-point number (type F) has a decimal point but no exponent, whereas the decimal floating-point (type E) has an exponent. On output the exponent always has the form shown, i.e., an "E" followed by a minus sign or blank and a two-digit integer. On input, however, the "E" or the entire exponent may be omitted on the external form. For example, the following are all equivalent E12.3 fields:

    10.987
    10987E-3
    10987-03

The field width w includes all of the characters (decimal point, signs, blanks, etc.) which comprise the number. If a number is too long for its specified field, the excess characters are lost. Since numbers are right justified in their fields, the loss is from the most significant part of the number.

During input, the appearance of a decimal point "." in an E or F type number overrides the d specification of the field. If there is no explicit decimal point, the point is positioned d places from the right of the field, not counting the exponent, if any.

For example, a number with external appearance 314159E-2 and specification E12.3 is interpreted as 314.159E-2, or 3.14159.

The same conversion may be used for several successive numbers by preceding the specification with a nonzero integer which represents the number of repetitions.

Scale factors can be specified for F and E type conversions. A scale factor has the from nP where n is a signed or unsigned integer specifying the scale factor. In F type conversions, the scale factor specifies a power of 10, such that

    external number = (internal number)*(power of 10)

With E type conversions, the scale factor is used to change the number by a power of 10 and then to correct the exponent such that the result represents the same real number as before, but now has a different form.

For example, if the statement

    'FORMAT' ''F12.3, E12.3''

corresponds to the output line

    -1.234   9.752E⊔01

then the statement

    'FORMAT' ''-2PF12.3, +1PE12.3''

may correspond to the line

    -.012   97.520E⊔00

The scale factor is assumed zero if none has been given. However, once a value has been given, it holds for all E and F type conversions following the scale factor. A zero scale factor can be used to return conditions to normal. (Scale factors are ignored in I conversions.)

On input, a comma may be used to terminate a numerical field. This allows simplified preparation of data records; a data record using the format

    'FORMAT' ''F12.3, 3I12''

may be punched

    .0, 15, 16, 20

instead of

    .0   15   16   20

The field width specified in the format must be greater than the number of characters encountered before the commas.

4.8.3.2 Transmission of alphanumeric information: A, X, H, and $ specifications. The A-specification resembles the I-, E-, and F-specifications in that it specifies the transmission of data to or from a quantity (that is, a constant, variable, or subscripted variable) in memory. The quantity involved must be of type real. Repetition of A-specifications is permitted.

The form of the specification is Ac, where c is the number of characters to be transmitted, to a maximum of 8. If $c>8$, the remaining characters are lost on input or replaced with blanks on output. If $c<8$, trailing blanks are supplied,

The X-specification specifies the output of a single blank; on input, a single character is ignored. Repetition of X-specifications is permitted.

The H-specification specifies the output of the characters (including blanks) which immediately follow the H. The integer preceding the H specifies the number of characters which follow it. On input, the same number of characters is read in; they replace the original characters in the format in memory: 6HGEORGE

The $-specification is identical to the H-specification except that no count is given; instead, the characters to be transmitted are placed (including blanks) between two dollar signs: $GEORGE$

4.8.4 Input/Output

Input and output are performed by executing the following procedure statements dynamically:

```
INPUT (code, format)          or "OUTPUT"
   .
   .
   .
IO (list)
   .
   .
   .
ENDIO
```

where the following definitions apply:

INPUT	standard procedure to initiate input
OUTPUT	standard procedure to initiate output
code	arithmetic expression the value of which specifies the unit involved and the form of data transferred
format	designational expression designating a format statement
IO	standard procedure (with a variable number of arguments) to perform input/output
list	real or integer variables, arrays, or (for output) constants, separated by commas
ENDIO	standard procedure to terminate input/output

The input/output codes are as follows:

0	paper tape, alphanumeric
1	typewriter, alphanumeric
2	punched cards, alphanumeric
3	line printer, alphanumeric
10+N	magnetic tape N, alphanumeric
20+N	magnetic tape N, binary

Execution of the call on INPUT or OUTPUT in alphanumeric mode causes the designated format to be scanned and the appropriate action taken, until the point where an I, E, F, or A specification is encountered. Execution of the program then continues. One or more calls to IO supply lists of parameters corresponding to the I, E, F, and A specification of the format; when an IO call is executed the conversions for its parameters take place. Finally, a call to ENDIO terminates the transmission until another call to INPUT or OUTPUT is executed.

Alphanumeric transmissions may be performed by calling the INPUT (or OUTPUT) and ENDIO procedures only. If one or more calls on the IO procedure occur, however, their arguments are transmitted as specified by the referenced format statement, and in that order.

The following examples illustrate the way in which format statements, input/output procedure calls, and data on external media interact:

Example 1

Consider a program containing the following declaration and statements:

```
'INTEGER'M1, M2, N1, N2...
XYZ: 'FORMAT' "I8"...
INPUT(2, XYZ)... IO(M1, M2)... IO(N1, N2)...
                  ENDIO
```

The statements on the third line may be written in any order, but are executed in the order shown.

The input statement requests input from cards (code 2). The format "I8" is equivalent to (I8/) since the parentheses and the final end-of-record are implied for all formats.

The sequence of events is as follows:

When the call to INPUT is executed, scanning of the format begins. The specification I8 which is encountered immediately cannot be processed until the program calls the IO procedure, so program execution continues until the first call on IO. Then the first 8 columns of the first data card are converted to a 22-bit binary integer and stored in the variable M1.

Scanning of the format continues. The "/" (implied) causes the remainder of the first data card to be ignored. The ")" (also implied) signals the format scanner to return to the previous "(", in this case the one implied at the beginning of the format.

Again the specification I8 is encountered. The first 8 columns of the second data card are now converted to a 22-bit integer, and that value stored in M2.

The process continues while values for N1 and N2 are stored. The call to ENDIO then terminates transmission. Further Input/Output may be initiated by re-executing this sequence of statements or by executing another similar sequence involving another call to INPUT (or OUTPUT).

Example 2

Suppose six variables of type real are to be output (to the typewriter) according to the following format:

/2E12.4, 2(A8, 2PF4.0/)SDS

The typeout might look like this:

```
⊔-1.3625E-11⊔⊔0.0000E⊔00⊔ALPHA=⊔100.
 E12.4        E12.4        A8       2PF.0

⊔⊔BETA=⊔⊔⊔6.
 A8       2PF4.0

SDS
⊔
$SDS$
```

Execution of the call on INPUT or OUTPUT in binary mode is similar to that of the alphanumeric mode with the the following exceptions:

1. A format statement is not used to control conversion. Instead, the values designated by the IO procedure calls are transmitted as binary words without conversion.
2. Although the second argument of the INPUT or OUTPUT procedure call is required to be in the statement, it is not used. The second argument must be a designational expression; however, it need not designate a format statement.

5. DECLARATIONS

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block, the particular identifier may be used for other purposes (cf. paragraph 4.1.3).

Dynamically, this implies the following: at the time of an entry into a block (through the 'BEGIN', since the labels inside are local and therefore inaccessible from outside), all identifiers declared for the block assume the significance implied by the nature of the declaration given. If these identifiers had already been defined by other declarations outside, they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through 'END' or by a GO TO statement), all identifiers which are declared for the block lose their local significance.

A declaration may be marked with the additional declarator 'OWN'. This has the following effect: upon reentry into the block, the values of OWN quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as OWN are undefined. OWN declarations are taken to declare their variables available to the current block but nonlocal to the entire program.

For the behavior of OWN variables under recursion, see paragraph 5.4.6.

Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. paragraphs 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax

declaration = type-declaration | array-declaration | switch-declarations | procedure-declaration | external-procedure-declaration

5.1 Type Declarations

5.1.1 Syntax

type-list = simple-variable | simple-variable type-list

type = 'REAL' | 'INTEGER' | 'BOOLEAN'

local-or-own-type = type | 'OWN' type

type-declaration = local-or-own-type type-list

5.1.2 Examples

'INTEGER' P, Q, S
'OWN' 'BOOLEAN' ACRYL, N

5.1.3 Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values 'TRUE' and 'FALSE'.

In arithmetic expressions, any position which can be occupied by a real declared variable may be occupied by

an integer declared variable, except as stated in paragraph 3.3.4.2.

For the semantics of 'OWN', see paragraph 5.

5.1.4 Initialization of OWN Variables

In an 'OWN' type declaration, each variable in the list may optionally have a value assigned to it at compile time by appending to the variable a replace operator followed by a value; e.g.,

```
'OWN' 'REAL' K := 3, BETA, RHO := 3.14;
'OWN' 'BOOLEAN' P := 'TRUE', Q := 'FALSE', F, R;
```

The value must be of the type declared.

5.2 Array Declarations

5.2.1 Syntax

```
lower-bound = arithmetic-expression
upper-bound = arithmetic-expression
bound-pair = lower-bound : upper-bound
bound-pair-list = bound-pair | bound-pair-list ,
                  bound-pair
array-segment = array-identifier (bound-pair-list) |
                array-identifier , array-segment
array-list = array-segment | array-list , array-segment
array-declaration = 'ARRAY' array-list | local-or-
                    own-type 'ARRAY' array-list
```

5.2.2 Examples

```
'ARRAY' A, B, C(7:N, 2:M), S(-2:10)
'OWN' 'INTEGER' 'ARRAY' A('IF' C<0 'THEN' 2
                'ELSE' 1:20)
'REAL' 'ARRAY' Q(-7:-1)
```

5.2.3 Semantics

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

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

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

5.2.3.3 Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given, the type 'REAL' is understood.

5.2.4 Lower Upper Bound Expressions

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

5.2.4.2 The expressions can only depend on variables and procedures which are nonlocal to the block for which the array declaration is valid. Consequently, in the outermost block of a program, only array declarations with constant bounds may be declared.

5.2.4.3 An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

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

5.2.5 The Identity of Subscripted Variables

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared 'OWN', the values of the corresponding subscripted variables will, at any time, be defined only for those variables which have subscripts within the most recently calculated subscript bounds.

5.2.6 OWN Array Declaration

In an OWN array declaration, the elements of an array may be assigned a set of values at compile time by appending to the array identifier a replace operator followed by a list of values which are to be assigned to successive elements of the array (the first subscript varies most rapidly). The value list is terminated by the semicolon which also terminates the statement. Elements of the array which are not explicitly assigned a value are assigned the value of the immediately preceding element; e.g.,

```
'OWN' 'INTEGER' 'ARRAY' PHI (0:12),
            OMEGA (0:3, -1:1) := 0, 1, 2, 4,
            0, ,,, 8 ;
```

will assign the values as follows:

```
OMEGA (0,-1) 0
OMEGA (1,-1) 1
OMEGA (2,-1) 2
OMEGA (3,-1) 4
OMEGA (0, 0) 0
OMEGA (1, 0) 0
OMEGA (2, 0) 0
OMEGA (3, 0) 0
OMEGA (0, 1) 8
OMEGA (1, 1) 8
OMEGA (2, 1) 8
OMEGA (3, 1) 8
```

The values assigned must be of the type declared.

5.3 Switch Declarations

5.3.1 Syntax

switch-list = designational-expression | switch-list, designational-expression

switch-declaration = 'SWITCH' switch-identifier := switch-list

5.3.2 Examples

```
'SWITCH' S := S1,S2,Q(M),'IF' V>-5 'THEN' S3
              'ELSE' S4

'SWITCH' Q := P1,W
```

5.3.3 Semantics

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

5.3.4 Evaluation of Expression in the Switch List

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

5.3.5 Influence of Scopes

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

5.4 Procedure Declarations

5.4.1 Syntax

formal-parameter = identifier

formal-parameter-list = formal-parameter | formal-parameter-list parameter-delimiter formal-parameter

formal-parameter-part = empty | (formal-parameter-list)

identifier-list = identifier | identifier-list, identifier

value-part = 'VALUE' identifier-list ; | empty

specifier = 'STRING' | type | 'ARRAY' | type 'ARRAY' | 'LABEL' | 'SWITCH' | 'PROCEDURE' | type 'PROCEDURE'

specification-part = specifier identifier-list ; | specification-part specifier identifier-list | empty

procedure-heading = procedure-identifier formal-parameter-part ; value-part specification part

procedure-body = statement

procedure-declaration = 'PROCEDURE' procedure heading procedure-body | type 'PROCEDURE' procedure-heading procedure body

5.4.2 Examples

```
'PROCEDURE' SPUR (A) ORDER : (N) RESULT : (S) ;
'VALUE' N ; 'ARRAY' A ; 'INTEGER' N ;
'REAL' S ;
'BEGIN' 'INTEGER' K ;
S := 0
'FOR' K := 1 'STEP' 1 'UNTIL' N 'DO'
S := S + A (K,K) 'END'

'PROCEDURE' INNERPRODUCT (A,B) ORDER :
            (K,P) RESULT : (Y) ;
'VALUE' K ;
'INTEGER' K,P ; 'REAL' Y,A,B ;
'BEGIN' 'REAL' S ; S := 0 ;
'FOR' P := 1 'STEP' 1 'UNTIL' K 'DO' S := S+A*B ;
Y := S
'END' INNERPRODUCT
```

5.4.3 Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement,

the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. paragraphs 3.2 Function Designators and 4.7 Procedure Statements), be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or nonlocal to the body depending on whether or not they are declared within the body. Those which are nonlocal to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently, the scope of any label labeling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in paragraph 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.4 Values of Function Designators

For a procedure declaration to define the value of a function designator, there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.5 Specifications

In the heading, a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part, no formal parameter may occur more than once. Specifications of formal parameters must be supplied, unless the parameter is called by name and the declarator is simply 'REAL' or 'INTEGER'.

If the specification of a formal parameter called by value conflicts in type with the corresponding actual parameter in that one is of type real and the other is of type integer, the value will be transferred to the type requested by the specification.

If the specification of a formal parameter called by name conflicts in type with the corresponding actual parameter in that one is of type real and the other is of type integer, the type of the actual parameter will be used.

If any other conflict between the specification and the corresponding declaration exists, the program is undefined.

5.4.6 Recursive Calling of Procedures

5.4.6.1 Definition. A procedure is said to be called recursively when the execution of one of its statements requires that the procedure be re-entered with new arguments. The procedure will eventually be exited repeatedly or the program will not terminate.

Every procedure is potentially recursive.

5.4.6.2 Kinds of recursive calls. Recursive calling may occur in any of the following ways:

A statement of the procedure may explicitly call the procedure.

A statement of the procedure may call another procedure which, perhaps at many levels' distance, calls the original procedure.

The procedure name may be an actual parameter to the procedure.

An expression which calls the procedure may be an actual parameter, called by name, to the procedure.

(In the last two cases, recursion occurs at every reference to the corresponding formal parameter.)

An expression which calls the procedure may be an actual parameter, called by value, to the procedure. In this case, one level of recursion occurs.

5.4.6.3 Behavior of variables under recursion. At each level of recursion, each non-OWN variable is redeclared and takes on a new identity. Its value is undefined at the entrance to the declaring block; at the exit from the declaring block, its previous value is regained. All current values of each variable are retained; previous values are used in evaluating parameters called by name.

Each OWN variable, however, has a unique identity. Its previous value is available for use at re-entrances to the declaring block; its new value, if any, is retained at the exit. If the variable participates in the evaluation of a parameter called by name, its then-current value is the one used.

5.5 External Procedure Declarations

5.5.1 Syntax

procedure-identifier-list = procedure-identifier|
procedure-identifier-list,
procedure-identifier

external-procedure-declaration = 'EXTERNAL'
procedure-identifier-list| type
'EXTERNAL' procedure-identifier-list

5.5.2 Examples

'EXTERNAL' INTEGRATION
'REAL' 'EXTERNAL' SINH, COSH, TANH

5.5.3 Semantics

An external procedure declaration declares that the identifiers associated with it are names of procedures to be compiled separately. Outside the block in which the external declaration appears, the procedure identifiers may be redeclared as other quantities; however, the names of the external procedures required by any compilation must be unique in the first eight characters.

The declaration of type serves to distinguish those procedures which may be called as function designators. If the type differs from the type declared with the procedure body when it is compiled in that one is of type real and the other is of type integer, the latter takes precedence. If any other conflict exists, the program is undefined.

5.5.4 Program Organization

A program may be compiled all together or it may be broken into two or more compilations according to the following rules.

1. Each compilation consists of a complete block or a complete procedure.
2. Compilations which are to be executed together are all procedures, except one.
3. The procedures are loaded first; the single block last. Each procedure is followed by the characters ; ‡. Each block is followed by the character ‡.

5.5.5 Non-ALGOL Code (see Appendix F)

Coding in languages other than ALGOL may appear in external procedures, provided that the entire procedure is in non-ALGOL code and that it adheres to the entry and exit conventions compiler output requires. The standard functions are examples of external procedures in non-ALGOL code.

APPENDIX A
DELIMITER CHARACTER SET

ALGOL Reference Language	SDS Typewriter Character	Printer Character
+	&	+
-	-	-
×	*	*
/	/	/
÷	\	\
↑	**	**
<	<	<
≤	=< or <=	same
=	=	=
≥	=> or >=	same
>	>	>
≠	<> or ><	same
≡	==	==
>	>>	>>
V	'OR'	'OR'
∧	'AND'	'AND'
¬	'NOT'	'NOT'
,	,	,
.	.	.
10	△	△
:	:	:
;	;	;
:=	:=	:=
( )	( ) or [ ]	[ ]
[ ]	( ) or [ ]	[ ]
' '	" "	" "

Notes

1. On output, (is converted to [and) is converted to].
2. Same: the characters in the source program will be printed.
3. The final character of each compilation is printed as ǂ .

APPENDIX B
RESERVED LEXICON WORDS

'AND'	'FOR'	'REAL'
'ARRAY'	'FORMAT'	'STEP'
'BEGIN'	'GO TO'	'STRING'
'BOOLEAN'	'IF'	'SWITCH'
'COMMENT'	'INTEGER'	'THEN'
'DO'	'LABEL'	'TRUE'
'ELSE'	'NOT'	'UNTIL'
'END'	'OR'	'VALUE'
'EXTERNAL'	'OWN'	'WHILE'
'FALSE'	'PROCEDURE'	

APPENDIX C
STANDARD FUNCTIONS AND PROCEDURES

Function	Description	Number of Arguments	Type of Result
ABS	absolute value	1	real
SIGN	+1, 0, or -1	1	integer
SQRT	square root	1	real
SIN	sine	1	real
COS	cosine	1	real
ARCTAN	arctangent	2	real
LN	logarithm	1	real
EXP	exponential	1	real
MIN	minimum	2	real
MAX	maximum	2	real
MOD	modulo	2	real
Procedures			
INPUT	initiate input	2	none
OUTPUT	initiate output	2	none
IO	transmit variables	variable	none
ENDIO	terminate input/output	none	none

INPUT and OUTPUT require one arithmetic expression and one designational expression as arguments.

Arguments of the other procedures and functions are arithmetic; i.e., either real or integer.

APPENDIX D
NOTES TO USERS OF THE ALGOL 60 COMPILERS

Procedures may reference themselves.

'PROCEDURE' P(X,4);...P(M,N)...;

The bounds of arrays are optionally signed integer constants.

Arithmetic OWN variables are automatically initialized to zero. Boolean OWN variables are automatically initialized to 'FALSE'. No other initialization is provided.

A GO TO that allows an undefined label (that is, in a switch) is left undefined.

The following features are included only in the expanded (ALGOL 60-8) system:

All identifiers may be used prior to declaration.

Dynamic array bounds are allowed.

Procedures can be called recursively with a call by value OWN array as an argument.

The following restrictions apply only to the basic (ALGOL 60-4) system:

Each identifier declared must be declared before any reference is made to it. Specifically, variables and arrays must be declared before being referenced. Switches must be fully declared before being referenced. The example below would be incorrect:

'SWITCH' S:=...S(N)...

Labels may be declared after being referenced; however, in this case the identifier representing the label may not be within the scope of another declaration of it.

Identifiers whose first eight characters are identical are treated as identical in the small system.

APPENDIX E
NOTES TO USERS OF THE ALGOL 60-4 EXECUTORS

Programs containing dynamic array bounds or arrays as parameters called by value cannot be executed using the smaller executors.

For this purpose a dynamic array is defined as an array any of whose bounds is other than an integer number as defined in Section II paragraph 2.5.1.

Input/output is restricted to paper tape, typewriter, and punched cards (code = 0, 1 or 2).

APPENDIX F
NON-ALGOL CODE

It may be necessary or desirable to write parts of an ALGOL program in a non-ALGOL language for reasons of efficiency, convenience, or extending the language. Machine language subroutines run faster than ALGOL subroutines since neither the command nor the address of an instruction needs to be interpreted. Subroutines may be conveniently included in an ALGOL program without the need for reprogramming. The ALGOL compiler has no facility for performing operations like packing and unpacking partial word data, input/output of octal data and string manipulation, for example. Including subroutines to perform these functions extends the language.

Non-ALGOL procedures are ordinarily coded in SYMBOL or META-SYMBOL. The ALGOL Loader incorporates all the features in the MONARCH Loader and is thus equipped to load SYMBOL or META-SYMBOL output.

REFERENCES TO NON-ALGOL PROCEDURES

Non-ALGOL procedures may be referenced by the ALGOL section of the program as though they were coded in ALGOL. They may be referenced by function designators or by procedure statements. The external declarations for them follow the rules for other external procedures. The values returned by function designators contribute to the arithmetic expression of which they are a part in the usual way. Parameters of non-ALGOL procedures may be as general as those of ALGOL procedures, and parameters may be called by name or by value. (Remember that the call procedure determines which parameters are called by name and which by value.)

EXECUTION LANGUAGE AND MACHINE LANGUAGE

Every non-ALGOL procedure must begin and end with several memory words corresponding to those generated by the compiler for ALGOL procedures. On entrance to all procedures the executor is processing and interpreting instructions in the execution language described in Appendix A of the ALGOL Technical Manual (publication 90 06 99A). Within the body of a non-ALGOL procedure, the programmer may switch freely back and forth between execution language and machine language provided that he executes the proper transitional commands. For example, a mixture of the two languages is ordinarily used in referencing formal parameters.

USE OF EXECUTION LANGUAGE IN SYMBOL AND META-SYMBOL

An illustration of one way the ALGOL execution language may be convenienily used through SYMBOL and META-SYMBOL follows. Two of the execution language formats, the instruction and the descriptor, are in common use. Instructions may be defined in both assemblers by means of OPD directives.

```
BPRO        OPD     04100000
              .
              .
              .
$SIN        BPRO    START, 0
```

The OPD directive defines the command (Begin Procedure) as equivalent to octal code 41 in the command field (commands and their codes are found in the ALGOL Technical Manual, Appendix A, Section IV). Thereafter, in coding, the execution language command and address correspond to the SYMBOL/META-SYMBOL command and address; the execution language flag (0 above) is written in the index field.

To define the descriptor format, a FORM directive is recommended:

```
DESC        FORM    3, 2, 4, 15
              .
              .
              .
X           DESC    2, 1, 0, BLOCK
```

The descriptor shown is of type 2 (call by name). It describes X, a real array (1 = real, 0 = no dimensions) in the block beginning at the symbol BLOCK. The language may be made more readable by defining additional symbols.

```
NAME        EQU     2
VALUE       EQU     4
INTEGER     EQU     0
REAL        EQU     1
BOOLEAN     EQU     2
VARIABLE    EQU     0
VECTOR      EQU     1
ARRAY       EQU     2
              .
              .
              .
X           DESC    NAME, REAL, VARIABLE,
                    BLOCK
```

In the examples below it is assumed that the following definitions have been presented to the assembler:

```
*ALGOL EXECUTION LANGUAGE COMMANDS

LOAD        OPD     00100000
BPRO        OPD     04100000
EPRO        OPD     04200000
EXIT        OPD     05100000

*DESCRIPTOR DEFINITIONS

DESC        FORM    3,2,4,15
NAME        EQU     2
VALUE       EQU     4
INTEG       EQU     0
REAL        EQU     1
VARI        EQU     0
```

Several quantities in the executor are needed by non-ALGOL procedures. They are

A — the 2-word pseudo-accumulator

ATYPE — an integer representing the type of the accumulator:

0 = INTEGER, 1 = REAL,
2 = BOOLEAN

EXEC — the entrance to the executor

The following definitions, which apply to all ALGOL versions, are also assumed in the examples below:

```
A      EQU   0160
ATYPE  EQU   0167
EXEC   EQU   0360
```

CODING NON-ALGOL PROCEDURES

For coding of non-ALGOL procedures the skeleton of the procedure is

```
$NAME  BPRO  START    the address of the body
       PZE   END      the address of the end
       PZE   NUMPAR   the number of parameters
       PZE   0
X      DESC  --
Y      DESC  --
       .
       .
START  LOAD  X,4
       EXIT  $+1      exits execution language

                      (the first actual param-
       .              eter is now in cells A and
       .              A + 1; its type is in
                      ATYPE.  Assembly code
                      is written here to save,
                      test, or operate on this
                      parameter.)

       BRM   EXEC     enters execution language
       LOAD  Y,4
       EXIT  $ + 1
       .              (the body of the pro-
       .              cedure goes here)
       BRM   EXEC
END    EPRO
       END
```

The programmer should note the following:

1. The name of the procedure is external and applies to the BPRO command.
2. The BPRO command is followed immediately by three special words and the parameter descriptors.
3. The instruction "EXIT" changes from interpretive mode to machine language and branches to the address given. The instruction "BRM EXEC" reenters the interpretive mode at the following instruction.
4. At exit the program must be in the interpretive mode. If the procedure is called as a function, A must contain the function's value and ATYPE must contain the type of the value.
5. The body of the procedure may make use of programmed operators (on 900 Series Computers), the ALGOL error subroutines, and other features of the executor. The ALGOL library furnishes examples of non-ALGOL procedures.

INDEX

- A -

Actual-Formal correspondence, II 4.7.4
Alphabetic Information, transmission of, II 4.8.3.2; II 4.8.4; also see A, H, X, and $ (dollar) Specifications.
AND, I 3.2.2; II 2.3; II 3.4.5
Arguments, II 3.2.5; also see Parameters.
Arithmetic Expressions, I 3.1; II 3.3; II 4.6.4.1
  elements, I 3.1.1
  operators, I 3.1.2; II 3.3.1; II 3.3.4
Arithmetics of REAL quantities, II 3.3.6
ARRAY, I 4.4.2; II 2.3; II 5.2.1; II 5.4.1
Array Declarations, II 5.2
Arrays, I 2; I 2.5; II 3.2.5; II 5.2
A Specification, II 4.8.3.2
Assignment Statement, I 2.1; I 4; I 4.1; II 1; II 3.2.5; II 4.2

- B -

Basic Concepts, II 2
BEGIN, I 4.9; I 4.10; II 1; II 2.3; II 4.1.1; II 5
Behavior of Variables under Recursion, II 5.4.6.3
Binary Information, transmission of, II 4.8.4
Blanks, I 1; I 2.3; II 2.3; II 2.6.3; II 4.8.3.2
Blocks, I 1.4; I 2; I 4; I 4.10; II 1; II 4.1; II 4.3.4; II 4.7.3.1
Body Replacement and Execution, II 4.7.3.3
BOOLEAN, I 2.1; I 4.4.2; II 2.3; II 2.8; II 4.2.4; II 5.1.1
Boolean Statements
  assignment statement, I 4; I 4.2
  elements, I 3.2.1
  expressions, II 3.3.1, II 3.4.
  operators, I 3.2.2; II 3.4.5
Bounds of Subscripts, II 5.2.3.1; II 5.2.4

- C -

Calls
  by name, I 4.4.1; II 4.7.3.2; II 5.4.5; II 5.4.6.3
  by value, I 4.4.1; II 3.2.5; II 4.7.3.1; II 4.7.5.4; II 5.4.5
COMMENT, I 1.2; II 2.3
Comments, I 1; I 1.2; II 2.2.2
Compilation, I 5; II 5.5.4
Compound Statement, I 4; I 4.9; II 4.1; II 4.3.4
Conditional Statement, I 4; I 4.7; II 4.5; II 4.5.3.2; II 4.5.4
Constants, I 2; I 2.2
Controlled Variables, Value of, II 4.6.5
Conversion of Data, II 4.8.3.1

- D -

$ Specification, II 4.8.3.2
Data, Conversion of, II 4.8.3.1
Declarations, I 1; I 1.3; I 2; II 1; II 5; II 5.2; II 5.2.6; II 5.3; II 5.4; II 5.5
Definition of Variables, I 2.1
Delimiters, II 2.3; II 4.7.6
Designational Expressions, I 3.3; II 3.5; II 4.3.3; II 4.4; also see Labels.
Digits, II 2.2.1
Dimensions, II 5.2.3.2
DO, I 1.1; II 2.3
Dummy Statements, I 4.11; II 4.4

- E -

ELSE, I 1.2; II 2.3; II 3.3.3; II 4.5.1
END, I 1.2; I 4.9; II 1; II 2.3; II 4.1.1; II 5
ENDIO, II 4.8.3; II 4.8.4
ENTIER, II 3.2.5; II 4.2.4
Equivalent (≡), II 3.4.5
E Specification, II 4.8.3.1
Evaluation of Switch Lists, II 5.3.4
Execution, I 1.4; I 2.8; II 4; II 4.7.3.3
Expressions, I 3; II 3; II 4.7.5.2
  Arithmetic, I 3.1
  Boolean, I 3.2
  Designational, I 3.3
EXTERNAL, I 2.8; I 5; II 5.5.1
External Procedure Declaration, II 5.5

- F -

FALSE, I 2.2; I 3.2.1; II 2.2.2; II 3.4.5; II 5.1.3
FOR, I 2.1; I 4; I 4.8; I 4.9; II 2.3; II 4.6; II 4.6.6
FOR List Elements, II 4.6.4
Formal Parameters, I 4.4.2; II 4.7.4
FORMAT, II 4.8
Format Statement, I 4; I 4.5; II 4.8
F Specification, II 4.8.3.1
Function Designators, I 4.4; II 3.2
Function Table, II 3.4.5

- G -

GO TO, I 1.1; I 4; I 4.3; II 2.3; II 4; II 4.3; II 4.6.6; II 5

- H -

H Specification, II 4.8.3.2

- I -

Identifiers, I 2; II 2; II 2.1; II 2.4; II 4.1.3; II 4.7.5.3; II 5
Identity of Subscripted Variables, II 5.2.5
IF Statement, I 4; I 4.6; I 4.9; II 2.3; II 3.3.1; II 3.3.3; II 4.5.1; II 4.5.3.1
Implies, II 3.4.5
Influence of Scopes, II 5.3.5
Initialization of OWN Variables, II 5.1.4
INPUT, I 4.5; II 4.8.3; II 4.8.4
Input/Output, II 4.8.4
INTEGER, I 2.1; I 4.4.2; II 2.3; II 2.5.4; II 2.8; II 3.1.4.2; II 3.2.5; II 3.3.4; II 4.2.4; II 5.1.1; II 5.4.5
Integer Values, I 2.2; II 3.1.4.2; II 3.3.6
IO, II 4.8.4
I Specification, II 4.8.3.1

- K -

Kinds of Recursive Calls, II 5.4.6.2

- L -

LABEL, I 4.4.2; II 2.3; II 5.4.1
Labels, I 2; I 2.6; II 1; II 4; II 4.1.3; II 4.3.4; also see Designational Expressions.
Language, Structure of, II 1
Letters, I 1; II 2.1
Lexicon Words, I 1.1; Appendix B
Logical Values, II 2.2.2
Lower Bounds, II 5.2.4

- N -

Named Statements, I 2.6
Name Replacement, II 4.7.3.2
Non-ALGOL Code, II 5.5.5; Appendix F
NOT, I 3.2.2; II 2.3; II 3.4.5
Numbers, I 1; II 2; II 2.5
  types of, II 2.8
  value of, II 2.8

- O -

Operators and Types, II 3.3.4
OR, I 3.2.2; II 2.3; II 3.4.5
OUTPUT, I 4.5; II 4.8.3; II 4.8.4
OWN, I 2.1; I 2.4; I 4.10.1; II 2.3; II 5; II 5.1.4; II 5.2.5; II 5.2.6; II 5.4.6.3
OWN Array Declaration, II 5.2.6

- P -

Parameter Delimiters, II 4.7.6
Parameters, Specification of Formal, I 4.4.2; II 4.7.3.1; II 4.7.4; II 4.7.5.2; II 5.4.5
Parentheses for Grouping; I 3.1.2; II 3.3.3; II 3.3.5.2; II 4.8.3
Precedence
  arithmetic operations, I 3.1.2; II 3.3.5
  Boolean operations, I 3.2.2; II 3.4.6
PROCEDURE, I 4.4.2; II 2.3; II 5.4.1
Procedure Declarations, II 5.4; II 5.5
Procedures, I 2; I 2.8; Appendix C
Procedures, Recursive Calling of, II 5.4.6
Procedure Statement, I 4; I 4.4; II 4.7
Program Form, I 1; II 1
Program Organization, II 5.5.4

- Q -

Quantities, Kinds and Scope, II 2.7
Quotation Marks (Single), I 1.1; I 2.3

- R -

REAL, I 2.1; I 4.4.2; II 2.3; II 2.8; II 3.2.5; II 3.3.4; II 3.3.6; II 4.2.4; II 5.1.1, II 5.2.3.3; II 5.4.5
Real Values, I 2.2; II 3.3.6
Recursive Calling of Procedures, II 5.4.6
References, I 2
Relations, II 3.4.5
Replacement Operator, I 4.1
Restrictions on Procedures, II 4.7.5

- S -

Scale Factor, II 2.5.3; II 4.8.3.1
Single Quotation Marks, I 1.1; I 2.3
Space, II 2.6.3
Special Characters, I 1
Specification of Formal Parameters, I 4.4.2; II 4.7.3.1; II 4.7.4; II 4.7.5.2; II 5.4.5
Standard Functions, I 5.1; II 3.2.4; Appendix C
Statements, I 1; I 1.4; I 2.6; I 4; II 4
STEP, II 4.6.1
Step-Until Element, II 4.6.4.2
STRING, I 4.4.2; II 2.3; II 5.4.1
Strings, I 2; I 2.3; II 2; II 2.1; II 2.6; II 4.7.5.1
Structure of the Language, II 1
Subscript Bounds, II 5.2.3.1
Subscripted Variables, II 5.2.5
Subscripted Expression, II 3.5.4
Subscripts, I 2.5; II 3.1.4; II 3.2.5; II 3.4.5
SWITCH, I 4.4.2; II 2.3; II 5.3.1; II 5.4.1
Switch Declarations, II 5.3
Switches, I 2; I 2.7; II 4.3.5
Switch List, II 5.3.4
Symbols, Basic, II 2

- T -

THEN, II 2.3; II 3.3.1; II 3.3.3; II 4.5.1
Transfers of Types, II 3.2.5
Transmission of Alphanumeric Data, II 4.8.3.2
TRUE, I 2.2; I 3.2.1; II 2.2.2; II 3.3.3; II 3.4.5; II 5.1.3
Truth Table, II 3.4.5
Tape Declarations, II 3.4.4; II 5.1
Types of Arrays, II 5.2.3.3
Types of Variables, I 2.1; II 2.5.4; II 3.1.3; II 3.3.4; II 3.4.4; II 4.2.4

- U -

UNTIL, II 2.3; II 4.6.1

- V -

VALUE, I 4.4.1; I 4.4.2; II 2.3; II 5.4.1
Value Assignment, II 4.7.3.1
Value of Variables, I 2.1; II 4.6.5; II 5.4.6.3
Values and Types of Numbers, II 2.8
Values of Function Designators, II 5.4.4
Variables, I 2; I 2.4; II 3.1; II 5.4.6.3
Variables, Subscripted, II 5.2.5

- W -

WHILE, II 2.3; II 4.6.1; II 4.6.4.3

- X -

X Specification, II 4.8.3.2