R Polivka

S

IBM

General Information Manual

FORTRAN

IBM®

General Information Manual
FORTRAN

Minor Revision
(December 1961)

This edition, F28-8074-1, supersedes but does not obsolete the previous edition, F28-8074. Changes have been made only to correct errors in the previous edition. Paragraphs that have been changed are indicated by the symbol • in the margin.

PREFACE

Most electronic computers are designed to respond to special command codes called "machine languages." These machine languages vary among computers; they are often difficult to work with and hard to learn. The IBM FORmula TRANslating system, FORTRAN, eliminates most of the difficulties associated with machine languages for problems which are primarily mathematical. FORTRAN is available for the IBM 650, 1620, 705, 7070, 704, 709, and 7090.

This manual will discuss FORTRAN and will prepare the reader to use the facilities which it provides. Parts I, II, and V are intended for all who will use FORTRAN. Part III is of limited value to 650 FORTRAN users, and is designed primarily for other users. Part IV is an analysis of individual FORTRAN processor requirements.

It is anticipated that the reader will be able to write some programs after reading this manual; there will, however, be many additional aspects which he will want to learn. For example, some FORTRAN processors are used within an operating system; such usage usually requires that the programmer be familiar with that operating system. Detailed instructions for the use of FORTRAN are given in reference manuals which are specifically oriented to each class of IBM computers for which FORTRAN is available. These publications are listed in Appendix A.

The use of this manual presupposes little familiarity with machine configurations, input/output devices, and the stored program concept. However, the reader who is completely unfamiliar with these terms may be interested in reading the Introduction to IBM Data Processing Systems, form F22-6517.

TABLE OF CONTENTS

PART I: GENERAL CONCEPTS

CHAPTER 1: INTRODUCTION

Most IBM computers are digital computers; that is, they count. This counting function, however, has greatly evolved since the invention of the abacus, the adding machine, and the desk calculator. Today's high-speed, electronic digital computers can handle alphabetic data as well as numerical data, and instead of being restricted to simple mathematical operations, can perform complicated calculations, manipulate information, and make logical decisions, all at tremendous speed.

ORGANIZATION OF DIGITAL COMPUTERS

A digital computer has the following elements in one computing system:

1. Input. Digital computers accept numbers, letters, and symbols. Information is usually fed into the system from punched cards, punched paper tape, or magnetic tape, or inserted manually from a keyboard or switches.

2. Control. The computer must operate under the direction of a control unit. The sequence of steps to be performed must be translated into detailed instructions which the computer can understand. A series of instructions is called a program. When it is retained in a storage device, it is called a stored program. These coded instructions in storage are available to the control unit as needed to direct and complete an entire sequence of operations. Special instructions may enable the logical-arithmetic unit to make decisions based on intermediate results; these decisions allow the computer to select the proper course among several alternatives for solving a problem.

3. Storage. Data can be internally stored by electro-mechanical, magnetic, or electronic devices, until needed. This information is stored in a manner quite similar to the way music or speech is stored on a tape for playback on a tape recorder, although the notation used is quite different. Stored information is accessible, can be referred to once or many times, and can be replaced whenever desired. The information stored by the computer can be original data, intermediate results, reference tables, or instructions. Each storage location is identified by an individual location number which is called an address. By means of these numerical addresses, a computer can locate data and instructions as needed during the course of a problem.

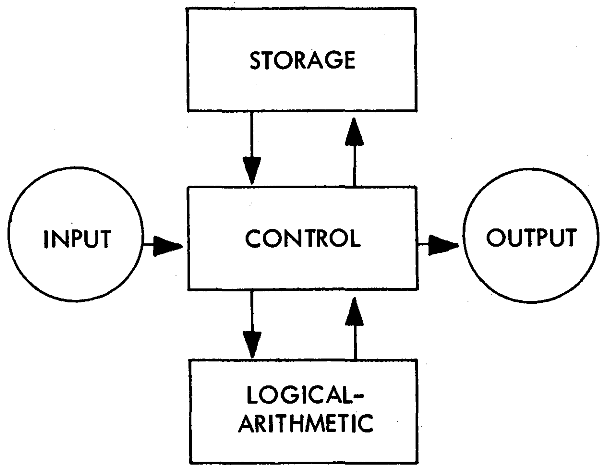
   The speed of computer operation is largely dependent on the access time — the length of time required to obtain a number from storage and make it available to other units of the computer system.

4. Logical-Arithmetic. The logical-arithmetic unit can add, subtract, multiply, divide, and compare numbers in a manner similar to a desk calculator, but at lightning speed. Complex calculations are usually

combinations of these basic operations. The logical-arithmetic unit can make logical decisions. It can distinguish positive, negative, and zero values and transfer this information to other units of the computer.

5. Output. After doing its work, the computer can produce answers in several forms. Results may be punched into cards, recorded on magnetic tape, or printed in report form. Printers provide high-speed computer output by printing an entire line of information at one time.

The organization of these components to form a computer may be illustrated as follows:



The functioning of the elements of a computer may be compared to the steps required for solving a problem by paper and pencil methods. The input would correspond to the information given in the problem. A knowledge of arithmetic controls the handling of the problem. The logical-arithmetic unit performs the same function as manual calculations. Storage may be compared to the work papers on which intermediate answers are noted. The answers are the output.

THE STORED PROGRAM

"Program" is just another way of saying "series of instructions and fixed data." A program must define in complete detail just what a computer is to do, under every conceivable combination of circumstances, with data which is subsequently fed into it.

One such instruction may tell what operation to perform and where to locate the number on which to perform it; another will tell what to do with the result. Stored in the computer's control unit, in the proper sequence necessary to accomplish a given task, these instructions form the stored program.

The various operations covered in these instructions are usually stated in a numerical or alphabetic code. Thus, the operations in a simple problem might be designated as follows:

Operation Code	Operation
10	Add
11	Subtract
21	Store the result

These operation codes might be used in a stored program in the following manner:

	Operation Code	Storage Location
Instruction #36	10	0679
Instruction #37	10	0680
Instruction #38	11	0681
Instruction #39	21	1027

Instruction #36 tells the computer to add the number stored at location 0679.

Instruction #37 — to add the number stored at location 0680.

Instruction #38 — to subtract the number stored at location 0681.

Instruction #39 — to store the result of the two additions and the one subtraction at location 1027.

The same program, coded in FORTRAN, might be:

```
D = A+X-Y
```

The number of instructions required for the complete solution of a problem may be a few hundred or many thousands, depending upon the problem. The computer refers to them one after another, or it can be instructed to repeat, modify or skip over certain instructions, depending on intermediate results or circumstances. However, such circumstances must be anticipated, and appropriate instructions included in the program.

The ability to repeat operations, usually called looping, combined with the other facilities of modifying and skipping over instructions, permits a significant reduction in the number of instructions required to perform any given job. For example, suppose two sets of numbers exist and it is desired to add the corresponding numbers of each set together. Instructions may be written to add the first number of the first set to the first number of the second set, and then to repeat this operation with the second, third, fourth, etc., numbers of each set; a few instructions may cause thousands of additions.

The decision-making ability enables computers to handle exceptions to standard procedures. Since a system will "remember" instructions for dealing with the exceptions, it can be made to handle automatically any situation that develops.

Up to this point, the computer has been thought of as a separate piece of equipment that is used by itself. However, in actual practice, the computer is used in conjunction with other equipment and with programming systems that are designed to aid the programmer in the preparation and operation of his programs. These total facilities for receiving information and producing desired results are called a data processing system. One part of such a system may be FORTRAN, which is a programming system that enables a programmer to write a program with less effort than would otherwise be required. The FORTRAN program is written in a relatively simple language — one which closely resembles the ordinary language of mathematics.

THE FORTRAN SYSTEM

FORTRAN has two parts: the language and the processor. The language is composed of the individual commands or statements of a program, operators (such as + or -), and expressions (such as A+B-C). The processor is a program for the computer which tells it how to translate the program written in the FORTRAN language into a program written in machine language.

The Source Program

The program which defines the operations which the computer is to do, and which is written by the programmer in the FORTRAN language, is called the source program.

The Object Program

The source program is then input to the computer along with the FORTRAN processor. The computer, following instructions from the processor, converts the source program into a program in machine language, ready to be run on the computer. This machine language program is called the object program. When the object program is run on the computer to cause the desired computations, it is said to be executed. That is, execution is the actual operation of the computer while the control unit is under the direction of the object program.

Source and Object Machine

Although the FORTRAN language is largely independent of the computer on which the object program is to be executed (the object machine), the FORTRAN processor is dependent on the object machine because it must produce an object program in that machine's language. For this reason, each object machine must have its own processor. (In this manual, reference to a single object machine, a single data processing system, etc., refers to all items in that category; thus, a single object machine might be all IBM 704 computers.)

Since each object machine has its own processor, and each processor is dependent upon that object machine, there is some variation in the FORTRAN language statements which may be input to that processor. The variations in source programs that are required by the variations in the processors programs may be prepared which are suitable for input to several processors; however, a separate object program must be produced for each object machine (i. e., the source program must be input to each applicable processor) on which the object program is to be executed.

Statements

Each FORTRAN source program is composed of a number of statements. Each statement deals with one aspect of the problem; it may cause data to be fed into the computer, calculations to be performed, decisions to be made, results to be printed, etc.

Some statements written by the programmer do not cause specific computer action, but rather provide information to the processor.

EXAMPLES OF FORTRAN STATEMENTS

Some examples of FORTRAN statements and their effects are:

READ 1, A	This causes the computer to read an IBM card and handle the data on it in such a way that if the card read has the number 97. 0 on it, then A will have the value 97. 0.
C = 3. *A	The asterisk (*) indicates multiplication. Thus, this statement means multiply A by 3. 0 and set C equal to the result. Using the data of the previous example, C would be given the value 291. 0 (3. 0 x 97. 0).

Notice that the computer is not instructed merely to find the value of C. It is given the data (in this case 97. 0), instructed how to make the computation, and told what to do with the result. Similarly, to find the roots of a quadratic equation, the computer must be told how to find the roots. Since there are two roots for each quadratic equation, the computer must be instructed how to find each root separately. Thus, using the formula:

$$\text{ROOT} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

The computer may be instructed as follows:

```
ROOT1 = (-B+SQRTF (B**2-4. *A*C))/(2. *A)

ROOT2 = (-B-SQRTF (B**2-4. *A*C))/(2. *A)
```

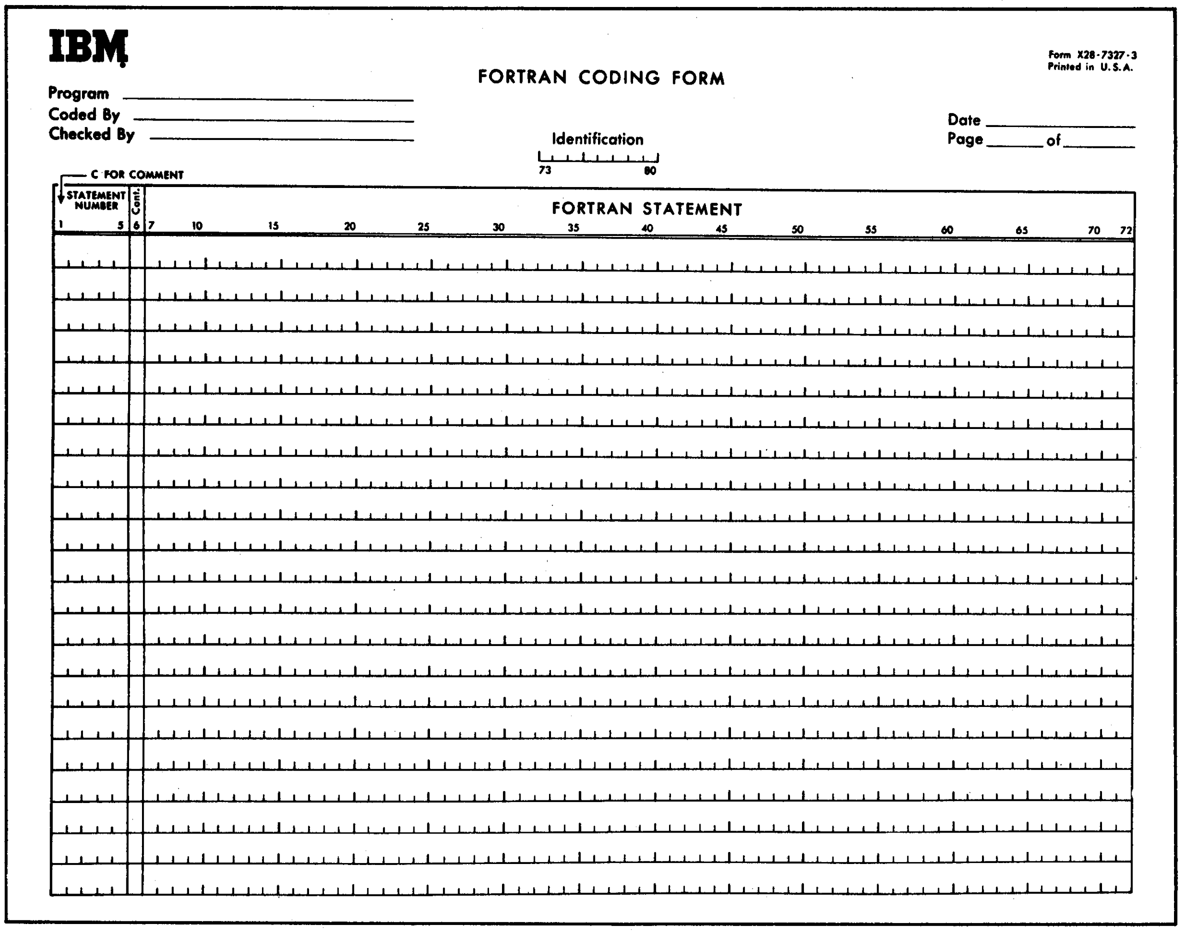
The rules for formulating these statements will be discussed later in this manual.

Writing FORTRAN Programs

FORTRAN cannot distinguish between "UPPER CASE" and "lower case" letters. Thus, UPPER CASE letters will be used in this manual to indicate actual coding; lower case letters will be used for symbolic representation only.

The FORTRAN Coding Form

The standard FORTRAN coding sheet is shown below.



IBM

FORTRAN CODING FORM

Form X28-7327-3
Printed in U.S.A.

Program ______
Coded By ______
Checked By ______

Identification
73 80

Date ______
Page ____ of ____

C FOR COMMENT

STATEMENT NUMBER | Cont. | FORTRAN STATEMENT

1 5 6 7 10 15 20 25 30 35 40 45 50 55 60 65 70 72

FORTRAN statements are written one to a line in columns 7-72. If a statement is too long for one line, some processors permit it to be continued on one or more successive lines by placing a character other than zero in column 6. (If desired, the characters in column 6 may be used to indicate the order of continuation lines.) Otherwise, column 6 — for initial lines of a statement — must be blank or zero.

Except in column 6 and in alphameric fields of FORMAT statements (see page 47), blanks are ignored by FORTRAN and may be used to improve the readability of a FORTRAN program.

Thus

A=B(I, J)-D-(C/E)-F**K

and

A = B(I, J) - D - (C/E) - F**K

are equivalent.

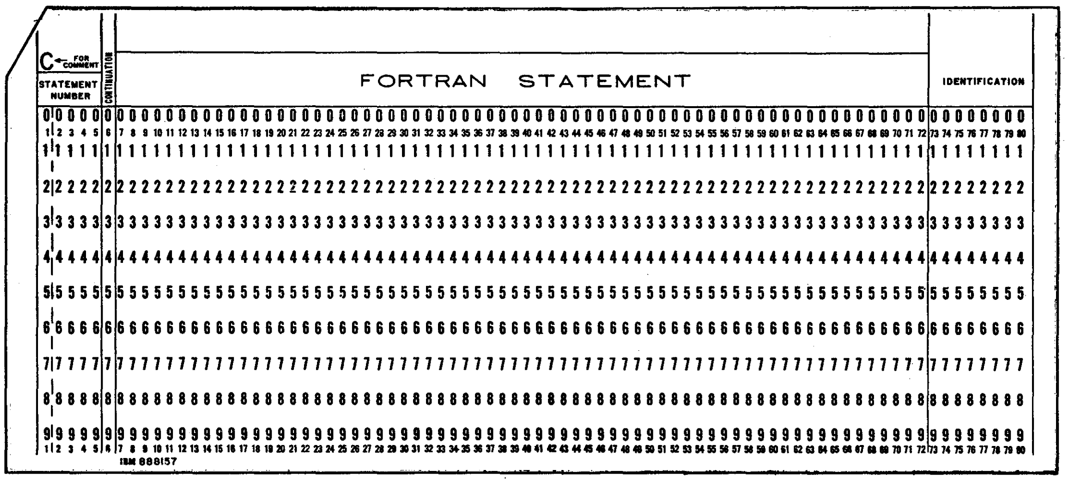
Columns 1-5 may be used to write numbers by which the statement may subsequently be referenced. The magnitude of these numbers is determined by the FORTRAN processor being used. These statement numbers may be assigned in any order; the sequence of operations is dependent only on the order of the statements, not on their statement numbers.

Columns 73-80 may be used for any desired identifying information.

Comments to explain the program may be written in columns 2-72 of a line with a C in column 1. These comments are not processed by FORTRAN but are printed on the program listings produced when the source program is translated into the object program.

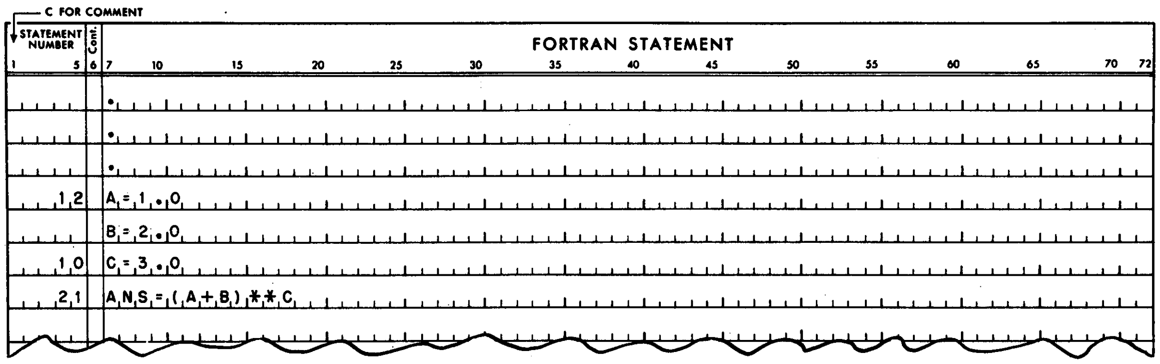
The FORTRAN Card

Each line of the coding sheet is then used to prepare a punched card. A standard FORTRAN card is shown below.



The information in column 1 of a line on the coding sheet is punched into card column 1, column 2 into card column 2, and so forth. After the cards are punched, they should be verified to prevent clerical errors from causing source and object program errors.

An example of a partial FORTRAN program is shown below as it would appear on a FORTRAN coding sheet:


C FOR COMMENT

STATEMENT NUMBER | Cont. | FORTRAN STATEMENT

```
      .
      .
      .
   12 A=1.0
      B=2.0
   10 C=3.0
   21 ANS=(A+B)**C
```

As a result of this sequence, ANS would be assigned the value 27. 0.

Elements of the Language

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

1. Constants, such as 27 or 3. 14159.
2. Variables, such as X or Y.
3. Subscripted variables, such as x_i or y_i, which are written in FORTRAN as X(I) or Y(I).
4. Arithmetic statements, which cause mathematical computations, such as $a=\frac{b}{c}$, which is written in FORTRAN as A=B/C.
5. Mathematical expressions, which are meaningful to FORTRAN, such as X+Y or 3*J.
6. Control statements which tell the computer what it is to do.
7. Instructions to the FORTRAN processor to assist it in producing the object program.
8. Input/output statements which are used for getting data into the computer and producing results.
9. Subroutine statements, which permit programs to be incorporated within larger programs, allowing the programmer to cause a computation without specifying each instruction every time the computation is to occur.

Each of these topics will be dealt with subsequently in this manual.

PROBLEMS

At various places throughout this manual, problems will be given. These problems may be used by the reader to review what has been discussed.

Answers to problems are given in Appendix C.

No specific problems will be stated at this point; however, the reader should thoroughly understand the following terms:

1. Source program

2. Object program

3. Source machine

4. Object machine

5. Execution of object program

6. Processor

CHAPTER 2: CONSTANTS, VARIABLES, AND SUBSCRIPTS

FORTRAN provides a means of expressing numerical constants, variable quantities, and subscripted variables. The rules for expressing these quantities are quite similar to the rules of ordinary mathematical notation. However, each of these quantities may be expressed in one of two basic modes.

INTEGER AND FLOATING POINT CALCULATIONS

Mode is important because FORTRAN permits two types of arithmetic calculations: integer (often called fixed point) and floating point.

Floating point calculations are carried out between two decimal numbers to an accuracy of several decimal digits. The exact accuracy depends on the FORTRAN system being used. Some typical floating point calculations are:

Arithmetic Statement	Result of Calculation
A=.4301/1.7	A=.253
B=5./2.	B=2.5
C=1.6*.7	C=1.12
D=-2.7+1.2	D=-1.5

Thus, floating point calculations are "conventional"; they are the familiar type. Rounding does not occur; excessive digits are simply dropped (truncated).

Integer calculations are carried out differently; calculations are with integers only, no decimal remainders are retained or used in computations. For example:

Arithmetic Statement	Result of Calculation
I=5/2	I=2 (instead of 2.5 since the .5 is truncated)
I=5/2+7/2	I=5 (intermediate truncation causes this to be computed as 2+3 rather than 12/2)
J=5*2	J=10
K=-4+1	K=-3

CONSTANTS

A constant is any number which is used in computations without change from one execution of the program to the next. It appears in its actual numerical form in the source statement. For example, in the statement

```
J=3*K
```

3 is a constant, since it appears in actual numerical form.

Two types of constants may be written in FORTRAN: integer constants and floating point constants (characterized by being written with a decimal point). The rules for writing each of these constants are given below.

Integer Constants

General Form of an Integer Constant
An integer is written without a decimal point, using the decimal digits 0, 1, ..., 9. A preceding + or - sign is optional. An unsigned constant is assumed to be positive. The magnitude of an integer constant must not exceed specific limits set for each FORTRAN processor. An integer constant of 1 to 4 decimal digits is acceptable to all processors.

Examples:

The following are valid integer constants:

```
   0
  +9
 186
-327
   6
  45
```

On the other hand, the following are <u>not</u> valid integer constants:

-3.2	(contains a decimal point)
27.	(contains a decimal point)
28913	(as an example, assume that this number exceeds the magnitude permitted by the processor)

Floating Point Constants

General Form of a Floating Point Constant
Any number written with a decimal point, using the decimal digits 0, 1, ..., 9. A preceding + or - sign is optional. An unsigned constant is assumed to be positive.   An integer exponent preceded by an E may follow a floating point constant. The decimal exponent may have a preceding + or - sign. An unsigned exponent is assumed to be positive.   The magnitude of a floating point constant must not exceed specific limits set for each FORTRAN processor. Also, for some processors there is a maximum number of digits permitted. A floating point constant of 1 to 8 decimal digits, with magnitude zero or between 10^{-38} and 10^{38} is acceptable to all processors.

Thus, a floating point constant may be an integer written with a decimal point, a decimal fraction, or a mixed integer and decimal fraction.

Examples:

The following are valid floating point constants:

1.	
.2	
3.4	
.0097	
6.0	
5.0E3	(means 5.0×10^3, i.e., 5000)
5.0E+3	(means 5.0×10^3, i.e., 5000)
5.0E-3	(means 5.0×10^{-3}, i.e., .005)

The following are not valid floating point constants:

4367	(no decimal point)
5.0E121	(as an example, assume that this exceeds the magnitude permitted by the processor)
234.48397639	(as an example, assume that this exceeds the number of digits permitted by the processor)

VARIABLES

A FORTRAN variable is a symbolic representation which will assume a value. This value may change either for different executions of the program or at different stages within the program. For example, in the statement

```
K = 3*I
```

both I and K are variables. The value of I will be assigned by a preceding statement and may change from time to time, and K will vary whenever this computation is performed with a new value of I.

As with constants, a variable may be integer or floating point, depending on whether the value which it will represent is to be integer or floating point, respectively.

In order to distinguish between variables which will derive their value from an integer as opposed to those which will derive their value from a floating point number, the rules for naming each type of variable are different.

Integer Variables

General Form of an Integer Variable
A series of alphameric characters (except special characters), of which the first is I, J, K, L, M, or N. The length of the series (that is, the number of characters) must not exceed specific limits set for each FORTRAN processor. An integer variable of 1 to 5 characters is acceptable to all processors.

Alphameric characters are all of the alphabetic and numerical characters, A to Z and 0 to 9. Alphameric characters also include the following special characters (which may not be used in variable names):

.	,	+	$	*
-	/	)	(	=

Examples:

The following are valid integer variables:

I
JOB1
JOB2
M3
NEXT
MAX

The following are <u>not</u> valid integer variables:

PMAX	(first character is not an integer indicator)
8ILO	(first character is not alphabetic)
IMIN$	($ not permitted – it is a special character)
IABCDEFG	(as an example, assume that this exceeds the number of characters permitted by the processor)

WARNING:

1. A variable must not be given a name which coincides with the name of a function without its terminal F. Thus, if a function is named TIMEF, no variable should be named TIME.

● 2. Unless their names are less than four characters in length, subscripted variables should not be given names ending with F, because FORTRAN may consider variables so named to be functions.

For a discussion of the meaning of "function," see page 53.

An integer variable may assume any value expressible as an integer constant in the FORTRAN processor being used. For example, I is an integer variable and 3 is an integer constant. Thus, I may be assigned the value 3. I may not be assigned the value 3.23, since that is a floating point constant.

Floating Point Variables

General Form of a Floating Point Variable
A series of alphameric characters (except special characters), of which the first is alphabetic but not one of the integer indicators (I, J, K, L, M, or N). The length of the series (that is, the number of characters) must not exceed specific limits set for each FORTRAN processor. A floating point variable of 1 to 5 characters is acceptable to all processors.

Examples:

The following are valid floating point variables:

A
B7
B22
DELTA
COST
VALUE
Z9X8Y

● The following are not valid floating point variables:

ITRNS	(first character is an integer indicator)
8BETA	(first character is not alphabetic)
ACST/	(/ not permitted - it is a special character)
ADELTAMAX	(as an example, assume that this exceeds the number of characters permitted by the processor)

The same warning concerning integer variable names applies to floating point variable names.

A floating point variable may assume any value expressible as a floating point constant in the FORTRAN processor being used.

Considerations in Naming Variables

The rules for naming variables allow for extensive selectivity. In general, it is easier to follow the flow of a program if meaningful symbols are used wherever possible. For example, to compute distance it would be possible to use the statement

```
X=Y*Z
```

but it would be more meaningful to write

```
D=R*T
```

or even

```
DIST=RATE*TIME
```

Similarly, if the computation were to be performed using integers, it would be possible to write

```
I=J*K
```

or

```
ID=IR*IT
```

or

```
IDIST=IRATE*ITIME
```

In other words, variables can often be written in a meaningful manner by using an initial character to indicate whether the variable is integer or floating point and using succeeding characters as an aid to memory.

Another aid to writing a program is to vary the last character of a variable name. For example, to compute four different quantities called HRS, the following could be used:

```
HRS1
HRS2
HRS3
HRS4
```

Again, each of these could be preceded by I, J, K, L, M, or N to indicate integer mode.

SUBSCRIPTS

An array is a group of quantities. It is often advantageous to be able to refer to this group by one name and to refer to each individual quantity in this group in terms of its place in the group.

For example, assume the following is an array named NEXT:

15
12
18
42
19

Suppose it is desired to refer to the second quantity in the group; in ordinary mathematical notation this would be $NEXT_2$. In FORTRAN this would be

NEXT(2)

The quantity "2" is called a subscript. Thus

NEXT(2) has the value 12
NEXT(4) has the value 42

Similarly, ordinary mathematical notation might use $NEXT_i$ to represent any element of the set NEXT. In FORTRAN, this might be written as NEXT(I) where I equals 1, 2, 3, 4, or 5.

The array could be two dimensional; for example, the array MAX:

	Column 1	Column 2	Column 3
Row 1	82	4	7
Row 2	12	13	14
Row 3	91	1	31
Row 4	24	16	10
Row 5	2	8	2

Suppose it is desired to refer to the number in row 2, column 3; this would be

MAX(2, 3)

"2" and "3" are the subscripts. Thus

MAX(2, 3) has the value 14
MAX(4, 1) has the value 24

Similarly, ordinary mathematical notations might use $MAX_{i,j}$ to represent any element of the set MAX. In FORTRAN, this might be written as MAX(I, J) where I equals 1, 2, 3, 4, or 5 and J equals 1, 2, or 3.

In some systems, the above notation may be extended to three dimensional arrays.

Form of Subscripts

General Form of a Subscript

A subscript must be in one of the following forms only, where v represents any unsigned non-subscripted integer variable, and c and c' any unsigned integer constant:

v
c
v+c or v-c
c*v
c*v+c' or c*v-c'

1620 FORTRAN has a more limited subscript format (see page 61).

Notice that a floating point quantity may not appear in a subscript and that constants may not be signed. Notice also that the rules for forming the quantities must be rigidly followed.

Examples:

The following are valid subscripts:

IMAX
19
JOB+2
NEXT-3
8*IQUAN
5*L+7
4*M-3

The following are not valid subscripts:

-I	(the variable may not be signed)
A+2	(A is not an integer variable)
I+2.	(2. is not an integer constant)
-2*J	(the constant must be unsigned)
I(3)	(a subscript may not be subscripted)
K*2	(for multiplication, the constant must precede the variable, thus 2*K is correct)
2+JOB	(for addition, the variable must precede the constant, JOB+2 is correct)

The value of the subscript is computed, and the quantity referred to is then located. The location of a specific quantity is dependent upon the arrangement of the array in storage. This is discussed elsewhere in this manual.

Subscripted Variables

General Form of Subscripted Variables
An integer or floating point variable, followed by parentheses enclosing one, two or three subscripts which are separated by commas. (Some processors do not permit three subscripts.)

Examples:

The following are valid subscripted variables:

```
A(I)
K(3)
BETA(5*J-2,K+2,L)
MAX(J,K,2)
```

The following are not valid subscripted variables:

A(1,)	(a comma is not allowed after the last subscript)
IMAX(A)	(a floating point variable cannot appear in a subscript)
NEXT(I(3))	(a subscript may not be subscripted)

Each variable which is subscripted must have the size of its array specified preceding the first appearance of the variable in subscripted form. This is done by a DIMENSION statement (see page 33).

PROBLEMS

For each of the following, determine the type of quantity and whether it is valid or invalid. If invalid, why?

1. MAX$8
2. QUANT(3*B)
3. -3.78.
4. IOVR7
5. AII
6. 427
7. A(+M)
8. .71E-8.
9. ABLE(3*N-5)
10. I71
11. RATE(3+I,2-J)
12. +17.2
13. JOBNO(I,3,L)
14. MIN(A)
15. AAA
16. 6ABLE
17. 5E+9
18. I(-K)
19. 12.E15
20. JACK(-3*K)

PART II: THE BASIC FORTRAN LANGUAGE

CHAPTER 3: ARITHMETIC STATEMENTS AND EXPRESSIONS

ARITHMETIC STATEMENTS

The arithmetic statement defines a numerical calculation; it very closely resembles a conventional arithmetic formula.

General Form of an Arithmetic Statement
"a = b" where a is a variable (subscripted or not subscripted) and b is an expression as defined below.

Examples:

The following are valid arithmetic statements:

```
A = B+C
D(I) = E(I) +2. -F
```

In a FORTRAN arithmetic statement, the equal sign means "is to be replaced by" rather than "is equivalent to." This distinction is important; for example, suppose an integer variable I has the value 3. Then, the statement

```
I = I + 1
```

would give I the value 4. This feature enables the programmer to keep counts and perform other required operations in the solution of a problem.

The following is an example of a series of FORTRAN arithmetic statements;

A = 3.0	Store the value 3.0 in A
B = 2.0	Store the value 2.0 in B
C = A + B	Add the values in A and B and store in C (3. + 2. = 5.)
C = C + 1.	Add 1. to the value in C (5. + 1. = 6.)

EXPRESSIONS

An expression in FORTRAN is a sequence of constants, variables (subscripted or not subscripted) and operation symbols which indicates a quantity or a series of calculations. It must be formed according to the rules for constructing expressions. It may include commas and parentheses and may also include functions (which will be discussed later). It appears on the right-hand side of arithmetic statements and in certain types of control statements.

Operation Symbols

The operation symbols are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Rules for Constructing Expressions

Since constants, variables, and subscripted variables may be integer or floating point quantities, expressions may contain either integer or floating point quantities; however, the two types may appear in the same expression only in certain ways. (In the following discussion, no mention is made of the rules for using integer and floating point quantities in functions. These rules will be stated when functions are discussed and will be considered as addenda to the following rules.)

1. The simplest expression consists of a single constant, variable or subscripted variable. If the quantity is an integer quantity, the expression is said to be in the integer mode. If the quantity is a floating point quantity, the expression is said to be in the floating point mode.

   Examples:

Expression	Type of Quantity	Mode of Expression
3	Integer constant	Integer
3.0	Floating point constant	Floating point
I	Integer variable	Integer
A	Floating point variable	Floating point
I (J)	Integer subscripted variable	Integer
A (J)	Floating point subscripted variable	Floating point

   In the last example, note that the subscript, which must be an integer quantity, does not affect the mode of the expression. The mode of the expression is determined solely by the mode of the quantity itself.

2. Exponentiation of a quantity does not affect the mode of the quantity; however, an integer quantity may not be given a floating point exponent. The following are valid:

I**J	Integer
A**I	Floating point
A**B	Floating point

   The following is not valid:

I**A	(Violates the rule that an integer quantity must not have a floating point exponent)

NOTE: The expression A**B**C is not permitted. It must be written A**(B**C) or (A**B)**C, whichever is intended.

3. Quantities may be preceded by a + or a - or connected by any of the operators (+, -, *, /, **) to form expressions, provided:

   a. No two operators appear consecutively.

   b. Quantities so connected are all of the same mode. (Exception: floating point quantities may have fixed point exponents; see 2 above.)

   c. No operators are "assumed" to be present.

   The following are valid:

   ```
   -A+B
   B+C-D
   I/J
   K*L
   ```

   The following are not valid expressions:

A+-B	(must be written as A+(-B))
A+I	(variables are of different modes)
3J	(must be written as 3*J if multiplication is intended)

4. The use of parentheses in forming expressions does not affect the mode of the expression. Thus, A, (A), and (((A))) are all floating point expressions.

5. Parentheses may be used to specify the order of operations in an expression. Where parentheses are omitted, the order is taken to be from left to right as follows:

**	Exponentiation
* and /	Multiplication and Division
+ and -	Addition and Subtraction

   For example, the expression

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

   will be taken to mean

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

   Using parentheses, the expression could be written

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

   which would be taken to mean

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

A valid expression will be evaluated when the object program is executed. An invalid expression may result in an error message from the FORTRAN processor or may result in inaccurate object program results.

MODE OF AN ARITHMETIC STATEMENT

Although expressions must be integer or floating point, the variable on the left-hand side of the equal sign in an arithmetic statement need not be of the same mode as the expression on the right-hand side.

If the variable on the left is an integer quantity and the expression on the right is floating point, the expression will first be evaluated in floating point, the portion following the decimal point will be dropped, and the remainder will be converted to an integer quantity. Thus, if the result is +3. 872, the integer stored will be +3, not +4. If the variable on the left is floating point and the expression on the right is integer, the latter will be evaluated as an integer expression, and the result will be converted to floating point.

Examples:

Arithmetic Statement	Result of Calculation
A=3/2	A=1.
A=3./2.	A=1.5
I=3/2	I=1
I=3./2.	I=1
I=3./2	Not allowed. "3." and "2" are not the same mode.

PROBLEMS

1. May the variable on the left side of an equal sign in an arithmetic statement be subscripted?

2. How is the mode of an expression determined?

3. When may an integer quantity appear in a floating point expression?

4. What notation should be used to indicate the order of computations in an expression?

5. Evaluate the following arithmetic statements where A=3., B=2., C=1., I=3, J=2, K=1:
   a. L=I/J
   b. M=J*K
   c. N=K-I
   d. D=C*A
   e. E=B/C
   f. F=C-A
   g. A=I/J
   h. M=A/B
   i. I=2*I
   j. C=C+1.

6. Which of the following are valid arithmetic statements?
   a. A(I, 3)=M2+J
   b. X(I+2, J)=-3. *D+(E-F)
   c. Y=I**A
   d. A(B)=I+2
   e. A+3. =B*C
   f. M=(A+B)
   g. Z=A**R
   h. I(J)=K(J)/J
   i. I(A)=H+14.
   j. W=I+(-D)

CHAPTER 4: CONTROL STATEMENTS AND THE SPECIFICATION STATEMENT

Normally, FORTRAN statements may be thought of as being executed sequentially. That is, after one statement has been executed, the statement immediately following it will be executed. However, it is often undesirable to proceed with each statement in this manner. This chapter will discuss some of the statements which may be used to alter sequential execution and some of the reasons why this may be desirable. This chapter will also discuss use of the DIMENSION statement, which provides information that the processor requires for the establishment of arrays.

UNCONDITIONAL GO TO

This statement is used to interrupt sequential execution; it indicates the statement that is to be executed next.

General Form of the Unconditional GO TO Statement
"GO TO n" where n is a statement number.

This statement causes statement number n to be executed next.

Examples:

```
GO TO 16
GO TO 137
```

A coding example is shown below:

```
        .
        .
        .
        A=3.
        B=4.
        GO TO 7
12      B=2.*A
 7      A=2.*B
        .
        .
        .
```

Statement 12 will not be executed. Statement 7 will be evaluated and A will be assigned the value 8.0.

COMPUTED GO TO

This statement also indicates the statement that is to be executed next. However, it allows the next statement to be executed to be different at various stages in the program.

General Form of the Computed GO TO Statement
"GO TO ($n_1, n_2, \ldots, n_m$), i" where $n_1, n_2, \ldots, n_m$ are statement numbers and i is a non-subscripted integer variable.   The parentheses enclosing the statement numbers, the commas separating the statement numbers, and the comma following the right parenthesis are all required punctuation.

This command causes transfer of control to the 1st, 2nd, 3rd, etc., statement in the list depending on whether the value of i is 1, 2, 3,..., etc. i must never have a value greater than the number of items in the list. The value which i has at any given time must be set by a preceding arithmetic statement.

Examples:

GO TO (5,7,8,2,4),J	If J is 3, transfer to statement 8.
GO TO (4,4,4,7,8,9),MAX	This example illustrates the fact that several values of i may cause a transfer to the same statement. In this case, when MAX has the values 1,2, or 3, transfer will be made to statement number 4.

Further use of the Computed GO TO is illustrated below:

```
        .
        .
        .
        A = 3.
        B = 4.
        C = 5.
        K = 0
   1    K = K+1
        GO TO (10,20,30),K
        .
        .
        .
  30    F = A-B
        GO TO 12
  20    E = A-C
        GO TO 1
  10    D = B-C
        GO TO 1
        .
        .
        .
  12
```

As a study of this example will show, D, E and F are computed, in that order, and control proceeds to statement 12. Of course, the example itself is highly simplified; if these were the only required calculations in this series, the programmer would just compute D, E, and F sequentially, in any desired order and without using the Computed GO TO.

IF

This statement permits a programmer to change the sequence of statement execution, depending upon the value of an arithmetic expression. This is called a logical decision.

General Form of the IF Statement
"IF(a)n_1, n_2, n_3" where a is an expression and n_1, n_2 and n_3 are statement numbers. The expression, a, must be enclosed in parentheses; the statement numbers must be separated from one another by commas.

Examples:

```
IF(A-B)10,5,7
IF(A(I)/D) 1,2,3
```

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

Suppose a value, A, is being computed. Whenever this value is positive, it is desired to proceed with the program. Whenever the value of A is negative, an alternative routine starting at statement 12 is to be followed, and if A is zero, an error routine at statement 72 is to be followed. This may be coded as:

```
        .
        .
        .
        A = (B+C)/(D**E)-F
        IF (A) 12, 72, 10
   10   .
        .
        .
   12   .
        .
   72   .
```

LOOPING

As discussed earlier, the ability of a computer to repeat the same operations with different data, called looping, is a powerful tool which greatly reduces programming effort. There are several ways to accomplish this looping; one way is to use an IF statement. For example, assume that a plant carries 1,000 parts in inventory. Periodically it is necessary to compute stock on hand of each item (INV), by subtracting stock withdrawals of that item (IOUT) from previous stock on hand. It would be wasteful of effort to write a program which would indicate each separate subtraction by a separate statement. (It would also be wasteful of computer storage, since each separate instruction to the computer must be in computer storage.) The same results could be achieved by the following program:

```
          .
          .
          .
 5        J = 0
10        J = J+1
25        INV(J) = INV(J) - IOUT(J)
15        IF (1000-J) 20,20,10
20        .
          .
          .
```

An index, J, is established which will be increased by 1 each time statement 10 is executed. Statement 5 initializes J to zero so that statement 10 will set J equal to 1 for the first execution of statement 25.

Statement 25 will compute the current stock on hand by subtracting the stock withdrawal from the previous stock on hand. The first time statement 25 is executed, the stock on hand of the first item in inventory INV(1), will be computed by subtracting the stock withdrawal of that item, IOUT(1). Statement 15 tests whether all items in stock have been updated. If not, the expression 1000-J will be positive and the program will transfer to statement 10, which will increment J by 1. Statement 25 will be executed again, this time for the stock on hand of item 2, INV(2), and the stock withdrawal of item 2, IOUT(2). This procedure will be repeated until the stock of item 1000 has been updated. At this point, J will be equal to 1000, and the expression in statement 15 will be equal to zero. Then statement 15 will cause transfer to statement 20 to continue with other parts of the program.

Notice that three statements (5, 10 and 15) were required for this looping; this could have been accomplished with a single DO statement.

Not only does the DO simplify the programming of loops, it also provides greater flexibility in looping.

General Form of the DO Statement
"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 non-subscripted integer variable, and m_1, m_2, and m_3 are each either an unsigned integer constant or non-subscripted integer variable. If m_3 is not stated, it is taken to be 1. The commas separating the statement numbers are required.

Examples:

```
DO 20 JBNO=1,10
DO 20 JBNO=1,10,2
DO 20 JBNO=K,L,3
```

The DO statement is a command to repeatedly execute the statements which follow, up to and including the statement with statement number n. The first time, the statements are executed with i = m_1. For each succeeding execution of the statements, i is increased by m_3. After the statements have been executed with i equal to the highest of this sequence which does not exceed m_2, control passes to the statement following the last statement in the range of the DO (the statement after statement number n).

Thus, the DO statement does three things:

1. It establishes an index which may be used as a subscript or in computations.
2. It causes looping through any desired series of statements, as many times as required.
3. It increases the index (by any amount that the programmer specifies) for each separate execution of the series of statements in the loop.

   Example:

```
        .
        .
        .
15      DO   25  J = 1,1000
25      INV (J) = INV(J) - IOUT(J)
35      .
        .
        .
```

Statement 15 is a command to execute the following statements up to and including statement 25; the first time J will be 1, thereafter J will be increased by 1 for each execution of the loop until the loop has been executed with J equal to 1000. After the loop has been executed with J equal to 1000, the statement following statement 25 will be executed.

The following is a comparison of statement 15 with the general form of the DO, and an introduction of some of the terms used in discussing DO statements:

$$\begin{array}{llcllll} \text{DO} & n & i & = & m_1, & m_2, & m_3 \\ \text{DO} & \underbrace{25}_{\text{Range}} & \underbrace{J}_{\text{Index}} & = & \underbrace{1,}_{\substack{\text{Initial}\\ \text{Value}}} & \underbrace{1000}_{\substack{\text{Test}\\ \text{Value}}} & \underbrace{}_{\text{Increment}} \end{array}$$

The <u>range</u> is the series of statements to be executed repeatedly. It consists of all statements following the DO, up to and including statement n. In this case, statement n is statement 25, and the range consists of only one statement. The range can consist of any number of statements.

The <u>index</u> is the integer variable which will change for each execution of the range. In the example, this index was used as a subscript, in another problem it might be used in computations, etc. (The index need not be used in the range, although it usually is.)

The <u>initial value</u> is the value of the index for the first execution of the range. Although the initial value was 1 for this example, in another problem it might be some different integer quantity. Often, the initial value will change at different times within the program. In such cases it may be stated as a non-subscripted integer variable. The variable must then be assigned a value before the DO is executed.

The <u>increment</u> is the amount by which the value of the index will be increased after each execution of the range. In the example, this is not coded because the increment desired is 1, and the general form permits omission of the increment when it is 1. As with the initial value, the increment may be written as an integer variable.

The <u>test value</u> is the value which the index may not exceed. After the range has been executed with the highest value of the index which does not exceed the test value, the DO is satisfied, and the program continues with the first statement following the range. In the example, the DO was satisfied after the range was executed with the index equal to the test value. In some cases, the DO is satisfied before the test value is reached. Consider, for example, the following DO:

```
        DO 5 K = 1, 9, 3
        .
        .
    5   .
```

In this example, the range will be executed with K equal to 1, 4 and 7. The next value of K would be 10; since this exceeds the test value, control passes to the statement following statement 5 after the range is executed with K equal to 7. The test value may also be written as an integer variable.

As a further example, consider the following program:

```
          .
          .
          .
          K = 0
          L = 10
          DO  5  JOB = 1, L, 2
          K = K+1
    5     A(JOB) = B(JOB)-K*JOB
```

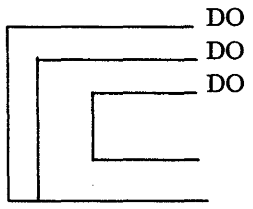
This would cause the following computations:

```
A(1) = B(1)-1*1
A(3) = B(3)-2*3
A(5) = B(5)-3*5
A(7) = B(7)-4*7
A(9) = B(9)-5*9
```

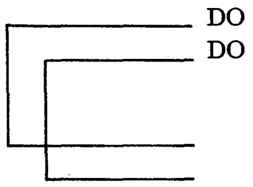
When using DO statements, the following rules must be followed:

1. Within the range of a DO may be other DOs. When this is so, all statements in the range of the latter must be in the range of the former. A set of DOs satisfying this rule is called a nest of DOs.

   For example:



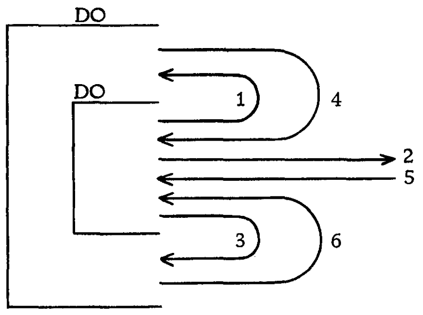
   is a permitted configuration (brackets are used to indicate the range of the DOs), but



   is not a permitted configuration.

2. Transfer of control from inside the range of a DO to outside its range is permitted at any time. However, the reverse is not allowed. A transfer is not 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.



3. When control leaves the range of a DO in the ordinary way (that is, when the DO becomes satisfied and control passes on to the next statement after the range), the exit is said to be a normal exit. In some systems, after a normal exit from a DO occurs, the value of the index controlled by that DO is not defined, and the index cannot be used again until it is redefined. That is, if J were the index and the DO were satisfied when J equalled 1000, J may not still have the value 1000 after the DO is satisfied. However, if exit occurs by a transfer out of the range, the current value of the index remains available for subsequent use. If exit occurs by a transfer which is in the range of several DOs, the current values of all the indexes controlled by those DOs are preserved for subsequent use.

4. No statement is permitted in the range of a DO which redefines the value of the index or of any of the indexing parameters (m's). That is, none of these quantities may appear on the left-hand side of the equal sign in an arithmetic statement (nor in the list of an input statement) within the range of a DO.

5. The first statement in the range of a DO must not be one of the non-executable FORTRAN statements (DIMENSION, FORMAT, etc.).

6. The range of a DO cannot end with a transfer (IF or GO TO type statements).

CONTINUE

This statement is used as the last statement in the range of a DO where the last statement would otherwise be a transfer-type command (which is not permitted by item 6, above).

General Form of a CONTINUE Statement
"CONTINUE"

As an example of a program which requires a CONTINUE, consider the following program:

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

This program will scan the 100-entry VALUE array until it finds an entry which equals the value of the variable ARG, whereupon it will transfer to statement 20 with the value of I available for use. If no entry in the array equals the value of ARG, a normal exit to the statement following the CONTINUE will occur.

PAUSE

This statement will cause the computer to come to a halt during object program execution. Depressing the Start key causes the computer to resume execution of the object program with the next FORTRAN statement.

General Form of the PAUSE Statement
"PAUSE"

STOP

This statement causes the computer to halt during object program execution in such a way that depressing the Start key has no effect. Therefore, in contrast to PAUSE, this statement is used where a final, rather than a temporary, stop is desired.

General Form of the STOP Statement
"STOP"

Some systems rely on supervisory programs (monitors) to control execution of the object program. In a system of this type, the PAUSE and STOP statements may not be desirable and should not be used.

THE SPECIFICATION STATEMENT

The following statement is considered to be non-executable because it does not give rise to any instructions in the object program. Instead, the statement provides the processor with information required to allocate locations in computer memory (where the various elements of an array are to be stored).

DIMENSION

General Form of the DIMENSION Statement
"DIMENSION $v_1, v_2, \ldots, v_n$" where each v is the name of a variable subscripted with 1, 2 or 3 unsigned integer constants.   The v's must be separated from each other by commas.

Examples:

```
DIMENSION A(10), B(5,15), CVAL(3,4,5)
DIMENSION I(100)
DIMENSION NEXT(10,10,10)
```

Each variable which appears in subscripted form must appear in a DIMENSION statement; the DIMENSION statement must preceed the first appearance of that variable. The DIMENSION statement lists the maximum dimensions of arrays; object program references to these arrays must never exceed the specified dimensions.

• The statement

```
DIMENSION JOB (10,15,5)
```

means that JOB is a three-dimensional array for which the subscripts never exceed 10, 15 and 5. The DIMENSION statement, therefore, causes 750 (that is, 10x15x5) storage locations to be set aside for the array JOB. Some processors do not permit three-dimensional arrays. A single DIMENSION statement may specify the dimensions of any number of arrays, and may include both integer and floating point arrays.

PROBLEMS

1. Which of the following are valid FORTRAN statements?

   a. DO 10 FEW = 1,10,4
   b. DO 51 K = 1,K-1,3
   c. GO TO M
   d. GO TO (1,2,3),A
   e. GO TO 187
   f. IF (A-I) 1,2,2
   g. GO TO 3,4
   h. DO 1,J = 1,50,1
   i. GO TO (4,7,3)K
   j. GO TO N-12

2. With the exception of input/output statements, write complete FORTRAN programs for the following problems. Assume all necessary data to be available to the program. Be sure to use DIMENSION statements and halt statements where appropriate.

a. In the integer mode, calculate the distance between ten sets of cities when the rate and time are known. Use the formula Distance = Rate x Time.

b. Rewrite the preceding program so that if the distance is over 1,000 miles, transfer will be made to statement 5; otherwise the program is to proceed to the next statement.

c. For an unknown quantity, X, find the value of X+(X-1)+(X-2)...+1. Assume that X is an integer greater than zero. Use the floating point mode.

d. Assume that for any employee, E, net pay is equal to base pay plus overtime pay minus income tax minus other deductions. Write a program to compute the net pay of I number of employees, where I is never greater than 500. Use floating point mode to compute pay.

e. Generate a two-dimensional array, A, such that the first column of A will consist of the integers 1 to 25 and each item in the second column will be 1 1/2 times the corresponding item in the first column. Use the floating point mode.

f. Compute the reciprocals of X(I) where I = 1,..., 20. The reciprocal of X is defined as $\frac{1}{X}$. If any $X_i = 0$, the reciprocal should be set equal to zero. Add the reciprocals of all of the X's together. Use the floating point mode.

CHAPTER 5: BASIC INPUT/OUTPUT STATEMENTS

One great advantage of computers is their ability to handle great quantities of data according to a fixed set of instructions. The data may vary from time to time, thus requiring a re-execution of the program with new or changed data.

To minimize programming cost, a program should not need to be rewritten each time the data changes. This can only be achieved if the program is written in such a manner that it calls in data for computations and prints out results, without affecting the program. Thus, the data may change without the programmer being concerned.

For example, suppose a payroll problem is being programmed. Taken in its simplest form, the number of hours a man works is to be multiplied by the rate he is paid for each hour he works; from this quantity are subtracted certain fixed amounts and percentage amounts for such items as hospitalization and Federal Income Tax. Instead of writing a program each week for all employees, it is essential that a method be found whereby the program can handle the payroll each week without being rewritten. This is the role of input/output. The computer can be instructed to read an IBM punched card and recognize the following things from that card:

1. The first 20 columns contain the employee's name.

2. The next five columns indicate the total number of hours the employee worked in the payroll period.

3. The next five columns indicate the hourly rate the employee is to receive.

4. The succeeding columns indicate whether deductions are to be taken out of the employee's pay for hospitalization, etc., and in the case of income tax, the number of dependents claimed.

The computer, through the program, is then instructed to read a card, use the information from that card to compute the salary, and print out a check for the proper amount to the man named on the card.

Although this example is highly simplified, it illustrates the manner of holding the programming fixed while the data changes, and illustrates the use and usefulness of input/output of data.

Most input/output statements specify three things:

1. What is to be done. This may be to read a card, print a line, read a magnetic tape, etc.

2. How the data is arranged for input or is to be arranged, for output. This consists of specifying format (e.g., what card columns represent a particular item, etc.). 650 FORTRAN has a fixed format for data; the other processors permit the programmer to specify how data is arranged.

Format specifications for 650 FORTRAN are given in the 650 reference manuals. For those systems which permit the programmer to specify format, a description of the FORMAT statement is found in Part III of this manual.

3. What data is to be transmitted; i. e., by what variable names the program will refer to the particular data items.

READ

This statement causes data to be read from an IBM punched card.

General Form of the READ Statement
"READ n, List" where n is the statement number of a FORMAT statement, and List is as described below.

Examples:

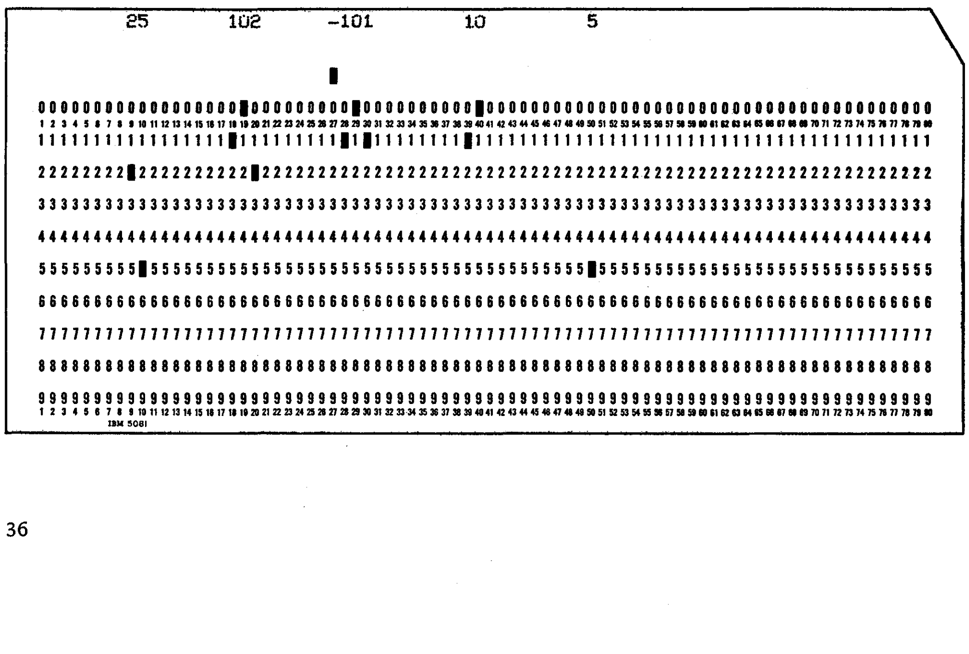
```
READ  1, A
READ  1, A(2), B(3)
READ  2, JOB, A
```

The n portion of all input/output statements is the statement number of a FORMAT statement. It is optional in 650 FORTRAN and will be ignored when it appears. If n is not used, the statement must be written with a comma after READ and before List.

The READ statement causes data to be read from a card and causes those quantities transmitted to the computer to become the values of the variable names given in the List.

LISTS FOR TRANSMISSION OF DATA

The List actually specifies what quantities are to be transmitted. For example, assume that a card is punched as follows:



Further, assume that the following statement appears in the source program:

```
READ5, I,J,K,L,M
```

The card will be read and the program will operate upon the data as though the following statements had been written:

```
 I = 25
 J = 102
 K = -101
 L = 10
 M = 5
```

Computations may take place, and then control may pass back to the READ statement. For the next iteration of the statement, I, J, K, L, and M will have new values depending upon what is punched in the next card to be read.

Indexing in Input/Output Lists

Some systems permit DO-type notation for the transmission of data. For example, suppose it is desired to transmit the five quantities A(1), A(2), A(3), A(4) and A(5). This may be accomplished by writing

```
READ 2, (A(I), I = 1, 5)
```

This would be very roughly equivalent to

```
      DO 12 I = 1, 5
12    READ 2, A(I)
```

In other words, I would be given the value 1 and the first quantity would become the value of A(1). I would then be increased by 1, and the second quantity would become the value of A(2). This would continue until the fifth quantity to be input becomes the value of A(5).

As with DO statements, a third indexing parameter may be used to specify the amount by which the index is to be incremented at each iteration. Thus

```
READ 9, (A(I), I = 1, 10, 2)
```

causes transmission of values for A(1), A(3), A(5), A(7) and A(9).

DO-TYPE NOTATION IN INPUT/OUTPUT LISTS

General Form of DO-Type Notation
"(v_1(i), v_2(i), . . . , v_n(i), i = m_1, m_2, m_3)" where each v is a variable name, i is a non-subscripted integer variable, and m_1, m_2 and m_3 are each either an unsigned integer constant or non-subscripted integer variable. If m_3 is not stated, it is taken to be 1.

As with DOs, i is the index, m_1 is the initial value, m_2 is the test value and m_3 is the increment. In addition, this notation may be nested.

Example:

```
((C(I, J), D(I, J), I = 1, 5), J = 1, 4)
```

would transmit data in the form

```
C(1,1),D(1,1),C(2,1),D(2,1),...,C(5,1),D(5,1),C(1,2),D(1,2),...,
     C(5,4),D(5,4)
```

Additional Details of Input/Output Lists

Any number of quantities may appear in a single list. Integer and floating point quantities may be transmitted by the same statement. However, each quantity must have the correct format as specified in a corresponding FORMAT statement.

If there are more quantities to be transmitted than there are in the list, only the number of quantities specified in the list are transmitted, and remaining quantities are ignored. Thus, if a card contains three quantities and a list contains two, the third quantity is lost.

Conversely, if a list contains more quantities than the input record (for READ, a single IBM card), succeeding cards will be read until all the items specified in the list have been transmitted.

In some processors, when an array name appears in an input/output list in non-subscripted form, all of the quantities in the array are transmitted. For example, the statements

```
DIMENSION A(25)
PRINT 1, A
```

may cause all of the quantities A(1), . . . , A(25) to be printed.

A more complex list is

```
A,B(3),(C(I),D(I,K),I=1,10),((E(I,J),I=1,10,2),F(J,3),J=1,K)
```

Notes:

1. Each item (variable, subscripted variable or parenthetical expression) in the list is separated by a comma.

2. The range of the implied DO statement must be clearly defined by means of parentheses.

3. Only variables (not constants) may appear in the list, except as indexing parameters or as subscripts.

4. A variable indexing parameter, in this case K, must be previously defined by the program before the list is given (unless it is defined by a preceding item in the input/output list).

The above statement will transmit the data 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(3, 1), . . . , E(9, 1), F(1, 3), E(1, 2), E(3, 2), . . . ,
     E(9, 2), F(2, 3), . . . , E(9, K), F(K, 3)
```

This is equivalent to 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, 2
8.    E(I, J)
9.    F(J, 3)
```

PUNCH

General Form of the PUNCH Statement
"PUNCH n, List" where n is the statement number of a FORMAT statement, and List is as described above.

Examples:

```
PUNCH 2, A
PUNCH 3, A(1), JOBNO, B(3)
```

As with READ, n need not be coded for 650 FORTRAN.

This statement is used for output; the items in the list are transmitted from the computer to the card punch. The value of the variable, as determined by the program, is punched into the card according to the format specified.

PROBLEMS

1. Write the DIMENSION statement and the required statement to transmit a 10 x 10 array, A, to the computer.

2. Write the required statement to transmit to the computer the following data:
   A(1), A(2), A(3), A(4), A(5), BJOB, NEXT, DELTA(2), E(3), E(5), E(7), E(9), E(11).

3. Write a statement to punch a card with the following data:
   F(2, 2), G(1, 4), G(2, 4), G(3, 4)

PART III: ADDITIONAL LANGUAGE FACILITIES

This Part consists of two chapters. Chapter 6, Input/Output, is not applicable to 650 FORTRAN. Chapter 7, Subroutines, is not specifically applicable to any single FORTRAN processor, but includes information that is of general interest to all FORTRAN users.

CHAPTER 6: INPUT/OUTPUT

This chapter provides information regarding format specification, and presents some additional input/output statements.

PRINT

General Form of the PRINT Statement
"PRINT n, List" where n is the statement number of a FORMAT statement, and List is a list of quantities for transmission.

Examples:

```
PRINT 2, (A(J), J=1, 10)
PRINT 5, A, I, NEXT, SOME
```

The PRINT statement causes the object program to print output data on the on-line printer. Successive lines are printed in accordance with the FORMAT statement, until the complete list has been satisfied.

Each print line has 120 print positions. As many lines as desired may be printed. Each line is built up by appropriate PRINT and FORMAT statements.

SPECIFYING FORMAT

In order for quantities to be transmitted from the external medium (the IBM card or IBM magnetic tape) to the computer, or from the computer to the external medium (card, magnetic tape, or printed line), it is necessary that the computer be informed in what form the data exists.

For purposes of simplification, the following discussion of format will deal first with the printed line. The concepts developed will later be extended to cover all permissible input/output.

FORMAT

This statement specifies "how" the data is to be transmitted.

General Form of the FORMAT Statement
"FORMAT (Specification)" where Specification is as described below. The Specification must be enclosed in parentheses.

Examples:

```
FORMAT (I15)
FORMAT (I4/F8.4)
FORMAT (E10.6,(I8))
```

Conversion of Numeric Data

Three types of conversion for numeric data are:

Internal	Conversion Code	External
Floating Point	E	Floating Point (with exponent)
Floating Point	F	Floating Point (without exponent)
Integer	I	Integer

Numbers printed by E-type conversion are printed as a decimal fraction to a power of 10. These numbers are normalized; that is, their first significant digit is to the right of the decimal point. For example:

232.3	may be printed as	0.2323Eb03
.003	may be printed as	0.30E-02
17.4	may be printed as	0.174Eb02

Numbers printed by F-type conversion are printed in a "normal" fashion; that is, they appear as output in a meaningful decimal notation without an exponent. Typical output might be:

```
 12.3   -0.726   102.
-17.2    1.318  -968.
289.1    0.009   721.
```

Numbers printed by I-type conversion are printed as integers. Typical output might be:

```
  12
 -17
2342
```

These basic numeric field specifications are given in the forms:

Iw   Ew.d   Fw.d

where I, E or F represents the type of conversion,

w represents the field width for the converted data, and

d represents the number of decimal places to the right of the decimal point.

The decimal point between the w and d portions of the specification is required punctuation.

I-CONVERSION

Thus, the specification

I10

may be used to print a number which exists in the computer as an integer quantity. 10 print positions are reserved for the number. It is printed in this ten-space field right-justified (that is, the units position is at the extreme right). If the number converted is greater than 10 spaces, the excess is lost; no rounding occurs. If the number has less than 10 digits, the leftmost spaces are filled in with blanks. If the quantity is negative, the space preceding the leftmost digit will contain a minus sign if sufficient spaces have been reserved.

The following examples show how each of the quantities on the left is printed according to the specification I3:

Internal	Printed	
721	721	
-721	721	*
-12	-12	
9	bb9	
8114	114	*
0	bb0	
-5	b-5	

*Inaccurate due to insufficient specification
(b is used here to indicate blanks.)

F-CONVERSION

For F-type conversion, w is the total field reserved, and d is the number of places to the right of the decimal point (the fractional portion). The fractional portion is truncated from the right if insufficient spaces are reserved; zeros are filled in from the right if excessive spaces are reserved. Within the remainder of the field, the integer portion is handled in much the same fashion as numbers converted by I-type conversion.

Included in the count, w, must be a space for the decimal point and a space for the sign. (For output, space for at least one digit preceding the decimal point should be reserved.)

The following example shows how each of the quantities on the left is printed according to the specification F5.2:

Internal	Printed	
12.17	12.17	
-41.16	41.16	*
-.2	-0.20	
7.3542	b7.35	**
-1.	-1.00	
9.03	b9.03	
187.64	87.64	*

*Inaccurate due to insufficient specification
**Last two digits of accuracy lost due to insufficient specification

E-CONVERSION

For E-type conversion, the fractional portion is again indicated by d. w includes the field d, plus spaces for a sign and the decimal point, plus four spaces for the exponent. (For output, space for at least one digit preceding the decimal point should be reserved.) The exponent is the power of 10 to which the number must be raised to obtain its true value. The exponent is written with an E followed by a space for a minus sign if the exponent is negative or a plus or blank if the field is positive, and two spaces for the exponent.

The following example shows how each of the quantities on the left is printed according to the specification E10.3:

Internal	Printed	
238.	0.238Eb03	
-.002	-0.200E-02	
.00000000004	0.400E-10	
-21.0057	-0.210Eb02	*

*Last three digits of accuracy lost due to insufficient specification

It is evident from the above examples that the programmer must know the data in order to specify a satisfactory format. Insufficient format specifications can result in inaccurate output. In general, specifications should provide for the largest quantities to be transmitted and the greatest accuracy desired.

Additional Rules for Specifying Format

The following rules permit variation in specifying format:

1. Field width may be specified greater than required in order to provide spacing. Thus, if a number is to be converted by I-type conversion and the number is not expected to exceed five spaces including sign, a specification of I10 will reserve five leading blanks.

2. A specification may be repeated as many times as desired (within the limits of the output device) by preceding the specification with an unsigned fixed point constant. Thus (2F10. 4) is equivalent to (F10. 4, F10. 4).

3. Succeeding specifications may be written in a single FORMAT statement by separating them with commas. Thus (I2, E10. 2) might be used to convert two separate quantities, the first integer and the second floating point.

4. The specifications in a FORMAT statement must have correspondence in mode with the items in the input/output statement; integer quantities require integer conversion, and floating point quantities require floating point conversion.

   Thus, the following statements are compatible:

```
             PRINT 2, A, B, I
      2      FORMAT (2F6. 4, I10)
```

5. Successive items in the input/output list are transmitted by successive corresponding specifications in the FORMAT statement until all items in the list are transmitted. If there are more items in the list than there are specifications, control transfers to the preceding left parenthesis of the FORMAT statement.

   For example, suppose the following statements are written into a program:

```
             PRINT 10, A, B, C, D, E, F, G
     10      FORMAT (F10. 3, E12. 4, F12. 2)
```

   then the following table shows the variable transmitted in the column on the left, and the specification by which it is converted in the column on the right.

Variable Transmitted	Specification
A	F10. 3
B	E12. 4
C	F12. 2
D	F10. 3
E	E12. 4
F	F12. 2
G	F10. 3

6. Quantities are transmitted to consecutive print positions, starting in print position 1. Quantities transmitted in excess of the 120 print positions will be lost.

7. 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). An additional level of parentheses is not permitted. Thus FORMAT (2(3(I6, E10. 2))) is not valid.

Multi-Record Format

To deal with a block of more than one line of print, a FORMAT specification may have several different one-line formats, separated by a slash (/) to indicate the beginning of a new line.

Thus

```
2   FORMAT (3F9.2,2F10.4/8E14.5)
```

would specify a multi-line block of print in which lines 1, 3, 5, ... have format (3F9. 2, 2F10. 4), and lines 2, 4, 6, ... have format (8E14. 5).

If a multi-line format is desired such that the first two lines will be printed according to a special format and all remaining lines according to another format, the last line specification should be enclosed in a second pair of parentheses; for example:

```
FORMAT (I2,3E12.4/2F10.3,3F9.4/(10F12.4))
```

If data items remain to be transmitted after the format specification has been completely "used," the format repeats from the last left parenthesis.

Blank lines may be introduced into a FORMAT statement by listing consecutive slashes. N+1 consecutive slashes produce N blank lines.

Unit Record

The discussion so far has been concerned only with printed output. At this point the discussion will be extended to all input/output by introducing the concept of unit record. This will apply to those aspects of input/output already discussed as well as those yet to be discussed. Except where noted, all references to printed line also apply to other input/output records.

A unit record may be:

1. A printed line with a maximum of 120 characters.

2. A punched card with a maximum of 72 characters. (Although the standard 80 column IBM card is used, the last 8 columns are reserved for identifying information and are not usually processed by FORTRAN.)

3. A BCD tape record with a maximum of 120 characters. The use of tape records will be discussed later in this chapter.

Thus, for example, a specification may be written for reading data from cards. Such a specification, used in conjunction with a READ statement, is a means of instructing the computer regarding the appearance of data in the external medium so that the data may properly be converted and assigned as the values of the variables listed in the input list.

Blank Fields

Blank characters may be provided in an output record, or characters of an input record may be skipped, by means of the specification wX where w is the number of blanks provided or characters skipped. When the specification is used with an input record, w characters are considered to be blank regardless of what they actually are, and are skipped over.

For example, if a card has six 10-column fields for integers, and it is not desired to read the second quantity, then the statement

```
FORMAT (I10,10X,4I10)
```

may be used along with the appropriate READ statement.

Alphameric Fields

There are two specifications available for input/output of alphameric information. The specification wH is used for alphameric data which is not going to be processed by the object program; the specification Aw is used for alphameric data which is to be operated upon by the program.

Information handled with the A specification is given a variable or array name and hence can be referred to by means of this name for processing and/or modification. Information handled with the H specification is not given a name and may not be referred to or manipulated in any way.

H-CONVERSION

The specification wH is followed in the FORMAT statement by w alphameric characters. For example,

```
24HbTHISbISbALPHAMERICbDATA
```

Note that blanks are considered alphameric characters and must be included as part of the count w. This is the only case (except for column 6) where blanks are not ignored in FORTRAN statements.

The effect of wH depends on whether it is used with input or output.

1. Input. w characters are extracted from the input record and replace the w characters included with the specification.

2. Output. The w characters following the specification (or the characters which replaced them as a result of input operations) are written as part of the output record.

For example, suppose that the following statements are executed:

```
              PRINT 2
        2     FORMAT (20HTIME/QUANTITYbREPORT)
```

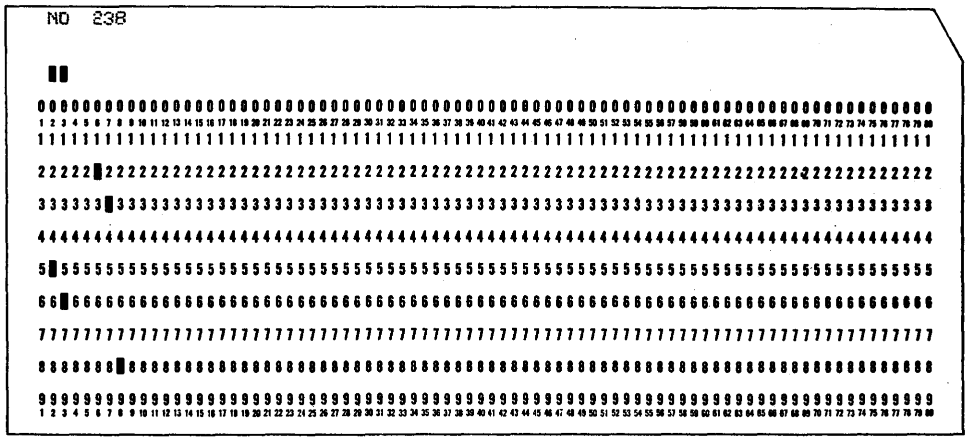
These would cause the following output to be printed:

```
        TIME/QUANTITY REPORT
```

On the other hand, suppose the statements:

```
              READ 1,I
        1     FORMAT (3HYES,I5)
```

are used to read the data card



and then the statement

```
              PRINT 1,I
```

is given. This would cause the following printed output:

```
              bNObb238
```

A-CONVERSION

The specification Aw causes w characters to be read into, or written from, a variable or array name. The name must be constructed in the same manner as an integer or floating point variable name. The maximum field width, w, varies depending upon the processor being used. For example, the statements

```
              PRINT 15,A,B,C,D,E,F
        15    FORMAT (3HXY=,F8.3,A5/)
```

might produce the following lines:

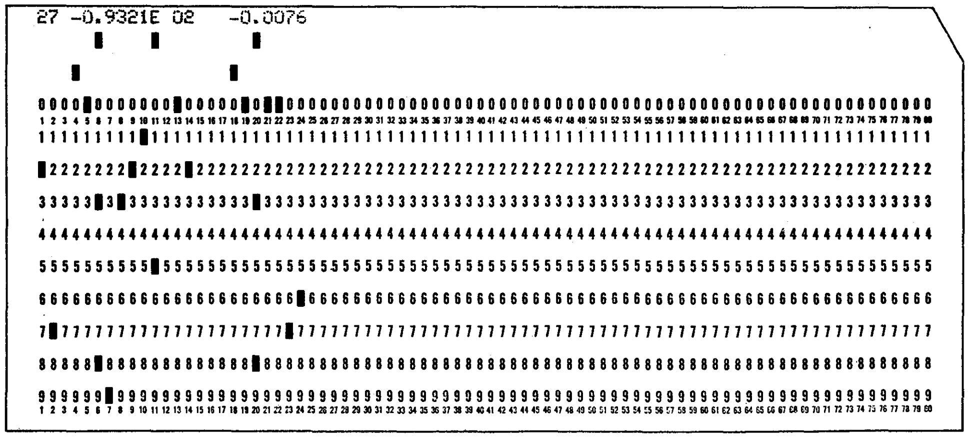
```
XY=b-93.210bbbbb
XY=9999.999OVFLW
XY=bb28.768bbbbb
```

(b is used to indicate blank characters.)

This example assumes that there are steps in the source program which read the BCD word "OVFLW," store this data in the word to be printed in the format A5 when overflow occurs, and stores blanks in the word when overflow does not occur.

Data Input to the Object Program

Numerical input data to be read by means of a READ or READ INPUT TAPE when the object program is executed must be in essentially the same format as given in the previous examples. Thus, a card to be read according to FORMAT (I2,E12.4,F10.4) might be punched



Within each field, all information must appear at the extreme right. Plus signs may be omitted or indicated by a +. Minus signs may be punched with an 11-punch. Blanks in numerical fields are regarded as zeros. Numbers for E- and F-type conversion may contain any number of digits, but only the high order digits of accuracy will be retained if the number exceeds the capacity of the system.

To permit economy in punching, certain relaxations in input data format are permitted.

1. Numbers for E-type 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, E+2, +2, +02, E02 and E+02 are all permissible exponent fields.

2. Numbers for E- and F-type conversion need not have their decimal point punched. If it is not punched, the format specification will supply it. For example, the number -09321+2 with the specification

E12.4 will be treated as though the decimal point had been punched between the 0 and the 9.

If the decimal point is punched in the card, its position overrides the position indicated in the FORMAT specification.

ADDITIONAL INPUT/OUTPUT STATEMENTS

The following statements are not available to all FORTRAN processors; however, they are common to 704, 705, 7070, 709, and 7090 FORTRAN.

WRITE OUTPUT TAPE

General Form of the WRITE OUTPUT TAPE Statement
"WRITE OUTPUT TAPE i, n, List" where i is a tape unit designation (unsigned fixed point constant or fixed point variable), n is the statement number of a FORMAT statement, and List is a list of quantities for transmission.

Examples:

```
WRITE OUTPUT TAPE 42,30,(A(J),J=1,10)
WRITE OUTPUT TAPE L,30,(A(J),J=1,10)
```

The WRITE OUTPUT TAPE statement causes the object program to write BCD information on tape unit i.

Further information regarding writing and reading tapes, I/O unit designation, and carriage control for off-line printing is included in the individual programming and operating manuals.

READ INPUT TAPE

General Form of READ INPUT TAPE Statement
"READ INPUT TAPE i, n, List" where i is a tape unit designation (unsigned fixed point constant or fixed point variable), n is the statement number of a FORMAT statement, and List is a list of quantities for transmission.

Examples:

```
READ INPUT TAPE 24,30,K,A(J)
READ INPUT TAPE N, 30,K,A(J)
```

The READ INPUT TAPE statement causes the object program to read BCD information from tape unit i. Record after record is brought in, in accordance with the FORMAT statement, until the complete list has been satisfied.

END FILE

General Form of the END FILE Statement
"END FILE i" where i is a tape unit designation (unsigned fixed point constant or fixed point variable).

Examples:

```
END FILE 29
END FILE K
```

The END FILE statement causes the object program to write an end-of-file mark on tape unit i.

REWIND

General Form of the REWIND Statement
"REWIND i" where i is a tape unit designation (unsigned fixed point constant or fixed point variable).

Examples:

```
REWIND 3
REWIND K
```

The REWIND statement causes the object program to rewind tape unit i.

BACKSPACE

General Form of the BACKSPACE Statement
"BACKSPACE i" where i is a tape unit designation (unsigned fixed point constant or fixed point variable).

Examples:

```
BACKSPACE 3
BACKSPACE K
```

The BACKSPACE statement causes the object program to backspace tape unit i by one record.

Some systems have additional storage facilities and/or output devices. Appropriate commands for these facilities are found in the individual reference manuals.

PROBLEMS

1. Write the minimum specification by which each of the following may be output (allow a blank for preceding plus signs).

   a. 12.256
   b. 18.032
   c. 27
   d. -0.3201E-04
   e. 0.2000Eb17
   f. 0.071Eb26
   g. -0.178E-01
   h. -321
   i. 1.12
   j. -7.01

2. Write the statements required to punch a card with the words THE FOLLOWING ARE PAYROLL CARDS

3. Which of the following are valid statements?

   a. FORMAT (I3,E12.8/(5F10.2))
   b. PRINT 2,A,(B(I),I=1,10)
   c. FORMAT (3I2)
   d. PRINT A
   e. PRINT 276,A,I,B,J
   f. FORMAT (3F10.4)
   g. PRINT 4,A,B,3.2,D
   h. FORMAT (2(F10.4,I4))
   i. FORMAT (20HGObTObNEXTbJOB,E10.6,I3)
   j. FORMAT (I100,F18.9,E15.7)

CHAPTER 7: SUBROUTINES

Suppose that a program is being written which, at various points, requires identical kinds of computation, with different data. It would simplify writing that program if the computation could be written only once and then could be referenced freely, each reference having the same effect as though the computation were written completely at the place where reference was made.

Likewise, programming would be simplified if pre-written routines could be easily incorporated when desired. For example, to take the square root of a number, a program must be written with this object in mind. If a program is already written to take a square root, it would be desirable to be able to incorporate that program into other programs where square root calculations are required.

FORTRAN provides for both of the above situations through the use of FORTRAN-coded subroutines and/or machine-language subroutines. A subroutine is considered to be any sequence of instructions which performs some desired operation.

Provisions for subroutines vary among processors; therefore, the reader is referred to the individual reference manuals for precise information concerning this subject. However, as an introduction to those manuals, the following is a generalized discussion of subroutines and a specialized sub-set of subroutines called functions.

FUNCTIONS

In mathematics, a function is a statement of the relationship among a number of variables; its value depends upon the values assigned to the variables (arguments) of the function. The same definition of function is true in FORTRAN, with one restriction. Whereas a function may normally be thought of as having one or more than one value, a function in FORTRAN always has a single value.

Suppose it is desired to use the function

$$f(x) = 7x^2+5x+3$$

Thus, f(x) states a series of computations to be carried out regardless of the value assigned to x. The value of f(x) will depend upon the value assigned to x, which is the argument of the function.

To use a function in FORTRAN, it is necessary to:

1. Define the function

2. Call the function

These two steps will be illustrated here and discussed later.

Definition. The following statement might define the function discussed above:

```
SOMEF(X)=7.0*X**2+5.0*X+3.0
```

Calling. The following statement employing SOMEF(X) might be used:

```
A=3.5+SOMEF(Y+Z)
```

In this use of SOMEF(X), the quantity Y+Z will be substituted wherever X appeared in the definition. SOMEF will then be computed and its value will be added to 3.5 to produce a value for A. The result will be the same as if the following statement had been written

```
A=3.5+7.0*(Y+Z)**2+5.0*(Y+Z)+3.0
```

Defining Functions

There are three steps in the definition of a function in FORTRAN:

1. The procedure for finding the value of the function must be stated. (This may be an arithmetic expression.)

2. The arguments of the function must be stated. All quantities which affect the value of the function and which are not arguments are treated as parameters; their values must be set before the function may be evaluated.

3. The function must be assigned a unique name by which it may be called.

FORTRAN provides four ways of stating the procedure for finding the value of a function.

1. Arithmetic statement functions. These functions are defined by a single arithmetic statement in the source program.

2. Built-in functions. These are functions which are pre-defined and exist in the processor. They are significantly different from other functions in that they are open subroutines. An open subroutine is one that is incorporated into the object program each time it is referred to by the source program. All other subroutines (closed subroutines) appear only once in the object program, regardless of the number of times they are referenced in the source program.

3. Library functions. These functions are pre-defined and exist in a program library, which may be a card deck or a magnetic tape. Library functions are originally coded in machine language.

4. FUNCTION subprogram. These functions are pre-defined and may exist in a program library. They differ from other functions in that they may originally be coded in FORTRAN and may consist of more than one FORTRAN statement.

Each type of function:

1. May use other functions in its definition.

2. May have as many of the variables as desired stated as arguments.

3. Must have names formed in accordance with the rules for naming functions. The rules for naming arithmetic statement functions, built-in functions, and library functions are the same; the rules for forming FUNCTION subprogram names are different.

Calling Functions

When the name of a function appears in any FORTRAN arithmetic expression, it is construed to be a call-in of the subroutine to evaluate the named function; thus, the appearance of the function with its arguments serves to cause the computations indicated by the function definition.

The following rules must apply when calling functions:

1. The function name indicates the mode of the single value that will be the result of the function evaluation. The function name may appear anywhere in an expression that a subscripted variable of the same mode may appear.

2. When calling a function, the name must be followed by parentheses enclosing the function arguments. These arguments must correspond in number, order and mode with the arguments which appeared in the function definition.

ARITHMETIC STATEMENT FUNCTIONS

At this point, in order to clarify some of the concepts that have been developed, some of the rules for using arithmetic statement functions will be discussed below.

The statement defining an arithmetic statement function closely resembles a conventional arithmetic statement:

```
FUNF(A,B)=3.*A/B**2
```

FUNF is the function name
A and B are the function arguments
The expression on the right defines only the type of computation to be performed; it causes no computations to be performed.

This function might be used in the following arithmetic statement:

```
C=FUNF(D,E)
```

which would be exactly equivalent to writing the arithmetic statement

```
C=3.*D/E**2
```

Note the correspondence between A and B in the function statement and D and E in the arithmetic statement.

The quantities enclosed in parentheses following the function name are the arguments of the function. They are dummy variables for which substitution is made when the function is used in an arithmetic statement. Any of the variables in the expression on the right may be listed as arguments in the parentheses following the function name on the left; however, those quantities on the right which are not arguments must have their values assigned by the program before the function is referred to in an arithmetic statement.

Example:

The arithmetic statement function

```
OTHERF(X,Y,I)=A*X+Y/2.-E**I
```

used in the arithmetic statement

```
D=B+C-OTHERF(R,Q,J)
```

would cause the following computations

```
D=B+C-(A*R+Q/2.-E**J)
```

General Form of an Arithmetic Statement Function Name
The name of the function consists of alphameric characters (except special characters), of which the last must be F and the first must be alphabetic. The first character must be X if and only if the value of the function is to be an integer.   The name of the function is followed by parentheses enclosing the arguments separated by commas.   The minimum and maximum number of characters in a function name varies among FORTRAN processors. For the following discussion, assume the name must consist of 4 to 7 characters.

Examples:

```
ABSF(B)
XMODF(MIN,K)
COSF(A)
FIRSTF(Z+B,Y)
```

Note that the integer indicators I, J, K, L, M, and N do not cause a function to have an integer value; an arithmetic statement function will have an integer value only if its first character is X.

Defining Arithmetic Statement Functions

These are functions which are defined by a single FORTRAN arithmetic statement and apply only to the particular program or subprogram in which their definition appears.

General Form of an Arithmetic Statement Function
"a=b" where a is a function name 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 defined previously.

Examples:

The following are valid arithmetic statement functions:

```
FIRSTF(X)=A*X+B
SECONDF(X,B)=A*X+B
THIRDF(D)=FIRSTF(E)/D
MAXF(A,I)=A**I+B-C
```

The following are not valid arithmetic statement functions:

```
MAXF(A,I)=A**B+I       (expression on the right is mixed mode)
NEXTF(3,J,K)=3*I+J**K  (arguments must be variables)
SOMEF(A(I))=A(I)/B+3   (argument must be non-subscripted in
                        the function definition)
```

As many as desired of the variables appearing in the expression on the right-hand side of the equal sign 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 floating point 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. Thus, if FIRSTF is defined in a function statement as FIRSTF(X)=A*X+B, then a later reference to FIRSTF(Y) will cause ay+b, based on the current values of a, b and y, to be computed.

The arguments of an arithmetic statement function reference may be expressions and may involve subscripted variables; thus a reference to FIRSTF(Z+Y(I)), with the above definition of FIRSTF, will cause $a(z+y_i)+b$ to be computed based on the current values of a, b, y_i and z.

Rule: All of the arithmetic statements defining functions to be used in a program must precede the first executable statement of the program.

Using Arithmetic Statement Functions

Functions may be used in arithmetic statements in the same manner as subscripted variables. As with the use of subscripted variables, the mode of an expression may not be mixed.

Examples:

The following are valid uses of functions:

```
A=SOMEF(B, C)+D
B=XSOMEF(K, L)+I/J
C=OTHERF(X+Y-Z, I)/SOMEF(E, F)
```

The following are not valid uses of functions:

```
I=A+XSOMEF(H, R)        (expression is mixed)
JOB1=MAX**MINF(I, J)    (an integer quantity may not have
                         a floating point exponent)
```

SUBROUTINE SUBPROGRAMS

In addition to functions, FORTRAN provides for the use of SUBROUTINE subprograms.

These subroutines are pre-defined and may exist in a program library. They may be coded in FORTRAN and may consist of more than one FORTRAN statement. These subroutines differ from functions in two ways:

1. They may not be referenced by their appearance in an arithmetic expression. To be called, they require a special statement named a CALL statement.

2. They may return more than one value.

MACHINE-LANGUAGE SUBROUTINES

All FORTRAN processors provide for the use of subroutines which have been coded in machine language. These subroutines may be given FORTRAN-type names and may be called by FORTRAN statements. These subroutines must be written in accordance with rules found in the individual reference manuals.

PART IV: ANALYSIS OF INDIVIDUAL FORTRAN SYSTEMS

This portion of the manual will deal with the individual FORTRAN processors and their characteristics.

Chapter 8 is divided into 8 sections which deal individually with each FORTRAN processor. These sections will state most of the more important characteristics and special features of each processor such as the magnitude of fixed point constants, types of subroutines available, etc.

Chapter 9 gives a chart which shows the commands available for each processor and gives the general form and examples of each statement in the FORTRAN language.

Additional details regarding the individual processors, actual machine compilation, object program execution, and so forth, are available in separate programming and operations reference manuals.

CHAPTER 8: SYSTEM SPECIFICATIONS

The following specifications are in addition to, and supplement, the rules given in preceding sections for writing FORTRAN programs.

SECTION 1: 650 FORTRAN

Note: Users of the IBM 650 may also be interested in Section 2, "650 FOR TRANSIT."

Source Machine. Basic IBM 650 with index registers, alphabetic device, and special character device — Group II.

Object Machine. Same as above, plus the floating point arithmetic device.

Representation of Integer Constants. 1 to 10 decimal digits.

Representation of Floating Point Constants. 1 to 8 decimal digits plus optional exponent. The magnitude of the constant must lie between approximately 10^{-50} and 10^{49} or be zero.

Length of Variable Names. 1 to 5 alphameric characters.

Subscripts. Two subscripts are permitted for any variable.

Arrangement of Arrays in Storage. Arrays are stored columnwise in the form $A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, \ldots, A_{m,n}$.

Statement Numbers. Must consist of 4 decimal characters including leading zeros.

Statements. Maximum of 125 characters excluding blanks, continued on as many cards as desired. Each initial card of a statement (whether continued or not) must have a zero punch in column 6. Continuation cards must have a non-zero punch in column 6. If a statement with a statement number is continued, the statement number must be punched in each continuation card. Statements are punched in columns 7-36 only; columns 37-80 must be blank.

Subroutines. Built-in functions are available and provision is made for Library functions. Machine language or symbolic language functions may be added to individual programs.

SECTION 2: 650 FOR TRANSIT

Source Machine. IBM 650 with special features as outlined in reference manual.

Object Machine. Basic IBM 650. (FOR TRANSIT II also requires index registers and automatic floating decimal arithmetic.)

Representation of Integer Constants. 1 to 10 decimal digits.

Representation of Floating Point Constants. 1 to 8 decimal digits plus optional exponent. The magnitude of the constant must be between 10^{-50} and 10^{50} or be zero.

Length of Variable Names. 1 to 5 alphameric characters.

Subscripts. Two subscripts are permitted for any variable.

Arrangement of Arrays in Storage. Arrays are stored columnwise in the form $A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, \ldots, A_{m,n}$.

Statement numbers. For FOR TRANSIT I and II, a statement number of 0000 to 0999 is required for all statements (all leading zeros must be punched). For FOR TRANSIT I(S) and II(S), a statement may or may not have a number, and that number may range from 1 to 999 without leading zeros.

Statements. The specific format for writing statements varies between FOR TRANSIT and FOR TRANSIT(S), and reference should be made to the manual.

Subroutines. Built-in functions are available, and function subroutines may be added by the programmer.

SECTION 3: 1620 FORTRAN

Note: Some 1620 FORTRAN systems use punched tape for input/output and thus the rules for preparing FORTRAN statements for these systems vary from other FORTRAN systems.

Source Machine. Any IBM 1620 with the same size storage and the same input/output devices as the object machine.

Object Machine. Any IBM 1620 with the same size storage and the same input/output devices as the source machine.

Representation of Integer Constants. 1 to 4 decimal digits.

Representation of Floating Point Constants. Any number of decimal digits, plus optional exponent, with magnitude between 10^{-50} and 10^{49}, or zero.

Length of Variable Names. 1 to 5 alphameric characters.

Subscripts. Two subscripts are permitted for any variable. Where c is a constant and v is a variable, each subscript may take one of the following forms:

c
v
v+c
v−c

Arrangement of Arrays in Storage. Arrays are stored columnwise in the form $A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, \ldots, A_{m,n}$.

Statement Numbers. Any number from 1 to 9999.

Statements. 72 characters including blanks; continuation cards are not permitted.

Subroutines. Only built-in functions are available.

• Input/Output. DO-type notation not permitted in Lists. A conversion not available.

SECTION 4: BASIC 7070/7074 FORTRAN

Note: Users of Basic 7070/7074 FORTRAN may also be interested in Section 5, "7070/7074 FORTRAN."

Source Machine. IBM 7070/7074 with minimum core storage.

Object Machine. Any IBM 7070/7074 with sufficient core storage, regardless of whether the source program is translated on the 7070 or 7074.

Representation of Integer Constants. 1 to 10 decimal digits.

Representation of Floating Point Constants. Any number of decimal digits, plus optional exponent, with magnitude between 10^{-50} and 10^{49}, or zero.

Length of Variable Names. 1 to 5 alphameric characters.

Subscripts. Two subscripts are permitted for any variable.

Arrangement of Arrays in Storage. Arrays are stored columnwise in the form $A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, \ldots, A_{m,n}$.

Statement Numbers. 1 to 5 digits; leading zeros are not required.

Statements. Columns 7-72 are used for the initial card of a statement, and each statement may have up to 9 continuation cards.

Subroutines. Library subroutines are available, and provision is made for symbolic language subroutines.

SECTION 5: 7070/7074 FORTRAN

Source Machine. IBM 7070/7074 with a minimum of six IBM 729 magnetic tape units.

Object Machine. Any IBM 7070/7074 with sufficient core storage regardless of whether the source machine was a 7070 or 7074.

Representation of Integer Constants. 1 to 10 decimal digits with magnitude less than the size of core storage.

Representation of Floating Point Constants. Any number of decimal digits, plus optional exponent, with magnitude between 10^{-50} and 10^{49}, or zero.

Length of Variable Names. 1 to 6 alphameric characters.

Subscripts. Three subscripts are permitted for any variable.

Arrangement of Arrays in Storage. Arrays are stored columnwise in the form $A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, \ldots, A_{m,n}$. Thus, the first of the subscripts varies most rapidly, and the last varies least rapidly. The same is true of 3-dimensional arrays.

Statement Numbers. 1 to 5 decimal digits.

Statements. Each statement may have as many as nine continuation cards.

Subroutines. Built-in functions, Library functions, and Arithmetic Statement functions are available as well as FORTRAN Functions and SUBROUTINE subprograms. Provision is made for symbolic language routines.

SECTION 6: 705 FORTRAN

Source Machine. IBM 705 with 40,000 memory positions and 8 tapes.

Object Machine. Any IBM 705 with sufficient memory.

Representation of Integer Constants. 1 to 10 decimal digits.

Representation of Floating Point Constants. Any number of decimal digits, plus optional exponent, with magnitude between 10^{-99} and 10^{99}, or zero.

Length of Variable Names. 1 to 10 alphameric characters.

Subscripts. Three subscripts are permitted for any variable.

Arrangement of Arrays in Storage. Arrays are stored row-wise (unless modified by the input/output list) in the form $A_{1,1}, A_{1,2}, \ldots, A_{1,m}, A_{2,1}, \ldots, A_{n,m}$. Thus, the first of the subscripts varies least rapidly, and the last varies most rapidly. The same is true of 3-dimensional arrays.

Statement Numbers. 1 to 5 digits.

Statements. Each statement may have up to nine continuation cards.

Subroutines. Library functions and Arithmetic Statement functions are available.

Input/Output. X specification not permitted.

SECTION 7: 704 FORTRAN

Source Machine. IBM 704 with at least 4,096 storage locations, 1 drum, 4 tape units, 1 on- or off-line card punch, 1 on-line card reader, and 1 on-line printer.

Object Machine. Any IBM 704 with sufficient core storage.

Representation of Integer Constants. 1 to 5 decimal digits with magnitude $< 2^{17}$.

Representation of Floating Point Constants. Any number of decimal digits, plus optional exponent, with magnitude between 10^{-38} and 10^{38}, or zero.

Length of Variable Names. 1 to 6 alphameric characters.

• Subscripts. Three subscripts are permitted for any variable. The value of a subscript may not exceed the size of core storage.

Arrangement of Arrays in Storage. Arrays are stored columnwise in the form $A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, \ldots, A_{m,n}$. Thus the first of the subscripts varies most rapidly, and the last varies least rapidly. The same is true of 3-dimensional arrays.

Statement Numbers. 1 to 32767.

Statements. Up to nine continuation cards are permitted for each statement.

Subroutines. All of the subroutine types are available.

Additonal Features. Provision is made for Boolean arithmetic and symbolic machine language in source statements.

SECTION 8: 709/7090 FORTRAN

Source Machine. IBM 709 or 7090 with at least 8,192 storage locations, 5 tape units, 1 on- or off-line card punch, 1 on-line card reader, and 1 on-line printer.

Object Machine. Any IBM 709 or 7090 with sufficient core storage, regardles of whether the source program was compiled on the 709 or 7090.

Representation of Integer Constants. 1 to 5 decimal digits with magnitude $< 2^{17}$.

Representation of Floating Point Constants. Any number of decimal digits, plus optional exponent, with magnitude between 10^{-38} and 10^{38}, or zero.

Length of Variable Names. 1 to 6 alphameric characters.

Subscripts. Three subscripts are permitted for any variable. The value of a subscript may not exceed 2^{15}.

Arrangement of Arrays in Storage. Arrays are stored columnwise in the form $A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, \ldots, A_{m,n}$. Thus the first of the subscripts varies most rapidly, and the last varies least rapidly. The same is true of 3-dimensional arrays.

Statement Numbers. 1 to 32767.

Statements. Up to nine continuation cards are permitted for each statement.

Subroutines. All of the subroutine types are available.

• Additional Features. Provision is made for Boolean, double-precision, and complex arithmetic and for source language debugging.

CHAPTER 9: THE FORTRAN STATEMENTS

The following table indicates the available commands for each of the FORTRAN processors. Statements that are common to all processors are so noted.

	COMMON	650	650 FOR TRANSIT	1620	705	Basic 7070/7074	7070/7074	704	709/7090
ACCEPT n, List				X					
ASSIGN i TO n					X		X	X	X
BACKSPACE i					X	X	X	X	X
CALL NAME $(a_1, a_2, \ldots, a_n)$							X	X	X
COMMON $(a_1, a_2, \ldots, a_n)$							X	X	X
CONTINUE	X	X	X	X	X	X	X	X	X
DIMENSION $v_1, v_2, \ldots, v_n$	X	X	X	X	X	X	X	X	X
DO n $i=m_1, m_2, m_3$	X	X	X	X	X	X	X	X	X
END $(I_1, I_2, I_3, I_4, I_5)$		1	1	2		2	2	2	2
END FILE i					X	X	X	X	X
EQUIVALENCE (a, b, c, . . .), (d, e, f, . . .), . . .			X			X	X	X	X
FORMAT $(s_1, s_2, \ldots, s_n)$				X	X	X	X	X	X
FREQUENCY n(i, j, . . .), m(k, l, . . .), . . .							3	X	X
FUNCTION Name $(a_1, a_2, \ldots, a_n)$							X	X	X
GO TO n	X	X	X	X	X	X	X	X	X
GO TO n, $(n_1, n_2, \ldots, n_m)$					X		X	X	X
GO TO $(n_1, n_2, \ldots, n_m)$, i	X	X	X	X	X	X	X	X	X
IF ACCUMULATOR OVERFLOW n_1, n_2					X	X	X	X	X
IF DIVIDE CHECK n_1, n_2					X	X	X	X	X
IF QUOTIENT OVERFLOW n_1, n_2					X	X	X	X	X
IF (a) n_1, n_2, n_3	X	X	X	X	X	X	X	X	X
IF (SENSE LIGHT i) n_1, n_2					X		X	X	X
IF (SENSE SWITCH i) n_1, n_2				X	X		X	X	X
PAUSE n	4	5	5	4	5	5	5	5	5
PRINT n, List				X	X	X	X	X	X
PUNCH n, List	6	6	6	X	X	X	X	X	X
PUNCH TAPE n, List				X					
READ n, List	6	6	6	X	X	X	X	X	X
READ DRUM i, j, List					3			X	X
READ INPUT TAPE i, n, List					X	X	X	X	X
READ TAPE i, List					X	X	X	X	X
● READ TAPE n, List				7					
RETURN							X	X	X
REWIND i					X	X	X	X	X
SENSE LIGHT i					X		X	X	X
STOP n	4	4	5	4	X	X	X	X	X
SUBROUTINE Name $(a_1, a_2, \ldots, a_n)$							X	X	X
TYPE n, List				X		X	X		
WRITE DRUM i, j, List					3			X	X
WRITE OUTPUT TAPE i, n, List					X	X	X	X	X
WRITE TAPE i, List					X	X	X	X	X

1. I_i are not permitted.
2. I_i are optional and may be ignored.
3. May be included but will be ignored.
4. The n is not permitted.
5. The n is optional and may be ignored.
6. The n is optional and is ignored.
7. This statement was changed to "ACCEPT TAPE n, List" after this manual went to press.

Following is a list of all FORTRAN statements. The general form of the statement is given and explained, followed by an explanation of what the statement does and examples of how it is coded.

GENERAL FORM: ACCEPT n, List where n is the statement number of a FORMAT statement, and List is a list of quantities to be transmitted.

Purpose: This statement causes the program to read information from the console typewriter in accordance with FORMAT statement n and transmit this information into core storage as the values of the variables in the list.

Example: ACCEPT 9,X,Y,Z

GENERAL FORM: ASSIGN i TO n where i is a statement number and n is a non-subscripted integer variable which appears in an Assigned GO TO statement.

Purpose: This statement causes a subsequent GO TO n, $(n_1, n_2, \ldots, n_m)$ to transfer control to statement number i where i is included in the series $n_1, n_2, \ldots, n_m$.

Example: ASSIGN 14 TO J

GENERAL FORM: BACKSPACE i where, depending on the system, i is a symbolic tape name (unsigned integer constant or integer variable) or an actual tape number.

Purpose: This statement causes the object program to backspace tape unit i.

Examples:
```
BACKSPACE 10
BACKSPACE LX
```

GENERAL FORM: CALL Name $(a_1, a_2, \ldots, a_n)$ where Name is the name of a Subroutine subprogram, and each a_i is an argument.

Purpose: This statement is used to call Subroutine subprograms; the CALL transfers control to the subprogram and presents it with the parenthesized arguments.

Examples:
```
CALL MATMPY (X,5,10,Y,7,2)
CALL QDRTIC (P*9.732,Q/4.536,R-S**2.0,X1,X2)
```

GENERAL FORM: COMMON $(a_1, a_2, \ldots, a_n)$ where each a_i is the name of a variable or non-subscripted array name.

Purpose: This statement causes each a_i to be assigned a location in common storage.

Examples: COMMON A, B, C, D, E
COMMON X, ANGLE, MATA, MATB

GENERAL FORM: CONTINUE

Purpose: This statement is used as the last statement in the range of a DO when the DO would otherwise end with an IF- or GO TO-type statement (which is not permitted).

Example: CONTINUE

GENERAL FORM: DIMENSION $v_1, v_2, \ldots, v_n$ where each v_i is the name of an array subscripted with 1, 2, or 3 unsigned integer constants. Each subscript indicates the size of one dimension of the array.

Purpose: This statement provides the information necessary to allocate storage in the object program for arrays.

Examples: DIMENSION A(10), B(5,5,5)
DIMENSION J(12,3), E(5)

GENERAL FORM: DO n $i=m_1, m_2, m_3$ where n is a statement number, i is a non-subscripted integer variable, and m_1, m_2, m_3 are each either an unsigned integer constant or a non-subscripted integer variable. If m_3 is not stated, it is taken to be 1.

Purpose: This statement is a command to execute repeatedly the statements which follow, up to and including the statement with statement number n. The first time the statements are executed with $i=m_1$. For each succeeding execution i is increased by m_3. After they have been executed with i equal to the highest of this sequence of values which does not exceed m_2, control passes to the statement following statement n.

Examples: DO 25 J=1, 15
DO 25 J=1, I, 2

GENERAL FORM: END $(I_1, I_2, \ldots, I_n)$ where each I_i, if permitted by the system, is 0, 1 or 2.

Purpose: This statement is used to indicate the end of the source program deck. Each I_i specifies an action to be taken by FORTRAN with regard to the setting of the individual Sense Switches.

Examples: END (0, 0, 0, 0, 1)
END (0, 1, 2, 0, 1)

GENERAL FORM: END FILE i where i, depending on the system, is a symbolic (unsigned integer constant or integer variable) or actual tape unit designation.

Purpose: This statement causes the object program to write an end-of-file mark on tape unit i.

Examples:
```
END FILE 14
END FILE INFLE
```

GENERAL FORM: EQUIVALENCE (a, b, c, . . .), (d, e, f, . . .), ... where a, b, c, d, e, f, . . . are variables optionally followed by a single unsigned integer constant in parentheses.

Purpose: This statement causes all of the variables specified by each parenthetical expression to be assigned the same location in storage.

Examples:
```
EQUIVALENCE (E, F(3))
EQUIVALENCE (I, J(3)), (A, E, G(14))
```

GENERAL FORM: FORMAT ($s_1, s_2, \ldots, s_n$) where each s_i is a format specification.

Purpose: This statement describes the type of conversion and format of data to be used in the transmission of an input/output list.

Examples:
```
FORMAT (I2/(E12.4, F10.4))
FORMAT (I10)
FORMAT (E15.6, F10.6/5I10)
```

GENERAL FORM: FREQUENCY n(i, j, . . .), m(k, l, . . .), ... where n, m, . . . are statement numbers and i, j, k, l, . . . are unsigned integer constants.

Purpose: This statement assists the FORTRAN executive program by indicating the relative frequencies of the branches of various transfer type statements.

Examples:
```
FREQUENCY 30(1, 2, 1), 40(11)
FREQUENCY 50(1, 7, 1, 1), 10(1, 7, 1, 1)
```

GENERAL FORM: FUNCTION Name ($a_1, a_2, \ldots, a_n$) where Name is the name of a function and the a_i are the arguments.

Purpose: This statement is used at the beginning of a FUNCTION-type subprogram to define its name and its arguments.

Examples:
```
FUNCTION ARCSIN(RADIAN)
FUNCTION ROOT(B, A, C)
```

GENERAL FORM: GO TO n where n is a statement number.

Purpose: This statement causes the program to transfer control to statement n.

Example: GO TO 3

GENERAL FORM: GO TO n, $(n_1, n_2, \ldots, n_m)$ where n is a non-subscripted integer variable appearing in a previously executed ASSIGN statement, and $n_1, n_2, \ldots, n_m$ are statement numbers which may have been assigned to n by a previously executed ASSIGN statement.

Purpose: This statement causes transfer of control to the statement with statement number equal to that value of n which was last assigned by an ASSIGN statement.

Example: GO TO K,(17,12,19)

GENERAL FORM: GO TO $(n_1, n_2, \ldots, n_m)$, i where $n_1, n_2, \ldots, n_m$ are statement numbers and i is a non-subscripted integer variable.

Purpose: This statement causes transfer of control to the first, second, etc., item in the list $n_1, n_2, \ldots, n_m$, depending on whether i is 1, 2,..., m.

Example: GO TO (30,42,50,9),J

GENERAL FORM: IF ACCUMULATOR OVERFLOW n_1, n_2 where n_1 and n_2 are statement numbers.

Purpose: This statement causes transfer to statement n_1 if overflow has occurred, or to n_2 if no overflow has occurred.

Example: IF ACCUMULATOR OVERFLOW 10, 7

GENERAL FORM: IF DIVIDE CHECK n_1, n_2 where n_1 and n_2 are statement numbers.

Purpose: This statement causes transfer to statement n_1 if divide check has occurred, or to statement n_2 if divide check has not occurred.

Example: IF DIVIDE CHECK 12,49

GENERAL FORM: IF QUOTIENT OVERFLOW n_1, n_2 where n_1 and n_2 are statement numbers.

Purpose: This statement causes transfer to statement n_1 if overflow has occurred, or to n_2 if overflow has not occurred.

Example: IF QUOTIENT OVERFLOW 12, 8

GENERAL FORM: IF (a) n_1, n_2, n_3 where a is an expression and n_1, n_2, and n_3 are statement numbers.

Purpose: This statement causes transfer of control to the statement numbered n_1 if the expression a is less than zero, n_2 if a is equal to zero, or n_3 if a is greater than zero.

Examples: IF (X+Y) 10, 5, 3
IF (I+3) 2, 7, 4

GENERAL FORM: IF(SENSE LIGHT i) n_1, n_2 where i is the number of a Sense Light and n_1 and n_2 are statement numbers.

Purpose: This statement causes transfer of control to statement n_1 or n_2 if Sense Light i is ON or OFF, respectively.

Example: IF (SENSE LIGHT 3) 30, 40

GENERAL FORM: IF (SENSE SWITCH i) n_1, n_2 where i is the number of a Sense Switch and n_1 and n_2 are statement numbers.

Purpose: This statement causes transfer of control to statement n_1 or n_2 if Sense Switch i is DOWN or UP, respectively.

Example: IF (SENSE SWITCH 3) 30, 108

GENERAL FORM: PAUSE n where n, if permitted in the system, is a number which is to be displayed on the operator's console.

Purpose: This statement causes the object program to halt in such a way that depressing the Start key causes the object program to continue execution with the next statement.

Example: PAUSE 3

GENERAL FORM: PRINT n, List where n, if permitted by the system, is the statement number of a FORMAT statement and List is a list of quantities to be transmitted.

Purpose: This statement causes the items in the list to be printed on-line in the format specified by statement n.

Examples: PRINT 12, A, I, J(3)
PRINT 2, (A(I), I=1, 10, 2)

GENERAL FORM: PUNCH n, List where n, if permitted by the system, is the statement number of a FORMAT statement and List is a list of quantities to be transmitted.

Purpose: This statement causes the items in the list to be punched on-line in the format specified by statement number n.

Examples: PUNCH 10, A, I, J(3)
PUNCH 12, (A(I), I=1, 10, 3)

GENERAL FORM: PUNCH TAPE n, List where n is the statement number of a FORMAT statement and List is a list of quantities to be transmitted.

Purpose: This statement causes the items in the list to be punched on paper tape in the format specified by statement number n.

Examples: PUNCH TAPE 10, A, I, J(3)
PUNCH TAPE 12, A, B, C

GENERAL FORM: READ n, List where n, if permitted by the system, is the statement number of a FORMAT statement and List is a list of quantities to be transmitted.

Purpose: This statement causes IBM cards to be read on-line and causes the values read in to be assigned as the values of the variables in the list.

Examples: READ 4, A, B, C, D
READ 7, (I(J), J=1, 12)

GENERAL FORM: READ DRUM i, j, List where i and j are each either an unsigned integer constant or an integer variable, with the value of i between 1 and 8 inclusive and List is a list of quantities to be transmitted.

Purpose: This statement causes information from drum i, beginning in location j, to be transmitted to core storage and assigned as the values of the variables in the list.

Examples: READ DRUM 3, 400, A, B, C, D(3)
READ DRUM K, J, A, B, C, D(3)

GENERAL FORM: READ INPUT TAPE i, n, List where i, depending on the system, is an unsigned integer constant or an integer variable symbolic tape designation or is an actual tape number; n is the statement number of a FORMAT statement, and List is a list of quantities to be transmitted.

Purpose: This statement causes the program to read BCD information from tape unit i in accordance with FORMAT statement n, and transmit this data into core storage as the values of the variables in the list.

Examples: READ INPUT TAPE 24,30,K,A(J)
READ INPUT TAPE N,30,K,A(J)

GENERAL FORM: READ TAPE i, List where i, depending on the system, is an unsigned integer constant or integer variable symbolic tape designation or is an actual tape unit, and List is a list of quantities to be transmitted.

Purpose: This statement causes the program to read information in internal machine-language notation from tape unit i, and transmit this data into core storage as the values of the variables in the list.

Examples: READ TAPE 24,A,B,C,D
READ TAPE K,A,B,C,D

• GENERAL FORM: READ TAPE n, List where n is the statement number of a FORMAT statement, and List is a list of quantities to be transmitted. *

Purpose: This statement causes the program to read information from paper tape in accordance with FORMAT statement n, and transmit this data into core storage as the values of the variables in the list.

Example: READ TAPE 21,A,B,C

GENERAL FORM: RETURN

Purpose: This statement is used to return control from a subprogram to the main program which called it.

Example: RETURN

GENERAL FORM: REWIND i where i, depending on the system, is an unsigned integer constant or integer variable symbolic tape designation or is an actual tape unit.

Purpose: This statement causes the object program to rewind tape unit i.

Examples: REWIND 3
REWIND K

GENERAL FORM: SENSE LIGHT i where i is the number of a Sense Light to be turned ON. If i is zero, the Sense Lights are turned OFF.

Purpose: This statement permits Sense Lights to be turned ON or OFF so that they may later be tested to cause a program branch by IF (SENSE LIGHT i) n_1, n_2.

Examples: SENSE LIGHT 4

*See footnote 7, page 65.

GENERAL FORM: STOP n where n, if permitted, is a number to be displayed on the operator's console.

Purpose: This statement causes the object program to halt. In most systems, the object program may not be continued after execution of the STOP statement.

Example: STOP

GENERAL FORM: SUBROUTINE Name ($a_1, a_2, \ldots, a_n$) where Name is the symbolic name of a subprogram, and each a_i is an argument.

Purpose: This statement is the first statement of a SUBROUTINE-type subprogram and defines it to be such, as well as defining its name and its arguments.

Examples:
```
SUBROUTINE MATMPY (A,N,M,B,L,C)
SUBROUTINE QDRTIC (B,A,C,ROOT1,ROOT2)
```

GENERAL FORM: TYPE n, List where n is the statement number of a FORMAT statement and List is a list of quantities to be transmitted.

Purpose: This statement causes the quantities in the list to be typed on the on-line typewriter in accordance with FORMAT statement n.

Examples:
```
TYPE 3,A,B,C
TYPE 3,A,I(3)
```

GENERAL FORM: WRITE DRUM i,j,List where i and j are unsigned integer constants or variables, with the value of i between 1 and 8 inclusive, and List is a list of quantities to be transmitted.

Purpose: This statement causes quantities in core storage to be transmitted to drum i, location j, in accordance with the list.

Examples:
```
WRITE DRUM 2,1000,A,B,C,D(6)
WRITE DRUM K,J,A,B,C,D(6)
```

GENERAL FORM: WRITE OUTPUT TAPE i,n,List where i, depending on the system, is an unsigned integer constant or integer variable symbolic tape designation or is an actual tape unit, n is the statement number of a FORMAT statement, and List is a list of quantities for transmission.

Purpose: This statement causes the object program to write its output on tape for later off-line printing according to FORMAT statement n.

Examples:
```
WRITE OUTPUT TAPE 42,30,(A(J),J=1,10)
WRITE OUTPUT TAPE L,30,(A(J),J=1,10)
```

GENERAL FORM: WRITE TAPE i, List where i, depending on the system, is an unsigned integer constant or integer variable symbolic tape designation or is an actual tape unit.

Purpose: This statement causes the object program to write the quantities specified by the list on tape unit i in internal machine notation.

Examples:

```
WRITE TAPE 24,A,B,C,D
WRITE TAPE K,A,B,C,D
```

<u>PART V: SAMPLE PROBLEMS</u>

This section will present several FORTRAN problems, along with possible solutions to these problems.

As an aid to explaining the solutions of these problems, the technique of diagramming will be introduced and used here.

Diagramming, often called flow charting or block diagramming, is a technique of schematically showing the steps which the computer must take to produce the answers required by the problem. Diagramming thus shows the logic of the program.

Diagrams serve two very important purposes:

1. They offer an easy notation for analyzing the steps required in the solution of a problem.

2. They provide basic documentation in the form of a "map" of the program, so that someone unfamiliar with the program can easily determine what the program does and how it does it.

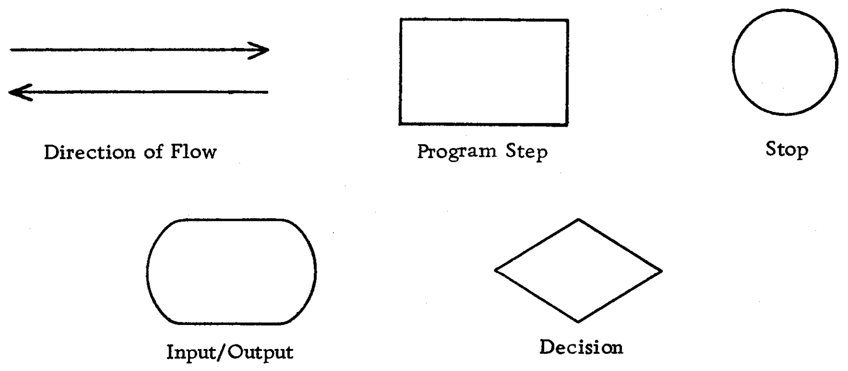
It is for the above reasons that diagramming is not only highly recommended, but is often required at data processing installations.

Techniques of diagramming vary greatly, as do the symbols used. In addition, detail in diagramming may range from very basic to the point where almost every single machine instruction is represented in the diagram.

The more complete the diagram, the easier is the job of actually writing the program; however, initial analysis of a problem can usually be noted only in major steps.

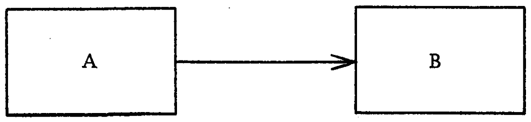
Since it will serve our purposes, only simple diagramming technique will be explained here. Further details of this important technique are available in the IBM reference manual <u>Flow Charting and Block Diagramming Techniques</u>, form C20-8008.

The symbols which will be used and which are explained below are:



The direction of flow simply shows the relationship of one symbol to another.

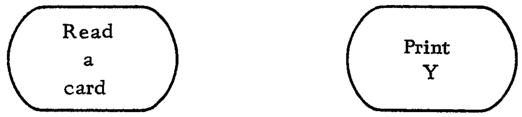
Example:



The above would show that A is executed and then B is executed.

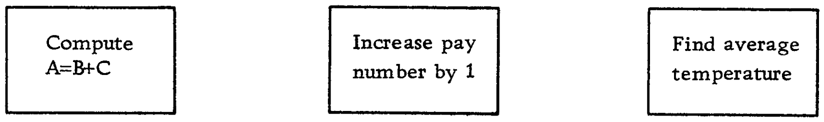
The input/output symbol is used to refer to any operation that involves an input/output device.

Examples:



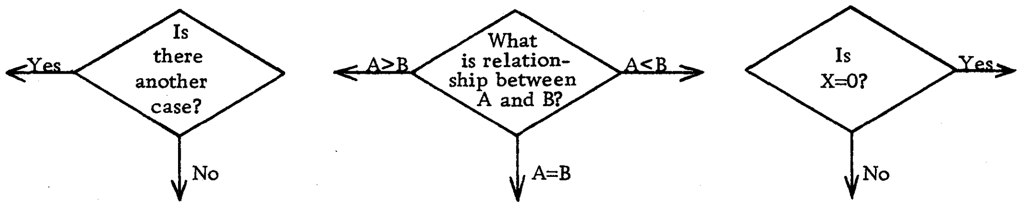
The program symbol is used to represent any steps in the program which are not represented by special symbols.

Examples:



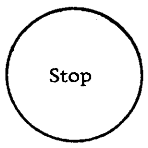
The decision symbol represents any logical decision that is contained in the program.

Examples:



The stop symbol is used to indicate the end of the program. If there are several ways to end the program, there may be several stop symbols.

Example:



The use of these symbols will become clearer by studying the diagrams used to explain the steps required to solve the sample problems.

In the problems below, statement numbers required by the logic of the program are either 1 or 2 digits (e.g., 1, 5, 18); statement numbers required for explanatory purposes only are 3 digits (e.g., 101, 782).

The problems below could be coded by any FORTRAN programmer; all formulas, etc., from specialized fields are given. The solutions presented are not the only possible solutions, nor necessarily the best and most efficient solutions. In addition, the problem solutions are not necessarily acceptable to all FORTRAN processors, although the basic statements are suitable. Where a solution may not be used in any given FORTRAN system, it is generally due to specialized rules required by that system for writing statement numbers or for input/output.

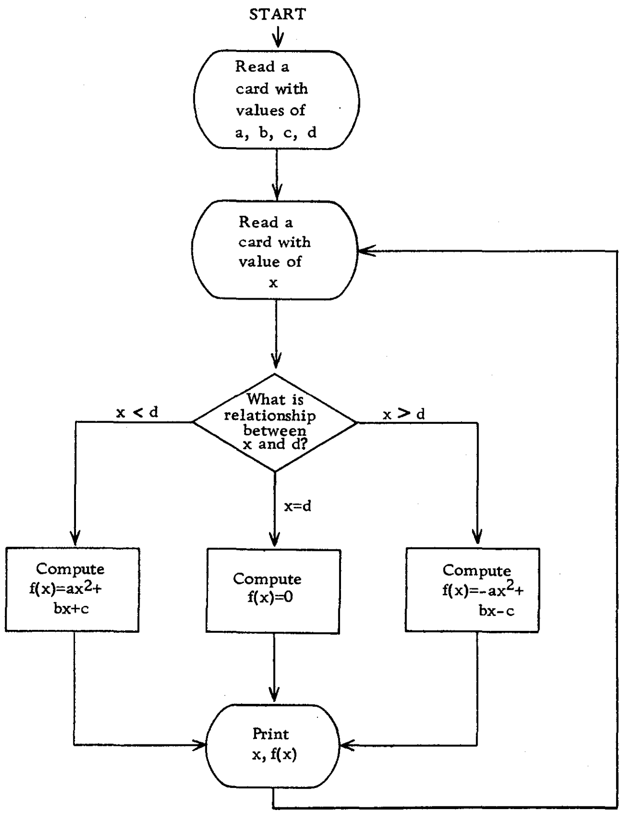
PROBLEM 1

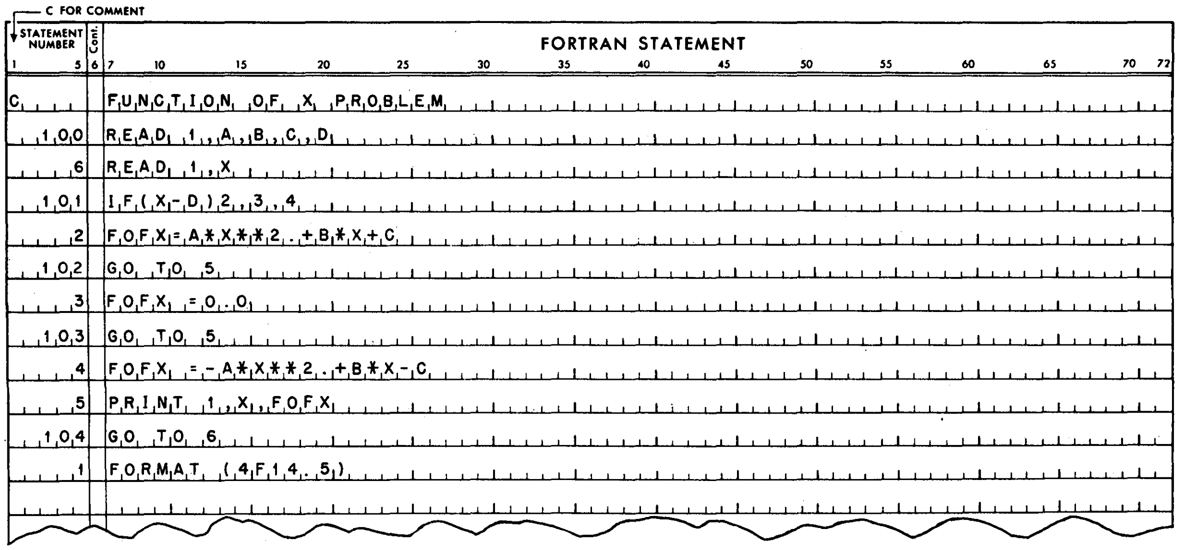
Given values for a, b, c, and d punched on a card, and a set of values for the variable x punched one per card, evaluate the function defined by

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

for each value of x, and print the value of x and f(x).

A possible FORTRAN program to solve this problem follows:


START
Read a card with values of a, b, c, d
Read a card with value of x
What is relationship between x and d?
x < d
x > d
x=d
Compute f(x)=ax²+ bx+c
Compute f(x)=0
Compute f(x)=-ax²+ bx-c
Print x, f(x)



```
C     FUNCTION OF X PROBLEM
  100 READ 1, A, B, C, D
    6 READ 1, X
  101 IF (X-D) 2, 3, 4
    2 FOFX= A*X**2. +B*X+C
  102 GO TO 5
    3 FOFX = 0.0
  103 GO TO 5
    4 FOFX = -A*X**2. +B*X-C
    5 PRINT 1, X, FOFX
  104 GO TO 6
    1 FORMAT (4F14.5)
```

Typical output, where A = 10.0, B = 11.0, C = 12.0, D = 13.0, might be:

```
  9.50000     1018.99996
 10.00000     1121.99988
 10.50000     1229.99985
 11.00000     1342.99986
 11.50000     1460.99986
 12.00000     1583.99988
 12.50000     1711.99988
 13.00000        0.
 13.50000    -1685.99977
 14.00000    -1817.99966
 14.50000    -1954.99973
 15.00000    -2096.99982
```

The first statement is a comment which will appear on source program listings.

Statement 100 causes the first card to be read and causes the values punched in that card to be assigned as the values of A, B, C, and D.

Statement 6 causes the next card to be read; it contains the first value of X to be used by the program.

Statement 101 determines the relationship between X and D and determines which formula to use in the computation of f(x). If X-D is negative (X< D), transfer is to statement 2; if X-D is zero (X=D), transfer is to statement 3; if X-D is positive (X >D), transfer is to statement 4.

Statements 2, 3, or 4 are used to determine the correct value of f(x), i.e., FOFX. Regardless of which computation occurs, control goes next to statement 5.

Statement 5 prints out the values of X and f(x).

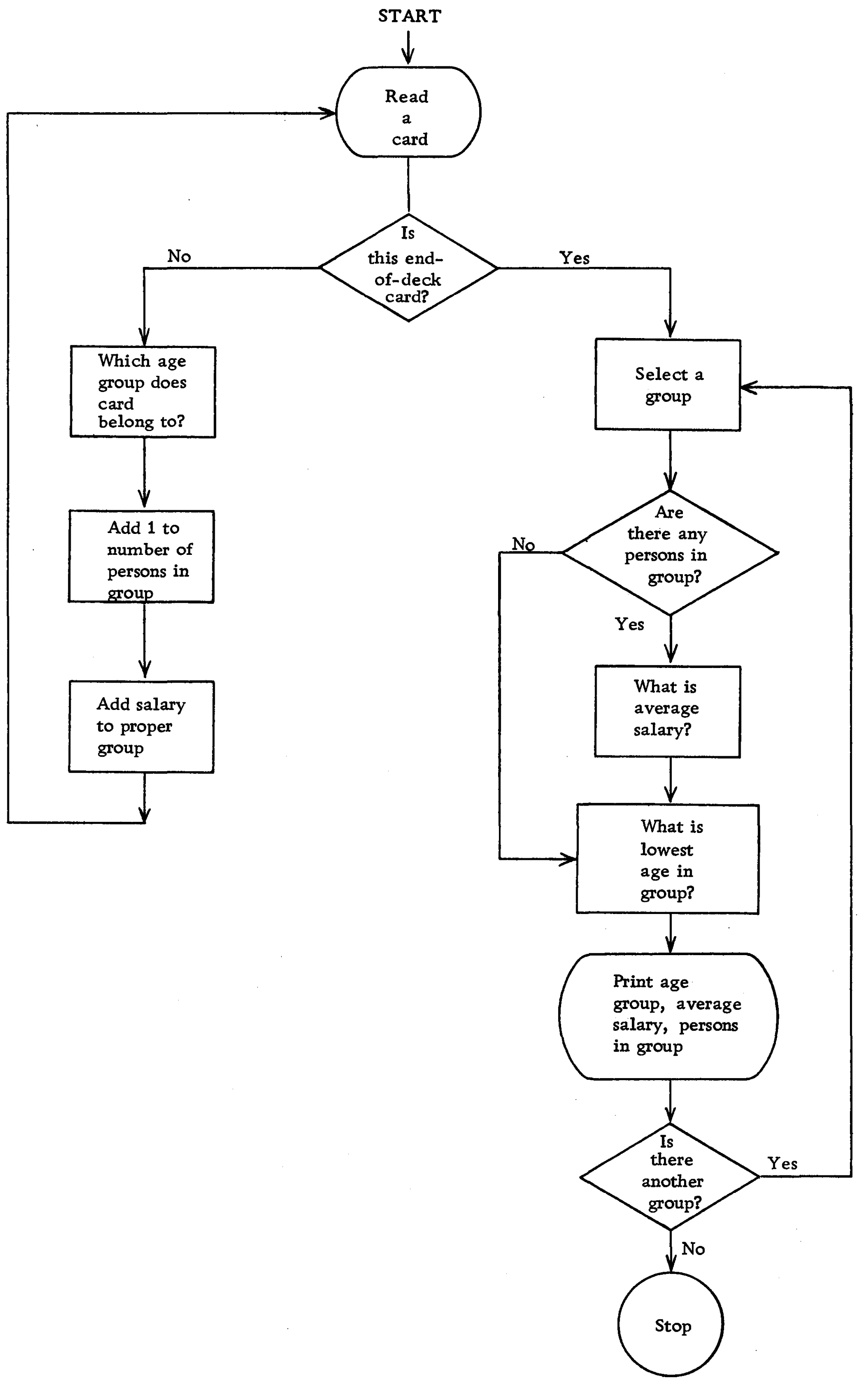
Statement 104 causes transfer to statement 6 to read in the next value of X, and the pattern continues until all of the X-cards have been processed.

The computer will automatically halt when it attempts to execute the READ with no more cards in the card reader.

PROBLEM 2

A deck of cards is given to be read in format (I 3, F 10. 2), one card for each person in a certain village. The number in the first field is the person's age; the number in the second is his income for 1960. Following this deck is a card with -1 in the first field; this information will be used to test for the end of deck.

Find the average salary of the people in each 5 yr. age group, i. e., 0-4, 5-9, 10-14, . . . , 95-99. Print out the lower age limit for each group, i. e., 0, 5, 10, . . . , 95, the average salary for that group, and the number of people in each group. Precautions should be taken to avoid division by zero in case there are no people in an age group.


START
Read a card
Is this end-of-deck card?
No
Yes
Which age group does card belong to?
Add 1 to number of persons in group
Add salary to proper group
Select a group
Are there any persons in group?
No
Yes
What is average salary?
What is lowest age in group?
Print age group, average salary, persons in group
Is there another group?
Yes
No
Stop

```
C     POPULATION AND SALARY PROBLEM
    3 FORMAT(I3,F10.2,F7.0)
  100 DIMENSION P(20),S(20)
  101 DO 9 N=1,20
  102 P(N)=0.0
    9 S(N)=0.0
    1 READ 3,K,SAL
  103 IF(K+1)8,4,2
    2 N=K/5+1
  104 P(N)=P(N)+1.0
  105 S(N)=S(N)+SAL
  106 GO TO 1
    4 DO 7 N=1,20
  107 IF(P(N))8,6,5
    5 S(N)=S(N)/P(N)
    6 K=5*N-5
    7 PRINT 3,K,S(N),P(N)
    8 STOP
```

The first statement is a comment card that will appear on program listings.

Statement number 3 is a format statement; notice that it provides for three fields and, thus, can provide for both the input and output requirements of this problem.

Statement 100 provides dimension information for the arrays P and S. P will be a count of the number of persons in each of the 20 age groups; S will be the total of the salaries earned by persons in each of the 20 age groups.

Statements 101, 102, and 9 are initialization statements. They set each of the elements in the arrays P and S equal to 0. Thus, if there is no entry in one of the elements of these arrays, that element will be equal to zero rather than equal to some quantity that may be present due to previous computations.

Statement 1 causes the first card to be read and assigns the quantity in the first three columns as the value of K and the quantity in the next 10 columns as the value of SAL.

Statement 103 is an end-of-deck test. If the end-of-deck card is read, K will have the value -1. The expression K+1 will then have the value 0 (-1+1) and transfer will occur to statement 4 which causes the final computation in the problem. If a valid card is read, K+1 will be positive, and the program will continue to statement 2. If for any reason K+1 is negative, there is an error and the program will stop.

Statement 2 determines which age group the card belongs to. Thus, if the age is 7, N will equal 2 (7/5+1, which will be truncated to 1+1); the card should be included in the group 5-9.

Statement 104 is a counter of the number of persons included in each member of the array.

Statement 105 is a summation of the salaries included in each member of the array.

Suppose, for example, that a card is read for a person 27 years old with salary of $5,000.00 per year. From statement 2, this would apply to the 6th member of the array. Thus statement 104 would increase P(6) by 1, and statement 105 would increase S(6) by $5,000.00.

Statement 106 causes transfer to statement 1, which will cause the next card to be read.

When the end-of-deck card is read, statement 103 will cause transfer to statement 4, which will complete the computations required by the program.

Statement 4 is a DO which controls the following statements with the exception of the STOP.

Statement 107 tests each member of the P array to determine if there are any persons in the group. If there are no persons in the group, statement 5 is skipped, and the program continues with statement 6; this avoids division by zero in statement 5. If there is a negative quantity in a group, there is an obvious error and the program stops. If there are persons in the group, the program continues with statement 5.

Statement 5 computes the average salary for each group by dividing the total salary for that group, S(N), by the persons in that group, P(N). By storing this average back in S(N), there is no need for a new variable to represent average salary.

Statement 6 determines the lowest age in each group and assigns this as the value of K. Thus, P(N), where N=2, refers to the age group starting at (5*2-5).

Statement 7 causes the following to be printed; the lowest age in each group, K; the average salary of that group, S(N); and the number of persons in that group, P(N). If the DO is not satisfied, the next group is computed. If the DO is satisfied, control passes to the next statement.

Statement 8 causes the computer to stop.

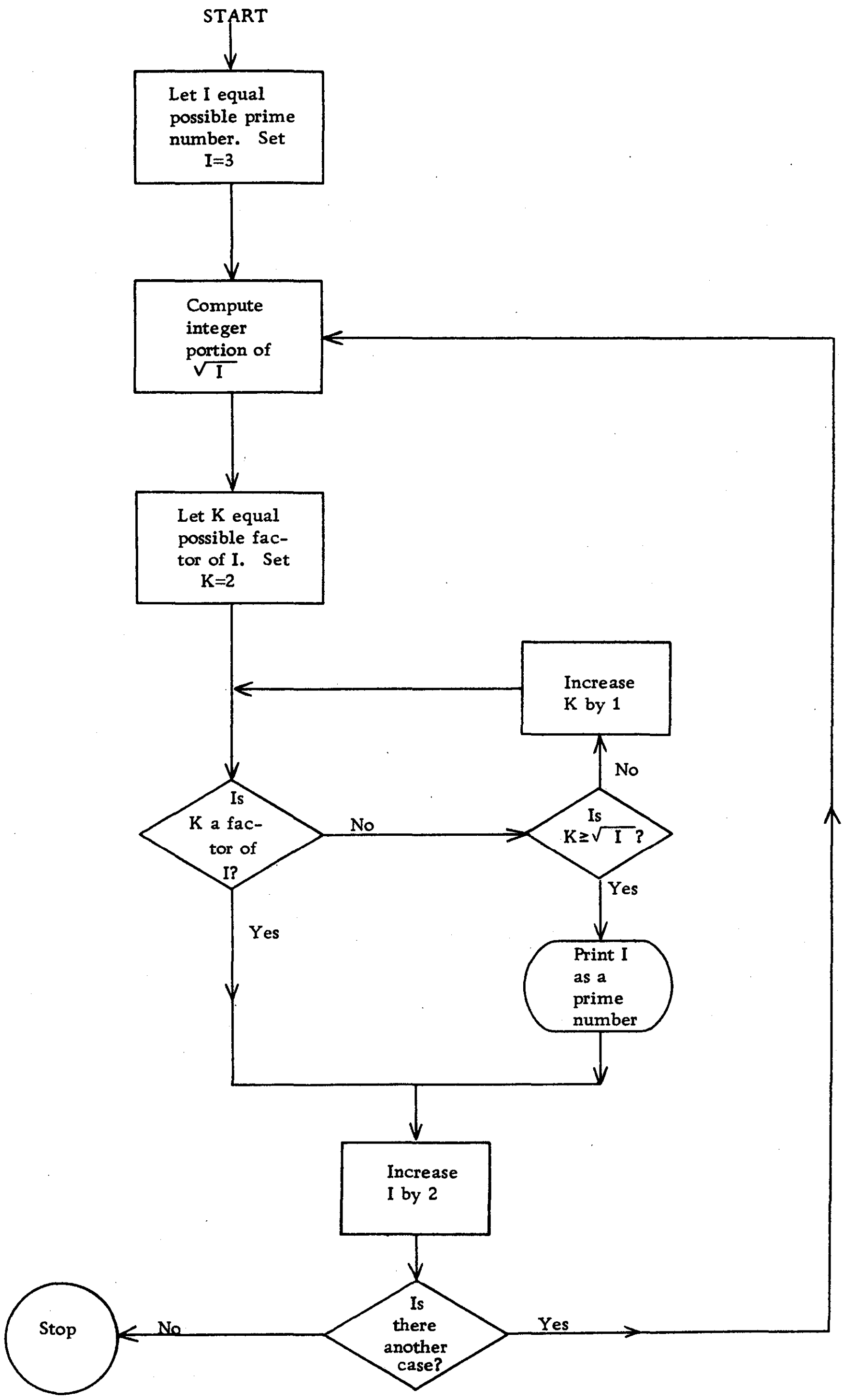
PROBLEM 3

Find all of the prime numbers between 1 and 1,000. A prime number is an integer that cannot be evenly divided by any integer except itself and 1. Thus, 1, 2, 3, 5, 7, 11,... are prime numbers. (9, for example, is not a prime since it can be evenly divided by 3.)

Since all even numbers have 2 as a factor, we can eliminate all even numbers from our list of primes, with the exception of the number 2 itself.

Further, it can be proven that if an odd number, X, is not prime, the number that can be divided into it will be in the range from 3 to the integer portion of the square root of X.

A program to solve the problem is shown below:


START
Let I equal possible prime number. Set I=3
Compute integer portion of $\sqrt{I}$
Let K equal possible factor of I. Set K=2
Increase K by 1
No
Is K a factor of I?
No
Is $K \geq \sqrt{I}$?
Yes
Yes
Print I as a prime number
Increase I by 2
Stop
No
Is there another case?
Yes

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–72)
C		PRIME NUMBER PROBLEM
100		PRINT 8
8		FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
	1	19X,1H1/19X,1H2)
101		I=3
3		A=I
102		A=SQRTF(A)
103		J=A
104		DO 1 K=2,J
105		L=I/K
106		IF(L*K-I)1,2,4
1		CONTINUE
107		PRINT 5,I
5		FORMAT (I20)
2		I=I+2
108		IF(1000-I)7,4,3
4		PRINT 9
9		FORMAT (14H PROGRAM ERROR)
7		PRINT6
6		FORMAT(31H THIS IS THE END OF THE PROGRAM)
109		STOP

The first statement is a comment that will appear on listings only.

Statement number 100 will cause a heading to be printed by the object program and will print the first two prime numbers (which are not computed by the program), as follows:

```
FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000.
                   1
                   2
```

Notice that the statement following statement 8 is a continuation card and thus may not be given a statement number even for reference purposes.

Statement 101 establishes an index, I, and sets its initial value equal to 3. I represents the number being examined as a possible prime.

Statement 3 sets up a floating point variable equivalent to the integer quantity in the index I. Thus, A has the value 3.0. The purpose of this statement is to permit use of the expression in statement 102.

Statement 102 uses a function, SQRTF, to find the square root of a floating point quantity. Thus, A no longer has the value 3.0, but has the value of the positive square root of that number.

Statement 103 converts the quantity in A to an integer quantity. Thus, we have determined the integer portion of the square root of the number that we are examining as a possible prime number (i.e., the square root of I).

Statement 104 is a DO; its index is K. Thus K represents the possible factors of I. For each possible prime number, this DO will permit execution of the statements to and including statement 1. The first time through the DO, K will have the value 2. Thereafter, K will be increased by 1 until it is about to exceed J. Notice, when we are examining 3 as a possible prime number, J will be equal to the integer portion of the square root of 3, i.e., J will be equal to 1. The statement we have written is thus equivalent to:

```
104  DO 1 K=2,1
```

This illustrates the fact that a DO will always be executed the first time through the loop, although the index (K=2) already exceeds the test value (J=1).

Statement 105 causes I to be divided by K. The first time through, this will be equivalent to 3/2 or L=1 (since the result will be truncated). If truncation occurs, we know that K is not a factor of I since it cannot evenly be divided into I. To determine whether truncation occurred, proceed to the next statement.

Statement 106 tests for truncation by multiplication. If truncation did not occur, L*K-I will be equal to zero. If truncation occurred, L*K-I will be less than zero. The first time through, I is equal to 3, K is equal to 2, and L is equal to 1. L*K-I is equal to -1; thus truncation occurred and K is not a factor. Consider a later case where 9 is being examined as a possible prime number. I would be equal to 9; when K is equal to 3, L would be equal to 3. L*K-I would be equal to 0, and thus K(i.e., 3) would be a factor of 9. If for any reason L*K-I is greater than zero, it is due to a program error and transfer occurs to statement 4, which is an error routine.

If K is a factor of I (truncation does not occur), transfer is out of the DO to statement 2 for examination of the next possible prime. If K is not a factor of I (truncation occurs), transfer is to statement 1.

Statement 1 is the last statement in the range of the DO. It is necessary because the DO would otherwise end with an IF statement, which is not permitted. If a factor has been found and the DO is not satisfied, K is increased by 1, and the DO continues. If a factor has not been found and the DO is satisfied, transfer occurs to statement 107.

Statement 107 prints I as a prime, since no factor has been found.

Statement 2 increases I by 2, since only odd numbers are being tested as possible prime numbers.

Statement 108 determines whether all the numbers in the range 1 to 1000 have been examined. If they have been examined, transfer occurs to statement 7, which prints an end-of-job message. If all the numbers in the range have not been examined, transfer is to statement 3 to examine the next number.

Statements 4 and 9 are an error routine.

Statements 7 and 6 are an end-of-job routine.

Statement 109 ends the program by stopping the computer.

PROBLEM 4

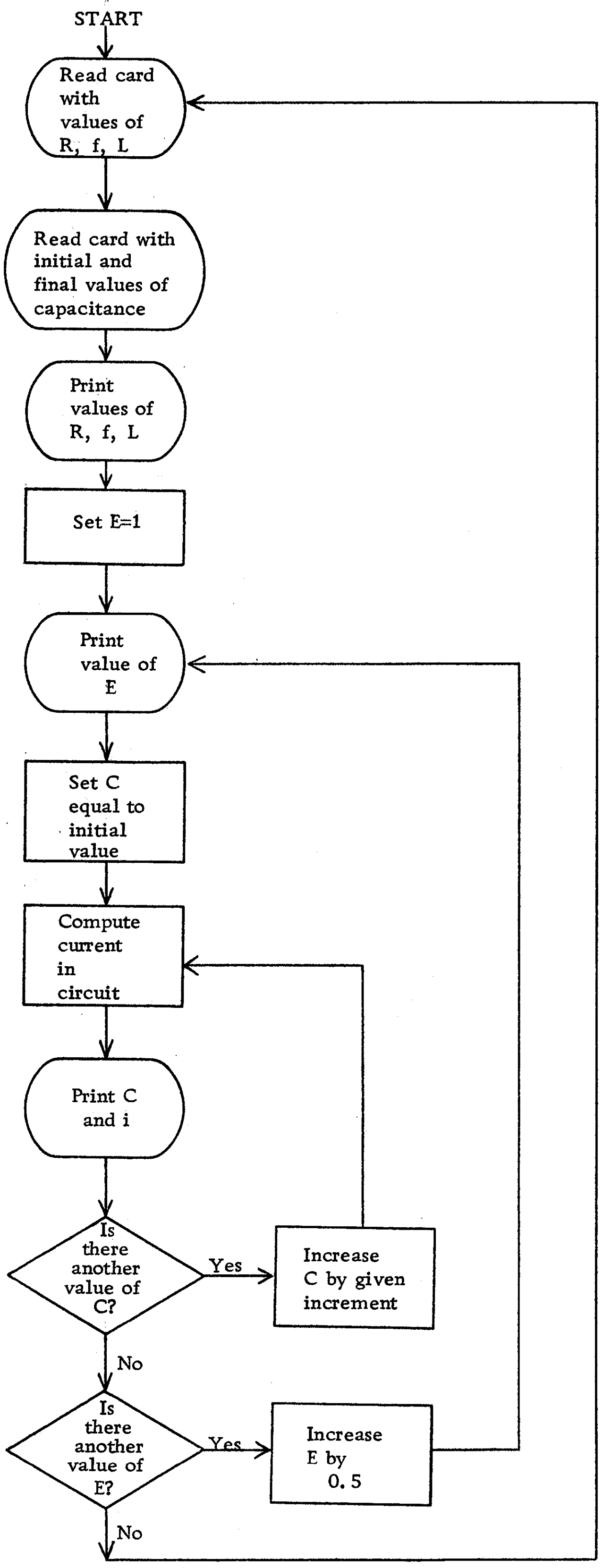
Determine the current in an alternating current circuit consisting of resistance, inductance, and capacitance in series, given a number of sets of values of resistance, inductance, and frequency. The current is to be determined for a number of equally spaced values of the capacitance (which lie between 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

A possible FORTRAN program to solve this problem follows:


START
Read card with values of R, f, L
Read card with initial and final values of capacitance
Print values of R, f, L
Set E=1
Print value of E
Set C equal to initial value
Compute current in circuit
Print C and i
Is there another value of C?
Yes
Increase C by given increment
No
Is there another value of E?
Yes
Increase E by 0.5
No

```
C FOR COMMENT
STATEMENT NUMBER | Cont. | FORTRAN STATEMENT
1     5 6 7   10   15   20   25   30   35   40   45   50   55   60   65   70  72
C     DETERMINATION OF CURRENT IN AC CIRCUIT
    7 READ 1, OHM, FREQ, HENRY
  100 READ 2, FRD1, FRDFIN
  101 PRINT 1, OHM, FREQ, HENRY
  102 VOLT = 1.0
    9 PRINT 1, VOLT
  103 FARAD =FRD1
    5 AMP = VOLT/SQRTF(OHM**2 +(6.2832*FREQ*HENRY
     1              -1./(6.2832*FREQ*FARAD))**2)
  104 PRINT 2,FARAD,AMP
  105 IF(FARAD-FRDFIN)3,4,4
    3 FARAD =FARAD +0.000 000 01
  106 GO TO 5
    4 IF(VOLT-3.0) 6,7,7
    6 VOLT = VOLT+0.5
  107 GO TO   9
    1 FORMAT (3F14.5)
    2 FORMAT (2E14.5)
```

The first statement is a comment that will appear when the source program is listed.

Statement number 7 causes the values of the resistance, the frequency, and the inductance to be read, in that order, from the first card.

Statement 100 causes the initial and final values of the capacitance to be read from the next card.

Statement 101 causes the values of the resistance, frequency, and inductance to be printed.

Statement 102 sets the initial value of the voltage.

Statement 9 prints the initial value of the voltage.

Statement 103 sets the current value of the capacitance (FARAD), equal to the first value of the capacitance to be used in calculation (FRD1).

Statement 5 specifies the actual calculation of the current in the circuit.

Statement 104 prints the current value of the capacitance and the result of the calculation in statement 5.

Statement 105 compares the current value of the capacitance with the final value to determine whether or not all values have been investigated. If all values have been investigated (the expression is zero or positive), transfer is to statement 4. If all values have not been investigated, transfer is to statement 3.

Statement 3 causes the current value of the capacitance to be increased by the given increment.

Statement 106 is a transfer to statement 5, which causes the calculation to be repeated for the new value of the capacitance. This pattern is repeated until all values of the capacitance have been investigated and control is in statement 4.

Statement 4 compares the value of the voltage with the upper bound to determine whether or not all specified values of the voltage have been used. If all the values of the voltage have been used, transfer is to statement 7 to begin the next case. If all values of the voltage have not been used, transfer is to statement 6.

Statement 6 causes the value of the voltage to be increased.

Statement 107 causes transfer to statement 9, which prints the new value of the voltage and continues the calculations with the new data.

The computer will halt when it attempts to execute statement 7 with no more cards in the card reader.

APPENDIX A: LIST OF FORTRAN PUBLICATIONS

650 FORTRAN — Automatic Coding System for the IBM 650 Data Processing System, Form C29-4047
FOR TRANSIT Automatic Coding System for the IBM 650 Data Processing System, Form C28-4028

IBM 1620 FORTRAN Specifications, Form J28-5598-0
IBM 1620 GOTRAN: Preliminary Specifications, Form J29-4205
IBM 1620 GOTRAN for Card Input/Output, Form J28-5557

704 FORTRAN Programming System, Form C28-6106
704 FORTRAN Operations, Form C28-6097

705 FORTRAN Programming System, Form J28-6122

7070 Basic FORTRAN, Form C28-6099
IBM 7070 FORTRAN, Form J28-6045

709/7090 FORTRAN Programming System, Form C28-6054-2
32K 709/7090 FORTRAN: Double-Precision and Complex Arithmetic, Form J28-6114-1
32K 709/7090 FORTRAN: Source Language Debugging at Object Time, Form J28-6133
32K 709/7090 FORTRAN: Adding Built-In Functions, Form J28-6135
FORTRAN Assembly Program (FAP) for the IBM 709/7090, Form J28-6098-1
709/7090 FORTRAN Operations, Form C28-6066-3

Advance Specifications: 7090 FORTRAN and FORTRAN Assembly Program (FAP), Form J28-6132

APPENDIX B: ADMISSIBLE CHARACTERS IN A FORTRAN SOURCE PROGRAM

The following chart indicates the list of characters which may be used in a FORTRAN source program.

Character	Card Code	Character	Card Code
Blank		M	11-4
.	12-3-8	N	11-5
)	12-4-8	O	11-6
+	12	P	11-7
$	11-3-8	Q	11-8
*	11-4-8	R	11-9
-	11	S	0-2
/	0-1	T	0-3
,	0-3-8	U	0-4
(	0-4-8	V	0-5
=	3-8	W	0-6
A	12-1	X	0-7
B	12-2	Y	0-8
C	12-3	Z	0-9
D	12-4	0	0
E	12-5	1	1
F	12-6	2	2
G	12-7	3	3
H	12-8	4	4
I	12-9	5	5
J	11-1	6	6
K	11-2	7	7
L	11-3	8	8
		9	9

NOTE: The character $ can be used in FORTRAN only as alphameric text in a FORMAT statement.

FORTRAN SPECIAL CHARACTERS

Only the special characters shown in the above chart are meaningful to the FORTRAN processor. They are always identified by their card codes; e.g., a column punched 12-8-4 will always be interpreted as a ")". However, peripheral equipment is available with a choice of special character sets. Depending on the set of characters supplied with a printer or a card punch, a column punched 12-8-4 may be equated to a "⌑" or a "<" or a ")". The various sets of characters are known as "Type Wheel Configurations A, B, C, D, etc." Type Wheel Configuration F is the special character set used by FORTRAN.

When punching a source program, the character must always be punched according to the card code. For example, when punching ")", the card must always be punched 12-8-4, regardless of whether a given printer interprets 12-8-4 as ")" or not.

The special characters which are used in FORTRAN are given below with their equivalents in other type wheel configurations.

SPECIAL CHARACTERS USED IN FORTRAN	IBM CARD CODE	TYPE WHEEL CONFIGURATION								
		A	B	C	D	E	F	G	H	K
+	12	&	/	&	-	-	+	+	+	+
.	12-8-3	.	.	.	.	.	.	.	.	.
)	12-8-4	⌑	⌑	⌑	⌑	<	)	⌑	)	)
-	11	-	-	-	-	/	-	-	-	-
$	11-8-3	$	$	$	$	°	$	$	$	$
*	11-8-4	*	*	*	*	*	*	*	*	*
/	0-1	/	&	0	/	&	/	/	/	/
,	0-8-3	,	,	,	,	,	,	,	,	,
(	0-8-4	%	%	%	%	%	(	%	(	(
=	8-3	#	#	#	#	#	=	+	=	=

↑

[illegible]

APPENDIX C: ANSWERS TO PROBLEMS

CHAPTER 2 (Page 18)

• 1. Special characters not permitted in variable names. 2. A floating point variable is not permitted in a subscript. 3. Two decimal points not permitted. 4. Valid integer variable. 5. Valid floating point variable. 6. Valid integer constant. 7. Variable in subscript may not be signed. 8. Decimal point not permitted in exponent. 9. Valid subscripted floating point variable. 10. Valid integer variable. 11. Subscripts not in proper format. 12. Valid floating point constant. 13. Valid subscripted integer variable. 14. A floating point variable is not permitted in a subscript. 15. Valid floating point variable. 16. A variable may not start with a digit. 17. Decimal point required. 18. Subscript not in proper format. 19. Valid floating point constant. 20. Subscript not in proper format.

CHAPTER 3 (Page 22)

1. Yes. 2. By the rules for forming expressions. 3. As subscripts, as exponents and as arguments. 4. Parentheses. 5a. L=1; 5b. M=2; 5c. N=-2; 5d. D=3.; 5e. E=2.; 5f. F=-2.; 5g. A=1.; 5h. M=1; 5i. I=6; 5j. C=2.; 6a. Valid; 6b. Valid; 6c. Integer quantity may not have a floating point exponent; 6d. A floating point quantity cannot be a subscript; 6e. An expression may not appear on the left; 6f. Valid; 6g. Valid; 6h. Valid; 6i. A floating point quantity may not be a subscript; 6j. The expression is mixed.

CHAPTER 4 (Page 33)

1a. FEW is a floating point variable; 1b. m_2 may not be an expression; 1c. M is a variable; 1d. A is a floating point variable; 1e. Valid; 1f. The expression is mixed; 1g. ,4 is not permitted; 1h. Comma not permitted before J; 1i. K must be preceded by a comma; 1j. N-12 is not permitted — it is an expression.

2a.
```
         DIMENSION IDIST(10),IRATE(10),ITIME(10)
         DO 12 I=1,10
   12    IDIST(I)=IRATE(I)*ITIME(I)
         STOP 777
```

2b.
```
         DIMENSION IDIST(10),IRATE(10),ITIME(10)
         DO 12 I=1,10
         IDIST(I)=IRATE(I)*ITIME(I)
         IF (1000-IDIST(I)) 5,12,12
   12    CONTINUE
         .
         .
         .
    5    .
         .
         .
         .
         STOP 777
```

2c.
```
          Y=X
      3   X=X-1.
          IF (X-0.) 4,4,5
      5   Y=Y+X
          GO TO 3
      4   STOP
```

2d.
```
          DIMENSION PAY(500), BASE(500), OVTM(500), TAX(500),
                                                    OTHRD(500)
          DO 2 J=1,I
      2   PAY(J)=BASE(J)+OVTM(J)-TAX(J)-OTHRD(J)
          STOP
```

2e.
```
          DIMENSION A(25,2)
          DO 5 I=1,25
          A(I,1)=I
      5   A(I,2)=1.5*A(I,1)
          STOP
```

● 2f.
```
          DIMENSION X(20),RCIPX(20)
          SUMX=0.
          DO 5 I=1,20
          IF(X(I)) 3,2,3
      2   RCIPX(I)=0.
          GO TO 5
      3   RCIPX(I)=1./X(I)
      5   SUMX=SUMX+RCIPX(I)
          STOP
```

CHAPTER 5 (Page 39)

1.
```
          DIMENSION A(10,10)
          READ 1,A
```

2.
```
          READ 1, (A(I),I=1,5),BJOB,NEXT,DELTA(2),
                  (E(I),I=3,11,2)
```

● 3.
```
          PUNCH 1, F(2,2), (G(I,4),I=1,3)
```

CHAPTER 6 (Page 52)

1a. F7.3; 1b. F7.3; 1c. I3; 1d. E11.4; 1e. E11.4; 1f. E10.3; 1g. E10.3; 1h. I4; 1i. F5.2; 1j. F5.2.

2.
```
          PRINT 7
      7   FORMAT (31HTHEbFOLLOWINGbAREbPAYROLLbCARDS)
```

3a. Valid; 3b. Valid; 3c. Valid; 3d. No FORMAT statement number or comma preceding A; 3e. Valid; 3f. Valid; 3g. Constants are not permitted in a List; 3h. Valid; 3i. Should be 14H; 3j. Exceeds unit record size (120 print positions).

GLOSSARY

The following terms are defined only as they relate to this manual. No attempt has been made to resolve slight inconsistencies which exist between the terms as they are used in FORTRAN and as they are used in the general field of computer programming.

ABSOLUTE CODING	Coding written in machine language. It does not require processing before it can be understood by the computer.
ACCUMULATOR	A part of the logical-arithmetic unit of a computer. It may be used for intermediate storage, to form algebraic sums, or for other logical-arithmetic operations.
ADDRESS	A label, name or number identifying a register, location, or unit where information is stored in the computer.
ALPHAMERIC CHARACTERS	A generic term for numeric digits, alphabetic characters, and special characters.
ALTERATION SWITCH	A switch on the console of a computer which may be set to ON or OFF. Statements may be included in a program to test the condition of these switches and to vary program execution based on these settings.
● ANALOG COMPUTER	A computer which produces output based upon measured input. Input is of a continuous nature such as the elevation of a lever or the temperature of a solution as distinct from a digital computer which accepts numbers, etc., which are discrete.
ARGUMENT	A variable upon whose value the value of a function depends. The arguments of a function are listed in parentheses after the function name, whenever that function is used. The computations specified by the function definition occur using the variables specified as arguments.
ARITHMETIC STATEMENT	A type of FORTRAN statement which specifies a numerical computation.
ARRAY	A series of items, not necessarily arranged in a meaningful pattern.
BACKSPACE TAPE	The operation of returning a magnetic tape to the beginning of the preceding record on that tape.

BCD (Binary Coded Decimal)	A system of representing numerical, alphabetic and special characters by means of binary notation.
BINARY DIGIT	Either of the digits 0 or 1 which may be used to represent the binary conditions ON or OFF.
BLANK	Either the absence of any information or the specific character which represents the presence of no information.
CARD FIELD	A fixed number of consecutive card columns assigned to a unit of information; e.g., card columns 15-20 can be assigned to an identification number.
CARD PUNCH	A machine which is used to punch holes in cards in order to record information.
CLOSED SUBROUTINE	A subroutine which is not stored in the normal sequence of the program. Instead, transfer is made from the program to the area where the subroutine is stored, and then, after the subroutine is executed, control is returned to the main program.
CODING	Writing instructions for a computer either in machine or non-machine language.
CODING SHEET	A form upon which computer instructions are written prior to being punched into IBM cards.
COMPATIBILITY	The quality of an instruction to be translatable or executable on more than one class of computer.
COMPUTER	A device for operating upon data so as to produce desired and meaningful results.
CONSOLE	The unit of a computer where the control keys and certain special devices are located. This unit may contain the Start key, Stop key, Power key, Sense Switches, etc., as well as lights which display the information located in certain registers.
CONTINUATION CARD	A card which follows a special format and which is used to permit a single statement to be written on more than one IBM card.
CONSTANT	A quantity that does not change either from one execution of a program to another, or during execution of that program; a number that remains fixed.

CONTROL STATEMENTS	A statement which is used to direct the flow of the program, either causing specific transfers or making transfers dependent upon meeting certain specified conditions.
DATA PROCESSING	Any procedure for receiving information and producing a specific result.
DEBUGGING	Process of locating errors in a program and correcting them.
DIGITAL COMPUTER	A computer which accepts alphameric information which it uses to produce results.
DIVIDE CHECK	An indicator which denotes that a division has been attempted or has occurred which is invalid.
DRUM	A device for storage of information in a computer.
END OF FILE	An indication that all records in the file have been read or written.
EXECUTION	The operation of computer under the direction of the object program.
EXPRESSION	A valid series of constants, variables, and functions which may be connected by operation symbols and punctuated, if required, to cause a desired computation.
FIXED POINT	A type of calculation with integers only and without any decimal point or decimal portions.
FLOATING POINT	A form of calculation where quantities are represented by a decimal number times a power of 10. Accuracy is to several decimal digits.
FLOW CHART	A group of diagramming techniques which are used to indicate the procedure for the solution of a problem.
FORMAT	The arrangement of information for input to a computer or the arrangement desired for output of information.
FORTRAN	A programming system, including a language and a processor, allowing programs to be written in a mathematical-type language. These programs are subsequently translated by a computer (under control of the processor) into machine language.

FUNCTION	A means of referring to a type or sequence of calculations within an arithmetic statement.
IBM CARD	A type of paper card which may have information recorded on it by means of punched holes and which may be read by a computer.
ILLEGAL CHARACTERS	Any character which is not part of the FORTRAN character set.
INPUT/OUTPUT	The process of transmitting information from an external source to the computer or from the computer to an external source.
LIBRARY	A group of standard, proven routines which may be incorporated into larger routines.
LIST	A string of items, written in a meaningful format, which designate quantities to be transmitted for input/output.
LOCATION	A unit of storage which may be identified by an address and in which information may be stored.
LOGICAL DECISION	The operation of selecting alternative paths of flow depending on intermediate program data.
MACHINE LANGUAGE	Information recorded in a form which may be used by a computer without prior translation.
MAGNETIC TAPE	A tape which has an oxide coating to permit the recording of information magnetically.
MAGNITUDE	The size of a quantity as distinct from its sign. Thus, +10 and -10 have the same magnitude.
MEMORY	An alternative term for storage.
MNEMONIC OPERATION CODES	Computer instructions written in a meaningful notation; e.g., ADD, MPY, STO.
MODE	The characteristic of a quantity being suitable for integer or for floating point computation.
MULTIPLIER-QUOTIENT	A register of a computer which is used for operations involving multiplication and division.
OBJECT MACHINE	The computer on which the object program is to be executed.
OBJECT PROGRAM	The machine language program which is the final output of a coding system.

OFF-LINE UNIT	Input/output device or auxiliary equipment not under direct control of the central processing unit.
ON-LINE UNIT	Input/output device or auxiliary equipment under direct control of the computer.
OPEN SUBROUTINE	A separately coded sequence of instructions which is inserted in another instruction sequence directly in the line of flow.
OPERATION CODE	The symbols which designate a basic computer operation to be performed.
OPERATORS	The characters which designate mathematical operations, such as +, -, etc.
OVERFLOW	The exceeding of the allowed capacity of a register.
PARAMETER	A constant or independent variable which is not stated as an argument and upon which an operation depends.
PRINTER	A device that prints output data.
PROCESSOR	A machine language program which performs the functions necessary to convert a source program into the desired object program.
PROGRAM	The plan for the solution of a problem, including the instructions which will cause a computer to perform the desired operation, and such required information as data descriptions and tables.
PROGRAMMING SYSTEM	Any method of programming problems, other than machine language, consisting of a Language and its associated processor(s).
PUNCHED PAPER TAPE	A tape on which information is recorded by punched holes.
QUANTITY	A constant, variable, function name, or expression.
READ	To transmit information from an input device to a computer.
REGISTER	A specialized storage device where data may be operated upon.
REWIND TAPE UNIT	The process of returning a magnetic tape to its beginning.

SCALE FACTOR	A method of modifying the location of a decimal point.
SENSE LIGHT	A light which may be turned on or off and may be interrogated by the computer to cause a program branch.
SENSE SWITCH	A switch on the console of a computer which may be set UP or DOWN. Statements may be included in a program to test the condition of these switches and to vary program execution based on these settings.
SOURCE LANGUAGE	The language in which the input to the FORTRAN processor is written.
SOURCE MACHINE	The computer on which the source program is translated into the object program.
SOURCE PROGRAM	A program coded in other than machine language which must be translated into machine language before use.
STATEMENT	An instruction (written in the FORTRAN language) to the computer to perform some sequence of operations.
STATEMENT NUMBER	A number which is associated with a single FORTRAN statement so that reference may be made to that statement in terms of its number.
STORAGE	A device in which data and instructions may be stored.
STORE	To place information in a location in storage so that it may be retrieved for later use.
STORED PROGRAM COMPUTER	A computer in which the instructions to be executed are stored in memory.
SUBPROGRAM	A part of a larger program which can be compiled independently.
SUBROUTINE	A program which defines desired operations and which may be included in another program to cause those desired operations.
SUBSCRIPT	A notation used to specify a particular member of an array where each member is referenced only in terms of the array name.
SUBSCRIPTED VARIABLE	A variable followed by one or more subscripts enclosed in parentheses.

SYMBOLIC CODING	Writing programs in any language other than absolute machine language.
TAPE UNIT	The mechanism upon which a magnetic tape is mounted for reading or writing.
TRANSFER	To terminate one sequence of instructions and begin another sequence.
TRANSFER INSTRUCTION	Any instruction which causes a transfer, whether conditional or not.
UNIT RECORD	A printed line with a maximum of 120 characters; a punched card with a maximum of 72 characters; a BCD tape record with a maximum of 120 characters.
VARIABLE	A symbol whose numeric value changes from one iteration of a program to the next or changes within each iteration of a program.
WRITE	To transfer information, usually from main storage to an output device.

INDEX

F28-8074-1

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York

F28-8074-1

IBM

International Business Machines Corporation

Data Processing Division

112 East Post Road, White Plains, New York