This manual is published by the Univac Division of Sperry Rand Corporation in loose leaf format. This format provides a rapid and complete means of keeping recipients apprised of UNIVAC® Systems developments. The information presented herein may not reflect the current status of the programming effort. For the current status of the programming, contact your local Univac Representative.

The Univac Division will issue updating packages, utilizing primarily a page-for-page or unit replacement technique. Such issuance will provide notification of software changes and refinements. The Univac Division reserves the right to make such additions, corrections, and/or deletions as, in the judgment of the Univac Division, are required by the development of its Systems.

UNIVAC is a registered trademark of Sperry Rand Corporation.
## CONTENTS

1. **INTRODUCTION**  
   1.1. GENERAL  
   1.2. THE ALGOL COMPILER  
   1.3. ALGOL 60 AND UNIVAC 1108 ALGOL  
      1.3.1. Extensions to ALGOL 60  
      1.3.2. Deviations from ALGOL 60  
   1.4. LANGUAGE CONVENTIONS  

2. **ELEMENTS OF THE LANGUAGE**  
   2.1. THE CHARACTER SET  
   2.2. IDENTIFIERS  
   2.3. CONSTANTS  
      2.3.1. Integer Constants  
      2.3.2. Real Constants  
      2.3.3. Double Precision Constants  
      2.3.4. Complex Constants  
      2.3.5. Boolean Constants  
      2.3.6. String Constants  

3. **DECLARATIONS**  
   3.1. GENERAL  
   3.2. TYPE DECLARATIONS  
   3.3. ARRAY DECLARATIONS  
   3.4. STRING DECLARATIONS  
      3.4.1. String Arrays  
   3.5. OWN DECLARATIONS  
   3.6. DEFAULT DECLARATIONS  
   3.7. THE COMMENT  
   3.8. FORMAT, LIST, SWITCH, PROCEDURE, LOCAL
## 4. Expressions

### 4.1. General

### 4.2. Arithmetic Expressions
- 4.2.1. Ordering Rules for Operations
- 4.2.2. Hierarchy of Operand Types
- 4.2.3. Operands of Arithmetic Expressions
  - 4.2.3.1. Subscripted Variables
  - 4.2.3.2. Function Designators

### 4.3. Boolean Expressions
- 4.3.1. Relational Expressions
- 4.3.2. Boolean Operators
- 4.3.3. Precedence of Boolean Operations

### 4.4. String Expressions

### 4.5. Designational Expressions

### 4.6. Conditional Expressions

## 5. Statements

### 5.1. General

### 5.2. Compound Statements

### 5.3. Assignment Statements
- 5.3.1. String Assignment Statements

### 5.4. Multiple Assignment Statements

### 5.5. Statement Labels

### 5.6. Punctuation

### 5.7. Dummy Statements

## 6. Control Statements

### 6.1. General

### 6.2. Unconditional Control Statements
- 6.2.1. The GO TO Statement
- 6.2.2. The SWITCH

### 6.3. Conditional Control Statements

### 6.4. Iterative Control Statements – The FOR Statement
- 6.4.1. Simple List Element
- 6.4.2. STEP – UNTIL List Element
- 6.4.3. WHILE List
- 6.4.4. Termination of FOR Statements
7. PROCEDURES

7.1. INTRODUCTION

7.2. VALUE ASSIGNMENT (CALL BY VALUE) AND NAME REPLACEMENT (CALL BY NAME)

7.3. SPECIFICATIONS

7.4. FUNCTION PROCEDURES

7.5. RECURSIVE PROCEDURES

7.6. EXTERNAL PROCEDURES
    7.6.1. ALGOL External Procedures
    7.6.2. FORTRAN Subprograms
    7.6.3. Machine Language Procedures

8. BLOCK STRUCTURE

8.1. GENERAL

8.2. BLOCKS

8.3. LOCAL AND GLOBAL IDENTIFIERS

8.4. THE LOCAL DECLARATION

9. INPUT/OUTPUT

9.1. GENERAL

9.2. FREE-FORMAT OUTPUT ON PRINTER AND CARD PUNCH

9.3. FREE-FORMAT INPUT FROM CARDS

9.4. LIST PARAMETERS – THE LIST DECLARATION

9.5. FORMATTED OUTPUT – THE FORMAT DECLARATION
    9.5.1. Nonediting Codes
    9.5.2. Editing Codes
    9.5.3. Repetition of Editing Codes
        9.5.3.1. Simple Repetition
        9.5.3.2. Variable Repetition
        9.5.3.3. Indefinite Repetition
    9.5.4. FORMATTED INPUT

9.6. FILE HANDLING
    9.7.1. Sequential Files
    9.7.2. Random Access Files
    9.7.3. Alternate Symbionts
9.8. OTHER DIRECTIVES
9.8.1. Rew ind 9-18
9.8.2. Mod ifiers and Position 9-18
9.8.2.1. Mod ifiers 9-18
9.8.2.2. Position 9-19
9.8.3. Labels 9-19
9.8.3.1. Position Procedure 9-19
9.8.3.2. Read Procedure 9-29
9.8.3.3. Write Procedure 9-21
9.8.3.4. Margin Procedure 9-21

10. OPERATION
10.1. SOURCE CARD FORMAT 10-1
10.2. OPERATING INSTRUCTIONS 10-1

APPENDICES

A. BASIC SYMBOLS AND THEIR CARD CODES A-1 to A-2
B. STANDARD PROCEDURES AND TRANSFER FUNCTIONS B-1 to B-4
C. ERROR MESSAGES
   C1. COMPILATION ERRORS C-1
   C2. RUN-TIME ERRORS C-4
D. EXAMPLES OF PROGRAMS D-1 to D-6
E. JENSEN'S DEVICE AND INDIRECT RECURSIVITY E-1 to E-1
F. FILE-HANDLING PROCEDURES
   The information in this section has been
   incorporated in Section 9.
G. SYNTAX CHARTS G-1 to G-38
   G1. GENERAL G-1
   G2. PROGRAM G-2
   G3. DECLARATION G-3
      G3.1. Type Declaration G-4
      G3.2. Array Declaration G-5
      G3.3. String Declaration G-6
      G3.4. String Array Declaration G-7
      G3.5. Switch Declaration G-8
      G3.6. External Procedure Declaration G-9
      G3.7. Procedure Declaration G-10
      G3.8. Local Declaration G-11
      G3.9. List Declaration G-12
      G3.10. Format Declaration G-13

Other trademarks of Sperry Rand Corporation appearing in the text of this
publication are: UNISERVO
                              FASTRAND
G4. STATEMENT
  G4.1. Block
  G4.2. Compound Statement
  G4.3. Assignment Statement
  G4.4. GO TO Statement
  G4.5. Conditional Statement
  G4.6. FOR Statement
  G4.7. Dummy Statement
  G4.8. Procedure Statement

G5. EXPRESSION
  G5.1. Variable
  G5.2. Function Designator
  G5.3. Arithmetic Expression
  G5.4. Boolean Expression
  G5.5. Designational Expression

G6. BASIC ELEMENTS
  G6.1. Identifier, Letter, Letter String, Digit
  G6.2. Number
  G6.3. String, Logical Value
  G6.4. Delimeter

G7. INPUT/OUTPUT PROCEDURES
  G7.1. Input Procedure Statement
  G7.2. Output Procedure Statement
  G7.3. Position Procedure Statement
  G7.4. Rewind Procedure Statement
  G7.5. Summary of Format Codes
  G7.6. Grouping of Format Codes

FIGURES
  8–1. Local and Global Identifiers

TABLES
  9–1. Output Nonediting Codes
  9–2. Output Editing Codes
  9–3. Input Editing Codes
UPDATING PACKAGE "A"

UNIVAC 1108 Multi-Processor System P.I.E. Bulletin 14, UP-4103.14, announces the release and availability of Updating Package "A" for the "UNIVAC 1108 Multi-Processor System ALGOL Programmers Reference Manual," UP-7544, 46 pages plus 1 Updating Summary Sheet. This material should be utilized in the following manner:

<table>
<thead>
<tr>
<th>SECTION</th>
<th>DESTROY FORMER PAGES NUMBERED</th>
<th>FILE NEW PAGES NUMBERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>3 and 4</td>
<td>3 Rev. 1 and 4 Rev. 1</td>
</tr>
<tr>
<td>Section 3</td>
<td>1 and 2</td>
<td>1* and 2 Rev. 1</td>
</tr>
<tr>
<td></td>
<td>3 and 4</td>
<td>3 Rev. 1 and 4*</td>
</tr>
<tr>
<td></td>
<td>1 and 2</td>
<td>1* and 2 Rev. 1</td>
</tr>
<tr>
<td>Section 5</td>
<td>5</td>
<td>5 Rev. 1</td>
</tr>
<tr>
<td>Section 6</td>
<td>3 and 4</td>
<td>3 Rev. 1 and 4*</td>
</tr>
<tr>
<td></td>
<td>3 and 4</td>
<td>3* and 4 Rev. 1</td>
</tr>
<tr>
<td></td>
<td>9 and 10</td>
<td>9* and 10 Rev. 1</td>
</tr>
<tr>
<td>Section 9</td>
<td>3 and 4</td>
<td>3* and 4 Rev. 1</td>
</tr>
<tr>
<td></td>
<td>1 and 2</td>
<td>1 Rev. 1 and 2*</td>
</tr>
<tr>
<td></td>
<td>3 and 4</td>
<td>3* and 4 Rev. 1</td>
</tr>
<tr>
<td></td>
<td>5 thru 23</td>
<td>5 Rev. 1 thru 22 Rev. 1</td>
</tr>
<tr>
<td>Section 10</td>
<td>1 and 2</td>
<td>1 Rev. 1</td>
</tr>
<tr>
<td>Appendix B</td>
<td>1 and 2</td>
<td>1 Rev. 1 and 2 Rev. 1</td>
</tr>
<tr>
<td>Appendix F</td>
<td>1 thru 6</td>
<td>N.A.*</td>
</tr>
<tr>
<td>Appendix G</td>
<td>33 thru 36</td>
<td>33 Rev. 1 thru 36 Rev. 1</td>
</tr>
</tbody>
</table>

* These pages, backups of revised pages, remain unchanged.
† Information in Appendix F has been greatly expanded and incorporated into Section 9.
1. INTRODUCTION

1.1. GENERAL

This manual describes the ALGOL language for the UNIVAC 1108 System. The basis for this language is the "Revised Report on the Algorithmic Language, ALGOL 60" (Communications of the ACM, Vol. 6, January 1963, 1-17). This implementation of ALGOL is very close to that of the report. Its one significant omission is the omission of dynamic own arrays. Some of its more significant additions include three new data types (STRING, COMPLEX, REAL 2), and default declarations. Provision is made for inclusion of procedures written in assembly language or FORTRAN V.

This manual is intended as an introduction to ALGOL 60 and as a reference manual in the use of UNIVAC 1108 ALGOL and is not intended as an exhaustive, self-contained description of ALGOL 60. The text consists principally of definitions and rules for writing ALGOL programs, examples of these rules, and some sample programs.

A set of appendices includes special sections on file-handling procedures, UNIVAC 1108 ALGOL syntax in chart form, and sample ALGOL programs; lists of basic symbols, library procedures, and diagnostic messages.

1.2. THE ALGOL COMPILER

The ALGOL compiler is a program which accepts statements expressed in ALGOL and produces programs for the UNIVAC 1108 System.

An ALGOL program is a sequence of statements written in ALGOL language. These are translated by the compiler into the language of the computer: machine language. The ALGOL statements are called the source code, and the translated statements are called the object code. The compiler itself is a program written in machine language and is called the UNIVAC 1108 ALGOL 60 Compiler. While translating the ALGOL statements, the compiler looks for errors in syntax (that is, for errors in the forms or construction of statements).

The compiler operates in two passes. The first pass scans the statements and does about 95 percent of the work required to produce the final object code. The second pass goes into operation immediately after all the statements have been scanned; it completes the remaining details of producing the object code. Upon successful compilation, the object code can be read into the main storage and executed. Activities that occur during compilation are sometimes referred to as compile-time activities; for instance, compile-time diagnostics. The execution phase is referred to as run-time.
1.3. ALGOL 60 AND UNIVAC 1108 ALGOL

There are several differences between ALGOL 60 as defined in the revised report and 1108 ALGOL 60. In that ALGOL is intended as a standard language and compatibility of programs between machines is becoming more and more important, those differences must be explicitly pointed out. They fall into two classes: extensions to ALGOL 60 and definition of things left undefined by the report; modifications or omission of ALGOL 60 entities.

1.3.1. Extensions to ALGOL 60

Extensions to ALGOL 60 include the following:

- **STRING** and **STRING ARRAY** variables enhance the value of ALGOL as a data processing language.
- New arithmetic types **COMPLEX** and **REAL 2** enhance the value of ALGOL to scientific users.
- **XOR**, an additional Boolean operator is provided.
- **EXTERNAL PROCEDURE** declarations are provided for convenience in programming large problems and for building libraries.
- I/O and other library procedures are provided and, related to them, are the **FORMAT** and **LIST** declarations.
- A compact form for **GO TO** and **FOR** statements is allowed.
- Variables are given values of zero or blank at the entrance to a block; thus initialization statements need not be made.
- The controlled variable of a **FOR** statement has a defined value when the statement is terminated by exhaustion of the **FOR** list.
- The **OTHERWISE** declaration or declaration by default is provided.
- The variables in a multiple assignment statement need not be the same type.
- Arguments of type **COMPLEX** and **REAL 2** are permitted for various standard functions.

1.3.2. Deviations from ALGOL 60

- Because of hardware requirements, identifiers are unique only to their first twelve characters and may contain no blanks; numbers may contain no blanks, and certain basic symbols are reserved identifiers (see Appendix A).
- **OWN** arrays are not dynamic.
- Numeric labels are not allowed.
- The comma is the only parameter delimiter allowed in a procedure call.
- A **LOCAL** declaration is required to resolve all forward references to identifiers.
- An integer raised to an integer power always produces a **REAL** value.
All the formal parameters of a procedure must be specified and must agree in type with the actual parameters.

These and other restrictions are covered in more detail in later sections of this manual.

1.4. LANGUAGE CONVENTIONS

ALGOL is described in terms of three languages in this manual: reference, publication, and hardware language.

The reference language is that which defines ALGOL in the ALGOL 60 Revised Report. It is computer independent and utilizes the basic ALGOL symbols to define the language syntax and semantics. Throughout the text, but sparingly, the syntax of 1108 ALGOL is described in terms of this reference language.

Example:

<identifier> := <letter> / <identifier><letter> / <identifier><digit>

(Read := as ‘is’ and / as ‘or’)

This says that an <identifier> is either a <letter> or an <identifier> followed by a <letter> or an <identifier> followed by a <digit>. Further discussion of identifiers is found in 2.2.

While having the advantage of compactness and precision, the formalism is not suitable as an introduction to ALGOL and so has been used only as a summary aid. Except for the formal definitions of the reference language, the hardware language (the language acceptable to the UNIVAC 1108) has been used throughout the text. All basic symbols which appear in the text, as well as all examples, are written in upper case letters. This is the form in which they appear in the hardware language.

For publication purposes, the boldface type delineates the basic UNIVAC 1108 ALGOL symbols. Transliteration rules for basic symbols are given in Appendix A.

Statements may be separated from each other by either the semicolon or the dollar sign. Because of keypunch limitations, the $ is commonly accepted and has been used in all examples throughout this manual.

The following symbols are considered equivalent:

' ' is equivalent to : (colon)

'=' is equivalent to := (replacement operator)
2. ELEMENTS OF THE LANGUAGE

2.1. THE CHARACTER SET

The ALGOL compiler employs a character set which is commonly available as a variant of the usual Hollerith code (FORTRAN H set) plus a few special UNIVAC 1108 characters. These are:

- **Letters**: A-Z
- **Digits**: 0-9
- **Special characters**: + - = ( ) $ / * . space
- **UNIVAC 1108 special characters**: &: < > '[ ]

In addition, some multiples of characters are given meaning as if they constituted a single character:

- `..` colon (interchangeable with :)
- `//` integer divide
- `**` exponentiation
- `&&` base 10 scale factor (double precision)
- `:=` replacement (instead of merely =)

A complete list of these characters and the transliteration rules from the ALGOL 60 report is given in Appendix A.

2.2. IDENTIFIERS

Identifiers are names that the programmer chooses to use to refer to the various things which make up a program - variables, labels, switches, formats, procedures, etc. Identifiers must begin with a letter and may be followed by any number of letters and/or digits. None of the special characters listed in 2.1 (including a space) may be used in an identifier. The compiler considers two identifiers identical if the first 12 characters are alike.

The following are all legitimate identifiers but the last two are not unique since their first 12 characters are identical.

- `X`  
- `SUMX`  
- `Y`  
- `ALTITUDE`

Some basic symbols have the same form as identifiers and are called RESERVED IDENTIFIERS (see Appendix A). These can never be used except in their intended context as basic symbols. For example, the word **BEGIN** can never be used as the name of a quantity because it has an inherent meaning within the language and cannot be redefined. On the other hand, several common arithmetic functions are available for use without being declared, but these names can be redefined as identifiers (see Appendix B). All identifiers, including reserved identifiers, must
be separated from each other in the source language by DELIMITERS. All the special characters listed in 2.1 are delimiters. The space is a rather unique delimiter in that a sequence of spaces is treated as one space.

2.3. CONSTANTS

Six types of constants may be used in the UNIVAC 1108 ALGOL source program language. They are integer, real (single precision, floating point), double precision, complex, Boolean, and string constants.

2.3.1. Integer Constants

Integers are whole numbers represented internally by 35 bits plus the sign. Range of an integer N (in magnitude) is from zero to \(2^{35} - 1\) inclusive (\(2^{35} - 1 = 34359738367\)).

If positive, the integer may be prefixed with a plus sign. If negative, it must be prefixed with a minus sign.

**EXAMPLES:**

\[
\begin{array}{ll}
0 & +23 \\
70 & 2222222222 \\
-204 & +0 \\
\end{array}
\]

2.3.2. Real Constants

A real constant is a string of eight or fewer digits with a decimal point. The point may precede, follow or be embedded within the digits. Internally it is represented as a floating point number with 9 bits for the sign of the number and exponent and 27 bits for the fraction. The plus sign is optional, but a negative sign must precede a negative constant. The magnitude ranges from approximately \(10^{-38}\) to \(10^{38}\) or it may be zero.

**EXAMPLES:**

\[
\begin{array}{ll}
3.1416 & 750.0 \\
0.0 & +1.7 \\
.645 & -20.4 \\
\end{array}
\]

If desired, a scale factor may be appended to a real constant to indicate that it is to be multiplied by the indicated power of 10. This scale factor is represented by an ampersand followed perhaps by a plus or minus sign and then by an integer. The integer specifies the power of 10 to be used and is limited to two digits.

- \(2.65\&6\) means \(2.65 \times 10^6\) or 2,650,000.
- \(-17.445\&-5\) means \(-17.445 \times 10^{-5}\) or \(-0.00017445\)

In addition a real constant may be written as an integer followed by a scale factor or a scale factor by itself may be used to signify a real constant.

- \(2\&-6\) means \(2.0\&-6\) or \(.000002\)
- \(7\&7\) means \(10^7\) or 10,000,000.0

2.3.3. Double Precision Constants

Double precision constants are used for double precision calculations. The magnitude ranges from approximately \(10^{-308}\) to \(10^{308}\) or it may be zero. The maximum number of digits is 18.
Double precision constants are differentiated from real constants by use of & for power of ten, or by inclusion of between 9 and 18 digits in the fixed point part. Double precision constants are ordinarily used only with variables of type REAL 2.

The following are acceptable double precision constants:

3.141592653589793
1.0496523456668-22
4.65588-4
1.082
480

2.3.4. Complex Constants

The general form of a complex constant is:

< R1, R2 >

where R1 and R2 are real or integer constants and where < and > are required.

Examples:

<1.0, 1.0> represents 1 + i
<7.08-2, -2> represents 0.07-2i
<0.0, 1.0> represents i

2.3.5. Boolean Constants

Only two Boolean constants are allowed:

TRUE
FALSE

2.3.6. String Constants

A string constant is a string of characters not containing a quote but enclosed by quotes. The maximum size of a string constant is 4095 characters.

Examples:

'STANDARD DEVIATION ='
'PRINCIPLE RATE PERIOD PAYMENT'
3. DECLARATIONS

3.1. GENERAL

An ALGOL program may be broken into logical segments called blocks which are complete and independent units. Their structure is discussed in Section 8. One important property of a block is that, at the beginning of the block, all of the local entities that are to be referenced inside the block must be declared. Declarations determine how the compiled program will treat certain of its elements; thus it is necessary to precede the use of an identifier with a declaration of type. An identifier may appear in only one declaration within a block; however a block may contain blocks within itself (as shown in 8.3). Any of these blocks may declare variables taking on names used in outer blocks, thus redefining them for the inner block.

3.2. TYPE DECLARATIONS

The type declaration defines the type of variable named by an identifier. This declaration specifies that all values which the identifier assumes must be of the designated type. The general form of type declaration is:

\[
\text{<type> <type list>}
\]

where \(<\text{type list}>\) is a list of identifiers separated by commas. Each declaration is terminated by a ; or $. The five possible type declarations are:

**INTEGER** – Integral values represented internally by 36 bits. The range of an integer (in magnitude) is zero thru \(2^{35} - 1\) inclusive.

**REAL** – Floating point numbers internally represented by 9 bits for sign of the number and the exponent and 27 bits for the fraction. The range of a real number (in magnitude) is \(10^{-38}\) to \(10^{38}\) and 0 with approximately 8 digits of precision. Any real quantity which is less than \(10^{-38}\) is represented by zero.

**REAL 2** – Floating point numbers internally represented by 12 bits for sign of the number and the exponent and 60 bits for fraction. The range of a REAL 2 number (in magnitude) is approximately \(10^{-308}\) to \(10^{308}\) and zero with approximately 18 digits of precision.

**COMPLEX** – Complex values of the form \(A + i*B\) where \(A\) and \(B\) are REAL numbers.

**BOOLEAN** – Truth values, TRUE or FALSE.
Examples of type declarations are:

```
INTEGER  I, J, K, COUNTER
REAL     X, Y, TEMP, VELOCITY
BOOLEAN  AJAX
COMPLEX  Z1, Z2, U, V
REAL 2   A, B, C
```

### 3.3. ARRAY DECLARATIONS

When declaring simple variables as described above, a different name must be used for each different variable being defined. The `ARRAY` declaration provides a means of referring to a collection of numbers that fall into a matrix or array by the use of a single identifier.

This `ARRAY` declaration specifies to the compiler the structure which is to be imposed on this collection. An array is a group or set of elements arranged so that each may be identified by its position within the group. The compiler considers all elements of array to be of the same type.

Arrays in this compiler are restricted to those of rectangular construction in n-dimensional space.

For example, the declaration `REAL ARRAY A(1:10)` defines an array `A` with ten elements which may be referenced by:

```
```

The general form of array declaration is

```
<type> ARRAY <array list> (<bound pair list>)
```

where `type` may be of any of the types given in 3.2. If `type` is omitted, it is assumed to be `REAL`. The array list specifies the names of the arrays. The bound-pair list consists of a bound pair for each array dimension. Each bound pair is of the form:

```
lower limit:upper limit
```

A complete array declaration for a single array is of the form:

```
ARRAY A(l1:u1, l2:u2, l3:u3, ..., ln:un).
```

Where `l`'s represent lower bounds and `u`'s represent upper bounds. Either or both of the bounds may be negative, but `l_i \leq u_i`.

For example:

```
INTEGER ARRAY I(0:4, 1:3)
```
defines an array composed of five rows and three columns of integers as follows:

\[
\begin{array}{ccc}
I(0,1) & I(0,2) & I(0,3) \\
I(1,1) & I(1,2) & I(1,3) \\
I(2,1) & I(2,2) & I(2,3) \\
I(3,1) & I(3,2) & I(3,3) \\
I(4,1) & I(4,2) & I(4,3)
\end{array}
\]

In the previous declaration, the parts 0:4 and 1:3 are called bound pairs, and each set of them defines a subscript position. The first digit of the bound pair specifies the lowest possible value for that subscript position, and the second specifies the highest.

An element of array \( I \) is referenced by the identifier \( I \) followed by a subscript list enclosed in parentheses (see 4.2.3.1). Since the lower bound of the first subscript position is 0, \( I(3,2) \) refers to the element in the fourth row and second column of array \( I \). There is no limit to the number of subscript positions an array may have. However, declarations like

\[
\text{REAL ARRAY } A(6:5)
\]

are not allowed, since the lower bound must not be greater than the upper bound. This declaration would result in an execution-time error.

Array identifiers of the same type, separated by commas, may be included in one declaration:

\[
\text{BOOLEAN ARRAY } A(1:2), B(1:10,14:22), C(-2:7,0:100)
\]

If two or more arrays are of the same type and same size, they may be listed sequentially with the dimension specification after the last array identifier in the group.

\[
\text{COMPLEX ARRAY COM, COMI, DECOM, COMCONJ(3:10)}
\]

This declaration defines four one-dimensional arrays. Each consists of eight complex numbers and the subscripts of the elements range from three to ten.

One of the most important feature of ALGOL is that the expressions for the bound pairs need not be constants; they can be any expression referring to non-local variables and constants.

Example:

\[
\text{REAL ARRAY } A(1:N, I/2:ENTIER(X), 0:TIMEMAX), B(DP1, DP2(-INFINITY: INFINITY), DP1, DP2(-INFINITY: INFINITY)}
\]

The size of these arrays depends on the values of \( N \), \( I \), \( X \), \( TIMEMAX \), and \( INFINITY \). Therefore, the size varies from one execution to another. Because of this, the actual storage cells for the array are allocated during execution each time the block (in which the array declaration occurs) is entered; i.e., at the place the array is declared. This is called 'dynamic' storage allocation. All the variables in a program except own variables (see 3.5) are allocated in storage in this way. Section 8 explains the process of allocating variable storage for ALGOL. Note that the dynamic allocation concept cannot be used in the outermost block (i.e., the bound pair list may contain only constants in the array declarations in the outermost block).
3.4. **STRING DECLARATIONS**

The **STRING** declaration provides a means of referring to a collection of alphanumeric characters in Field data code by the use of a single identifier, and at the same time specifies to the compiler the structure which is to be imposed on this collection. The string declaration defines the name and length of the string:

**STRING S(80)** $\$

Strings may have substrings, either named or unnamed:

**STRING S(L(40),R(40))** $\$

defines a string S as having a length of 80 characters with the first 40 characters being a string L and the second 40 a string R.

**STRING S(6,S7(10),4,Q(8,Q9(1),2))** $\$

\[ \begin{array}{cccc}
  6 & 10 & 4 & 8 \\
   &   &   & 1 \\
   &   &   & 2 \\
\end{array} \]

The above declaration defines the strings S, S7, and Q and Q9. It also gives their relative position since 6, 4, and 2 are unnamed substrings. The expression for the length of a string must be positive and less than 4096. Strings, like simple variables and arrays, may be declared with an identifier list:

**STRING CARD (80), LINE(132), ITEM(CODE(DEPT(2), SECTION(8)), 5*NAME(30), RATE(5), TIME(5), GROSS(10), NET(10))** $\$

The string CARD holds 80 characters corresponding to a card image. Correspondingly, the string LINE holds one print line image. The string ITEM, on the other hand, has the somewhat complicated structure shown below:

<table>
<thead>
<tr>
<th>DEPT(2)</th>
<th>SECTION(8)</th>
<th>CODE(10)</th>
<th>NAME(30)</th>
<th>Rate(5)</th>
<th>Time(5)</th>
<th>Gross(10)</th>
<th>Net(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ITEM(75)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ITEM has 75 characters partitioned into the strings CODE, NAME, RATE, TIME, GROSS, and NET. In addition, the string CODE of 10 characters is partitioned into the strings DEPT and SECTION. Thus

**ITEM(8) ≡ CODE(8) ≡ SECTION(6).**

3.4.1. String Arrays

A combination of the string and array declarations defines a quantity known as a string array. A string array is an array whose elements are strings. The form of declaration is:

**STRING ARRAY S(<string part> : <array part>)**
where <string part> specifies the length of each element of S (and also defines any substrings just like a string declaration) and <array part> is the list of bound pairs just as for a simple array (see 3.3).

Example:

```
STRING ARRAY S(L(40),R(40):1:10, 1:10) $
```

defines a two-dimensional array S with ten rows and ten columns. Each element of the array is a string of 80 characters. Furthermore each string consists of substrings L and R each 40 characters long. Referencing of substrings is discussed in 4.4.

### 3.5. OWN DECLARATIONS

Whenever a block is entered, the simple variables and arrays that are declared within that block are given the value zero, and strings are given the value (blank) in each of their character positions. The additional symbol OWN in front of any one of these types of declarations changes this initialization in the following way: the first time the block is entered they are given initial values as above. In subsequent entrances to the block they have the same value as they had on the last exit from the block.

Examples:

```
BEGIN INTEGER I $
  REAL FX, FY $
  OWN BOOLEAN ALPHA, BETA $
  OWN REAL ARRAY DEV (1:10, 1:10) $

In general all declarations allowed in 3.2, 3.3, and 3.4 of this chapter are also permitted as OWN declarations. The exception to this rule is that the length of a string or the length of any of the subscript positions of an array does not change after the first entrance to the block. Thus, if a block begins by:

```
BEGIN OWN ARRAY A(0:N)
```

and N has the value six (elements are numbered zero through six), the length of A remains seven throughout the program even if N has a different value at the next entrance to the block.

### 3.6. DEFAULT DECLARATIONS

The OTHERWISE declaration allows the programmer to specify that all simple variables (those without subscripts), unnamed in a type declaration are assumed to be of a given type.

```
BEGIN REAL X, FX, FPX $
  INTEGER OTHERWISE
```
means that any other simple variables besides $X$, $FX$, $FPX$ that are encountered in this block are to be integers. The `OTHERWISE` declaration may not be used in connection with an array or string. A hazard of this declaration is that it carries the danger that 'new' variables may be created unintentionally and not noticed.

Example:

```algor
BEGIN INTEGER OTHERWISE $
  \text{ROOMBOOM} = 2$
  \text{AEN} = 4$
\text{BOOMBOOM} = ((\text{BOOMBOOM}+\text{AEN})*\text{BOOMBOOM}+\text{AEN})*\text{BOOMBOOM}
```

The variable `BOOMBOM` has crept into the calculation when `BOOMBOOM` was the proper one. Therefore the `OTHERWISE` declaration must be used with care. Another type declaration may follow the `OTHERWISE` declaration.

### 3.7. THE COMMENT

The `COMMENT` allows the programmer to include such things as clarifying remarks and identifying symbols in the printed compilation. A comment may serve any purpose the programmer desires once it is ignored by the compiler.

Two commonly used forms are:

- `BEGIN COMMENT <any sequence not containing ; or $ >;` ; `COMMENT <any sequence not containing ; or $ >;`

Example:

```algor
BEGIN COMMENT SAMPLE PROGRAM USING UNIVAC 1108 ALGOL $
```

Any characters following an `END` and preceding another `END, ELSE,` or a semicolon are also treated as comments.

Examples:

```algor
END OF INNER LOOP END OF OUTER LOOP $
END THIS TERMINATES THE THEN PART ELSE 
END OF HEAT TRANSFER PROGRAM $
```

### 3.8. FORMAT, LIST, SWITCH, PROCEDURE, LOCAL

The other declarations are of a more complicated nature and appear in other parts of the manual. `FORMAT` and `LIST` are concerned only with input/output and are discussed in Section 9. Procedures are discussed in Section 7 and switches in Section 5. The `LOCAL` declaration is added to the language to allow faster (one pass) translation into object code. It is discussed in Section 8.
4. EXPRESSIONS

4.1. GENERAL

An expression is a rule for computing a value. There are four kinds of expressions: arithmetic, Boolean, string, and designational. Expressions are composed of operands, operators and parentheses. Operands are constants, variables, function designators, or other expressions. Operators are symbols which designate arithmetic, relational, or logical operations, and parentheses are used to determine the sequence of operations to be performed. The value of an arithmetic expression is a number of the type INTEGER, REAL, REAL 2 or COMPLEX. The value of a Boolean expression is either TRUE or FALSE, and the value of a string expression is a string of symbols. The value of a designational expression is a statement label. Expressions must be formed in accordance with mathematical convention and with the rules discussed in the following paragraphs.

4.2. ARITHMETIC EXPRESSIONS

Arithmetic quantities are combined into arithmetic expressions by means of the following arithmetic operators:

+ denotes addition
- denotes subtraction
* denotes multiplication
/ denotes division
** denotes exponentiation
//: denotes integer division

The expression

A//B

can be written only when A and B are both of type INTEGER. The expression has the integer value of the unrounded quotient of A by B.

Thus 5//3 = 1

The expression

A**B

means A raised to the power B.
4.2.1. Ordering Rules for Operations

When arithmetic expressions are evaluated, the arithmetic operations are performed according to the following rules of priority or precedence.

<table>
<thead>
<tr>
<th>Class</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>2</td>
<td>Unary minus</td>
</tr>
<tr>
<td>3</td>
<td>Multiplication</td>
</tr>
<tr>
<td></td>
<td>Division</td>
</tr>
<tr>
<td></td>
<td>Integer division</td>
</tr>
<tr>
<td>4</td>
<td>Addition</td>
</tr>
<tr>
<td></td>
<td>Subtraction</td>
</tr>
</tbody>
</table>

Expressions with operators in different classes are evaluated in order (1, 2, 3, and then 4) unless parentheses are used to change the order. Expressions containing operators in the same class are evaluated from left to right. Parentheses may be used to override the given order of evaluation. Expressions within innermost parentheses are evaluated first.

Examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Compiler Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-B-C</td>
<td>(A-B)-C</td>
</tr>
<tr>
<td>A-B**C</td>
<td>A-(B**C)</td>
</tr>
<tr>
<td>A<strong>B-D-C</strong>D</td>
<td>(A<strong>B)-(C</strong>D)</td>
</tr>
<tr>
<td>A+B/C</td>
<td>A+(B/C)</td>
</tr>
<tr>
<td>A/B/C</td>
<td>(A/B)/C</td>
</tr>
<tr>
<td>A<strong>B</strong>C</td>
<td>(A**B)**C</td>
</tr>
<tr>
<td>-A**2</td>
<td>-(A**2)</td>
</tr>
</tbody>
</table>

4.2.2. Hierarchy of Operand Types

Mixed-mode arithmetic is permitted. The evaluated arithmetic expression assumes the type of the dominant operand type in the expression. The order of dominance is COMPLEX, REAL 2, REAL, and INTEGER. Exponentiation routines for COMPLEX **REAL 2 and REAL 2**COMPLEX have not been implemented.

Example:

```
INTEGER  I $
REAL      R $
REAL 2    R2 $
COMPLEX   C $
```

then

```
I*R      IS REAL
R2*R     IS REAL 2
C-R2+I   IS COMPLEX
```

\[\text{Note that the order of precedence for the unary minus and exponentiation has been reversed from 1107/1108 EXEC II ALGOL. Under the EXEC II system } Y = -A**2 \text{ would always produce a positive result for } Y.\]
There are two exceptions to the above rule:

- \( A/B \) is \textbf{REAL} when \( A \) and \( B \) are \textbf{INTEGER}
- \( A**B \) is \textbf{REAL} when \( A \) and \( B \) are \textbf{INTEGER}

4.2.3. Operands of Arithmetic Expressions

The operands of arithmetic expressions are constants, variables, function designators (defined below), or other arithmetic expressions.

4.2.3.1. Subscripted Variables

A variable may be either a simple variable or a subscripted variable. A subscripted variable represents one of the following:

1. A single element of an array denoted by the identifier which names the array followed by a subscript list enclosed in parentheses,
2. A portion of a string variable, or
3. A combination of both (1) and (2) in the case of \texttt{STRING} arrays. A subscript list consists of arithmetic expressions separated by commas.

The following are examples of subscripted variables:

\[
\begin{align*}
A(I+J) \\
M(I+1,J+1) \\
V(F(P+1),Q+12) \\
Z(W(T),X(T),Y(T),Z(T)) \\
X(13) \\
A(I*2,I//2)
\end{align*}
\]

The expressions which make up the subscript may be of any complexity. \textbf{REAL} values are allowed, in which case the real number is rounded to the nearest integer. Each subscript expression must have a value which is not less than the minimum and not greater than the maximum specified by the \texttt{ARRAY} declaration or for the string as specified by the \texttt{STRING} declaration (see Section 3). The number of subscript expressions must equal the number of dimensions of the array as given in the \texttt{ARRAY} declaration. Thus, if an array is declared as follows,

\[
\text{REAL ARRAY } A(1:10,1:10)
\]

then \( A(3,11) \) is undefined.

4.2.3.2. Function Designators

A function designator is either a call on a declared function procedure (see Section 7) or a call on a standard function. These standard functions are the ones commonly employed in mathematics, such as the square root, sine, and arctangent functions. A complete list of the available functions is given in Appendix B. For example, the function whose value is the square root of \( X \) is called \texttt{SQRT}; therefore if

\[
\text{REAL } X \ \frac{\sqrt{}}
\]
then

\[ \text{SQRT} \ (X) \]

is a function designator.

Operands themselves may be arithmetic expressions, and combining them by means of operators may give rise to more arithmetic expressions. Assuming the declarations:

\[
\text{REAL} \ R \\
\text{INTEGER} \ I \\
\text{INTEGER ARRAY} \ A(0:10)
\]

Then the following are valid arithmetic expressions:

\[
\frac{R \times I}{A(1)+7} \quad (A(A(2))-1 \times 3) \times \text{MOD}(A(7),4)
\]

\text{MOD} is an example of a standard function. In these examples, liberal use is made of parentheses to indicate order of evaluation.

4.3. BOOLEAN EXPRESSIONS

The only Boolean constants are \text{TRUE} and \text{FALSE} and these have their fixed, obvious meaning. A Boolean operand may be either a simple variable that has occurred in a Boolean declaration, a subscripted variable that has appeared in a Boolean array declaration or a Boolean function designator such as \text{NUMERIC}, or a Boolean constant (see Appendix B). Boolean expressions are:

- Boolean operands
- Arithmetic or string expressions connected by the relational operators \text{LSL}, \text{LEQ}, \text{EQL}, \text{GEQ}, \text{GTR} or \text{NEQ}
- Boolean expressions connected by the logical operators \text{NOT}, \text{AND}, \text{OR}, \text{IMPL}, \text{EQIV}, or \text{XOR}

4.3.1. Relational Expressions

The relational operators have the semantic meanings:

<table>
<thead>
<tr>
<th>ALGOL EXPRESSION</th>
<th>MATHEMATICAL NOTATION</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{A LSS B}</td>
<td>A &lt; B</td>
<td>Less than</td>
</tr>
<tr>
<td>\text{A LEQ B}</td>
<td>A \leq B</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>\text{A EQ B}</td>
<td>A = B</td>
<td>Equal to</td>
</tr>
<tr>
<td>\text{A GEE B}</td>
<td>A \geq B</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>\text{A GTR B}</td>
<td>A &gt; B</td>
<td>Greater than</td>
</tr>
<tr>
<td>\text{A NEQ B}</td>
<td>A \neq B</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>
For arithmetic or string expressions A and B, the Boolean expression:

\[ A \text{ < relational operator> } B \]

is **TRUE** if the relation holds and **FALSE** if it does not. A and B may be mixed mode. If either A or B is **COMPLEX**, only the relations **EQL** and **NEQ** can be used. If A and B are both string expressions (see 4.4), the strings are compared character-by-character starting at the left. The shorter string is considered to be filled out with blanks to the length of the longer. The collation sequence of characters is that of Fieldata.

**4.3.2. Boolean Operators**

The six Boolean (logical) operators are:

- **NOT**  negation
- **AND**  conjunction
- **OR**   inclusive disjunction (inclusive OR)
- **IMPL** implication
- **EQUIV** equivalence
- **XOR**  exclusive disjunction (exclusive OR)

The value of a Boolean expression of the form:

\[ A \text{ < Boolean operator> } B \]

is obtained from the following table. **NOT** is a unary operator.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>NOT A</th>
<th>A OR B</th>
<th>A AND B</th>
<th>A XOR B</th>
<th>A IMPL B</th>
<th>A EQUIV B</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Assume the following declarations:

```plaintext
BOOLEAN A,B $
REAL X,Y,Z $
STRING S(100) $
```

Then the following are legitimate Boolean expressions.

```plaintext
B AND A
10.0 LEQ X AND X LEQ 99.0
NOT B OR A
NOT (X LSS Y IMPL Z EQL Z**2)
NUMERIC(S) OR NOT ALPHABETIC(S)
```

In the above example **NUMERIC** and **ALPHABETIC** are standard functions (see Appendix B).
4.3.3. Precedence of Boolean Operations

Parentheses may be used to specify the order of operations in Boolean expressions. If parentheses are omitted (or within parentheses), Boolean expressions are scanned from left to right, and operations are performed according to the following precedence:

1. Arithmetic operations according to 4.2.1.
2. Relational operations
3. NOT
4. AND
5. OR XOR
6. IMPL
7. EQIV

Example:

\[ A + 1 \text{ GTR } B \text{ OR } C \text{ AND } G + 2 \text{ NEQ } H \]

will be computed as

\[ ((A + 1) \text{ GTR } B) \text{ OR } (C \text{ AND } (G + 2) \text{ NEQ } H) \]

Parentheses should be used in compound expressions to avoid confusion or misinterpretation by human readers (not the compiler).

4.4. STRING EXPRESSIONS

Strings have no operators which produce a string result. Substrings of a declared string are defined by giving a starting position and a length. For example, if \( S \) is a string variable

\[ \text{STRING } S(120) \]

then

\[ S(i) \]

denotes the \( i \)th character in the string \( S \) where the characters are numbered from the left starting with one. Thus, \( S(6) \) is the sixth character in string \( S \).

\[ S(i,j) \]

denotes a string of \( j \) characters taken in ascending order from string \( S \) starting with the character in the \( i \)th position. \( S(1,10) \) is the substring of \( S \) consisting of the first ten characters of \( S \). The substring \( S(1) \) is equivalent to \( S(1, 1) \). That is, if the length is omitted, it is taken to be 1.
In summary, consider string $S$.

$$\text{STRING } S(10,R(20):\text{STACK}(3,\text{OP}(5)):4)$$

In this string the following three string expressions all have the same value; that is, they all refer to the same five characters.

$$S(34,5)$$
$$\text{STACK } (4,5)$$
$$\text{OP}$$

Partial string-array variables are subscripted variables with two separate subscript lists separated by a colon (\text{:}). A subscripted string variable is written as

$$S(<\text{string part}>:<\text{subscript list}>)$$

If $S$ is a two-dimensional string array, then

$$S(i:j, k)$$

denotes the $i$th character in the $j$, $k$th element of the string array $S$. If a group of characters is desired, then

$$S(i, l:j, k)$$

will denote the group of $l$ characters taken in ascending order starting with the character in the $i$th position from the $j$, $k$th element in the string array. On the other hand, if the entire string from the $j$, $k$th element in the string array is desired, then the colon may be omitted. Thus,

$$S(j, k)$$

specifies the entire string.

Example:

$$\text{STRING ARRAY } S(L(40), R(40):1:10, 1:10)$$

then

$$S(41, 40:4:10)$$

specifies the last 40 characters of the string in the fourth row and tenth column of the two-dimensional array. Each of the elements consists of 80 characters.

A numeric string expression may be used as an arithmetic operand:

$$S(1, 5)+1$$
$$S(1) \text{ EOL } 1$$
When used in this context, the string expression is converted to an integer expression by a transfer function. If the string does not represent an integer, an error message is printed (see Appendices B and C). If the integer value of the string is greater than \((2^{35}-1)\), an error message results.

4.5. DESIGNATIONAL EXPRESSIONS

Designational expressions are expressions whose values are statement labels (see 5.5). The form of a designational expression is either a label, a switch variable, or a conditional expression in which the value of a Boolean expression determines which of two designational expressions to use. For further details see 6.2 and 6.3.

4.6. CONDITIONAL EXPRESSIONS

The value of an expression may depend on the value of a Boolean expression. For example,

\[
\text{IF } X \text{ EQL } Y \text{ THEN } 1 \text{ ELSE } 2
\]

is an integer expression whose value is 1 or 2 depending on whether \(X\) equals \(Y\). The general form of a conditional expression is:

\[
\text{IF <Boolean expression> THEN <simple expression> ELSE <expression>}
\]

The expression following \(\text{THEN}\) and the expression following \(\text{ELSE}\) must be of the same kind (arithmetic, Boolean, or string). The expression following \(\text{ELSE}\) may be another conditional expression.

Example:

\[
\text{IF } X \text{ GTR } 0 \text{ THEN } 0 \text{ ELSE } (X \text{ EQL } 0)
\]

is illegal because in some cases it has an arithmetic value (0) and in other cases a Boolean value \((X \text{ EQL } 0)\). In the cases where the constituent expressions are arithmetic, then the type of the entire expression is always the more general of the two expressions:

\[
\text{IF } X \text{ EQL } 0 \text{ THEN } <1.0,2.0> \text{ ELSE } 2.0
\]

has a value of either \(<1.0,2.0>\) or \(<2.0,0.0>\); that is, a value of type \textit{COMPLEX}. Note that \(\text{ELSE}\) must always be present in conditional expressions.

The \(<\text{simple expression}>\) may be any expression not starting with \textit{IF}, or any expression put into parentheses. For example, the following is not a simple expression because it begins with \textit{IF}, but it contains a simple expression in the parentheses.

\[
\text{IF } A \text{ THEN } X+(\text{IF } B \text{ THEN } X \text{ ELSE } Z) \text{ ELSE } \text{IF } B \text{ THEN } Z \text{ ELSE } 0
\]
5. STATEMENTS

5.1. GENERAL

The ALGOL statement is the fundamental unit of operation within the language. The operations to be performed are specified by statements which may be divided into two classes:

- Assignment statements
- Control statements

This section discusses assignment statements, combination of statements, statement labels, and rules for punctuating statements. Section 6 is devoted exclusively to control statements.

5.2. COMPOUND STATEMENTS

A number of statements may be grouped to form a compound statement which is to be considered as a single statement. The general form of a compound statement is:

\[
\text{BEGIN } S_1 \$ S_2 \$ \ldots \$ S_n \text{ END}
\]

where \( S_1 \) through \( S_n \) are single statements or other compound statements. The words \text{BEGIN} and \text{END} serve as opening and closing statement parentheses. Note the absence of \$ between \( S_n \) and \text{END}.

5.3. ASSIGNMENT STATEMENTS

An assignment statement is of the form:

\[
v = e \quad \text{or} \quad v := e
\]

Where \( v \) is a variable (simple or subscripted), \( e \) is an expression, and the equal sign is known as the replacement operator. This replacement operator is not equivalent to the equal sign in mathematical notation. The assignment statement specifies that the expression \( e \) is to be evaluated and this value is to replace the current value of the variable \( v \).
Examples:

```
SUM = 0 $
X = Y + Z $
A(I) = X1 $
N = N + 1 $
```

Replace the current value of SUM with zero.
Replace the current value of X with the value of (Y + Z).
Replace the current value of the Ith element of array A with the value of X1.
Increase the value of N by 1.

Note that the last example is not a valid algebraic equation, but it is a valid assignment statement. On the other hand, a valid algebraic equation such as:

```
Z**2 = X**2 + Y**2
```

has no meaning to the compiler as Z**2 is not an identifier. Neither an expression nor a constant may appear to the left of the replacement operator.

In the statement

```
N = IF X EQL Y THEN 1 ELSE 2
```

N is assigned the value 1 or 2 depending on whether X equals Y. In the assignment statement

```
v = e
```

v must be compatible in type with e. The compiler includes transfer functions to transfer from the type of e to the type of v. The available transfer functions are summarized at the end of Appendix B. If v and e are of different types, then the compiler converts e to the type of v before the assignment is made.

If the conversion is from REAL to INTEGER then the result is rounded to the nearest integer as in the following example:

```
INTEGER I $
I = 1.57 $
```

The assignment statement assigns the value 2 to I. This is equivalent to writing \( I = \text{ENTIER} (1.57 + 0.5) \) where \( \text{ENTIER} \) is a standard function which returns the integral part of the argument.

If the expression e is BOOLEAN, then v must also be BOOLEAN. If e is a string expression then v must be type STRING or INTEGER; type INTEGER applies only if the expression is a numeric string (see Appendix B). If e is an arithmetic expression then the v must be arithmetic. v may be type STRING if e is INTEGER.
5.3.1. String Assignment Statements

If the variable \( V \) in \( V = S \) is a string variable, then this is known as a string assignment statement. In this case, the expression \( S \) must be either arithmetic or of type STRING. If \( S \) is an arithmetic expression then it is first converted to type INTEGER and then into its associated string.

In all cases, the replacement is made such that the leftmost character of the right-hand side replaces the leftmost character in the left-hand variable. If the string on the left-hand side is longer, extra spaces are supplied to the right as necessary to fill out the left-hand string. If the string on the right-hand side is longer, any excess of characters from the right-hand side is dropped (that is, the replacement is left justified in the left-hand string variable).

As an illustration, consider the following uses in which \( A \) is a string variable:

<table>
<thead>
<tr>
<th>A before</th>
<th>Statement</th>
<th>A after</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDEF</td>
<td>( A = 'XYZUVW' )</td>
<td>XYZUVW</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>( A = 'LOOP-DE-LOOP' )</td>
<td>LOOP-DE-LOOP</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>( A = 'HOW' )</td>
<td>HOW</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>( A(2) = 'Q' )</td>
<td>AQCDEF</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>( A(2,3) = 'XYZ' )</td>
<td>AXYZDEF</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>( A(2,3) = 69 )</td>
<td>A69 EF</td>
</tr>
</tbody>
</table>

It is preferable to write the last statement in the form \( A(2,3) = 69 \) so as to avoid the time consuming integer-to-string conversion. One word of caution, the string replacements are performed a character at a time starting with the leftmost character; hence, a replacement of the form

\[
S(2, N-1) = S(1, N-1)
\]

will result in the character in the 1, 1 position, \( S(1,1) \), being propagated down the string (i.e. the first \( N \) characters of the string \( S \) will all be the same as the character in \( S(1,1) \)).

Characters in a string can be shifted right or left by this type of statement. To shift the characters in a string right, first move the string into another string of the same length and then perform a replacement operation. For example, let \( S \) and \( T \) be strings of the same length; then the following statements shift \( S \) right one position and leave the first character unchanged.

\[
T = S \\
S(2, N-1) = T(1, N-1)
\]

Similarly \( S \) can be shifted left by

\[
S(1, N-1) = S(2, N-1)
\]
5.4. MULTIPLE ASSIGNMENT STATEMENTS

The same value can be assigned to a number of variables by means of a multiple assignment statement. If the variables to which a value is being assigned are mixed in type, then type conversions are performed. Assume X, Y, and E are REAL, and I is INTEGER. Then the statement

\[ X = I = Y = E \]

evaluates E and assigns this value to Y; the value of Y is then rounded to an integer and assigned to I; the value of I is converted to REAL and assigned to X.

The general form of assignment statement is of the form

\[ V_1 = V_2 = V_3 \ldots V_n = E \]

where the \( V \)'s are variables (simple or subscripted) and E is an expression. If the \( V \)'s include subscripted variables, then the order of evaluation is as follows:

1. Any subscript expressions are evaluated in sequence from left to right.
2. The expression E is evaluated.
3. The value of the expression is assigned to all variables proceeding from right to left (as in the example \( X = I = Y = E \)) with the subscripts having values as determined in 1.

If the value of I is 1 before this statement is encountered,

\[ A(I) = B(I+1) = I = I+1 \]

then evaluation continues as follows (A and B real arrays)

1. The subscript for A is determined as 1 and for B as 2.
2. I is incremented by 1 thus it becomes 2.
3. The integer is converted to REAL and assigned to A(1) and B(2).

Thus \( A(1)=B(2)=2.0 \)

5.5. STATEMENT LABELS

In order to identify a statement a name may be attached to it. This name is called a statement label and permits one statement to refer to another. A label is an identifier - a string of letters and digits beginning with a letter. Numeric labels are not permitted in this implementation of ALGOL. The string may be of any length but, like any identifier they must be unique within the first 12 characters (see 2.2). The label precedes the statement and is separated from it by a "::" (or .). Multiple labels are permitted.

Examples:

\[
\begin{align*}
L1 & : \quad Y = A*X + B*C \\
\text{START} & : \quad \text{SUMX} = 0 \\
\text{START} & : \quad \text{SUMX} = 0 \\
L1 & : \quad L2 : \quad L3 : \quad L4 \quad X = Y \\
\end{align*}
\]
A label is defined by its actual occurrence and is therefore local to the block in which it occurs. Of course, each label must be different from all other identifiers referenced within the block. See Section 8 for further discussion of labels and blocks.

5.6. PUNCTUATION

Each statement as well as each declaration must be terminated by a $ or a ;. In a compound statement, a $ preceding the END would be redundant and may be omitted.

The end of a line has no meaning as punctuation. There is no restriction as to the number of cards that may be used for forming a statement. Spaces must not appear within numbers, labels, or in basic symbols (except for GO TO and REAL 2). However, spaces must be used to separate adjacent symbols composed of letters or digits. Spaces may be used freely for indentation or to facilitate reading.

5.7. DUMMY STATEMENTS

A dummy statement performs no operation. It can be used to place a label.

Example:

L1: END
DO $
6. CONTROL STATEMENTS

6.1. GENERAL

The compiler translates successive statements in the order in which they appear in the program. The statements are also executed in this same order unless the programmer interrupts this normal sequence with a "transfer of control." Once the transfer has taken place, successive statement sequencing continues from the new point of reference.

Transfer of control in ALGOL is accomplished through use of three kinds of control statements – unconditional, conditional, and iterative.

6.2. UNCONDITIONAL CONTROL STATEMENTS.

The \texttt{GO TO} statement causes an unconditional transfer of control to another part of the program.

6.2.1. The \texttt{GO TO} Statement

The \texttt{GO TO} statement may be written in any one of three ways:

- \texttt{GO TO <designational expression>}
- \texttt{GOTO <designational expression>}
- \texttt{GO <designational expression>}

There are three forms of designational expressions, the label being the simplest.

Example:

\begin{verbatim}
L:q = SIN(SQRT(Z)) $
$
$
$
$
GO TO L $
\end{verbatim}

\texttt{GO TO L} interrupts the normal sequence of instructions and restarts at the statement with the label \texttt{L}. 
Alternatively the designational expression may take the form of a conditional expression.

Example:

```
GO TO IF X EQL Y THEN L1 ELSE L2 $
```

In this case if X equals Y, control is transferred to the statement labeled L1; otherwise, the transfer is to L2.

A third form of designational expression is a `SWITCH` variable explained below.

### 6.2.2. The `SWITCH`

The `SWITCH` declaration names a group of alternative points in a program to which control may be transferred. It includes a means for selecting a given designational expression from the `SWITCH` list by means of a subscript expression (evaluated at execution time) with the `SWITCH` identifier. In effect, the `SWITCH` declaration defines a `SWITCH` variable which is similar to a one-dimensional array except that the elements are designational expressions.

To start with, a switch must be described by a `SWITCH` declaration prior to its use as a switch variable. The range of subscripts is from 1 to n, where n is the number of elements in the switch list. If a subscript expression on a switch variable falls outside the defined range of the switch, then the switch operation is ignored.

The general form of the declaration is

```
SWITCH <switch identifier> = <switch list>
```

or

```
SWITCH SWITCH 1 = e_1, e_2, e_3, ..., e_n $
```

where `SWITCH 1` is the name of the switch and `e_1, e_2, ..., e_n` are designational expressions.

A switch element is referenced in a `GO TO` statement by means of the switch identifier with the appropriate subscript:

```
GO TO SWITCHN1(I)
```

where `I` is an arithmetic expression. This expression is evaluated when the `GO TO` is executed. Control is transferred to the statement designated by element `I` in the switch list of the `SWITCH` declaration (counting from left to right).

To illustrate, assume that it is necessary to transfer to statements labeled L1, L2, L3, and L4 depending on whether the value of `J` is 1, 2, 3 or 4. This could be accomplished with the following `GO TO` statement:

```
GO TO IF J EQL 1 THEN L1 ELSE IF J EQL 2 THEN L2 ELSE IF J EQL 3 THEN L3 ELSE IF J EQL 4 THEN L4 $
```
However, it is much easier to set up a switch to accomplish the same thing

```
SWITCH S = L1, L2, L3, L4 $
  .
  .
  .
  GO TO S(J) $
```

Example:
```
SWITCH S = L1, IF X GTR Y THEN L2 ELSE L3, L4, T(I+6), L5 $
```

If the switch variable S is referenced from a GO TO statement
```
GO TO S(J),
```

the following transfer of control is made depending upon the value of J:

1. If J = 1 then control transfers to L1.
2. If J = 2 then control transfers to either L2 or L3 depending upon X and Y.
3. If J = 3 then control transfers to L4.
4. If J = 4 then control transfers to the label which is the value of the (I+6)th designational expression of the switch T.
5. If J = 5 then control transfers to L5.
6. If J < 1 or J > 5 then no transfer is executed.

6.3. CONDITIONAL CONTROL STATEMENTS

Conditional control statements cause certain statements to be executed or skipped depending on values of Boolean expressions. The IF statement provides for executing a statement if, and only if, some relation is true and for skipping over a statement if this relation is false.

The IF statement may take the form:
```
IF B1 THEN S1 $ S2 $
```

where B1 is a Boolean expression, S1 is a statement not beginning with IF, and S2 is any statement. If B1 is true, then S1 is executed after which control passes to S2. If B1 is false, then S1 is skipped and control continues at S2.

In diagram form:
```
IF B1 THEN S1 $ S2 $  
   B1 True
   B1 False
```

The general form of the IF statement is:

IF B1 THEN S1 ELSE S2 $ S3 $

If B1 is true, statement S1 is executed and statement S2 is skipped; if B1 is false, statement S1 is skipped and S2 is executed. In either case, control continues with statement S3 (except when either S1 or S2 contains a GO TO statement).

In diagram form

```
B1 True
   IF B1 THEN S1 ELSE S2 $ S3 $
   B1 False
```

In conditional statements, the statement following THEN can not start with IF. It may be conditional only if it is enclosed by a BEGIN - END pair. There is no restriction on the type of statement following ELSE.

Example:

IF BOOL THEN BEGIN IF C GEQ -5 THEN GO TO CHECK END ELSE V = V+1 $

The following example illustrates "nested" conditional statements:

```
B1 True
   IF B1 THEN S1 ELSE IF B2 THEN S2 ELSE S3 $ S4 $
   B1 False

B2 True
   IF B2 THEN S2 ELSE S3 $ S4 $
   B2 False
```

Example:

IF DISC LSS 0 THEN GO TO IMAGROOTS
   ELSE IF DISC EQL 0 THEN X1 = X2 = -B/(2*A)
   ELSE BEGIN
      SDISC = SQRT(DISC) $
      X1 = (-B+SDISC)/(2*A) $
      X2 = (-B - SDISC)/(2*A)
   END $

6.4. ITERATIVE CONTROL STATEMENTS – THE FOR STATEMENT

The FOR statement facilitates programming iterative operations. A part of the program is iterative if it is to be executed repeatedly a specified number of times, if it is to be executed for each one of a designated set of values assigned to a variable, or if it is to be executed repeatedly until some condition is fulfilled. The FOR statement handles any of these three conditions.
The general ALGOL FOR statement consists of a `<FOR clause>` followed by a statement S (simple or compound) where a `<FOR clause>` is:

```
FOR <variable> = <FOR list> DO
```

The `<FOR list>` is a sequence of `<FOR list elements>` separated by commas. The value of each `<FOR list element>` is assigned to the controlled or iteration variable in turn from left to right and the statement S is executed once for each value.

All FOR list elements must be of a type compatible with the controlled variable which may be any type of simple or subscripted variable.

There are three possible kinds of FOR list elements:

- `<arithmetic expression>`
- `<arithmetic expression> STEP <arithmetic expression> UNTIL <arithmetic expression>`
- `<arithmetic expression> WHILE <Boolean expression>`

### 6.4.1. Simple List Element

```
FOR V = <arithmetic expression> DO S $
```

or

```
FOR V = e_1, e_2, e_3, e_4, \ldots, e_N DO S $
```

The controlled variable V is successively given the values of the arithmetic expressions, e_1, e_2, e_3, \ldots, e_N. The statement S is executed once for each value of V.

Example:

```
FOR X = 1.0, 1.5, 2.5, 3.5, 7.5 DO S $
```

### 6.4.2. STEPN-UNTIL List Element

```
FOR V = <arithmetic expression> \> STEP <arithmetic expression > \> UNTIL <arithmetic expression > DO S $
```

or

```
FOR V = E_1 \> STEP E_2 UNTIL E_3 DO S $
```

where E_1 is the starting or initial value of V

- E_2 is the increment by which V is increased algebraically
- E_3 is the limiting or terminal value of V
The effect of the **FOR** statement is probably best described by the equivalent **ALGOL** statements:

\[
V = E1 \$
L1: \text{IF } (V-E3) \times \text{SIGN}(E2) \leq 0 \text{ THEN}
\begin{align*}
S & \$
V = V + E2 & \$
\text{GO TO } L1
\end{align*}
\text{END}
\]

In all cases if the test fails initially, the statement S is not executed at all. **SIGN** (X) is a call on a standard function which will return the value 1, 0, or -1 depending on whether the value of the argument X is positive, zero, or negative, respectively. This can be shown graphically as follows:

The statement S may redefine V as well as the variables appearing in E2 and E3. Changing E1 will have no effect on the execution of the **FOR** statement as the initial value is assigned to V before S is executed. Extreme care must be taken in assigning values to V within S as this may prevent V from reaching the terminal value.

The more compact form of the **FOR** statement

\[
\text{FOR } V = (E1,E2,E3) \text{ DO } S \$
\]

may be used instead of

\[
\text{FOR } V = E1 \text{ STEP } E2 \text{ UNTIL } E3 \text{ DO } S \$
\text{FOR } I = 1 \text{ STEP } 1 \text{ UNTIL } N \text{ DO } S \$
\text{FOR } I = (1,1,N) \text{ DO } S \$
\text{FOR } X = (3.2,.1,9.9) \text{ DO } S \$
\]
6.4.3. **WHILE** List

\[
\text{FOR } V = \text{ <arithmetic expression> } \text{WHILE } \text{<Boolean expression>} \text{ DO } S$
\]

or

\[
\text{FOR } V = E \text{ WHILE } B \text{ DO } S$
\]

First \(V\) is set equal to the arithmetic expression \(E\). If \(B\) is true, statement \(S\) is executed. After the execution of \(S\), \(V\) is replaced by \(E\) and again \(B\) is tested. If, on the other hand, \(B\) is false, then \(S\) is skipped and control resumes with the statement following the **FOR** statement.

This can be represented graphically as follows:

![Diagram](image)

The statement \(S\) may redefine \(V\) or the variables in the expressions \(E\) and \(B\).

Example (taken from Problem 2 Appendix D):

\[
\text{FOR } B = 0.5 * (A/OLDB + OLDB) \text{ WHILE } \text{ABS}(B-OLDB) \text{ GTR } 10**(-6)*B \text{ DO } OLDB = B$
\]

In this example the **FOR** statement is executed until \(B\), the square root of \(A\), is accurate to six digits.

The three forms of the list elements may be combined:

\[
\text{FOR } K = 1,3,5,10 \text{ STEP 2 UNTIL } 20,50 \text{ WHILE } B \text{ DO } S$
\]

The statement \(S\) will be executed for

\[K = 1,3,5,10,12,14,16,18,20\]

and will then assume the value 50 as long as \(B\) is true.
6.4.4. Termination of FOR Statements

The following section should be carefully read because it deals with concepts that are not defined in rigorous ALGOL 60. The problem is that a program written in UNIVAC 1108 ALGOL 60 which utilized these concepts would possibly not work on a machine with a different version of ALGOL. The concern here is with the value of the iteration variable in a FOR statement when the FOR statement is terminated. The ALGOL 60 report leaves this value as undefined when the FOR statement is terminated by exhaustion of the <FOR list>, but in UNIVAC 1108 ALGOL it is well defined, and indeed, very useful. It is because of its usefulness that it is documented here with the warning that it may not work on another machine.

If the statement S has a GO TO statement leading out of the FOR statement, the value of the iteration variable is the same as it was before the GO TO statement was executed. (This is also true in ALGOL 60.) If the exit is made from the FOR statement because of the exhaustion of the <FOR list>, then the value of the variable is that value it held last as may be determined from the equivalent ALGOL statements. For example, to find the first nonblank character of a string, either one of two methods could be used.

```algol
STRING S(120)$
INTEGER I,R$
I:=0$
FOR I:=I+1 WHILE (I LSS 121 AND S(I) EQL ' ') DO$
IF I EQL 121 THEN GO TO STRINGALLBLANK$
ELSE FOUNDIT: R=RANK(S(I))$

That method depends on the exhaustion of the <FOR list>, either because the whole string has been scanned or because a nonblank character has been found. In one case, the final value of I is 121 and in the other it is the index to the nonblank character. Note that a dummy statement DO $ follows the FOR statement (see 5.7). RANK is a standard function returning the Fielddata equivalent of the first character of the string.

The second method is as follows:

```algol
FOR I=(1,1,120) DO IF S(I) NEQ ' ' THEN GO TO FOUNDIT$ GO TO STRINGALLBLANK$
FOUNDIT: R=RANK(S(I))$

This method produces the correct value of I because an exit is made from the FOR statement by a GO TO statement.
A GO TO statement from outside a FOR statement referring to a label within the FOR statement may result in an undefined situation and should thus be avoided.

```
FOR I=(1,1,N) DO
  BEGIN
    L:
    END
  GO TO L
```

The above statement, GO TO L, is not allowed.

However, it is easy to program the above logic by not using the FOR statement.

Example:

```
I = 0
LOOP: I = I+1
L:
  IF I LSS N GO TO LOOP
  GO TO L
```
7. PROCEDURES

7.1. INTRODUCTION

A procedure in ALGOL is used to specify an independent section of a program (which usually represents an algorithm) that can be called or executed at different points throughout the same program or may be used in other programs. The operations to be performed are fixed, but a list of parameters makes it possible for a procedure to be used with varying values and/or variables.

A procedure must be declared in the declaration part of the block in which the procedure is referenced. More than one procedure may be defined at the beginning of a block. During program execution when a block is entered, the first statement executed is the first executable statement following the procedures (if any).

The procedure declaration consists of a procedure heading and a procedure body. The heading consists of a procedure identifier, a formal parameter list, if any, a value list, if any, and specifications, if any. The procedure body follows the specifications and consists of a statement, compound statement, or a block.

Example:

`PROCEDURE NFACT (ARG1, ARG2) $`
`INTEGER ARG1, ARG2 $`
`BEGIN $`
`INTEGER I $`
`ARG2 = 1 $`
`FOR I = 1 STEP 1 UNTIL ARG1 DO $`
`ARG2 = ARG2*I$`
`END $`

In the above example NFACT is the identifier for a procedure that calculates the value of N factorial. ARG1 and ARG2 are the formal parameters (arguments) for the procedure. ARG1 is the number whose factorial is to be calculated, and ARG2 is the result after the procedure has been executed. ARG1 and ARG2 are INTEGER variables. The BEGIN-END pair sets off the body of the PROCEDURE. The BEGIN-END pair can be dropped if the procedure body is just one statement. Since I is declared as an INTEGER, it is local to NFACT.

A "call" of the above procedure would be of the form:

`NFACT (N,FACT) $`
`... $`
`... $`
`NFACT (N1,FACT1) $`
In the first call, the actual parameters N and FACT are substituted for the formal parameters ARG1 and ARG2. Later the parameters N1 and FACT1 are substituted for ARG1 and ARG2 in the same fashion. Thus procedure is a closed subroutine, and the call establishes a linkage to the subroutine.

An alternate form of the parameter list allows comments to be inserted between the formal parameters since the comma separating formal parameters is equivalent to:

) < string > : (  

PROCEDURE NFACT (ARG1, ARG2)

could be written as:

PROCEDURE NFACT (ARG1) AND STORE RESULT IN: (ARG2) $

or

PROCEDURE NFACT (ARG1) ARG1 INPUT AND ARG2 OUTPUT: (ARG2) $

The following is a procedure for the multiplication of two matrices:

PROCEDURE MATMUL (A1,B1,C1,N1,M1,P1) $
REAL ARRAY A1,B1,C1, $  
INTEGER N1,M1,P1 $  
BEGIN INTEGER I,J,K $  
REAL TEMP $  
FOR I=1 STEP 1 UNTIL N1 DO  
FOR J=1 STEP 1 UNTIL P1 DO  
BEGIN TEMP=0.0 $ FOR K=1 STEP 1 UNTIL M1 DO  
TEMP=TEMP+B1(I,K)*C1(K,J) $  
A1(I,J)=TEMP END I,J LOOP  
END MATMUL $

A procedure statement calls for the execution of a procedure body.

Given the declaration

REAL ARRAY A1(1:10,1:7), A2(1:10,1:15), A3(1:15,1:7) $

then the procedure statement

MATMUL (A1,A2,A3,10,15,7) $

has the effect of multiplying the two matrices A2 and A3 and storing the results in A1.

Expressions may also be used as actual parameters. Care must be taken to match the type and kind of each formal and actual parameter in any call.
7.2. VALUE ASSIGNMENT (CALL BY VALUE) AND NAME REPLACEMENT (CALL BY NAME)

The above procedure NFACT makes use only of the value of ARGl whereas it changes the value of the actual parameter which replaces ARG2. Thus NFACT could be rewritten as follows:

```
PROCEDURE NFACT (ARG1, ARG2)
  VALUE ARGl $
  INTEGER ARGl, ARG2 $
  BEGIN
    INTEGER I $
    ARG2 = 1 $
    FOR I = 1 STEP 1 UNTIL ARGl DO
      ARG2 = ARG2*I
    END$
```

The procedure statement

```
NFACT (NUMBER, FACTORIAL) $
```

has this effect: the value of the actual parameter, NUMBER, replaces ARGl when the procedure statement is encountered and NFACT does not have access to the location assigned to NUMBER. ARG1 is known as a <Call by value> parameter.

A value parameter must also have a type specification and cannot appear to the left of the replacement operator in the procedure.

Any parameter (such as ARG2) which is not listed in the VALUE part of the procedure declaration is said to be a <Call by name> parameter. The name FACTORIAL replaces the name ARG2. The value of FACTORIAL is changed as the procedure is executed.

In the following examples, numeric values or expressions are used as actual parameters:

```
NFACT (15, FACT1) $
NFACT (J+K+M, FACT2) $
```

It should be noted that a value parameter which is an array or string identifier requires that the entire array or string supplied by the procedure call be copied locally within the procedure. As a result large amounts of working storage may be used unexpectedly when the procedure is called. All calculations in the procedure use that temporary copy. As an example, suppose it is necessary to find the determinant of a matrix without destroying the matrix. The usual computational methods for finding determinants destroy the matrix with which they are working. Thus the original matrix must be copied somewhere. Specifying the array as VALUE accomplishes this:
REAL PROCEDURE DET(A) SQUARE MATRIX WHOSE DIMENSION IS:(N) $
VALUE A,N$ 
REAL ARRAY A $ 
INTEGER N $ 
BEGIN 
(STATEMENTS) 
DET=... END DET $ 

This is an example of a function procedure explained in 7.4. If an expression is used as an actual parameter, and if the parameter is called by name, then the expression is reevaluated at each occurrence of the formal parameter in the procedure.

7.3. SPECIFICATIONS

The <type> of all formal parameters defined by a procedure declaration must be specified in the specification part of a procedure heading. The format of the specification part is as follows:

<s specification> <identifier list> ;

The specification may be in any one of the following forms:

<type>
ARRAY
<type> ARRAY
STRING
STRING ARRAY
PROCEDURE
<type> PROCEDURE
LABEL
SWITCH
FORMAT
LIST

and< type> is one ALGOL type: INTEGER, REAL, REAL 2, BOOLEAN, or COMPLEX.

The <identifier list> consists of the formal parameter identifiers contained in the procedure declaration separated by commas.

The reason that all formal parameters must be specified is that the compiler must know the type and kind or class of all parameters in order to compile proper machine code.

Examples:

INTEGER I, K ;
REAL X, Y ;
REAL ARRAY Z ;
BOOLEAN PROCEDURE F ;
STRING S ;
Specifications do not include information about lengths of strings, the dimensions and bounds of arrays, the formal parameter parts of procedures, or the contents of formats and lists. The actual declarations of these exist elsewhere in the program. The details of constructing a procedure can be illustrated by an example:

```algol
PROCEDURE TRANSPOSE (A) ORDER:(N) $
VALUE N $
ARRAY A $
INTEGER N $
BEGIN
    REAL w $
    INTEGER I, K $
    FOR I = 1 STEP 1 UNTIL N DO
        FOR K = 1+I STEP 1 UNTIL N DO
            BEGIN
                W = A(I,K) $
                A(I,K) = A(K,I) $
                A(K,I) = W
            END
END
END TRANSPOSE$
```

7.4. FUNCTION PROCEDURES

Procedures which are to be used as functions (e.g., SIN, EXP) must have a type associated with the procedure identifier (i.e. procedure name). This type declaration must be the first symbol of the procedure declaration. Also for the function procedure to have a value associated with it, the procedure identifier must occur at least once as the left part of an assignment statement in the procedure body. In addition, at least one of these assignment statements must be executed on a given procedure call for a value to be assigned to the procedure. If more than one such assignment statement is executed within the body, then the last one executed before exiting from the procedure determines the value associated with the procedure. Any other occurrences of the procedure identifier within the body of the procedure are considered as recursive calls on the procedure.

The procedure NFACT could be written so that the only parameter would be N and the value of NFACT would be N factorial.

```algol
INTEGER PROCEDURE NFACT (ARG) $
INTEGER ARG $
BEGIN
    INTEGER I, TEMP $
    TEMP = 1 $
    FOR I = 1 STEP 1 UNTIL ARG DO TEMP = TEMP*I$
    NFACT = TEMP
END NFACT $
```

The call for the above procedure would be of the form

```algol
COMMENT SET FACT = N FACTORIAL$
FACT = NFACT(N) $
```
A function procedure is referenced by a function designator which defines a single numerical or logical value. NFACT(N) is a function designator which will have an integral value and can thus be used in any expression in which an integer variable could be used.

7.5. RECURSIVE PROCEDURES

In the example above, a new variable TEMP was used to store the intermediate result of the calculation of N factorial. Then the extra statement NFACT = TEMP was needed to give NFACT the proper value. The reason for this is that inside the procedure body, whenever the name of the procedure occurs on the left-hand side of an assignment statement, it is a <procedure assignment> statement, that is, the statement which assigns the value to the procedure. Wherever else the name occurs, it is a call on the procedure.

This kind of construction can be used to produce another version of NFACT which is even simpler to write. In fact, it requires only one statement in the procedure body:

\[
\text{INTEGER PROCEDURE NFACT(N) \$}
\text{INTEGER N \$}
\text{NFACT = IF N EQL 0 THEN 1 ELSE N*NFACT(N-1) \$}
\]

which is equivalent to the recursive definition

\[
\text{factorial(n) = 1 \quad n = 0}
\text{= n*factorial(n-1) \quad n > 0}
\]

A procedure call with the actual parameter 4 (FACT = NFACT(4)) has the following effect. After the subroutine linkage is set up, the procedure body is virtually changed to:

\[
\text{NFACT=IF 4 EQL 0 THEN 1 ELSE 4*}
\text{NFACT (3)}
\]

NFACT(3) is another call of the functional procedure having the result that the call is replaced with the procedure body:

\[
\text{NFACT=IF 4 EQL 0 THEN 1 ELSE 4*}
\text{(IF 3 EQL 0 THEN 1 ELSE 3*NFACT(2))}
\]

This produces another call on NFACT resulting in another change of the statement. This goes on until finally:

\[
\text{NFACT=IF 4 EQL 0 THEN 1 ELSE 4*}
\text{(IF 3 EQL 0 THEN 1 ELSE 3*(IF 2 EQL 0 THEN 1 ELSE 2*(IF 1 EQL 0 THEN 1 ELSE 1*(IF 0 EQL 0 THEN 1 ELSE 1*(NFACT(0)))))))}
\]

The process is terminated when 0 \textbf{EQL} 0 occurs in the relation of the conditional expression. The usual expression for the factorial is obtained after the unnecessary parts of the above statement have been removed:

4 factorial = 4*3*2*1

All procedures written in ALGOL may be called from within themselves. But it should be mentioned that recursive procedures are not always put to good use. For example, using the recursive properties of procedures makes a much neater looking \texttt{NFACT}, but also a much less efficient one. However, recursive procedures may have practical uses. For example, multiple integration programs use a quadrature procedure to evaluate the inner function as well as to do the integration. (See ACM Algorithm 233 "Simpson's Rule for Multiple Integration," Communications of the ACM Vol. 7, No. 6, June 1964.)

7.6. EXTERNAL PROCEDURES

External procedures are procedures whose bodies do not appear in the main program. They are compiled separately and linked to the main program at its execution. The \texttt{EXTERNAL} declaration serves the purpose of informing the compiler of the existence of these procedures, their types (if any), and the proper manner to construct the necessary linkages. The general form of the external declaration is:

\texttt{EXTERNAL \langle kind\rangle \langle type\rangle PROCEDURE \langle identifier list\rangle}

where \langle type\rangle is the arithmetic type or is empty, \langle identifier list\rangle is a list of identifiers of external procedures, and

\langle kind\rangle ::= \texttt{empty/FORTRAN/NON-RECURSIVE}

The words 'FORTRAN' and 'NON-RECURSIVE' have special significance only in this context. Procedures of kind \texttt{empty} are ALGOL procedures and are treated exactly like an ordinary procedure declared within the program. However, they need not be written in ALGOL language. Procedures of kind 'FORTRAN' are FORTRAN subroutines or functions and procedures of kind 'NON-RECURSIVE' are necessarily written in machine language. In the following paragraphs we assume a knowledge of the UNIVAC 1108 Executive System, FORTRAN (see 7.6.2), and the UNIVAC 1108 Assembler (see 7.6.3).

7.6.1. ALGOL External Procedures

An ALGOL program which consists entirely of a procedure is nonexecutable because it contains only a procedure declaration (see 7.1.). When such a program is compiled, the name of the procedure is marked as an entry point when the program is entered into the program file. Like all names in the program file the first six characters of the procedure name must define it. Such a procedure may be referenced from another ALGOL program as an external procedure.
Example:

PROGRAM 1
BEGIN REAL procedure DET(A*N) $
REAL ARRAY A $ 
INTEGER N $ 
VALUE A, N $ 
BEGIN
COMMENT THIS PROCEDURE FINDS THE DETERMINANT OF A REAL N BY N 
MATRIX A, LEAVING A UNCHANGED AND ASSIGNING THE VALUE TO DET $

DET = . . . END DET 
END $ 

PROGRAM 2
BEGIN REAL ARRAY MATRIX (1:10,1:10) $
EXTERNAL REAL PROCEDURE DET $ 

WRITE(DET(MATRIX,10)) $ 

END $

A user could build a library of procedures that are useful to him and then refer to 
whichever he needed by merely declaring them as external procedures in his main 
program.

7.6.2. FORTRAN Subprograms

A FORTRAN subroutine or a FORTRAN function may be made available to an 
ALGOL program by the declaration:

EXTERNAL FORTRAN <type> PROCEDURE <identifier list>

Actual parameters in calls on such procedures may be either expressions or arrays. 
(Labels, string expressions, and string arrays are specifically excluded.) The 
FORTRAN subprogram is a subroutine or function depending on the absence or 
presence of <type> in the external declaration. A FORTRAN function is used 
like an ALGOL functional procedure i.e., as an expression. For example, if 
DET (above) were a FORTRAN subroutine:

PROGRAM 1

SUBROUTINE DET(A,N,D)
DIMENSION A(N,N)
C DET FINDS THE DETERMINANT OF A REAL NXN 
C MATRIX A AND LEAVES THE RESULT IN D, 
C DESTROYING A.

D= . . . 
END
PROGRAM 2
BEGIN REAL ARRAY MATRIX(1:10,1:10) $
REAL DETVALUE $
EXTERNAL FORTRAN PROCEDURE DET $
$. 
DET(MATRIX,10,DETVALUE) $
END $

7.6.3. Machine Language Procedures

A procedure written in 1108 Assembler language may be referenced in either of two ways. The more difficult manner occurs when the procedure is declared exactly as an ALGOL external procedure. In this case the Assembler procedure must behave like an ALGOL procedure (that is, it must be able to handle recursive calls). Here the nonrecursive case is considered. The form of the declaration for these is:

EXTERNAL NON-RECURSIVE <type> PROCEDURE <identifier list>

To understand how to write such procedures consider the coding produced by the ALGOL compiler as the result of a call in the following program:

EXTERNAL NON-RECURSIVE PROCEDURE PUNCH $ INTEGER Q, S $ 
$. 
PUNCH(Q,S) $

The statement PUNCH (Q,S) results in the four lines of coding:

LMJ 11,PUNCH 
 2 
F 00,01,01,Q 
F 00,01,01,S

The second line states the number of parameters being handed through and the following lines provide information about each parameter in turn. The actual form of F is defined elsewhere in the system by a FORM directive

F FORM 6, 3, 3, 24

which specifies the number of bits in each field of F. (See "UNIVAC 1108 Assembler Programmers Reference Manual," UP-4040.)
The four fields of F are defined and encoded as follows:

**KIND**

- 00 = Expression
- 010 = Array
- 050 = Label

**TYPE**

- 01 = INTEGER
- 02 = REAL
- 03 = COMPLEX
- 04 = BOOLEAN
- 05 = STRING
- 06 = REAL 2

**REFERENCE**

- 00 = Constant
- 01 = Name
- 02 = Indirect
- 06 = Result

**LOCATION**

The location field specifies the location of the parameter. Indirect addressing may be specified and index register 11 is usually designated in this 24-bit field. With this in mind, the following rules should be followed in writing an Assembler procedure:

1. The return point for a call with N parameters is \((X_{11}) \cdot N + 1\).
2. The value of the procedure (if any) must be left in register A2 (and A3 for COMPLEX and REAL 2), absolute 14.
3. Registers 1–4 may not be used without saving and restoring.
4. Register 10 must never be destroyed.
5. The Ith parameter should be referenced by an indirect command, e.g.,

   \( \text{LA A2,*I,X11} \)

   If the parameter under reference is a double-word quantity (COMPLEX or REAL 2) its second half is in the next location, and the parameter should be referenced with a DL \( \text{A2,*I,X11} \)

6. When using an arithmetic or a Boolean expression, the word referenced in rule (5) is the value of the expression.
7. When using either an arithmetic or a Boolean array, the word referenced in (5) is itself the address of the following information:

   - number of elements
   - precision
   - lower bound (N)
   - lower bound (1)
   - address of first element
   - N
   - length (N)
   - length (1)
where N is the number of dimensions of the array, lower bound (I) is the value of the lower bound of subscript position (I) and length (I) is its length.

(8) When using a string expression, the word referenced in (5) is a string descriptor of the form:

\[
F \quad \text{FORM} \quad 1,6,18
\]

\[
F \quad \text{length, start, address}
\]

'length' is the length of the string which starts at character position 'start' of the word at 'address.' For this purpose 'start' is coded as 0–5 to select a sixth of a word, S1–S6.

(9) When using a string array, the word referenced in (5) consists of two addresses, the left one being the address of a string descriptor and the right one the address of the array information (as in 7). The address field of the string descriptor must be added to the first element address to find the first element of the string array. Both arrays and string arrays are stored by column.

(10) The name of the external procedure must be the entry point of the Assembler procedure.
8. BLOCK STRUCTURE

8.1. GENERAL

In ALGOL 60 a program is a block. In turn, this block may contain subblocks. This structure serves to facilitate the construction of a program, for these subblocks may be checked out independently before being fitted together in the final program.

8.2. BLOCKS

A block consists of two parts, a heading and a body. The heading comes first and is identified by the symbol BEGIN or <label>:BEGIN. This is followed by all the declarations needed within the block: simple variables, arrays, strings, lists, formats, procedures, etc. The body of the block contains the statements of the block. The end of the block comes with the matching END to the BEGIN of the block head.

Example:

```
BLOCK1: BEGIN
  REAL ARRAY X(1:10) $
  INTEGER I $
  FOR I=(1,1,10) DO X[I]=SIN(I) $
END
```

The first two lines constitute the block head and the other statements constitute the body. The block head may appear in the body of another block. The inner block is then said to be 'nested' in the first (outer) block:

```
BEGIN INTEGER N $
  READ(N) $
  BEGIN REAL ARRAY A(1:N,1:N) $
  .
  .
  END INNER BLOCK
END OUTER BLOCK
```

As a special case, the first (outermost) block of the program need not be enclosed in a BEGIN- END pair, although formally this is required. Instead a program may start with a declaration. All other blocks do require a BEGIN-END pair. The double use of the BEGIN-END parentheses should be noted by the user. A group of statements enclosed by a BEGIN-END pair forms a compound statement if the statement following the BEGIN is not a declaration. If this statement is a declaration, a new block is defined rather than a compound statement.
8.3. LOCAL AND GLOBAL IDENTIFIERS

All identifiers declared within a block are called local identifiers (i.e., local to the given block). Any that do not occur in declarations in the given block, but that do appear in a block containing the given block, are called global or nonlocal identifiers (to the given block). Each block introduces, at the time it is entered, a new level of nomenclature in the sense that all identifiers declared for the block assume the meaning implied by the declaration. However, identifiers in blocks containing the given block which are not redeclared retain their old significance. When the block is left, all the identifiers declared within it lose all their significance. When a variable is declared in an outer and also in an inner block, the variable assumes the value last assigned in the outer block on exit from the inner block.

Example:

```
BLOCK1: BEGIN REAL X $
   INTEGER I $
   X=2 $
BLOCK2: BEGIN REAL Y $
   INTEGER I $
   Y=X $
   I=X $
END BLOCK 2 $
I=X $
END BLOCK 1
```

This example has two blocks, the first with variables X and I and the second with variables Y and I. The I of block 1 cannot be referenced from within block 2 because block 2 has its own I.

Note that an identifier such as Y in the example, existing in an inner block, can never be referenced from an outer block because the identifier has no meaning in the outer block. In particular, this means that all labels (which are defined not by declaration, but by occurrence) are local to the block in which they occur, and so cannot be referenced from outside. Thus, a GO TO statement cannot lead into the middle of a block. All blocks must be entered through their headings. Exit is made from a block either by "falling through" the bottom of the block through the last END, or by a GO TO statement which must lead to an outer block.

The OTHERWISE declaration has the effect of stopping any reference to global (simple variable) identifiers.

Example:

```
BEGIN REAL X $
   .
   .
BEGIN INTEGER OTHERWISE $
   X=0 $
   .
   .
END $
   .
END $
```
In the inner block, the variable X is an integer variable (now defined in this block) rather than the REAL X of block 1. If, in the above example, a third block was nested in block 2 and reference was made to an undeclared variable called Z, then Z would be defined as an integer local to the block with the OTHERWISE declaration. Blocks need not be nested; they can be disjoint, in which case there is no communication of identifiers:

```
BEGIN REAL ...
·
·
END $
BEGIN INTEGER...
·
·
END $
```

The programs may be written by two different people, checked out separately, and then joined together in this manner with absolutely no problems about conflicting use of identifiers.

8.4. THE LOCAL DECLARATION

Since the 1108 ALGOL 60 compiler examines the source language only once, it must never meet an identifier without knowing what the identifier represents. For example, suppose P and Q are procedures declared in the same block and in the body of P is a call to Q. The declarations could be arranged so that Q is first:

```
BEGIN PROCEDURE Q $
    BEGIN
        ·
        ·
    END Q $
PROCEDURE P $
    BEGIN
        ·
        ·
        Q $
        ·
        ·
    END P $
```
But if Q also calls P (P and Q are mutually recursive) then clearly it is impossible to put them both first. The **LOCAL** declaration must be used to resolve such forward references (using an identifier before it is defined):

```plaintext
BEGIN LOCAL PROCEDURE P$
  PROCEDURE Q$
    BEGIN
      P$
      Q$
    END
  PROCEDURE P$
    BEGIN
      •
    END
END
```

The allowable uses of the **LOCAL** declaration are:

- **LOCAL LABEL**
- **LOCAL PROCEDURE**
- **LOCAL <type> PROCEDURE**
- **LOCAL SWITCH**
- **LOCAL FORMAT**
- **LOCAL LIST**

The only relaxation of this rule is that a label serving as the object of a **GO TO** statement or in a **SWITCH** declaration need not appear in a **LOCAL** declaration; instead it is assumed that it will eventually be defined in the block. Thus the **GO TO** statement or **SWITCH** declaration has the same effect as a **LOCAL** declaration. However, if the label is already defined in an outer block of the given block (either by occurrence, **LOCAL** or **SWITCH** declaration, or as the object of a **GO TO** statement) then that definition of the label is assumed for the inner block.

To further illustrate the concept of local and global identifiers consider the block displayed in Figure 8–1. The variables I, J, K, X, Y, Z and labels L1, L2, are local identifiers in block 1. Only I, K, X, Y, Z, L1, L2 will be global to block 2, while J is redefined and local in block 2 along with L, M, U, V. In block 3, I, Y, Z, L1, L2 from block 1 are global along with J, L, U, V from block 2. K, M, N, W, X, and L3 are local in block 3. Consider the statement, L3. The global variable Y will be replaced by the sum of the local variable X with the product of the global variables V, Z. The statement L4 in block 4 looks the same as L3 in block 3. However, the variable X in this case refers to the variable X in block 1 rather than the variable X in block 3; hence the effect of the statements will be different.
In block 5, the statement labeled L5 has an erroneous `GO TO L4` after the word `THEN`. The label L4 is defined in block 4 and has no meaning within block 5 since the blocks are disjoint. However, the `GO TO L1` is correct and will send control to the statement labeled L1 in block 1. This in effect will cause the program to re-enter block 2.

**Figure 8-1. Local and Global Identifiers**
9. INPUT/OUTPUT

9.1. GENERAL

Input and output operations are accomplished in UNIVAC 1108 ALGOL by means of library procedures. The two main ones, READ and WRITE, are more flexible than ordinary procedures written in ALGOL because the number of parameters in an actual call or even the order of the parameters is not rigidly specified. The general form of I/O call is:

\[
\langle I/O \text{ procedure}\rangle (\langle \text{device}\rangle, \langle \text{format identifier}\rangle, \langle \text{modifier list}\rangle, \langle \text{parameter list}\rangle, \langle \text{actual label list}\rangle)
\]

where \( \langle I/O \text{ procedure}\rangle \) is READ or WRITE;
\( \langle \text{device}\rangle \) specifies the external medium;
\( \langle \text{format identifier}\rangle \) is the name of the format specifying output editing or card layout for input;
\( \langle \text{modifier list}\rangle \) specifies parameters whose action is to output markers in the information which later may be used for positioning;
\( \langle \text{parameter list}\rangle \) is a list of I/O variables and expressions;
\( \langle \text{actual label list}\rangle \) specifies where control will be transferred in case of contingencies.

The two other important I/O procedures, POSITION and REWIND, are concerned exclusively with magnetic tape and tape-simulated drum operations.

9.2. FREE-FORMAT OUTPUT ON PRINTER AND CARD PUNCH

Arrays and values of expressions can be printed by simply calling the WRITE procedure in the following way:

\[
\text{WRITE(PRINTER, } v_1, v_2, \ldots, v_n)\]

where each \( v_i \) is an expression or an array identifier. In a similar manner, if the output is to be punched, the call is:

\[
\text{WRITE(PUNCH, } v_1, v_2, \ldots, v_n)\]

PRINTER and PUNCH are device names which specify the output unit to be used. If no device is named, PRINTER is assumed:

\[
\text{WRITE( } v_1, v_2, \ldots, v_n)\]

In the following description the word 'print' \( v \) is used in discussing the action of WRITE, but the word 'punch' may be substituted. Significant differences between the two devices are noted.

The action of WRITE is to evaluate the expressions in the order they are listed in the call and print their values in the following manner: except for string expressions, 10 values are printed on each line (6 per card if punching). Each value occupies a field of 12 character positions or columns. If the actual parameter is an array it is decomposed by columns. Each occurrence of WRITE begins printing on a new line.
Examples:

```algol
definition fig .

REAL ARRAY A(1:10)
*.

WRITE(PRINTER,A)
```

acts the same as

```algol
WRITE(PRINTER,A(1),A(2),...,A(10))
```

For multidimensional arrays the decomposition is such that the leftmost subscript varies most frequently. Thus the sequence

```algol
BOOLEAN ARRAY A(1:2,1:2,1:2)
*.

WRITE(PRINTER,A)
```

acts like

```algol
WRITE(PRINTER,A(1,1,1),A(2,1,1),A(1,2,1),A(2,2,1),
    A(1,1,2),A(2,1,2),A(1,2,2),A(2,2,2))
```

The expression or array element is printed in a form consistent with its type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>Integer form, right justified in the field. Includes a leading minus if the expression is negative. Leading zeros are not printed.</td>
</tr>
<tr>
<td>REAL and REAL 2</td>
<td>Both types are printed right justified in the form X.XXXX,±NN, where NN represents the power of ten, preceded by the appropriate sign. A negative number is preceded by a minus sign.</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Either TRUE or FALSE is left justified in the field.</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>The real and imaginary parts are each given a field as for REAL. Thus, only five expressions of type COMPLEX can be printed on the same line.</td>
</tr>
</tbody>
</table>

Strings are a slight exception in that they always start a new line. Whenever a string expression occurs as a parameter, the previous expressions, whether their number is a multiple of 10 or not, are printed. Then the string is printed on a new line. The next parameter will be printed on the following line. For example, if A and B are REAL and have the values 7.0 and 0.004 respectively, then the statement

```algol
WRITE('A=',A,'B=',B,'A OVER F',A/B)
```
would produce the following lines on the printer:

\[
\text{A=} \\
7.0000,00 \\
\text{B=} \\
4.0000,-03 \\
\text{A OVER B} \\
1.7500,03
\]

Example:

\[
\text{INTEGER ARRAY} \ A(1:15) \ \$ \\
\text{BOOLEAN ARRAY} \ B(1:2,1:2) \ \$ \\
\text{INTEGER} \ I,J \ \$. \\
\text{FOR} \ I=(1,1,15)\ DO \ A(I)=I-3 \ \$. \\
\text{FOR} \ I=(1,1,2) \ \text{DO} \ \text{FOR} \ J=(1,1,2) \ \text{DO} \ B(I,J)=I \ \text{GEQ} \ J \ \$. \\
\text{WRITE('VECTOR} \ A',A,'MATRIX} \ B',B) \ \$. \\
\]

produces the following output:

**VECTOR A**

\[
\begin{array}{cccccc}
-2 & -1 & 0 & 1 & 2 & 3 \\
8 & 9 & 10 & 11 & 12 & 13 \\
\end{array}
\]

**MATRIX B**

\[
\begin{array}{cccc}
\text{TRUE} & \text{TRUE} & \text{FALSE} & \text{TRUE} \\
12 & 12 & 12 & 12 \\
\end{array}
\]

9.3. **FREE-FORMAT INPUT FROM CARDS**

For reading punched cards in free-format mode, the procedure READ is called with device CARDS:

\[
\text{READ(CARDS, v1,v2,\ldots,v_n)}
\]

where each \( v_i \) is a variable or array identifier. Again, if no device is written, CARDS is assumed:

\[
\text{READ(v1, v2, \ldots, v_n)}
\]

This procedure reads the next input card and scans the information on it. Each constant on the card is assigned to the next parameter in the order it appears in the call.
Arrays are handled in the same manner as for WRITE. Constants on the cards must be punched in the same form as they appear in the ALGOL source language (see 2.3) with the exception that a comma(,) may be used in place of the ampersand (&). Constants on a card are delimited by one or more blanks and by the end of the card. Therefore, there is no restriction as to where a constant may appear on the card. If there is not enough information on the first card to satisfy the READ procedure, a second card is read, and so on. Any information not taken from the last card is lost (i.e., the next call to READ reads a new card). An * punched on a card causes the remainder of the card to be ignored. Otherwise, all 80 columns of the card are scanned for information. An example of a call on READ:

```
REAL A,B $
INTEGER COUNTER $
.
READ(CARDS,A,B,COUNTER) $
```

Data Card

```
-7.2  .099  362236
```

assigns the values -7.2 to A, .099 to B and 362236 to COUNTER. It is not necessary for the type of the constant on the card to match the type of the actual parameter. Transfer functions are used automatically if such functions are defined (see Appendix B).

9.4. LIST PARAMETERS – THE LIST DECLARATION

A LIST declaration associates a set of ordered expressions with a LIST identifier. The identifier used in the expressions must be defined prior to their inclusion in LIST declaration. Thus, the other declarations should precede the LIST declaration.

A LIST may include three kinds of elements:

- Expressions
- Array identifiers
- FOR clauses

Example:
```
REAL ARRAY A(1:N,1:N) $
INTEGER I,J $
LIST L1(FOR I=(1,1:N) DO FOR J=(1,1:N) DO A(I,J)) $
.
.
READ(L1) $
WRITE(L1) $
```

This example uses the same LIST for both input and output. Expressions in a list which are to be within the scope of the FOR statement are surrounded by parentheses and not by a BEGIN-END pair.
Another example utilizing LIST:

```algol
REAL X, XY $
INTEGER WIDDLE $
BOOLEAN ARRAY Q (1:10) $
STRING BEAN (36*5) $ 
INTEGER I $ 
LIST L(1.0, X, XY, FOR I=(1, 1, 5) DO(Q(I), BEAN(36*(I-1)+1, 36)), SQRT(WIDDLE**3)) $
```

The LIST L defines the following sequence of expressions:

1.0 X XY Q(1) BEAN(1, 36) Q(2) BEAN(37, 36) Q(3) BEAN(73, 36)
Q(4) BEAN(109, 36) Q(5) BEAN(145, 36) SQRT(WIDDLE**3)

Besides being allowable as parameters to READ and WRITE, lists may also be used as parameters to MAX and MIN. All of the elements in the list are treated as call-by-name parameters and are not evaluated until they are referenced.

One precaution is required when using a LIST. Do not use a list containing an iteration variable within the scope of an outer iteration using the same variable as in the following:

```algol
INTEGER I $ 
ARRAY ARGGGHA (1:10) $ 
LIST LISP(FOR I=10 STEP -1 UNTIL 1 DO ARGGGHA(I)) $
FOR I=(1, 2, 47) DO 
BEGIN 
  WRITE (LISP) $
END $
```

In this case the value of I would be changed in the course of the write statement causing an infinite loop.
9.5. FORMATTED OUTPUT – THE FORMAT DECLARATION

It is often desirable to print or punch information in a specific manner rather than to accept the positioning automatically provided by the WRITE procedure.

The FORMAT declaration, which is included with the other declarations at the beginning of the block, provides a means of specifying how a printed page (or punched card) is to be formatted. A format is a set of specifications that can be interpreted by the I/O procedures to control the editing of information. The format takes this form:

```
FORMAT <identifier>(<format specifications>)
```

The following lines specify two formats, FEIN and FTWAIN:

```
FORMAT FEIN(X10,D7.2,X5,F17.8,A1.1),
       FTWAIN(B6,S10,I5,X2,T14.9,A3,E)
```

A single format identifier may be included as a parameter in a call on WRITE, but its position in the call does not matter. For example, the two following calls in WRITE are equivalent.

```
REAL A,B
*
* WRITE(PRINTER,FEIN,A,B)
WRITE(A,B,FEIN)
```

A format specification consists of a series of editing and/or nonediting codes separated by commas. An editing code corresponds to a value to be printed and specifies how the value is to be edited. A nonediting code controls printing, spacing or insertion of blanks or constants into the print line. The action of WRITE, when a format is being used, is to pair each output expression with its corresponding editing code in the format. Non-editing codes are executed as they are encountered.
9.5.1. Nonediting Codes

In nonediting codes listed below, s and t are unsigned integers and w indicates the number of character positions. Two conventions are that As is the same as As.0 and A is the same as A0.0. For phases that require a t, both s and t must be less than 64.

<table>
<thead>
<tr>
<th>FORMAT CODE</th>
<th>ACTION(PRINTER)</th>
<th>ACTION(PUNCH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>As.t</td>
<td>Activate</td>
<td>Prints the line just edited. Skip s lines before printing and t lines after.</td>
</tr>
<tr>
<td>Es</td>
<td>Eject</td>
<td>Ejects the page to logical line s-1 if s-1 is below margin on current page. The next line to be printed, if it specifies A1.t, prints on line s. If s-1 is on the current page and is below the current line, Es skips to s-1 and the page is not ejected.</td>
</tr>
<tr>
<td>Xw</td>
<td>Expunge</td>
<td>Skips the next w character positions (i.e., inserts w blanks in the line).</td>
</tr>
<tr>
<td>'&lt; any string not containing a '&gt;'</td>
<td>Insert literal</td>
<td>Inserts the string enclosed in quotes into the line.</td>
</tr>
</tbody>
</table>

Table 9-1. Output Nonediting Codes
9.5.2. Editing Codes

The editing codes are the same for both printing and punching. Each code acts on one value to be printed. The \( w \) specifies the field width (that is, the total number of character positions to be used in the editing, including signs, decimal points and comma). If \( w \) is too small to do the proper numeric editing, two asterisks are printed and the value is edited according to R12.5. Any editing done beyond the edge of the output medium (132 or 128 columns on the printer, 80 columns on the card punch) is lost. The \( d \) is interpreted differently for different codes. In format codes that use no \( d \) (as Sw), \( w \) must be smaller than 4096. In codes that include \( d \), \( w \) must be enough larger than \( d \) to include at least the decimal point, a sign, and the exponent with its sign if there is one.

<table>
<thead>
<tr>
<th>FORMAT CODE</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bw Boolean</td>
<td>Prints <strong>TRUE</strong> or <strong>FALSE</strong> in the field, left justified. If the field is too short as much as possible is printed, e.g., B1 results in T or F.</td>
</tr>
<tr>
<td>Dw.d Decimal</td>
<td>Prints a decimal number with ( d ) places after the decimal point, right justified in the field, and with a leading minus sign if negative.</td>
</tr>
<tr>
<td>Iw.d Integer</td>
<td>Prints an integer number right justified in the field with a leading minus sign if negative. The integer is printed to the base ( d ) where ( d=0 ) and ( d=10 ) are equivalent. In the latter cases the .( d ) can be omitted. Range of ( d ): ( 2 \leq d \leq 10 ).</td>
</tr>
<tr>
<td>Rw.d Real</td>
<td>Prints ( d ) significant digits of a <strong>REAL</strong> or <strong>REAL 2</strong> variable in the form X.X . . . . X, NN for <strong>REAL</strong> or X.X . . . . X, NNN for <strong>REAL 2</strong>. A leading minus is printed if the number is negative. If the power of ten, NN or NNN, is negative it is preceded by a minus sign. Note that ( w ) must always exceed ( d ) by 6 or more (7 or more for <strong>REAL 2</strong>) to allow for ( \pm ), ( \pm NN ) or ( \pm NNN ).</td>
</tr>
<tr>
<td>Sw String</td>
<td>Prints the first ( w ) characters of a string left justified in the field. If the string is shorter than ( w ) characters, the rest of the field is space filled.</td>
</tr>
<tr>
<td>Tw.d Truncated</td>
<td>Prints a number with a decimal point right justified in the field. Only the first ( d ) significant digits are printed; a leading minus sign is printed if negative.</td>
</tr>
</tbody>
</table>

*Table 9–2. Output Editing Codes*

The type of the actual parameter is transferred to the type demanded by the editing code in any case for which there are transfer functions defined. A complex number is edited using two successive editing codes, the first for the real part and the second for the imaginary part.
The following examples illustrate the use of various editing codes:

```plaintext
REAL A, B $  
FORMAT F1(X2, D7.2, X5, R17.8, A1.1) $  
.  
A = 72.474 $  
B = -12345678 $  
WRITE(F1, A, B) $  
```

The above coding would print A and B as follows:

```
F1 (X2, D7.2, X5, R17.8, A1.1) $  
ΔΔΔΔ72.47ΔΔΔΔΔΔΔB=ΔΔΔ=1.2345678,-01  
1 blank line  
```

If F1 above were

```
F1('A=', D7.2, X5, 'B=', R17.8, A2.1)  
```

then the printout would be

```
A=Δ72.47ΔΔΔΔΔΔB=ΔΔΔ=1.2345678,-01  
2 blank lines
```

1 blank line

To compare the three real codes D, R and T, suppose

```plaintext
REAL A $  
FORMAT OK(D10.4, R10.4, T10.4, A1) $  
A = 0.001107 $  
WRITE(A, A, A, OK) $  
```

The printed line would then be as indicated below:

```plaintext
OK (D10.4, R10.4, T10.4, A1) $  
ΔΔΔΔΔ.0011Δ1.107,-03ΔΔΔ.001107  
1 blank line  
```
9.5.3. Repetition of Editing Codes

A single editing code (or a group of editing codes) can be used a number of times without actually repeating the code itself in the format statement. Three different methods can be used:

- Simple repetition – used when the number of repetitions required is known.
- Variable repetition – used when the number of repetitions depends on data.
- Indefinite repetition – used when the number of repetitions is indeterminate.

9.5.3.1. Simple Repetition

An editing code may be repeated by prefixing it with an unsigned nonzero integer constant which specifies how many times that code is to be repeated.

Example:

```
FORMAT F1(R16.8*R16.8*R16.8*A1)
```

is equivalent to

```
FORMAT F1(3R16.8*A1)
```

It is also possible to repeat a group of editing codes by enclosing them in parentheses and preceding this parenthetical group with an unsigned nonzero integer constant indicating the number of repetitions of the group. There is no limit to the depth of nesting, editing codes or groups of codes.

Given the declaration

```
BOOLEAN ARRAY BOOL(1:7,1:4)
```

the following format would permit printing the array elements with only one row per line

```
FORMAT FORM (7(6B7,A1))
```

9.5.3.2. Variable Repetition

A second type of repetition is the variable repeat. Instead of an integer, an arithmetic expression or Boolean expression enclosed in colons specifies the number of repeats. The Boolean values **TRUE** and **FALSE** are equivalent to one and zero, respectively.

Example:

```
INTEGER ARRAY A(1:N,1:M)
FORMAT F(:N:(:M:(R16.8)*A1))
```

would print the array one row per line (so M should be less than 9). The expression (N or M) is evaluated every time the variable repeat is encountered during the format scan. If N or M have a value of zero or less, the group of format codes under control of that repeat expression is skipped. Note that in this type of repetition, the codes to be repeated must be enclosed in parentheses even if there is only one such code.
9.5.3.3. Indefinite Repetition

A final variant of editing code repetition is the indefinite or unlimited repeat. This is accomplished by enclosing a group of format codes in parentheses without preceding this parenthetical expression with an integer constant. The innermost parenthetical group that is not preceded by an integer constant is unlimited and will be used repeatedly until the output list is exhausted.

Given:

```
FORMAT ONANDON (E1,'VECTOR B',A1,(D7.2,A1)) $
REAL 2 ARRAY B(1:M) $
then
WRITE(B,ONANDON) $
```

will produce the following on a new page

```
VECTOR B

XXXX.XX
XXXX.XX
XXXX.XX
XXXX.XX

As an extension of this feature note that the parentheses surrounding an entire format are indefinite repeats. If there are more values to print upon reaching the end of the format string the whole format is repeated. Writing stops when the two following conditions are satisfied: there are no more values to be edited, and the right parenthesis of an indefinite repeat is encountered. Any editing code encountered when there is nothing more to be edited is treated as Xw. Nonediting codes are honored.

Example:

```
REAL A,B $
FORMAT Z(5D7.2,A2.3,'ABOVE IS DEBUG IT'),A1) $
A = 17.00 $
B = -18.00 $
WRITE (A,A/2,B,B/2,Z) $
```

produces

```
A A 17.00 A A 18.00 A A 9.00
ABOVE IS DEBUG 1
```

A common error to watch for is the omission of an activation code within an indefinite repeat (or within a format declaration in general):

```
FORMAT QTE(E,(8R16.8),A1.2)
```
would skip to the top of the next page but would not print anything. This error could be corrected as follows:

```
FORMAT QTE(E,*((8R16.8,A1.2))
```

In this format, E is equivalent to E0 which has the effect of skipping to the bottom line of the current page. E0 followed by A1 will print on the top line of the next page.

Format codes to the right of an indefinite repeat or unlimited group can never be reached. The following output format:

```
FORMAT FORM9(X5,I5,D10.3,A1,(*I5,4D10.3,A1),X10,I5,A1,2)
```

used with a WRITE statement will cause the first two values to be printed according to I5 and D10.3. Since the inner parenthetical group is not preceded by an integer, printing will continue according to specifications within the parentheses until the output list is exhausted. The last three codes will never be reached.

The following example illustrates how versatile formats and lists can be. It prints an (N,M) array in a real format:

```
TEST,TEST
COMPILED BY 1c03 0005 ON 04/19/68 AT 10:37:43

S1
1 INTEGER M, N, I1, I2, I, J
2 REAL R
3 REAL ARRAY A(1:50,1:50)
4 FORMAT ALL(X11,ENTIER(MIN(I1+9,N))=I1-1:('COL',I3,X6),A1:('ROW',I3,ENTIER(MIN(I1+9,N))=I1-1:(R12.2),A1),A2)
5 LIST HAIR (FOR I1=(1,1,N) DO (FOR I2=(I1,1,M) DO A(I2,J)))"$
R = 0.0
M = 24$
N = 18$
FOR I = (1,1,N) DO
FOR J = (1,1,M) DO

B1
13 BEGIN
14 R = R + 1.0
15 A(I,J) = R

E1
16 END
17 WRITE (ALL,HAIR)"
```

START COMPIILATION TIME IS 10:37:42
END COMPIILATION PHASE 1 10:37:43
END COMPIILATION PHASE 2 10:37:43
F1
COMPIILATION COMPLETE
The WRITE statement using format ALL and list HAIR has the effect of printing the elements of a real array by rows with ten elements per line. The output has this form:

<table>
<thead>
<tr>
<th>ROW</th>
<th>COL 1</th>
<th>COL 2</th>
<th>COL 3</th>
<th>COL 4</th>
<th>COL 5</th>
<th>COL 6</th>
<th>COL 7</th>
<th>COL 8</th>
<th>COL 9</th>
<th>COL 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00000Def</td>
<td>2.00000Def</td>
<td>3.00000Def</td>
<td>4.00000Def</td>
<td>5.00000Def</td>
<td>6.00000Def</td>
<td>7.00000Def</td>
<td>8.00000Def</td>
<td>9.00000Def</td>
<td>10.00000Def</td>
</tr>
<tr>
<td>2</td>
<td>2.00000Def</td>
<td>3.00000Def</td>
<td>4.00000Def</td>
<td>5.00000Def</td>
<td>6.00000Def</td>
<td>7.00000Def</td>
<td>8.00000Def</td>
<td>9.00000Def</td>
<td>10.00000Def</td>
<td>11.00000Def</td>
</tr>
<tr>
<td>3</td>
<td>3.00000Def</td>
<td>4.00000Def</td>
<td>5.00000Def</td>
<td>6.00000Def</td>
<td>7.00000Def</td>
<td>8.00000Def</td>
<td>9.00000Def</td>
<td>10.00000Def</td>
<td>11.00000Def</td>
<td>12.00000Def</td>
</tr>
<tr>
<td>5</td>
<td>5.00000Def</td>
<td>6.00000Def</td>
<td>7.00000Def</td>
<td>8.00000Def</td>
<td>9.00000Def</td>
<td>10.00000Def</td>
<td>11.00000Def</td>
<td>12.00000Def</td>
<td>13.00000Def</td>
<td>14.00000Def</td>
</tr>
<tr>
<td>8</td>
<td>8.00000Def</td>
<td>9.00000Def</td>
<td>10.00000Def</td>
<td>11.00000Def</td>
<td>12.00000Def</td>
<td>13.00000Def</td>
<td>14.00000Def</td>
<td>15.00000Def</td>
<td>16.00000Def</td>
<td>17.00000Def</td>
</tr>
<tr>
<td>10</td>
<td>10.00000Def</td>
<td>11.00000Def</td>
<td>12.00000Def</td>
<td>13.00000Def</td>
<td>14.00000Def</td>
<td>15.00000Def</td>
<td>16.00000Def</td>
<td>17.00000Def</td>
<td>18.00000Def</td>
<td>19.00000Def</td>
</tr>
<tr>
<td>13</td>
<td>13.00000Def</td>
<td>14.00000Def</td>
<td>15.00000Def</td>
<td>16.00000Def</td>
<td>17.00000Def</td>
<td>18.00000Def</td>
<td>19.00000Def</td>
<td>20.00000Def</td>
<td>21.00000Def</td>
<td>22.00000Def</td>
</tr>
<tr>
<td>15</td>
<td>15.00000Def</td>
<td>16.00000Def</td>
<td>17.00000Def</td>
<td>18.00000Def</td>
<td>19.00000Def</td>
<td>20.00000Def</td>
<td>21.00000Def</td>
<td>22.00000Def</td>
<td>23.00000Def</td>
<td>24.00000Def</td>
</tr>
</tbody>
</table>

EXIT EXCO ALGOL LIBRARY.
9.6. FORMATTED INPUT

As with writing, a format may be included as a parameter to the READ procedure. A format tells how the card is laid out. The major advantage in using formats is that constants need no longer be delimited by blanks, and strings need not be enclosed in string quotes — the format specifies the 'fields' in which information lies. The discussion that follows is based on an example designed to illustrate most of the fundamentals of reading formatted cards.

The format declaration is the same as described in 9.5. The difference between a format being used as a parameter to WRITE and one being used as a parameter to READ is that the editing and nonediting codes are interpreted slightly differently. However, they are enough alike that in many cases the same format may be used with both procedures.

The following example illustrates reading data from cards according to a specified format. The information pertains to student records with each card having the following format:

<table>
<thead>
<tr>
<th>Column</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-5</td>
<td>Student number</td>
</tr>
<tr>
<td>6-7</td>
<td>Student initials</td>
</tr>
<tr>
<td>8-21</td>
<td>Student name</td>
</tr>
<tr>
<td>22</td>
<td>Status</td>
</tr>
<tr>
<td>23-24</td>
<td>Curriculum</td>
</tr>
<tr>
<td>38-44</td>
<td>Course name</td>
</tr>
<tr>
<td>47</td>
<td>Credit hours</td>
</tr>
<tr>
<td>60</td>
<td>Letter grade</td>
</tr>
</tbody>
</table>

The problem is to read the above data in a form that will make the manipulations easy and permit printing all the information. It is this type of problem which gives rise to the necessity of specifying the card format. The steps necessary to achieve this result are:

1. Read a card
2. Accept columns 1–5 as an integer, Student number
3. Accept columns 6–7 as a string, Student initials
4. Accept columns 8–21 as a string, Student name
5. Accept column 22 as an integer, Status
6. Accept columns 23–24 as an integer, Curriculum
7. Skip the next 13 columns
8. Accept columns 38–44 as a string, Course name
9. Skip 2 columns
10. Accept column 47 as an integer, Credit hours
11. Skip 12 columns
12. Accept column 60 as a string, Grade
The **FORMAT** declaration can be used to take care of all the above functions. For example, the format could be

```
FORMAT VAYDR EL( A, I5, S2, S14, I1, I2, X13, S7, X2, I1, X12, S1)
```

Note there is one entry in the format for each numbered line above. Each of the items in the above format is referred to as a 'format code'. Of course, the initial A is analogous to the terminal AS.t of the write and is required to activate the subsequent **READ** procedure.

A reasonable program segment for the above problem would be:

```
INTEGER STUDNO, STATI, CURCI, CREDS
STRING INITIALS(2), NAME(14), COURSE(7), GRADE(1)
FORMAT VAYREI(A, I5, S2, S14, I1, I2, X13, S7, X2, I1, X12, S1)
LIST FRIEDRICH(STUDNO, INITIALS, NAME, STATI, CURCI, COURSE, CREDS, GRADE)
READ(CARDS, FRIEDRICH, VAYREI)
```

Only two nonediting codes are permitted with an input format:

- **A** activation code – required as the first code of the format to activate the subsequent **READ** procedure.

- **Xw** – skip over w columns on the card

The editing codes (some of which are not in the example) have the following meanings, and w and d are restricted as before (see 9.5).

<table>
<thead>
<tr>
<th>FORMAT CODE</th>
<th>ACTION</th>
</tr>
</thead>
</table>
| Bw          | Boolean
| Dw.d       | Real
| Iw.d       | Integer
| Rw.d       | Real
| Sw         | String
| T w.d      | Real |

- **Bw** accepts Boolean information from the field – either **TRUE**, **FALSE**, or 1, 0.
- **Dw.d** accepts real information from the field. If the number is already real, (i.e., has a decimal point or exponent part) then that determines the decimal. Otherwise, a decimal point is inserted d places to the left of the right edge of the field.
- **Fw** accepts an unspecified number of values from the field. These numbers must be punched in free format mode; that is, values may be punched anywhere within w character positions.
- **Iw.d** accepts information from the field as being integer to the base d. d=0 is equivalent to d=10.
- **Rw.d** is same as Dw.d
- **Sw** accepts the whole field as a string.
- **T w.d** is same as Dw.d

*Table 9-3. Input Editing Codes*
9.7. FILE HANDLING

The general form of I/O call is:

\[ \text{<I/O Procedure>} \left( \text{<device>}, \text{<format>}, \text{<modifier list>}, \text{<parameter list>}, \text{<actual label list>} \right) \]

Where

\text{<I/O Procedure>} is either \text{READ} or \text{WRITE}
\text{<device>} is CARD, PRINTER, PUNCH, FILE (filename, location), APRINTER or APUNCH
\text{<modifier list>} see section 9.8.2.1
\text{<parameter list>} is a list of all of the I/O variables of lists
\text{<actual label list>} see section 9.8.3

When the device is FILE the above call takes either of two forms:

Sequential files

\[ \text{<I/O Procedure>} \left( \text{File(filename)}, \text{<modifier list>}, \text{<parameter list>}, \text{<actual label list>} \right) \]

Random Files

\[ \text{<I/O Procedure>} \left( \text{FILE(filename, location)}, \text{<modifier list>}, \text{<parameter list>}, \text{<actual label list>} \right) \]

In these forms \text{<filename>} is the internal name of the file. According to system requirements, the name must be at most 12 characters long, left justified, and space-filled. If the string is less than 12 characters long, the compiler supplies trailing blanks.

\text{<location>} is an integer specifying the location relative to the beginning of the file at which the I/O operation is to begin.

The I/O operation is unformatted whenever the device is FILE. In case of doubt concerning allowable operations on files, and for information concerning the assignment and use of files, see the "UNIVAC 1108 Executive Programmers Reference Manual", UP-4144 (current version). The ALGOL linkages permit the user complete access to the file handling capabilities of the 1108 Executive.

To simulate magnetic tape on other devices, such as drum or FASTRAND, all files can be handled as sequential access storage devices that can be parameters to \text{REWIND} and \text{POSITION} statements (see 9.8).

If a non-existent file is referenced, the compiler assigns a temporary FASTRAND mass storage file to that name. Since the information in a sequential file is written out in blocks containing various extra words that specify information about the block, one should be careful about accessing the same file both randomly and sequentially.
9.7.1. Sequential Files

If the device name `FILE` is followed by only one parameter, then the file is treated as if it were a sequential file. Information in such files is stored in blocks of 252 words or less. The format of these blocks is:

```
<table>
<thead>
<tr>
<th>H1</th>
<th>S4</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>LB</td>
<td>BSN</td>
</tr>
<tr>
<td>DATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>249</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- **N** - number of words in block. Must be 249 words or less.
- **LB** - last block flag.
  - = 0 for not last block.
  - = 1 for last block.
- **BSN** - block sequence number in logical record.

The `WRITE` statement writes one logical record. Information is packed in blocks with all but the last block containing 249 words of data. The last block (LB flag = 1) may contain less than 252 words. One logical record is the information written out by one write statement.

The `READ` procedure reads blocks of data from the device until all of the variables have been filled. All information in the last logical record read which is not used by that `READ` is lost, i.e., the next call on `READ` starts reading the following logical record.

See Section 9.8 for operations on sequential files.

9.7.2. Random Access Files

If the device name `FILE` is followed by two parameters, the first one is the name of the file, as before. The second is the location relative to the beginning of the file at which this `READ` or `WRITE` operation is to begin in the file.

Magnetic tape files cannot be referenced randomly. Integer, real and `BOOLEAN` variables take one word per variable. Real 2 and complex variables occupy two words. Strings are always left-justified in the first word in which they are written. Thus an N-character string occupies `ENTIER(N + 5)/6` words. No descriptor is written out with any of the information, so it is not difficult to determine how many words any block of information will occupy.
9.7.3. Alternate Symbionts

The 1108 Executive allows the user to define print and punch files other than the standard two. Information is these files is written out when the output device is free. In particular, the file will be written out at the end of the run. To use this ability from ALGOL, specify device APUNCH or A PRINTER with one parameter. This parameter is the internal name of the file to which the information is to be written pending output. If the file is a tape file, the output will not be written out automatically. This operation must be initiated by the appropriate Executive control statement.

9.8. OTHER DIRECTIVES

REWIND and POSITION operations are useful in manipulation of files. In addition supplementary marks can be made on a file in order to facilitate access to information.

9.8.1. REWIND

The form of the REWIND statement is:

REWIND (<file list>_1, INTERLOCK, < file list > 2)

where either < file list > may be empty. If < file list >_1 is empty then INTERLOCK should be left out.

This statement rewinds all of the files in < file list >_1 with interlock or if they are temporary non-tape files, it releases them. All of the files in < file list >_2 are rewound without interlock.

9.8.2. Modifiers and POSITION

9.8.2.1. Modifiers

A modifier list may be provided as a parameter to WRITE. This list may contain either EOF (< expression >) or KEY (< expression >) or both. Note that EOF and KEY mean the same thing as EOF (0) and KEY (0). If the modifier list contains KEY, then a KEY record is written preceding the usual record written out. If it contains EOF, then an EOF record is written after the usual record.

If < expression > is a string, the first six characters identify the record. The modifier EOI may also be used, in which case an end-of-information mark is written out after the usual record and EOF block.
9.8.2.2. **POSITION**

The procedure `POSITION` positions a file to a previously written KEY or EOF record, to the end of information, or advance it over a given number of ordinary records. The call is:

```
POSITION(FILE(filename), <position parameter>, <label list>)
```

where the position parameter is:

- `EOF (<expression>)`
- `-EOF (<expression>)`
- `KEY (<expression>)`
- `-KEY (<expression>)`
- integer expression
- `EOI`
- `-EOI`

The direction of positioning is indicated by the sign of the position parameter, positive for forward and negative for backward. If the position parameter is `EOI`, the file is positioned to the next EOI mark. If the position parameter is an integer expression, the command advances over that many logical records ignoring KEY records.

Key records are also ignored when encountered by a READ statement. Abnormal exits from the `POSITION` procedure are listed in section 9.8.3.

The procedure `POSITION` always positions over the record it is looking for in the direction indicated.

9.8.3. **Labels**

In many of the I/O operations unexpected situations may be encountered. In order to inform the user of their existence and to enable him to recover, several label parameters may be supplied to the procedure. These are used as alternate exits in abnormal situations. If more labels are provided than are expected, the procedure will respond with an *Improper Parameter* message.

9.8.3.1. **POSITION** Procedure

At most two parameters are allowed. The following defines the exits that occur in various situations:

<table>
<thead>
<tr>
<th>Parameter to <code>POSITION</code></th>
<th>Condition</th>
<th>Exit to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EOF</code></td>
<td>End of Information</td>
<td>First Label</td>
</tr>
<tr>
<td><code>KEY</code></td>
<td>End of Information</td>
<td>First Label</td>
</tr>
<tr>
<td>integer expression</td>
<td>EOF Record</td>
<td>First Label</td>
</tr>
<tr>
<td></td>
<td>End of Information</td>
<td>Second Label</td>
</tr>
</tbody>
</table>

If only one label is given, all exits to the second label go to the first. If no label is given, exit is made normally, that is to the next ALGOL statement.
9.8.3.2. **READ Procedure**

Control cards are identified by a master space (a 7–8 punch) in column 1. Except for the EOF control card, these cards are not read by an ALGOL program. However, the programmer can detect them by using labels as parameters to the **READ** procedure. In that case, if an attempt is made to read such a card, the **READ** procedure terminates reading and exits to the given label.

When an EOF card is encountered by the **READ** procedure, reading terminates. No new values are assigned to the remaining parameters in the input list. If a label is present as a parameter, the exit is made to that label. Otherwise, exit is made to the next ALGOL statement. The next time the **READ** procedure is called, it begins by reading the card after the EOF card. Thus, the purpose of EOF is to give the programmer a convenient method of separating sections of data of unknown length.

If a control card other than an EOF card is encountered, no more data cards can be read for that run.

At most three labels may be supplied as parameters to **READ**. If only one is specified all exits to the second label will be made to the first. If no third label is specified the program terminates upon encountering a situation that would require exiting to the third label.

The editing of the **READ** procedure is controlled in the following way, depending on the number of labels in the **READ** call:

<table>
<thead>
<tr>
<th>DEVICE</th>
<th>CONDITION</th>
<th>EXIT TO LABEL NO.</th>
<th>NO LABEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARD</td>
<td><strong>EOF</strong> Card</td>
<td>1</td>
<td>normal return</td>
</tr>
<tr>
<td></td>
<td>any other control card</td>
<td>2</td>
<td>program terminated</td>
</tr>
<tr>
<td></td>
<td>error in read</td>
<td>3</td>
<td>program terminated</td>
</tr>
<tr>
<td>FILE</td>
<td><strong>EOF</strong> record</td>
<td>1</td>
<td>normal exit</td>
</tr>
<tr>
<td></td>
<td>end of information</td>
<td>2</td>
<td>normal exit</td>
</tr>
<tr>
<td></td>
<td>error in read</td>
<td>3</td>
<td>program terminated</td>
</tr>
</tbody>
</table>

**Example:**

```
BEGIN ARRAY A(1:20)
LOCAL LABEL OK, FIN

READ (CARDS, A, OK, FIN)

OK: WRITE ('EOF - CARD READ OFF') $ GO TO READ-$
FIN: WRITE ('PROGRAM TERMINATED BY CONTROL CARD') $ END-$
```
9.8.3.3. **WRITE Procedure**

Only one label is allowed as a parameter to **WRITE**. Exit is made to this label if an attempt is made to write past the physical end of the FILE.

9.8.3.4. **MARGIN Procedure**

The procedure **MARGIN** provides the ALGOL user the means for controlling the form of the output of the PRINT symbiont (See "1108 Executive Programmer's Reference Manual" UP-4144). The form of the call on MARGIN is:

```
MARGIN(<control string>)
```

where `<control string>` is a string containing one or more control functions.

Spaces are ignored prior to the first, or between functions. Each function begins with a single letter, followed by a comma, followed by any special information required, and terminated by a period. The format of the information character string varies according to the function but must not contain a period.

The following control functions are allowed:

- **L** - Space printer to logical line \( nn \), where logical line is defined as the line number relative to the top margin setting (see M below). All line positioning and printing is performed within the defined margin settings. (The bottom logical line of a page is identical to the top logical line \(-1\) of the next page.) Positioning to a logical line on printers with space-print operation is to logical line \( n - 1 \); therefore when \( n = 1 \), the logical line setting is the last line of the current page. This is also true when \( n = 0 \), or when \( n \) is greater than the length of the logical page. When \( n \) is less than or equal to the current line of the current page, the succeeding page is positioned to the logical line \( n - 1 \). The format of this function is:

  - **L**, \( nn \),

- **H** - Initiate heading printing. This function provides the user with an automatic means of printing a heading on each succeeding page of his print file. The format of this function is:

  - **H**, option, page \( # \), text of heading

  If the option field contains the letter X, a page and date will not be printed as part of the heading. Option n turns the heading off. A page count is maintained by the processing symbiont. When the **page #** field is blank, the page count current to the field is used to begin page numbering. When coded, **page #** is made the page number. In addition to the page number, the current date is included in the heading, and both will appear in the upper right corner of each page. This position of the heading is the second line above logical line 1. If the upper margin is one line or non-existent, no heading is printed. As many as 17 words of heading text may be supplied.
M — Set margins. This function supplies the information for re-adjusting page length and top and bottom margins. The standard print page definition is 66 lines per page with a top margin setting of six lines, and a bottom margin setting of three lines. Note that the top and bottom margins refer to the number of blank lines at the top and bottom of the page respectively. Thus the standard margin setting is 66, 6, 3 giving 57 printable lines. This page definition is assumed at the beginning of each print file. When the M function is used, a page alignment procedure is initiated with the page length parameter. This function is also used to return to the standard page length. The format of this function is:

M, length, top, bottom

W — Set maximum line width. The standard of 22 words (132 characters) is assumed unless the W control is used. The format of the function is:

W, width

where width specifies the maximum line width in words.

S — Special form request. This function enables the user to instruct the operator to load a special form required to process the print or punch file. The format of this function is:

S, message text

where the message text can be up to ten words long. When this function is encountered by the processing symbiont, the message is displayed on the operator's console in the form:

run ID/filename c/u options
message text

The user's message text is displayed on the line following the symbiont message. The options available to the operator for answering the message depend on the symbiont. The following options are included in the 0755 HSP, Card Punch and the 1004 Printer and Card Punch symbionts:

A — Begin processing the output file.

Q — Return file to symbiont queue. The print or punch file will be passed temporarily and placed behind the next file of this symbiont queue.
10. OPERATION

10.1. SOURCE CARD FORMAT

The source language statements to the compiler must come initially from punched cards. Only columns 1–72 are read for information in free format and anything following column 72 is considered to be a (space) delimiter. Columns 73–80 can be used for any purpose desired, e.g., short comments or serial identification. There is no restriction on placing statements on a card but the usual practice is to arrange them for easy reading and modification. The full 80 columns may be utilized for input data at execution time.

10.2. OPERATING INSTRUCTIONS

The ALGOL compiler operates like all the processors in the UNIVAC 1108 Executive System and, besides the standard options, includes some unique to itself. The available options are:

A Accept the results of compilation even if errors were detected.

S Single spaced listing of ALGOL source statements.

L The compiled assembly language instructions are listed along with the source code.

N (or lack of any other print option) Suppress all printing by the processor. If ‘N’, disregard any other print option.

T Print the timing for phases 1 and 2 of compilation.

X Abort the run immediately if any error is found.

Z Delete the formation of run-time diagnostic information.

O,R References to subscripted variables normally generate a call to a library procedure. This procedure, besides calculating the proper address, also checks that the requested operation is legal (i.e., that the subscript variables are in the range of the declaration). With the R option (remove checking) this checking is not done but the address calculation is, thus giving greater speed to the object program. The O option (open) is even faster in that the necessary coding to calculate the address is compiled in line, thus removing the call and return times from the reference. Of course, the O option requires more main storage. When the program is working, and if the subscript expressions are not data-dependent, then the R option should be used. If main storage permits, the O option should be used. Neither should be used when the program is being debugged.
# APPENDIX A. BASIC SYMBOLS AND THEIR CARD CODES

This appendix lists the basic symbols of UNIVAC 1108 ALGOL 60, with the corresponding symbol for the reference language and the punched card code.

<table>
<thead>
<tr>
<th>REFERENCE LANGUAGE</th>
<th>UNIVAC 1108 ALGOL 60</th>
<th>CARD CODE *</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>12</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>≥</td>
<td></td>
<td></td>
</tr>
<tr>
<td>≤</td>
<td>≤</td>
<td></td>
</tr>
<tr>
<td>≥</td>
<td>≥</td>
<td>11-4-8</td>
</tr>
<tr>
<td>≤</td>
<td>≤</td>
<td>0-1</td>
</tr>
<tr>
<td>GTR</td>
<td>GTR</td>
<td>0-1</td>
</tr>
<tr>
<td>LEQ</td>
<td>LEQ</td>
<td></td>
</tr>
<tr>
<td>EQL</td>
<td>EQL</td>
<td></td>
</tr>
<tr>
<td>EQIV</td>
<td>EQIV</td>
<td></td>
</tr>
<tr>
<td>IMPL</td>
<td>IMPL</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>NOT</td>
<td></td>
</tr>
<tr>
<td>GO TO</td>
<td>GO TO or GOTO</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>IF</td>
<td></td>
</tr>
<tr>
<td>THEN</td>
<td>THEN</td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td>FOR</td>
<td>FOR</td>
<td></td>
</tr>
<tr>
<td>DO</td>
<td>DO</td>
<td>0-3-8</td>
</tr>
<tr>
<td>, (comma)</td>
<td>, (comma)</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>12-3-8</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;</td>
<td>2-8</td>
</tr>
</tbody>
</table>

* Symbols for which a card code is not specified are reserved identifiers.
## Reference Language

<table>
<thead>
<tr>
<th>Reference Language</th>
<th>UNIVAC 1108 ALGOL</th>
<th>Card Code *</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>; or .</td>
<td>5–8 or 12–3–8 12–3–8</td>
</tr>
<tr>
<td>$</td>
<td>$ or ;</td>
<td>11–3–8 or 11–6–8</td>
</tr>
<tr>
<td>:=</td>
<td>= or :=</td>
<td>3–8 or 5–8 3–8</td>
</tr>
<tr>
<td><strong>STEP</strong></td>
<td><strong>STEP</strong></td>
<td></td>
</tr>
<tr>
<td><strong>UNTIL</strong></td>
<td><strong>UNTIL</strong></td>
<td></td>
</tr>
<tr>
<td><strong>WHILE</strong></td>
<td><strong>WHILE</strong></td>
<td></td>
</tr>
<tr>
<td><strong>COMMENT</strong></td>
<td><strong>COMMENT</strong></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td>0–4–8</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
<td>12–4–8</td>
</tr>
<tr>
<td>[</td>
<td>[ or ]</td>
<td>0–4–8 or 12–5–8</td>
</tr>
<tr>
<td>]</td>
<td>] or ]</td>
<td>12–4–8 or 11–5–8</td>
</tr>
<tr>
<td>' (apostrophe)</td>
<td>' (apostrophe)</td>
<td>4–8</td>
</tr>
<tr>
<td><strong>BEGIN</strong></td>
<td><strong>BEGIN</strong></td>
<td></td>
</tr>
<tr>
<td><strong>END</strong></td>
<td><strong>END</strong></td>
<td></td>
</tr>
<tr>
<td><strong>OWN</strong></td>
<td><strong>OWN</strong></td>
<td></td>
</tr>
<tr>
<td><strong>BOOLEAN</strong></td>
<td><strong>BOOLEAN</strong></td>
<td></td>
</tr>
<tr>
<td><strong>INTEGER</strong></td>
<td><strong>INTEGER</strong></td>
<td></td>
</tr>
<tr>
<td><strong>REAL</strong></td>
<td><strong>REAL</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ARRAY</strong></td>
<td><strong>ARRAY</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SWITCH</strong></td>
<td><strong>SWITCH</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PROCEDURE</strong></td>
<td><strong>PROCEDURE</strong></td>
<td></td>
</tr>
<tr>
<td><strong>LABEL</strong></td>
<td><strong>LABEL</strong></td>
<td></td>
</tr>
<tr>
<td><strong>VALUE</strong></td>
<td><strong>VALUE</strong></td>
<td></td>
</tr>
<tr>
<td>&lt; (for complex 12–6–8 constants)</td>
<td>12–6–8</td>
<td></td>
</tr>
<tr>
<td>&gt; constants)</td>
<td>6–8</td>
<td></td>
</tr>
</tbody>
</table>

*Symbols for which a card code is not specified are reserved identifiers.*

In addition, the following reserved identifiers have been introduced into the language:

**LIST**
**FORMAT**
**EXTERNAL**
**OTHERWISE**
**LOCAL**
**GO**
**TO**
**XOR**
**STRING**
**COMPLEX**
**REAL 2**
APPENDIX B. STANDARD PROCEDURES AND TRANSFER FUNCTIONS

The following procedures are available for use without being declared. The names are not reserved identifiers and may be redefined in any block. The * indicates "not applicable"

<table>
<thead>
<tr>
<th>NAME</th>
<th>NO. OF PARAMETERS</th>
<th>TYPES OF PARAMETERS</th>
<th>RESULT</th>
<th>TYPE OF RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>1</td>
<td>INTEGER, REAL, REAL 2, COMPLEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALPHABETIC</td>
<td>1</td>
<td>STRING</td>
<td>TRUE if the string consists of all spaces or alphabets (A-Z); FALSE otherwise</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>ARCCOS</td>
<td>1</td>
<td>REAL, REAL 2</td>
<td>Arccos(x)</td>
<td>REAL, REAL 2</td>
</tr>
<tr>
<td>ARCSIN</td>
<td>1</td>
<td>REAL, REAL 2</td>
<td>Arcsin(x)</td>
<td>REAL, REAL 2</td>
</tr>
<tr>
<td>ARCTAN</td>
<td>1</td>
<td>REAL, REAL 2</td>
<td>Arctan(x)</td>
<td>REAL, REAL 2</td>
</tr>
<tr>
<td>CARDS</td>
<td>0</td>
<td>*</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>CLOCK</td>
<td>0</td>
<td>*</td>
<td>Present time of day in seconds since 00:00</td>
<td>INTEGER</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>2</td>
<td>INTEGER, REAL</td>
<td>The complex number x+iy</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>NAME</td>
<td>NO. OF PARAMETERS</td>
<td>TYPES OF PARAMETERS</td>
<td>RESULT</td>
<td>TYPE OF RESULT</td>
</tr>
<tr>
<td>------------</td>
<td>------------------</td>
<td>---------------------------</td>
<td>-------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>COS</td>
<td>1</td>
<td>REAL, REAL 2, COMPLEX</td>
<td>Cos(x), Cos(x), Cos(x)</td>
<td>REAL, REAL 2, COMPLEX</td>
</tr>
<tr>
<td>COSH</td>
<td>1</td>
<td>REAL, REAL 2, COMPLEX</td>
<td>Cosh(x), Cosh(x), Cosh(x)</td>
<td>REAL, REAL 2, COMPLEX</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>1</td>
<td>INTEGER, REAL</td>
<td>Real 2 representation of x</td>
<td>REAL 2</td>
</tr>
<tr>
<td>DRUM</td>
<td>1</td>
<td>INTEGER</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>ENTIER</td>
<td>1</td>
<td>REAL, REAL 2</td>
<td>Integer part of x</td>
<td>INTEGER</td>
</tr>
<tr>
<td>EOF</td>
<td>0 or 1</td>
<td>Any expression</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>EOI</td>
<td>0</td>
<td>*</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>EXP</td>
<td>1</td>
<td>REAL, REAL 2, COMPLEX</td>
<td>Exp(x), Exp(x), Exp(x)</td>
<td>REAL, REAL 2, COMPLEX</td>
</tr>
<tr>
<td>IMAGINARY</td>
<td>1</td>
<td>COMPLEX</td>
<td>Imaginary part of x</td>
<td>REAL</td>
</tr>
<tr>
<td>INTEGER</td>
<td>1</td>
<td>REAL, REAL 2</td>
<td>Entier (x+0.5)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>KEY</td>
<td>0 or 1</td>
<td>Any expression</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>LENGTH</td>
<td>1</td>
<td>STRING</td>
<td>Length of string</td>
<td>INTEGER</td>
</tr>
<tr>
<td>LN</td>
<td>1</td>
<td>REAL, REAL 2, COMPLEX</td>
<td>Ln(x), Ln(x), Ln(x)</td>
<td>REAL, REAL 2, COMPLEX</td>
</tr>
<tr>
<td>MARGIN</td>
<td>3 or 4</td>
<td>INTEGER first 3</td>
<td>(see section 9)</td>
<td>*</td>
</tr>
<tr>
<td>MAX</td>
<td>List of expressions</td>
<td>REAL, INTEGER</td>
<td>Algebraic largest element of list</td>
<td>REAL</td>
</tr>
<tr>
<td>MIN</td>
<td>List of expressions</td>
<td>REAL, INTEGER</td>
<td>Algebraic smallest element of list</td>
<td>REAL</td>
</tr>
<tr>
<td>NAME</td>
<td>NO. OF PARAMETERS</td>
<td>TYPES OF PARAMETERS</td>
<td>RESULT</td>
<td>TYPE OF RESULT</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>---------------------</td>
<td>---------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>MOD</td>
<td>2</td>
<td>INTEGER</td>
<td>x(mod y)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>1</td>
<td>STRING</td>
<td>TRUE if the string is acceptable to the string-integer transfer function; FALSE otherwise</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>POSITION</td>
<td>Special list</td>
<td>*</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>PRINTER</td>
<td>0</td>
<td>*</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>PUNCH</td>
<td>0</td>
<td>*</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>RANK</td>
<td>1</td>
<td>STRING</td>
<td>The field data equivalent of the first character of the string</td>
<td>INTEGER</td>
</tr>
<tr>
<td>READ</td>
<td>Special list</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>REAL</td>
<td>1</td>
<td>INTEGER, REAL 2</td>
<td>Real representation of x</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMPLEX</td>
<td>Real part of x</td>
<td>REAL</td>
</tr>
<tr>
<td>REWIND</td>
<td>Special list</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>SIGN</td>
<td>1</td>
<td>INTEGER</td>
<td>1 x &gt; 0</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 x = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1 x &lt; 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIN</td>
<td>1</td>
<td>REAL, REAL 2, COMPLEX</td>
<td>Sin(x)</td>
<td>REAL, REAL 2, COMPLEX</td>
</tr>
</tbody>
</table>
## UNIVAC 1108 ALGOL

### Appendix B

#### SECTION:

### NAME | NO. OF PARAMETERS | TYPES OF PARAMETERS | RESULT | TYPE OF RESULT
---|---|---|---|---
SINH | 1 | REAL, REAL 2, COMPLEX | Sinh(x), Sinh(x), Sinh(x) | REAL, REAL 2, COMPLEX
SQRT | 1 | REAL, REAL 2, COMPLEX | Sqrt(x), Sqrt(x), Sqrt(x) | REAL, REAL 2, COMPLEX
TAN | 1 | REAL, REAL 2, COMPLEX | Tan(x), Tan(x), Tan(x) | REAL, REAL 2, COMPLEX
TANH | 1 | REAL, REAL 2, COMPLEX | Tanh(x), Tanh(x), Tanh(x) | REAL, REAL 2, COMPLEX
TAPE | 1 | INTEGER | None | *
WRITE | Special list | * | * | *

The following transfer functions transfer an expression of one type to another type. These functions are evoked automatically by the compiler whenever necessary. Functions for which the arguments are listed may be called explicitly.

### FUNCTION | TYPE OF ARGUMENT | TRANSFERRED TO TYPE
---|---|---
REAL(X) | INTEGER | REAL
DOUBLE(X) (intrinsic) | INTEGER | REAL
COMPLEX(REAL(X),0) | INTEGER | COMPLEX
INTEGER(X) | REAL | INTEGER
DOUBLE(X) | INTEGER | REAL
COMPLEX(X,0) | INTEGER | REAL
INTEGER(X) | REAL 2 | INTEGER
REAL(X) | STRING | COMPLEX
COMPLEX(REAL(X),0) (intrinsic) | STRING | INTEGER
APPENDIX C. ERROR MESSAGES

C1. COMPILATION ERRORS

The following is a list of error messages that can occur during compilation with an explanation of what may be the cause of an error. An * is printed under the approximate location of the offending syntax, but in some cases this may not be of help; for example, a missing left parenthesis is not found until the whole statement has been scanned. Spurious error messages may be printed for particularly malformed programs. These usually disappear after the first few errors have been eliminated.

<table>
<thead>
<tr>
<th>ERROR</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal character pair</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Constant too large</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Improper block structure</td>
<td>A declaration has occurred other than at the head of a block</td>
</tr>
<tr>
<td>Improper declaration</td>
<td>An element in the identifier list is not an identifier</td>
</tr>
<tr>
<td>Duplicate declaration/specification</td>
<td>A name in the identifier list has already been defined in this block</td>
</tr>
<tr>
<td>Improper declaration/specification</td>
<td>The identifier list is malformed</td>
</tr>
<tr>
<td>Improper specification</td>
<td>OWN, LOCAL, or EXTERNAL has been used in a procedure specification part</td>
</tr>
<tr>
<td>Improper specification</td>
<td>The name is not a formal parameter</td>
</tr>
<tr>
<td>Improper own declaration</td>
<td>OWN not followed by INTEGER, REAL, REAL2, COMPLEX, ARRAY</td>
</tr>
<tr>
<td>Improper external declaration</td>
<td>EXTERNAL not followed by &lt;type&gt;, FORTRAN, NONRECURSIVE or PROCEDURE</td>
</tr>
<tr>
<td>Duplicate value specification</td>
<td>The parameter has already been specified as value</td>
</tr>
<tr>
<td>Improper label specification</td>
<td>It does not occur in a specification part of a procedure</td>
</tr>
<tr>
<td>Improper value specification</td>
<td>It does not occur in a procedure specification part</td>
</tr>
<tr>
<td>Improper array declaration</td>
<td>Bound pairs malformed</td>
</tr>
<tr>
<td>Improper array declaration</td>
<td>Bound pairs either not separated by a comma or a : missing</td>
</tr>
<tr>
<td>ERROR</td>
<td>EXPLANATION</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Improper list declaration</td>
<td>A malformed list</td>
</tr>
<tr>
<td>Improper switch declaration</td>
<td>Malformed switch list</td>
</tr>
<tr>
<td>Improper switch declaration</td>
<td>The '=' is missing</td>
</tr>
<tr>
<td>Improper procedure declaration</td>
<td>The next symbol after the procedure name is not a ( or $</td>
</tr>
<tr>
<td>Improper procedure parameter</td>
<td>A formal parameter is not an identifier</td>
</tr>
<tr>
<td>Duplicate procedure parameter</td>
<td>There is another formal parameter of this name already</td>
</tr>
<tr>
<td>Improper parameter delimiter</td>
<td>The parameter delimiter is neither a comma nor ) &lt;letter string&gt; : (</td>
</tr>
<tr>
<td>Improper procedure specification</td>
<td>This parameter has been named in a value specification</td>
</tr>
<tr>
<td>Improper label definition</td>
<td>A numeric or other malformed label</td>
</tr>
<tr>
<td>Duplicate label definition</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Improper format phrase</td>
<td>The word field is too large</td>
</tr>
<tr>
<td>Improper format phrase</td>
<td>A string field has more than 132 char.</td>
</tr>
<tr>
<td>Improper format phrase</td>
<td>An extra )</td>
</tr>
<tr>
<td>Improper repeat phrase</td>
<td>A zero repeat phrase</td>
</tr>
<tr>
<td>Improper repeat phrase</td>
<td>A noninteger expression as a variable repeat phrase</td>
</tr>
<tr>
<td>Undefined format symbol</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Improper string declaration</td>
<td>Expression for length is malformed</td>
</tr>
<tr>
<td>Improper string array declaration</td>
<td>Either length expression or subscript pairs are malformed</td>
</tr>
<tr>
<td>Improper procedure call</td>
<td>Incorrect number of arguments for library procedure</td>
</tr>
<tr>
<td>Improper procedure call</td>
<td>Arguments for library procedure are of incorrect type</td>
</tr>
<tr>
<td>Improper procedure assignment</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Improper IF statement</td>
<td>Malformed conditional statement</td>
</tr>
<tr>
<td>Improper IF statement</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Improper use of THEN</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Improper use of ELSE</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Error</td>
<td>Explanation</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>Improper FOR statement</td>
<td>Malformed FOR list</td>
</tr>
<tr>
<td>Improper GO statement</td>
<td>GO not followed by a designational expression</td>
</tr>
<tr>
<td>Improper GO statement</td>
<td>A malformed designational expression</td>
</tr>
<tr>
<td>Extra right parenthesis</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Extra left parenthesis</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Missing operator</td>
<td>Implied multiplication has been used</td>
</tr>
<tr>
<td>Missing operator</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Missing operand</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Extra END</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Missing END</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Improper use of //operator</td>
<td>One or both operands are not integer</td>
</tr>
<tr>
<td>Improper assignment statement</td>
<td>The left-hand side is not a variable</td>
</tr>
<tr>
<td>Undefined transfer function</td>
<td>The transfer function called for by this statement is not defined</td>
</tr>
<tr>
<td>Improper use of a list identifier</td>
<td>List used in other than a procedure call</td>
</tr>
<tr>
<td>Improper use of label</td>
<td>A label appears out of context</td>
</tr>
<tr>
<td>Improper use of a reserved identifier</td>
<td>Reserved identifier appears out of context</td>
</tr>
<tr>
<td>Improper use of an array identifier</td>
<td>Array identifier appears out of context</td>
</tr>
<tr>
<td>Undefined relational operand</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Improper string expression</td>
<td>Malformed string expression</td>
</tr>
<tr>
<td>Misplaced semicolon</td>
<td>A semicolon ($) appears out of context</td>
</tr>
<tr>
<td>Misplaced comma</td>
<td>A comma appears out of context</td>
</tr>
<tr>
<td>Undefined variable</td>
<td>Reference is made to an undeclared variable</td>
</tr>
<tr>
<td>Misplaced colon</td>
<td>A colon appears out of context</td>
</tr>
<tr>
<td>Improper correction</td>
<td>Correction card out of order</td>
</tr>
<tr>
<td>Compiler capacity exceeded</td>
<td>Internal tables of compiler exceeded</td>
</tr>
</tbody>
</table>
C2. RUN-TIME ERRORS

An error during execution results in the printing of an error message, the name of the library procedure involved, if applicable, and the line number of the ALGOL program at which execution was currently taking place. The program is then terminated. The following is the list of the possible error messages.

<table>
<thead>
<tr>
<th>ERROR</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect number of arguments</td>
<td>Incorrect number of arguments to an ALGOL or library procedure</td>
</tr>
<tr>
<td>Memory capacity exceeded</td>
<td>Space for dynamic storage of variables has been exceeded</td>
</tr>
<tr>
<td>Bad input/checksum error</td>
<td>Tape or drum hardware error, or possibly an attempt to read a non-ALGOL formatted tape</td>
</tr>
<tr>
<td>Undefined type conversion</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Insufficient data for program</td>
<td>Non-EOF control card read with no exit label in READ call</td>
</tr>
<tr>
<td>Improper parameter</td>
<td>A formal and actual parameter do not agree in either type or kind</td>
</tr>
<tr>
<td>Improper array declaration</td>
<td>A lower bound expression is greater than the corresponding upper bound expression</td>
</tr>
<tr>
<td>Improper string declaration</td>
<td>The string length is negative or greater than 4095</td>
</tr>
<tr>
<td>Unrecoverable tape/drum error</td>
<td>Hardware fault</td>
</tr>
<tr>
<td>Attempt to pass end of record</td>
<td>Record is shorter than the input list</td>
</tr>
<tr>
<td>Constant out of range</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Characteristic overflow</td>
<td>A real number of magnitude greater than $10^{38}$ generated</td>
</tr>
<tr>
<td>Attempted division by zero</td>
<td>Self explanatory</td>
</tr>
<tr>
<td>Improper number of dimensions</td>
<td>Reference to a subscripted variable has a different number of subscript positions than was declared for the array</td>
</tr>
<tr>
<td>Subscript out of range</td>
<td>Subscript expression is out of the range of the declaration</td>
</tr>
<tr>
<td>Result undefined</td>
<td>Arithmetic library function not defined for argument</td>
</tr>
<tr>
<td>Argument out of range</td>
<td>Argument out of range for a meaningful result for an arithmetic library procedure</td>
</tr>
<tr>
<td>Illegal character</td>
<td>Data card has an illegal character</td>
</tr>
<tr>
<td>Illegal format phrase</td>
<td>Self explanatory</td>
</tr>
</tbody>
</table>
APPENDIX D. EXAMPLES OF PROGRAMS

This appendix contains some simple examples illustrating the use of UNIVAC 1108 ALGOL 60. Each has been run on the 1108 and some sample input and results are shown.

BEGIN
COMMENT
EXAMPLE 1
CALCULATION OF VALUE OF ARITHMETIC EXPRESSION WITH READ IN VARIABLES
REAL A,B,C $ INTEGER TOILL $
READ (CARDS,A,B,C) $ TOILL = A+B**C/A $ WRITE (PRINTER,A,B,C,TOILL) $
DATA
5 6.2 1.222
RESULTS:
5.0000,+00 6.2000,+00 1.2220,+00

BEGIN
COMMENT
EXAMPLE 2
CALCULATION OF SQUARE ROOT, B, OF A REAL NUMBER, A, WITH 6 DIGITS ACCURACY BY NEWTON-RAPHSON ITERATION
REAL A,B,OLDB $
READ (CARDS,A) $ OLDB = 1.0 $
FOR B = 0.5*(A/OLDB+OLDB) WHILE ABS(B-OLDB) GTR 10**(-6)*B DO OLDB = B $ WRITE (PRINTER,A,B) $
END PROGRAM $
DATA
5.77777
RESULTS:
5.7778,+00 2.4037,+00
BEGIN
COMMENT EXAMPLE 3
VALUE OF A POLYNOMIAL Y=B(0)+B(1)*X+...+B(N)*X**N
REAL X,Y
INTEGER K,N
READ (CARDS,N)
COMMENT DEGREE OF POLYNOMIAL READ FROM CARDS. INNER BLOCK PERFORMS
READING OF COEFFICIENTS AND CALCULATIONS AND PRINTING OF
RESULTS
BEGIN
REAL ARRAY B(0:N)
READ (CARDS,B)
READ (CARDS,X)
Y = B(N)
FOR K=N-1 STEP -1 UNTIL 0 DO Y = Y*X+B(K)
WRITE (PRINTER,'VALUE OF A POLYNOMIAL OF DEGREE','N=','N,
'COEFFICIENTS',B,'X=',X,'Y=',Y)
END CALCULATION
END PROGRAM

DATA
4
1.223 3.5 7.52 -4.02 -3.5
5.55

RESULTS:
VALUE OF A POLYNOMIAL OF DEGREE
N= 4

COEFFICIENTS
1.2230E+00 3.5000E+00 7.5200E+00 -4.0200E+00 -3.5000E+01
X=
5.5500E+00
Y=
-3.2220E+04
BEGIN
COMMENT
EXAMPLE 4
PROGRAM WITH A REAL PROCEDURE, BIG, WHICH FINDS THE LARGEST OF THE N LOWER-INDEXED ELEMENTS (STARTING WITH INDEX=1) OF A ONE-DIMENSIONAL ARRAY, A, WITH POSITIVE ELEMENTS

REAL PROCEDURE BIG(N,A) $
VALUE N $
INTEGER N $
REAL ARRAY A $
BEGIN
INTEGER B $
REAL C,D $
B = 1 $
D = A(1) $
L:
C = D - A(B+1) $
IF C LSS 0 THEN D = A(B+1) $
B = B+1 $
IF B LSS N THEN GO TO L $
BIG = D $
END BIG $
REAL ARRAY F(1:50) $
REAL H,K $
READ (CARDS,F) $
COMMENT CALL OF BIG TO FIND THE LARGEST OF THE 20 LOWER ELEMENTS OF F $
H = BIG(20,F) $
WRITE (PRINTER,H) $
COMMENT LARGEST ELEMENT IN F $
K = BIG(50,F) $
COMMENT USE OF BIG IN MORE COMPLEX EXPRESSION $
H = H + BIG(10,F)/K * BIG(15,F) $
WRITE (PRINTER,H,K) $
END PROGRAM $

DATA
1.22 3.55 1 22.2 0.5 7.2 8.12 21.4 4.1 22.5 0.422
55.2 0.12345 5.88 3.55 7.53 4 5 2 3 1 77 5 22.1
5.1 2.3 3.2 4.2 9.85 8.99 5.66 66 44 11 2 44.7
55.12 44.1 2.89 7.521 8.56 5.42 4.88 6.789 5.423
7.1234 9.753 8.741 5 6

RESULTS:
5.5200,+01
7.1330,+01 7.7000,+01
Example 5. Newton's Method of Successive Approximations

Given: An area $A$ defined by a circular arc of radius $r$ and its chord.

Required: Find the value of angle $x$ subtended by the arc.

Solution: The relationship between $A$ and $x$ is:

$$A = \frac{r^2}{2} (x - \sin x)$$

Like many practical problems, this one has no analytic solution. However, methods have been developed to find approximate solutions to such problems. The method to be used here is called Newton's Method. If the solution $x$ to

$$f(x) = 0$$

is to be found, then a sequence of values approximating the solution $x$ is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$  

For this problem

$$f(x_n) = \frac{1}{2}r^2(x_n - \sin x_n) - A$$

and

$$f'(x_n) = \frac{1}{2}r^2(1 - \cos x_n).$$
Therefore, using elementary algebra, the approximation scheme is

\[ x_{n+1} = x_n - \frac{x_n - \sin x_n - 2A/r^2}{1 - \cos x_n} \]

This equation is solved repeatedly, each time with the previous value of \( x_{n+1} \) substituted for \( x_n \) to compute a new value for \( x_{n+1} \). The second term of the equation is the difference between successive approximations.

When this difference becomes less than some specified value, the sequence of approximations is said to have converged to a solution. The iteration procedure is then terminated and the problem is considered solved.

Practical considerations place a limitation on the number of iterations permitted. If the sequence of approximations does not converge within a prescribed number of iterations, the procedure is terminated and the approximate solution is rejected.

The conditions used in this example are:

- Area = 1.5
- Radius = 5.0

The first approximation is \( x_1 = 1.0 \). The iteration procedure is then performed for a maximum of nine iterations. If the successive approximations differ by less than 0.00001, then the sequence of approximations is considered convergent. The iteration procedure is then terminated and the sequence of approximations and differences is printed out in the form of a table. Otherwise, the program is terminated with no output.

The following identifiers in the program represent the corresponding physical quantities:

- **AREA**: Area enclosed by chord and arc (A)
- **RADIUS**: Radius of circle (r)
- **ANGLE**: Approximation to the angle \( x \)
- **CHANGE**: Difference between successive approximations
- **SMALL**: Criterion for convergence
- **G**: For convenience, the quantity \( 2A/r^2 \)
The program is as follows:

BEGIN
COMMENT EXAMPLE 5
SAMPLE PROGRAM USING UNIVAC 1108 ALGOL $
REAL AREA, RADIUS, SMALL, G$
INTEGER I, K$
REAL ARRAY ANGLE(1:10), CHANGE(1:9)$
FORMAT F10(X9,'ITERATION',X5,'ANGLE',X9,'CHANGE',A11),
F11(X13,I1,D15.6,D14.5,A1),
F12(X9,'THE ITERATION PROCEDURE HAS CONVERGED',A1)$
COMMENT SET UP VALUES TO BE USED IN PROBLEM$
AREA = 1.5$
RADIUS = 5.0$
SMALL = 1.0E-5$
G = (2.0*AREA)/(RADIUS**2)$
COMMENT BEGIN ITERATION LOOP -- MAXIMUM OF 9 ITERATIONS$
ANGLE(1) = 1.0$
FOR I = 1 STEP 1 UNTIL 9 DO
BEGIN
COMMENT COMPUTE CHANGE IN APPROXIMATE SOLUTION$
CHANGE(I) = (ANGLE(I)-SIN(ANGLE(I))-G)/(1.0-COS(ANGLE(I)))$
COMMENT TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION$
IF ABS(CHANGE(I)) LSS SMALL THEN GO TO L110$
COMMENT APPROXIMATION HAS NOT CONVERGED -- COMPUTE NEXT APPROXIMATION$
ANGLE(I+1) = ANGLE(I) - CHANGE(I)$
END$
COMMENT END OF LOOP -- ITERATION PROCEDURE HAS NOT CONVERGED$
GO TO FIN$
COMMENT THE ITERATION PROCEDURE HAS CONVERGED$
L110: WRITE (PRINTER,F10) $
WRITE (PRINTER,F11, FOR K=1 STEP 1 UNTIL I DO
(K,ANGLE(K),CHANGE(K)))$
WRITE (F12) $
FIN:
END OF PROGRAM$

Note that a completely blank card gives a blank line in print.

The sample gave the following result:

<table>
<thead>
<tr>
<th>ITERATION</th>
<th>ANGLE</th>
<th>CHANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.000000</td>
<td>.08381</td>
</tr>
<tr>
<td>2</td>
<td>.916186</td>
<td>.00742</td>
</tr>
<tr>
<td>3</td>
<td>.908770</td>
<td>.00006</td>
</tr>
<tr>
<td>4</td>
<td>.908714</td>
<td></td>
</tr>
</tbody>
</table>

THE ITERATION PROCEDURE HAS CONVERGED

This is in excellent agreement with the theory.
APPENDIX E. JENSEN'S DEVICE AND INDIRECT RECURSIVITY

The purpose of this section is to acquaint the reader with two interesting programming techniques, namely Jensen's Device and Indirect Recursivity. A thorough treatment of the recursive concept may be found in "The Use of Recursive Procedures in ALGOL 60", H. Rutishauser The Annual Review in Automatic Programming, Pergamon Press, London, 1963.

Jensen's Device comprises the use of two parameters in a procedure call, in which one is a function of the other. Neither may be a value parameter.

The following example is a method of evaluating an approximation to the definite integral of a function by means of Simpson's Rule over one interval. The algorithm may be written:

```
REAL PROCEDURE SIMPS (X, ARITH, A, B) $  
VALUE A, B $ REAL X, ARITH, A, B $  
BEGIN REAL FA, FM, FB $  
X=A $ FA=ARITH $ X=B $ FB=ARITH $  
X=(B-A)/2 $ FM=ARITH $  
SIMPS=(B-A)*(FA+4*FM+FB)/6  
END SIMPSON INTEGRATION $  
```

In a call of SIMPS, ARITH may be any arithmetic expression. Jensen's Device refers to the case when ARITH is a function of X. For example, the call:

```
I=SIMPS(Z, EXP(Z*Z), 0.0, 1.0)  
```

would cause ARITH to be replaced by EXP(Z*Z) in the running program. This call evaluates an approximation to the integral

\[
\int_{0}^{1} e^{x^2} \, dx
\]

In evaluating an approximation to the double integral

\[
\int_{0}^{1} \int_{0}^{1} e^{xy} \, dy \, dx
\]

indirect recursivity may be used by making the parameter corresponding to ARITH a call to SIMPS itself, thus

```
I=SIMPS(X, SIMPS(Y, EXP(X*Y), 0.0, 1.0), 0.0, 1.0)  
```

APPENDIX G. SYNTAX CHARTS

G1. GENERAL

This appendix summarizes the syntax of the UNIVAC 1108 ALGOL 60 compiler in chart form. Charts for the input/output procedures are also included as well as a brief description of possible format specifications.

The use of the charts is very simple and almost self explanatory. The concept being defined is specified in a rectangle at the top of each chart.

![Type declaration chart]

The definition consists of a series of symbols connected by lines indicating the flow of symbols which define the concept. Two kinds of symbols are used: those with round corners (or circles) and those with square corners. The round-cornered boxes contain symbols that stand for themselves. Square-cornered boxes contain names of concepts which are defined elsewhere in the chart and may be found by a quick reference to the Table of Contents for the appendix.

In some places a special "or" symbol has been used to conserve space. It should be understood as follows:

In some sections a pair of letters may mark two spots in a definition. Underneath that section that letter pair followed by a name appears. This means that name will be used in lieu of the string of symbols between the letter pair in other charts.

The charts use only one of the two possible representations for some symbols in ALGOL. The following equivalences should be noted:

<table>
<thead>
<tr>
<th>Symbol used in this chart</th>
<th>Alternate representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>[ ]</td>
</tr>
<tr>
<td>:</td>
<td>.</td>
</tr>
<tr>
<td>=</td>
<td>:=</td>
</tr>
<tr>
<td>GO TO</td>
<td>GO or GOTO</td>
</tr>
<tr>
<td>$</td>
<td>;</td>
</tr>
</tbody>
</table>

In addition, comments may be inserted in the program by means of the following equivalences:

$ COMMENT <any sequence not containing a >$ equivalent to $

BEGIN COMMENT <any sequence not containing a "$ " BEGIN

END <any sequence not containing END or ELSE or "$ " END

The charts make no mention of the use of spaces within ALGOL. A space has no meaning in the language (outside of strings) except that it must not appear within numbers, identifiers, or basic symbols, and must be used to separate adjacent symbols composed of letters or digits. Spaces may be used freely to facilitate reading.
AA: 1108 program

Explanation: A program is a complete set of declarations and statements which define an algorithm for solving a problem. The logic of this algorithm (its correctness) is the business of the programmer. The compiler only checks that the syntax (form) is correct.

A UNIVAC 1108 program is simply an ordinary program without the outermost BEGIN-END pair.

Notice that the $ is used to separate declarations and statements and is not inherently a part of a declaration or statement. Nevertheless, it will be shown in most examples for clarity.
G3. **Declaration** ::=

Explanation: There are 10 types of declarations each of which is defined in detail on the following pages.
G3.1. **Type Declaration**

Explanation: A type declaration declares the mode of arithmetic that the following identifiers will assume in the block. Types `REAL 2` and `COMPLEX` associate two 1108 words with the identifier, the others one. Upon entrance to a block, identifiers are given the value zero, unless they are also declared `OWN`, in which case they have the same value they had on the last exit from the block.

Examples:

```
INTEGER I4,PAK,LOOPCNT
OWN BOOLEAN ANYLEFT,LASTOUT
COMPLEX C,CINVS
REAL 2 DP
OWN REAL QIN,QOUT,MAXITEM
```
G3.2. **Array Declaration**

Explanation: An array declaration associates an identifier with a one-dimensional or larger matrix of values. The arithmetic expressions define the lower and upper limits of each dimension. The type plays the same role as for simple variables. If omitted, type `REAL` is assumed.

Examples:

```algol
COMPLEX ARRAY CCON4 (0:N),CP1(1:N+1) $
BOOLEAN ARRAY BAND,BOR,BXOR(-4:4) $
REAL ARRAY B(I-1:I+1),XINITIAL,YINITIAL(-N:N,-N:N,1:2) $
OWN INTEGER ARRAY I(1:5),J*K*L(ENTIER(X):P112) $
ARRAY XYZ4(1:N*2) $
```
Explanation: A string declaration associates an identifier with a variable whose value is a string of characters. The number of characters in the string must be less than 4096. A group of characters of a string may be named as a substring.

Examples:

```
STRING ST1(36)*NAME(INITIALS(2),LAST(16)) $
STRING PI(N+2)*QUOTE(1) $
OWN STRING NEXTOUT(80) $
STRING ALPHA(BETA(2),GAMMA(4),2)*DELTA(EPSILON(6)),20) $
```
Explanation: A string array is a matrix whose elements are strings. Appended to the length part of the declaration are the bound pairs for each dimension, just as for an ordinary array.

Examples:

STRING ARRAY SA(80:0:100), CARD(LABEL(8),OP(6),2,OPERAND(64):1:N) $ OWN STRING ARRAY LASTFILE (CLENGTH:1:507) $
G3.5. Switch Declaration ::= 

Explanation: A switch declaration associates an identifier with an ordered list of designational expressions. A switch is used to transfer to a label depending on the value of some variable.

Examples:

```
SWITCH JUMP = L1,START,FEIL4,CALC $
SWITCH BRANCH = IF BETA EQL 0 THEN L1 ELSE JUMP(J),START $
```
G3.6. **External Procedure Declaration**

**Explanation:** This declaration specifies a list of identifiers which are to be the names of procedures not found in the program. These procedures may be written in assembly language (NON-RECURSIVE), FORTRAN or ALGOL. The type of the external procedures is specified if they are functional procedures.

**Examples:**

```
EXTERNAL FORTRAN REAL PROCEDURE CBRT $
EXTERNAL FORTRAN PROCEDURE NTRAN, INVS $
EXTERNAL PROCEDURE ROOTFINDER, KEYIN, KEYOUT $
EXTERNAL NON-RECURSIVE PROCEDURE TYPEIN, TYPEOUT $
```
Explanation: A procedure declaration associates an algorithm with a procedure identifier. The principal constituent of a procedure declaration is a statement which is executed when the procedure is "called" (see 7.4). The procedure heading specifies that certain identifiers appearing within the procedure body are formal parameters. A parameter may also be specified as "VALUE" in which case the procedure statement, when called has access only to the value of the corresponding actual parameter, and not to the actual parameter itself.

Examples:

PROCEDURE ZEROSET (A:N) $ VALUE N $ INTEGER N $ ARRAY A $ BEGIN COMMENT THIS PROCEDURE ZEROES AN ARRAY ASSUMED DECLARED ARRAY A(1:N) $ INTEGER I $ FOR I = 1 STEP 1 UNTIL N DO A(I) = 0 END ZEROSET $ INTEGER PROCEDURE FACTORIAL (NUMBER) $ VALUE NUMBER $ INTEGER NUMBER $ FACTORIAL = IF NUMBER LESS 2 THEN 1 ELSE NUMBER * FACTORIAL (NUMBER-1) $ BOOLEAN PROCEDURE BOOL $ BOOL = NOT (FINISHED AND OFF OR FIRST AND LAST) $
G3.8. Local Declaration

Explanation: The local declaration is a preliminary declaration of identifiers before they are actually declared (or, in the case of a label, used). This is necessary to allow forward references, use of an identifier before it has been defined.

Examples:

LOCAL LABEL L1:ENDFILE $
LOCAL SWITCH SALPHA $
LOCAL BOOLEAN PROCEDURE SLASH:ENDTAPE $
G3.9. **List Declaration**

A list defines an ordered sequence of expressions and array identifiers. A list may only be used as a parameter to a procedure, and, ultimately, only be a procedure written in some language other than ALGOL.

Examples:

```
LIST OUT (A+1,N+1,FOR I = (1,1,NMAX) DO(Q(I),QRES(I)) )
LIST L1(A,B,C), .2(IF MOD(Q,2)EQL 0 THEN B ELSE Q)
```
Explanation: A format is a special string of symbols which are passed on to an input/output routine for editing and control. Integers in front of a format code specify the number of times that code is to be repeated.

Examples:

```algol
FORMAT NEWPAGE(£,'X-COORDINATE',X28,'Y-COORDINATE',A1) $
FORMAT REP(5(4 R16.8,A1),A0.2,S12,'=',D10.1,S12,'=',E10.1,A1) $
FORMAT VECTOR (10T10.4,A1),PATTERN('SWITCHES ARE',8B,A1) $
FORMAT MATRIX (:N:(:M:((D4.2,A1)))) $
```
Explanation: Statements define the sequence of operations to be performed by the program. The eight types of statements are each defined in the following pages.
Explanation: A block automatically introduces a new level of nomenclature by a set of declarations. This means that any identifier declared in the block has the meaning assigned by the declaration, and any entity represented by such an identifier outside the block is completely unaccessible inside the block. The identifiers declared within a block are said to be local (to that block) while all other identifiers are nonlocal or global to that block.

Example:

L:BEGIN INTEGER ARRAY A(1:10) $
A(1) = 1$
FOR J = (2,1,10) DO A(J) = A(J-1) + J
FOR J = (1,1,10) DO WRITE (J,A(J))
END $
G4.2. **Compound Statement** :=

Explanation: A compound statement serves to group a set of statements by enclosing them with a BEGIN-END pair. This group is then treated as a single statement.

Example:

```
BEGIN T= 0 $ FOR I = 1 STEP 1 UNTIL M DO
  T= B(J,I) * C(I,K) +T $
  IF T GTR 820 OR OVFLOW THEN GO TO SPILL $
END$
```
G4.3. **Assignment Statement**

![Diagram of assignment statement]

**Explanation:** An assignment statement assigns the value of the expression on the right-hand side to the variable and procedure identifiers on the left-hand side. A procedure identifier is only permitted on the left-hand side if the statement appears in the body of that functional procedure. If any of the left-part variables are subscripted variables, they are evaluated before the expression is evaluated. Transfers of type are automatically evoked when necessary.

**Examples:**

\[
\begin{align*}
A(1) \ &= \ B(1) \ = \ 835 \\
AANDB \ &= \ A \ \text{AND} \ B \ \text{OR} \ EPS1 \ \text{GEQ} \ EPS2 \\
P \ &= \ SQRT(B**2 - 4*A*C) \\
T \ &= \ S - MYO*EPS0*(2*PI*F)**2 \\
S(V,K-2) \ &= \ COS(ANGLE) + 0.5 *(IF\ S1\ \text{THEN} \ K**3\ \text{ELSE} \ K**5) \\
NAME(1, 6:P + 1) \ &= \ 'IFTHEN' \\
\end{align*}
\]
G4.4. GO TO Statement

Explanation: A **GO TO** statement transfers control to the statement with the label determined by the designational expression.

Examples:

```
GO TO PART4 $
GO TO OPS (I-2) $
GO TO IF ALPHA GTR 0 THEN Q17 ELSE JUMP(-ALPHA) $
GO TO TRACK (IF MOD(P,2) EQ 1 THEN I ELSE A(I)) $
```
Explanation: The IF statement causes the execution of one of a pair of statements depending on the value of a Boolean expression. If this expression is TRUE then the statement after the THEN is executed and the statement after the ELSE is skipped. If FALSE, then the statement after the ELSE is executed, if the ELSE clause is present.

Examples:

IF C1 GTR 10 THEN A(0,0) = KMAX(I) ELSE GO TO LOOP

IF BOOL(J) IMPL BOOL(J+1) THEN STEP(J) = 'VALID' ELSF STEP(J) = 'INVALID'

IF I GEQ 0 THEN BEGIN FOR K = -I STEP 1 UNTIL I DO B(K) = -COS(A-I) END ELSE
BEGIN IF I EQL -1 THEN GO TO ERROR ELSE GO TO NEXT END
G4.6. **FOR Statement** ::= 

Explanation: The **FOR** statement controls the execution of the statement following the **DO** a number of times while the variable to the left of the = is assigned the values determined by the **FOR** list. The (,,) construction is equivalent to the **STEP-UNTIL** construction.

Examples:

```
FOR I = 1 STEP 1 UNTIL N DO FOR J = 1 STEP 1 UNTIL M DO
   A(I,J) = 0 $
FOR S = S + 1 WHILE P(S) NEQ 'A' AND S LEQ 80 DO BEGIN
   N = N*10 + P(S) $ IF OVFLOW THEN GO TO SIZERR END $
FOR S = (1,2*S=S, 2**10),-1,-2,-4 DO IF LOGAND(S,VAR)
   THEN GO TO YES $
```
Explanation: A dummy statement does nothing. It may serve to place a label.

Examples:

```
FOR I = (1,1:N) DO FOR J = (1,1:N) DO BEGIN
    IF I EQL J THEN GO TO ENDDO:
    ...
    ...
    ...
ENDDO:
END $
```

```
S = 0 $
FOR S = S + 1 WHILE P(S) NEQ 'A' DO $
```
Explanation: A procedure statement is a call on a declared procedure. The actual parameters of the call replace the formal or dummy parameters throughout the body of the declared procedure. If the corresponding formal parameter has been "VALUE" specified then only the value of the actual parameter is used by the procedure.

Examples:

MARGIN (62,56,4) $
\text{P(A,B,C,I,J,K)}$

ROOTFINDER (NO,ERGDET,KOEF,-4&0,5,0&-1,1000) $
Explanation: There are three types of expressions, classified according to their values. An arithmetic expression has a numerical or a string value, a Boolean expression is either TRUE or FALSE, and a designational expression has a label as a value.
G5.1. Variable ::= 

variable identifier

array identifier

arithmetic expression

substring part

arithmetic expression

arithmetic expression

subscript list

subscript list

Explanation: A variable is a designation given to a single value. A variable identifier is a variable named in a type declaration.

Examples:

DELTA
BOOLV(7)
CARD
CARD(4)
CARD(I, 6)
A(P(4), N*SIN(ANG), 3)
CUROUT(J, K)
CUROUT(I, J, K)
CUROUT(1, 6: J, K)
G5.2. Function Designator 

Explanation: A function designator defines a single numeric or logical value by applying the rules of the procedure declaration to the actual parameters. Only a procedure which has a type associated with it can be a function designator. Besides those functional procedures declared in the program, several standard ones are available for use without being declared.

Examples:

CLOCK
ARCTAN(1.0)
BACKSLASH(A1*A2)
G5.3. **Arithmetic Expression** ::= 

Explanation: An arithmetic expression is a rule for computing a numerical value.

Examples:

\[
A(4) + 2 * \text{SQRT}(D**3) - \text{DELTA}
\]

\[
\text{IF A LSS 8-5 THEN 0 ELSE A+5}
\]

\[
Q(\text{MOD}(N, 2) + 1) * (\text{IF FIRST THEN 10 ELSE RATIO}) \div 3
\]
G5.4. **Boolean Expression** ::=

Explanation: A Boolean expression is a rule for computing a logical value.

Examples:

FIRST AND NOT SPECIAL
A LSS DELTA OR ITERATIONS GTR MAXN
IF BETA THEN TRUE ELSE IF STEP(I) IMPL STEP(I+1) THEN QVALUE
IF BETA THEN TRUE ELSE IF STEP(I) IMPL STEP(I+1) THEN QVALUE(P,I)
ELSE QVALUE(P,I-1)
G5.5. **Designational Expression**

```
=>
```

**Explanation:** A designational expression is a rule for computing the label of a statement. A switch identifier followed by an arithmetic expression in parenthesis refers to the label in the corresponding position in the switch declaration.

**Examples:**

```
L10
IF BETA THEN CALC ELSE NEXT (K//2)
```
G6. BASIC ELEMENTS

G6.1. Identifier ::= 
  Letter ::= 
  String ::= 
  Digit ::= 

Explanation: An identifier is a name chosen to represent a variable, array, etc. Only the first 12 characters of an identifier uniquely define it.

Examples:

P47
DELTA
SQRTR000F2
E1C4PDG
G6.2. Number ::= 

Explanation: A number is written in its usual decimal notation with the conventions of & for power of ten and corner brackets for complex numbers. Numbers are of four types: REAL, INTEGER, REAL 2 and COMPLEX. REAL 2 is differentiated from REAL by use of && for power of ten, or there may be between 9 and 18 digits in the fixed point part. COMPLEX numbers are distinguished by the corner brackets, where the first number is the real part and the second the imaginary.

Examples:

1
-1009
-.4031
3.1459
-18.0&&
<1,0>
20&&
+1800.&&
&&
< -0.6, &&-2>
G6.3. String ::= 

Logical Value ::= 

Explanation: A string constant is any string of characters which are used as parameters to procedures or with string variables.

Examples:

'DOGGENBURG STR. 22'
'NEQ'
'BJARNE WIST'
'227 KALPHA'
'REAL ARRAY'

Explanation: A logical value is a Boolean constant.
G6.4. Delimiter ::=
**Explanation:** A call on procedure READ reads data from the specified input device into the variables indicated by the list elements. The designational expressions are used as exit points in case end-of-file or end-of-information conditions are met on that device. Note that READ( ) is a legitimate statement but the effect is the same as "No operation".

**Examples:**

```
READ(CARDS,LEOF,LEOI,A,B,C,S,EPSILON) $
READ(FILE(INDEX), FOR I=(1,1,KMAX) DO FOR J=(1,1,LMAX) DO ERG(I,J)) $
READ(DATE) $
```
G7.2. Output Procedure Statement  ::= 

Explanation: A call on procedure WRITE outputs the values defined by the list to the device specified. Modifiers (KEY, EOF, EOI) produce special marks on tape. A format controls editing on paper and punched cards. The designational expression is used as a return point if the output device functions abnormally. Note that WRITE( ) is the same as "No operation".

Examples:

\begin{verbatim}
WRITE (PRINTER, F10, FOR I=(1,1,N) DO A(I,J))$ 
WRITE ('CHECKPOINT CHARLIE', N) $ 
WRITE (FILE('TAPE1'), KEY(I), ABORTLAB, DUMPLIST) $ 
WRITE (FILE('OUTPUT'), EOF('LAST'), EOI) $ 
\end{verbatim}
G7.3. **Position Procedure Statement**

Explanation: The procedure POSITION positions a file forward or backward a number of records or searches for a KEY, EOF, or EOI marker. The designational expressions are used as exits in case the search fails.

Examples:

```
POSITION (FILE('TAPE'), -2) $
POSITION (FILE('INPUT'), KEY('PRICES'), ABORT) $
POSITION (FILE('OUTPUT'), EOI) $
```
Explanation: A call on procedure REWIND rewinds the specified files. The modifier INTERLOCK will cause all previously named files to be rewound with interlock (read/write protect).

Examples:

REWIND (FILE('INPUT'), FILE('OUTPUT')) $
REWIND (FILE('TAPE1'), INTERLOCK, FILE('TAPE2'))$
## G7.5. Summary of Format Codes

A format code of the form Qw.d where Q is a letter and w and d are unsigned integers is interpreted according to the following table. The integer w, except where noted, always specifies the width of the field under consideration. Also, Qw = Qw.0 and Q \equiv Q0.0. The word "print" has been used in the description of output action, but "punch" may be freely substituted.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Activate</td>
<td>Read one card</td>
</tr>
<tr>
<td>B</td>
<td>Boolean</td>
<td>Accept a logical value from the field either TRUE, FALSE, or 1,0.</td>
</tr>
<tr>
<td>D</td>
<td>Decimal</td>
<td>Accept a real value. If the actual number is of INTEGER type then insert a decimal point d places to the left of the right end of the field.</td>
</tr>
<tr>
<td>E</td>
<td>Eject</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Free</td>
<td>Accept an unspecified number of values from the field punched in free-format mode.</td>
</tr>
<tr>
<td>I</td>
<td>Integer</td>
<td>Accept an integer from the field.</td>
</tr>
<tr>
<td>R</td>
<td>Real</td>
<td>Same as letter D.</td>
</tr>
<tr>
<td>S</td>
<td>String</td>
<td>Accept a string from the field.</td>
</tr>
<tr>
<td>T</td>
<td>Significance</td>
<td>Same as letter D.</td>
</tr>
<tr>
<td>X</td>
<td>Skip</td>
<td>Skip the field.</td>
</tr>
</tbody>
</table>
G7.6. Grouping of Format Codes

Format codes may be repeated in execution by four methods:

(a) Placing an unsigned integer in front of a format code:

\[ 7D9.2 \]

This has the same effect as if the phrase D9.2 was written 7 times.

(b) Enclosing a group of format codes in parentheses and placing an unsigned integer before the parenthetical expression:

\[ 7(6R18.8\cdot A1) \]

This has the effect of expanding the phrase inside the parenthesis 7 times.

(c) Similar to (b) above but using an integer or Boolean expression enclosed in colons before the parenthetical expression. The value of the expression determines the number of times the enclosed code or group of codes is to be repeated:

\[ :NMAX/\sqrt{3+1}: (6R18.8\cdot A1) \]

(d) Enclosing a group of format codes in parentheses but not preceding this parenthetical expression with an integer constant. This means the enclosed codes are to be used until there is no more output (or input) to process. The parentheses surrounding the entire format string are interpreted in this manner.

\[ \text{FORMAT FOUT}(4(510\cdot X5)\cdot A2) \]