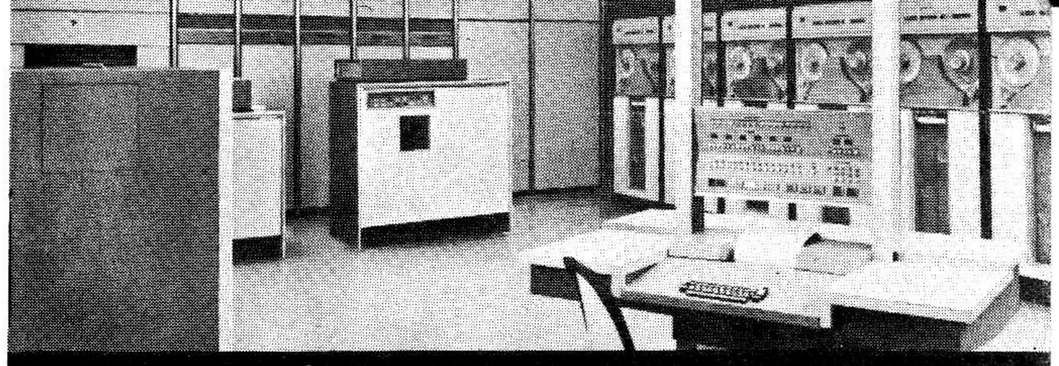


UNIVAC® 1107

TECHNICAL BULLETIN

Programmer's Guide

U-3540 C

This manual is published by the Univac Division of Sperry Rand Corporation in loose leaf format as 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 hardware and/or 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 respective Systems.

CONTENTS

CONTENTS (cont.)

CONTENTS (cont.)

UPDATING PACKAGE A

The attached sheets contain editing corrections, major revisions, and additions to the UNIVAC 1107 FORTRAN Programmer's Guide U-3540.

The major revisions are:

1. Limitations on Integer values; Section II, p. 8.
2. Subscripted Variables; Section II, p. 14.
3. DO statement; Section III, p. 2.
4. FORMAT Editing codes; Section IV, p. 5.
5. DATA Statement; Section V, p. 6-8.
6. RETURN Statement; Section VII, p. 11.

The major additions are a table of Built-In Functions and a table of Library Functions that will be provided with the UNIVAC 1107 FORTRAN package. (Section VII, p. 5 & 7.)

The following replacements should be made in the UNIVAC 1107 FORTRAN Programmer's Guide:

1. Replace Index, pages 1, 2, and 3 with new pages 1, 2, 3.

2. Replace Section II, pages 3, 4, 5, 6, 7, 8, 13, and 14 with new pages 3, 4, 5, 6, 7, 8, 13 and 14.

3. Replace Section III, pages 1, 2, 3, 4, 33, 34, 35, and 36 with new pages 1, 2, 3, 4, 33, 34, 35, and 36.

4. Replace Section IV, pages 3, 4, 5, 6, 7, 8, 9, 10, 11, and 12 with new pages 3, 4, 5, 6, 7, 8, 9, 10, 11, and 12.

5. Replace Section V, pages 5, 6, 7, and 8 with new pages 5, 6, 7, and 8.

6. Replace Section VII, pages 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15 with new pages 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15.

UNIVAC 1107
FORTRAN Programmer's Guide, U 3540

March 23, 1964

UPDATING PACKAGE "B"

This bulletin releases UNIVAC 1107 FORTRAN Programmer's Guide, U 3540, Updating Package "B", 32 pages.

This Updating Package should be utilized in the following manner:

	Destroy Former Pages Numbered	Replace With New Pages Numbered
SECTION II	5 and 6(Rev.1)	5(Rev.1) and 6(Rev.2)
	9 thru 12	9(Rev.1) thru 12(Rev.1)
	31 and 32	31(Rev.1) and 32(Rev.1)
SECTION III	1 thru 4	1(Rev.1) thru 4(Rev.1)
	13 and 14	13(Rev.1) and 14(Rev.1)
SECTION IV	1 thru 6(Rev.1)	1(Rev.1) thru 6(Rev.2)
	11(Rev.1) thru 16	11(Rev.2) thru 16(Rev.1)
SECTION V	7(Rev.1) and 8(Rev.1)	7(Rev.2) and 8(Rev.2)
SECTION VII	3(Rev.1) and 4(Rev.1)	3(Rev.2) and 4(Rev.2)
APPENDIX 3	1 and 2	1(Rev.1) and 2(Rev.1)

1. INTRODUCTION

The FORTRAN language provides a mathematical notation suitable for use in many industrial and scientific applications.

The FORTRAN language for UNIVAC 1107 incorporates all of the FORTRAN IV language specifications as announced by UNIVAC on December 11, 1961. This language is enriched with many new features that have resulted from extensive user experience. FORTRAN programs that have been written for UNIVAC computers, or those of other manufacturers, may be processed on UNIVAC 1107 with minimal change. These changes reflect the insignificant extent to which FORTRAN language might be considered machine dependent. Most existing FORTRAN programs will require absolutely no modification in order to be processed by the UNIVAC 1107 FORTRAN processor.

This manual provides both a basic introduction to the FORTRAN language and the specifications of the FORTRAN language for the UNIVAC 1107.

A programmer new to the FORTRAN language should pay particular attention to the basic section of the manual. In addition, as an aid to learning basic FORTRAN, a sample problem is presented. This problem is built up gradually, statement by statement, until the complete problem is presented. It can be found in Section II, BASIC FORTRAN.

Anyone already familiar with the language can skim through the sections of general interest, but should be attentive to those passages where the specifications particular to UNIVAC 1107 FORTRAN are presented. (Sections III through VII).

Of particular interest for the experienced FORTRAN user should be the PROGRAMMER'S REFERENCE MANUAL for UNIVAC 1107 FORTRAN, U-3569, the reference manual upon which this primer is based.

The presentation of Section III, Control Statements, and Section VII, Functions and Subprograms, is an adaptation of sections found in FORTRAN General Information Manual F28-8074, copyright 1961 by International Business Machines Corporation. Throughout this manual there may be similarities to other publications by IBM dealing with the FORTRAN language. UNIVAC gratefully acknowledges the permission of IBM to adapt their material and appreciates their expressed understanding of any other similarities.

2. A BASIC INTRODUCTION TO THE FORTRAN LANGUAGE

A. GENERAL

1. Computers and Languages

In order to solve a problem a computer must be given a series of instructions which determine how the computer is to operate. In addition, the computer must be given one or more sets of data upon which to operate. This combination of instructions and data is called a program. A program must define in complete detail exactly what the computer is to do, under every conceivable combination of circumstances, with the data which is read into or processed by the computer. The number of instructions required for the complete solution of a problem may be a few hundred or many thousands, depending upon the problem. The computer may refer to these instructions one after another. It can also be instructed to repeat, modify, or skip over certain instructions, depending on intermediate results or circumstances. The ability to repeat operations, usually called looping, combined with the other facilities of modifying and skipping over instructions, permits a significant reduction in the number of instructions required to perform any given job. For example, two sets of numbers exist and it is desired to add the corresponding numbers of each set together. Instructions may be written to add the first number of the first set to the first number of the second set, and then to repeat this operation with the second, third, fourth, etc., numbers of each set. In this way a few instructions may cause thousands of additions.

Since the computer does not respond to the English language, the program must be encoded in a form known as machine language. Considerable time and effort have been spent in developing programming systems that allow the programmer to write in a symbolic language more easily comprehensible to him than machine language. Associated with each programming system is a machine language program called a processor. The processor accepts a program written in the symbolic language (source program) and converts it into a machine language program (object program) that can be utilized by the computer. Early programming languages were machine-oriented; that is, they were phrased in terms of the operations which could be performed by the computer. A significant improvement in programming languages occurred with the development of a problem-oriented language known as FORTRAN (FORmula TRANslator).

2. FORTRAN

FORTRAN is a language closely resembling the language of mathematics. The FORTRAN language is problem-oriented. This means that the programmer may think in terms of the problem to be solved and the method of solution, rather than thinking in terms of the computer which is used to solve the problem. While initially designed for scientific applications, it has proved quite convenient for many commercial and industrial applications. The main advantages of the FORTRAN language are its conciseness and form. Frequently, a scientific problem can be phrased in terms of a set of scientific formulae. When writing in a machine-oriented language, the programmer must write an instruction corresponding to each operation indicated in the formula. For example, in order to evaluate

$$F = (A + B - C) D/E$$

the programmer might write

Load	A
Add	B
Subtract	C
Multiply by	D
Divide by	E
Store the result in	F

In contrast, when using the FORTRAN language, the formula can be evaluated by a single FORTRAN statement:

```
F = (A + B - C)*D/E
```

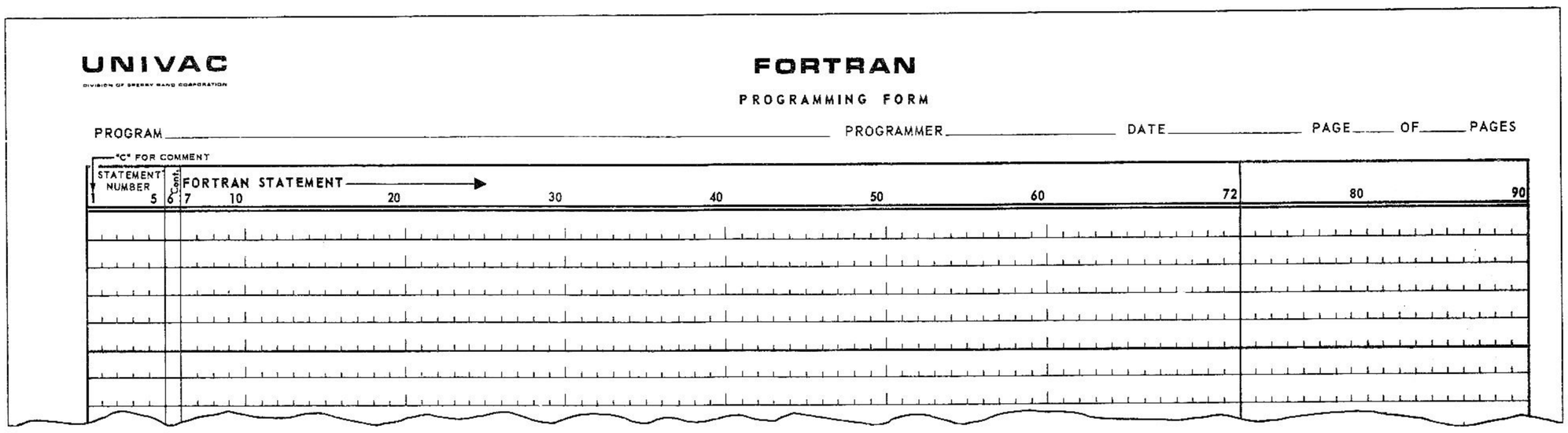
Clearly, one FORTRAN statement is often the equivalent of many machine language instructions.

It is the function of the processor to examine the FORTRAN language program and to form an efficient machine language program. The processor must also take care of supporting details such as reserving space in storage for data and instructions.

a. Writing FORTRAN Programs

FORTRAN has no provision for "lower case" letters. Therefore, only "CAPITAL" letters should be used in writing FORTRAN

statements. FORTRAN programs should be written on the standard FORTRAN programming form. A copy of this form is shown below.


UNIVAC
DIVISION OF SPERRY RAND CORPORATION

FORTRAN
PROGRAMMING FORM

PROGRAM _____ PROGRAMMER _____ DATE _____ PAGE ____ OF ____ PAGES

"C" FOR COMMENT

STATEMENT NUMBER 1 5 | Cont. 6 | FORTRAN STATEMENT 7 10 20 30 40 50 60 72 | 80 90

Each FORTRAN statement is written on a line in columns 7-72. Column 6 should contain a zero or a blank. If a statement is too long for one line, it may be continued on one or more successive lines (maximum of 19 continuation lines) by placing a non-blank character in column 6. (Frequently the character in column 6 is used to indicate the order of continuation lines. That is, the first continuation line would have a 1 in column six, the second a 2 in column 6 and so on. Sequential numbering is good practice and should be used whenever possible.)

Statement numbers may be written in columns 1-5. These numbers permit the programmer to refer to statements within the program. For example, the statement GO TO 10 would result in a transfer of control to the FORTRAN statement numbered 10. It is not necessary to number every statement. The statement numbers need not be in numerical order. The sequence in which statements are executed is dependent upon the order of the statements, not on the value of the statement numbers. Any number from 1 to 32,767 may be used in a statement number.

It is often desirable to include explanatory comments within the FORTRAN program. In order to do this, a C must be placed in column 1 of the comment line. Comments may then be written in columns 2-72. The comment line is ignored by the FORTRAN processor, except that these comments will be included in the print-out of the source program.

Columns starting with column 73 may be used in any manner. Information contained in those columns is not utilized by the FORTRAN processor, but will be included in the listing of the source program.

With the exception of column 6 and the alphanumeric fields of FORMAT and DATA statements (see Sections IV and V of this manual), blanks are ignored by the FORTRAN processor and may be used freely by the programmer to improve readability. For example: The FORTRAN statements

```
F=(A+B-C)*D/E
```

and

```
F = ( A + B - C ) * D / E
```

are equivalent.

An illustration of part of a FORTRAN program is shown below.

```
ANGLE ( 1 ) = 1.0
DO  100 I = 1, 9
CHANGE ( I ) = (  ANGLE ( I ) - SIN ( ANGLE ( I ) - G ) / ( 1.0 - COS (ANGLE ( I ) )
```

In order to write FORTRAN programs, it is necessary to learn the rules for writing the following:

1. Constants, such as 2.71821828 or 72

2. Variables, such as X or Y

3. Subscripted variables, such as x_i or y_j written in the FORTRAN language as X(I) or Y(J)

4. Control statements which alter the order in which the FORTRAN statements are to be executed.

5. Arithmetic statements, used to perform computation.

6. Input/output statements, used in transmitting data into and out of the computer.

7. Specification statements, used to assist the FORTRAN processor in producing the object program.

8. Subroutine statements that permit complex procedure programs to be incorporated within larger programs.

Each of these topics will be discussed subsequently in this manual.

b. STATEMENTS

A FORTRAN source program is made up of any number of statements. Each statement is concerned with one part of the problem; it may cause data to come into the computer, calculations to occur, decisions to be made, results to appear in printed form on the printer, etc.

Some examples of FORTRAN statements and what they do are:

A = 35.0 + 62.0	This statement causes the computer to compute the sum of 35.0 and 62.0 . The variable A is then given the value 97.0 (35.0 + 62.0) .
Q = A/4.0	The slash (/) indicates division. This causes the computer to divide the value of A by the number 4.0 . Using the value of A obtained in the previous example (97.0), Q would be given the value 24.25 .

Some statements are written which do not cause any coding and resultant computer action but which contain information for the use of the FORTRAN processor.

DIMENSION BLOCK (20)	This statement does not cause specific computer action. Instead, it instructs the FORTRAN processor to provide storage for 20 real (floating point) quantities which will be stored in the array "BLOCK" .

Different kinds of statements constitute the FORTRAN language. These statements are control statements, arithmetic statements, input/output statements, function and subprogram statements, and specification statements.

Arithmetic statements are used to evaluate arithmetic expressions or formulae. The statements below are examples of arithmetic statements:

```
N = 1
F = (A + B - C) * D / E
AREA = SQRT ( (S-A) * (S-C) * S)
```

Control statements are used to alter the order in which FORTRAN statements are executed. Ordinarily FORTRAN statements are executed sequentially. By using a control statement, the programmer can cause the computer to repeat or skip a sequence of statements. The statements below are examples of control statements:

```
DO  100 I = 1, 10
IF (ALPHA) 205, 206, 205
GO TO 461
```

Input/output statements are used to control the flow of data into and out of the computer. The statements enable the programmer to determine the amount of data to be transmitted, the external medium to be used (punched cards, printed page, magnetic tape) and the format in which the data is to be arranged (that is, the number of spaces, lines to be skipped, etc.)

The statements below are examples of input/output statements:

```
      READ (1, 10) A, B, C, D, J
      WRITE (5) (A(I), I = 1, 10)
      REWIND 3
10    FORMAT  (5X, 4F13.3, I9, 5X, 7HNO DATA)
```

Function and subprogram statements allow the programmer to make use of previously written programs. These statements permit the programmer to call upon such special purpose programs during the execution of his main program. In this way, the programmer can cause a complex procedure to occur without specifying each statement every time the procedure is to occur.

Many mathematical routines are maintained in the form of a library of subroutines. These subroutines are available to the programmer whenever he may desire to utilize them in his FORTRAN program. The subroutine library lends itself to ease of programming and reduction of redundant programming effort.

The function and subprogram statements allow the programmer to divide large programs into smaller subprograms or subroutines. This technique is a great aid to the programmer in the all important process of removing programming errors (debugging). Program corrections may be made and programs expanded or reduced in scope by processing the appropriate subprogram rather than the entire program. Frequently this results in a considerable saving of computer time.

The statements below are examples of subprogram statements:

```
FUNCTION SIGMA (N, X)
SUBROUTINE OUTPUT (ALPHA, B, N, X)
SUBROUTINE SAVE
```

Specification statements are used to indicate to the FORTRAN processor the manner in which the data is to be stored in the computer, and the quantity of data to be stored. Some data may be in the form of alphabetic information, other data may be in the form of integers. Additional types may be in the form of real (floating-point) numbers, double-precision (double-precision floating-point) numbers, and complex numbers. The statements below are examples of specification statements:

```
COMMON A, B, C, D, E
DIMENSION A (15), B (10, 10), E (12)
INTEGER E, G, ALPHA
EQUIVALENCE (E(1), GAMMA), (A, XMN)
```

c. Integer, Real (floating-point), Double-precision (double-precision floating-point), Complex, and Parameter numbers.

Before writing a program, it is necessary to understand the four kinds of numbers that are written in the FORTRAN language:
These are:

integer, real, double-precision, and complex numbers.

An integer number is an ordinary whole number from one to eleven decimal digits in length (less than 2^{35}). If it is positive, it may be prefixed with a plus sign; if it is negative, it must be prefixed with a minus sign. Integer numbers occupy one word of UNIVAC 1107 storage.

If an integer appears in a statement with quantities of a different type (i.e., real, double-precision, or complex), then the integer must be less than 2^{27}.

The following are acceptable integer numbers:

0
15
+ 753
-999999
2501
96742578348

A real (single-precision floating-point) number may be expressed with from one to nine significant decimal digits. One and only one decimal point must appear either at the beginning, at the end, or between two digits. If the number is positive, it may be prefixed with a plus sign. If it is negative, it must be prefixed with a minus sign. A decimal exponent may be applied to the number by following the number with an E followed by a signed (+ optional) one or two digit exponent. The magnitude of a real number must be between the approximate limits 10^{-38} and 10^{38} or be zero. Real numbers occupy one word of UNIVAC 1107 storage.

The following are acceptable real numbers:

0.0
19.5
+7.01
-15.07
0.00095
4.0E2 (This means $4.0 * 10^{2}$)
4.0E+2 (This means $4.0 * 10^{2}$)
4.0E-2 (This means $4.0 * 10^{-2}$)

Quite clearly in writing real numbers in FORTRAN notation, the letter "E" is used to replace "times 10 to the power".

A double precision number is represented as a real number followed by a D and a signed (+ optional) one or two digit exponent. The magnitude of the constant must lie between the approximate limits of 10^{-38} and 10^{+38} or be zero. Double precision numbers occupy two consecutive words of UNIVAC 1107 storage.

The following are acceptable double-precision numbers:

0.0 D 0
16.9 D+1
+8.897 D-10
-1750.0 D+19

2.0D3 (This means $2.0 * 10^{3}$)
2.0D+3 (This means $2.0 * 10^{3}$)
2.0D-3 (This means $2.0 * 10^{-3}$)

A complex number consists of two single-precision floating-point numbers separated by a comma and enclosed in parentheses. The floating-point numbers to the left and right of the comma represent, respectively, the real and imaginary parts of the complex number. Complex numbers occupy two words of the UNIVAC 1107 storage.

The following are acceptable complex numbers:

(2.0 , 2.0)
(3426.78 , 293.6)
(4.12 E2, 6.5 E3)
(4.12 E-2 , 6.5 E3)
(4.12 E-2 , 6.5 E-3)

Certain integer numbers may appear numerous times throughout a program. They are represented by mnemonic references. Different integer numbers may be assigned to the same mnemonic references on different compilations. For these reasons, it may be convenient to refer to these integer numbers by variable names, rather than explicitly as numbers. This may be done by defining the variable names with the appropriate integer values in a PARAMETER Statement.

A parameter is represented by a variable name (see below). In any one compilation, it represents a single integer number defined in a PARAMETER statement.

The general form of the PARAMETER Statement is:

$$\text{PARAMETER } i_1 = n_1,\ i_2 = n_2, \ldots i_k = n_k$$

where the i's are variable names and the n's are integer numbers. For every occurrence of the name of a parameter variable, that name will be replaced by the integer number assigned to it. The PARAMETER Statement defining a name must appear before any other reference to the name.

Examples of PARAMETER Names:

```
A
I
ZEST
A1B2C3
```

Examples of PARAMETER Statements:

```
PARAMETER  A = 1, I = 52, ZEST = 654
PARAMETER  A1B2C4 = 523481
```

d. Constants, Variables, and Arrays

1) General

Any quantity which appears in a FORTRAN statement in the form of a number is called a Constant. Any quantity which appears in an arithmetic statement in the form of a name is called a Variable. For example, in the FORTRAN statement

```
I = 2
```

the variable 'I' and the constant '2' appear. 'I' may take on different values within the FORTRAN program. The result of this FORTRAN statement is the replacement of the former value of 'I' with the value '2'. Other examples of constants are 107.3, 2,3 D+1, 6, -5.631, 544. Other examples of variables are ABC, ALPHA, NET, TOTAL.

FORTRAN distinguishes between the different kinds of variables in the following manner. Any variable whose name appears in a REAL statement is in the real mode. Any variable whose name appears in an INTEGER statement is in the integer mode. Any variable whose name appears in a DOUBLE PRECISION statement is in the double-precision mode. Any variable whose name appears in a COMPLEX statement is in the complex mode. Any variable whose name appears in a LOGICAL statement is in the logical mode. If the name of a variable does not appear in a Type statement then its mode is determined according to the following convention:

> If the name of the variable begins with I, J, K, L, M or N, then the variable is in the integer mode. If the name begins with any letter other than I, J, K, L, M or N, then the variable is in the real mode. (See the discussion of Type statements in Section VI).

The values of integer, real, and logical variables occupy one word of UNIVAC 1107 storage. The values of double-precision and complex variables occupy two consecutive words of the UNIVAC 1107 storage.

The name of the variable may contain as many as six letters or numbers. However, the first character must be a letter. No special characters (i.e., + * - / , . ( ) $ = ') are permitted within the name of the variable.)

The following are acceptable integer variable names:*

N
IJK
I
NUMBER

The following are acceptable real variable names:*

A
GAMMA2
T123AB
GOOD
BETTER
UNIVAC

2) Choosing Names for Variables

The conventions for naming variables permit a wide choice of names. Whenever possible, it is desirable to select names suggesting the meaning or role that the variables play in the problem to be solved. For example, to compute linear velocity, it would be possible to use the FORTRAN statement:

SPEED = DIST/TIME

This type of variable naming helps to make the FORTRAN program more self-explanatory and easy to follow.

* Note that there are not any unique names for Logical, Complex, and Double-Precision variables as there are for Integer and Real variable. These three types must be explicitly defined in Type Statements.

3) Subscripted Variables (Arrays)

It is often desirable to perform computations on groups of numbers. These groups are called arrays. All of the items within a group are identified by the same name. They are distinguished from one another by subscription. For example, the set of numbers 1.0, 5.9, 3.6, 2.17, 15.0 might be such a group of numbers. The entire group might be arbitrarily given the name 'VALUE', and they could be ordered in such a way that

```
VALUE (1) = 1.0
VALUE (2)  = 5.9
VALUE (3)  = 3.6
VALUE (4)  = 2.17
VALUE (5)  = 15.0
```

These five FORTRAN statements establish the values of the items in the array 'VALUE'. This subscripted form may also be used in order to perform computations using a loop. For example, if it is desired to double each number in the array called 'VALUE' and place the result in a corresponding array called 'DOUBLE', the program might take the form:

```
      DO 115 I = 1, 5

115 DOUBLE (I) = 2.0*VALUE (I)
```

As a result of this program, the numbers 2.0, 11.8, 7.2, 4.34, 30.0 would be placed in DOUBLE (1) through DOUBLE (5) respectively.

A variable may appear with one to seven subscripts. A subscript is an integer quantity of rigid form, which is:

$$\pm M_1 \pm M_2 \pm M_3 \ldots \pm M_k$$

where M_i must be one of the following:

1. an integer constant
2. an integer variable
3. $n*K_1*K_2*\ldots K_m$, where n is an integer constant and K_i are integer variables. Only one K_i in such a product may be a DO index.

If a variable has more than one subscript then the total number of variables appearing in the subscripts must be less than 30. The subscripts are separated by commas, enclosed in parentheses, and are appended to the right of the variable name to which they apply; e.g., A(3*I-J+1,K).

Each subscripted variable must have its dimensions declared prior to its occurrence in an executable statement function.

Note: On the UNIVAC 1107, all subscripts are treated as modulo 2^{16}.

The following are acceptable subscripts:

(3)
(I)
(I+3)
(JOB-5)
(7*IJK) NOTE: IJK is the name of a SINGLE integer variable.
(3*I+4)
(6*M-1)
(I,J*K+3,L-M,6*N)

The following are not acceptable subscripts:

(K(3)) A subscripted variable may not be used as a subscript.

For example, if J had the value 7, then the subscript (3*J+5) would have the value 26. A reference to BLOCK (3*J+5) would be interpreted as a reference to BLOCK (26), which is the 26th element of the array BLOCK.

The subscripted variables (arrays) which have been discussed previously are called single-subscripted variables or one-dimensional arrays. The UNIVAC 1107 FORTRAN language permits the use of up to a seven-dimensional array. These arrays can be visualized in the following manner:

A one-dimensional array may be thought of as a column of numbers. The sixth item in the column might be called ITEM (6).

A two-dimensional array may be thought of as a table of numbers containing several columns. The position of an item in the table could then be specified by indicating the position of the item within the column and the particular column which contains the item. The notation TABLE (6, 3) would then indicate the 6th item of the 3rd column of the table.

A three-dimensional array may be considered a book of tables. The position of an item within the book may be specified by indicating the position of the item within the column, the column containing the item, and the page containing that particular column. The notation BOOK (6, 3, 51) might be used to denote the 6th item of the 3rd column of the 51st page of the book.

The general form of a subscripted variable is:

ARRAY(N1) or ARRAY (N1, N2) or ARRAY (N1, N2, N3)... or

ARRAY (N1, N2, N3, N4, N5, N6, N7)

ARRAY is the name of a subscripted variable. N1, N2, ... N7, are subscripts and each may take any of the forms indicated previously. Every pair of subscripts must be separated by a comma.

The following are acceptable subscripted variables:

A(J)
BB(38, 22, 36)
SET(2*I+4, L-2)
BUNCH(I, J, K)
GROUP(3*J+K+L, IN, N, N-6)

The following are not acceptable subscripted variables:

BAD(I,)	Commas are not permitted to precede the first subscript or follow the last subscript.
WORSE, J, K	The subscripts must be enclosed in parentheses.

B. AN ILLUSTRATIVE FORTRAN PROGRAM

There are many scientific problems for which no exact analytic solution can be found. However, it is often possible to approximate the solution to such problems as accurately as desired. A computer is ideally suited for solving such problems. The technique usually used is known as the Method of Successive Approximations. An initial approximation of a solution is made. This first approximation is inserted in a procedure, and the procedure yields an improved approximation to the solution. The procedure is repeated until the desired degree of accuracy is achieved. Since the programmer has the ability to repeat certain sequences of FORTRAN statements, he is ideally equipped to cope with problems where this method is applicable.

One very simple problem which can be solved in this manner is the problem of finding the square root of a number.

It is known that the square root of a positive number, A, can be found by using the procedure:

$$X_{n+1} = 1/2\ (X_n + A/X_n)$$

The first approximation, X_1, is used to compute the value of X_2; X_2 is used to compute the value of X_3, and so forth. Each approximation is closer to the exact square root than its predecessor. By attempting to solve a problem for which the solution is known, the operation of this technique will become apparent. Suppose it is desired to find the square root of 25. If 10 is used as the initial approximation then

$$X_1 = 10$$

X_2 is found by making use of the square root procedure.

$$X_2 = 1/2\ (10+25/10) = 6.25$$

Repeating the procedure yields

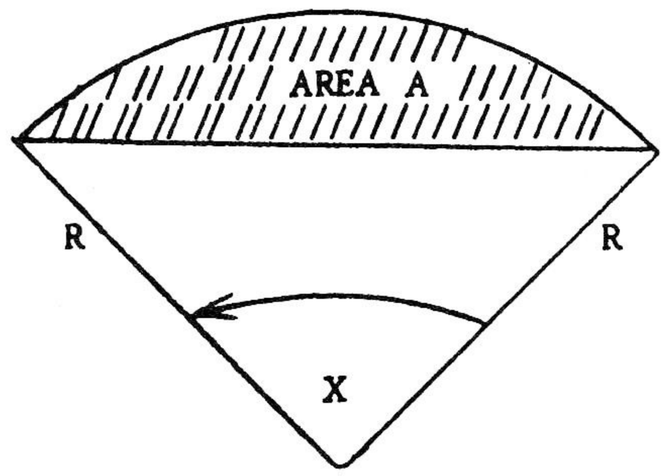
$$X_3 = 1/2\ (6.25+ 25/6.25) = 5.125$$

$$X_4 = 1/2\ (5.125+25/5.125) = 5.00152$$

By continuing in this fashion, the square root of 25 may be approximated to as great a degree an accuracy as desired. Although the exact solution 5 will not be determined, close approximations are sufficient for practical problems.

Statement of the Problem

The remainder of this section presents the solution, using FORTRAN language, of a problem in geometry. The problem is to determine the angle, X, for which the arc and chord of a given circle will enclose a prescribed area, A. (See figure below.)



Although the analytic solution to this problem cannot be found, it is possible to obtain an approximate solution through the use of an iterative technique.

The relationship between the area, A, and the angle, X, is given by the formula

$$A = (1/2)R^2X - (1/2)R^2 \sin X = (1/2)R^2(X - \sin X)$$

The general iteration formula to be used is

$$X_{n+1} = X_n - F(X_n)/F'(X_n)$$

for this problem

$$F(X_n) = (1/2)R^2(X_n - \sin X_n) - A$$

and

$$F'(X_n) = (1/2)R^2 (1 - \cos X_n)$$

Therefore, the iteration formula for this problem is

$$X_{n+1} = X_n - (X_n - \sin X_n - 2A/R^2)/(1 - \cos X_n)$$

The difference between successive approximations is given by the formula

$$\text{Difference}_n = (X_n - \sin X_n - 2A/R^2)/(1 - \cos X_n)$$

When the difference between successive approximations becomes less than a predetermined value, then 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, then the iteration procedure is terminated and the approximate solution is rejected.

The conditions which will be used in this example are:

Area = 1.5
Radius = 5.0

The first approximation, X_1, will have the value 1.0. The iteration procedure will be repeated for a maximum of nine iterations. If the successive approximations differ by less than 0.0001, then the sequence of approximations will be considered convergent and the iteration procedure will be terminated and the sequence of approximations and differences will be printed out in the form of a table. Otherwise, a stop will occur with no printed output.

The following names will be used in the FORTRAN program used to solve this problem:

The area A has the name "AREA".

The radius R has the name "RADIUS".

The approximation of the angle X has the name "ANGLE".

The difference between successive approximations has the name "CHANGE".

The parameter to be used in the test for convergence has the name "SMALL".

For convenience, the quantity 2A/R**2 is called "G".

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                                    1
      DIMENSION ANGLF(10), CHANGE(9)                                        2
   10 FORMAT ( 9X  9HITERATION  5X  5HANGLE  9X  6HCHANGE  )                3
   11 FORMAT ( 13X  I1, F15.5, F14.5  )                                     4
   12 FORMAT ( 9X 38HTHE ITERATION PROCEDURE HAS CONVERGED. )               5
C --- SET UP VALUES TO BE USED IN PROBLEM                                   6
      AREA = 1.5                                                            7
      RADIUS = 5.0                                                          8
      SMALL = 1.0E-4                                                        9
      G = ( 2.0*AREA )/( RADIUS**2 )                                        10
C --- BEGIN ITERATION LOOP - MAXIMUM OF 9 ITERATIONS                        11
      ANGLE(1) = 1.0                                                        12
      DO 100  I = 1,9                                                       13
C --- COMPUTE CHANGE IN APPROXIMATE SOLUTION                                14
      CHANGE(I) = ( ANGLE(I) - SIN(ANGLE(I)) - G )/( 1.0 - COS(ANGLE(I))    15
C --- TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION                          16
      IF ( ABS(CHANGE(I)) .LT. SMALL )  GO TO 110                           17
C --- APPROXIMATION HAS NOT CONVERGED - COMPUTE NEXT APPROXIMATION          18
  100 ANGLE(I+1) = ANGLE(I) - CHANGE(I)                                     19
C --- END OF LOOP - ITERATION PROCEDURE HAS NOT CONVERGED                   20
      STOP                                                                  21
C --- THE ITERATION PROCEDURE HAS CONVERGED                                 22
  110 WRITE ( 2,10 )                                                        23
      WRITE ( 2,11 ) ( K, ANGLE(K), CHANGE(K), K = 1,I )                    24
      WRITE ( 2,12 )                                                        25
      STOP                                                                  26
      END                                                                   27
```

On the opposite page, the complete FORTRAN solution of this problem is presented. On subsequent pages, the program will be reconstructed step by step in a logical fashion. The entire program is first presented here for completeness.

The program is described as one written using Basic FORTRAN. Basic FORTRAN is not any defined set of statements, merely those that most commonly appear in most FORTRAN programs.

These statements, with a line number* referencing the program are the following:

Comments Statements (line numbers 1, 6, 11, 14, 16, 18, 20, 22)

Comments improve the readability of the FORTRAN program. By using many comments statements, as well as mnemonic names for variables, a FORTRAN program can be made highly self-documenting.

Specification Statements (lines 2, 3, 4, 5, 27)

These statements assist the FORTRAN processor in constructing the object program.

The Format statements, when they appear, are always used in conjunction with input/output statements.

Input/Output Statements (lines 23, 24, 25)

These statements transmit information into and out of the computer. In this problem, there are no input statements. All the values required to initiate computation are contained within the program.

Arithmetic Statements (lines 7, 8, 9, 10, 12, 15, 19)

These statements can cause computations to be performed and assign new values to the variables whose names appear on the left-hand side of the equal symbol (=).

---

* The line numbers are not part of the FORTRAN program. They are introduced here to aid the discussion. It is recommended that columns above 72 contain line numbers as an aid to documenting FORTRAN programs.

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                    1
      ................                                       2
      ................                                       3
      ................                                       4
      ................                                       5
C --- SET UP VALUES TO BE USED IN PROBLEM                   6
      AREA = 1.5                                             7
      RADIUS = 5.0                                           8
      SMALL = 1.0E-4                                         9
      ................                                      10
      ................                                      11
      ANGLE(1) = 1.0                                        12
      ................                                      13
      ................                                      14
      ................                                      15
      ................                                      16
      ................                                      17
      ................                                      18
      ................                                      19
      ................                                      20
      ................                                      21
      ................                                      22
      ................                                      23
      ................                                      24
      ................                                      25
      ................                                      26
      ................                                      27
```

Control Statements (lines 13, 17)

The DO statement controls the execution of the statements on lines numbered 15 through 19. The IF statement on line number 17 will control whether statement number 110 is performed next, or whether statement number 100 is performed next.

EXPLANATION OF CODING ON OPPOSITE PAGE

The first job to be done is to write a comments statement describing the problem. With the introduction of each successive part in the construction of the program, comments statements will be written to accompany the related section of the program.

Next, the initial values of variables are established. The area, radius of the circle, and the convergence criterion are set equal to their given values. An initial guess or first approximation to the unknown angle is also established. ANGLE (1) is the name of the subscripted variable whose value is this first guess.

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                          1
      ................                                              2
      ................                                              3
      ................                                              4
      ................                                              5
C --- SET UP VALUES TO BE USED IN PROBLEM                         6
      AREA = 1.5                                                    7
      RADIUS = 5.0                                                  8
      SMALL = 1.0E-4                                                9
      ................                                              10
C --- BEGIN ITERATION LOOP - MAXIMUM OF 9 ITERATIONS               11
      ANGLE(1) = 1.0                                                12
      DO 100  I = 1,9                                               13
C --- COMPUTE CHANGE IN APPROXIMATE SOLUTION                       14
      ................                                              15
C --- TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION                 16
      ................                                              17
      ................                                              18
100   ................                                              19
      ................                                              20
      ................                                              21
      ................                                              22
      ................                                              23
      ................                                              24
      ................                                              25
      ................                                              26
      ................                                              27
```

The control statement is set up that will cause up to nine successive approximations to the angle. This DO statement (line 13) will cause the variable I to assume the nine values 1, 2, 3, ... 9. The DO statement controls the execution of a loop that includes statements from line 15 through line 19. The comments statements are prepared for two necessary parts of the procedure, i.e., to compute the change in the approximate solution, and to test whether convergence has occurred.

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                              1
      ................                                                 2
      ................                                                 3
      ................                                                 4
      ................                                                 5
C --- SET UP VALUES TO BE USED IN PROBLEM                            6
      AREA = 1.5                                                       7
      RADIUS = 5.0                                                     8
      SMALL = 1.0E-4                                                   9
      ................                                                 10
C --- BEGIN ITERATION LOOP - MAXIMUM OF 9 ITERATIONS                   11
      ANGLE(1) = 1.0                                                   12
      DO 100  I = 1,9                                                  13
C --- COMPUTE CHANGE IN APPROXIMATE SOLUTION                           14
      ................                                                 15
C --- TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION                     16
      IF ( ABS(CHANGE(I)) .LT. SMALL )  GO TO 110                      17
C --- APPROXIMATION HAS NOT CONVERGED - COMPUTE NEXT APPROXIMATION     18
  100 ................                                                 19
      ................                                                 20
      ................                                                 21
C --- THE ITERATION PROCEDURE HAS CONVERGED                            22
  110 ................                                                 23
      ................                                                 24
      ................                                                 25
      ................                                                 26
      ................                                                 27
```

Line 17 contains the control statement that tests for convergence. If the current value of the difference between successive approximations is in absolute value less than the convergence criterion, then the problem is solved. For each of the nine iterations, a new value of CHANGE(I), the difference, will be used. As I is given the values 1, 2, 3, ... 9, CHANGE(I) will have the values of CHANGE (1), CHANGE (2), ... CHANGE (9). The IF statement will cause statement 110 to be executed next if CHANGE(I) is in absolute value less than the variable SMALL. In this case, the procedure is terminated.

It should be noted that the DO loop need not be executed nine times. The first time a value of CHANGE(I) satisfies the convergence criterion, the DO loop is terminated through transfer of control to another part of the program.

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                              1
      ................                                                 2
      ................                                                 3
      ................                                                 4
      ................                                                 5
C --- SET UP VALUES TO BE USED IN PROBLEM                            6
      AREA = 1.5                                                       7
      RADIUS = 5.0                                                     8
      SMALL = 1.0E-4                                                   9
      ................                                                 10
C --- BEGIN ITERATION LOOP - MAXIMUM OF 9 ITERATIONS                   11
      ANGLE(1) = 1.0                                                   12
      DO 100  I = 1,9                                                  13
C --- COMPUTE CHANGE IN APPROXIMATE SOLUTION                           14
      ................                                                 15
C --- TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION                     16
      IF ( ABS(CHANGE(I)) .LT. SMALL )  GO TO 110                      17
C --- APPROXIMATION HAS NOT CONVERGED - COMPUTE NEXT APPROXIMATION     18
  100 ................                                                 19
C --- END OF LOOP - ITERATION PROCEDURE HAS NOT CONVERGED              20
      STOP                                                             21
C --- THE ITERATION PROCEDURE HAS CONVERGED                            22
  110 ................                                                 23
      ................                                                 24
      ................                                                 25
      STOP                                                             26
      ................                                                 27
```

For practical reasons the number of iterations to be executed was limited to nine. In the event that nine iterations were insufficient, control of the program would pass to the next executable statement after statement 100 (line 19). This next statement (line 21) is a STOP statement which will cause the entire program to be terminated. Line 26 also contains a STOP statement, which ends the execution of the program after successful performance.

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                                    1
      ................                                                       2
      ................                                                       3
      ................                                                       4
      ................                                                       5
C --- SET UP VALUES TO BE USED IN PROBLEM                                   6
      AREA = 1.5                                                           7
      RADIUS = 5.0                                                         8
      SMALL = 1.0E-4                                                       9
      G = ( 2.0*AREA )/( RADIUS**2 )                                       10
C --- BEGIN ITERATION LOOP - MAXIMUM OF 9 ITERATIONS                       11
      ANGLE(1) = 1.0                                                       12
      DO 100  I = 1,9                                                      13
C --- COMPUTE CHANGE IN APPROXIMATE SOLUTION                               14
      CHANGE(I) = ( ANGLE(I) - SIN(ANGLE(I)) - G )/( 1.0 - COS(ANGLE(I))15
C --- TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION                         16
      IF ( ABS(CHANGE(I)) .LT. SMALL )  GO TO 110                          17
C --- APPROXIMATION HAS NOT CONVERGED - COMPUTE NEXT APPROXIMATION         18
  100 ANGLE(I+1) = ANGLE(I) - CHANGE(I)                                    19
C --- END OF LOOP - ITERATION PROCEDURE HAS NOT CONVERGED                  20
      STOP                                                                 21
C --- THE ITERATION PROCEDURE HAS CONVERGED                                22
  110 ................                                                     23
      ................                                                     24
      ................                                                     25
      STOP                                                                 26
      ................                                                     27
```

In line 10, a quantity with the variable name "G" is computed. Although G is used in line 15, it is computed on line 10 before the DO loop is executed. By removing the computation G to line 10, the value of $2A/R^2$ is computed only once, rather than each time line 15 is performed.

Line 15 is the FORTRAN arithmetic statement that corresponds to the formula for the difference between successive approximations. That formula is repeated here:

$$\text{Difference} = (X_n - \sin X_n - 2A/R^2) \; / \; (1 - \cos X_n)$$

Using the names that appear in the FORTRAN program, the formula may be rewritten as

$$\text{CHANGE}_n = (\text{ANGLE}_n - \sin \text{ANGLE}_n - G) \; / \; (1 - \cos \text{ANGLE}_n)$$

This formula is quite similar to line 15 in appearance.

The statement in line 19 produces the next (successive) approximation to the angle. This new value will be used to compute the new difference between successive approximations.

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                                    1
      ................                                                      2
   10 ................                                                      3
   11 ................                                                      4
   12 ................                                                      5
C --- SET UP VALUES TO BE USED IN PROBLEM                                   6
      AREA = 1.5                                                            7
      RADIUS = 5.0                                                          8
      SMALL = 1.0E-4                                                        9
      G = ( 2.0*AREA )/( RADIUS**2 )                                        10
C --- BEGIN ITERATION LOOP - MAXIMUM OF 9 ITERATIONS                        11
      ANGLE(1) = 1.0                                                        12
      DO 100  I = 1,9                                                       13
C --- COMPUTE CHANGE IN APPROXIMATE SOLUTION                                14
      CHANGE(I) = ( ANGLE(I) - SIN(ANGLE(I)) - G )/( 1.0 - COS(ANGLE(I))    15
C --- TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION                          16
      IF ( ABS(CHANGE(I)) .LT. SMALL )  GO TO 110                           17
C --- APPROXIMATION HAS NOT CONVERGED - COMPUTE NEXT APPROXIMATION          18
  100 ANGLE(I+1) = ANGLE(I) - CHANGE(I)                                     19
C --- END OF LOOP - ITERATION PROCEDURE HAS NOT CONVERGED                   20
      STOP                                                                  21
C --- THE ITERATION PROCEDURE HAS CONVERGED                                 22
  110 WRITE ( 2,10 )                                                        23
      WRITE ( 2,11 ) ( K, ANGLE(K), CHANGE(K), K = 1,I )                    24
      WRITE ( 2,12 )                                                        25
      STOP                                                                  26
      ................                                                      27
```

If the iteration procedure has converged, it is desired that the sequence of approximations and differences be put out on some external device in the form of a table. Three output statements are utilized to accomplish this. Let us assume that eventually a printed report will be prepared. The first output statement (line 23) will cause a heading line to be printed. The second (line 24) will cause a table to be printed. The last part of this statement, K = 1, I indicates how many lines the table will contain. The last value that I assumed before transfer from the DO loop will be this number of lines. The first line in the table will be values of K, ANGLE(K), and CHANGE(K) for K = 1, i.e., 1, the value of ANGLE(1), and the value of CHANGE(1). The second line would be the number 2, the value of ANGLE(2) and the value of CHANGE(2).

There will be a total of I lines to this table which contains three columns. The last WRITE statement (line 25) is used to print some final information. For each of these WRITE statements there will be a corresponding FORMAT statement, to be described on the next page. The statement numbers of these FORMAT statements are 10, 11 and 12 respectively. The number "2" in each of these output statements refers to some output device. This device assigned the number 2 would either be an on-line printer, or a magnetic tape, or punched cards. Either of the last two outputs could subsequently be used to produce a printer listing of the results.

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                                    1
      ................                                                       2
   10 FORMAT ( 9X  9HITERATION  5X  5HANGLE  9X  6HCHANGE  )                3
   11 FORMAT ( 13X  I1, F15.5, F14,5  )                                     4
   12 FORMAT ( 9X 38HTHE ITERATION PROCEDURE HAS CONVERGED. )               5
C --- SET UP VALUES TO BE USED IN PROBLEM                                   6
      AREA = 1.5                                                            7
      RADIUS = 5.0                                                          8
      SMALL = 1.0E-4                                                        9
      G = ( 2.0*AREA )/( RADIUS**2 )                                        10
C --- BEGIN ITERATION LOOP - MAXIMUM OF 9 ITERATIONS                        11
      ANGLE(1) = 1.0                                                        12
      DO 100  I = 1,9                                                       13
C --- COMPUTE CHANGE IN APPROXIMATE SOLUTION                                14
      CHANGE(I) = ( ANGLE(I) - SIN(ANGLE(I)) - G )/( 1.0 - COS(ANGLE(I))    15
C --- TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION                          16
      IF ( ABS(CHANGE(I)) .LT. SMALL )  GO TO 110                           17
C --- APPROXIMATION HAS NOT CONVERGED - COMPUTE NEXT APPROXIMATION          18
  100 ANGLE(I+1) = ANGLE(I) - CHANGE(I)                                     19
C --- END OF LOOP - ITERATION PROCEDURE HAS NOT CONVERGED                   20
      STOP                                                                  21
C --- THE ITERATION PROCEDURE HAS CONVERGED                                 22
  110 WRITE ( 2,10 )                                                        23
      WRITE ( 2,11 ) ( K, ANGLE(K), CHANGE(K), K = 1,I )                    24
      WRITE ( 2,12 )                                                        25
      STOP                                                                  26
      ................                                                      27
```

If we assume that the output unit 2 refers to a printer, then the FORMAT statement in line 3 controls the editing and spacing of the fields named in the output statement in line 24. The FORMAT statement contains codes having a definite interpretations:

13X means skip the next (in this case first) thirteen positions on the printer line.

I means convert the next (first) variable from integer representation on the printer line.

1 means allow one print position for this result.

F means convert the internal real representation of the next variable to a fixed point decimal representation on the printed line.

15 means that the rightmost position of the field should be fifteen print positions to the right of the previous field on the line.

.5 means to allow 5 decimal places.

F14.5 has a similar meaning to F15.5.

The FORMAT statements on lines 2 and 4 describe the spacing and printing of constant information contained in these FORMAT statements. Although there are no variables to be edited in the corresponding output statements on lines 23 and 25, spacing of the constant information is controlled by the two FORMAT statements. For example, line 4 indicates that the first nine print positions are to be skipped. 38H indicates that the next thirty-eight characters that follow the letter "H" are to be printed. Thus, "THE ITERATION PROCEDURE HAS CONVERGED. " will be printed on the printer. Similarly, FORMAT statement 10 causes successive spacing and printing of the words ITERATION, ANGLE and CHANGE. Convergence in this case occurs after three iterations. A copy of the output of this program follows.

```
ITERATION       ANGLE          CHANGE

    1          1.00000        0.08381

    2          0.91619        0.00742

    3          0.90877        0.00006

THE ITERATION PROCEDURE HAS CONVERGED.
```

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                                    1
      DIMENSION ANGLE(10), CHANGE(9)                                        2
   10 FORMAT ( 9X  9HITERATION  5X  5HANGLE  9X  6HCHANGE  )                3
   11 FORMAT ( 13X  I1, F15.5, F14.5  )                                     4
   12 FORMAT ( 9X 38HTHE ITERATION PROCEDURE HAS CONVERGED. )               5
C --- SET UP VALUES TO BE USED IN PROBLEM                                   6
      AREA = 1.5                                                            7
      RADIUS = 5.0                                                          8
      SMALL = 1.0E-4                                                        9
      G = ( 2.0*AREA )/( RADIUS**2 )                                        10
C --- BEGIN ITERATION LOOP - MAXIMUM OF 9 ITERATIONS                        11
      ANGLE(1) = 1.0                                                        12
      DO 100  I = 1,9                                                       13
C --- COMPUTE CHANGE IN APPROXIMATE SOLUTION                                14
      CHANGE(I) = ( ANGLE(I) - SIN(ANGLE(I)) - G )/( 1.0 - COS(ANGLE(I))    15
C --- TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION                          16
      IF ( ABS(CHANGE(I)) .LT. SMALL )  GO TO 110                           17
C --- APPROXIMATION HAS NOT CONVERGED - COMPUTE NEXT APPROXIMATION          18
  100 ANGLE(I+1) = ANGLE(I) - CHANGE(I)                                     19
C --- END OF LOOP - ITERATION PROCEDURE HAS NOT CONVERGED                   20
      STOP                                                                  21
C --- THE ITERATION PROCEDURE HAS CONVERGED                                 22
  110 WRITE ( 2,10 )                                                        23
      WRITE ( 2,11 ) ( K, ANGLE(K), CHANGE(K), K = 1,I )                    24
      WRITE ( 2,12 )                                                        23
      STOP                                                                  26
      ................                                                      27
```

A specification statement must be written to aid the processor in reserving space for the subscripted variables that appear in the program. The DIMENSION statement in line 2 reserves spaces for the maximum of ten values of ANGLE and nine values of CHANGE.

```
C --- SAMPLE PROBLEM USING BASIC FORTRAN                                    1
      DIMENSION ANGLE(10), CHANGE(9)                                        2
   10 FORMAT ( 9X  9HITERATION  5X  5HANGLE  9X  6HCHANGE  )                3
   11 FORMAT ( 13X  I1, F15.5, F14.5  )                                     4
   12 FORMAT ( 9X 38HTHE ITERATION PROCEDURE HAS CONVERGED. )               5
C --- SET UP VALUES TO BE USED IN PROBLEM                                   6
      AREA = 1.5                                                            7
      RADIUS = 5.0                                                          8
      SMALL = 1.0E-4                                                        9
      G = ( 2.0*AREA )/( RADIUS**2 )                                        10
C --- BEGIN ITERATION LOOP - MAXIMUM OF 9 ITERATIONS                        11
      ANGLE(1) = 1.0                                                        12
      DO 100  I = 1,9                                                       13
C --- COMPUTE CHANGE IN APPROXIMATE SOLUTION                                14
      CHANGE(I) = ( ANGLE(I) - SIN(ANGLE(I)) - G )/( 1.0 - COS(ANGLE(I)) )  15
C --- TEST FOR CONVERGENCE OF APPROXIMATE SOLUTION                          16
      IF ( ABS(CHANGE(I)) .LT. SMALL )  GO TO 110                           17
C --- APPROXIMATION HAS NOT CONVERGED - COMPUTE NEXT APPROXIMATION          18
  100 ANGLE(I+1) = ANGLE(I) - CHANGE(I)                                     19
C --- END OF LOOP - ITERATION PROCEDURE HAS NOT CONVERGED                   20
      STOP                                                                  21
C --- THE ITERATION PROCEDURE HAS CONVERGED                                 22
  110 WRITE ( 2,10 )                                                        23
      WRITE ( 2,11 ) ( K, ANGLE(K), CHANGE(K), K = 1,I )                    24
      WRITE ( 2,12 )                                                        25
      STOP                                                                  26
      END                                                                   27
```

The last statement, END, is a specification statement to the processor which informs it that the last line of the FORTRAN program has been written.

Everything necessary for the FORTRAN processor to use in compilation has been specified and the problem is complete.

3. CONTROL, ARITHMETIC AND LOGICAL STATEMENTS

Control Statements

Normally FORTRAN statements are executed sequentially. The control statements are used to alter the order in which FORTRAN statements are executed. By using control statements, the programmer can cause the computer to repeat or skip or interrupt a sequence of statements. This section presents all of the Control Statements in FORTRAN for the UNIVAC 1107. The control statements are:

DO
CONTINUE
unconditional GO TO
conditional GO TO
computed GO TO
ASSIGN
assigned GO TO
arithmetic IF
logical IF
hardware IF
PAUSE
STOP

A. DO

The DO statement is a command to execute repeatedly the statements which follow it up to and including the statement with a particular statement number. The set of statements which is executed repeatedly is called the range of the DO. The DO statement, together with its range, is called a DO-loop.

Through the use of the DO statement, the programmer may perform many basic tasks in computing. The first and probably most important task is counting. One way to instruct the computer to count is to prescribe an initial value to start the count, an increment to enable the computer to secure the next value in the count, and a final value, so that it might cease counting. For example, to count from three to eleven by twos, the numbers 3, 11 and 2 would have to be stated in the DO statement. The values 3, 5, 7, 9, and 11 are succesively assigned to a non-subscripted variable, called an index, whose name also appears in the DO statement. Frequently the initial value, final value and increment are not known, but are computed values, input values, or defined in a PARAMETER Statement (See Section II, A., 2. c.).

To accomodate these cases, the DO statement permits the use of non-subscripted integer variables in place of any or all of the integer constants that control the count. Counting, the basic facility of the DO, controls the process called looping, i.e., the repetition of a series of statements or steps for a prescribed number of times.

The DO statement is one of the few FORTRAN statements that can directly control the execution of one or more other FORTRAN statements. The statements it controls are those immediately following it, up to and including the statement whose statement number is written directly after the word DO in the DO statement.

Another advantage of using the DO is also directly connected to its counting ability. As the index of a DO changes, the value of subscript may also be changed. Thus by changing the index in a DO statement, different elements in a subscripted array may be referenced. For example, assume there were subscripted variables controlled by a DO-loop: A (K), B (2 *K), C (2*K-1). If the Index K were to assume the values one, two and three, then

A (1) , A (2) , and A (3) ,
B (2) , B (4) , and B (6) ,
C (1) , C (3) , and C (5)

could be referenced.

The DO statement may be written in either of two forms:

DO n i = j, k, m or DO n i = j, k

where n is a statement number, i is an integer variable, and j, k, and m are integer variables (positive or negative values), or integer constants.

i is the index which takes on a new value each time through the range of the DO.

j specifies the initial value of the index, i.e., the value of the index during the initial execution of the range.

k specifies the test value of the index, i.e. the value which the index cannot exceed (or be less than). As soon as i becomes greater than (or less than) k, execution of the range ceases, and control passes to the statement following statement n. The DO is then said to be satisfied.

m specifies the increment (or decrement) by which i is increased (or decreased) after each execution of the range.

None of the parameters of a DO may be changed within the range of the DO.

The second form of the DO statement (without m) is equivalent to the first form with m equal to 1. That is, if the increment is 1, then both the number 1 and the preceding comma may be omitted.

For example, the statement

```
DO  12  KK = 2, 10
```

will cause the execution of all statements following the DO statement, up to and including statement 12. KK will have the value 2 during the first execution of the range. During the succeeding executions of the range, KK will assume the values 3, 4, 5, . . . , 10. Then control will pass to the statement following statement 12.

m will either increment or decrement j according to the value of m as specified in the DO statement. The three values j, k, and m must all be consistent in order to generate a correct program. (e.g. if j > k, then m must be a negative quantity.

The index, i, is available for use in statements throughout the range as long as its value is not changed. It can be used either as a variable or in subscripts. The same is true of j, k, or m, if they are variables. The value of i is defined when control leaves the range of the DO other than by satisfaction of the DO. The value of i is also defined if it appears in a CALL statement and i is listed in a COMMON statement; if there is a reference to an internal subprogram and the index variable is global*; if there is a reference to an ABNORMAL function, and the index variable is in COMMON; or if the index variable is used as other than part of a subscript.

In any of these cases or in case i is used as a variable, then i is said to be materialized. This means that there is a definite storage location associated with i by the occurrence of any one of these events and that i is available for reference elsewhere in the program. (CALL and COMMON statements are discussed in Sections VII and V, respectively.) The indexing

---

*See Section VII E

parameter m must not be zero at the time the DO statement is encountered.

Example of a DO-loop:

```
      DO  12  I = 1, 25
   12 A (I) = B (2 * I-1)
```

The loop is equivalent to execution of the statements

```
A (1) = B (1)
A (2) = B (3)
A (3) = B (5)
   .
   .
   .
A (25) = B (49)
```

In this example, the integer variable I is the index; the initial value of the index is 1; the test value is 25; the increment is 1; and the range of the DO is statement 12.

The above example may be written so that the index is decremented as follows:

```
      DO  12  I = 25, 1
   12 A (I) = B (2 * I-1)
```

The loop is equivalent to execution of the statements:

```
A (25) = B (49)
A (24) = B (47)
   .
   .
   .
A (1) = B (1)
```

In this example, the variable I is the index; the initial value of the index is 25; the test value is 1; the decrement is 1; and the range of the DO is statement 12.

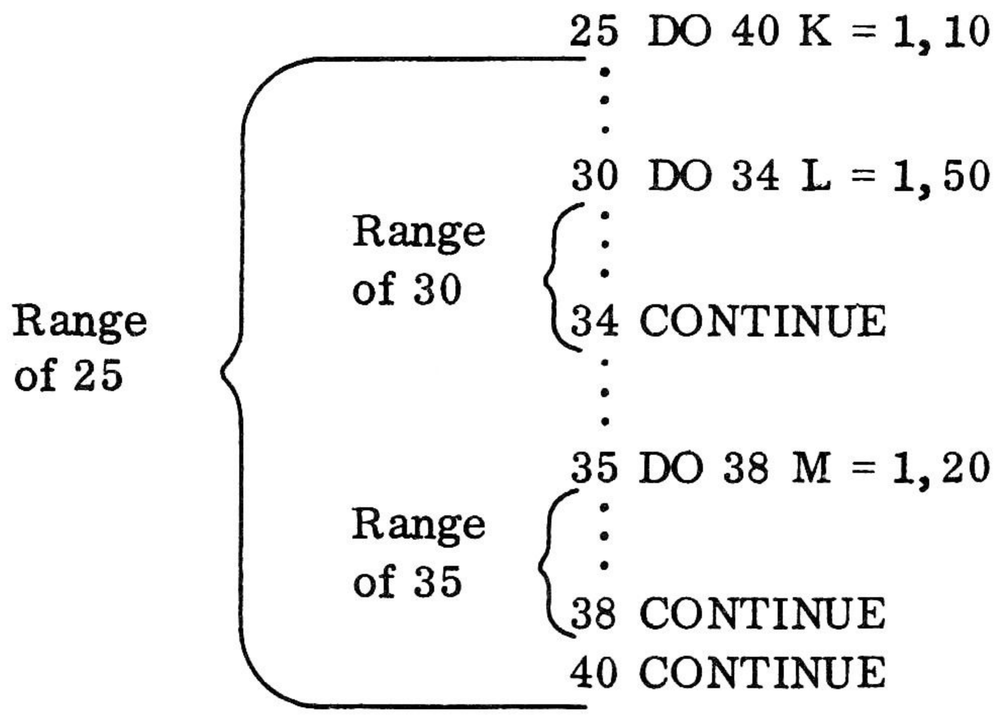
Nesting DO statements

A DO-loop may be contained within the range of another DO-loop. If this situation occurs, the following rules must be observed:

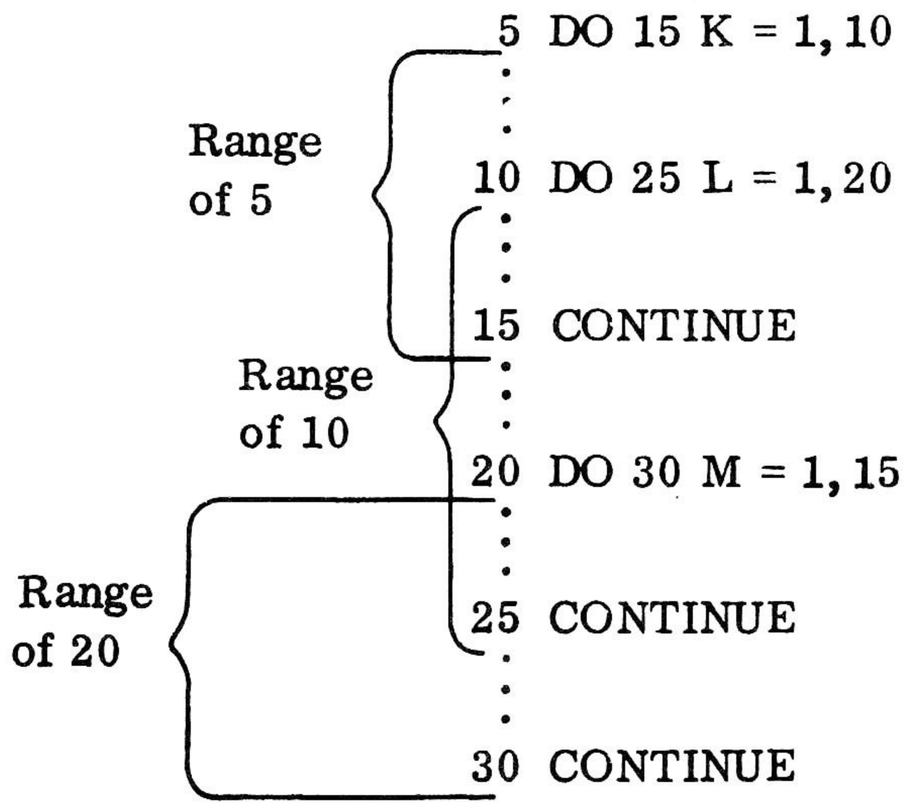
Rule 1: If the range of a DO includes another DO statement, then all of the statements in the range of the inner DO must lie within the range of the outer DO.

The following diagrams illustrate this rule.

Permitted



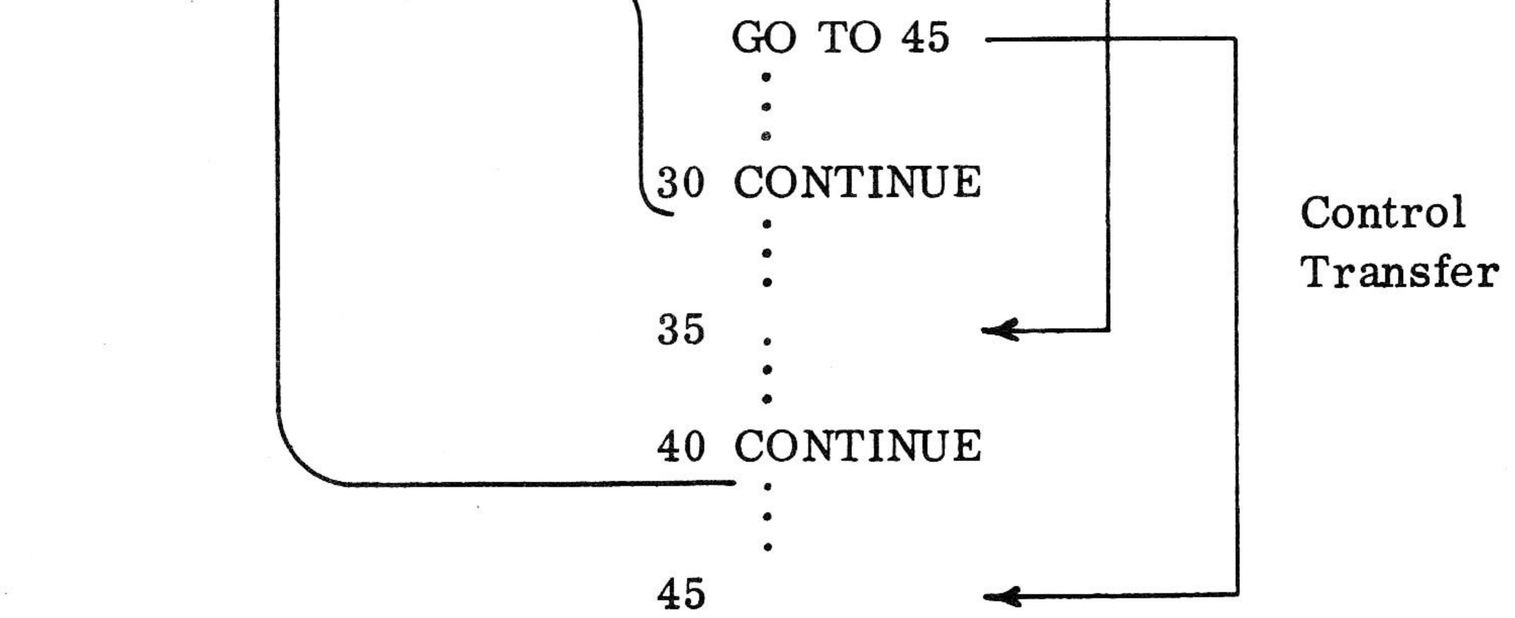
Violation of Rule 1



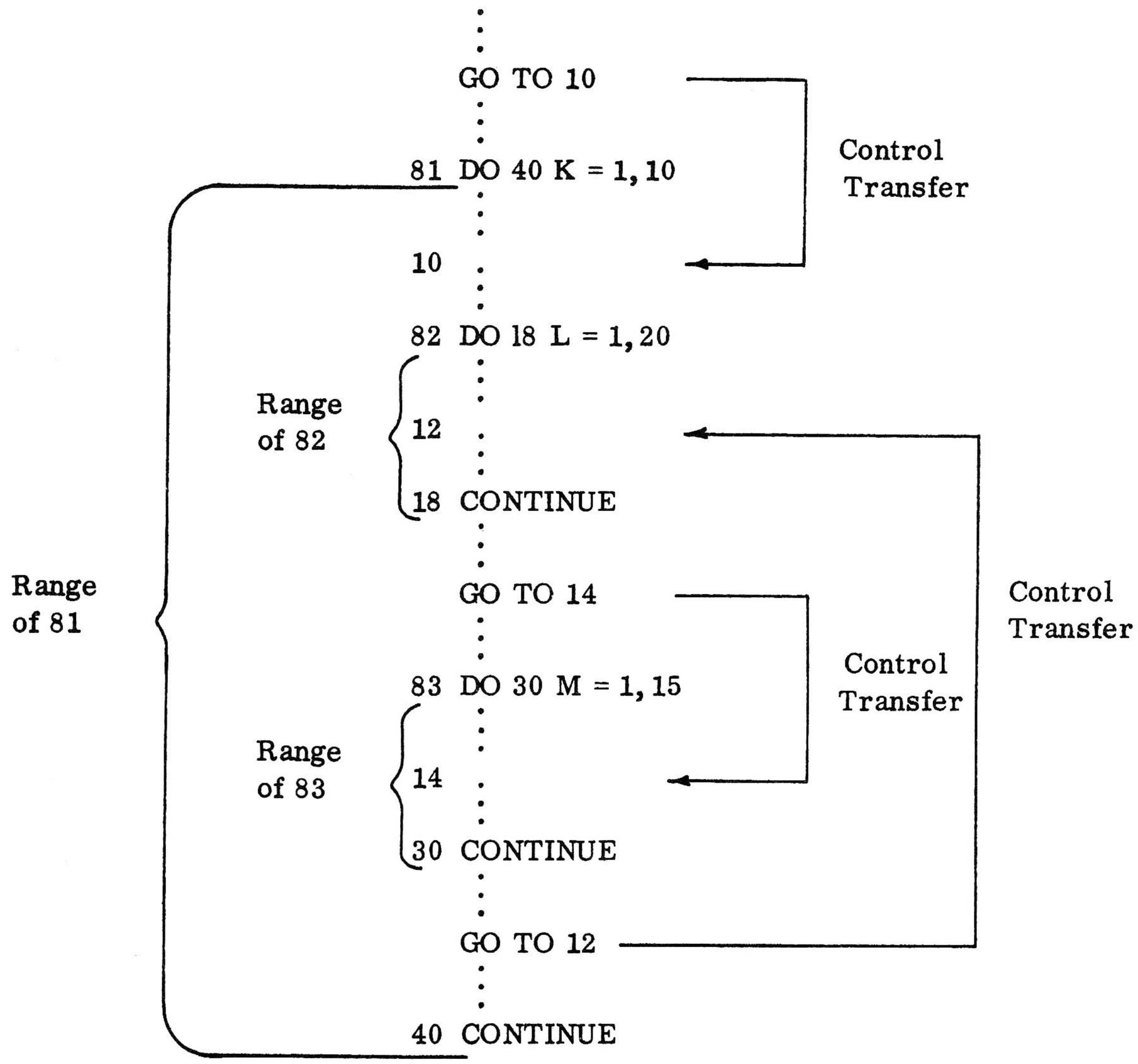
Rule 2: No transfer of control by IF or GO TO statements is permitted <u>into</u> the range of any DO statement from outside its range. Such transfers would not permit the DO-loop to be properly indexed.

The following diagrams illustrate this rule.

<u>Permitted</u>



Violations of Rule 2:



Note: Reference to a function or a subroutine within a DO-loop is not a violation of Rule 2.

Other Restrictions on Statements in the Range of a DO

There is only one kind of statement which is not permitted in the range of a DO. This statement is one which redefines the value of the index or any of the indexing parameters. The indexing of a DO-loop must be completely set before the range is entered and must be undisturbed by any statement in the range.

The last executed statement of the range of a DO cannot be any form of IF or GO TO statement. If it is necessary to end a DO-loop with the execution of some form of an IF or GO TO statement, then the dummy statement CONTINUE must be used to terminate the range of the DO-loop. This is the entire reason for the existence of the CONTINUE statement in the FORTRAN language.

For example:

```
13  DO  48  LL = 1,4
              .
              .
              .
45  GO  TO  60

48  CONTINUE
              .
              .
              .
60 .........
```

Since statement 45 is a GO TO statement, it cannot appear as the last statement in the range of a DO-loop. Therefore, the dummy statement CONTINUE has been used to terminate the range.

B. CONTINUE

CONTINUE is a dummy statement which gives rise to no instructions in the object program. CONTINUE must be used as the last statement in a DO-loop which would, without it, end with the execution of some form of an IF or GO TO statement. (See the discussion of CONTINUE in "Other Restrictions on Statements in the Range of a DO", which immediately precedes this paragraph.)

The form of the CONTINUE statement is:

CONTINUE

CONTINUE is necessary as the last statement of a DO-loop when it is desired conditionally to skip the last several statements in the range and proceed with the next iteration of the loop since the last executed statement would be some form of an IF statement. (The three forms of IF statements are discussed in III H, J, and L following.) The IF statement (conditional transfer) must GO TO the CONTINUE rather than to the first statement in the range in order for proper incrementation (or decrementation) of the index to take place.

For example:

```
      LAST = 50
      PROD = 1.0
      DO 13 N = 1, LAST
11    IF (A(N) .EQ. 0.0) GO TO 13
12    PROD = PROD * A(N)
13    CONTINUE
```

The DO-loop computes the product of non-zero numbers in the array A. Statement 11 (a logical IF statement) tests the values of the members of array A. If one of the members has a value of zero, then statement 12 is skipped. Clearly, the range of the DO must include statement 12. Adherence to the rules for usage for the dummy statement CONTINUE means that is must appear as the last statement in the range of the DO. Moreover, the conditional transfer of statement 11 must GO TO the CONTINUE statement.

C. Unconditional GO TO

This statement is used to unconditionally alter the normal sequential execution of statements. It indicates the statement that is to be executed next.

The form of an Unconditional GO TO statement is:

```
GO TO n
```

where n is the number of the next statement to be executed. This statement transfers control to the statement number n.

Examples:

```
GO TO 2

GO TO 17

GO TO 163
```

The Unconditional GO TO statement is used in the following example:

```
 9    ALPH = 3.0

      GO TO 17

10    ALPH = 2.0

17    BETA = ALPH * GAMMA
```

If control reaches statement 9, ALPH will be given the value 3.0. The statement GO TO 17 will cause statement 10 to be skipped and statement 17 to be executed next. Statement 17 will cause BETA to be given three times the value of GAMMA.

D. Conditional GO TO

The form of a Conditional GO TO statement is:

```
GO TO m
```

where m is a simple (switch) variable. This statement transfers control to the statement which was last assigned to m in the ASSIGN statement. (See below.)

Examples:

```
GO TO INIT

GO TO MATE

GO TO NOOP
```

The Conditional GO TO statement is used in the following example:

```
5     ASSIGN 11 TO JUMP
      .
      .
      .
10    ASSIGN 13 TO JUMP
      .
      .
      .
      GO TO JUMP
11    HEIGHT = 5.0
12    GO TO 14
13    HEIGHT = 10.0
14    RESULT = HEIGHT * SOLID
```

If statement 10 was the last ASSIGN statement executed, statements 11 and 12 would be skipped, and control would be transferred from the GO TO JUMP statement to statement 13.

E. Computed GO TO

The computed GO TO statement determines the statement to be executed next dependent upon the value of an integer variable. The computed GO TO is useful in implementing a multiple-path transfer of control.

The form of the computed GO TO statement is:

GO TO (transfer list), i

where i is a positive non-subscripted integer variable. The transfer list specifies the numbers of the statements to which control may be transferred dependent upon the value of i. The transfer list must be enclosed by parentheses and followed by a comma. The statement numbers in the transfer list must be separated by commas. The value of the integer variable determines the statement to which this command will transfer control. If i has the value 1, the computed GO TO will transfer control to the statement whose number appears first in the transfer list. If i has the value 2, control will be transferred to the statement whose number appears second in the transfer list.

The maximum value of the integer variable I must be identical to the number of entries in the transfer list. There are no restrictions on the value of I other than those on any integer variable except that it must be positive. The value of i must be set before the computed GO TO is encountered.

For example:

GO TO (7, 9, 12) , J

If J is given the value 2, control will be transferred to statement 9, the statement whose number is second in the list. Similarly, if J is given the value 3, control will be transferred to statement 12. It is not proper to give J the value 4 since there are only three entries in the transfer list.

If there is a chance that J were given a value greater than 3, it is the responsibility of the program to test this. The FORTRAN Processor does not make this test.

For example:

```
      DO  10  J = 1, 5
      .........
      GO  TO (6, 7, 8, 9, 11) , J
      .........
6     .........
7     .........
8     .........
9     .........
10    CONTINUE
      .........
11    .........
```

This illustration shows how certain steps in a DO-loop can be omitted selectively by means of a computed GO TO.

F. ASSIGN

The ASSIGN statement is used only for the assignment of a statement number to a variable which will later be used by a control statement which is not a DO.

The form of the statement is:

```
ASSIGN  n  TO  i
```

where n is a statement number and i is a non-subscripted integer variable that appears in a control statement.

The execution of an ASSIGN statement presets to n the destination of all Control Statements pertaining to i.

Although ASSIGN is not a Control Statement in the strict sense, it does determine subsequent program flow, and is necessary to the proper execution of Control Statements which refer to the switch variable i.

The ASSIGN statement may be used to assign the value of a variable in more than one Control Statement.

For example:

```
5    ASSIGN 10 TO J
      .
      .
      .
6    GO TO J, (10,20,30)
      .
      .
      .
7    GO TO J, (15,25,10)
      .
      .
      .
8    GO TO J
```

Statement 5 assigns the value 10 to J. This causes statements 6, 7, and 8 to be executed as if they had been written:

```
6    GO TO 10
      .
      .
      .
7    GO TO 10
      .
      .
      .
8    GO TO 10
```

Note that

ASSIGN 10 TO J

does <u>not</u> have the same meaning as

J = 10

In particular, the sequence

```
ASSIGN 10 TO J

K = J + 1
```

will <u>not</u> produce a meaningful result.

G. <u>Assigned GO TO</u>

Like the computed GO TO, the assigned GO TO indicates the statement number of the next statement to be executed dependent on the value of an integer variable. However, in the assigned GO TO the value of the integer variable is the <u>statement number</u> itself, rather than an indication of the position of the statement number within the transfer list. The value of the integer variable can be assigned <u>only</u> by means of an ASSIGN statement. The assigned GO TO provides another method for implementing a multiple-path transfer of control.

The form of the assigned GO TO statement is:

```
GO TO i , (transfer list)
```

where i is non-subscripted integer variable appearing in a previously executed ASSIGN statement. The transfer list specifies the numbers of the statements to which control may be transferred. The transfer list must be enclosed by parentheses and preceded by a comma. The statement numbers in the transfer list must be separated by commas.

Example:

```
51  ASSIGN 4 TO J
     .
     .
     .
63  GO TO J, (7, 9, 4)
```

In the example statement 51 assigns the value 4 to J. This causes statement 63 to be executed as if it had been written:

```
63  GO TO 4
```

H. Arithmetic IF

The arithmetic IF statement determines the statement to be executed next dependent upon the value of an arithmetic expression. Before the arithmetic IF statement can be properly discussed, it is necessary to discuss Arithmetic Expressions.

1. Arithmetic Expressions

The simplest form of an arithmetic expression is a single quantity. This quantity may be an arithmetic constant, an arithmetic variable, or an arithmetic function. Both arithmetic constants and variables have been treated in Section II previously. Arithmetic functions may be considered as equivalent to arithmetic variables for the purposes of this discussion. A more complete treatment of arithmetic functions is to be found in SectionVII.

2. Arithmetic Operators

Compound arithmetic expressions may be formed by combining the simple arithmetic expressions through the use of the arithmetic operators. The arithmetic operators are:

+	ADD
-	SUBTRACT
*	MULTIPLY
/	DIVIDE
**	EXPONENTIATE

If A and B are any arithmetic expressions, then these operators are defined in the following manner:

A + B	means the value of A plus the value of B
A - B	means the value of A minus the value of B
A * B	means the value of A multiplied by the value of B (AB is a variable name)
A / B	means the value of A divided by the value of B
A ** B	means the value of A raised to the power B (A^B).

For example:

```
SUM + FIELD (J)
0.7854 ** 1.333
7 **I
X(I) / SQRT( Z)
SIN(R) * COS( T)
3.14 + A(I)
```

The application of an arithmetic operator to a pair of simple arithmetic expressions to form a compound arithmetic expressions results in an arithmetic expression. In fact, any arithmetic expressions may be operated upon in pairs by arithmetic operators, always yielding arithmetic expressions. This is illustrated by the following examples of arithmetic expressions:

```
(SUM + FIELD (J)) + 13.6
(SIN(R) * COS( T)) + (3.14 + A(I))
7.0 ** D + (X (I) / SQRT( Z))
```

Where arithmetic expressions are combined using operators, it is not permitted that two operators may appear consecutively. This rule is obvious if the fact that operators are defined only in terms of pairs of arithmetic expression is recalled. In mathematics, such operators are called binary operators, i.e., operators that operate on two objects. The only failure to observe this rule, when writing FORTRAN programs, stems from the dual nature of the minus (-) symbol. The subtraction operator and a negative arithmetic expression both use this same symbol in FORTRAN (as they do in mathematics). In FORTRAN, however, the use of the minus symbol to denote a negative arithmetic expression must be carefully distinguished from its use to denote the subtract operator. This is easily done by the use of parentheses as follows:

```
3.0 / ( - ALIFT)
SQRT (TIME) * ( - 36.5 + RADIUS)
LENTH ** ( - 4)
```

In expressions with three or more variables, some means for describing the exact order in which operations are to be performed is necessary. For example, the expression

A + B * C

may be computed in two ways. One way is to add A and B and multiply the sum by C. The other alternative is to multiply B by C and add A to the product. In FORTRAN, the latter calculation is correct. In all cases where such confusion can occur, the FORTRAN language prescribes a set of firm rules. These rules are called ordering rules or precedence rules. The programmer can either override or ignore these rules by using parentheses to clarify his intent to the FORTRAN processor. If he had meant the first interpretation (add A to B then multiply the result by C), he could write the expression as

(A + B) * C (Note: the * is necessary)

If his intent were the second interpretation, he could write

A + B * C

as before or

A + (B * C)

The pair of parentheses is redundant to the FORTRAN processor, but simplifies reading and comprehension. Free usage of parentheses is recommended.

3. Ordering Rules for Arithmetic Expressions

In the absence of parentheses, the ordering rules for operations in arithmetic expressions are:

operation	name	
**	Exponentiate	Evaluated first
* and /	Multiply and Divide	
+ and -	Add and Subtract	Evaluated last

Exponentiation is said to have a higher order than multiplication and division, which themselves have a higher order than addition and subtraction. This ordering is completely illustrated by the following examples:

2 ** 3 - 4	gives 4	Exponentiation is performed before addition or subtraction.
2 ** 4 / 2	gives 8	Exponentiation is performed before division or multiplication.
4 + 6 * 2	gives 16	Multiplication is performed before addition or subtraction.

The expression

4 + 3 * 2 - 6 / 2 would produce 7 as a result, whereas

((4 + 3) * 2 - 6) / 2 would produce 4.

When two or more operations of the same level are to be computed, the ordering rules require computation from left to right. For example:

16 / 2 * 4 gives 32

Multiplication and division are operations of the same level. The left to right ordering rule gives 32, whereas computing from the right end of the expression to the left would give 2. If it were intended that the multiplication occur first, the expression should be written

16 / (2 * 4)

4. Mode of an Arithmetic Expression

The mode of an Arithmetic Expression is determined by the following rules:

a. A variable or constant is an expression of its own mode. For example:

1)	ALPHA	real mode
2)	5.6	real mode
3)	BETA	double-precision mode (assuming that it was defined in a DOUBLE PRECISION type statement)
4)	10.8 D+0	double precision mode
5)	10	integer mode
6)	MEAN	integer mode
7)	GAMMA	complex mode (assuming that it was defined in a COMPLEX type statement)

b. A combination of two or more variables or constants is best explained by the following examples:

1)	ABLE + BAKER	The result of the expression is in the real mode.
2)	TIME + INC	The result of the expression is in the real mode, even though INC is an integer variable.
3)	KIND * LOT	The result of the expression is in the integer mode.

c. A variable or constant must not be complex if the associated variable or constant in an expression is double-precision. For example:

1)	VOLUME * DENS	This expression is not permitted if Volume is a complex variable and DENS is a double-precision variable.

d. * An expression is in the complex mode if any one of the associated variables or constants are in the complex mode. If one variable is complex and the other is real or integer, then the operation will be performed by treating the real or integer quantity as the real part of a complex quantity with an imaginary part of zero. For example:

1) BOB + MARY — If BOB is a complex variable, and MARY is an integer variable, then MARY is converted to the real mode and used as the real part of a complex quantity with an imaginary part of zero. MARY is then added to BOB.

e. * In an expression, if one quantity is real or integer and another is double-precision, then the operation will be performed in double-precision mode by supplying the single-precision quantity with a least significant part of zero. If any element of an expression is double-precision, the entire expression is computed in double-precision, with the following exceptions:

1) The mode of an expression which is a function argument does not effect the mode of any other argument, any expression element outside of the function's arguments, or the mode of the function itself.

2) The mode of an expression which contains a function reference does not effect the mode of the function or any argument of the function.

5. Integer Arithmetic

If a division is performed in the integer mode, then the fractional part is truncated (not rounded).

5/4 will give the result 1.

4/5 will give the result 0.

---

*In d. and e., an integer will be converted to a real quantity prior to use.

But in real mode

5.0/4.0 will give the result 1.25.

4.0/5.0 will give the result .8--quite different from the integer mode result !

It is now possible to complete the discussion of the arithmetic IF statement.

The form of the arithmetic IF statement is:

IF (arithmetic expression) j, k, m

where j, k, m are statement numbers, or simple (switch) variables which have been previously assigned a statement number by an ASSIGN statement. The arithmetic expression must be enclosed in parentheses. The statement numbers of simple variables must be separated by commas.

If the value of the arithmetic expression is negative, control is transferred to the statement numbered j. If the value is zero, control is transferred to the statement numbered k. If the value is positive, control is transferred to the statement numbered m.

For example:

IF (X-Y) 101, 105, 110	If the value of X-Y is negative, statement 101 is executed next. If the value is zero, statement 105 is executed next. If the value is positive, statement 110 is executed next.
IF (NUMBER) 5, 10, 5	If the value of NUMBER is zero, control is transferred to statement 10; otherwise control is transferred to statement 5.
IF (J-6) I, I, K	If the value of J is less than or equal to 6, statement I is executed next. If the value of J is greater than 6, statement K is executed next. (Note that I and K must have been previously assigned a statement number by an ASSIGN statement.)

I. Arithmetic Statements

One of the most useful statements in the FORTRAN language is the arithmetic statement. The general form of the arithmetic statement is:

Arithmetic Variable = Arithmetic Expression

This statement serves a dual purpose. The first is to cause the computation of the value of the arithmetic expression that appears on the right hand side of the equals symbol (=). The resulting number then replaces the previous value of the variable whose name appears on the left hand side of the equals symbol. This replacement operation, the second purpose of the FORTRAN arithmetic statement, is most important. It is one of the few ways that the value of a variable may be changed. This use of the equals symbol in FORTRAN is quite different from its use in ordinary mathematics. In mathematics, the symbol establishes identity between two sides of an equation. In a FORTRAN arithmetic statement, its meaning is replacement. It is an operator of a different nature than the arithmetic operators: it could be called "a binary replacement operator". The left hand side is always a single variable which names the "result" field.

In mathematics, a statement such as

N = N + 1

is false, whereas in FORTRAN it is a perfectly correct statement which has the effect of adding one to the current value of N and placing the result in N.

As the word "variable" is used in the above paragraph, it refers to either a non-subscripted or a subscripted variable. The use of one or the other depends, naturally, on the problem itself. For example, in the general form of the statement given above, the left-hand side may be a subscripted variable:

SUM (I) = FIRST + SECOND (J)

The variable on the left of an arithmetic statement does not have to be in the same mode as the expression on the right. The following rules and examples will explain this.

1.	I = A * B	If I is in the integer mode, and A * B is in the real mode, A * B will be computed in the real mode and truncated to the largest integer less than or equal to the resultant multiplication, A * B; I will then assume this integer value.
2.	A = K + I	If A is in the real mode, and K + I is in the integer mode, K + I will be computed in the integer mode and then converted to the real mode; A will then assume this real value.
3.	Z = U / V	If Z is in the double-precision mode, U / V will be computed in the double-precision mode. However, in the case of function arguments, the mode of computation of each argument is dictated solely by the rules specified in the Mode of an Arithmetic Expression above.
4.	T = W - Y	If T is in the real mode, and W - Y is in the double-precision mode, W - Y will be computed in double-precision, and the most significant part of the result will be stored in T.
5.	N = C + D	If N is in the integer mode and C + D is in the double-precision mode, C + D will be computed in double-precision. The most significant part of the result will be truncated to an integer, and the result stored in N.
6.	S = J * L	If S is in the double-precision mode and J * L is in the integer mode, J * L will be computed in the integer mode, and the result will be converted to real and stored as the most significant part of S. The least significant part of S will be zero.
7.	E = F + G	If E is in the complex mode and F + G is in the integer or real mode, F + G will be computed accordingly, the real part of E will be set to the real value of F + G, and the complex part of E will be set to zero.
8.	I = Q	If I is in the real or integer mode and Q is in the complex mode, then Q is computed in the complex mode and the real part of Q is set equal to I.

J. Logical IF

The logical IF statement is very similar to the arithmetic IF statement. This statement permits a programmer to change the sequence of statement execution depending upon the value of a logical expression. Before the logical IF statement can be properly discussed, it is necessary to discuss Logical Expressions.

1. Logical Expressions

A logical expression has the value .TRUE. or .FALSE. The periods are part of the logical value notation and must be present. The simplest form of a logical expression is a single quantity. This quantity may be one of the following:

a logical constant, either .TRUE. or .FALSE.

a logical variable such as SWITCH*

a logical function such as TEST (SWIT 1, SWIT 2)

Logical functions may be considered as equivalent to logical variables for the purpose of this discussion. A more complete treatment of logical functions is to be found in Section VII.

2. Logical Operators

Compound logical expressions may be formed by combining simple logical expressions through the use of the logical operators. The logical operators are:

.OR.

.AND.

.NOT.

The periods are part of the logical operator notation and must be present.

---

*SWITCH must appear in a LOGICAL type statement.

If A and B are logical expressions, then the logical operators are defined as follows:

A .OR. B	has the value .TRUE. if either A or B or both have the value .TRUE. ; otherwise it has the value .FALSE.
A .AND. B	has the value .TRUE. only if both A and B have the value .TRUE. ; otherwise it has the value .FALSE.
.NOT. A	has the value .TRUE. if A has the value .FALSE. ; it has the value .FALSE. if A has the value .TRUE.

For example:

.TRUE. .OR. .FALSE.	always has the value .TRUE.
.TRUE. .AND. .FALSE.	always has the value .FALSE.
SWIT 1 .OR. SWIT 2	may have the value .TRUE. or the value .FALSE. depending upon the values of the logical variables SWIT 1 and SWIT 2

It is evident that the logical operators .OR. and .AND. are binary operators from the discussion of arithmetic operators in Paragraph H preceding. Analogously with the case there, compound logical expressions formed using these two logical operators are logical expressions as well. Compound logical expressions may themselves be combined in pairs using .OR. and .AND. and always yield logical expressions.

.NOT. is not a binary operator, however. In fact, .NOT. seems to operate very analogously with the minus symbol (-) as used to denote a negative arithmetic expression. .NOT. is another kind of operator as is the usage of the minus symbol. .NOT. and minus (denoting negative arithmetic expression) are unary operators, i.e., operators which operate on only one object. Moreover, the object operated upon must be immediately to the right. In FORTRAN, .NOT. operates only on the logical constant or variable or parenthesized logical expression immediately to the right.

For example:

.NOT..TRUE. has the value .FALSE.

.NOT..TRUE..AND..FALSE. has the value .FALSE.

but,

.NOT.(.TRUE..AND..FALSE.) has the value .TRUE.

(X.AND..NOT.Y).OR(.NOT.X.AND.Y) may have the value .FALSE. depending on the values of the logical variables X and Y

It is now possible to proceed with the discussion of the logical IF statement.

The form of the logical IF statement is:

IF ("logical expression") "FORTRAN statement"

where the parentheses enclose a logical expression which has the value .TRUE. or the value .FALSE. Any executable FORTRAN statement may follow the parentheses, except a DO statement or another logical IF. If the logical expression has the value .TRUE., the remainder of the statement is executed. If the logical expression has the value .FALSE., the remainder of the statement is skipped over and control proceeds to the next statement.

The following example illustrates the use of a logical IF statement:

```
100 TF = .TRUE.
    ..........
    ..........
105 AREA = 0.0
110 IF (TF)  AREA =3.14159
120 SIGMA = AREA + 5.0
```

In statement 100 the logical variable TF is given the value .TRUE.
In statement 105 the arithmetic variable AREA is given the value 0.0.
In statement 110 the value of the logical variable TF is tested. Since TF has the value TRUE, the remainder of the statement is executed and AREA is given the value 3.14159.

In statement 120 the number 5.0 is added to the value of AREA and the sum (5.0 + 3.14159 = 8.14159) becomes the value of SIGMA.

Note: If the logical variable TF had had the value .FALSE., the remainder of statement 110 would not have been executed and control would have proceeded to statement 120. In that case, the value of AREA would have remained 0.0 and in statement 120, SIGMA would have been given the value 5.0.

3. Relational Operators

There are available in the FORTRAN language a further extension to the set of logical operators which are called relational operators. These are also binary operators and their application to logical expressions yield logical expression. However, these are distinguished as a special kind of logical expression by being called relational expressions. A relational expression is defined by a single relational operator which associates two arithmetic expressions. (The arithmetic expressions must be in the same mode.) The relational operators and their names are:

.GT.	greater than
.GE.	greater than or equal to
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to

The periods before and after the operator are part of the operator notation and must be present.

If I and J are any two arithmetic expressions, then the relational operators are defined as follows:

I .GT. J — has the value .TRUE. only if the value of I is greater than the value of J; otherwise it has the value .FALSE.

I.GE. J — has the value .TRUE. only if the value of I is greater than or equal to the value of J; otherwise it has the value .FALSE.

I.LT. J — has the value .TRUE. only if the value of I is less than the value of J; otherwise it has the value .FALSE.

I. LE. J — has the value of .TRUE. only if the value of I is less than the value of J; otherwise it has the value .FALSE.

I.EQ. J — has the value .TRUE. only if the value of I is equal to the value of J; otherwise it has the value .FALSE.

I.NE. J — has the value .TRUE. only if the value of I is not equal to the value of J; otherwise it has the value .FALSE.

For example:

X .LE. 5.7 — (This relational expression has the value .TRUE. only if the value of the arithmetic variable X is less than or equal to 5.7 . If the value of X is greater than 5.7, then the expression has the value .FALSE.)

N .NE. J+5 — (This relational expression has the value .TRUE. only if the value of the arithmetic variable N is not equal to the value of the arithmetic expression J+5. If the arithmetic variable N had the same value of the arithmetic expression J+5, then the relational expression would have the value .FALSE.)

4. Ordering Rules for FORTRAN Expressions

Parentheses may be used to override the ordering rules which specify which operations are to be performed when expressions are evaluated. In the absence of parentheses, the ordering rules are as follows:

Operation	Name	
Function Evaluation		Evaluated first
**	(Arithmetic) exponentiation	
* and /	(Arithmetic) multiplication and division	
+ and -	(Arithmetic) addition and subtraction	
.LT., .LE., .EQ., .NE., .GT., .GE.	relational operators	
.NOT.	Logical operator	
.AND.	Logical operator	
.OR.	Logical operator	Evaluated last

This chart is similar to the chart appearing under Ordering Rules for Arithmetic Statements earlier in this section. This is the complete chart for all types of FORTRAN expressions.

Any and all functions appearing in a FORTRAN expression would be evaluated first, then any exponentiation, arithmetic multiplication and division next, then addition and subtraction, the relational operators, the logical operators in the order given from top to bottom until finally, the .OR. operator could be evaluated.

In these levels of evaluation, any time an expression within a logical statement contains more than one operator of the same order, the operators are evaluated in the order they appear in the line reading from left to right.

The following example demonstrates how the ordering rules govern the order of evaluation of a logical expression.

The logical expression

```
(A+B .EQ. C+D)  .OR.  (S+T .LT. 7.0)
```

would be evaluated as if it had been written

```
((A+B) .EQ. (C+D))  .OR.  ((S+T) .LT. 7.0)
```

The expressions would be evaluated as follows:

a. The arithmetic expressions

(A+B) (C+D) (S+T) are evaluated first.

b. The relational expressions

((A+B .EQ. (C+D)) ((S+T) .LT. 7.0)

are evaluated and assume the value .TRUE. or .FALSE.

c. The logical expression

((A+B) .EQ. (C+D)) .OR. ((S+T) .LT. 7.0)

is then evaluated and assumes the value .TRUE. or .FALSE.

It is now possible to conclude the discussion of the logical IF statement by giving two examples of relational expressions (which are logical expressions) in use in logical IF statements.

Example 1

```
      IF (N .GE. 16)  GO TO 50
40    .........
```

This example illustrates the use of a relational expression only in a logical IF statement.

The expression (N .GE. 16) has the value .TRUE. only if the value of N is greater than or equal to 16; otherwise it has the value of .FALSE. If the relational expression has the value .TRUE., then the statement GO TO 50 is executed. If the relational expression has the value .FALSE., then the statement GO TO 50 is not executed and the control proceeds to statement 40.

Example 2

```
      IF ((X .GT. 5.0) .AND. (X .LT. 10.0))  Y = Y/3.14159
25    .........
```

This example illustrates the use of logical and relational operators in a logical IF statement. The effect of this statement is the modification of the value of Y, if the value of X lies between 5.0 and 10.0; otherwise the value of Y remains unchanged.

That is, the relational expression (X .GT. 5.0) has the value .TRUE. only if the value of X is greater than 5.0; the relational expression (X .LT. 10.0) has the value .TRUE. only if the value of X is less than 10.0; the logical expression (X .GT. 5.0) .AND. (X .LT. 10.0) has the value .TRUE. only if both relational expressions have the value .TRUE.

If the logical expression has the value , FALSE., the statement Y = Y/3.14159 is not executed but ignored. Control is then transferred directly to statement 25.

K. Logical Statements

The logical statement is very similar to the arithmetic statement. The general form of the logical statement is:

Logical Variable = Logical Expression

The arithmetic statement computes the value of an arithmetic expression and replaces the previous value of an arithmetic variable. The logical statement computes the value of a logical expression and replaces the previous value of a logical variable. Arithmetic variables and expressions take on numerical values. Logical variables and expressions can only take on the logical values .TRUE. or .FALSE. The value of a logical variable or expression can be determined by using a logical IF statement. Some examples of the simplest form of logical statements are the following:

```
SWITCH = .FALSE.
```

The logical variable SWITCH is given the constant value .FALSE.

```
POINT (2*J - 3) = TEST (SWITCH, PATH)
```

The subscripted logical variable, POINT (2*J - 3) is given the value .TRUE. or .FALSE. depending on the result of the logical function named TEST.

Some examples of logical statements that include logical operators are the following:

```
POINT (2*J -3) = ON .OR. TEST(SWITCH, PATH)
```

If both result of the logical function TEST and the logical variable ON are .FALSE., then POINT (2*J-3) is given the value .FALSE.; otherwise it is given the value .TRUE.

```
SELLS =  FREE .AND. EASY
```

If both the logical variables FREE and EASY are .TRUE., then the logical variable SELLS will be given the value .TRUE. If either FREE or EASY is .FALSE., then SELLS will be .FALSE.

```
SELLS = .NOT. FREE .AND. EASY
```

If FREE has the value .FALSE. and EASY has the value .TRUE., then and only then will SELLS be given the value .TRUE.

An example of the use of relational expressions in logical expressions is

```
SWITCH = (ALPHA .EQ. BETA)
```

If the values of ALPHA and BETA are numerically equal, SWITCH will be given the value .TRUE.

An example that contains both a relational operator and a logical operator is:

```
FLAG = (A .GT. B) .OR. SWITCH
```

If the numerical value of A is not greater than B and the value of SWITCH is .FALSE., then the logical variable FLAG is given the value .FALSE. If either A is numerically greater than B or SWITCH is .TRUE., then FLAG is given the value .TRUE.

L. Hardware IF

Computers differ from one another with regard to "hardware" switches and indicators. The status of these switches or indicators may be modified under internal or external control. A method has been provided within UNIVAC 1107 FORTRAN language whereby the following actions may be taken:

1. The status of the switches or indicators may be modified.

2. The status of the switches or indicators may be tested and control may be transferred, depending on the status of the switch or indicator.

These tests are accomplished by Subroutine subprograms which are referenced by CALL statements. Given that i is an integer expression and j is any integer variable, the Subroutine subprograms are as follows:

1. SLITE (i). If i = 0, all sense lights will be turned OFF. If i = 1, 2, 3, or 4, 5, 6, the corresponding sense light will be turned ON.

2. SLITET (i, j). Sense light i (1, 2, 3, 4, 5 or 6) will be tested and turned OFF. The variable j will be set to 1 or 2, if the tested sense light was ON or OFF, respectively.

3. SSWTCH (i, j). Sense switch i (1, 2, 3, 4, 5, or 6) is tested and j is set to 1 or 2 if the tested sense switch was DOWN or UP, respectively.

4. OVERFL (j). The variable j is set to 1 if an overflow condition exists, or to 2 if a non-overflow condition exists. The machine is left in a non-overflow condition after execution.

5. DVCHK (j). The variable j is set to 1 or 2 if the Divide Check indicator was ON or OFF, respectively. The Divide Check indicator is turned OFF.

In order to transfer control as a result of any one of the above tests, the IF or computed GO TO statement is used. For example,

```
      6  CALL SLITE (I)
         .
         .
         .
      7  CALL SLITET (I, J)
         .
         .
         .
      8  IF (J - 1) 1, 2, 3
         .
         .
         .
      2
         .
         .
         .
      3
```

In the above example if I equals 1, statement 6 will set sense light 1, ON. Statement 7 tests sense light 1; if it is ON, J is set to 1; if it is OFF, J is set to 2. Statement 8 will transfer control to statement number 2 if J equals 1 and to statement number 3 if J equals 2.

The following "hardware" IF statements which have been used in previous versions of FORTRAN will be accepted by UNIVAC 1107 FORTRAN:

SENSE LIGHT i

IF (SENSE LIGHT i) n_1, n_2

IF (SENSE SWITCH i) n_1, n_2

IF ACCUMULATOR OVERFLOW n_1, n_2

IF QUOTIENT OVERFLOW n_1, n_2

IF DIVIDE CHECK n_1, n_2

M. PAUSE

Note: The execution of a program which contains a PAUSE statement, will require the manual intervention of the computer operator. For this reason, the use of the PAUSE statement is discouraged.

The form of the PAUSE statement is:

PAUSE n

where n is any integer constant.

The PAUSE statement will cause the object program to halt during execution and n to be displayed. If n is not specified, no number will be displayed. The object program will await restarting by the operator. Upon restart by the operator, the program will resume execution with the next FORTRAN statement.

N. STOP

The STOP statement causes the immediate termination of execution of the object program. When the program is terminated, control is transferred to the executive routine.

The form of the STOP statement is:

STOP

The STOP statement should occur at the logical end of the program, rather than the physical end. More than one STOP statement may be used in a program.

Note: Some previous versions of FORTRAN permit the use of the FORTRAN statement CALL EXIT. The statement CALL EXIT will have the same effect as the execution of statement STOP.

4. INPUT AND OUTPUT STATEMENTS

A. GENERAL

The input-output statements are used to control the flow of data into and out of the computer. These statements enable the programmer to determine the kind and amount of data to be transmitted, the external medium to be used (punched cards, printed pages, or magnetic tape), and the format of the external data representation, i.e., the number of spaces, lines to be skipped, etc.).

The input-output statements are:

FORMAT
READ (Unit) List
READ (Unit, Format) List
WRITE (Unit) List
WRITE (Unit, Format) list
END FILE unit
REWIND unit
BACKSPACE unit

The two READ statements and the two WRITE statements cause the transmission of data between external media and computer storage. Associated with these statements is the list of FORTRAN variable names of the data to be transmitted. The order of the variable names in the list must be the same as the order in which the data fields exist on an input medium or will exist on an output medium. Data originating from sources other than the computer may be represented in formats different than normally expected by the computer. Similarly, for output, a choice of the representation on cards, tape, printed page, or other medium is provided.

The linkage between the internally stored data and its external representation on some output medium is accomplished through the specification of an editing format. This is done by specifying, in the output or input statement, the statement number of a FORMAT statement. The FORMAT statement contains a set of editing codes which enable the translation of internal representation to external representation. In the READ and WRITE statement above, the statement number of the FORMAT statement associated with each replaces the word Format. It is also possible to introduce editing formats during the execution of the object program. This is discussed in detail in the FORMAT paragraph following.

There are both READ and WRITE statements given above which do not have provision for editing format reference. These are used where work tapes or other internal storage extensions are used. Under these conditions, Format Specification is not required. Data in internal representation may be written out using this WRITE statement and subsequently read back into the computer using this READ statement.

As it appears in the statements above, unit is a logical reference which can, through the use of an input-output unit table, specify any desired input-output device. The word unit may be replaced in an input-output statement by an integer constant or a nonsubscripted integer variable.

B. List

The simplest way to introduce a List is through the use of examples. If ANGLE is a simple variable (or the name of an array) then the statement:

```
READ (3,14) ANGLE
```

will cause the reading of the simple variable ANGLE (or the entire array) from the input unit logically designated as 3, edited by the FORMAT statement numbered 14, into the proper position(s) of computer storage. In this case, the single FORTRAN name ANGLE is the entire List.

In the illustrative FORTRAN Program in Section II, when convergence occurred it was desired to print out a table. Line 25 of the program reads:

```
WRITE (2,11) (K, ANGLE(K), CHANGE(K), K=1, I)
```

This specifies the writing on output unit 2, under control of the FORMAT statement number 11, the elements of the List. The second set of parentheses enclose the List. If a List is of this form, the entire List should be enclosed in parentheses.

```
(K, ANGLE(K), CHANGE(K), K = 1, I)
```

First the variables are named in the list: K, ANGLE (K) and CHANGE (K). Then the extent of the table to be printed is given: K=1, I. The first line to be printed will contain a 1 for K, the value of the angle in the first iteration, ANGLE (1), and the amount of change from the first approximation, CHANGE (1).

It appears as follows:

1 1.00000 0.08381

Then K would be changed to 2, and the values resulting from the second iteration would be printed, as follows:

2 0.91619 0.00742

The output would continue until the value of K reaches the value of I.

A List may be relatively simple as those above or more complicated depending on the nature of the desired input or output data. The following illustrates a more complicated List.

Z, Y(5), (X(N), W(N, 2), N=1, 8), ((V(N, M), N=1, 8, 2), U(M, 3), M=1, 2)

When the above List is used with an output statement, the information will be written on the output medium in this order:

Z, Y(5), X(1), W(1, 2), X(2), W(2, 2), ... until X(8), W(8, 2), then V(1, 1), V(3, 1, ... until V(7, 1), then U(1, 3), V(1, 2), V(3, 2), ... until V(7, 2) and finally U(2, 3).

Similarly, if this List were used with an input statement, the successive words as they were read from the external medium would be placed into the sequence of storage locations as given above.

Thus, Lists of the form given in the second and third examples are interpreted by the processor from left to right with repetition for variables enclosed within parentheses. Only variables, and not constants, may be listed. The execution is exactly that of a DO-loop, as though each opening parenthesis (except subscripting parentheses) were a DO, with indexing given immediately before the corresponding closing parenthesis, and with the DO range extending up to the comma preceding that indexing information.

The order of the above List can thus be considered the equivalent to the following "output (or input) program".

```
      Z
      Y (5)
1     DO 2 N = 1, 8
      X (N)
2     W (N, 2)
3     DO 4 M = 1, 2
5     DO 6 N = 1, 8, 2
6     V (N, M)
4     U (M, 3)
```

The general form of a DO-type List is:

$$(A_1, A_2, \ldots A_n, I = m_1, m_2, m_3)$$

where each A is a subscripted or nonsubscripted variable name, I is a nonsubscripted integer variable, and m_1, m_2, m_3, are constants or nonsubscripted integer variables. Care must be taken to insure that the nonsubscripted integer variable used is not currently in use by a DO statement whose range includes the List. Usually each A is subscripted, and the subscript usually contains the index, I. (See Section III for more information on the DO statement.)

The range of the DO-type List in input/output statements must be clearly defined by means of parentheses. Only variables may appear in the List. Constants may appear only as indexing parameters or as parts of subscripts.

All quantities (variables, subscripted variables, or parenthetical expressions) in Lists must be separated by commas.

C. FORMAT

Where internally stored data is to be externally produced or where external data is to be introduced into the computer for internal use, it is necessary to specify the FORMAT of the data. This is done, by means of a FORMAT statement. This FORMAT statement is referenced by statement number by either the READ (Unit, Format) List or the WRITE (Unit, Format) List statement.

The general form of the FORMAT statement is:

FORMAT (Format Specification)

where Format Specification is composed of a collection of editing codes discussed below.

The FORMAT statement specifies the type of editing or conversion to be performed upon each member of a List. FORMAT statements are considered to be non-executable since they do not result in the production of object program instructions by the FORTRAN processor. They are actually, therefore, specification statements but are presented in this section rather then in the following section because of their intimate association with input-output statements.

1. Numeric Fields

There are five editing control symbols governing editing numeric data.

Symbol	Editing Action			General Editing Code
	Internal		External	
I	Integer variable	to/from	decimal integer	Iw
E	Real variable	to/from	floating point decimal number	Ew.d
D	Double-Precision variable	to/from	floating point decimal number	Dw.d
F	Real variable	to/from	fixed point decimal number	Fw.d
O	Binary number	to/from	octal integer	Ow

In the table above, w and d are unsigned integer constants.

Editing codes are written by field, from left to right, using the editing control symbols given above. The unsigned integer constant w specifies the width of a given field. w may be greater than required to contain a given number, thus providing spacing between numbers on an output medium. Where w is larger than needed, the contents of the field are

right justified. The unsigned integer constant d specifies the number of decimal places to be represented to the right of the decimal point in a field under control of an E -, D-, or F-type editing code.

Editing codes for successive fields must be separated by commas. The entire set of editing codes for a given FORMAT statement cannot provide for more characters than the input-output unit record may contain. Thus, a set of editing codes for printed output cannot provide for more than 128 characters per line, a set of editing codes for an 80 column card input or output cannot provide for more than 80 characters, and a set of editing codes for 90 column card input or output cannot provide for more than 90 characters.

Data to be treated by O-type editing codes must be given real or integer variable names. Internally, such data will be represented as a 12 octal-digit number.

As an illustration, the FORMAT statement

```
FORMAT (I3, E9.2, F8.3, O6)
```

might cause a line of printed output to appear as

```
136Δ0.17EΔ06ΔΔ-0.543073421
I3  E9.2    F8.3  O6
```

where Δ denotes a blank space. Each editing code is related to the field that it governs.

2. Alphanumeric Fields

UNIVAC 1107 FORTRAN provides two means by which alphanumeric information may be either read or written. The editing control symbols used are A and H. The editing codes using these symbols are Aw and wH, where w is an unsigned integer constant. Alphanumeric information associated with the editing code wH is generally used for header or fixed alphanumeric fields and is not available to the FORTRAN programmer internally for any purpose. Alphanumeric information associated with the Aw editing code is available through FORTRAN variable names to the programmer for whatever purpose he may desire.

Where the Aw editing code is used, six-bit UNIVAC 1107 alphanumeric character codes will be used for internal representation. Since alphanumeric information governed by the Aw editing code requires naming, it is necessary to know that a real or integer variable may contain up to 6 alphanumeric characters. If it is desired to have longer alphanumeric character strings for use internally, then either real variable or integer variable arrays may be used.

If the editing code wH is used, it must be immediately followed by w alphanumeric characters. Blank spaces appearing in the w alphanumeric characters following the editing code wH <u>are</u> counted as characters. This is the only usage of blank spaces in the FORTRAN language where they are not ignored by the FORTRAN processor.

For example:

```
41HΔTHISΔISΔANΔEXAMPLEΔOFΔALPHANUMERICΔDATA.
```

The editing action of the editing symbol H is dependent upon whether the set of editing codes in which it appears are used for FORMATting input or output. An example of the action on output FORMATting is:

```
6 FORMAT (5HMASS=F10.3)
```

Note that no comma is required to separate wH editing code from the subsequent editing code. This FORMAT statement might produce the following line of printing:

```
MASS=ΔΔ4326.437
MASS=ΔΔΔΔ78.509
MASS=ΔΔΔΔΔ7.600
```

if the printer were logically designated as Unit 4 and the following WRITE statement were used:

```
WRITE (4,6) AMASS
```

It can be seen that the 5 alphanumeric characters following H have been reproduced in the output <u>independent</u> of the List of the WRITE statement. If fact, it is possible to write either WRITE or READ statements (referring to FORMAT statements having <u>only</u> wH editing code entries) with <u>no</u> Lists.

For example:

13 FORMAT (41HΔTHISΔISΔANΔEXAMPLEΔOFΔALPHANUMERICΔDATA.)

WRITE (4, 13)

would produce: ΔTHISΔISΔANΔEXAMPLEΔOFΔALPHANUMERICΔDATA.

on the printed page, if the printer were logically designated as Unit 4.

The action of the wH editing code on input FORMATting permits a high degree of flexibility in annotative use of alphanumeric output.

To illustrate:

6 FORMAT (5HMASS=F10.3)

READ (3, 6) TIME

If the card reader were logically designated as Unit 3 and the first 15 columns of the card read were punched with:

TIME=000406.392

the value of the real variable TIME would be replaced by 406.392, TIME would replace MASS= and if the next statement executed were:

WRITE (4, 6) TIME

then the output produced on the printer would be

TIME=ΔΔΔ406.392

From this illustration, it can be seen that the w characters occurring on the input medium replace the w characters of a wH editing code when the FORMAT statement containing the wH editing code is used in conjunction with an input statement. Subsequent usage of FORMAT statement 6 with a WRITE statement will be as if FORMAT statement 6 were:

6 FORMAT (5HTIME=F10.3)

provided that FORMAT statement 6 is not used again with a READ statement. However, FORMAT statement 6 may be used as many subsequent times as desired with READ statements.

Combinations of the editing codes Aw and wH can be used to produce self-explanatory printed output.

For example:

```
13   FORMAT  (5HTIME = F10.3, A8)

     WRITE (4, 13)(TIME(I), (UNITS(I), I = 1, 3)
```

In this example, it is assumed that the FORTRAN program would be computing some TIME in SECONDS, MINUTES and HOURS. Recalling the required storage for alphanumeric data controlled by A-type editing code, it can be seen that if it is desired to print:

```
TIME =27360.000 SECONDS

TIME =   456.000 MINUTES

TIME =     7.600 HOURS
```

then the alphanumeric variable values SECONDS, MINUTES, HOURS must be internally stored in the double-precision name* array UNITS having the members:

```
UNITS (1) = SECONDS

UNITS (2) = MINUTES

UNITS (3) = HOURS
```

and the real variable TIME must have three forms of output desired internally stored in the real array named TIME, having the members:

```
TIME (1) = 27360.000

TIME (2) =   456.000

TIME (3) =     7.600
```

3. Skipped Fields

The editing control symbol X provides for ignoring characters of input or insertion of space characters in output. The editing code using this symbol is wX, where w is an unsigned integer constant specifying

---

* Double precision since SECONDS and MINUTES occupy more than 6 characters.

the number of characters to be skipped or the number of spaces to be inserted. The editing code wX is independent of the List of any input/output statement referencing the FORMAT statement in which it appears. No comma is required to separate the wX editing code from the subsequent editing code.

For example:

FORMAT (5XE4.2) could produce the output ΔΔΔΔΔ0.12

4. Repetition of Editing Codes

If it is desired to edit more than one element of a List in precisely the same manner and their corresponding editing codes would occur in succession in a Format Specification, then it is possible to shorten the Format Specification. The editing code to be repeated may be written only once, prefixed by an unsigned integer constant, n, indicating the number of times repetition is desired. It is not possible to repeat wH or wX editing codes by this method.

For example:

FORMAT (3I3, 3F6.3)

could produce the output: 439Δ45Δ35Δ1.234Δ2.678-5.876

5. Repetition of Groups of Editing Codes

If it is desired to edit more than one group of elements of a List in precisely the same manner and the corresponding groups of editing codes would occur in succession in a Format Specification, then it is possible to further shorten the Format Specification. The group of editing codes to be repeated may be written only once, enclosed in parentheses and prefixed by an unsigned integer constant, n, indicating the number of times repetition of the group is desired.

For example:

FORMAT (4(F12.1, E5.2) is equivalent to:

FORMAT (F12.1,E5.2,F12.1,E5.2,F12.1,E5.2,F12.1,
E 5.2)

6. Multiple Record Format Specification

If a group of editing codes is followed by a / (slash), then the remainder of the record being edited is ignored on input or filled with spaces on output and the editing codes following the slash (if any) are used to edit the next record. Thus, FORMAT (3F9.2, 2F10.4/ 8E14.5) could edit a two-line block of printing (two records), the first line of which would have its editing controlled by the codes 3F9.2, 2F10.4 and the second line of which would have its editing controlled by the code 8E14.5. Entire records may be ignored on input or blank (skipped) records may be produced on output simply by writing consecutive slashes.

For example:

```
4122  FORMAT (7HSPIN IS2E10.5, A12/6F7.2 /////)
```

used with an output statement directed to a printer would produce (editing of a line controlled by the FORMAT statement, 4122), a line whose editing was controlled by 7HSPIN IS2E10.5, A 12, a line whose editing was controlled by 6F7.2, and four blank (skipped) lines. If it is desired to skip n lines, then the Format Specification must contain n + 1 slashes.

7. Scale Factor Usage

To permit a more general use of F and E type conversion, a scale factor followed by the letter P may precede the specification. The scale factor for F type conversion is defined such that:

External Representation = Internal Representation X10 scale factor

For values of elements of lists whose editing is controlled by editing codes of the form scale factor PNEW.d, the output values are multiplied by 10 scale factor before output.

Thus using the original data of:

=.80654E+2 =.00686E0 .006636E+2

the statement FORMAT (1P3F12.3) could cause to be printed

ΔΔΔΔΔΔ=806.540ΔΔΔΔΔΔ-0.069ΔΔΔΔΔΔΔ6.636

A positive scale factor may be used with E type conversion to increase the number and decrease the exponent.

For values of element lists whose editing is controlled by editing codes of the form scale factor PNEW.d, the mantissas are multiplied by 10 scale factor, and the exponents are decreased by the scale factor.

Thus using the above data the statement FORMAT (1P3E12.2) could cause to be printed

-806.540E-01ΔΔ-6.860E-03ΔΔΔ6.636E-01

Once a value has been given for a scale factor within a FORMAT specification, it will be applied to all values edited by succeeding E- and F-type editing codes within the same Format Specification. This is true for both single-record and multiple-record Format Specifications. To terminate the effect of a previously given scale factor within a specific Format Specification, a subsequent scale factor of 0 must be given. Scale factors have no effect on any other editing save that which is controlled by E- and F- editing codes.

8. Object Time Introduction of Format Specification

Either of the input/output statements which require FORMAT statement reference may have an array name in the position where a FORMAT statement number would normally appear. This makes it possible to introduce Format Specifications at object time. The array name used in such an input/output statement must have been included in the list of a previously executed input statement. It must also have previously had been declared in a DIMENSION statement. (See Section V.) The Format Specification must be present on the input medium in exactly the form as has been previously defined, i.e.,

(Format Specification)

9. The Format Specification Scan

During the execution of an input/output statement, the associated Format Specification is scanned by the FORTRAN processor from left to right. If, during the course of this scan, the end of the Format Specification is reached before all List elements have been edited, the scan is reinstated beginning with the character of the Format Specification following the last preceding left parenthesis.

For example; suppose it is desired to print the first line of a page as header information and all remaining lines according to another Format Specification; and the number of lines to be printed on a given page were dependent upon the value of some internally computed variable. The output of the illustrative example of Section II could have been accomplished in the following manner:

```
 10  FORMAT(9X9HITERATION5X5HANGLE9X6HCHANGE/

     (13XI1, F15.5, F14.5))
     .
     .
     .
110  WRITE (2, 10)  (K, ANGLE (K), CHANGE (K), K = 1, I)
```

D. READ

The READ statement allows the programmer to introduce data into the computer from an input medium. There are two general forms of the READ statement.

The form for an edited READ statement is:

READ (Unit, Format) List

The form for an unedited READ statement is:

READ (Unit) List

In both these forms Unit is the logical reference which, by means of an input/output table, specifies the input medium . It may be written as either an integer constant or a nonsubscripted integer variable. List is an input/output List.

Format, which appears only in the edited READ statement form, is the statement number of a FORMAT statement or an array name specifying an object-time introduced Format Specification.

The unedited READ statement is used where work tapes or other internal storage extensions are desired. Data may be read into the computer from such internal storage extensions without specifying any format provided that it is in the format normally produced by the unedited WRITE statement. (See E below)

In previous versions of the FORTRAN language, the statements:

READ Format, List

READ INPUT TAPE t, Format,List

where t is either an unsigned integer constant or an integer variable specifying the logical tape transport number from which reading is to take place, have been used. Either of these two statements are acceptable alternates to READ (Unit, Format) List. Similarly, the statement READ TAPE t, List is an earlier form of READ (Unit) List and is also an acceptable alternate.

E. WRITE

The WRITE statement allows the programmer to produce output data on external media from within the computer. There are two general forms of the WRITE statement. The form for an edited WRITE statement is:

WRITE (Unit, Format) List

The form for an unedited WRITE statement is:

WRITE (Unit) List

In both these forms Unit is the logical reference which, by means of an input/output table, specifies the output medium. It may be written as either an integer constant or a nonsubscripted integer variable. List is an input/output List. Format, which appears only in the edited WRITE statement form is the statement number of a FORMAT statement or an array name specifying an object-time introduced Format Specification. The unedited WRITE statement is used where work tapes or other internal storage extensions are desired. Data may be written from the computer on to such internal storage extensions without specifying any format.

In previous versions of the FORTRAN language, the statements:

PRINT Format, List

PUNCH Format, List

WRITE OUTPUT TAPE t, Format, List

where t is either an unsigned integer constant or an integer variable specifying the logical tape transport number upon which writing is to take place, have been used. Any of these three statements are acceptable alternates to WRITE (Unit, Format) List. Similarly, the statement

WRITE TAPE t, List

is an earlier form of WRITE (Unit) List and is also an acceptable alternate.

F. Magnetic Tape-Positioning Statements

There are three magnetic tape-positioning statements available in the FORTRAN language. They are:

REWIND Unit

BACKSPACE Unit

END FILE Unit

where unit is as before. The execution of a REWIND statement causes the tape mounted on a UNISERVO specified to be rewound. The BACKSPACE statement causes the magnetic tape mounted on the specified UNISERVO to be positioned at the beginning of the logical record last written or read. The END FILE statement causes an end-of-file indication to be written on the tape mounted on the specified UNISERVO.

G. Carriage Control for Printed Output.

The carriage control characters are:

Blank = single space

0 = double space

1 = skip to top of page

+ = suppress spacing

Example:

```
10 FORMAT (25H1 TITLE OF PROGRAM IS SORT)
```

The printer will skip to the top of a new page and print.

TITLE OF PROGRAM IS SORT

in print position 2 thru 25.

5. SPECIFICATION, DATA AND END STATEMENTS

A. Specification Statements

Specification statements provide the processor with information which describes the manner in which data is to be stored in the computer. These statements are considered to be non-executable because they do not result in instructions in the object program. There are four kinds of specification statements: DIMENSION, COMMON, EQUIVALENCE, and FORMAT. (The FORMAT statement has been previously discussed in Section IV and will not be discussed in this section.)

In some versions of FORTRAN, a fifth specification statement is used, FREQUENCY. In UNIVAC 1107 FORTRAN and other current versions, this statement is not used and will be ignored if it appears in a FORTRAN program.

1. DIMENSION

The DIMENSION statement specifies the maximum size of each array in a FORTRAN program. It provides the programmer with a simple means of allocating storage for arrays. The DIMENSION statement must occur before any executable statement of a FORTRAN program or subprogram.

The form of the DIMENSION statement is:

DIMENSION array 1 (parameters), array 2 (parameters),..., array n (parameters)

where array i indicates the name of an array, followed by parentheses which enclose the parameters specifying the maximum value each subscript may take during the execution of the object program. The dimensioning parameters must be unsigned integer constants or unsigned integer variables. (See discussion of Adjustable Dimensions in Section VII.) If more than one parameter is given, the parameters must be separated by commas. If more than one array is named in a DIMENSION statement, the name of each subsequent array must be preceded by a comma. All array names used in a FORTRAN program must have their dimensions declared in that program prior to their appearance in executable statements.

For example:

```
DIMENSION VOLUME (2, 3)

DIMENSION WEIGHT (10), AREA (5,10), SIZE (2, 4, 6)

DIMENSION A(1, 2, 3, 4, 5, 6, 7)
```

In the first example, the DIMENSION statement causes the allocation of storage for the two-dimensional real array VOLUME. Storage is provided for up to six items of data. The maximum values with which VOLUME may be subscripted are 2 and 3, respectively.

2. COMMON

It is possible to economize on data storage among separately compiled portions of a FORTRAN program through the use of the COMMON statement. During the execution of a FORTRAN program, while different variables may be required at different times by different separately compiled portions, it is not necessarily true that all such variables must occupy distinct storage locations. The COMMON statement enables such variables to share storage locations.

The form of the COMMON statement is:

```
COMMON /Block name/ Variable names/Block name / Variable names/...
```

where Block name is from one to six alphanumeric characters, the first of which is alphabetic. Block name bears no relationship whatsoever to any variable name which may occur elsewhere in the same or any other FORTRAN program.

The Variable names occurring immediately following the slash after a Block name have their values stored in the COMMON block having that name. Any given list of Variable names occurring in a common statement is terminated by the slash preceding a new Block name or by the end of the COMMON statement. Should a list of Variable names be so long that it is not possible to contain it in one COMMON statement, the list may be continued by using another COMMON statement in which the Block name is repeated and followed by the continuation of the list of Variable names. Each variable name must be separated from other Variable names in the same list by commas.

If anywhere in a COMMON statement, a Block name is omitted from between the slashes where it would normally appear, the block defined by the Variable name list following is given a "Blank" name. Such a block

is called a "blank COMMON" block. Since there is no advantage to the use of such "blank COMMON" blocks, the use of these is discouraged. They are provided in UNIVAC 1107 FORTRAN only for compatability with earlier version of the FORTRAN language.

All COMMON statements appearing in any single separately compiled portion of a FORTRAN program having the same Block names will cause the extension of the size of those COMMON blocks. It is stressed that the COMMON statement is a means for economizing on storage among separately compiled portions of a FORTRAN program. (See EQUIVALENCE below.) The size of a COMMON block is the sum of the storage required for each member whose name appears in the Variable name list associated with that block. (Also see COMMON and EQUIVALENCE below.)

The assignment of storage to members of the variable name lists of COMMON statements is determined solely from consideration of COMMON and DIMENSION statements. Elements declared to be members of a given COMMON block within any single separately compiled portion of a FORTRAN program are always assigned unique storage locations, contiguous in the order declared. COMMON blocks having the same name and different variable name lists, each of which appears in a COMMON statement in more than one separately compiled portion of a FORTRAN program, must be of the same size.

The appearance of identically named COMMON blocks in COMMON statements in more than one separately compiled portion of a FORTRAN program is sufficient to insure the sharing of storage within the computer for the values of the list of variable names that follow the COMMON block name.

COMMON and DIMENSION

An array name which is to be assigned to a COMMON block may have its dimensions declared in a DIMENSION statement, or in the COMMON statement in which it appears. For example:

```
COMMON / DBLOCK / TIME, MASS, SPACE (14, 14, 14)
```

The dimensions of the array SPACE have here been given in a COMMON statement and, therefore, need not occur in a DIMENSION statement. However, this COMMON statement must appear in the FORTRAN program prior to the appearance of any member of the array SPACE in an executable statement.

COMMON AND EQUIVALENCE

The sole effect of EQUIVALENCE Statements upon COMMON assignment may be the lengthening of a COMMON Block (labeled or blank). Furthermore, the only such lengthening permitted is that which extends a COMMON Block beyond the last assignment for that block made directly by COMMON Statements.

Indeed, if EQUIVALENCE Statements have the effect of relating variables or arrays declared in COMMON Statements, the relationships thus established is either redundant or incorrect.

3. EQUIVALENCE

Within any separately compiled portion of a FORTRAN program, it is possible to economize on storage locations through the use of the EQUIVALENCE statement. During the execution of a separately compiled portion of a FORTRAN program, while different variables may be required at different times, it is not necessary that they occupy distinct storage locations. The EQUIVALENCE statement enables such variables to share storage locations.

The form of the EQUIVALENCE statement is:

EQUIVALENCE (Variable names), (Variable names),...

Enclosed within each pair of parentheses are the names of the variables whose values are to share the same storage locations. The variable names are separated by commas and each pair of parentheses are also separated by commas.

The FORTRAN programmer cautioned that good practice dictates that only variables of the same mode be made equivalent. This is recommended because of the ease of error associated with making equivalent variables of different modes.

Arrays or a particular element in an array may be used in the Variable name list of EQUIVALENCE statements. The programmer is cautioned that if an array name is used in an EQUIVALENCE statement variable name list without referring to a specific numbered element of that array, then it will be treated as a reference to the first member of that array. This is different from any other FORTRAN usage of the name of an array.

For example:

```
DIMENSION ALPHA (5), BETA (6), GAMMA (5,7)

DELTA = 39.6

EQUIVALENCE (ALPHA, GAMMA (1,6), DELTA, BETA)
```

The storage allocation for the members of the three arrays and the non-subscripted variable that share storage will be:

Storage Location	Values of Variables Stored			
L	GAMMA (1,1)			
	.			
	.			
	.			
L + 25	GAMMA (1, 6)	ALPHA (1)	BETA (1)	DELTA
L + 26	GAMMA (2, 6)	ALPHA (2)	BETA (2)	
L + 27	GAMMA (3, 6)	ALPHA (3)	BETA (3)	
L + 28	GAMMA (4, 6)	ALPHA (4)	BETA (4)	
L + 29	GAMMA (5, 6)	ALPHA (5)	BETA (5)	
L + 30	GAMMA (1, 7)		BETA (6)	
L + 31	GAMMA (2, 7)			
	.			
	.			
	.			
L + 34	GAMMA (5,7)			

Note that if one element of an array is equivalenced to one element of another array, the entire arrays are equivalenced as in the above example.

Note, also, that double-precision and complex variables require special consideration, since they each occupy two computer words. A double-precision quantity is stored in two adjacent words with the most significant part in the first word. Complex quantities are, likewise, stored in two adjacent words with the real part occupying the first word. References to complex or double-precision variables in EQUIVALENCE Statements are always references to the first word of the corresponding word pair.

B. DATA Statements

1. DATA

The DATA statement enables the FORTRAN programmer to cause data to be produced internally by the FORTRAN processor at the time of initial loading of the object program. It is especially useful in the creation of tables or in the initialization of arrays. The form of the DATA statement is:

DATA List /Literal list/, List /Literal list/, ...

where List is defined as in Section IV, B. List, with the following exceptions:

a. An array name may not appear without explicit subscripting.

b. The indexing parameters which appear in DO-type Lists must be integer constants.

c. If a subscripted variable appears not under the control of a DO-type List, then its subscripts must be integer constants.

The Literal list may consist of any combination of the following forms. Each distinct form must be separated from the other members of the list by commas.

a. Integer constants.

b. Real constants.

c. Double-precision constants.

d. Complex constants.

e. Alphanumeric characters. They are written using the form "nH", followed by n alphanumeric characters, where spaces are counted among the n characters. The n characters are apportioned among the indicated variables as follows: If a variable is integer, logical, or real it will receive six characters. If after these six have been assigned, there remain other characters in the string, they are assigned to the next variable(s) in line. Should six characters not remain in the string when a variable is to be assigned its characters, those available will be placed in the variable, left-justified and the remainder of the computer word(s) filled out with spaces.

It is possible to repeat literals without explictly rewriting them by indicating the number of times the literal is to be repeated followed by an asterisk. The literal preceding the first slash does not require comma separation from the slash.

The DATA statement must be preceded by all COMMON, DIMENSION and Type statements that relate to the variable names in the DATA statement's Lists.

Examples:

```
DATA  DICK,  BOB ,  DON /11HEND PROGRAM, 7.8/
```

The real variable DICK will contain END PR, the real variable BOB will receive OGRAM followed by one space to fill the remainder of its computer storage and DON will receive the real number 7.8 .

```
DATA (CARGO (I), WEIGHT (I), I = 1,10,2) /0.0, 0.0,8*100.0/
```

The memory locations will appear as follows:

CARGO (1)	0.0	WEIGHT (1)	0.0
CARGO (3)	100.0	WEIGHT (3)	100.0
CARGO (5)	100.0	WEIGHT (5)	100.0
CARGO (7)	100.0	WEIGHT (7)	100.0
CARGO (9)	100.0	WEIGHT (9)	100.0

2. BLOCK DATA

It is very useful to compile data into named COMMON blocks. However, the compilation of data into named COMMON blocks must be done in a separate subprogram, an independently compiled portion of a FORTRAN program. This subprogram must have a very restrictive form. There is provided a special form of FORTRAN statement to denote such a subprogram. This form is:

```
BLOCK DATA
```

There are only 11 types of statements which may occur in a BLOCK DATA subprogram:

1. BLOCK DATA
2. COMMON
3. DIMENSION
4. INTEGER
5. REAL
6. COMPLEX
7. DOUBLE PRECISION
8. LOGICAL
9. PARAMETER
10. DATA
11. END

All of these statements are non-executable statements. All elements appearing in a COMMON block whose name is used in a COMMON statement in a BLOCK DATA subprogram must appear in the COMMON statement in the BLOCK DATA subprogram even though this is the only appearance of such variable names in the BLOCK DATA subprogram.

These variable names and COMMON block names will be ignored by the BLOCK DATA subprogram; however their inclusion provides necessary information at object time.

For example:

```
BLOCK DATA
COMMON / BLOK 3 /P I, E, TWO, ALPHA/BLOK4/SUN, SEA, PVE
DIMENSION ALPHA (2,3)
INTEGER TWO
DATA (ALPHA (I), I = 1,6) /800.0, 485.0,2* 481.0, 470.5, 466.0/
      PI /3.14159/, E /2.7183 /, TWO /2 /
END
```

C. END

The END statement is used to indicate the end of a given compilation to the FORTRAN processor. The form of the END statement is:

END

The END statement must be the last physical statement of every FORTRAN compilation. The END statement is not executed.

6. TYPE STATEMENTS

A. GENERAL

Type Statements are used to declare the types (modes) of variables, arrays, and functions as integer, real (single-precision floating-point), double-precision (double-precision floating-point), complex, or logical. In addition, they may specify the dimensions of an array (as in a COMMON or DIMENSION statement). The subroutines and external functions whose names appear as arguments of subroutine and function calls must be declared as such.

In the absence of a Type Statement for a variable, array, or function, the corresponding name will be of the Type INTEGER (i.e., a fixed-point quantity) if the first character of the name is one of the letters I, J, K, L, M, or N; otherwise, it will be of the Type REAL (single-precision floating-point). This convention will be known as the alphabetic naming convention.

The general form of the statements will be

INTEGER a, b, c, ...

REAL a, b, c, ...

DOUBLE PRECISION a, b, c, ...

COMPLEX a, b, c, ...

LOGICAL a, b, c, ...

EXTERNAL e, f, g, ...

ABNORMAL e, f, g, ...

where a, b, c may be subscripted (to indicate its dimensions) or unsubscripted, and e, f, and g are Function or Subroutine names.

B. TYPE STATEMENT RULES

1. If a variable or function name occurs in a Type Statement the Type Statement must precede all other references to the name.

2. A variable name may appear in at most one Type Statement and that must be one of the following:

INTEGER

REAL

DOUBLE PRECISION

COMPLEX

LOGICAL

A function name may also appear in at most any one of these Type Statements, but in addition, it may appear in an ABNORMAL Type Statement, and/or it may appear in an EXTERNAL Type Statement (See Section VII).

3. The occurrence of a variable or function name in an INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL Type Statement overrides the alphabetic naming convention for that name and declares the variable or function to be of the corresponding type.

4. The occurrence of a function or subroutine name in an EXTERNAL Statement indicates that that name is the name of an external function or subroutine, and is to be used (without arguments of its own) as an argument of a subroutine CALL or function reference.

7. FUNCTIONS, SUBPROGRAMS AND SUBROUTINES

A. GENERAL

Functions and subprogram statements allow the programmer to make use of previously written programs. These statements permit the programmer to call upon such programs during the execution of his main program. In this way, the programmer can cause a complex procedure to occur without specifying each statement of the procedure every time the procedure is to occur. Many mathematical routines are maintained in the form of a library of subroutines. These subroutines are available to the programmer whenever he may desire to utilize them in his FORTRAN program. The subroutine library lends itself to ease of programming and reduction of redundant programming effort.

B. FUNCTIONS

In mathematics, if the value of one quantity is dependent upon the value or values of several other quantities, then it is said to be a function of the others. The first quantity is called the function and the other quantities are called the arguments.

Examples:*

SIN (X) is an example of a function called Sine, whose argument is X.

Square Root (A + B) is a function called square root, whose argument is the expression A + B.

A function may have more than one argument. For example, maximum (J, K) is a function which selects the larger of a pair of arguments, i.e. the larger of two variables, J and K.

Maximum (4, 1) = 4
Maximum (4, 5) = 5

This function has two arguments.

---

* These are mathematical functions, not FORTRAN functions.

There are four kinds of functions available in the FORTRAN language.

a. External Functions
b. Statement Functions
c. Built-in Functions
d. Internal Functions

A reference to a function is a request to a computational procedure for the production of a single value, identified by and associated with the function name. All four kinds of FORTRAN functions conform to the alphabetic naming conventions which may be overridden by the various type statements. (See Section VI.)

A function name (with its arguments) may be used anywhere in a FORTRAN expression where a variable name might be used. The computational procedure referred to above; which, in essence, defines a function in FORTRAN, may either be defined independently of the program containing the function references or it may be defined internally to that program or it may be among the procedures available to the processor internally. The preceding sentence distinguishes external functions, statement functions and built-in functions, respectively. Each of the kinds of functions differ to some extent in their forms. Generally, however, the form of a function reference is:

$$f(a_1, a_2, \ldots, a_n)$$

where f is the function name and a_i are the n arguments.

Every function reference must contain at least one argument. If f is a reference to an internal or external function then any of the arguments a_i may refer to a statement number. The form of a_i in this case must be:

$$\$n$$

where n is the statement number. A return to the i^{th} argument will be a reference to the statement labeled n.

1. External Functions

There are two types of external functions, library functions and function subprograms.

The discussion of the function subprograms will be deferred to the discussion of subprograms.

Library functions are prewritten and exist in the UNIVAC 1107 FORTRAN library.

The form of a library function is:

f (a)

where f is the name of the library function and a is the argument(s). a may be any arithmetic expression, variable name or constant. Their names should not be used in any manner except to refer to the library functions. The table on the following page lists and defines forty-one library functions available to the UNIVAC 1107 FORTRAN. (See Table Section VII, p. 4)

2. Statement Functions

This kind of function is created by the programmer. Therefore, several cautions are given at the outset. Function names must be unique. They must not be the same as any variable name appearing elsewhere in the FORTRAN program in which they appear. A function whose value is logical must be declared LOGICAL.

Statement functions are defined by a single arithmetic or logical statement in the source program and apply only to the particular program or subprogram in which their definition appears. They have the form:

$$f(a_1, a_2, \ldots a_n) = e$$

where f is a function name; a_i are distinct, nonsubscripted variable names (formal arguments), and e is either an arithmetic expression if the mode of f is arithmetic, or a logical expression if the mode of f is logical. Normally, the formal arguments will appear in the expression e. Upon execution of the statement function, they are replaced by the actual arguments of the function reference. They may each be of any mode. The actual arguments of a given reference must agree in number and mode with their corresponding formal arguments. It is not necessary that the formal argument names be distinct from any other FORTRAN names appearing in any other FORTRAN statement. The expression e may have subscripted variables appearing in it. However, the dummy arguments may not be subscripted. The expression e may also contain references to previously defined statement functions, built-in functions or external functions*. All statement functions of a given FORTRAN program must precede the first executable statement of that program.

---

* References to Functions in the function statement may not contain statement labels as arguments (i.e. F ($5)).

FORTRAN Name	No. of Args.	Function and Definition		Mode of Argument	Mode of Function
SIN	1	Trigonometric Sine:	SIN (X)	Real	Real
DSIN			DSIN (X)	D-P	D-P
CSIN			CSIN (X)	Complex	Complex
COS	1	Trigonometric Cosine:	COS (X)	Real	Real
DCOS			DCOS (X)	D-P	D-P
CCOS			CCOS (X)	Complex	Complex
TAN	1	Trigonometric Tangent:	TAN (X)	Real	Real
DTAN			DTAN (X)	D-P	D-P
CTAN			CTAN (X)	Complex	Complex
ASIN	1	Trigonometric Arcsine:	ASIN (X)	Real	Real
DASIN			DASIN (X)	D-P	D-P
ACOS	1	Trigonometric Arccosine:	ACOS (X)	Real	Real
DACOS			DACOS (X)	D-P	D-P
ATAN	1	Triginometric Arctangent:	ATAN (X)	Real	Real
DATAN			DATAN (X)	D-P	D-P
ATAN2	2		ATAN(X_1, X_2)	Real	Real
DATAN2	2		DATAN (X_1, X_2)	D-P	D-P
SINH	1	Hyperbolic Sine:	SINH (X)	Real	Real
DSINH			DSINH (X)	D-P	D-P
CSINH			CSINH (X)	Complex	Complex
COSH	1	Hyperbolic Cosine:	COSH (X)	Real	Real
DCOSH			DCOSH (X)	D-P	D-P
CCOSH			CCOSH (X)	Complex	Complex
TANH	1	Hyperbolic Tangent	TANH (X)	Real	Real
DTANH			DTANH (X)	D-P	D-P
CTANH			CTANH (X)	Complex	Complex
EXP	1	Exponential (e^X):	EXP (X)	Real	Real
DEXP			DEXP (X)	D-P	D-P
CEXP			CEXP (X)	Complex	Complex
ALOG	1	Natural Logarithm ($LOG_e x$):	ALOG (X)	Real	Real
DLOG			DLOG (X)	D-P	D-P
CLOG			CLOG (X)	Complex	Complex
ALOG10	1	Common Logarithm ($LOG_{10} x$):	ALOG10 (X)	Real	Real
DLOG10			DLOG10 (X)	D-P	D-P
SQRT	1	Square Root $(X)^{1/2}$:	SQRT (X)	Real	Real
DSQRT			DSQRT (X)	D-P	D-P
CSQRT			CSQRT (X)	Complex	Complex
CBRT	1	Cube Root $(X)^{1/3}$:	CBRT (X)	Real	Real
DCBRT			DCBRT (X)	D-P	D-P
CCBRT			CCBRT (X)	Complex	Complex
CABS	1	Determine the absolute value of the argument: CABS (X) For $X = a + ib$ $X = (a^2 + b^2)^{1/2}$		Complex	Real
DMOD	2	Remainder: subtract from the first argument the appropriate integer multiple of the second argument so that the remainder is less than the second argument but non-negative. DMOD (X_1, X_2)		D-P	D-P

Note: If the result of the Function is Double-Precision or Complex, the Function name must be declared in a Type statement.

3. Built-In Functions

These functions are integral-internal parts of the FORTRAN processor. They are available to the programmer and are incorporated into the object program as open subroutines (in-line object code) each time they are referred to by the source program.

The form of a Built-In Function is:

$$f(a) \text{ or } f(a, a_2, \ldots, a_n)$$

where f is the name of the Built-In Function and a_n are the arguments. a_n may be any arithmetic expression, variable name or constant.

The table on the following page lists and defines the thirty-one Built-In Functions of the UNIVAC 1107 FORTRAN processor.

FORTRAN Name	No. of Args.	Function	Mode of Argument	Mode of Function
ABS	1	Determine the absolute value of the argument.	Real	Real
IABS			Integer	Integer
DABS			D-P	D-P
AINT	1	Truncate; eliminate the fractional portion of the argument.	Real	Real
INT			Real	Integer
DINT			D-P	D-P
AMOD	2	Remainder; subtract from the first argument the appropriate integer multiple of the second argument so that the remainder is less than the second argument but non-negative.	Real	Real
MOD			Integer	Integer
AMAX0	≥ 2	Select the largest value.	Integer	Real
AMAX1			Real	Real
MAX0			Integer	Integer
MAX1			Real	Integer
DMAX1			D-P	D-P
AMIN0	≥ 2	Select the smallest value.	Integer	Real
AMIN1			Real	Real
MIN0			Integer	Integer
MIN1			Real	Integer
DMIN1			D-P	D-P
FLOAT	1	Convert from integer to real.	Integer	Real
IFIX	1	Convert from real to integer.	Real	Integer
DBLE	1	Convert from real to double-precision.	Real	D-P
CMPLX	2	Convert two real arguments to one complex number.	Real	Complex
SIGN	2	Replace the algebraic sign of the first argument by that of the second.	Real	Real
ISIGN			Integer	Integer
DSIGN			D-P	D-P
DIM	2	Positive difference; subtract the smaller of the two arguments from the first argument.	Real	Real
IDIM			Integer	Integer
SNGL	1	Obtain the most significant part of a double-precision argument.	D-P	Real
REAL	1	Obtain the real part of a complex argument.	Complex	Real
AIMAG	1	Obtain the imaginary part of a complex argument.	Complex	Real
CONJG	1	Obtain the conjugate of a complex argument.	Complex	Complex

4. ABNORMAL FUNCTIONS

The definition of a Normal Function is an External Function which has all of the following properties:

a. It has no implied (COMMON) arguments.

b. None of its arguments are function outputs. Its single output is through its name.

c. If a function call with a given set of arguments produces a given result, then all references to the function with all arguments identical with the given set will give the identical result. This excludes functions which, for example, have local variables whose values are saved from reference to reference.

d. The function contains no I/O statements.

An ABNORMAL Function is one which fails to possess one or more of the properties listed above. No attempt will be made to optimize the computation of common subexpressions which contain references to ABNORMAL Functions.

The following rules will apply:

a. If an ABNORMAL statement does not occur in a program, then all function references are considered to be ABNORMAL.

b. If an ABNORMAL statement occurs, then all functions so declared will be assumed to be ABNORMAL, and all other functions will be assumed to be Normal.

c. If one and only one ABNORMAL Statement occurs, and that statement lists no function names, then all functions are treated as Normal.

For example:

```
      .
      .
      .
1  ABNORMAL  F
      .
      .
      .
2  A  =  F(X) + F(X)
      .
      .
      .
3  A  =  2 * F(X)
      .
      .
      .
```

In the above example, F is defined in statement number 1 as an ABNORMAL Function. This function might contain statements that rewind or backspace magnetic tape, set hardware indicators, etc. If F(X) backspaces magnetic tape one record, then in statement number 2, the magnetic tape unit would be backspaced two records, and the output value of F would be doubled and replace the value in A. In statement number 3, only one record on the magnetic tape unit is backspaced, and the output value of F would be doubled, and replace the value in A. Therefore, these statements do not necessarily have the same result. If F(X) were normal the results of statements 2 and 3 would have been identical.

C. SUBPROGRAMS

Within the FORTRAN system, it is possible to define subprograms which may be called upon by other FORTRAN programs. Generally, they are FORTRAN source language programs that cannot be defined by a single statement and are not commonly enough used to warrant their inclusion in a library. Subprograms must be independently compiled and incorporated into other FORTRAN programs for execution. They are not independently executable as a general rule. There are three types of subprograms; BLOCK DATA, FUNCTION and SUBROUTINE subprograms.

BLOCK DATA subprograms have been previously discussed in Section V and are included here only for completeness of classification.

FUNCTION subprograms and SUBROUTINE subprograms are discussed below.

1. FUNCTION Subprograms

The first statement of a FUNCTION subprogram must be of the form

type FUNCTION f (a_1, a_2, ..., a_n)

where type is either REAL, INTEGER, LOGICAL, DOUBLE-PRECISION, COMPLEX, or absent, f is a function name and a_i are the formal arguments. a_i may be array names, non-subscripted variable names, or the name of other FUNCTION Subprograms. Each name must be distinct. Any a_i may be the character, $. When the $ character is used as the i^{th} parameter, the i^{th} parameter is never referenced in the function f except in the statement:

RETURN k

If type is absent then the mode of f is determined by a type statement or by the alphabetic naming conventions. If an a_i, a formal argument, is an array name, then that array name must appear in a DIMENSION statement of the subprogram in which it occurs prior to any reference by an executable statement or by a statement function definition. The DIMENSION statement may only in this case of subprogram usage with formal subscripted arguments, assume a special form. Normally, the only entries that may be made in the subscript positions of an array named in a DIMENSION statement are integer constants, and these integer constants specify the maximum dimensions of the named array. It is

possible in FUNCTION or SUBROUTINE subprograms to have maximum dimensions of arrays appear as nonsubscripted integer variables in DIMENSION statements. In this case, the array name and all of the variable subscript names must be listed as formal arguments of the subprogram. This facility enables the subprogram to be made object time variable in its requirements for data storage. The program which contains reference to such a subprogram must also contain a DIMENSION statement which specifies the actual dimensions of such arrays. The nonsubscripted integer variables which appear in such a subprogram as those formal arguments specifying the maximum dimensions of an array can neither appear on the left of an arithmetic or logical statement, nor can they be changed in value in any other way by the subprogram.

All the formal argument names must occur in at least one executable statement in the subprogram. Any actual argument for a FUNCTION subprogram reference may be either an arithmetic expression, a logical expression, an array name or another function or subroutine. which also must appear in an EXTERNAL statement in the referencing program.

The name of a function itself must not appear in a DIMENSION, or statement in the subprogram. However, the name must appear at least once on the left side of a logical or an arithmetic statement, as an element of an input list or as an argument or a subroutine call. (See RETURN below.)

The following are permissible arguments of a FUNCTION Subprogram reference:

a. Any arithmetic expression.

b. Any logical expression.

c. An array name.

d. The name of another function or subroutine which also must appear in an EXTERNAL Statement in the calling program unless the name appears as an explicite reference elsewhere in the program.

e. A statement number preceded by the character $.

f. The form:

nHhhh. . .h

where n is any unsigned integer, and hhhh...h is a string of any n alphanumeric or special characters including "blank".

2. RETURN

A subprogram will normally contain at least one RETURN Statement. Each such statement marks the logical end of flow for the subprogram and causes, in the case of a FUNCTION Subprogram, a return to the statement in which function reference was imbedded, or in the case of a subroutine, a return to the first Executable Statement following the CALL Statement. It will have the general form:

RETURN

A second form of a RETURN statement is:

RETURN k

where k is an integer constant or integer variable. If this form was used, the value of the integer, i , specifies that the return will be made to the i^{th} argument of the referencing statement, (presumably the i^{th} argument was a statement number preceded by $).

If a RETURN statement is not specified then the END statement will serve as a RETURN statement.

A third form of a RETURN statement is:

RETURN 0

where 0 is an integer constant, 0. The execution of the RETURN O statement will result in a transfer to the system error program.

D. SUBROUTINE SUBPROGRAM

A SUBROUTINE subprogram has so many points of similarity to a FUNCTION subprogram that the treatment here will consist of citing how it differs from a FUNCTION subprogram.

The major difference is that a SUBROUTINE subprogram may produce many values from one reference whereas a FUNCTION subprogram may produce one.

The SUBROUTINE subprogram returns values, if any, only through its arguments (by causing values to be stored in the locations associated with its actual arguments as specified in its CALL statement) or through variables in COMMON blocks. No value is associated with the name of a SUBROUTINE subprogram.

The form of a SUBROUTINE subprogram statement is either:

SUBROUTINE s(a_1, a_2, . . . , a_n)

or

SUBROUTINE s

where s is a subroutine name and a_i are the n arguments. a_i may be any arithmetic or logical expression, variable name or constant.

Reference to a SUBROUTINE subprogram, is made through the use of a special statement which has no other use. This statement is the CALL statement and has the form

CALL s (a_1, a_2, . . . , a_n)

or

CALL s

where s is the subroutine name and a_i are the <u>actual</u> arguments. a_i may take on all forms described for FUNCTION subprogram references.

E. INTERNAL FUNCTION AND INTERNAL SUBROUTINE SUBPROGRAMS

Internal subprograms are compiled in conjunction with a **Main** Program, a SUBROUTINE Subprogram, or a FUNCTION Subprogram. In the latter two cases, the first line of compilation must be a FUNCTION or SUBROUTINE statement.

Subprograms are indicated as internal by either of the following subprogram statements:

1. SUBROUTINE S $(a_1, a_2, \ldots, a_n)$
2. FUNCTION F $(a_1, a_2, \ldots a_n)$

where S and F are the subprogram names and a_i are the formal parameters. The same rules apply to the subprogram names and the formal parameters as in the previously defined subprograms.

Internal subprograms are referenced in the same manner as external subprograms; i.e., an internal SUBROUTINE is referenced by a CALL statement; an internal FUNCTION is referenced as other functions are referenced. However, such references and calls must be made within the main program or other internal subprograms contained in the main program.

The compiler will assume that all statements appearing between two internal subprogram statements belong to the first internal subprogram. The occurence of the Internal FUNCTION or SUBROUTINE Subprogram statements will mark the end of the previous program. The END statement will mark the end of the entire set of programs.

Identifiers (names) which are available to the entire set of programs are referred to as "global". The identifiers which are available only to a particular internal subprogram will be referred to as "Local".

The following are Global identifiers:

1. COMMON Block names
2. Names of Internal Subprograms
3. All variable names or Function names in the program appearing before a FUNCTION or SUBROUTINE statement which is not the first statement of a program
4. The names of Built-In Functions

The following are Local identifiers:

1. Dummy arguments of the internal subprogram
2. labels, Formats, and Statement Functions
3. Variables in Type and DIMENSION statements
4. Variables in COMMON statements

Note:

1. Any variable which is Local but has the same name as a Global variable must be declared as indicated in statements 3 and/or 4 of the list of Local identifiers before it is referenced.

2. The Equivalence statement will not force a variable to be Local.

Example:

```
 1    DIMENSION D(50), B(I)
            .
            .
            .
      A = C + D(I)
      GO TO 10
            .
            .
            .
10    CALL Z (A, 20)
      X = A + Y (R+S, $ 6, U)
            .
            .
            .
 6    .......
            .
            .
            .
      SUBROUTINE Z (W, V)
      DIMENSION D(25)
10    ......
            .
            .
            .
      CALL Q (D(1), $ 12, D(2))
            .
            .
            .
```

```
11   GO TO 10
       .
       .
       .
12   V = D (J)/B(J)
       .
       .
       .
     SUBROUTINE Q (X, $, T)
       .
       .
       .
     RETURN 2
       .
       .
       .
     CALL Z (E, F)
       .
       .
       .
     END
```

The SUBROUTINE Statement, SUBROUTINE Z (W, V) indicates the end of the first program. Subroutine Z is terminated by the SUBROUTINE Statement, SUBROUTINE Q (X, $, T) and SUBROUTINE Q is terminated by the statement RETURN 2 or the END Statement. The END statement also terminates the end of the entire set of programs. The global variables for this program are A, B, C, D, Z, Y, R, S, and U. The local variables in the internal SUBROUTINE Z are W, V, and D. Note that D is also a global variable. The local variable D is a unique variable and not the same as the global variable D, even though they have the same name. The only global variable used in SUBROUTINE Z is the variable B. The statement number 10 is local to SUBROUTINE Z and different from the global statement number 10 in the first program.

The internal subroutine, SUBROUTINE Q (X, $, T) has the local variable X, T, E, and F. If the RETURN 2 statement is executed, it will return control to the statement number that is inserted in the second dummy argument. If the END statement is used as the return statement of the internal subroutine. control is transferred to the next executable statement of the referencing program.

APPENDIX 1. SORT SUBROUTINE

This is a subroutine that sorts a variable length array in ascending or descending order. In the subroutine X represents the array of numbers and N represents the number of elements in the array. M determines whether the array is sorted in ascending order (M must be a positive 1), or descending order (M is any number other than 1). If NOOP is any positive number, then the sorted data will be printed on the line printer (unit 2). If it is negative, then the sorted data will not be printed. $ is the statement number that is desired for the return of control.

UNIVAC
DIVISION OF SPERRY RAND CORPORATION

FORTRAN

PROGRAMMING FORM

80/90

PROGRAM SØRT SUBRØUTINE PROGRAMMER REIDER ANDERSEN DATE AUGUST 1, 196- PAGE 1 OF 1 PAGES

"C" FOR COMMENT

STATEMENT NUMBER (5)	Cont. (6)	FORTRAN STATEMENT (7–72)	(80–90)
C		SØRTS VARIABLE LENGTH ARRAY X IN ASSENDING ØR DECENDING ØRDER.	
200		SUBRØUTINE SØRT (N,X,NØØP,K,M)	
201		DIMENSIØN X(N)	
202		IF (M .EQ. 1) GØ TØ 3	
C		DECENDING SØRT LØØP	
9		JUMP = 0	
203		DØ 8 K = 2,N	
204		IF (X(K - 1) .LT. X(K)) CALL SWAP (K)	
8		CØNTINUE	
205		IF (JUMP) 9,4,9	
C		ASSENDING SØRT LØØP	
3		JUMP = 0	
206		DØ 1 I = 2,N	
207		IF (X(I - 1) .GT. X(I)) CALL SWAP (I)	
1		CØNTINUE	
208		IF (JUMP) 3,4,3	1
4		IF (NØØP) 6	
5		CALL PRNT	
6		RETURN 4	
C		INTERNAL SUBRØUTINE THAT EXCHANGES TWO NUMBERS	
209		SUBRØUTINE SWAP (J)	
2		TEMP = X(J - 1)	
210		X(J - 1) = X(J)	
211		X(J) = TEMP	
212		JUMP = 1	
C		INTERNAL SUBRØUTINE THAT PRINTS THE SØRTED DATA	
213		SUBRØUTINE PRNT	
10		FØRMAT (I5,E20.6)	
214		WRITE (2,10) (L, X(L), L = 1,N)	
215		END	

UP-2508

UNIVAC 1107 FORTRAN

REVISION:

MANUAL NUMBER: U-3540

SECTION: Appendix 1

PAGE: 2

The first statement is a comment that describes the program and appears when the source program is listed.

Statement 200 is the Subroutine Subprogram statement and gives the name of the subroutine and lists the input and output variables.

Statement 201 is a DIMENSION Statement that defines the size of the array X depending on the value of the adjustable dimension variable N.

Statement 202 tests to see if the array is sorted is ascending or descending order.

Statement 9 sets the switch variable JUMP to zero.

Statement 203 is a DO Statement that controls the descending sort loop. The index is K and the limit is the integer variable N.

Statement 204 is a logical IF Statement that compares the elements in the array to determine the need of the internal subroutine SWAP, in order to arrange the elements in the desired order . If the test is satisfied the Subroutine SWAP will be executed. If the test is not satisfied the Subroutine CALL will be skipped and the next executable statement, Statement 8, will be performed.

Statement 8 is a dummy statement needed to complete the descending sort DO loop. CONTINUE is needed as the DO loop is not permitted to end in an IF Statement.

Statement 205 is an IF Statement which tests for the completion of the sort on the array X. If the array is in the desired order, then control will transfer to Statement 4; otherwise, it will transfer to Statement 9 to complete the sort.

Statement 3 sets the switch variable JUMP to zero.

Statement 206 is a DO Statement that controls the ascending sort loop. The index is I and the limit is the integer variable N.

Statement 207 is a logical IF Statement that compares the elements in the array to determine the need of the internal subroutine SWAP, in order to arrange the elements in the desired order. If the test is satisfied the Subroutine SWAP will be executed. If the test is not satisfied the Subroutine CALL will be skipped and the next executable statement, Statement 8, will be performed.

Statement 1 is a dummy statement needed to complete the ascending sort DO loop. CONTINUE is needed as the DO loop is not permitted to end in an IF statement.

Statement 208 is an IF statement which tests for the completion of the sort on the array X. If the array is in the desired order, then control will transfer to statement 4; otherwise, it will transfer to statement 3 to complete the sort.

Statement 4 is an IF statement which tests the switch LOOP to see if it is desired to print the array X.

Statement 5 is a CALL statement that calls the internal subroutine PRNT.

Statement 6 returns control to the calling program and transfers control to the statement number that replaced the dummy argument $.

Statement 209 terminates the main subroutine, SORT, and it is the internal SUBROUTINE subprogram statement for the SWAP subroutine.

Statements 2, 210, and 211 exchange the numbers in X (J) and X (J-1).

Statement 212 sets the switch, JUMP, that statements 205 and 208 test for in each sort completion.

Statement 213 terminates the internal subroutine SWAP, acts as its return statement, and is the internal subroutine subprogram statement for the PRNT subroutine.

Statement 10 is the Format Statement for Statement 214.

Statement 214 is the output statement that prints a sequence number and the ordered values of array X.

Statement 215 is the END statement which acts as a return statement for the subroutine PRNT and terminates the entire Subroutine Program.

APPENDIX 2. WAYS OF CHANGING VALUES IN STORAGE LOCATIONS

For purposes of planning the sharing of memory, the following statements will cause new values to be stored in a location:

1. An arithmetic or logical statement will cause a value to be stored for the variable on the left-hand side of the equal symbol. (=)

2. Execution of a DO will sometimes store a new value of the index variable.

3. Execution of an ASSIGN j to k will store a value in k.

4. Execution of Input Statements will store values in locations specified by the input list.

5. Execution of a CALL may cause values to be stored in variables given as parameters or in common storage.

6. Reference to an external function which is not "abnormal" will not cause new values to be stored anywhere with respect to the referencing program. If the function is abnormal, then the reference will be treated as a CALL.

APPENDIX 3. LIST OF AVAILABLE FORTRAN STATEMENTS

The following table is a list of available statements for the UNIVAC 1107 FORTRAN Processor.

Category	Form
	ABNORMAL
	ASSIGN n to i
	BACKSPACE Unit
	BLOCK DATA
	CALL s (a_1, a_2, ..., a_n) or CALL s
	COMMON/Block name/Variable names/Block name/ Variable names
	CONTINUE
	COMPLEX
	DIMENSION array 1 (parameters), array 2 (parameters)....
	DATA
	DO n i = j, k, m
	DOUBLE PRECISION
	END
	END FILE Unit
	EQUIVALENCE (Variable names), (Variable names, ...),...
	EXTERNAL
	FORMAT (Format Specification)
	FUNCTION f (a_1, a_2, ... a_n)
conditional GO TO	GO TO m
unconditional GO TO	GO TO n
assigned GO TO	GO TO i (n_1, n_2, ..., n_m)
computed GO TO	GO TO (transfer list), i
arithmetic IF	IF (arithmetic statement) j, k, m
logical IF	IF ("logical Expression") "FORTRAN statement"
	INTEGER
	INTEGER FUNCTION
	LOGICAL
	LOGICAL FUNCTION
	PARAMETER
	PAUSE n (n may be omitted)
	READ (Unit) List
	READ (Unit, Format) List
	REAL
	REAL FUNCTION
	RETURN
	REWIND Unit
	STOP
	SUBROUTINE Name (a_1, a_2, ..., a_n)
	WRITE (Unit) List
	WRITE (Unit, Format) List

The following statements are maintained for the purpose of compatibility. They are not a part of the new language but will be acceptable to the UNIVAC 1107 FORTRAN Processor and interpreted appropriately.

IF ACCUMULATOR OVERFLOW n_1, n_2

IF QUOTIENT OVERFLOW n_1, n_2

IF DIVIDE CHECK n_1, n_2

IF (SENSE LIGHT i) n_1, n_2

IF (SENSE SWITCH i) n_1, n_2

PRINT Format, List

PUNCH Format, List

READ n, List

READ INPUT TAPE i, n, List

READ TAPE i, List

SENSE LIGHT i

WRITE OUTPUT TAPE t, Format, List

WRITE TAPE t, List

FORTRAN

UNIVAC

DIVISION OF SPERRY RAND CORPORATION

U-3540 C PRINTED IN U.S.A.