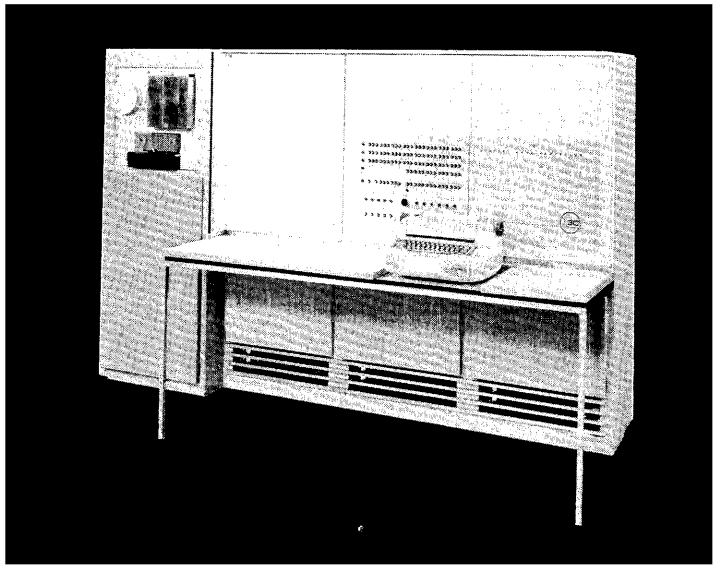
DDP-24



FORTRAN II MANUAL

PRELIMINARY
FORTRAN II MANUAL
FOR THE DDP-24
GENERAL PURPOSE COMPUTER

COMPUTER CONTROL COMPANY

CONTENTS

SECTION I

INTRODUCTION

A high speed electronic computer is an extremely powerful tool for the engineer. It can perform large amounts of calculation rapidly and accurately that might otherwise be completely impractical. However, the computer must be furnished with a complete program made up of instructions recognizable by the machine so that each and every step to be performed is defined. As computers have been developed during the past decade more operations have been included in the language of the computer; both primary and secondary storage have been increased, input and output equipment has been improved and made more flexible. While all of these developments are definite advantages to the user of the computer the process of writing the program has become more complex. The actual elapsed time for writing the instructions and checking out the program can be formidable. Consequently, as computers have been developed, techniques of programming have had to progress.

Much work has been done in the development of compilers which will translate a series of problem oriented statements called a source program into a machine language called an object program. The compiler will allocate proper storage locations for variables and constants, help search for programming or keypunching errors and produce a consistently efficient translation of the source program statements. In fact, the compiler can do all this faster and with fewer errors than the coder can write the machine language code.

FORTRAN, a contraction of Formula Translation, was originally written by IBM for the IBM 704, but since has been offered by several computer manufacturers including Computer Control Company. FORTRAN is relatively easy to learn; very little training is required. Of great importance is the fact that a FORTRAN source program can be compiled on many different computers, thereby avoiding difficulties which stem from changes in computers. The DDP-24 FORTRAN II language is intended to be capable of expressing any problem of numerical computation. In particular, it deals easily with problems containing large sets of formulae and many variables and permits any variable to have up to three independent subscripts. For those problems in which machine words have a logical rather than a numerical meaning, provisions have also been made in the DDP-24 FORTRAN language for logical and/or boolean computation. In those instances where FORTRAN may not be particularly suited to the specific problem solution, provision has been made for the incorporation of machine language subroutines and in-line machine language statements with FORTRAN statements. Certain statements in the FORTRAN language cause the object program to be equipped with its necessary input and output program. Those which deal with decimal information include conversion to and from binary and permit considerable freedom of format in the external medium. Arithmetic in the object program will generally be performed with extended precision floating-point numbers. These numbers provide 38 binary digits (about 11 decimal digits) of precision and may have magnitude between 10^{-75} and 10^{+75}, and 0. Full word fixed-point arithmetic is also provided.

The program produced by the FORTRAN compiler may be executed following the compilation. If the results are incorrect, the list of statements should be examined and the required corrections made. The program may then be recompiled and executed.

Provisions can be made by FORTRAN for data input and desired output either on equipment directly attached to the computer or on peripheral equipment. This peripheral equipment can consist of magnetic or paper tape units attached to a printer, card equipment or paper tape equipment which is independent of the computer. The use of these units afford a saving of time during input and output.

FORTRAN coding and operating procedures are given in this manual. In learning the FORTRAN system, the novice will profit by reading the primer carefully, then continuing on through the FORTRAN fundamentals in the following section.

WRITING PROGRAMS

Preparation and organization is the vital part of a program written in any system of coding, including FORTRAN. The engineering aspect of a problem is of primary consideration in the determination of the physical problem and its mathematical representation. The numerical methods require careful selection; they form the basis of calculation for operations such as integration, solution of differential equations, determination of special and transcendental functions. The success or failure of the program depends in large part upon the numerical methods used; critical values and range of variables are items to be included in this consideration. The numerical methods incorporated in the program should be scrutinized carefully for possible improvement or even complete change regardless of the fact that they may have been used for a long period of time. A flow diagram is extremely helpful in analyzing the problem. During the course of developing the flow diagram it will probably be revised and rewritten several times. In its early stages it will consist of a general diagram composed of blocks each one of which represents a logical part of the problem; later each block can be analyzed in detail. In usage, many of these blocks can be considered as building blocks, subprograms or subroutines.

The technique of building blocks is a fundamental concept in programming. To explain the concept, consider the oversimplified example of a common erector set. Standard parts can be assembled into many different types of structures. The kinds of structures that are built depend on the creative ability and ingenuity of the builder. The standard parts are of course building blocks. If it were known in advance that only certain types of structures were to be built, then it would be possible to prefabricate partial assemblies that would greatly simplify and speed up the overall construction. These partial assemblies are also building blocks but on a higher level (namely, level 2).

We can think of the available FORTRAN statements as standard parts. These statements may be combined in many ways to form different kinds of programs; the program is the result of the ingenuity of the programmer. FORTRAN statements can also be combined to form partial programs; the mechanism for doing this is subsequently described. The obvious advantage of these subprograms is that they speed up the coding of a problem. A less obvious but more important advantage is that a FORTRAN subprogram is really an enrichment of the basic FORTRAN language; it is a step in building a higher level language. With a suitable

library of building blocks (subprograms), the engineer has the ability to formulate his problem in a much more powerful, exact and concise language. There is no reason for limiting the language to two levels. Subprograms can be coded to use other subprograms to form a level 3 language, and so forth.

What should the building blocks be? The novice thinks immediately of certain mathematical subroutines that are frequently used, yet this is only a starting point. To exploit the building block technique a much more sophisticated approach is necessary; the class of problems to be solved must be looked at in its entirety. Building blocks should be able to use each other and must fit together easily in many different programs - there is no simple answer to the entire question. However, tremendous success has been achieved in the past by formulating the building blocks to correspond to actual parts of the general physical system. After the program has been planned with deliberation and care, the program can be written completely in FORTRAN statements on FORTRAN code sheets. This source program includes the statements to read the data and type results in precisely the form selected by the programmer. Statements that will use the building blocks at the appropriate time are also included. These written statements are now key-punched into paper tape or cards which make up the source program.

These statements are now ready to be translated by the FORTRAN compiler into computer language. The use of this compiler obviates the necessity of the programmer k..owing the language or internal code of the particular computer. In addition, it performs the clerical work of assigning storage locations. The program resulting from compilation is the object program. It may be punched into cards or paper tape or written on magnetic tape; the procedures for preparing a program to be compiled and executed are described in the following.

We are indebted to the Numerical Sciences Group, Los Angeles Division of North American Aviation, Inc., for the use of their Engineer's Computing Manual during preparation of the present document.

FORTRAN CODING FORM

PROGRAMMER | DATE | PAGE OF

PROGRAM | CHARGE

C for COMMENT / STATEMENT NUMBER	Continue	FORTRAN STATEMENT	IDENTIFICATION
1 5	6	7 18 29 40 51 62 72	73 80

SECTION II
PRIMER

Every type of electronic computer is designed to respond to a special code, called a "machine language," which differs for different types of computers. A program (set of instructions) telling a computer what steps to perform to solve a problem must ultimately be given to the computer in its own language. However, FORTRAN makes it unnecessary for the programmer to learn the appropriate machine language for the computer that will be used. The name FORTRAN comes from "FORmula TRANslation," and was chosen because many of the statements look like algebraic formulas.

FORTRAN has been developed to enable the programmer to state in a relatively simple language, resembling familiar usage, the steps to be carried out by the computer. The program written in the FORTRAN language is entered into the computer and is automatically translated by the FORTRAN compiler into a program called the "object" program, expressed in a language the machine can understand. Second, the computer solves the problem by executing the object program.

The FORTRAN language consists of various statements, which may be grouped into six classifications: arithmetic and logical statements, control statements, input-output statements, specification statements, subprogram facility statements, and in-line machine language statements. The programmer uses this language to state the steps to be carried out.

The FORTRAN compiler is a large set of machine language instructions which cause the computer to translate a FORTRAN program into an efficient machine language program.

Virtually any numerical or logical procedure may be expressed in the FORTRAN language. The FORTRAN system is intended to substantially reduce the time required to produce an efficient machine language program for the solution of a problem, and to relieve the programmer of a considerable amount of manual clerical work, minimizing the possibility of human error by relegating the mechanics of coding and optimization of the computer.

CAPABILITIES OF COMPUTERS

In order to clarify what a computer can do and what it cannot do consider the problem of finding the roots of a quadratic equation. The computer cannot be given an equation of the form

$$3x^2 + 1.7x - 31.92 = 0$$

and be directed to find its roots. The computer can, however, be directed to computer the value of

$$\frac{1.7 + \sqrt{(1.7)^2 - 4(3)(-31.92)}}{2(3)}$$

which gives one of the roots of the preceding equation. That is, the computer must be told how to find the answer. It will do the work.

Most calculations performed by the computer are carried out in REAL (floating-point) form. Numbers are represented in the machine in this form, and the results produced by the arithmetic operations are usually in this form. For example, calculation of the product

$$5.0 \times 0.0037$$

would be carried out by the computer in a form analogous to

$$(.5000\ 0000 \times 10^{1}) \times (.3700\ 0000 \times 10^{-2}) = (.1850\ 0000 \times 10^{-1}).$$

This would be the case even though the numbers were entered as 5.0 and 0.0037 and the result were printed as 0.0185. All real numbers in the computer are carried to about 11 significant decimal digits. Numbers outside the range 10^{-75} and 10^{+75} (other than zero) cannot normally be accommodated.

PROCESSING THE FORTRAN JOB

This primer describes the writing of FORTRAN programs for a general-purpose, high-speed, internally stored program digital computer such as the DDP-24.

CONTENTS OF THE FORTRAN PRIMER

As an aid to efficient study, the FORTRAN language is approached cumulatively through three stages. The division into three parts is convenient for the description of successively more complex problem-solving procedures.

First, it is possible to direct the computer to read individual numbers from tape or cards, combine them according to formulas involving arithmetic operations and standard functions (such as sine, square root, log, etc.), make tests and follow different directions depending upon the outcome of each test, and finally type or print the results.

Secondly, additional types of statements are presented which provide for the definition and use of functions peculiar to the problem to be solved; the iterative manipulation of subscripted variables (the elements of vectors or lists of numbers); the use of paper tape for input of information; and greater flexibility in the format of input and output information. When magnetic tape is used for input and output information, the computer can read or write at a much greater rate than is possible when the computer reads cards or paper tape and prints or types results directly. Since all information to be used or processed by the computer (other than magnetic tape output from a previous computer operation) must initially be recorded on punched cards or paper tape, and since it is often desirable to maintain permanent records in punched or in printed form, separate peripheral equipment may be used to transfer information from cards to tapes, from tapes to cards, and from tapes to printed form.

Finally, to these facilities is added the ability to handle matrices and three-dimensional arrays of numbers, to perform more complex iterative procedures, and to direct the flow of control within a program more flexibly. Since FORTRAN language statements have to be translated by a computer before a problem is ready to be run, FORTRAN statements must be written in exactly the proper form. The machine has no ability to understand what was meant; it can only translate what was written. Therefore, the omission of a single decimal point or operation symbol will make a FORTRAN statement incapable of being translated correctly. For this reason, the rules for writing FORTRAN statements must be carefully followed. In devising

the FORTRAN System, considerable effort was devoted to making these rules consistent and having them conform to familiar usage wherever possible. A number of examples have been provided to illustrate these rules without having to include specific statements of them in the text; a list of items to check is also given, which should answer many questions. Some of the rules in the check lists are not explicitly stated elsewhere.

PAPER TAPE INPUT, ARITHMETIC OPERATIONS, STANDARD FUNCTIONS, TYPED OUTPUT

The DDP-24 FORTRAN compiler provides for several source input media (e. g., card reader, magnetic tape, etc.); however, we are primarily concerned with the standard Input/Output devices offered with the DDP-24; paper tape reader, paper tape punch, and typewriter. The section concerned with operating procedures discusses the non-standard I/O devices and their compatibility with the devices described herein.

INTRODUCTION

Consider the quadratic equation example previously presented

$$3x^2 + 1.7x - 31.92 = 0$$

The algebraic representation for one of the two roots of the equation could be written

$$\text{root} = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

where A = +3
B = +1.7
C = -31.92

The complete FORTRAN program which describes this calculation and provides for typing the result may be written in six separate statements as follows, plus three statements which provide for the end of the job.

Example 1:

```
      A = 3.
      B = 1.7
      C = -31.92
    4 ROOT = (-B + SQRTF(B**2 - 4. * A * C))/(2. * A)
      TYPE 7, ROOT
    7 FORMAT(1H␣6E17.8)
      CALL EXIT
      STOP
      END
```

NOTE: ␣ indicates a blank or space is required

The first statement means: "Assign the value 3. to the variable A." The next two statements have a similar meaning. The fourth statement means: "Evaluate the expression on the right side and assign the result to the variable ROOT." The fifth statement types the computed value of ROOT in a form indicated by the sixth statement. The last three statements indicate that this job is complete.

Notice the sequential nature of the program. The computer executes instructions in the same order as the order of the statements. For example, if the fourth statement were to be moved up and made the first statement, then the computer would evaluate ROOT before obtaining the desired values of A, B and C. ROOT would therefore be evaluated using some arbitrary unknown values for these variables.

The above example was written to illustrate the use of variables. However, the same result could be obtained by writing statement number 4 as follows:

```
4 ROOT = (-1.7 + SQRTF(1.7**2 - 4. * 3. * (-31.92)))/(2. * 3.)
  TYPE 7, ROOT
```

in which the actual numerical values appear in the statement describing the evaluation of ROOT.

The program in Example 1 illustrates the use of four types of FORTRAN statements which will be introduced in this subsection.

1) Arithmetic statements (e.g., first four statements in the program).

2) Input-output statements. The fifth and sixth statements are output statements.

3) Control statements, including

   IF

   Unconditional GO TO

   STOP

   END

4) Subprogram facility statement:

   CALL EXIT

ARITHMETIC STATEMENTS

The first four statements in Example 1 are called arithmetic statements. An arithmetic statement looks like a simple statement of equality. The discussion of arithmetic statements will here be restricted to those in which the left side of the equality is a simple variable. (In a later paragraph, additional facilities will be presented for handling arithmetic statements in which the left side is a function of one or more variables.) The right side of all arithmetic statements is an expression which may involve parentheses, operation symbols, constants, variables, and functions, combined in accordance with a set of rules much like that of ordinary algebra.

The fourth statement in Example 1 illustrates the use of six of the ten basic operations in the FORTRAN language.

Operator	Definition	Example
+	Add	a + b
-	Subtract	a - b
*	Multiply	a * b
/	Divide	a / b
**	To the exponent	a**b
=	Replaced by (1)	a = b

If only the types of statements presented in this section are used, all calculations will be carried out by the computer in REAL (floating-point) arithmetic, and the programmer must so instruct the computer by writing all constants with a decimal point.(2) All numbers (and only numbers) are considered constants, with the exception of statement numbers.

ORDER OF OPERATIONS

The FORTRAN arithmetic expression

```
A**B*C + D**E/F - G
```

will be interpreted to mean

$$A^{B}C + \frac{D^{E}}{F} - G$$

That is, if parentheses are not used to specify the order of operations, the order is assumed to be:

1) exponentiation

2) multiplication and division

3) addition and subtraction

Parentheses are used in the usual way to specify order, for example

$$(A(B + C))^{D}$$

is written in FORTRAN as

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

(1) Note that the meaning of the = operator is not precisely the same as its meaning in standard arithmetic notation.

(2) Exception: Do not use a decimal point with an integer used as an exponent.

There are only three exceptions to the ordinary rules of mathematical notation. These are:

1) In ordinary notation AB means A · B or A times B. However, AB never means A*B in FORTRAN. The multiplication symbol cannot be omitted.

2) In ordinary usage, expressions like A/B · C and A/B/C are considered ambiguous. However, such expressions are allowed in FORTRAN and are interpreted as follows:

A/B*C means (A/B)*C

A*B/C means (A*B)/C

A/B/C means (A/B)/C

Thus, for example, A/B/C*D*E/F means ((((A/B)/C)*D)*E)/F. That is, the order of operations is simply taken from left to right, in the same way that

A + B - C + D - E

means

(((A + B) - C) + D) - E

3) The expression A^{B^C} is often considered meaningful. However, the corresponding expression using FORTRAN notation, A**B**C, is not allowed in the FORTRAN language. It should be written as (A**B)**C if $(A^B)^C$ is meant, or as A**(B**C) if $A^{(B^C)}$ is intended.

FUNCTIONS

Besides the ability to indicate constants (such as 3.57 and 2.0), simple variables (like A and ROOT), and operations (such as - and *), it is also possible to use functions. In Example 1, SQRTF ( ) indicates the square root of the expression in parentheses.

Since the number of possible functions is very large, each installation will have its own list of available functions, with information about their use. Functions given in this list must be referred to exactly as indicated.

Some functions which might appear in a typical list are:

FORTRAN Symbol	Function
ABSF(X)	$\|X\|$ (absolute value of X)
SQRTF(X)	$\sqrt{X}$
SINF(X)	sin X
COSF(X)	cos X
EXPF(X)	e^X
ARCTANF(X)	arctan X

FORTRAN Symbol	Function
LOGEF(X)	$\log_e X$
LOG10F(X)	$\log_{10} X$
INTF(X)	integral part of X
MAX1F(X, Y, Z)	maximum of X, Y, and Z

Notice, as in the last example, that a function may have more than one argument; as in general mathematical usage, multiple arguments are separated by commas. Facilities for defining functions peculiar to the problem at hand and not currently available from the subroutine library list will be presented later.

INPUT-OUTPUT STATEMENTS

These paragraphs present FORTRAN statements which can direct the compiler to take numbers from a paper tape reader and, after carrying out the desired calculations, type the results. Consider again the example of finding a root of a quadratic equation.

In many cases it will be desired to find ROOT for a number of sets of values of A, B, and C. To do this, the computer would have to be directed to read a paper tape in which values for A, B, and C have been punched, compute the value of ROOT, type ROOT (along with A, B, and C), read more paper tape with different values for A, B, and C, compute and type the corresponding value of ROOT and so on. In this case, the FORTRAN program could be written:

Example 2:

```
10 READ PAPER TAPE 5, A, B, C
 5 FOTMAT(6E12.8)
   ROOT = (-B + SQRTF(B**2 - 4. * A * C))/(2. * A)
   TYPE 7, A, B, C, ROOT
 7 FORMAT(1H 6E17.8)
   GO TO 10
   END
```

The first statement (which has been given the number 10 for reference purposes) causes the computer to read the first information which was punched on the paper tape. Three numbers should be on this tape, represented by three sets of punches. The value of the first number is named A; the value of the second number, B; and the value of the third number, C. The computer then proceeds to compute ROOT as before, after which it types (on one line across the page) the values names A, B, C and ROOT in that order. Upon reaching the last statement, the computer is directed: "Go to the statement numbered 10 and do what it says." Thus the computer reads the next set of values for A, B, and C, then computes ROOT, provides for typing the current values of A, B, and C and the computed value of ROOT, and again returns to statement 10. This process continues as long as there are data to be read. When the data are exhausted (a stop code is punched at the conclusion of the data), the computer will type "END OF JOB" and halt.

Note that when the program concludes in this manner, with a transfer to a statement which directs the computer to read more data, the CALL EXIT and STOP statements are not needed. However, an END statement must always be placed at the end of the program to be compiled.

FORTRAN provides facilities for specifying the format of input data and of typed output in a great variety of ways. FORMAT statements are used to specify the desired arrangement for input paper tape and for typing.

A few simple FORMAT statements are given here and further details about FORMAT statements are given later. Each input or output statement refers to a FORMAT statement which specifies the form of the information punched on the paper tape or to be typed. Two types of input and output are discussed here, namely, REAL and INTEGER (fixed-point).

REAL. Data is read into the computer by means of an input statement such as

READ PAPER TAPE 15, A, B, C

where 15 represents the statement number of the FORMAT statement to be used, and A, B, C are the items to be read. If the FORMAT statement is

15 FORMAT (3E12.8)

the data will be assumed to be written in REAL notation. A REAL number is represented by a fraction multiplied by a power of 10. For example, the number 496.82 can be written in REAL form as 0.4982×10^3. The numbers to be read with the noted FORMAT statement may be written as in Example 3.

Example 3:

```
␣3␣␣␣␣␣␣+01          First case
␣17␣␣␣␣␣+01           First case
-3192␣␣␣␣+02

␣.15794␣␣+01
-17.3␣␣␣␣+00          Second case
␣.23␣␣␣␣␣-03
```

␣ represents a space

Note that if no decimal point is written, as in the first case, the number is considered to have 8 decimal places as indicated by the "8" in the FORMAT (3E12.8) statement. This would place the decimal point immediately to the left of the numbers in the first case.

Data for succeeding cases can be written in a similar way, each case separately. The sign is written in the first position of each field although a positive sign may be omitted. A sign, either positive or negative, must precede each decimal scale.

The information can be typed by means of an output statement

TYPE 20, A, B, C, ROOT

and its associated FORMAT statement

20 FORMAT (1H␣4E17.8)

The typing would appear as

```
0.30000000E 01   0.17000000E 01   -0.31920000E 02   0.29908501E 01
0.15794000E 01  -0.17300000E 02    0.23000000E-03   0.10953513E 02
```

Notice that the FORMAT statement used for output begins with the characters "1H" followed by a blank space. This provides for moving the typewriter carriage and results in single-spaced lines of typing. There are several different ways to provide for spacing the typed lines.

FIXED POINT. Data read into the computer by means of the input statement

READ PAPER TAPE 20, A, B, C

and the associated FORMAT statement

20 FORMAT (3F12.8)

is assumed to be written in fixed-point notation. A fixed-point number read into the computer in this manner is converted to a REAL number for internal use. Data for Example 3 might be written as in the following example.

Example 4:

```
b3.0bbbbbbbb
b1.7bbbbbbbb          First case
-31.92bbbbbb

 0015794bbbb
-0173bbbbbbb          Second case
b00000023bbb
```

Two different ways of indicating the decimal point are shown: by writing it, as in the first case, and by using the number of decimal places shown in the FORMAT statement, as in the second case. Since the statement FORMAT (3F12.8) calls for 8 decimal places, the decimal point is considered to be between the third and fourth digits on each line of the second case.

The sign is written in the first position; it may be omitted for a positive number. If the decimal point is written, the number may be placed anywhere in the field.

Information may be typed in fixed-point notation using the following statements:

TYPE 22, A, B, C, ROOT

22 FORMAT (1H 4F16.5)

The output is typed in this manner:

3.00000	1.70000	-31.92000	2.99085
1.57940	-17.30000	0.00023	10.95351

This format should not be used for numbers having more than eight digits to the left of the decimal point. Also, note that five decimal places are specified.

CALL EXIT, STOP, AND END STATEMENTS

CALL EXIT is a subprogram facility statement which tells the computer that the end of the job has been reached. The END statement is for the information of the compiler, and is not used during execution of a program.

The CALL EXIT statement may be omitted in certain cases, such as that encountered in Example 2 where the absence of data indicates the end of the calculation. In this case, the STOP statement should also be omitted. However, an END statement must always be the physically last statement of the program. This tells the compiler that there are no more statements to be translated into machine language in this program.

CONTROL STATEMENTS - IF AND UNCONDITIONAL GO TO

The unconditional GO TO statement is used to specify at some point in a program that the next statement to be executed is not the one following, as it normally would be, but, instead, the statement numbered n. The statement GO TO n transfers control to statement n, and execution proceeds from there.

As an introduction to another type of control statement, consider the following problem:

> Given values a, b, c, and d punched on paper tape, and a set of values for the variable x punched one per line, evaluate the function defined by

$$f(x) = \begin{cases} ax^2 + bx + c & \text{if } x < d \\ 0 & \text{if } x = d \\ -ax^2 + bx - c & \text{if } x > d \end{cases}$$

for each value of x, and type x and f(x).

The FORTRAN program for this problem could be written as follows:

Example 5:

```
10 READ PAPER TAPE 5, A, B, C, D
11 READ PAPER TAPE 5, X
 5 FORMAT(6E12.8)
12 IF(X - D) 13, 15, 17
13 FOFX = A * X**2 + B * X + C
14 GO TO 18
15 FOFX = 0
16 GO TO 18
17 FOFX = -A * X**2 + B * X - C
18 TYPE 6, X, FOFX
 6 FORMAT(1H 2E17.8)
19 GO TO 11
   END
```

The values for A, B, C, and D are read (statement 10), and the first value of X is then read (statement 11). Statement 12 is then executed. Statement 12 means: "If the quantity (X-D) is negative, go to statement 13; if it is zero, go to statement 15; and if it is positive, go to statement 17." Hense, if X < D, the value of FOFX is calculated by means of the proper formula, and the execution of statement 14 transfers control to statement 18, which is executed next. Similarly, if X = D, control goes from the IF statement to the proper formula (statement 15), and then from statement 16 to statement 18. If X > D, the IF statement selects statement 17, after which statement 18 is automatically taken next. Thus, in all three cases, control eventually reaches statement 18, the output statement, which provides for typing the values of X and FOFX. Statement 19 then returns control ot the READ statement, which reads in the next value of X. The whole pattern repeats until all of the X-data have been processed. The computer will consider this program completed when there are no more data to read.

As has been illustrated, the IF statement is a kind of conditional, three-way GO TO statement. It often happens, as in the above problem, that the computer must choose one of alternative paths depending on whether the current value of an expression is negative, positive, or zero. This is done, as in the above program, by writing

IF (E) n_1, n_2, n_3

where (E) is an expression, and the number of the statement to which control is to be transferred is n_1 if (E) is negative; n_2, if (E) is zero; and n_3, if (E) is positive.

WARNING

Provision for each branch must always be included.

It is not necessary to number every FORTRAN statement in a program. The only statements which must be numbered are those to which reference is made (as in GO TO or IF statements). Any numbers between 1 and 32767 may be used, provided two different statements are never given the same number.

SAMPLE PROBLEMS

Problem 1: Find the approximate numerical solution of the ordinary differential equation

$$dy/dx = xy + 1$$

in the interval $0 \leq x \leq 1$, given that

$$y = 0 \text{ when } x = 0$$

A method which can be used to approximate the solution of this equation is as follows:

Assume that a point (x_0, y_0) of the solution function is known. Thus the point $x_0 = 0$, $y_0 = 0$ is known. It is then known from the differential equation that dy/dx, the rate of change of y with respect to x, at this point is $x_0y_0 + 1$. Hence, an increment in x of Δx would produce an approximate change in y of

$$\Delta y = \Delta x(x_0y_0 + 1)$$

Let Δx be the interval between successive x_i terms ($i = 0, 1, 2, \ldots$). Then $y(\text{at } x_1) = y(\text{at } x_0 + \Delta x) = y_0 + \Delta y = y_0 + \Delta x(x_0y_0 + 1)$.

After the point (x_1, y_1) has been obtained, it can be used to find (x_2, y_2) in a similar way:

$$y_2(\text{at } x_2) = y_2(\text{at } x_1 + \Delta x) = y_1 + \Delta x(x_1y_1 + 1)$$

The procedure is continued until the point x = 1 is reached, the upper bound of the interval for which the solution is being found. In general, the equation for stepping forward is

$$y_{i+1} = y_i + \Delta x(x_iy_i + 1).$$

Since it depends on the mesh size, Δx, the error of approximation is left as a parameter in the program. The solution for various values of Δx can be compared to give an empirical idea of the error. To type the value of every point obtained would be unnecessary and costly, since Δx must be quite small, so the program has been arranged to type only at intervals of 0.01. The program for this example, with an explanation of some of the statements, follows.

Example 6:

```
      READ PAPER TAPE 10, DELTAX
   10 FORMAT(1E21.8)
      TYPE 12, DELTAX
   12 FORMAT(1H 2E17.8)
      XPRINT = 0.01
      X = 0.0
      Y = 0.0
    3 Y = Y + DELTAX * (X * Y + 1.0)
      X = X + DELTAX
      IF(X - XPRINT) 3, 4, 4
    4 TYPE 12, X, Y
      XPRINT = XPRINT + 0.01
      IF(X - 1.0) 3, 5, 5
    5 CALL EXIT
      STOP
      END
```

The only input to the program is the value of DELTAX. The first statement causes the computer to read DELTAX into core storage. The third statement causes the computer to provide for DELTAX to be typed. The fifth statement initialized XPRINT to 0.01; the first value of X which equals or exceeds XPRINT will be the next value typed. The next two statements assign the proper initial values to X and Y. Statement 3 is the basic equation for finding the next value of Y. The next statement calculates the new value of X. This value of X is then compared with the value of XPRINT; if it is less than this value, control goes back to calculate the next point. As soon as X equals or exceeds XPRINT, the calculation is interrupted to allow the current values of X and Y to be typed according to statement 4 (in REAL form).

The value of XPRINT is increased by 0.01 for the next value to be typed. Then a test is made to determine whether the value of X has reached 1.0. If X equals or exceeds 1.0, the problem is finished and the computer types "END OF JOB" and halts (statement 5); if not, control returns to statement 3 to calculate the next point.

Problem 2: Determine the current in an alternating current circuit consisting of resistance, inductance, and capacitance in series, given a number of sets of values of resistance, inductance, and frequency. The current is to be determined for a number of equally spaced values of the capacitance (which lie between specified limits) for voltages of 1.0, 1.5, 2.0, 2.5, and 3.0 volts.

The equation for determining the current flowing through such a circuit is

$$I = \frac{E}{\sqrt{R^2 + \left(2\pi fL - \frac{1}{2\pi fC}\right)^2}}$$

where I = current, amperes

E = voltage, volts

R = resistance, ohms

L = inductance, henrys

C = capacitance, farads

f = frequency, cycles per second

π = 3.1416

The FORTRAN program could be written as follows:

Example 7:

```
10 READ PAPER TAPE 5, OHM, FREQ, HENRY
11 READ PAPER TAPE 5, FRD1, FRDFIN
 5 FORMAT(3E12.8)
12 TYPE 7, OHM, FREQ, HENRY
 7 FORMAT(1H 3E17.8)
13 VOLT = 1.0
14 TYPE 7, VOLT
15 FARAD = FRD1
16 AMP = VOLT / SQRTF(OHM**2 + (6.2832 * FREQ * HENRY)
  X           -1/ (6.2832 * FREQ * FARAD))**2)
17 TYPE 7, FARAD, AMP
18 IF (FARAD - FRDFIN) 19, 21, 21
19 FARAD = FARAD + 0.000 000 01
20 GO TO 16
21 IF (VOLT - 3.0) 22, 10, 10
22 VOLT = VOLT + 0.5
23 GO TO 14
   END
```

Statement 10 causes the values of the resistance, the frequency, and the inductance to be read in that order. Statement 11 causes the initial and final values of the capacitance to be read. The values of the resistance, frequency and inductance are typed (statement 12). The initial value of the voltage is introduced and typed (statements 13 and 14). Statement 15 initializes the current value of the capacitance (denoted by FARAD) to the first value to be used in calculation (denoted by FRD1). The actual calculation is specified by statement 16. The result of that calculation, together with the current value of the capacitance, is typed (statement 17). The current value of the capacitance is compared with the final value to determine whether or not all values have been investigated (statement 18). If not, the expression (FARAD - FRD1) is negative and the program proceeds to statement 19, which causes the current value of the capacitance to be increased by the given increment. This is followed by a transfer (statement 20) to statement 16 which causes the calculation to be repeated for the new value of the capacitance. If the expression in statement 18 is zero or positive, all values of the capacitance have been investigated and the program transfers to statement 21.

At this point the value of the voltage is compared with the upper bound to determine whether or not all specified values of the voltage have been used. If not, the expression in statement 21 (VOLT - 3.0) is negative and the program proceeds to statement 22, which causes the value of the voltage to be increased. Following this, a transfer (statement 23) is made to statement 14, which causes the new value of the voltage to be typed. The program proceeds to statement 15, and the entire process of investigating all values of the capacitance is begun again.

If all values of the voltage have been used (the expression in statement 21 is zero or positive), the calculations for the current set of values of resistance, frequency, and inductance are finished. The program is returned to statement 10 so that the two sets of data defining the next case may be read and the program repeated. This process is repeated until all of the cases have been considered; i.e., all of the data have been read.

CHECK LIST

In the preceding descriptions of statements no attempt was made to cover in detail all of the information necessary or helpful in writing a program using these statements. The following list of items, together with what has already been presented, supplies this information.

1) The basic characters which may be used in writing a FORTRAN statement are:

a) A, B, C, . . ., Z (26 alphabetic characters, capital letters only)

b) 0, 1, 2, . . ., 9 (10 numerical characters)

c) Special characters:

+ (plus); - (minus); *Asterisk); / (slash); ((Left parenthesis);) (right parenthesis); , (comma); = (equal sign); . (decimal point); and ' (apostrophe)

2) The digits 5, 2, 1, and 0 must be carefully distinguished from the alphabetic characters S, Z, I, and O.

3) If calculations involving a constant (i.e., any number except a statement number) are to be carried out in REAL arithmetic, as is always the case if only the types of statements presented in this section are used, the constant must be written with a decimal point. Exception: Do not use a decimal point with an integer used as an exponent.

4) A variable symbol can consist of eight or fewer characters. It must satisfy the following conditions:

a) The first character must be alphabetic.

b) The first character cannot be I, J, K, L, M, or N, unless the variable name has been declared in a REAL specification statement.

c) Any character following the first may be alphabetic or numerical, but not one of the special characters.

d) The names of all functions either with or without the terminal F, must not be used as variable symbols. For example, if SINF is used as the name of a function, neither SINF nor SIN can be used as a variable symbol.

5) If a function is used, the name of the function as written by the programmer must agree exactly with the name as it appears on the list.

6) The argument of a function is enclosed in parentheses; e.g., SINF (X).

7) If a function has more than one argument, the arguments are separated by commas; e.g., MAX1F (X, Y, Z).

8) The left side of a statement must never be a constant. In the type of arithmetic statement covered in this section, the left side is always a simple variable; e.g., A. In a following paragraph, arithmetic statements will be extended to include function statements, in which the left side is a function of one or more variables.

9) Never omit the intended operation symbol between two quantities; e.g., do not write AB for A*B.

10) Never write two operation symbols in a row; e.g., do not write A * -B for A * (-B). There are no exceptions. The exponentiation symbol ** may seem to be an exception, but it is regarded as a single symbol.

11) Blank spaces can be used or not used, as desired, since blanks are ignored in the translation. (Exception: In output FORMAT statements described in this section, the 1H used for carriage control must be followed by a blank.) For example,

A=0.1

could be written

A = 0.1

and

GO TO 25

could be written as

GOTO25

12) The prescribed form for the various non-arithmetic statements must be followed exactly, except for the arbitrary use of blank spaces. For example, the statements

READ PAPER TAPE 5 A, B

IF A-B, 5, 6, 7

are incorrectly written, they should be written

READ PAPER TAPE 5, A, B

IF (A-B) 5, 6, 7

with the punctuation marks appearing exactly as shown.

13) The magnitude of every non-zero REAL quantity must lie between 10^{-75} and 10^{75} By "quantity" is meant any constant or any value assumed by a variable or function in the course of the calculation.

14) When a number is read by means of a READ PAPER TAPE 5, statement, where statement 5 is FORMAT (6E12.8), only eight digits of the number are used. The exponent must have two digits and a sign, and the number is preceded by a sign, making a maxmimum of twelve characters to a field.

15) A number to be typed by means of a TYPE 7, statement, where statement 7 is FORMAT (6E17.8), will be typed as a REAL number whose fractional part has 8 digits and whose exponent part has 2 digits and sign.

16) The program statement which is written last should be a statement which causes a transfer to some other statement in the program (a GO TO or an IF statement). If this is not practical, a CALL EXIT statement followed by a STOP statement can be used.

17) An END statement must be the physically last statement of every FORTRAN source program.

DEFINITION OF FUNCTIONS, MANIPULATION OF SINGLE-SUBSCRIPTED VARIABLES, MAGNETIC TAPE INPUT AND OUTPUT

INTRODUCTION

The part of the FORTRAN language presented in the preceding subsection can be used to direct the operation of the computer in the solution of certain problems. However, it is difficult or impossible to program the solution of some problems using only the types of statements described in the previous subsection. Those statements, grouped by type, were as follows:

Arithmetic statements	
Input-output statements	(READ PAPER TAPE (TYPE (FORMAT
Control statements	(IF (Unconditional GO TO (STOP (END
Subprogram facility statement	CALL EXIT

In this subsection, arithmetic statements will be extended to include function statements, and additional types of statements will be introduced which make is possible to direct the computer in the solution of problems more complex than those dealt with previously.

SUBSCRIPT NOTATION

If programming were done using only the types of statements previously presented, laborious programming would be necessary to carry out relatively simple iterative calculations or logical steps such as are encountered in the addition of two vectors or the selection of a certain number from a list of numbers. However, it is possible, using the additional types to be presented here to employ the subscript notation of mathematics to make the programming of such problems easier.

A mathematician would denote that c_i is the sum of the vectors (a_1, a_2, a_3) and (b_1, b_2, b_3) by writing

$$c_i = a_i + b_i \qquad i = 1, 2, 3$$

Notice that the first part of the statement

$$c_i = a_i + b_i$$

is a general statement which, in effect, becomes three specific statements

$$c_1 = a_1 + b_1$$

$$c_2 = a_2 + b_2$$

$$c_3 = a_3 + b_3$$

by assigning the values 1, 2, and 3 to i.

By using the FORTRAN language, it is possible to make general statements like $c_i = a_i + b_i$, and to make other statements which assign the desired values to i. When a general statement is executed, it is always executed in one of its specific senses. For example, if the variable I has the value 3 when the FORTRAN equivalent of $c_i = a_i + b_i$

```
C(I) = A(I) + B(I)
```

is executed, the values denoted by A(3) and B(3) are added and the sum is assigned as the value of C(3). Thus, to compute the sum vector

```
(C(1), C(2), C(3))
```

it is necessary to execute the general statement 3 times, each time with I having one of the values 1, 2, 3. Therefore, in addition to providing for arithmetic statements with subscripted variables, it is necessary to provide for a method of stating that a given set of such statements should be executed repetitively for certain values of the subscript. The FORTRAN statement which provides this ability is called a DO statement. An example of a DO statement, followed by an explanation, is

```
DO 20 I = 1, 250
```

This statement instructs the computer: "Execute all statements which immediately follow, up to and including the statement numbered 20, 250 times (the first time for I = 1, the second time for I = 2, and so on, and the last time for I = 250), and then go on to the statement following statement 20." Thus, to return to the example of vector addition, the FORTRAN statements necessary to add A(I) and B(I) are

```
      DO 1 I = 1, 3
1     C(I) = A(I) + B(I)
2     . . .
```

When the statement numbered 2 is encountered, the values of C(1), C(2), and C(3) will have been computed and stored.

Problem: It is required to compute the following quantities

$$P_i = \sqrt{\sin^2 (A_i B_i + C_i) + \cos^2 (A_i B_i - C_i)}$$

$$Q_i = \sin^2 (A_i + C_i) + \cos^2 (A_i - C_i)$$

for i = 1, . . ., 100. A possible FORTRAN program for this calculation follows.

Example 8:

```
 1 DIMENSION A(100), B(100), C(100), P(100), Q(100)
 2 TRIGF(X, Y) = SINF(X + Y)**2 + COSF(X - Y)**2
 3 READ PAPER TAPE 4, A, B, C
 4 FORMAT(3F12.8)
 5 DO 7 I = 1, 100
 6 P(I) = SQRTF(TRIGF(A(I) * B(I), C(I)))
 7 Q(I) = TRIGF(A(I), C(I))
 8 TYPE 9, (A(I), B(I), C(I), P(I), Q(I), I = 1, 100)
 9 FORMAT(1H 5F17.4)
10 CALL EXIT
   STOP
   END
```

Statement 2 defines the function TRIGF (X, Y) as equal to the expression ($\sin^2$ (X+Y) + $\cos^2$ (X-Y)). The DIMENSION statement indicates that the arrays A, B, C, P, and Q each have 100 elements. A, B, and C in the input statement will cause all elements of A, then all elements of B, and then all elements of C to be read into the computer. Notice that the input statement refers to a new type of FORMAT statement. The FORMAT statement specifies the external arrangement for input and output data. In this FORMAT statement, 3F12.8 means: "There are 3 fixed-point decimal fields per case, each field being 12 columns wide with 8 decimal places to the right of the decimal point." Statement 5 says: "DO the following statements through statement 7 for I = 1, I = 2, . . ., I = 100." Statements 6 and 7 compute P_i and Q_i.

The output statement says "Type the arrays A, B, C, P, and Q for I = 1, . . ., 100 as specified by FORMAT statement 9." Statement 10 indicates the end of the program.

Each number will be typed in a field of 17 columns; there will be 4 digits to the right of the decimal point and the number will be placed to the extreme right of the field. There will be as many as 5 numbers to the line; the first line will contain A_1, B_1, C_1, P_1 and Q_1.

The method of subscript notation and the use of the DO, FORMAT, DIMENSION, and function statements which have been introduced here will be further illustrated in the following pages of this section.

INTEGER CONSTANTS AND VARIABLES

In the previous subsection, only REAL constants (which must have a decimal point) and REAL variables (which did not begin with I, J, K, L, M, or N) were considered. However, it should be clear that REAL numbers are not always desirable nor necessary for use as subscripts; i.e., $X_{1.3}$ is not generally a useful notation, and $X_{3.0}$ is redundant and wastes space. INTEGER constants and INTEGER variables are more useful for this purpose. The rules which follow describe the method of writing such numbers:

1) INTEGER constants are numbers written without a decimal point.

2) INTEGER variables must begin with I, J, K, L, M, or N unless the over-ride capabilities provided by the REAL specification statement is utilized. For the purposes of this primer, the TYPE specification statements will not be discussed; therefore, INTEGER variables will always begin with I, J, K, L, M, or N.

When used in FORTRAN statements, a subscripted variable is written as the name of the variable followed by the subscript (an expression composed of constants or variables) in parentheses. For example, A(3) is the FORTRAN representation of A_3 and X(I) is the FORTRAN representation of X_i.

Subscripts are not restricted to single or INTEGER quantities. Any meaningful numerical expression may be used. The DDP-24 version of FORTRAN will perform the indicated computation and use the positive, truncated result as the subscript. For example,

Y(3.5*3) means Y_{10}

X(5-SQRTF(9.)) means X_2

Thus it is clear that mixed expressions are allowed as subscripts.

As noted above, REAL notation for subscripts is not generally used; however, there may be cases where the more sophisticated programmer can take advantage of this feature. Thus, it has been decided to allow the capability in DDP-24 FORTRAN. Since this area is considered to be a more advanced application of subscripting, it will not be covered in the primer. Rather all examples used will be the simpler INTEGER subscripts.

If a REAL variable, for example, A, is used as a subscripted variable, it represents the collection of variables A(1), A(2), A(3), . . . etc. and may not be used without a subscript except in an input-output statement when it is desired to transfer the entire array, or in an arithmetic statement where A will be interpreted as A(1). Thus it is not possible to use B(J) and B in different statements and expect to have both a vector, B(J), and a non-subscripted variable, B.

Reference to a subscripted variable whose subscript is an INTEGER variable, for example, X(N), is always interpreted in a specific sense determined by the value of N. Therefore, some statement which assigns a value to N, such as

```
N = I + J
```

or

```
DO 10 N = 1, 20
```

or

```
READ PAPER TAPE 5, N
```

should always be encountered before reaching a statement which refers to X(N).

MIXED EXPRESSIONS

INTEGER quantities may be used in REAL expressions. Such expressions would be considered mixed, since both REAL and INTEGER notation are represented. An example of a mixed expression is

```
A = I*B+2.5-(C/3)
```

Mixed expressions are always computed using REAL arithmetic. Thus, in the example above, the INTEGER variable I and the INTEGER constant 3 would first be converted to REAL numbers and then the indicated computation would be performed. In every element of an expression on the right side of an arithmetic statement were INTEGERS, the expression would not be considered mixed and computation would be performed in INTEGER arithmetic. The overall mode (i.e., REAL or INTEGER) of a result is determined by the mode of the left side of an arithmetic statement. For example, the formula

```
I = A+B**J
```

instructs the computer to compute the value of $A+B^J$ using REAL arithmetic, truncate the result (i.e., drop any fractional part), and assign the INTEGER so obtained as the value of I. This meaning results from the fact that the expression on the right is a REAL expression, whereas the variable on the left is an INTEGER variable. Conversely, the formula

```
A = JOB + N/3
```

instructs the computer to compute the value of JOB+N/3 using INTEGER arithmetic, put the resulting INTEGER in REAL form, and assign this as the value of A. Note that INTEGER arithmetic gives an INTEGER RESULT even for N/3. Thus, the value of 8/3 would be 2, the largest INTEGER not exceeding 8/3, whereas the value of 8./3. in a REAL expression is 2.66666...

DIMENSION STATEMENTS

Whenever a subscripted variable appears in a FORTRAN program, it is necessary to include a statement which indicates the size of the array referred to by this variable. This type of statement is a DIMENSION statement. A DIMENSION statement causes the compiler to assign the proper number of storage locations to each subscripted variable.

A DIMENSION statement consists of the name of each subscripted variable followed by an INTEGER in parentheses which represents the greatest number of elements which will ever be included in the array. The variables are separated by commas, and the whole group of names is preceded by the word DIMENSION. The INTEGER that specifies the greatest number of elements must be a number, not a symbol.

If the subscripted variables ALPHA(i), GAMMA(J), and VECTOR(N) appear in a FORTRAN program, then a DIMENSION statement mentioning these variables must also be included. Assume that the number of elements in ALPHA(I) will never exceed 100, the number in GAMMA(J) will never exceed 25, and the number in VECTOR(N) will never exceed 12. The DIMENSION statement must then be written

```
DIMENSION   ALPHA(100),   GAMMA(25),   VECTOR(12)
```

DIMENSION statements are not actually executed. No instructions corresponding to this statement will appear in the translated machine language program. In the FORTRAN source program, however, a DIMENSION statement giving the size of each array must precede the first executable statement in the program. A single DIMENSION statement, including all subscripted variables mentioned in the program, may be used, or separate statements may be used defining the size of each array.

DO STATEMENTS

An example of the use of a DO statement of the unconditional type appeared in Example 8. The usefulness of such a statement for carrying out repetitive calculations was mentioned then. The standard form for an unconditional DO statement is

DO N I = m_1, m_2

where N is a statement number

I is a variable

m_1 and m_2 are expressions

The meaning of the DO statement is: "Execute the statements immediately following this DO statement, up to and including the statement numbered N, first with I equal to m_1, then with I equal to $m_1 + 1$, etc., and finally with I = m_2, and then go to the statement following statement N."

The set of statements immediately following the DO statement and extending through statement N is called the range of the DO statement. In a later subsection the use of "nests" of DO statements, with one or more DO statements in the range of another, will be discussed. In the use of DO statements discussed in the present subsection, no DO statement will contain another DO statement within its range. However, the range of a DO statement may contain GO GO or IF statements, and these may transfer control out of this range.

As a further illustration of the usefulness of the DO statement, consider a number B and a set of fifty numbers, A(J). The problem is to select the smallest of the values of J for which B = A(J). A program to accomplish this could be written as follows:

Example 9:

```
10 DO 12 J = 1, 50
11 IF (B - A(J)) 12, 20, 12
12 CONTINUE
13                          .        If control reaches statement 13,
 .                          .        the search has not been
 .                          .        successful.
 .                          .
 .                          .
 .                          .
20                          .        If control reaches statement 20,
 .                          .        the desired value of J is avail-
 .                          .        able for use.
 .                          .
```

Control passes to statement 20, out of the range of the DO statement, as soon as J, the index of the DO statement, reaches a value for which B - A(J) equals zero. Any reference which is now made to J will be interpreted for J equal to that specific value. Whenever B - A(J) is not zero, it is desired to increase J and begin the range again. To accomplish this, control must reach the last statement in the range (which cannot be the IF statement) even though no more work remains to be done with the current value of J. In this example, therefore, the last statement in the range of the DO statement must be CONTINUE, which means "do nothing."

FUNCTION STATEMENTS

Within the limits of the part of FORTRAN introduced in the previous subsection, certain functions (specified by the installation) were permitted in writing arithmetic expressions, such as square root, sine, log, etc. The functions were restricted to those appearing in the list furnished by the computing center.

It is also possible, however, to write expressions involving functions peculiar to the problem at hand. Each desired function is defined by means of a function statement. For example, suppose it is desired to use the function

$$F(X) = 1.3\ E\ \sqrt{4.1X + X^2}$$

several times in a program. An "arithmetic statement function" defining G(X) might be written as follows:

```
GXXF(X) = 1.3 + SQRTF(4.1*X + X**2)
```

A later arithmetic formula in the program, employing GXXF, might be

```
Y = 10.3*GXXF(ALPHA*BETA) + 14.7
```

In this use of GXXF, before the value of the function is computed, the quantity ALPHA* BETA will be substituted for X in the expression defining GXXF.

In general, arithmetic statement functions must obey the following rules:

1) All arithmetic statement functions must be the first executable statements in that program.

2) The function name must have four to nine alphabetic or numerical characters; the first must be alphabetic, and the last must be F.

3) The name of the function is followed by parentheses enclosing the argument or arguments. Multiple arguments are separated by commas. Each argument must be a single nonsubscripted variable.

4) Any argument which is a REAL variable in the definition of a function should be a REAL quantity in any subsequent use of the function. A similar rule applies to INTEGER arguments.

5) The value of a function is a REAL quantity unless the name of the function begins with X; in that case the value is an INTEGER quantity.

The following example illustrates some properties of arithmetic statement functions:

Example 10:

```
 1 FIRSTF(X) = X**2 + A**2
 2 SECONDF(R,S) = SQRTF(FIRSTF(R/(R + S)))
                          .
                          .
                          .
                          .
                          .
15 Q(I) = FIRSTF(Y*B(I))
                          .
                          .
                          .
                          .
                          .
27 P = SECONDF(1.7*DELTA, ALPHA)*PI
```

Notice that it is permissiable to use a previously defined function in the definition of subsequent functions. Notice also that the variable A is involved in the definition of FIRSTF but is not an argument. The variable A may be used in the same way as any other variable in the problem, and its current value is used each time FIRSTF is evaluated.

THE MEANING OF A LIST

Examples of lists have already appeared in input and output statements in this section, although they were not identified as such. A list is a set of items separated by commas; when a list appears in an input or output statement, the order of reading or writing is the order of the items in the list.

For example, the statement

```
TYPE 20, A, B, C
```

has a list A, B, C; the quantities A, B, and C will be typed in that order. If any of the items A, B, or C have been specified in a DIMENSION statement as arrays, then the values of each element of the array will be typed. For example, if A and C are simple variables and B has been specified in a DIMENSION statement as a subscripted variable having 3 elements, then the quantities which would be typed by means of the output statement above are

```
A, B(1), B(2), B(3), C
```

If A and B were large arrays and one wished to specify the reading or writing of the quantities

```
A(1), B(1), A(2), B(2), . . . A(100), B(100)
```

in that order, the list would consist of the single item

(A(I), B(I), I = 1, 100)

If one wished to specify the first seven elements of the array A, followed by the first five elements of the array B, the list would consist of the two items

(A(I), I = 1, 7), (B(I), I = 1, 5)

However, if A and B had dimensions seven and five respectively, the simpler list

A, B

would give the elements of the array A in order A_1, A_2, . . . A_7, and then the elements of the array B in order, B_1, B_2, . . . B_5.

When, as above, an item in a list specifies part of an array or a mixture of arrays, the item must be enclosed in parentheses and the variables inside must be separated by commas as shown. The indexing information (e.g., I = 1, 100) is written exactly as in a DO statement.

FORMAT STATEMENTS

An input or output statement specifies the variables which are to receive values or are to be typed. It also refers to the number of a FORMAT statement which specifies the arrangement of a line of input and/or output data. The FORMAT statement contains the specifications for each field in the line. There are three general forms for a field specification

Iw, Ew.d, Fw.d

where Iw indicates an INTEGER decimal number having a field width of w columns; Ew.d indicates a floating decimal point number (E), having a field width of w columns, and d places to the right of the decimal point; Fw.d indicates a Fixed decimal point number, having a field width of w columns, and d places to the right of the decimal point. (The number d is treated modulo 12.) For example, the statements

```
25 FORMAT (E10.4, F8.3, F7.5, E9.2, I3, F4.1)
   READ PAPER TAPE 25, A, B, C, D, I, E
```

might be used to instruct the computer to read the following input data:

Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
+.8765 E+06	+345.648	+.56872	-2.34 E+01	+81	-1.5
-.1223 E-02	+124.785	-.78963	-6.78 E+09	+15	+9.8
+.1034 E+05	-728.654	+.12345	+4.35 E-07	-28	-2.3

Note that the field width includes a position each for the sign, decimal point, and, in the case of floating decimal point numbers, the four characters of the exponent: the letter E, the sign of the exponent, and the two digits of the exponent. Floating decimal or fixed decimal point numbers may have any number of digits in the input field, depending on specifications in the FORMAT statement; however, only eleven significant digits will be retained in the calculations. If the decimal point punched does not agree with the specification in the FORMAT statement, the decimal point overrides the specification; i.e., calculations will be performed with

the decimal point as placed. If no decimal point is given, the number is treated as if the decimal point were located according to the specification. No provision is made for handling decimal integers larger than 8,338,607. A line of input data may have a maximum of 72 characters. It is recommended that columns 73-80 be used only for cards identification and sequence numbers.

Prior to punching the input data sheets, the positions for each of the fields must be specified. Positions 1-72 are available for use. For this example, the positions for field 1 should be specified as positions 1-10 (since 10 positions are specified by the FORMAT statement for the first field); field 2, positions 11-18; field 3, positions 19-25; field 4, positions 26-34; field 5, positions 35-37; and field 6, positions 38-41.

Specifying a field width larger than the number of characters in the field is particularly valuable for use with output statements.

```
      TYPE 10, A, B, C, D, I, E
10 FORMAT (1H E14.4, F11.3, F10.5, E13.3, I6, F7.1)
```

would cause the data to be typed as follows:

```
 0.8765E 06   345.648  .56872-0.234E 02 81-1.5
-0.1223E-02   124.785 -.78963-0.678E 10 15 9.8
 0.1034E 05  -728.654  .12345 0.435E-06-28-2.3
```

Note the three-position separation between fields (represented by) provided for by the FORMAT statement (10). In the case of floating decimal numbers, the field width includes the zero proceeding the decimal point. Floating decimal numbers are typed with the first significant digit immediately to the right of the decimal point; therefore, these numbers have as many significant digits as there are decimal places specified. However, no more than eleven significant digits are possible. A maximum of 85 characters may be typed per line.

The FORMAT statement is not executed and may be placed anywhere in the program. The field specifications are enclosed in parentheses with commas between the specifications for successive fields. Successive fields having the same format may be specified by inserting a coefficient (indicating the number of identical fields) before the code letter E, F, or I. Thus

```
FORMAT (I3, 2E12.3, F8.4)
```

is equivalent to

```
FORMAT (I3, E12.3, E12.3, F8.4)
```

SAMPLE PROBLEMS

Several examples which illustrate the use of many of the statements introduced in this section appear below.

Problem 1: It is required to calculate the amount of heat necessary to raise the temperature of a mixture of ten gases from a given base temperature, T_1, to a series of higher temperatures. These temperatures are 25 degrees apart and range from T_1 up to a maximum of T_2.

The heat required may be calculated by multiplying the heat capacity of the gas mixture by the temperature difference. However, the heat capacity is dependent upon the temperature. The mean heat capacity over a given range may be estimated by using the equation

$$C_p = a + \frac{b}{2}(T + T_0) + \frac{c}{3}(T^2 + TT_0 + T_0^2)$$

where

C_p = the mean heat capacity

T = the upper temperature, degrees Kelvin

T_0 = the lower temperature, degrees Kelvin

a, b, c = empirical constants, different for each gas

(degrees Kelvin = degrees Centigrade + 273.1)

Input data to the program must therefore include the amount of each gas present (x), the three empirical constants (a, b, c) for each gas, the base temperature (T_1, in °C), and the maximum temperature (T_2, in °C).

A possible FORTRAN program to carry out this calculation appears below. It has been written to provide the individual heat capacities in each range as well as the total heat requirement. The program incidentally illustrates the fact that statements need not be numbered sequentially.

Example 11:

```
 9 DIMENSION X(10), A(10), B(10), C(10), CP(10)
10 FORMAT(6F12.8)
11 FORMAT(6E12.8)
12 READ PAPER TAPE 10, X, A, T1, T2
13 READ PAPER TAPE 11, B, C
14 T1K = T1 + 273.1
15 TK = T1K
16 TK = TK + 25.0
17 SUM = 0.0
18 IF((TK - 273.1) - T2) 19, 19, 27
19 DO 21 I = 1, 10
20 CP(I) = A(I) + B(I) * (TK + T1K) / 2.0 + C(I) * (TK**2 + TK
  X        *T1K + T1K**2) / 3.0
21 SUM = X(I) * CP(I) + SUM
22 HEAT = SUM * (TK - T1K)
23 T = TK - 273.1
24 TYPE 31, T1, T, HEAT
25 TYPE 32, X
36 TYPE 34, CP
26 GO TO 16
27 IF(T2 - 2500.) 12, 33, 33
31 FORMAT(1H 2F10.1, E15.5)
32 FORMAT(1H 10F8.3)
34 FORMAT(1H 5E14.5)
33 CALL EXIT
   STOP
   END
```

The DIMENSION statement sets aside storage locations for the constants and results. The FORMAT statements 10 and 11 describe the way the input data should be converted when it is read into the program.

FORMAT statements 31, 32, and 34 describe the way in which the output data should be typed.

Statements 12 and 13 cause the data for a case to be transferred into the computer from the paper tape reader. The calculation of the absolute temperature in degrees Kelvin from the base temperature is carried out by statement 14. Statement 15 sets the initial value of the temperature range to T_1 °K. Statement 16 causes the range to be increased by the specified increment. Statement 17 sets the location designated as SUM to zero. The upper limit of the range is compared to the maximum temperature specified for this case. If the maximum has not been reached, control reaches the DO statement (statement 19). The statements in the range of the DO (statements 20 and 21) cause the specific heat of each component to be calculated and weighted according to the fraction of that component in the mixture. The actual calculation of the heat requirement is described by statement 22. Statement 23 causes the upper limit of the range to be expressed in degrees Centigrade. Typing of the results, along with the fractions of each component, on the typewriter is accomplished by statements 24, 25, and 36. A transfer to begin the calculation for the next range is effected by statement 26.

If the comparison at statement 18 indicates that the maximum temperature for the given case has been exceeded, control reaches statement 27. At this point, the maximum temperature is examined to determine whether it exceeds 2500°C (which is the indication that the problem has been completed). If the problem has been completed, control reaches statement 33; the computer types "END OF JOB" and halts. If the problem has not been completed, control is transferred to statement 12, which causes data for a new case to be read from the input tape.

Problem 2: Given X_i, Y_i, Z_j for i = 1, . . ., 10 and j = 1, . . ., 20, compute:

$$\text{PROD} = \left(\sum_{i=1}^{10} A_i\right) \left(\sum_{j=1}^{20} Z_j\right)$$

where

$$A_i = X_i^2 + Y_i \quad \text{if} \quad |X_i| > |Y_i|$$

$$A_i = X_i + Y_i^2 \quad \text{if} \quad |X_i| < |Y_i|$$

$$A_i = 0 \quad \text{if} \quad |X_i| = |Y_i|$$

A possible FORTRAN program follows for this problem:

Example 12:

```
 3 DIMENSION X(10), Y(10), Z(20)
 4 FORMAT(6F12.8)
 5 READ PAPER TAPE 4, X, Y, Z
 6 SUMA = 0.0
 7 DO 12 I = 1, 10
 8 IF(ABSF(X(I)) - ABSF(Y(I))) 9, 12, 11
 9 SUMA = SUMA + X(I) + Y(I)**2
10 GO TO 12
11 SUMA = SUMA + X(I)**2 + Y(I)
12 CONTINUE
13 SUMZ = 0.0
14 DO 15 J = 1, 20
15 SUMZ = SUMZ + Z(J).
16 PROD = SUMA * SUMZ
17 TYPE 18, SUMA, SUMA, PROD
18 FORMAT(1H 3E17.8)
19 GO TO 5
   END
```

The DIMENSION statement sets aside storage locations for the input data. Statement 4 specifies the input data as fixed point numbers having 8 decimal places. The READ PAPER TAPE statement reads the input data into the computer. Statement 6 sets the quantity SUMA to zero. Statements 8-12, under control of the DO statement 7,

compute $\sum_{i=1}^{10} A_i$. Statement 15 computes $\sum_{j=1}^{20} Z_j$ under the

control of DO statement 14. Statements 15 and 16 compute PROD. Statement 12, CONTINUE, serves as a common reference point; and since it is the last statement in the range of the DO, I is increased after its completion and the next repetition is begun.

CHECK LIST

1) All subscripted variables must appear in a DIMENSION statement. This statement must appear in the program before the first executable statement.

2) Negative subscripts are not permitted.

3) Subscripting of subscripts is not permitted.

4) INTEGER constants are written without a decimal point; INTEGER variables must begin with I, J, K, L, M, or N unless the variable has been declared in an INTEGER specification statement (see Section

5) The names of all functions defined in the program or available on the FORTRAN library tape, as well as those names without the terminal F, must not be used as variable symbols. (Although the terminal F is, by definition, part of the name, the FORTRAN system does not use it throughout the compilation and execution of the job.) If SINF is a function, neither SINF nor SIN may be variable symbols.

6) If a subscripted variable has 4 or more characters in its name, the last of these must not be an F. For example, PREF(I) cannot be used as a subscripted variable, regardless of whether or not PREF is used as the name of a function.

7) The last statement in the range of a DO must not be a transfer.

8) Decimal integers larger than 8,388,607 are not permitted, and generally will be handled incorrectly by FORTRAN statements.

9) No constants may be given in a list for an input/output statement, only variables.

10) Every output statement should refer to a FORMAT statement given in the program. For single-spaced typing, each output format specification should begin with the characters "1H" followed by a blank space.

MANIPULATION OF TWO- AND THREE-DIMENSIONAL ARRAYS

INTRODUCTION

The following is a list of the 12 types of statements, grouped into classifications, which have been presented thus far.

Arithmetic Statements	
Input-output statements	(READ PAPER TAPE (TYPE (FORMAT
Control statements	(IF (Unconditional GO TO (STOP (END (DO (CONTINUE
Specification statement	DIMENSION
Subprogram facility statement	CALL EXIT

Several of the statements introduced in the last subsection offered a convenient method for handling one-dimensional arrays in an iterative manner. In the present subsection, subscripting to handle two- and three-dimensional arrays will be described. This provision greatly facilitates the solving of many engineering and scientific problems which require matrix manipulations for their solution. In addition, several new statements will be introduced.

The following example of matrix multiplication will serve to illustrate DO nests and multiple subscripts. (A DO nest is a set of two or more DO statements, the range of one of which includes the ranges of the others.)

Given the matrix A with dimensions 10 x 15 and the matrix B with dimensions 15 x 12, compute the elements C_{ij} of the matrix C = AB. To compute any element C_{ij}, select the i row of A and the j column of B, and sum the products of their corresponding elements. The general formula for this computation is

$$C_{ij} = \sum_{k=1}^{15} A_{ik}B_{kj}$$

The following is a possible FORTRAN program for this matrix multiplication.

Example 13:

```
   DIMENSION A(10,15), B(15,12), C(10.12)
 2 FORMAT(5E14.5)
 3 READ PAPER TAPE 2, A, B
 4 DO 30 I = 1, 10                          Range of 1st DO
 5 DO 30 J = 1, 12                          Range of 2nd DO
 6 C(I, J) = 0.0
10 DO 20 K = 1, 15                          Range of 3rd DO
20 C(I, J) = C(I, J) + A(I, K) * B(K, J)
30 TYPE 50, I, J, C(I, J)
50 FORMAT(1H 2I5, E16.7)
60 GO TO 3
   END
```

The DIMENSION statement says: "Matrix A is of size 10 x 15, matrix B is of size 15 x 12, and matrix C is of size 10 x 12." The READ PAPER TAPE statement reads all elements of the matrix A, and then all elements of the matrix B into the computer; the format is specified by statement 2. Since the two dimensional arrays are stored row-wise, the matrices A and B must be punched row-wise; i.e., all the A_i of row 1, followed by all the A_i of row 2, etc. ($A_{1,1}$, $A_{1,2}$, $A_{1,3}$, . . ., $A_{10,15}$), and similarly for matrix B.

Notice that statements 6 through 30 constitute a program similar to programs considered previously. Whatever values I and J have at the moment, this program computes and types C(I, J) along with I and J.

Statement 5 says that this program is to be repeated 12 times, first for J = 1, then for J = 2, . . ., J = 12. Notice that for each repetition of statements 6 through 30, statement 20 is executed 15 times, first for K = 1, then for K = 2, and so on. Thus, when the process called for by statement 5 is complete, the I row of the product matrix has been computed and typed. Control then returns to statement 4 to obtain a new value of I, and statements 5 through 30 are repeated for this value. The process continues until all of the rows of the product matrix are produced.

This example illustrates the fact that one or more DO statements may appear in the range of a DO statement. This nesting of DO statements can result in a single statement being the last statement in the range of several DO statements. For example, statement 30 is the last one in the range of DO statements 4 and 5. Consequently, a more general rule is needed to describe the flow of control and the incrementing of indices following the last statement in the range of a DO; the following rule holds for DO ranges which have the same last statement:

> Upon the completion of the last statement in the range of a DO, control passes to the first statement in the range of the nearest preceding DO which is not yet completed and which has the same last statement, and the index of that DO is incremented. The last statement in the range of a DO may not be a control statement (e.g., IF, GO TO, DO, etc.). If all DO ranges containing this last statement as the end of their range are completed, control passes to the next statement.

SUBSCRIPTS FOR TWO- AND THREE-DIMENSIONAL ARRAYS

In the preceding example of matrix multiplication, A, B, and C were two-dimensional arrays. As was noted, each variable had two subscripts which were separated by commas, and the set of two subscripts was enclosed in parentheses. For example:

```
A(I, K)

B(K, J)

C(I, J)
```

Three-dimensional arrays are denoted by the use of three subscripts. For example:

```
X(M, N + 10, 5*L)
```

The same rules already presented regarding the information of subscripts apply to the two- and three-dimensional cases.

The DIMENSION statement is similarly extended to two- or three-dimensional arrays. For example, the statement

DIMENSION W(10, 10, 15), ALPHA (15, 5), V(20, 10)

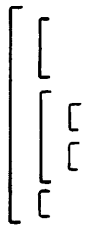
causes 3000 locations in storage to be set aside for the three-dimensional array W, 150 locations for the two-dimensional array ALPHA, and 400 locations for the two-dimensional array V (each floating-point number in DDP-24 FORTRAN requires two storage location).

DO NESTS

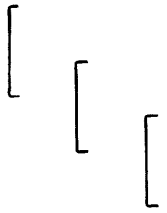
There are certain rules which must be observed when using DO statement within the range of another DO statement:

1) If the range of a DO statement includes another DO statement, all statements in the range of this second statement must also be in the range of the first DO statement. The following diagram illustrates this rule.

Permitted



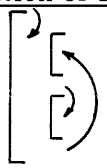
Violation of Rule 1



2) No transfer of control by IF or GO TO statements is permitted into the range of any DO statement from outside its range, since such transfers would not permit the DO loop to be properly indexed. The following diagram illustrates this rule.

Permitted

Violation of Rule 2



All the DO statements presented so far were written in the form

DO N I = m_1, m_2

In these cases, the index, I, started at the specified value, m_1, and was increased by one each time the statements in the range of the DO were executed, until the value of I equaled m2. It is possible, however, to achieve greater flexibility in the DO statements by adding a third expression so that the general form is

DO N I = m_1, m_2, m_3

In this case, the value of the index, I, starts at m_1 (as before), but it is increased by m_3 (which may be different from one) each time, until the value of I equals or exceeds m_2, at which point the DO is satisfied. It is not necessary to include the increment, m_3, in the DO statement unless the increment is different from one; i.e., the statements

```
DO 20 I = 1, 10
```

and

```
DO 20 I = 1, 10, 1
```

are equivalent.

Every type of calculation is permitted in the range of a DO with one exception. No calculation which changes the value of the index, I, or any of the indexing parameters (m_1, m_2, m_3) is permitted within the range of that DO statement. The indexing parameters (m_1, m_2, m_3) may be either INTEGER or REAL expressions (REAL numbers will be truncated to INTEGERS).

LISTS FOR TWO- AND THREE-DIMENSIONAL ARRAYS

The extension of the input-output statements to govern the transfer of two- and three-dimensional arrays to or from magnetic core storage requires only that the subscripting information given earlier be used when writing the list. If the list

```
JOBNO, CASE, RUN, K, (X(I), Y(I, K), I = 1, 4),
      ((Z(I, J), I = 1, 3), W(J, 3), J = 1, 3)
```

were used with an input statement, the successive elements, as they were read into the computer, would be interpreted as the following sequence of variables and placed in the storage locations (previously assigned by FORTRAN) in that same order:

```
JOBNO, CASE, RUN, K, X(1), Y(1, K), X(2), Y(2, K),
X(3), Y(3, K), X(4), Y(4, K), Z(1, 1), Z(2, 1), Z(3, 1),
W(1, 3), Z(1, 2), Z(2, 2), Z(3, 2), W(2, 3), Z(1, 3),
Z(2, 3), Z(3, 3), W(3, 3)
```

Note that a variable subscript (K) was used at one point. This is permissible only if that variable has been assigned a value previously (in this case, a value would have been read in earlier).

To transfer a complete array, subscripting and index information is not necessary. Such information is provided, in this case, by the DIMENSION statement. Using the example above, the statements

```
DIMENSION ALPHA (5, 15)
READ PAPER TAPE 6, ALPHA
```

would cause the entire 75 element array

```
ALPHA(1, 1), ALPHA(1, 2), ALPHA(1, 3), ALPHA(1, 4), . . .,
ALPHA(1, 15), ALPHA(2, 1), ALPHA(2, 2), ALPHA(2, 3),
ALPHA(2, 4), . . ., ALPHA(5, 15).
```

to be transferred into magnetic core storage.

ASSIGNED GO TO STATEMENTS

One modification of the GO TO statement which allows greater freedom in directing the logical flow of a program is the assigned GO TO statement. The assigned GO TO statement requires a companion statement, an ASSIGN statement, which must be previously executed.

As an example of the use of the assigned GO TO statement, suppose it is desired to calculate several average values such as average temperature, pressure, and density. The following program might be used:

Example 14:

```
      DIMENSION X(25)
    5 ASSIGN 30 TO N
   10 READ PAPER TAPE 60, X
      SUM = 0.0
   15 DO 20 I = 1, 25
   20 SUM = SUM + X(I)
   25 AVG = SUM / 25.0
   26 GO TO N, (30, 40, 50)
   30 AVGTEM = AVG
   31 ASSIGN 40 TO N
      GO TO 10
   40 AVGPRE = AVG
   41 ASSIGN 50 TO N
      GO TO 10
   50 AVGDEN = AVG
      TYPE 61, AVGTEM, AVGPRE, AVGDEN
      CALL EXIT
      STOP
   60 FORMAT(6E12.8)
   61 FORMAT(1H 6E17.8)
      END
```

In this example, statement 26 transfers control to one of the three statements referred to in the list, i.e., 30, 40, or 50, depending upon the value of N at the time of execution,which is determined by the last preceding ASSIGN statement. The first execution of statement 26 causes control to be transferred to statement 30, since statement 5, the last preceding ASSIGN statement, assigned the value of 30 to N. Statement 31 assigns the value of 40 to N; hence the second execution of statement 26 transfers control to statement 40. The third execution of statement 26 transfers control to statement 50, the value of 50 having been assigned to N by statement 41.

In general terms, the assigned GO TO statement is written

GO TO N, (n_1, n_2, . . ., n_m)

where N is a variable appearing in a previously executed ASSIGN statement, and n_1, n_2, . . ., n_m stand for statement numbers. These statement numbers are, in effect, a list of values which may be assigned to N. Note the comma which is inserted between the variable and the left parenthesis; it must always be included.

The statement

ASSIGN 30 TO N

is not equivalent to the arithmetic formula

N = 30

A variable N which currently has a value is either an assigned variable or an ordinary variable, never both simultaneously. It is an assigned variable if its current value has been established by an ASSIGN statement (e.g., ASSIGN 30 TO N); it is an ordinary variable if its current value has been established by an arithmetic formula (e.g., N = 30). The current value is the one given by the last previous ASSIGN statement or arithmetic formula, whichever was

most recently executed. A variable N which is currently an assigned variable is effective only in assigned GO TO N statements. An ordinary variable N is effective in all statements involving N except GO TO N statements.

No transfer of control by an assigned GO TO statement is permitted into the range of any DO statement from outside its range, since such transfers would not permit the DO loop to be properly indexed. (See diagram on page .) This means that the statements to which the assigned GO TO statement may transfer may be (1) statements within the same DO range as the assigned GO TO, or (2) statements outside the range of any DO. Types (1) and (2) cannot be mixed in the same assigned GO TO; they must be all of type (1) or all of type (2).

If this condition cannot be met, it may be possible by suitable programming changes to use a computed GO TO to accomplish the desired branching, since there is no such restriction on this type of statement.

COMPUTED GO TO STATEMENTS

Computed GO TO statements are similar to assigned GO TO statements in that both types establish a many-way fork. They differ in that an assigned GO TO statement requires a companion statement (ASSIGN) to pre-set or assign a current value to the variable in the GO TO statement in order to select the proper branch. The value of the variable in a computed GO TO statement may be arrived at by computation; no companion statement (comparable to ASSIGN) is necessary.

Problem:

Given: A_i, B_i, N_i, X_i, Y_i for i = 1, . . ., 10, where,

for each i, N_i = 1 or 2, compute

$$Z_i = \sqrt{A_i X_i^2 + B_i Y_i} \qquad \text{for } N_i = 1$$

$$Z_i = \sqrt{A_i X_i^2 - B_i Y_i} \qquad \text{for } N_i = 2$$

A possible FORTRAN program follows:

Example 15:

```
   DIMENSION A(10), B(10), N(10), X(10), Y(10), Z(10)
 2 READ PAPER TAPE 3, (A(I), B(I), N(I), X(I), Y(I), I = 1, 10)
 3 FORMAT(2E13.5, I3, 2E13.5)
 5 DO 21 I = 1, 10
 7 GO TO (10, 20), J(I)
10 Z(i) = SQRTF(A(I) * X(I)**2 + B(I) * Y(I))
11 GO TO 21
20 Z(I) = SQRTF(A(I) * X(I)**2 - B(I) * Y(I))
21 TYPE 23, A(I), B(I), N(I), X(I), Y(I), Z(I)
23 FORMAT(1H 2E13.5, I3, 3E13.5)
   GO TO 2
   END
```

In this program, statement 7 transfers control to statement 10 if J = 1 or to statement 20 if J = 2. The ten values of N_i read into the program are each either 1 or 2. Since J is set equal to N_i by statement 6, the correct formula for Z_i is selected, depending on whether the current value of N_i is 1 or 2.

As illustrated in the program of Example 15, computed GO TO statements have the form

GO TO (n_1, n_2, . . ., n_m), I

where the n_1, n_2, . . ., n_m stand for statement numbers, and I is a variable. Control is transferred to the first statement in the list (statement n_1), if, at the time of execution, the value of I is one; it is transferred to the second statement in the list (statement n_2) if the value of I is two, etc. Any number of statement numbers may appear in the list. The current value of I may be arrived at in any manner desired (e. g., in the program above, by an arithmetic formula modified by DO indexing), and its value at the time of execution of the computed GO TO statement determines which branch will be taken by the program.

Note the comma which is inserted between the right parenthesis and the variable.

FORMAT STATEMENTS

In an earlier subsection the basic field specifications Iw, Ew. d, and Fw. d were introduced. In the present subsection, scale factors, Hollerith fields and multiple-line formats will be discussed.

SCALE FACTORS. The use of scale factors allows greater flexibility in an output format. The specification

(2E14. 4)

might type the following output line (^ stands for blank space):

-0. 4321E^04^^^^0. 5674E-06

If the specification were written as

(2P2E14. 4)

the same output data would be typed with six significant digits, with the decimal point four places from the right. For example, the output data shown above might type as

-43. 2147E 02^^^^56. 7439E-08

The scale factor 2P causes the floating-point number to be multiplied by 10^2, and the exponent to be reduced by 2 prior to typing (i. e., the <u>value</u> has not been changed).

Only a positive scale factor may be used with an E-type specification. However, positive or negative scale factors may be used with an F-type specification. For example, the specification

(-1PF10. 3, 7PF12. 3)

would type the following data

-4321. 47 .0000005674

as

-432. 148 5. 674

Note that for F conversions, the <u>value</u> is changed.

If it is desired to specify a scale factor of zero subsequent,to another scale factor within the same FORMAT statement, 0P must be written. For example, the specification

(1PF10.1, F12.9)

would type the preceding data as

-43214.7 .000005674

The same data would be typed by the specification

(1PF10.1, 0PF12.9)

as

-43214.7 .000000567

The scale factor has no effect on I-conversion.

HOLLERITH FIELDS: English test may be typed by specifying a Hollerith field. Such fields are designated by the letter H preceded by a number designating the number of characters in the text; the field designation is followed by the desired English characters (including blanks). The characters desired are written in a FORMAT statement in the program. For example, to type the factors X and Y along with their product, the FORMAT statement

```
10      FORMAT (3H␣X= F8.3, 4H␣␣Y= F8.3, 5H␣␣XY= F8.3)
```

could be used to type the output line

X=␣␣10.723␣␣Y=␣-12.561␣␣XY=-134.692

The symbol ␣ may be used to indicate blanks on the FORTRAN code sheet, or the colums may simply be left blank.

Note that there is no comma after a Hollerith field specification (e.g., 4H␣␣) in the FORMAT statement.

It is also possible to read in or type out alphanumerical characters as data, using the A-conversion.

MULTIPLE-LINE FORMATS: In our discussion up to this point, two FORMAT statements would have been necessary to type the following lines of output data:

␣-67.8912E-03␣␣106.23␣␣-73

␣␣␣␣␣␣732␣␣␣␣␣␣82.976␣6.25

Two suitable FORMAT statements to type these two lines are:

```
10      FORMAT (1H 2PE13.4, 0PF8.2, I5)
11      FORMAT (1H I9, F12.3, F5.2)
```

It is possible to use one FORMAT statement to type multiple lines, each with a different format, by using a slash (/) to separate the formats for the different lines. One FORMAT statement that would type both the preceding example lines is

```
12      FORMAT (2PE13.4, 0PF8.2, I5/I9, F12.3, F5.2)
```

If a series of lines were typed using this FORMAT statement, lines 1, 3, 5,. . . would have the format (2PE13.4, 0PF8.2, I5), and lines 2, 4, 6, . . . would have the format (I9, F12.3, F5.2).

SAMPLE PROBLEM AND PROGRAM

The following example illustrates the use of many of the types of instructions presented in the three sections of this primer.

Use the least-squares method to find the m degree polynomial

$$y \quad a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m$$

which approximates a curve through n given points (x_i, y_i).

In order to obtain the coefficients $a_0, a_1, \ldots\ldots, a_m$, it is necessary to solve the normal equations

(1) $S_0 a_0 + S_1 a_1 + \ldots + S_m a_m = V_0$

(2) $S_1 a_0 + S_2 a_1 + \ldots + S_{m+1} a_m = V_1$

. . . .
. . . .
. . . .

(m + 1) $S_m a_0 + S_{m+1} a_1 \ldots + S_{2m} a_m = V_m$

where

$$S_0 = n \qquad V_0 = \sum_{i=1}^{n} y_i$$

$$S_1 = \sum_{i=1}^{n} x_i \qquad V_1 = \sum_{i=1}^{n} y_i x_i$$

$$S_2 = \sum_{i=1}^{n} x_i^2 \qquad V_2 = \sum_{i=1}^{n} y_i x_i^2$$

. .
. .
. .

$$S_{2m} \sum_{i=1}^{n} x_i^{2m} \qquad V_m \sum_{i=1}^{n} y_i x_i^m$$

After the S's and V's have been computed, the normal equations are solved by the method of elimination, which is illustrated by the following solution of the normal equations for a second degree polynomial (m 2).

(1) $S_0 a_0 + S_1 a_1 + S_2 a_2 = V_0$

(2) $S_1 a_0 + S_2 a_1 + S_3 a_2 = V_1$

(3) $S_2 a_0 + S_3 a_1 + S_4 a_2 = V_2$

The forward solution is as follows:

1) Divide equation (1) by S_0.

2) Multiply the equation resulting from step 1 by S_1 and subtract from equation 2.

3) Multiply the equation resulting from step 1 by S_2 and subtract from equation 3.

The resulting equations are

(4) $$a_0 + b_{12}a_1 + b_{13}a_2 = b_{14}$$

(5) $$b_{22}a_1 + b_{23}a_2 = b_{24}$$

(6) $$b_{32}a_1 + b_{33}a_2 = b_{34}$$

where

$$b_{12} = \frac{S_1}{S_0}, \qquad b_{13} = \frac{S_2}{S_0}, \qquad b_{14} = \frac{V_0}{S_0}$$

$$b_{22} = S_2 - b_{12}S_1, \qquad b_{23} = S_3 - b_{13}S_1, \qquad b_{24} = V_1 - b_{14}S_1$$

$$b_{32} = S_3 - b_{12}S_2, \qquad b_{33} = S_4 - b_{13}S_2, \qquad b_{34} = V_2 - b_{14}S_2$$

Steps 1 and 2 are repeated, using equations (5) and (6), with b_{22} and b_{32} instead of S_0 and S_1. The resulting equations are

(7) $$a_1 + c_{23}a_2 = c_{24}$$

(8) $$c_{33}a_2 = c_{34}$$

where

$$c_{23} = \frac{b_{23}}{b_{22}}, \qquad c_{24}\ \frac{b_{24}}{b_{22}}$$

$$c_{33} = b_{33} - c_{23}b_{32}$$

$$c_{34} = b_{34} - c_{24}b_{32}$$

The backward solution is:

(9) $$a_2 = \frac{c_{34}}{c_{33}}$$ from equation (8)

(10) $$a_1 = c_{24} - c_{23}a_2$$ from equation (7)

(11) $$a_0 = b_{14} - b_{12}a_1 - b_{13}\ a_2$$ from equation (4)

The following is a possible FORTRAN program for carrying out the calculations for the case: n = 100, $m \leq 10$. S_0, S_1, S_2, . . ., S_{2m} are stored in SUM(1), SUM(2), SUM(3), . . ., SUM(2M + 1), respectively. V_0, V_1, V_2, . . ., V_m are stored in V(1), V(2), V(3), . . ., V(M + 1), respectively.

Example 16:

```
      DIMENSION X(100), Y(100), SUM(21), V(11), A(11), B(11,12)
      READ PAPER TAPE 3, M, N
    3 FORMAT (I2, I3)
      READ PAPER TAPE 4, (X(I), Y(I), I 1, N)
    4 FORMAT (4E14.7)
      LS = 2*M + 1
      LB = M + 2
      LV = M + 1
      DO 5 J = 2, LS
    5 SUM(J) = 0.0
      SUM(1) = N
      DO 6 J = 1, LV
    6 V(J) = 0.0
      DO 16 I = 1, N
      P = 1.0
      V(1) = V(1) + Y(I)
      DO 13 J = 2, LV
      P = X(I)*P
      SUM(J) = SUM(J) + P
   13 V(J) = V(J) + Y(I)*P
      DO 16 J = LB, LS
      P = X(I)*P
   16 SUM(J) = SUM(J) + P
   17 DO 20 I = 1, LV
      DO 20 K = 1, LV
      J = K + 1
   20 B(K,I) = SUM(J -1)
      DO 22 K = 1, LV
   22 B(K, LB) = V(K)
   23 DO 31 L = 1, LV
      DIVB = B(L, L)
      DO 26 J = L, LB
   26 B(L, J) = B(L, J)/DIVB
      I1 = L + 1
      IF (I1 - LB) 28, 33, 33
   28 DO 31 I = I1, LV
      FMULTB = B(I, L)
      DO 31 J = L, LB
   31 B(I, J) = B(I, J) - B(L, J)*FMULTB
   33 A(LV) = B(LV, LB)
      I = LV
   35 SIGMA = 0.0
      DO 37 J = I, LV
   37 SIGMA = SIGMA + B(I -1, J)*A(J)
      I = I -1
      A(I) = B(I, LB) - SIGMA
   40 IF (I -1) 41, 41, 35
   41 TYPE 42, (A(I), I = 1, LV)
   42 FORMAT (1H 5E15.6)
      CALL EXIT
      END
```

The elements of the SUM and V arrays, except SUM(1), are set equal to zero. SUM(1) is set equal to N. For each value of I, X_I and Y_I are selected. The powers of X_I are computed and accumulated in the correct SUM counters. The powers of X_I are multiplied by Y_I and the products are accumulated in the correct V counters. In order to save machine time when the object program is being run, the previously computed power of X_I is used when computing the next power of X_I. Note the use of variables as index parameters. By the time control has passed to statement 17, the counters have been set as follows:

$$SUM(1) = N \qquad V(1) = \sum_{I=1}^{N} Y_I$$

$$SUM(2) = \sum_{I=1}^{N} X_I \qquad V(2) = \sum_{I=1}^{N} Y_I X_I$$

$$SUM(3) = \sum_{I=1}^{N} X_I^2 \qquad V(3) = \sum_{I=1}^{N} Y_I X_I^2$$

$$\vdots \qquad\qquad \vdots$$

$$SUM(2M+1) = \sum_{I=1}^{N} X_I^{2M} \qquad V(M+1) = \sum_{I=1}^{N} Y_I X_I^{M}$$

By the time control has passed to statement 23, the values of $S_0, S_1, \ldots S_{2m}$ have been placed in the storage locations corresponding to columns 1 through M + 1, rows 1 through M + 1, of the B array, and the values of $V_0, V_1, \ldots, V_m$ have been stored in the locations corresponding to column M + 2, rows 1 through M + 1, of the B array. For example, for the sample problem (M = 2), columns 1 through 4, rows 1 through 3, of the B array would be set to the following computed values:

$$\begin{matrix} S_0 & S_1 & S_2 & V_0 \\ S_1 & S_2 & S_3 & V_1 \\ S_2 & S_3 & S_4 & V_2 \end{matrix}$$

This matrix represents equations (1), (2), and (3), the normal equations for M = 2. The forward solution, which results in equations (4), (7), and (8) in the sample problem, is carried out by statements 23 through 31. By the time control has passed to statement 33, the coefficients of the A_I terms in the M + 1 equations which would be obtained in hand calculations have replaced the contents of the locations corresponding to columns 1 through M + 1, rows 1 through M + 1, of the B array, and the constants on the right-hand side of the equations have replaced the contents of the locations corresponding to column M + 2, rows 1 through M + 1, of the B array. For the problem, columns 1 through 4, rows 1 through 3, of the B array would be set to the following computed values:

$$\begin{matrix} 1 & b_{12} & b_{13} & b_{14} \\ 0 & 1 & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \end{matrix}$$

This matrix represents equations (4), (7), and (8) on page 2-37.

The backward solution, which results in equations (9), (10), and (11) in the problem, is carried out by statements 33 through 40. By the time control has passed to statement 41, which prints the values of the A_I terms, the values of the M + 1 A_I terms have been stored in the M + 1 locations for the A array. For the problem, the A array would contain the following computed values for a_2, a_1, and a_0, respectively:

Location	Contents
A(3)	$\frac{c_{34}}{c_{33}}$
A(2)	$c_{24} - c_{23}a_2$
A(1)	$b_{14} - b_{12}a_1 - b_{13}a_2$

The resulting values of the A_I terms are then typed according to the FORMAT specification in statement 42.

DEBUGGING

In order to debug a FORTRAN program, it is recommended that extra output statements under the control of a piece of data be used. For example, the value of DT can indicate whether or not extra typing is desired during the execution of the program. If DT $\neq$ 0, extra typing will occur which is helpful during the checkout period. If DT = 0, no extra typing will occur.

The following problem illustrates the use of data as an aid in debugging a program.

Problem:

Given: a_i, b_i and c_i for i = 1, . . ., 10, compute and type

$$\text{RESULT} = \sum_{i=1}^{10} (a_i c_i)^2 \qquad \sum_{i=1}^{10} (b_i - c_i) \qquad \sum_{i=1}^{10} (a_i b_i - c_i^2)$$

Assume that the following FORTRAN program has been written and compiled (i. e., translated into machine language by the computer by means of the FORTRAN system) and is to be tested:

Example 17:

```
      DIMENSION A(10), B(10), C(10)
      SUM1 = 0.0
      SUM2 = 0.0
      SUM3 = 0.0
    3 READ PAPER TAPE 20, DT, (A(I), B(I), C(I), I = 1, 10)
      DO 10 I = 1, 10
      SUM1 = SUM1 + (A(I)*C(I))**2
      SUM2 = SUM2 + B(I) - C(I)
      SUM3 = SUM3 + A(I)*B(I) - C(I)
      IF (DT) 5, 10, 5
    5 TYPE 25, SUM1, SUM2, SUM3
   10 CONTINUE
      RESULT = SUM1*SUM2/SUM3
      TYPE 25, RESULT
```

```
20 FORMAT (6F12.8)
25 FORMAT (1H 6E17.8)
   GO TO 3
   END
```

A test case is run using the compiled program. Assume the test case has the following input data

$a_1 = -.23456$ $b_1 = 12.3411$ $c_1 = 27.86523$

Then the first line of output is

42.72019 -15.52412 -30.75996

Hand calculations for i = 1, using the original formula for RESULT, show that

SUM1 = 42.72019

SUM2 = -15.52412

SUM3 = -779.36577

The hand-computed results for SUM1 and SUM2 agree with the output results; however, SUM3 results do not agree. By examining the FORTRAN statement which computes SUM3, the error is located. The statement is changed from

```
SUM3 = SUM3 + A(I)*B(I) - C(I)
```

to

```
SUM3 = SUM3 + A(I)*B(I) - C(I)**2
```

After the indicated change is made, the FORTRAN program is again compiled and the test re-run. This time the machine results agree with the hand-computed results for all three sums. In the future runs, the additional typing will not be required, so DT will be set = 0.

MASTER CHECK LIST

1) The basic characters which may be used in writing a FORTRAN statement are

a) A, B, C, . . ., Z (26 alphabetic characters, capital letters only)

b) 0, 1, 2, . . ., 9 (10 numerical characters)

c) + (plus); - (minus); * (asterisk); / (slash); ((left parenthesis);) (right parenthesis; , (comma); = (equal sign); . (decimal point); and ' (apostrophe).

2) A variable symbol can consist of eight or less characters. It must satisfy the following conditions:

a) The first character must be alphabetic.

b) The first character is usually not I, J, K, L, M, or N, unless the symbol is an INTEGER variable, representing a fixed point INTEGER. Symbols for INTEGER variables normally begin with I, J, K, L, M, or N (see specification statements).

c) Any character following the first may be alphabetic or numeric, but not one of the special characters.

d) The names of all functions defined in the program or available on the FORTRAN library tape, as well as these names without the terminal F, must not be used as variable symbols. For example, since SINF is the name of a function, neither SINF nor SIN can be used as a variable symbol. (Although the terminal F is, by definition, part of the name, the FORTRAN system does not use it throughout the compilation and execution of the job.

e) If a subscripted variable has four or more characters in its name, the last of these must not be an F. For example, PREF(I) cannot be used as a subscripted variable.

3) The name used for a function in programming must agree exactly with the name appearing in the list of functions.

4) The argument of a function is enclosed in parentheses; e.g., SINF(X).

5) If a function has more than one argument, the arguments are separated by commas; e.g., TRAGF (X, Y, Z).

6) The left side of an arithmetic formula must always be a variable or a function of one or more variables.

7) Never omit the operation symbol between two quantities; e.g., do not write AB for A*B.

8) Never have two operations symbols in a row; e.g., do not write A*-B for A*(-B). The exponentiation symbol ** may appear to be an exception, but it is regarded as a single symbol.

9) Blank spaces can be used or not used as desired, since blanks are ignored except as specified in Hollerith fields within FORMAT statements.

10) The prescribed form for the various non-arithmetic statements must be followed exactly except for the arbitrary use of blank spaces.

11) The magnitude of every non-zero REAL quantity must lie between 10^{-75} and 10^{75}. By "quantity" is meant any constant or any value assumed by a variable or function in the course of the calculation.

12) The program should provide for a proper termination; this can be a return to read more data.

13) All subscripted variables must appear in a DIMENSION statement, which must appear in the program before the first executable statement.

14) Subscripts for two- and three-dimensioned arrays should be separated by commas.

15) INTEGER constants are written without a decimal point.

16) Decimal INTEGERS larger than 8,388,607 are not permitted; in general, they will be handled incorrectly by FORTRAN statements.

17) If the range of a DO includes another DO, then all statements in the range of this second DO must also lie within the range of the first DO.

18) Transfers into the range of any DO from outside its range are not permitted.

19) The first statement in the range of a DO must be an executable statement. The last statement in the range must not be a transfer.

20) No calculation which changes the index or indexing parameters of a DO is permitted within the range of that DO.

21) Assigned GO TO statements have a comma between the variable and the left parenthesis.

22) Computed GO TO statements have a comma between the right parenthesis and the variable.

23) An ASSIGN statement must be encountered by the program prior to encountering an assigned GO TO statement.

24) The ASSIGN statement is not equivalent to an arithmetic formula.

25) When an assigned GO TO lies in the range of a DO, all statement numbers to which control may be transferred must lie in a single part of the DO nest which includes the range, or be completely outside the nest.

26) No constants may be given in a list in an input-output statement, only variables.

27) FORMAT statements for output must be so written that the first character of the first field provides for spacing of the typed lines. In this primer, single-spaced typing was provided by beginning the format specification with the characters 1H. See Section III for other spacing.

28) The physically last statement of every FORTRAN source program must be an END statement.

SUMMARY OF FORTRAN STATEMENTS

The complete FORTRAN language provides for six classes of statements, which may be grouped as follows:

1) Arithmetic and Logical statements

2) Control statements (17 types)

3) Input-output statements

4) Specification statements (7 types)

5) Subprogram facility statements (4 types)

6) In-line machine language statements

This primer has covered the following statements:

1) Arithmetic statements

2) The following 9 types of control statements:

   a) Unconditional GO TO

   b) Assigned GO TO

   c) Computed GO TO

   d) ASSIGN

   e) IF

f) DO

g) CONTINUE

h) STOP

i) END

3) The following 3 types of input-output statements:

a) FORMAT

b) READ PAPER TAPE

c) TYPE

4) The following type of specification statement:

a) DIMENSION

5) One subprogram facility statement:

a) CALL EXIT

The types of FORTRAN statements which have not been covered in this section are:

1) Logical statements

2) The following 7 types of control statements:

a) SENSE LIGHT

b) IF (SENSE LIGHT)

c) IF ACCUMULATOR OVERFLOW

d) IF QUOTIENT OVERFLOW

e) IF DIVIDE CHECK

f) PAUSE

g) STOP (except as associated with CALL EXIT)

3) The following 14 types of input-output statements:

a) ACCEPT

b) PUNCH TAPE

c) READ INPUT TAPE

d) WRITE OUTPUT TAPE

e) READ TAPE

f) WRITE TAPE

g) REWIND

h) BACKSPACE

i) END FILE

j) READ

k) PUNCH

l) PRINT

m) READ DRUM

n) WRITE DRUM

4) The following 6 types of specification statements:

a) EQUIVALENCE

b) FREQUENCY

c) COMMON

d) REAL

e) INTEGER

f) LOGICAL

5) Subprogram facility statements:

a) FUNCTION

b) SUBROUTINE

c) RETURN

d) CALL (other than CALL EXIT)

6) In-line machine language statements

Having approached the FORTRAN language cumulatively through the three stages presented in the sections of this primer, the reader should have little difficulty in extending his knowledge of FORTRAN to include the entire FORTRAN language as presented in FORTRAN Fundamentals.

SECTION III

FUNDAMENTALS

GENERAL PROPERTIES OF A FORTRAN SOURCE PROGRAM

A FORTRAN source program consists of a sequence of source statements. There are six different classes of statements, which are described in detail below. The FORTRAN source program is translated by the FORTRAN compiler into machine language for the computer to be used. The program obtained is called the object program.

In the FORTRAN primer, all of the examples were coded and described as though only the standard I/O devices were available on the DDP-24 (paper tape reader, paper tape punch and typewriter). In this section, a larger system will be assumed. The assumed I/O configuration will be:

1) Magnetic tape for input

2) Magnetic tape for output

3) Cards as the initial source input media

4) Cards as the final object output media

5) A line printer

EXAMPLE OF A FORTRAN PROGRAM

The following brief program will serve to illustrate the general appearance and some of the properties of a FORTRAN program.

Example 18:

```
C       PROGRAM FOR FINDING THE LARGEST          V
        ALUE ATTAINED BY A SET OF NUMBERS
        DIMENSION A(999)
        FREQUENCY 30 (2, 1, 10), 5 (100)
        READ INPUT TAPE 5, 1, N, (A(I), I = 1, N)
    1   FORMAT (I3/12F6 · 2))
```

```
      BIGA = A (1)
    5 DO 20 I = 2, N
   10 BIGA = A(I)
   20 CONTINUE
      WRITE OUTPUT TAPE 6, 2, N, BIGA
    2 FORMAT (22H1 THE LARGEST OF THESE I3, 12H NUMBERS
      IS F7 · 2)
      CALL EXIT
      STOP
      END
```

The purpose of the program is to determine the largest value attained by a set of n numbers, a_i (i = 1, . . . , n), and to write the number on magnetic tape for off-line listing. The numbers exist on punched cards, 12 to a card, each number occupying a field of six columns. There are no more than 999 numbers; the actual number is punched on the leading card and it is the only number on that card. These numbers, or data, are transferred to the input tape by means of an off line process, and read into the computer under control of the program.

The program sets BIGA equal to a_1. Next, the DO statement causes the succeeding statements to and including statement 20 to be carried out repeatedly, first with i = 2, then with i = 3, etc., and finally with i = n. During each repetition of this loop, the IF statement compares BIGA with a_i; if BIGA is less than a_i, statement 10, which replaces BIGA by a_i, is executed before continuing. An appropriate sentence indicating the largest value of the set of numbers is then written on the output tape, which is then transferred to peripheral equipment for printing.

KEY-PUNCHING THE SOURCE PROGRAM

Key-punching the FORTRAN source program is only slightly different for cards or paper tape. In the description that follows, a line is used as the unit of information that is processed by the compiler. When working with cards, a line is defined to be the information contained in the first 72 columns (columns 1-72) of the card. When working with paper tape, a line is defined to be the information contained in all of the frames up to a carriage return; under no circumstance, however, may a line exceed 72 characters (not including the carriage return). The first five positions of a line can be ignored if the tab is used. A tab code causes the compiler to start processing the line with position 6.

Each statement of a FORTRAN source program is punched as a separate line. The order of the source statements is governed solely by the order of the statement lines.

However, if a statement is too long to fit on a single line, it may occupy up to a total of ten lines, the initial line and up to nine continuation lines. Position 6 is reserved for this purpose. For each such statement, the initial line must contain a blank in position 6; on the continuation lines, position 6 should be used to number the lines consecutively from 1 to 9. (Actually, the compiler recognizes any character other than blank in position 6 as an indication of a continuation line; it does not check numerical order.)

Lines which contain a "C" in position 1 are not processed by FORTRAN. Such lines may, therefore, be used to carry comments which will appear when the source program is listed. Continuation lines for comments need not be punched in position 6; only the "C" in position 1 is necessary.

Numbers less than 32,768 may be punched in positions 1-5 of the initial line of a statement. When such a number appears in these positions, it becomes the statement number of the statement. These statement numbers permit cross references within the source program, and also help the programmer correlate the object program with his source program.

The statements themselves are punched in positions 7-72, both on initial and continuation lines. Thus a statement may consist of not more than 10 x 66 = 660 characters (i.e., 10 lines).

Blank characters may be used freely to improve the readability of the source program listing. FORTRAN will ignore all blanks except in certain FORMAT statements.

Position 73-80 are not processed by FORTRAN. They should be used to number cards consecutively so that the deck can be sorted in proper order when necessary. If the last digit is initially always a zero (i.e., the cards are numbered by 10's), insertions can be made in the deck without disturbing the sequence of the original cards.

PREVIEW OF THE FORTRAN STATEMENTS

The six types of statements which can be used in a FORTRAN program are as follows:

1) The Arithmetic or Logical Formula, which specifies a numerical or logical computation.

2) The Control Statements, which govern the flow of control in the object program.

3) The Input-Output Statements, which provide the object program with its input and output routines.

4) The Specification Statements, which provide information required, or desirable, to make the object program efficient.

5) The Subprogram Statements, which enable the programmer to define and use subprograms.

6) The in-line machine language statements.

CONSTANTS, VARIABLES, AND SUBSCRIPTS

Any programming language must provide for expressing numerical constants and variable quantities. FORTRAN also provides a subscript notation for expressing 1, 2, or 3-dimensional arrays of variables.

CONSTANTS

Three types of constants are permissible: INTEGER or fixed-point (restricted to integers), REAL or floating-point (characterized by being written with a decimal point) and LOGICAL (preceded by an apostrophe signifying octal numbers).

INTEGER Constants

GENERAL FORM	EXAMPLES
1 to 7 decimal digits. A preceding + or - sign is optional. The magnitude of the constant must be less than 8, 388, 608.	3 +1 -28987

Any unsigned INTEGER constant less than 32768 may be used as a statement number.

REAL Constants

GENERAL FORM	EXAMPLES
Any number of decimal digits, with a decimal point at the beginning, at the end, or between two digits. A preceding + or - sign is optional. A decimal exponent preceded by an E may follow.	17. 5.0 -.0003 5.0E3 ($=5.0 \times 10^3$) 5.0E+3 ($=5.0 \times 10^3$) 5.0E-7 ($=5.0 \times 10^{-7}$)

The magnitude of the number thus expressed lies between the approximate limits of 10^{-75} to 10^{75}, or can be zero.

LOGICAL Constants

GENERAL FORM	EXAMPLES
1 to 8 octal digits. A preceding + or - sign is optional. Logical constants must always be preceded by an apostrophe (')	'77456071 '-0 (=40000000) '+0 (=00000001) '-1 (=40000001) '14 (=00000014)

VARIABLES

Four types of variables are also permissible: INTEGER, REAL, LOGICAL and Boolean. Variables are referenced in the FORTRAN source language by symbolic names consisting of alphabetic characters and, if desired, numerical digits. However, INTEGER variables are distinguished by the fact that the first character of their symbolic name is I, J, K, L, M, or N unless the INTEGER specification statement is used (see Specification Statements).

INTEGER Variables.

GENERAL FORM	EXAMPLES
1 to 8 alphabetic or numeric characters (not special characters) of which the first is I, J, K, L, M, or N	1 M2 JOBNO

An INTEGER variable can assume any integral value whose magnitude is less than 8, 388, 608. Further information on INTEGER arithmetic is contained below.

REAL Variables

GENERAL FORM	EXAMPLES
1 to 8 alphabetic or numeric characters (not special characters) of which the first is alphabetic but not I, J, K, L, M, or N.	A B7 DELTA

A REAL variable can assume any value expressible as a normalized floating-point number; i.e. zero, or with magnitude between approximately 10^{-75} and 10^{75}.

LOGICAL Variables

GENERAL FORM	EXAMPLES
1 to 8 alphabetic or numeric characters.	LOGICALV
	VARIABLE
	INTEGER
	BETA

Since LOGICAL variables must be defined in a LOGICAL specification statement, there is no reason to be concerned with the beginning character of the LOGICAL variable name.

BOOLEAN Variables

A Boolean variable is defined to be any variable that has been defined in a LOGICAL specification statement, but used in an arithmetic expression. Thus, if A has been defined as a LOGICAL variable, then

```
B = A*5
```

would store five (in floating-point) or zero in B dependent on A being a value (true) or being zero (false), respectively.

The fact that A has been defined as LOGICAL instructs the compiler to convert A to one or zero, if it appears in an arithmetic operation. The actual value assigned to A in storage is not modified; the conversion of A for Boolean significance is part of the arithmetic operation only. This conversion holds true for REAL or INTEGER expressions, but not for LOGICAL expressions. If IBOOL has been defined as LOGICAL and IJOB is INTEGER,

```
IJOB = IJOB*IBOOL
```

would be interpreted as: if IBOOL ≠ 0, set IJOB to IJOB (unchanged)
if IBOOL = 0, set IJOB to zero

however

```
IJOB = IJOB.AND.IBOOL
```

would be interpreted as: form the logical 24-bit product of IJOB and IBOOL
and store the result back in IJOB.

A logical variable is treated as one or zero (if true or false, respectively) in any arithmetic operation. Thus, if IBOOL is LOGICAL the value of the expression

5*IBOOL

is five, if IBOOL is true and zero, if IBOOL is false.

The value of the expression

5+IBOOL

is six, if IBOOL is true and five, if IBOOL is false.

The value of the expression

5-IBOOL

is four, if IBOOL is true and five, if IBOOL is false.

To avoid the possibility that a variable may be considered by FORTRAN to be a function (fully discussed later), the following warnings should be observed with respect to the naming of variables:

1) A variable cannot be given a name which coincides with a name of a function minus its terminal F. Thus, if a function is named TIMEF, no variable should be TIME.

2) Unless their names are three characters or less in length, subscripted variables must not be given names ending with F, as FORTRAN will consider variables so named to be functions.

SUBSCRIPTS AND SUBSCRIPTED VARIABLES

A variable can be made to represent any element of a 1, 2, or 3-dimensional array of quantities by appending to it 1, 2, or 3 subscripts; the variable is then a subscripted variable. The subscripts are quantities whose values determine the member of the array to which reference is made. A subscript may be any legal arithmetic expression; however, the resulting value will be a truncated integer.

Subscripted Variables

GENERAL FORM	EXAMPLES
A REAL, INTEGER or LOGICAL	A(I)
variable followed by parentheses enclosing	K(3)
1, 2, or 3 subscript expressions separated	BETA(5*J-2, K+2, L)
by commas.	GAMMA (SQRTF(Y), 5/2, 3*A)

For each variable that appears in subscripted form, the size of the array (i.e. the maximum values which its subscripts can attain) must be stated in a DIMENSION statement preceding the first executable statement in the source program.

The value of a subscript, exclusive of its addend, if any, must be greater than zero and not greater than the corresponding array dimension.

ARRANGEMENT OF ARRAYS IN STORAGE. A two-dimensional array A will, in the object program, be stored sequentially in the order $A_{1,1}$, $A_{1,2}$, . . ., $A_{1,m}$, $A_{2,1}$, $A_{2,2}$, . . ., $A_{2,m}$, . . ., $A_{n,m}$. Thus it is stored with the first of its subscripts varying least rapidly, and the last varying most rapidly. The same is true of three-dimensional arrays.

ARRAY	ARRANGEMENT IN STORAGE	
$A_{1,1}$ $A_{1,2}$ $A_{1,3}$	$A_{1,1}$	(1)
$A_{2,1}$ $A_{2,2}$ $A_{2,3}$	$A_{1,2}$	(2)
$A_{3,1}$ $A_{3,2}$ $A_{3,3}$	$A_{1,3}$	(3)
	$A_{2,1}$	(4)
	$A_{2,2}$	(5)
	$A_{2,3}$	(6)
	$A_{3,1}$	(7)
	$A_{3,2}$	(8)
	$A_{3,3}$	(9)

All arrays are stored forwards in storage; i.e., the above sequence is in the order of increasing absolute locations.

ARITHMETIC STATEMENTS, EXPRESSIONS, AND FUNCTION DEFINITIONS

ARITHMETIC STATEMENTS

GENERAL FORM	EXAMPLES
"a = b" where a is a variable	Q1 = K
(subscripted or not subscripted)	A(I) = B(I) +
and b is an expression.	SINF(C(I))

An arithmetic statement (or arithmetic formula) defines a numerical calculation. A FORTRAN arithmetic formula resembles very closely a conventional arithmetic formula; it consists of a variable to be computed, followed by an = sign, followed by an arithmetic expression.

However, in a FORTRAN arithmetic formula, the = sign means "is to be replaced by," not "is equivalent to." Thus, the arithmetic formula

```
Y = A - B*C
```

means "replace the value of Y by the value of A - B*C". An arithmetic formula therefore instructs the computer to compute the value of the right-hand side and to store that value in the memory location specified by the left-hand side.

The result will be stored in INTEGER, REAL or LOGICAL form if the variable to the left of the = sign is an INTEGER, REAL or LOGICAL variable, respectively.

If the variable on the left is INTEGER and the expression on the right is REAL, the result will first be computed in floating-point and then truncated and converted to an INTEGER. Thus, if the result is +3.872, the INTEGER number stored will be +3, not +4. If the variable on the left is REAL, and the expression on the right INTEGER, the latter will be computed in INTEGER, and then converted to REAL.

Some examples of arithmetic formulas are given below.

FORMULA	
A = B	Store the value of B in A.
I = B	Truncate B to an INTEGER and store in I.
A = I	Convert I to floating point, and store in A.
I = I + 1	Add 1 to I and store in I. This example illustrates the point that an arithmetic formula is not an equation but a command to replace a value.

FORMULA	
A = 3.0*B	Replace A by 3B.
A = 3*B	Convert 3 to floating-point, multiply by B and store the floating-point result in A.
A = I*B	Convert I to floating-point, multiply by B and store the floating-point result in A.

EXPRESSIONS

As noted at the beginning of this subsection, the right-hand side of a formula consists of an expression. An expression is any sequence of constants, variables (subscripted or not subscripted), and functions, separated by operation symbols, commas, and parentheses so as to form a meaningful expression.

SEQUENCE OF OPERATIONS. When the order of operations in an expression is not explicitly specified by the use of parentheses, it is understood to be in the following order (from innermost operations to outermost):

Exponentiation

Multiplication and Division

Addition and Subtraction

Logical

For example, the expression

A + B/C + D**E*F - G.AND.H

will be taken to mean

(A + (B/C) + (D^{E}*F) -G).AND.H

ORDERING WITHIN A SEQUENCE. Parentheses which have been omitted from a sequence of consecutive multiplications and division, consecutive additions and subtractions, or consecutive logical operations will be understood to be grouped from the left. Thus, if . represents * or /, + or -, or .AND. or .OR. or .NOT. or .SHFT., then

A · B · C · D · E

will be taken to mean

((((A · B) · C) · D) · E)

ARITHMETIC EXPRESSIONS

Considerable attention is given by FORTRAN to the matter of efficiency of the object program arising from an arithmetic expression, regardless of how the expression has been written. FORTRAN assumes that mathematically equivalent expressions are computationally equivalent. Hence, a sequence of consecutive multiplications (and/or subtractions) not grouped by parentheses may give rise in the object program to a series of computations differing in sequence, although not in result, from conventional methods of considering unparenthesized expressions. This change in computation sequence makes the object program as efficient as possible in terms of storage locations used and execution time required.

WARNING

Although the above stated assumption concerning mathematical and computational equivalence holds true for REAL expressions, special care must be taken to indicate the order of INTEGER multiplication and division. FORTRAN INTEGER arithmetic is "greatest integer" arithmetic (i.e., truncated or remainderless). Therefore, the expression

5 x 4/2

which is conventionally taken to mean ((5 x 4)/2, is computed in a FORTRAN object program as

((5/2) x 4)

i.e., from left to right after permutation of the operands, to minimize storage accesses. The result of a FORTRAN computation in this case would be 8, instead of the 10 that would normally be expected. Therefore, to insure accuracy of INTEGER multiplication and division, it is suggested that parentheses be inserted into the expression involved.

VERIFICATION OF THE CORRECT USE OF PARENTHESES. The following procedure can be used for checking that the parentheses in a complicated expression correctly express the desired operations:

Label the first open parenthesis "1"; thereafter, working from left to right, increase the label by 1 for each open parenthesis and decrease it by 1 for each closed parenthesis. Then the label of the last parenthesis should be 0; the mate of an open parenthesis labeled n will be the next parenthesis labeled n - 1.

LOGICAL Expressions

A logical expression is defined to be any expression in which one or more of the following operators are used:

.AND.	(logical product)
.OR.	(logical sum)
.NOT.	(logical difference)
.SHFT.	(logical shift)

If A and B are described as logical variables, the expression

A.AND.B

results in the logical product of corresponding bits in the two variables. Logical products are defined as:

1.AND.1=1, 1.AND.0=0, 0.AND.1=0, 0.AND.0=0.

The expression

A.OR.B

results in the logical sum of the corresponding bits in the two variables. Logical sums are defined as:

1.OR.1=1, 1.OR.0=1, 0.OR.1=1, 0.OR.0=0.

The expression

A.NOT.B

results in the logical difference of the corresponding bits in the two variables. Logical differences are defined as

1.NOT.1=0, 1.NOT.0=1, 0.NOT.1=1, 0.NOT.0=0.

The logical operator .SHFT. is used for shifting the bit pattern of any variable right or left. The direction of shift is specified by the sign of the expression of the second operand, a positive value shifts right, a negative value shifts left.

The operands for logical operators may be any expression composed of variables or constants in any mode (REAL, INTEGER or LOGICAL). However, the operation will be performed on only single word, 24-bit quantities; therefore, if A and B are REAL numbers, the expression

A.AND.B

would form the logical product of the most significant words of the double length (floating-point) operands.

The variable on the left side of a logical expression determines the mode of the overall expression on the right side of the expression. Thus, if A is a REAL variable and B and C are INTEGER variables, the expression

```
A = B.OR.C
```

would form the logical sum of B and C, convert the result to floating-point and store the answer in A.

Hierarchy

The four logical operators are equivalent in hierarchy to each other, but lower than all other operators. Thus, the order of hierarchy is

**	exponentiation
*, /	multiplication, division
+, -	addition, subtraction
.AND.,.OR.,.NOT.,.SHFT.	logical operators

It is clear then that parentheses are used in logical expressions in much the same way they are used in normal arithmetic expressions: to denote the sequence of operations to be performed within an expression.

The expression

```
A.OR.B.AND.C
```

is equivalent to

```
(A.OR.B).AND.C
```

but not equivalent to

```
A.OR.(B.AND.C)
```

It should be emphasized that the operands for logical operators are full word (24-bit) quantities.

Example 1

Consider the following simple problem:

It is necessary to take the right half of ALPHA (12 bits) and insert this information into the right half of BETA. The program to accomplish this could be:

```
St. No.     FORTRAN Statement
  10        LOGICAL ALPHA, BETA
  20        BETA = (ALPHA.AND.'7777).OR.(BETA.AND.'77770000)
  30        CALL EXIT
  40        STOP
            END
```

The first statement (10) declares that ALPHA and BETA are logical variables. The second statement (20) masks the low order 12 bits of ALPHA and clears the high order 12 bits; masks the high order 12 bits of BETA and clears the low order 12 bits; merges these two words into one containing the high order 12 bits of BETA and the low order 12 bits of ALPHA and then stores the result in BETA. The remainder of the program is the normal exit procedure.

Example 2

It is necessary to take the low order 12 bits of ALPHA, place this information in the high order 12 bits of GAMMA and store the results of this operation in the third element of the ten element BETA array. The program to accomplish this could be:

```
St. No.     FORTRAN Statement
  10        DIMENSION BETA(10)
  20        LOGICAL ALPHA, BETA, GAMMA
  30        BETA(3) - (ALPHA.SHFT.-12).OR.(GAMMA
          X           .AND.'00007777)
  40        CALL EXIT
  50        STOP
            END
```

FUNCTIONS

In order to better clarify the meaning and uses of functions, they will first be discussed in their relation to subprograms as a whole. A subprogram is any sequence of instructions which performs some desired operation. It is frequently necessary in the writing of a large program to use a basic group of instructions on several different occasions to perform a specific job. It is obviously wasteful to write the same group of statements at many different places in the same program. What is needed is some way to write the steps only once, then arrange to refer to that series of statements each time the operation is required.

One way to do this is to prepare the subprograms which perform commonly recurring operations once for all; they can then be kept in a library tape and used by the program at execution time. Another way is to include in the FORTRAN compiler itself certain basic subprograms which will be compiled as part of the object program (whenever this is indicated in the source program). In addition, the programmer may find that certain sub-units of his program are frequently repeated, and he may wish to write each of these only once and call for them many times. In this case, he has a choice of several types of subprograms. (Another advantage in constructing programs from subprograms is that it may be possible to use the same "building blocks" in other problems or in modifications to the original program.)

In FORTRAN, there are five types of subprograms. Of these, four result in a single value and are called "functions"; the fifth (which may result in more than one value) is called a "subroutine" These types are as follows:

Library Functions

Built-In Functions

Arithmetic Statement Functions

FORTRAN Functions (FUNCTION-Type Subprograms)

Subroutines (SUBROUTINE-TYPE Subprograms)

For each type of subprogram, there are standard practices which must be followed in referring to (or calling) the subprogram, in naming it, and in defining (or generating) it.

INCORPORATING FUNCTIONS INTO THE PROGRAM. All functions are incorporated into the object program through making a source program reference of the name of the function in the expression part (right-hand side) of an arithmetic formula. Following are examples of arithmetic expressions including function names.

```
Y = A - SINF (B - C)

C = MINOF(M, L) + ABC(B*FORTF(Z), E)
```

The names of Library, Built-In, Arithmetic Statement, and FORTRAN functions are all used in this way. The appearance in the arithmetic expression serves to "call" the function; the value of the function is then computed, using the arguments which are supplied in the parentheses following the function name. Only one value, or single numerical quantity, is produced by these four functions.

NAMING OF FUNCTIONS. The following paragraphs describe the rules for naming Library, Built-In, and Arithmetic Statement functions.

GENERAL FORM	EXAMPLES
The name of the function consists of 4 to 9 alphabetic or numeric characters (not special characters), of which the last must be F and the first must be alphabetic. Also, the first must be X if and only if the value of the function is to be fixed point. The name of the function is followed by parentheses enclosing the arguments (which may be expressions), separated by commas.	SINF (A + B) SOMEF (X, Y) SQRTF (SINF(A)) XTANF(3.*X)

Mode of a Function and its Arguments. Consider a function of a single argument. It may be desired to state the argument either in fixed or floating-point; similarly the function itself may be in either of these modes. Thus a function of a single argument has 4 possible mode configurations; in general a function of n arguments will have 2^{n+1} mode configurations.

A separate name must be given, and a separate routine must be available, for each of the mode configurations which is used. Thus a complete set of names for a function might be

SOMEF	INTEGER argument, REAL function
SOMEQF	REAL argument, REAL function
XSOMEF	INTEGER argument, INTEGER function
XSOMEQF	REAL argument, INTEGER function

The X's and F's are compulsory, but the rest of the naming is arbitrary.

FORTRAN functions are named in a manner different from the previous three types. These functions are named in exactly the same way as ordinary variables of the program, except that no name of a FORTRAN function which is 4 to 8 characters long may end in F. This means that the name of an INTEGER FORTRAN function must have I, J, K, L, M, or N for its first character or have been defined as an INTEGER.

DEFINITION OF FUNCTIONS. Each of the four types of functions is defined (or generated) in a different way.

Built-In Functions are contained in the FORTRAN system; they are listed in the Appendix. Each of these subroutines will be compiled into the object program whenever an arithmetic statement is encountered in the source program which calls for one of them.

Library Functions are prewritten functions of a special type. Historically, they were designed for use on the library tape - hence the name. These functions are "closed" subroutines; i.e., instead of appearing in the object program each time that a reference is made to them in the source program, they appear only once regardless of the number of references.

Arithmetic Statement Functions are defined by a single FORTRAN arithmetic statement(1), and apply only to the particular program or subprogram in which their definition appears.

GENERAL FORM	EXAMPLES
"af = b" where a is a function name ending in F and followed by parentheses enclosing its arguments (which must be distinct non-subscripted variables) separated by commas, and b is an expression which does not involve subscripted variables. Any functions appearing in b must be built-in, or available on the library tape, or already defined by preceding function statements.	FIRSTF(X) = A*B + B SECONDF(X, B) = A*X + B THIRD(D) = FIRSTF(E)/D FOURTHF(F, G) = SECONDF(F, THIRDF(G)) FIFTH(I, Z) = 3.0*A**I SIXTHF(J) = J + K XSIXTHF(J) = J + K

If the name begins with an X, the answer will be expressed in INTEGER; if it begins with any other letter, a REAL answer will be given.

The right-hand side of a function statement may be any expression not involving subscripted variables, that meets the requirements specified for expressions. It may involve functions freely, including Built-In Functions, Library Functions, previously defined Arithmetic Statement Functions, and FORTRAN Function Subprograms.

Of course, no function can be used as an argument of itself.

As many as desired of the variables appearing in the expression on the right-hand side may be stated on the left-hand side to be the arguments of the function. Since the arguments are really only dummy variables, their names are unimportant (except as indicating INTEGER or REAL mode) and may even be the same as names appearing elsewhere in the program.

Those variables on the right-hand side which are not stated as arguments are treated as parameters. The naming of parameters, therefore, must follow the normal rules of uniqueness.

A function defined by a function statement may be used just as any other function. In particular, its arguments may be expressions and may involve subscripted variables; thus a reference to FIRSTF(Z + Y(1)), with the same definition of FIRSTF as in the preceding paragraph, will yield $a(z + y_i) + b$ on the basis of a, b, y_i, and z.

(1) An arithmetic statement which defines a function may also be called a "function statement" or a "function definition formula".

Functions defined by function statements are always compiled as closed subroutines.

NOTE

> All the arithmetic statements defining functions to be used in a program must precede the first executable statement of the program. (An executable statement is, in general, one which processes or moves data, or which alters the flow of control in the program.)

FORTRAN Functions are those subprograms which, on the one hand, cannot be defined by only one arithmetic statement, and on the other are not utilized frequently enough to warrant a place on the library tape.

They are called FORTRAN Functions because they may conveniently be defined by a conventional FORTRAN program. In this instance, compiling a FORTRAN program produces a Function subprogram in exactly the form required for object program execution.

In an earlier discussion, the concept of employing building blocks as standard parts for a complete structure was presented. In computing terminology, the complete structure is the whole program, the building blocks are called subprograms. A subprogram normally carries out a well defined mathematical or logical operation. Since each engineering group has certain calculations that are used frequently in many programs, it is convenient to have each of these computations written as a subprogram, which can be written and checked out (separately in advance) so that it is ready for use at any time. Each group may set up a "library" of such subprograms for its own use. In addition, general subprograms, such as the calculation of a trigonometric function, are available for all programmers on the library tape.

FORTRAN SUBPROGRAMS

It is possible to program, in the FORTRAN language, subroutines which are referred to by other programs. These subroutines may, in turn, refer to still other lower level subroutines which may also be coded in FORTRAN language. It is therefore possible, by means of FORTRAN, to code problems using several levels of subroutines. This configuration may be thought of as a total problem consisting of one main program and any number of subprograms.

Because of the interrelationship among several different programs, it is possible to include a block of hand-coded instructions in a sequence including instructions compiled from FORTRAN source programs. It is only necessary that hand-coded instructions conform to rules for subprogram formation, since they will comprise a distinct subprogram.

We will now discuss two types of FORTRAN coded subprograms: the FUNCTION subprogram and the SUBROUTINE subprogram. Four statements, described subsequently, are necessary for their definition and use: SUBROUTINE, FUNCTION, CALL, and RETURN.

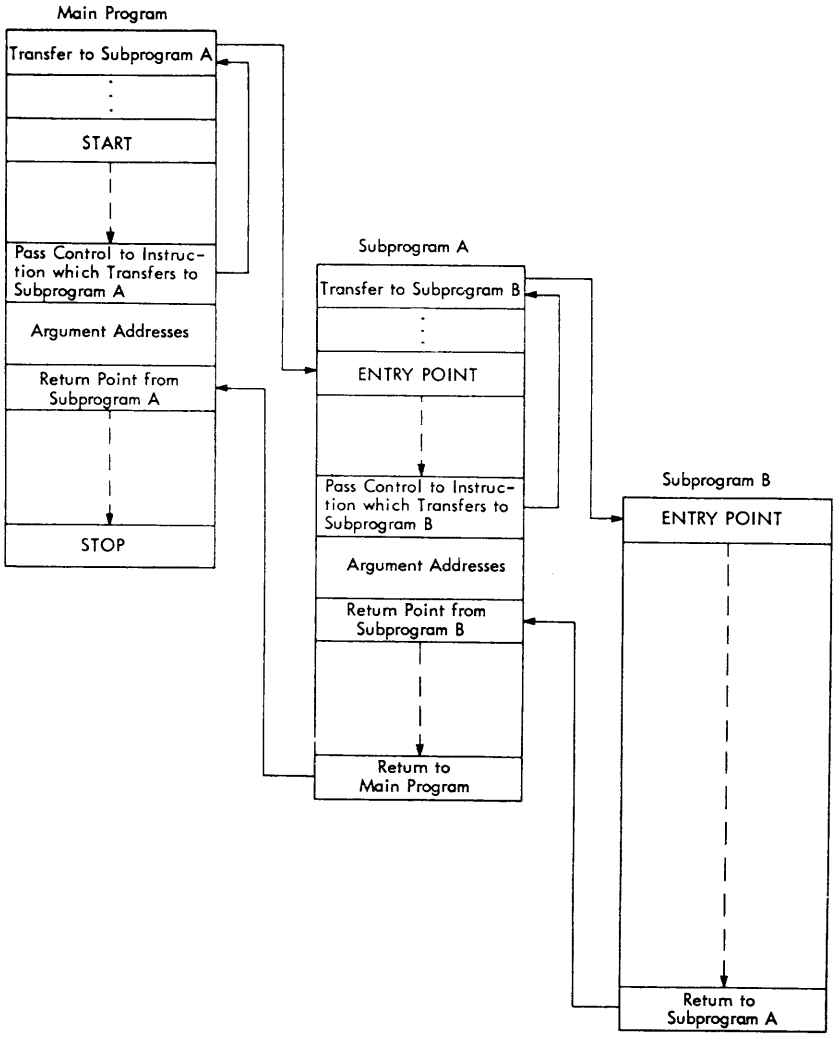
Although FUNCTION subprograms and SUBROUTINE subprograms are treated together and may be viewed as similar, it must be remembered that they differ in two fundamental respects.

1) The FUNCTION subprogram, which results in a FORTRAN function is always single-valued, whereas the SUBROUTINE subprogram may be multi-valued.

2) The FUNCTION subprogram is called or referred to by the arithmetic expression containing its name; the SUBROUTINE subprogram can only be referred to by a CALL statement.

Each of these two types of subprogram, when coded in FORTRAN language must be regarded as independent FORTRAN programs. In all respects, they conform to rules for FORTRAN programming.

Schematically, the relationship among nested main and subprograms can be shown as follows. This diagram, further, indicates the main division of the internal structure of each program.



CLASSIFICATION OF SUBPROGRAMS

Subprograms are either "closed" or "open". An open subprogram is inserted directly in the sequence of machine operations in the object deck; it is inserted each time it is used in the program. A closed subprogram is normally not stored within the range of the main program; control is transferred from the main program to the subprogram when it is required. After the calculation in the subprogram has been performed, control reverts to the main program. Although a subprogram may be used several times in the total structure, a closed subprogram appears in storage only once.

Subprograms may also be classified in another way: those that must be compiled independently, or those that are not compiled independently. A discussion has already been given for functions on the library tape, built-in functions and function definitions. None of these are compiled independently.

For our purposes we may also distinguish subprograms in still another way - according to the output of the subprogram. When the subprogram has only a single result, it will be called a function. A subprogram capable of having one or more results will be called a subroutine.

A summary of the characteristics of the different kinds of subprograms appears on the following page.

FUNCTION SUBPROGRAMS

There are situations in which it is desired to use a particular function in an arithmetic statement, but this function cannot be defined by a single arithmetic statement. However, if this mathematical relationship has a single result, the function subprogram may be used.

This function subprogram must be compiled independently; it may have several pieces of input, but a single output. The name of the function can be used in any arithmetic statement in the program.

WRITING A FUNCTION SUBPROGRAM

The general form in which a function subprogram is written is:

FUNCTION NAME (Argument$_1$, Argument$_2$, ...)
. . . . .) Arithmetic statements to evaluate
. . . . .) the function
NAME = Final Calculation
RETURN

SUBPROGRAM SUMMARY							
TYPE	NUMBER OF ARGUMENTS	NUMBER OF RESULTS	HOW COMPILED	KIND OF SUBPROGRAM	NAME AND MODE (3)	DIMENSION STATEMENTS	METHOD OF USING
Built-in Function	N (1)	1	Part of FORTRAN	Open	Reserved in FORTRAN; first character X for INTEGER.	Main program only.	By name in statement in program.
Library Function (4)	N (2)	1	On FORTRAN Library Tape	Closed	4 to 9 alphanumeric characters, first is alphabetic, last is F. First is X for INTEGER.	Main program only.	By name in statement in program.
Arithmetic Statement Functions	N (1)	1	Compiled with program	Closed	Same as Library function.	Main program only.	By name in statement in program.
FORTRAN Function Subprograms	N (1)	1	Compiled independently	Closed	1 to 8 alphanumeric characters. First is alphabetic. Last is not F if there are 4, 5, or 6 characters. For INTEGER first is I, J, K, L, M, or N (or defined as INTEGER)	Same in main program and Subprogram.	By name in statement in main program.
Subroutine Subprogram	N	M	Compiled independently	Closed	1 to 8 alphanumeric characters. First is alphabetic. Last is not F if there are 4, or more characters. Name must not appear in DIMENSION statement of any program having a CALL for the subroutine.	Same in main program and subprogram.	By CALL statement in main program.

(1) Depends upon relationship, but there must be at least 1.
(2) May be 1 or more.
(3) Mode is floating point unless specified.
(4) Subroutine subprograms can be put on the FORTRAN library tape; then the information concerning its name and use follows the description given for subroutine subprograms, not Library Functions.

The FUNCTION statement must be the first statement of the subprogram and defines it as such. The function subprogram may consist of many statements of any type except the statements FUNCTION or SUBROUTINE.

The name of the function subprogram consists of 1 to 8 alphabetic or numeric characters, the first of which must be alphabetic; the first character must be I, J, K, L, M or N if and only if the value of the function is to be INTEGER (for exceptions, see specification statements). The last character must not be F if the total number of characters is 4 or more. The function name must not occur in a DIMENSION statement in the function subprogram, nor in a DIMENSION statement in any program which uses the function. The function name must not be the same as that of any variable appearing elsewhere in the program.

There must be at least one argument, although there may be as many as required in the subprogram. The arguments must be non-subscripted variable names. If any of the arguments are arrays, a DIMENSION statement involving these arguments is necessary. The arguments may be any variable names occurring in executable statements in the subprogram. Actually, these are dummy variables and the calculation is set up in terms of these dummy variables. A dummy variable in a function subprogram should not normally appear on the left side of an arithmetic statement except as a subscript. The reason is that it is generally undesirable to change the value of the arguments supplied to the subprogram by the main program. Similarly, a dummy variable should not appear in an ASSIGN statement unless it is re-established as an ordinary variable subsequently in the subprogram. In the example below, the dummy variables are A and B; both are arrays, so that a DIMENSION statement is required.

The arithmetic statements to evaluate the function are written in the normal fashion in terms of arguments and constants. The subprograms must evaluate a single-valued function (one which has one and only one value for a given set of arguments). The name of the function must be used as a variable and evaluated by an arithmetic statement; or stated another way, the name of the function must appear on the left-hand side of an arithmetic statement. It is the value of the function name, used as a variable, that is returned as the function value.

A RETURN statement indicates the conclusion of the subprogram. The form of this statement is simply

```
RETURN
```

This statement terminates a subprogram and returns control to the main program. A RETURN statement must be the last statement to which control passes in a function subprogram; that is, it must be the last statement logically, but not necessarily physically.

Example:

```
      FUNCTION SUM (A, B)
      DIMENSION A(500), B(500)
      SUM = A(1) + B(1)
      DO 5 J = 2, 500
5     SUM = SUM + A(J) + B(J)
      RETURN
      END
```

USE OF FUNCTION SUBPROGRAMS IN MAIN PROGRAM

Statement. A subprogram introduced by a FUNCTION statement is called for in the main program by an arithmetic formula involving the function name. For example, the subprogram introduced by FUNCTION ARCSIN (RADIAN) could be called for in the main program by the arithmetic formula:

```
A = B - ARCSIN (X)
```

Arguments. The list of arguments in the main program may contain any legitimate FORTRAN constant, variable (subscripted or non-subscripted), function expression, or name of any array, provided the corresponding dummy variable in the subprogram has the same mode. A Hollerith argument may not be used. There must be agreement in number, order and mode between the argument list following the function name in the main program and the argument list (dummy variables) in the FUNCTION statement. Identical DIMENSION statements are necessary in this subprogram and main program.

Example:

(Subprogram)

```
    FUNCTION AVRG (ALIST, N)
    DIMENSION ALIST (500)
    SUM = ALIST (1)
    DO 10 I = 2, N
10  SUM = SUM + ALIST (I)
    AVRG = SUM/FLOATF(N)
    RETURN
    END
```

(Main Program)

```
    DIMENSION SET (500)
    READ INPUT TAPE 5, 5,(SET(I), I = 1,200)
 5  FORMAT (6F12.8)
    TEXT = AVRG (SET, 200)
    WRITE OUTPUT TAPE 6, 10, TEXT
10  FORMAT (19H1 AVERAGE OF SET IS
   X E14.5)
    CALL EXIT
    STOP
    END
```

Note that the DIMENSION statement in the main program specifies the same length (500) for the array named SET as the DIMENSION statement in the subprogram specifies for the dummy variable ALIST. This is required even though the actual length of SET is only 200. The argument 200 is supplied to the subprogram from the main program and is used in the subprogram as an index maximum.

SUBROUTINE SUBPROGRAMS

Some desirable building blocks have multiple outputs; these can be compiled as subroutine subprograms. Each may also have multiple inputs and the calculation may require many statements.

The subroutine subprogram is compiled independently. In the main program it is called for by separate statement.

WRITING A SUBROUTINE SUBPROGRAM

When it is desired to use a subroutine subprogram the main program contains a statement of the form:

CALL NAME (Argument$_1$, Argument$_2$, ...).

Control is transferred at this point to the specified subroutine; when the calculations in the subroutine are finished, control is transferred to the statement following the CALL in the main program.

The general form in which a subroutine subprogram is written is:

```
SUBROUTINE NAME (Argument1, Argument2, . . .)
. . . . . ) Arithmetic statements to evaluate
. . . . . ) required results
. . . . . )
RETURN
```

The SUBROUTINE statement must be the first statement of the subprogram; it defines it as a subroutine.

The name of a subroutine consists of 1 to 8 alphanumeric characters, the first of which is alphabetic; the final character must not be F if the total number of characters is 4 or more. Also, the subroutine name must not occur in a DIMENSION statement in the subroutine, nor in a DIMENSION statement in any program having a CALL for this subroutine. In fact, the subroutine name must not be the same as any variable appearing elsewhere in the program, subscripted or not.

The arguments stated in the subroutine are dummy variables representing input and output variables; they are non-subscripted variables and they may be any variable names occurring in the executable statements in the subroutine. If an argument is the name of an array, it must appear in a DIMENSION statement following the SUBROUTINE statement. The arguments in the SUBROUTINE list will usually contain one or more dummy variables representing the result or results to be returned to the main program.

The intermediate part of the subroutine may contain any of the usual FORTRAN statements, arithmetic, control, input-output, or specification, except the two statements FUNCTION and SUBROUTINE. Usually it is undesirable for an input dummy variable to appear on the left side of an arithmetic statement except as a subscript. Also, a dummy variable should not appear in an ASSIGN statement unless it is re-established as an ordinary variable subsequently in the subroutine. The dummy variables representing the results of the subroutine may be used freely on the left side of arithmetic statements; each dummy variable standing for a result should appear at least once on the left side of a statement so that the value will be stored for future use.

A subroutine is terminated by a RETURN statement, which is the last statement to which control passes in a subroutine; that is, it must be the last statement logically but not necessarily physically. The last physical card in each subprogram must be an END card.

For an example it is desired to multiply matrix A, N rows and M columns, by matrix B, M rows and L columns; the product matrix C has N rows of L columns. The following subroutine accomplishes this operation.

(Subroutine)

```
1   SUBROUTINE MATMPY (A, N, M, B, L, C)
    DIMENSION A (10, 15), B(15, 12), C(10, 12)
    DO 5 I = 1, N
    DO 5 J = 1, L
3   C (I, J) = 0.0
    DO 5K = 1, M
5   C(I, J) = C(I, J) + A(I, K) * B(K, J)
    RETURN
    END
```

The DIMENSION statement following the SUBROUTINE statement specifies the maximum size of the matrices that may be used.

USING A SUBROUTINE SUBPROGRAM

When it is desired to use a subroutine subprogram in a program, a CALL statement is used to transfer control to the subroutine. The CALL statement is of the form

CALL NAME (Argument$_1$, Argument$_2$, . . .)

where NAME represents the symbolic name of a subroutine. This subroutine must be available to the main program at the time of execution of the program.

The arguments may have any of the eight forms herein described:

1) INTEGER constant.

2) REAL constant.

3) INTEGER variable, with or without subscripts.

4) REAL variable, with or without subscripts.

5) LOGICAL constant.

6) LOGICAL variable, with or without subscripts.

7) The name of an array, without subscripts.

8) A FORTRAN arithmetic expression.

The list of arguments in the CALL statement must agree in number, order, and mode with the list given in the SUBROUTINE statement. If any of the arguments are arrays, equivalent DIMENSION statements must appear in the subroutine and main program.

Suppose it is desired to find and print two product matrices. The previous subroutine is used in the following main program.

(Main Program)

```
    1   DIMENSION X(10,15), Y(15,12), Z(10,12),
      1 D(10,15), E(15,12), F(10,12)
        READ INPUT TAPE 5, 4, ((X(I,J), J = 1,10),
      1   I = 1,5),
      1 ((Y(I,J), J - 1,7), I = 1,10)
    4   FORMAT (6E12.8)
    5   CALL MATMPY (X, 5, 10, Y, 7, Z)
        READ INPUT TAPE 5,4, ((D(I,J), J = 1,8),
      1   I = 1,6),
      1 ((E(I,J), J = 1,5), I = 1,8)
   10   CALL MATMPY (D, 6, 8, E, 5, F)
        DO 13 J = 1, 7
   13   WRITE OUTPUT TAPE 6, 15, (Z(I,J), I = 1,5)
        DO 14 J = 1, 5
   14   WRITE OUTPUT TAPE 6, 15, (F(I,J), I = 1,6)
   15   FORMAT (1H06E17.6)
        CALL EXIT
        STOP
        END
```

There must be agreement in number, order, and mode between the argument list following the subroutine name in the CALL statement and the argument list in the SUBROUTINE statement.

CONTROL STATEMENTS

The second class of FORTRAN statements is the set of control statements, which enable the programmer to state the "flow" of his program.

Unconditional GO TO

GO TO n where n is a statement number.

As the name indicates, this statement will always cause a transfer in the operation flow. The next statement to be executed will be n. For example,

GO TO 10

will cause statement 10 to be the next statement executed.

Computed GO TO

GO TO (n_1, n_2, . . ., n_i), where n_1, n_2, . . ., n_i are statement numbers and i is an expression.

This form of control statement includes a list of statements, any one of which can receive control. The statement to be executed is determined by computing the value of the expression (i). The result will be an INTEGER, irrespective of the original mode of the expression.

For example, if the calculated value of the expression is 1.345, statement n_1 will be the next one executed (truncated). Similarly for integers up to i. If the truncated value is greater than the number of statements listed, that is, greater than i, unpredictable results may occur.

Assigned GO TO

GO TO m, (n_1, n_2, . . ., n_i), where n_1, n_2, . . ., n_i are statement numbers and m is a variable name.

This statement, like the computed GO TO is used to transfer control to one of a number of statements, However, the statement selected is not computed but assigned exactly by the value of the variable m. The value of m is set by the use of an ASSIGN statement defined below. It must be equal to a statement number rather than a position in the statement list.

ASSIGN

ASSIGN i TO m where m is a variable and i is a statement number.

This statement is used to set m with a statement number i for use in an assigned GO TO statement. The following examples show how the same transfer in control is accomplished but by different methods.

Example 1:

Computed GO TO

```
I = 3
GO TO (100, 120, 110), I
Statement 110 is the next
to be executed.
```

Example 2:

Assigned GO TO

```
ASSIGN 110 TO J
GO TO J, (110, 120, 100)
Statement 110 receives
control.
```

In example 2, it is not permissible to replace the ASSIGN statement with:

```
J = 110
GO TO J, (110, 120, 100)
```

IF Statement

IF (a) n_1, n_2, n_3 where n_1, n_2, n_3 are statement numbers and a is any expression.

Control is transferred to the statement whose number is n_1, n_2, or n_3 depending on whether the value of the expression (a) is less than zero, zero, or greater than zero, respectively.

Example:

```
IF (COUNT - 1) 2, 3, 4
```

will cause transfer to statement 2 when COUNT < 1

will cause transfer to statement 3 when COUNT = 1

will cause transfer to statement 4 when COUNT > 1

SENSE LIGHT

SENSE LIGHT i where i is an expression such that $1 \leq i \leq 24$

Sense lights are not physically present in the DDP-24 computer. To simulate these, the FORTRAN compiler reserves a memory word and considers each of the 24 bits of the word as a sense light. They are initially set to zero or OFF.

The SENSE LIGHT statement causes a bit or "sense light" to be set to 1 (turned ON). The expression is evaluated and if necessary, converted to an INTEGER. The resulting number must be from 1 to 24 and will turn ON the corresponding sense light. If the sense light was turned on by a prior sense light instruction, it will stay on.

Examples:	Result
SENSE LIGHT 3	Turns on sense light 3.
SENSE LIGHT K + 1	Turns on the sense light which is equal to the sum of K and 1.

IF (SENSE LIGHT)

IF(SENSE LIGHT i) n_1, n_2 where n_1 and n_2 are statement numbers and i is an expression such that $0 \leq i \leq 24$.

The expression i is evaluated as in the SENSE LIGHT statement. Control is transferred to the statement specified by n_1 if the sense light defined by the expression i is ON, (equal to 1). Otherwise control is transferred to statement n_2. If the sense light is ON, it is turned OFF. Consequently, the tested sense light is always OFF after execution of this statement. If i is zero, all the sense lights will be turned OFF.

IF (SENSE SWITCH)

IF (SENSE SWITCH i) n_1, n_2 where n_1, n_2 are statement numbers and i is an expression such that $1 \leq i \leq 6$.

This statement is similar to the preceding IF (SENSE LIGHT) statement. The sense switch on the console corresponding to the value i is interrogated and if it is ON, (up) control is transferred to statement n_1; if it is OFF (down) control is transferred to statement n_2.

IF ACCUMULATOR OVERFLOW

IF ACCUMULATOR OVERFLOW n_1, n_2 where n_1, n_2 are statement numbers.

Control is transferred to statement n_1 if accumulator overflow has occurred (overflow indicator is ON); otherwise control is transferred to statement n_2. The overflow indicator is turned off by this statement regardless of its prior condition.

IF QUOTIENT OVERFLOW

IF QUOTIENT OVERFLOW n_1, n_2

This statement has no affect on the DDP-24 FORTRAN system. However, it is included to allow programs written for other versions of FORTRAN to be compiled without requiring modification.

IF DIVIDE CHECK

IF DIVIDE CHECK n_1, n_2 where n_1, n_2 are statement numbers.

This statement transfers control to n_1 if an illegal division has been attempted (improper divide indicator is ON). Otherwise control is transferred to statement n_2. The improper divide indicator is turned off by this statement regardless of its prior condition.

THE DO STATEMENT

GENERAL FORM	EXAMPLES
"DO n i = m_1, m_2" or "DO n i = m_1, m_2, m_3" where n is a statement number, i is a variable, and m_1, m_2, m_3 are each an expression. If m_3 is not stated it is taken to be 1.	DO 30 I = 1, 10 DO 30 I = 1, M, 3 DO 30 I = 5, 2*N, J/2

The DO statement is a command to "DO the statements which follow, to and including the statement with statement number n, repeatedly, the first time with i = m_1 and with i increased by m_3 for each succeeding time; when i is equal to the highest of this sequence of values which does not exceed m_2, let control reach the statement following the statement with statement number n".

The range of a DO is the set of statements which will be executed repeatedly; it is the sequence of consecutive statements immediately following the DO, and including the statement numbered n.

The index of a DO is the variable i, which is controlled by the DO in such a way that its value begins at m_1 and is increased each time by m_3 until it is about to exceed m_2. Throughout the range it is available for computation, either as an ordinary variable or as the variable of a subcript. During the last execution of the range, the DO is said to be satisfied.

Suppose, for example, that control has reached statement 10 of the program

```
      .
      .
      .
10    DO 11 I = 1, 10
```

```
11    A (I)  = I*N(I)
12
      .
      .
      .
```

The range of the DO is statement 11, and the index is I. The DO sets I to 1 and control passes into the range. The value of 1*N(1) is computed, converted to REAL, and stored in A (1). Now, since statement 11 is the last statement in the range of the DO and the DO is unsatisfied, I is increased to 2 and control returns to the beginning of the range, also statement 11; 2N(2) is now computed and stored in A(2). This continues until statement 11 has been executed with I = 10. Since the DO is satisfied, control now passes to statement 12.

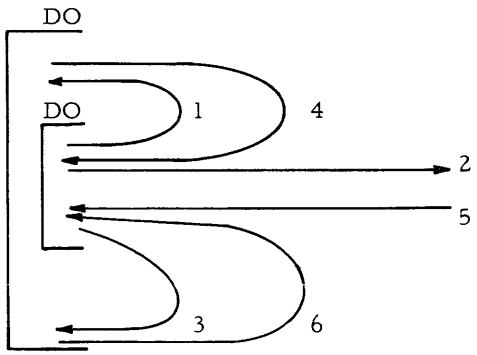
DO's WITHIN DO's. Among the statements in the range of a DO may be other DO statements. When this is so, the following rule must be observed.

Rule 1. If the range of a DO includes another DO, then all of the statements in the range of the latter must also be in the range of the former.

A set of DO's satisfying this rule is called a nest of DO's. No more than 25 DO's are permitted in one nest; there is no restriction on the number of levels in which they may be arranged.

TRANSFER OF CONTROL AND DO's. Transfers of control from and into the range of a DO are subject to the following rule:

Rule 2. No transfer is permitted into the range of any DO from outside its range. Thus, in the configuration below, 1, 2, and 3 are permitted transfers, but 4, 5, and 6 are not.



Exception. There is one situation in which control can be transferred into the range of a DO from outside its range. Suppose control is in the range of the innermost DO of a nest of DO's which are completely nested (i.e., every pair of DO's in the nest is such that one contains the other). Suppose also that control is transferred to a section of the program, completely outside the nest to which these DO's belong, which makes no change in any of the indexes (i's) or indexing parameters (m's) in the nest. This provision makes it possible to exit temporarily from the range of some DO's to execute a subroutine.

RESTRICTION ON ASSIGNED GO TO's IN THE RANGE OF A DO. When an assigned GO TO is in the range of a DO, it may not transfer into the range of any other DO. The statements to which the assigned GO TO statement may transfer may be (1) statements within the same DO range as the assigned GO TO statement, or (2) statements outside the range of any DO. Types (1) and (2) cannot be mixed in the same assigned GO TO statement; they must all be of type (1) or all of type (2).

RESTRICTION ON CALCULATIONS IN THE RANGE OF A DO. Almost every type of calculation is permitted in the range of a DO. Only one type of statement is not permitted, namely any which redefines the value of the index or of any of the indexing parameters (m's). In other words the indexing of a DO loop must be completely set before the range is entered.

The first statement in the range of a DO must be executable; i.e., it must not be a FORMAT, DIMENSION, EQUIVALENCE, FREQUENCY, or COMMON statement.

The following paragraph discusses restrictions on the last statement in the range of a DO.

EXIT FROM RANGE OF DO. When control leaves the range of a DO in the ordinary way (i.e., by the DO becoming satisfied and control passing on to the next statement after the range) the exit is said to be a normal exit. After a normal exit from a DO occurs, the value of the index controlled by that DO is not defined, and the index can not be used again until it is redefined.

However, if exit occurs by a transfer out of the range, the current value of the index remains available for any subsequent use. If exit occurs by a transfer which is in the ranges of several DO's, the current values of all the indexes controlled by those DO's are preserved for any subsequent use.

Transfer exits from the range of a DO in the form of a CALL statement to a handcoded subprogram can bring about changes in the preserved indexing values; a knowledge of the way in which the indexes are handled in the subprogram is required for safety.

The range of a DO cannot end with a transfer, i.e., GO TO, IF, etc., since this would leave the DO unsatisfied, i.e., have incorrect values for the indexes involved. For example,

```
      .
      .
      .
10    DO 11 I = 1, 100
11    IF (ARG - VALUE (I)) 12, 20, 12
12
      .
      .
      .
```

would not work. Instead, the statement CONTINUE (which means "do nothing") should be used as the last statement of such a range. (In the example it should be statement 12.)

SPECIAL DETAILS ABOUT DO STATEMENTS

TRIANGULAR INDEXING. Indexing such as

```
      DO 30 I = 1, 10
      DO 30 J = I, 10
```

or

```
      DO 40 I = 1, 10
      DO 40 J = 1, I
```

is permitted and simplifies work with triangular arrays. These are simply special cases of the fact that an index under control of a DO is available for general use as a variable.

The diagonal elements of an array may be picked out by the following type of indexing:

```
      DO 50 I = 1, 10
50    A(I, I, I) = some expression
```

STATUS OF THE CELL CONTAINING I. A DO loop with index I does not affect the contents of the object program storage location for I except under certain circumstances:

1) If an IF-type or GO TO-type transfer exit occurs from the range of the DO.

2) If I is used as a variable in the range of the DO.

3) If I is used as a subscript in combination with a relative constant whose value changes within the range of the DO. (A relative constant is a subscript the variable of which is not currently under control of a DO.)

Therefore, if a normal exit occurs from a DO to which cases 2 and 3 do not apply, the I cell contains what it did before the DO was encountered. After normal exit where 2 or 3 do apply, the cell I contains the first value of the I-sequence which exceeds m_2. After a transfer exit the I cell contains the current value of I.

What has just been said applies only when I is referred to as a variable. When it is referred to as a subscript, I is undefined after any normal exit and is the current value after any transfer exit.

PAUSE

PAUSE or PAUSE i where i is an expression

A halt will be generated in the object program by this statement. The expression i, if present, will be evaluated and displayed in the address portion of the Z-register on the console at the time of the halt. The value of i will be displayed in binary. Since the address portion of the Z-register is only 15 bits, the low order 15 bits of i will be displayed; therefore, the decimal value (INTEGER) of i should not exceed 32767. Computation will continue with the next statement when the start button is pressed.

STOP

STOP or STOP i where i is an expression

A halt will be generated in the object program by this statement. The expression i, if present, will be evaluated and displayed in the address portion of the Z-register on the console at the time of the halt. The value of i will be displayed in binary. Since the address portion of the Z-register is only 15 bits, the low order 15 bits of i will be displayed; therefore, the decimal value (INTEGER) of i should not exceed 32767. Computation cannot be continued by depressing the start button (if the start button is depressed, control will be transferred back to the STOP statement).

DATA TRANSMISSION

FORTRAN input-output statements are provided for the standard external devices available on the DDP-24. These include the typewriter, paper tape reader, and paper tape punch. In addition, provision is made for the inclusion of statements to control reading and writing on magnetic tape and punched cards.

SPECIFYING LISTS OF QUANTITIES

Several of the input-output statements include a list of the quantities to be transmitted. This list is ordered, and its order must be the same as the order in which the words of information exist (for input), or will exist (for output), in the external medium.

The formation and meaning of a list is best described by an example.

A, B(3), (C(I), D(I, K), I = 1, 10), ((E(I, J), I = 1, 10, 1), F(J, 3), J = 1, K)

Suppose that this list is used with an output statement. Then the information will be written in the external medium in the order

A, B(3), C(1), D(1, K), C(2), D(2, K), ..., C(10), D(10, K),
E(1, 1), E(2, 1), ..., E(10, 1), F(1, 3),
E(1, 2), E(2, 2), ..., E(10, 2), F(2, 3), ..., F(K, 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 just given.

Thus the list reads from left to right and with repetition of variables enclosed within parentheses. Only variables, and not constants, may be listed. The repetition is exactly that of a DO loop, as if each open parenthesis (except subscripting parentheses) were a DO, with indexing given immediately before the matching closing parentheses, and with range extending up to that indexing information. The order of the above list is the same as of the "program".

1) A

2) B(3)

3) DO 5I = 1, 10

4) C(I)

5) D(I, K)

6) DO 9 J = 1, K

7) DO 8 I = 1, 10, 1

8) E(I, J)

9) F(J, 3)

Notice that indexing information, as in DO's, consists of 3 constants or variables, and that the last of these may be omitted, in which case it is taken to be 1.

For a list of the form K, (A(K)) or K, (A(I), I = 1, K), where an index or indexing parameter itself appears earlier in the list of an input statement, the indexing will be carried out with the newly read-in value.

INPUT-OUTPUT IN MATRIX FORM

FORTRAN in effect treats variables according to conventional matrix practice. A list for either input or output in the form,

((A(I, J), J = 1, 3), I = 1, 2)

specifies that I x J items of information be transmitted in the order

$A_{1,1}$, $A_{2,1}$, $A_{1,2}$, $A_{2,2}$, $A_{1,3}$, $A_{2,3}$

This is the order in which the items are output; for input this is the order in which the data should be written on the data sheet. If it is desired to write the data by columns or to print the items by columns, the list is

((A(I, J), I = 1, 2), J = 1, 3)

INPUT-OUTPUT OF ENTIRE MATRICES

When input-output of an entire matrix is desired, then an abbreviated notation may be used for the list of the input-output statements; only the name of the array need be given and the indexing information may be omitted. Thus the list

A

is sufficient to read in all of the items for matrix A in their natural order. This natural order is considered to be

((A(I, J), J = 1, 3), I = 1, 2)

In such a case, FORTRAN will examine to see whether a DIMENSION statement has referred to the name of the variable. If such a reference is not made, only a single element will be transmitted.

INPUT-OUTPUT STATEMENTS

TYPE n, list	where n is the statement number of the FORMAT statement.

The variables specified in the list are converted in accordance with the FORMAT statement and typed on the typewriter.

ACCEPT n, list	where n is the statement number of the FORMAT statement.

This statement allows information to be input to computer storage by means of the console typewriter. The information is converted as defined by the FORMAT statement n and stored as specified by the variable list.

READ PAPER TAPE n, list	where n is the statement number of the FORMAT statement.

Information punched on paper tape is input to computer storage and converted as stated in the FORMAT statement. It is stored as ordered by the variable list.

PUNCH TAPE n, list	where n is the statement number of the FORMAT statement.

The variables in the list are formatted and positioned as specified by the FORMAT statement and punched on paper tape.

The following input-output statements are applicable to expanded systems with additional input-output capabilities.

READ INPUT TAPE i, n, list	where n is the statement number of the FORMAT statement for the specified variable list and i is an expression.

The expression i is evaluated and converted to an integer. The resulting number becomes the number of the magnetic tape unit to be selected. A BCD tape is used and information is input

as defined by the FORMAT statement. The tape may have been written by a WRITE OUTPUT TAPE statement which is defined later in the chapter, or may have been generated off-line.

WRITE OUTPUT TAPE i, n, list	where i is an expression and n is the statement number of the FORMAT statement for the variable list.

The variables in the list are converted to BCD and written on magnetic tape under control of the FORMAT statement. The integer value of the expression i defines the magnetic tape unit which is selected for the operation. The BCD tape produced may subsequently be read by the READ INPUT TAPE statement or listed by an off-line process.

READ TAPE i, list	where i is an expression

Binary information is read into the computer from the magnetic tape unit specified by the expression i. A FORMAT statement is not referenced as the information read must always be binary as prepared by the WRITE statement which is described later. The information is stored according to the variable list.

WRITE TAPE i, list	where i is an expression.

Information as specified by the list of variables is written in binary form on magnetic tape. The integer value of expression i is used to select the magnetic tape unit which receives the information. A FORMAT statement is not required as the data is written in binary and no conversion is made.

Tapes generated by this instruction may be read by the READ TAPE statement.

REWIND i	where i is an expression

The magnetic tape on the tape unit specified by the integer value of expression i is positioned to the beginning of the tape by this statement.

BACKSPACE i	where i is an expression

This statement is used to move a magnetic tape back one record. Magnetic tapes are written in a forward direction from the beginning to the end. BACKSPACE moves the tape in the opposite direction toward the beginning. No information is transmitted to or from computer storage by this statement. The tape unit selected for the operation is specified by the integer value of expression i. Note: An end-of-file mark is one record.

END FILE i	where i is one expression

This statement causes an end-of-file mark (as a single record) to be written on magnetic tape. The integer value of the expression i is the number of the magnetic tape unit selected.

READ n, list	where n is the statement number of the FORMAT statement for the specified variable list.

Information from punched cards is input to computer storage. The amount of information accepted and its conversion is determined by the FORMAT statement n. The information may subsequently be referred to in the program by the variable names in the list. Cards are read continuously until the list is satisfied.

PUNCH n, list	where n is the statement number of the FORMAT statement for the variable list.

This statement is used to punch cards which contain the values of the variables in the list. Each variable is converted to punches on the card as specified by the FORMAT statement. Cards are punched continuously until the list is satisfied.

PRINT n, list — where n is the statement number of the FORMAT statement for the variable list.

The variables in the statement list are converted and printed on the on-line printer as defined by the FORMAT statement.

READ DRUM i_1, i_2, list — where i_1 and i_2 are expressions

The integer value of expression i_1 is the number of the drum unit which is to be selected for reading binary information. Expression i_2 specifies the first word address on the drum of the information to be read and stored according to the variable list. Since the information transfer is binary word to binary word, a FORMAT statement is not allowed.

WRITE DRUM i_1, i_2, list — where i_1 and i_2 are expressions

The variables in the list are written in binary on the drum unit which is specified by the integer value of expression i_1. The integer value of expression i_2 is the address on the drum of the first variable in the list. Following variables are stored sequentially until the list is satisfied. No conversion is possible in this statement; therefore, a FORMAT statement is not allowed.

FORMAT

Some I-O statements and lists are associated with a corresponding FORMAT statement. This statement describes what conversion is necessary for the data in the list. Only one FORMAT statement is allowed for each I-O statement. It must be assigned a statement number; and that number precedes the data list in the I-O statement.

FORMAT statements are required for input-output statements which concern:

1) Paper tape

2) Console typewriter

3) Magnetic tape written in BCD

4) Punched cards

Input-output statements which transfer binary information cannot be controlled by FORMAT statements.

RECORDS

The information being transmitted between computer storage and external media by one input-output statement is separated into physical groups called records. The definition of a record varies with the I-O device. The following chart defines a record for each I-O unit. It is the programmer's responsibility to avoid exceeding the maximum record size for a given device.

Unit	Record Definition
Typewriter	One line of type as terminated by a carriage return character
Paper tape	Punched information terminated by a carriage return character
Magnetic tape	Binary or BCD logical tape record. The FORTRAN programmer need not be concerned with the physical length

Unit	Record Definition
Card Reader	One card, 80 columns
Card Punch	One card, 80 columns

External records may contain the values of several variables; each variable is considered a field of information. In order to specify the manner of conversion on input or output, it is necessary to state the size or width of the data field and if the variable is floating point, the position of the decimal point. The width of the field is the total number of characters necessary to describe the information on a coding form. However, fields may be larger if leading blanks are included, or smaller if truncation is desired. The 3C FORTRAN compiler relieves the programmer of determining the actual binary size of the field as it is stored in memory and automatically allocates the necessary storage registers.

FORMAT

The form of the FORMAT statement is

FORMAT (S_1, S_2, ---, S_n) where S_i are descriptions of the external form of the variables comprising a record. The S_i provide the compiler with the information necessary for conversion from and to external form.

CONVERSION

On reading or writing of data, 3C FORTRAN automatically converts numbers from external to internal or internal to external form. The FORMAT statements are used to describe the external forms in which data may appear. 3C FORTRAN accepts data in five different forms:

1) Floating-point decimal. These numbers are characterized as containing a decimal scale factor. The scale factor may be indicated within the number by the presence of the letter E, an algebraic sign, or both. For example, .314E1, .314+1, .314E+1, .314E01 all are permissible.

2) Fixed-point decimal. These numbers may contain either an actual or implied decimal point but may not contain a decimal scale factor.

3) Integers.

4) Octal numbers.

5) Alphanumeric. Alphanumeric information in external BCD form may be read into or out of a memory location identified by a variable name. Up to four characters may be stored in a cell.

BASIC FIELD SPECIFICATIONS

A FORMAT specification describes the line to be converted by giving, for each field in the line (from left to right, beginning with the first character) a basic field specification written in the form:

nKw.d

where

n Is a positive integer indicating the number of successive fields within one unit record which are to be converted according to the same specification. If n = 1, it may be omitted.

K Is a control character specifying the type of conversion to be used. This character may be I, E, F, O, A, X, or H.

w Is the width of the field.

d Is the number of positions in the field which appear to the right of the decimal point; used only with E- and F-type conversions. (Note: d is treated modulo 12.)

Within the unit record, field specifications are separated by commas:

Iw, Ew.d, nX, Fw.d, nAw

(Exception: A comma need not follow a field specified by an H or X control character)

SUMMARY OF CONTROL CHARACTERS. There are seven different types of control characters, five of which provide for the conversion of data between the internal machine language and the external notation.

INTERNAL	TYPE	EXTERNAL
INTEGER variable	I	Decimal integer
REAL variable	E	Floating-point, decimal
REAL variable	F	Fixed-point, decimal
INTEGER, REAL or LOGICAL variable	O	Octal integer
BCD variable	A	Alphanumeric characters

A sixth control character, X, provides for the skipping of characters in input or the specification of blank characters in output.

The last character, H, designates Hollerith fields; it may be used to output alphanumerical characters originating in the source program and for carriage control in printing or typing.

SUCCESSIVE FIELDS (n). An example of how "n" may be used before the control character to type n successive fields within one record is shown by the statement FORMAT (1X, I2, 3E12.4), which might give:

```
27 -0.9321E 02  -0.7580E-02  0.5536E 00
```

FIELD WIDTH (w). The field widths may be made greater than necessary to provide spacing blanks between the items on a line. Thus, a field specification of nK12, where only 4 digits are to be printed, would result in 8 blanks preceding the digits. Within each field the printed output will always appear in the rightmost positions.

For printed or typed output, the FORMAT statement should always provide for a carriage control character as the first character of the line. In many instances, this will be a blank.

I-TYPE CONVERSION

Field specification: Iw or nIw

The number of characters specified by w will be converted as a decimal integer. On

input, numbers for I-conversion will be treated modulo 8,388,608. No plus signs or decimal points will be printed or typed.

E-TYPE CONVERSION

Field specification: Ew.d or nEw.d

The number of characters specified by w will be converted as a floating-point number, with the number of digits specified by d to the right of the decimal point. For example, the statement

FORMAT (X, I2, E12.4, E15.4)

might give the line

27 -0.9321E 02 -0.7580E-02

In this case, there is one blank following the 27, one blank after the first E (automatically supplied except in cases of a negative exponent, when a minus sign will appear), and four blanks after the 02. (The X control character may also be used, of course, to provide blank fields.)

F-TYPE CONVERSION

Basic field specification: Fw.d or nFw.d

The number of characters specified by w will be converted as a fixed-point number, with the number of digits specified by d to the right of the decimal point.

SCALE FACTOR. To permit more general use of F-conversion, a scale factor followed by the letter P may precede the specification. The scale factor is so defined that

$$\text{Converted number} = \text{internal number} \times 10^{\text{scale factor}}$$

Thus, the statement FORMAT (X, I2, 1P3F11.3), used with the data in the previous example would give

27 -932.096 -0.076 5.536

while FORMAT (X, I2, -1P3F11.3) would give

27 -9.321 -0.001 0.055

A positive scale factor may also be used with E-conversion to increase the number and decrease the exponent. Thus FORMAT (X, I2, 1P3E12.4) would give, with the same data

27 -9.3210E 01 -7.5804E-03 5.5361E-01

Note: The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it will hold for all E- and F-conversions following the scale factor within the same FORMAT statement. This applies to both single-record and multi-record formats. Once a scale factor has been given, a subsequent scale factor of zero in the same FORMAT statement must be specified by OP. Scale factors have no effect on I-conversion.

RESTRICTIONS ON F-TYPE CONVERSION. There are some restrictions in the use of F-type conversion for output. This F-type conversion should not be used with numbers having an absolute value greater than 2^{38} Numbers exceeding 2^{38} in absolute value will be incorrect by a factor of a power of 2. Also, numbers whose absolute values are less than 0.5 may have

truncation errors that will affect the last significant digit when F-type conversion is used for output. The scale factor, P, used with an F-type conversion can circumvent these restrictions.

O-TYPE CONVERSION

Basic field specification: Ow or nOw

On input, the number of digits specified by w will be stored. If w exceeds 8, only the 8 rightmost characters will be significant. If w is less than 8, the number will be right-adjusted and filled out with zeros. Leading and trailing blanks will be treated as zeros.

On output, the number of characters specified by w will be the result of conversion to octal. If w exceeds 8, the excess will be blanks. If w is less than 8, only the w rightmost digits will be significant. Leading zeros will be converted to blanks, and the number will be signed, if negative. However, 8 or more significant digits will be unsigned.

A-TYPE CONVERSION

Basic field specification: Aw or nAw

The use of the control character A results, at input time, in the storage in BCD form of the number of alphanumeric characters specified by w. If w is greater than 4, only the 4 rightmost characters will be stored. If w is less than 4, the w leftmost characters will be transmitted and the word will be filled with blanks. (In BCD, a blank is considered a significant character.)

At output time, the number of characters specified by w will be transmitted without conversion. If w is greater than 4, only the 4 rightmost characters will be transmitted, preceded by w - 4 blanks. If w is less than 4, the w leftmost characters of the word will be transmitted.

CONTROL CHARACTER X

Basic field specification: nX

On input, the number of characters specified by n will be skipped. On output, n characters of output will be blanks.

CONTROL CHARACTER H

Basic field specification: nH (No comma is used to separate this type of field specification from a following field specification.)

The control character H may be used to designate a Hollerith (alphanumeric) field, in which case English text will be printed in it. The field width, followed by the desired characters, should appear in the appropriate place in the specification. For example,

```
FORMAT (5H XY = F8.3, 4H Z = F6.2, 9H  W/AF = F7.3)
```

would give

```
XY = -93.210 Z = -0.01  W/AF = 0.554
```

Notice that any Hollerith characters, including blanks, may be printed or typed. This is the sole exception to the statement that FORTRAN ignores blanks.

It is possible to print or type Hollerith information only, by giving no list with the input-output statement and setting up no field specifications containing other types of control characters (except X) in the FORMAT statement.

USING THE H CONTROL CHARACTER FOR CARRIAGE CONTROL. WRITE OUTPUT TAPE prepared a tape which may be later used to obtain off-line printed output. The off-line printer is manually set to operate in one of three modes: single space, double space, and program control. Under program control, which gives the greatest flexibility, the first character of each BCD unit record controls the carriage, and that character is not printed. The control characters and their effects are

Blank	Single space before printing
0	Double space before printing

Thus, a FORMAT specification for WRITE OUTPUT TAPE for printing with program control will usually begin with 1H followed by the appropriate BCD character. This is true for the typewriter and on-line printer also.

REPETITION OF GROUPS. A limited parenthetical expression is permitted in order to enable repetition of data fields according to certain FORMAT specifications within a longer FORMAT statement specification. Thus FORMAT (2(F10.6, E10.2), I4) is equivalent to FORMAT (F10.6, E10.2, F10.6, E10.2, I4). More than one level of parentheses within the outer parentheses is not permitted.

Note, however, that in some cases the result may not be precisely what you might expect. In the following example:

```
40      FORMAT (1H0, I6, 2F12.0, 3(I3, 2X, 2A4), I5, E12.7)
```

if the FORMAT specification has been completed and data items remain to be transmitted, the format will repeat from the last opening parenthesis; i.e., the next specifications will be I3, 2X, 2A4, I3, 2X, 2A4, I3, 2X, 2A4, I5, E12.7, I3, 2X, . . . (Note, however, that the last closing parenthesis is treated as the end of a record each time it is encountered. The first character of the next piece of data to be printed or typed (with format I3 in the example) will be examined and treated as a carriage control character rather than as a character to be printed or typed).

MULTI-RECORD FORMATS. To deal with a block of output, a FORMAT specification may have several different line formats, separated by a slash /. Thus

```
10      FORMAT (1H 3F9.2, 2F10.4/8E14.5)
```

would specify a block in which lines 1, 3, 5,. . . have format 3F9.2, 2F10.4, and lines 2, 4, 6, . . . have format 8E14.5.

CAUTION

If the list of a READ PAPER TAPE statement is such that the last item read is directly followed by a /, the following line will be read and then the list will be examined. Since there are no more items on the list, the contents of that line will be lost. For example:

```
        READ PAPER TAPE 100, A, B, C, D, E, F
100     FORMAT (6E12.8/2F12.8)
        READ PAPER TAPE 100, G, H, O, P, Q, R, S, T
```

The contents of the line following F will be lost.

If data items remain to be transmitted after the format specification has been completely "used," the format repeats from the last opening parenthesis or (if no second pair of parenthesis is present) from the beginning of the entire format specification.

Both the slash and the last closing parenthesis of a FORMAT statement signal the end of a record. If no format specifications are written between two such end-of-record indications, the carriage moves one line down on the page and, in effect, writes a completely blank record, or line. N + 1 consecutive end-of-record indications produce N blank lines. The following example shows the effect of the use of multiple slashes (and should discourage the programmer from using this device indiscriminately).

```
      TYPE 20, XMN
20    FORMAT (1H0 12A4)
      TYPE 30, A, B, C
30    FORMAT (//1H0 6E12.8//1H0 2F12.4, 3E14.5)
```

Closing parenthesis indicates end of record. Typing takes place. The carriage does not move.

First slash indicates end of another record. Since nothing has been set up for typing since the last end-of-record indication, the carriage moves one line and a "blank" record is written. The second slash similarly moves the carriage one more line.

Resultant Line Setup

The "0" carriage control character moves the carriage two lines.

```
PRINTED LINE
Blank line
Blank line
Blank line
PRINTED LINE
Blank line
PRINTED LINE
```

The third slash indicates the end of a record. Typing takes place but the carriage does not move.

The fourth slash signals the end of another record; the carriage moves one line and a "blank" record is written.

The (space) carriage control character moves the carriage one line. Typing will take place on this line.

RELATIONSHIP OF FORMAT STATEMENT TO LIST. The FORMAT statement indicates, among other things, the size of each record to be transmitted. In this connection, it must be remembered that the FORMAT statement is used in conjunction with the list of some particular input-output statement, except when a FORMAT statement consists entirely of Hollerith fields. In other words, all specifications in the FORMAT statement except those with the control characters X or H are used to control the transmission of data to or from records.

During input-output of data, the object program in effect scans the FORMAT statement to which the relevant input-output statement refers. When a specification for a field is found, and list items remain to be transmitted, input-output according to the specification takes place, and scanning of the FORMAT statement resumes. If no items remain in the list, transmission ceases and execution of that particular input-output statement has then been completed. Thus if you come to the end of a list, and simultaneously the next field specification in the FORMAT statement contains some control character other than X or H (or the end of the FORMAT statement has been reached), input-output will be brought to an end.

DATA INPUT TO THE OBJECT PROGRAM

Decimal input data to be read by means of a READ PAPER TAPE statement when the object program is executed must be in essentially the format specified by the associated FORMAT statement. Thus a line to be read with FORMAT (I2, E12.4, F10.4) might be punched

```
27 -0.9321E 02      -0.0076
```

Within each field all information must be pushed to the extreme right, except when the decimal point is written. (See paragraph 2) below). Positive signs may be indicated by a blank or a + punch; negative signs should be punched with an - punch. Blanks in numeric fields are regarded as zeros. Numbers for E- and F-conversion may contain any number of digits, but only 11 digits of accuracy will be retained.

To permit economy in punching, certain relaxations in input data format are permitted.

1) Numbers for E-conversion need not have 4 columns devoted to the exponent field. The start of the exponent field must be marked by an E, or, if that is omitted, by a + or - (not a blank). Thus E2, E02, +2, +02, E 02, and E+02 are all permissible exponent fields. However, the exponent field must be right-adjusted; i.e., written at the extreme right of the field.

2) A number for E- or F-conversion need not have its decimal point punched. If it is not punched, the FORMAT specification will effectively supply it; for example, -09321+2 with E12.4 will be treated as if the decimal point had been punched 4 places before the start of the exponent field, that is, between the 0 and the 9. If the decimal point is punched, its position overrides the value of d given in the FORMAT specification.

3) If the data are always positive in the particular field, a column need not be allocated for the sign (regardless of the format).

SPECIFICATION STATEMENTS

Specification statements are used to allocate storage and define data formats. Their use in a FORTRAN program is optional (except to define dimensional arrays). They allow the programmer to gain program efficiency based on a knowledge of the individual data items to be operated upon.

Specification statements do not generate executable machine language instructions; however, since they dictate the form and order of the instructions that are generated, they must appear before the first executable FORTRAN statement.

TYPE SPECIFICATIONS

Three type specifications are available in 3C FORTRAN.

1) REAL

REAL a, b, c . . .

The REAL statement defines all variables appearing on the right as floating-point numbers.

2) INTEGER

INTEGER a, b, c . . .

The INTEGER statement defines all variables appearing on the right as 23-bit signed fixed-point integers.

3) LOGICAL

LOGICAL a, b, c . . .

The LOGICAL statement defines all variables appearing on the right as 24-bit logical quantities. Also, any variable defined as LOGICAL and used in an arithmeiic operation is treated as Boolean (one or zero).

Data items not defined by a type statement are assigned types on the basis of the first letter of their names.

STORAGE SPECIFICATIONS

Four storage specification statements are available in FORTRAN.

1) DIMENSION

DIMENSION a (N_1, N_2, N_3), b (N_1, N_2, N_3), . . .

The DIMENSION statement defines data arrays. The numbers in parentheses, following the variable names, are called subscripts; each variable may have from one to three non-zero subscripts. The subscript integers define the maximum size of the array. For example, the statement:

DIMENSION NAME1 (5, 10, 4), NAME2 (2, 4)

defines NAME1 as being a cubic array containing a maximum of 4 levels of information. The items appear in 5 rows of 10 items in each level. NAME2 is a matrix containing 2 rows of 4 items each. A storage area equal to the product of the subscripts is allocated for each data item defined by a DIMENSION statement. (Real numbers, being double length, would be assigned twice the product of the subscripts). NAME1 would be assigned 200 words of storage (400 if the items are real); NAME2 would be assigned 8 (or 16) words of storage.

Arrays are stored in rows, forward in memory. For example, the matrix A:

1 2 3

4 5 6

defined by the statement:

DIMENSION A (2, 3)

would be stored in sequential higher numbered memory positions as:

1, 2, 3, 4, 5, 6

2) EQUIVALENCE

EQUIVALENCE (a, b, . . .), (c, d . . .), . . .

The EQUIVALENCE statement allows the programmer to utilize the available storage more efficiently by assigning separate data items or arrays to the same memory locations. A data item appearing in an equivalence statement may be subscripted, for example,

EQUIVALENCE (NAME1, NAME2 (5))

equates the memory location of NAME1, and the fifth item in the array, NAME2. Care must be observed when assigning equivalent memory locations between INTEGER and REAL variables since each REAL number uses two storage locations, but an INTEGER requires only one location.

3) COMMON

The COMMON statement allows the programmer to assign data items and arrays to fixed memory positions. This allows the main program and subroutines to use the same data area without the need for parameter substitution. The form of the COMMON statement is

COMMON a, b, - - -

Array names are not subscripted.

The COMMON storage area is fixed in the upper part of the machine memory; that is, in the area of higher numbered memory addresses. Names assigned to COMMON, reserve memory locations in the opposite order of their appearance; however, arrays are stored forward (see DIMENSION). Array names appearing in COMMON statements must also appear in DIMENSION statements.

4) FREQUENCY

The FREQUENCY statement has no affect on the compilation in DDP-24 FORTRAN. It is included to allow programs written for other versions of FORTRAN to be compiled without requiring modification.

IN-LINE MACHINE LANGUAGE CODING

In-line machine language coding with FORTRAN statements is allowed in DDP-24 FORTRAN. This capability should not be confused with DAP coding, since the two are entirely different. The only DAP pseudo-operations that are allowed by the FORTRAN compiler are PZE and MZE. All other pseudo-operations will not be translated. In addition, literals are not allowed in the machine language coding.

Machine language is far removed from the algebraic language of FORTRAN. If the programmer wishes to use machine language in his FORTRAN program, it is assumed that he is entirely aware of this difference and that he is familiar with the intricate workings of the compiler and the computer. Therefore, the following is devoted merely to the rules that must be followed when using machine language coding and not to the explanation of machine language programming.

The punching of an S in the first position of the coding line indicates that the line of coding is machine language. This S must be repeated on each line that machine language is used. Further, a machine language statement can only be used after an entire FORTRAN statement has been completed (i.e., a machine language statement could not be used between continuation lines of a FORTRAN statement).

After the initial punching of S in position one of the coding line, a machine language statement must begin in position six (the position usually reserved for continuation codes in FORTRAN statements).

In addition to PZE (plus zero) and MZE (minus zero), any three-letter mnemonic listed in the DDP-24 REFERENCE MANUAL may be used as an instruction. At the conclusion of the three letters, an asterisk (*) should be used if indirect addressing is desired (position nine).

The address portion begins in position 12. If a symbol is used in the address portion, it must refer to a variable data location. If the variable has been used in the FORTRAN statements, the address of that variable will be inserted in the machine language instruction. If the variable has not been used in the FORTRAN statements, a location will be assigned by FORTRAN (two if the variable is REAL) and the address will be inserted in the instruction. Thus, it is not possible to define data locations arbitrarily, the compiler must make the assignments. A decimal or octal (preceded by an apostrophe) integer may be used in the address portion if absolute reference is desired to a memory location. If it is desired to reference another instruction in a sequence of machine language statements, relative addressing must be used (e.g., *+3, *-14, etc.).

If indexing is desired, the address portion should be followed by a comma and numeric index register designation.

If comments are desired, they may be placed on the same line starting in position 30.

Example:

It is desired to read a fixed-point (6-digit) decimal number from the typewriter and place this value in every element of the A array (REAL) using a combination of FORTRAN and machine language coding. A program to accomplish this could be:

```
C       PROGRAM FOR ILLUSTRATING USE OF MACHINE LANGUAGE CODING
        DIMENSION A(50)
   40   TYPE 50
   50   FORMAT(//// 35H YOU MAY TYPE IN THE 6-DIGIT NUMBER////)
        ACCEPT 60, B
   60   FORMAT(F7.0)
        TYPE 70, B
   70   FORMAT(//// 15H THE NUMBER IS F7.0,32H IF THIS IS CORRECT, PRESS START/
     X              48H IF NOT, PLACE SENSE SWITCH 1 UP AND PRESS START////)
   75   PAUSE
        IF(SENSE SWITCH 1) 40, 80
   80   CONTINUE
S      LDA    B           GET HIGH ORDER PART OF B
S      LDB    B+1         GET LOW ORDER PART OF B
S      LDX    -100,1      INITIALIZE INDEX REGISTER
S      STA    A+100,1     STORE HIGH ORDER PART OF B
S      JXI    *+1,1       INCREMENT INDEX REGISTER
S      STB    A+100,1     STORE LOW ORDER PART OF B
S      JXI    *-3,1       INCREMENT INDEX REGISTER AND JUMP BACK IF NOT ZERO
        STOP
        END
```

APPENDIX A

STATEMENT SUMMARY

APPENDIX B

BUILT-IN FUNCTIONS

NAME	ARGUMENT MODE	FUNCTION MODE	NUMBER OF ARGUMENTS	PURPOSE AND DEFINITION
ABSF XABSF	Floating Fixed	Floating Fixed	1	Computes absolute value of Argument, |Argument|
DIMF XDIMF	Floating Fixed	Floating Fixed	2	Computes positive difference ARG_1 minus the smaller of the two arguments ARG_1 - min (ARG_1, ARG_2)
FLOATF	Fixed	Floating	1	Changes a fixed point to a floating point number.
XFIXF	Floating	Fixed	1	Changes a floating point to a fixed point number. Same as XINTF.
INTF XINTF	Floating Floating	Floating Fixed	1	Truncates the argument to the largest possible integer value which is $\leq$ argument. Sign if unchanged
MAXOF MAXIF XMAXOF XMAXIF	Fixed Floating Fixed Floating	Floating Floating Fixed Fixed	≥ 2	Determines argumetn with largest value MAX(ARG_1, ARG_2; . . . ARG_n)
MINOF MINIF XMINOF XMAXIF	Fixed Floating Fixed Floating	Floating Floating Fixed Fixed	≥ 2	Determines the smallest argument MIN (ARG_1, . . . , ARG_n)
MODF XMODF	Floating Fixed	Floating Fixed	2	Calculates remainder of $ARG_1 \div ARG_2$ FOR MODF the remainder is defined as $ARG_1 - ARG_1/ARG_2 \; ARG_2$ where ARG_1/ARG_2 is the integer quotient. For XMODF the true remainder is obtained.
SIGNF XSIGNF	Floating Fixed	Floating Fixed	2	Transfers sign of ARG_1 to the absolute value of ARG_2 ARG_1^2 times $\lvert ARG_2 \rvert$

APPENDIX C

TYPEWRITER CODES

OCTAL CODE	TYPEWRITER L/C	TYPEWRITER U/C	PAPER TAPE 8	7	6	P	4	S	3	2	1
00	Ø	b				O		•			
01	1							•			O
02	2							•		O	
03	3					O		•		O	O
04	4	:						•	O		
05	5	@				O		•	O		O
06	6	√				O		•	O	O	
07	7	>						•	O	O	O
10	8						O	•			
11	9					O	O	•			O
13	#	-					O	•		O	O
20	*	¢			O			•			
21	/				O	O		•			O
22	S				O	O		•		O	
23	T				O			•		O	O
24	U	=			O	O		•	O		
25	V	%			O			•	O		O
26	W	"			O			•	O	O	
27	X	'			O	O		•	O	O	O
30	Y				O	O	O	•			
31	Z				O		O	•			O
33	'				O	O	O	•		O	O
40	-	-		O				•			
41	J				O		O	•			O
42	K				O		O	•		O	
43	L				O			•		O	O
44	M	)			O		O	•	O		
45	N	*			O			•	O		O
46	O	△			O			•	O	O	
47	P	;			O		O	•	O	O	O
50	Q				O		O	O	•		
51	R				O			O	•		O
52	tab				O			O	•		O
53	$				O		O	O	•		O	O
54	backspace	↙			O			O	•	O	
56	space				O		O	O	•	O	O
60	&	&			O	O	O		•		
61	A				O	O			•			O
62	B				O	O			•		O	
63	C				O	O	O		•		O	O
64	D	(			O	O			•	O		
65	E	□			O	O	O		•	O		O
66	F	/			O	O	O		•	O	O	
67	G	<			O	O			•	O	O	O
70	H				O	O		O	•			
71	I				O	O	O	O	•			O
73	.	∨			O	O		O	•		O	O
74	lower shift				O	O	O	O	•	O		
75	upper shift				O	O		O	•	O		O
76	car. return				O	O		O	•	O	O	
77	line feed				O	O	O	O	•	O	O	O
stop	backspace		O	O		O	O	•	O			