DOCUMENT RESUME

ED 143 510 SE 022 987

AUTHOR Charp, Sylvia; And Others
TITLE Algorithms, Computation and Mathematics (Fortran Supplement). Student Text. Revised Edition.
INSTITUTION Stanford Univ., Calif. School Mathematics Study Group.
SPONS AGENCY National Science Foundation, Washington, D.C.
PUB DATE 66
NOTE 137p.; For related documents, see SE 022 983-988; Not available in hard copy due to marginal legibility of original document

EDRS PRICE MF-$0.83 Plus Postage. HC Not Available from EDRS.
DESCRIPTORS Algorithms; *Computers; *Instructional Materials; *Programing Languages; Secondary Education; *Secondary School Mathematics; *Textbooks
IDENTIFIERS *FORTRAN; *School Mathematics Study Group

ABSTRACT

This is the student's textbook for Algorithms, Computation, and Mathematics (Fortran Supplement). This computer language supplement is split off from the main text to enable a school to choose the computer language desired, and also to make it easier to modify the course as languages change. The chapters in the text are designed to add language capability. Each can be read in conjunction with the main text, section by section. (RH)

ALGORITHMS, COMPUTATION AND MATHEMATICS

(Fortran Supplement)

Student Text

Revised Edition

The following is a list of all those who participated in the preparation of this volume:

Sylvia Charp, Dobbins Technical High School, Philadelphia, Pennsylvania
Alexandra Forsythe, Gunn High School, Palo Alto, California
Bernard A. Galler, University of Michigan, Ann Arbor, Michigan
John G. Herriot, Stanford University, California
Walter Hoffmann, Wayne State University, Detroit, Michigan
Thomas E. Hull, University of Toronto, Toronto, Ontario, Canada
Thomas A. Keenan, University of Rochester, Rochester, New York
Robert E. Monroe, Wayne State University, Detroit, Michigan
Silvio O. Navarro, University of Kentucky, Lexington, Kentucky
Elliott I. Organick, University of Houston, Houston, Texas
Jesse Peckenham, Oakland Unified School District, Oakland, California
George A. Robinson, Argonne National Laboratory, Argonne, Illinois
Phillip M. Sherman, Bell Telephone Laboratories, Murray Hill, New Jersey
Robert E. Smith, Control Data Corporation, St. Paul, Minnesota
Warren Stenberg, University of Minnesota, Minneapolis, Minnesota
Harley Tillitt, U. S. Naval Ordnance Test Station, China Lake, California
Lyneve Waldrop, Newton South High School, Newton, Massachusetts

The following were the principal consultants:

George E. Forsythe, Stanford University, California
Bernard A. Galler, University of Michigan, Ann Arbor, Michigan
Wallace Givens, Argonne National Laboratory, Argonne, Illinois

Financial support for the School Mathematics Study Group has been provided by the National Science Foundation.

TABLE OF CONTENTS

Chapter F2

INPUT-OUTPUT AND ASSIGNMENT STATEMENTS

F2-1 Introduction

In Chapter 2 we developed an appreciation of input, output and assignment steps as components of algorithms expressed in the form of flow charts. So far, we have viewed flow charts as a means for conveying a sequence of computation rules primarily from one person to another. We have tacitly assumed that only man can read, understand and carry out the intent of such flow charts. Naturally we want to include computers in the set of all things which can read, understand and carry out procedures.

FORTRAN II--language and processor

Programming languages like ALGOL and FORTRAN accomplish this objective. The steps of a programming language are called statements. They correspond roughly to the boxes of a flow chart.

FORTRAN, meaning FORmula TRANslation, is an English-like programming language developed by IBM about ten years ago. It was historically the first of a series of similar languages. FORTRAN II is an improvement over the original version and is probably the most widely used of available programming languages. More advanced versions are also in common use. They are known by various names, the most widely used name being FORTRAN IV.

FORTRAN II was designed with these objectives.

1. A wide variety of algorithms can be described with this language. Its chief area of application is for expressing algorithms which deal with scientific and engineering computation. Algorithms for many other types of problems can also be expressed satisfactorily in FORTRAN II.

2. The rules or "grammar" of the language are defined precisely (unlike English). The net result is that an algorithm written in FORTRAN II can mean precisely the same thing to each person who reads it, i.e., it means the same thing everywhere to people who have learned this language. Hence, FORTRAN II provides a means for communicating algorithms via correspondence and publications, from one person to another.

3. With minor additions or modifications, FORTRAN II can be "implemented" on many types of digital computers. By implementation we mean that a computer can be programmed to accept algorithms as "source" programs that are written in a version of FORTRAN II and automatically convert them to sequences of computer instructions, often called "target" programs, which can then be executed by the computer.

These features of FORTRAN II have led to its wide acceptance among mathematicians and scientists. In this chapter, we shall begin to describe FORTRAN II as it is typically implemented for use on a computer, calling it simply FORTRAN.

Each statement in FORTRAN is written so that when typed or punched on a card it can be transferred to the computer's memory. Here it can be scanned character by character† and analyzed for its full intent.

Programs which analyze these statements are called compiler or processor programs. A typical FORTRAN processor reads statements originating on punch cards and analyzes them by converting each statement into an equivalent sequence of computer instructions. Were these instructions to be executed, it would amount to having the computer carry out the intent of the statement which is itself distinct from the machine's particular instruction repertoire.

Target programs and source programs

The processor program will read and analyze all the statements of a FORTRAN Program to generate a complete set of instructions or "target" program, sufficient, if executed, to carry out the intent of the entire process. The "target" program gets its name because it is the target or objective of the processor program. In turn, the processor has received as input a "source" program written in the FORTRAN language. The processor program finishes generating the target program, the computer may execute these instructions right away. This is feasible because the target program is developed and kept in the computer's memory. When the target program is too big to fit in high speed memory along with the large processor program, the generated target is stored temporarily in some form of auxiliary memory media such as on magnetic drums, discs, or tapes, or in the form of decks of punched cards. When stored in auxiliary memory media, especially punched

† This process is further described in Section 2-4 of the main text.

cards, target programs can be recalled for execution at any subsequent time, as suggested in Figure F2-1. We would rarely wish to read or study the target program ourselves, but in principle this can always be done by causing the processor to print the target program.

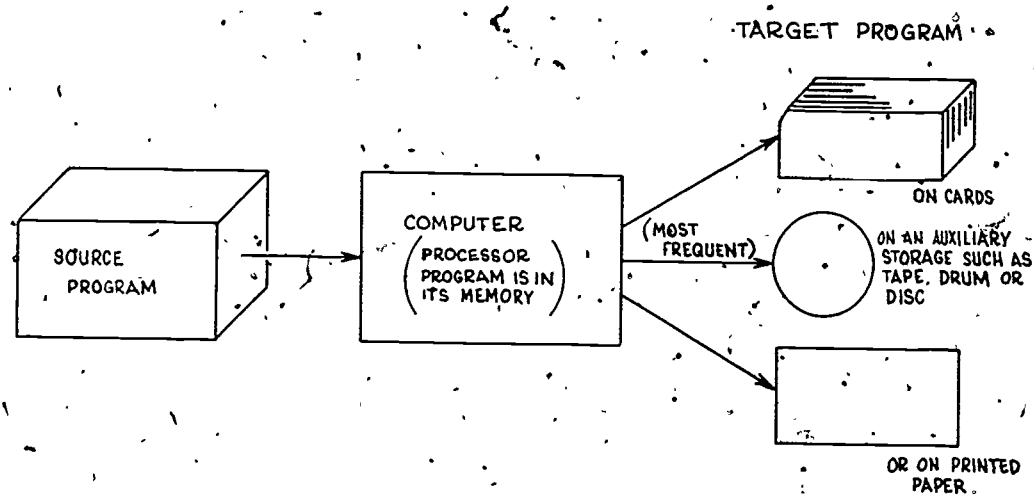


Figure F2-1. The "compiling" process

It may be intriguing to you to learn how the processor program does its job. After all, it is also a flow-chartable process and hence could easily be within our abilities to understand it. Chapter 8 will shed some light on this interesting process. For the present, however, we will avoid any head-on discussion of this topic because our first interest must be to learn to write simple algorithms for solving mathematical problems in FORTRAN. We will, however, be making occasional comments which bear indirectly on the nature and structure of the processor.

General appearance of a FORTRAN program

Recall the process for computing

$$D = \sqrt{A^2 + B^2 + C^2}$$

whose flow chart we displayed in Figure 2-2. Each box can be written as a FORTRAN statement as shown in Figure F2-2.

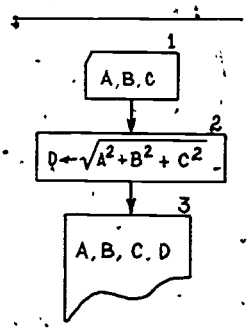
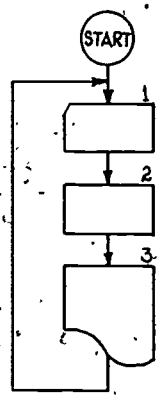


Figure F2-2. FORTRAN Statements compared with Flow Chart Boxes

Notice the similarity between the boxes and the FORTRAN statements. The differences are largely superficial; that is, the symbols used in each case may be different, but the ideas appear to be the same.

We don't have to connect the statements with arrows, because, when we write FORTRAN statements one below the other, we imply that they are to be carried out one after the other from top to bottom. In order to suggest repeating the process for many sets of data, we drew a line from Box 3 back to Box 1 in the flow chart. There is an analogy in FORTRAN to accomplish the same objective. We can simply give the READ statement a numeric label, corresponding to its box number in the flow chart, which is 1 in this case. Then we add after the PRINT a statement which sends control to the designated statement. This is shown in Figure F2-3.



LABEL	STATEMENT
1	READ 100, A, B, C
	D = SQRTF (A**2 + B**2 + C**2)
	PRINT 101, A, B, C, D
	GO TO 1

Figure F2-3. The GO TO statement directs control to any desired statement.

In other words, we give the READ statement the label 1 and then produce a FORTRAN statement,

GO TO 1

for the purpose of indicating a jump or transfer to that statement. The jump statement consists of the symbol GO TO followed by a numeric label. Because it reads like English, the jump statement is easy to understand and we shall say no more about it at this time.

As we focus on an example of a new language, it always takes a while for its features, its special symbols and punctuation patterns to stand out clearly. You have probably observed, and correctly, that in FORTRAN

1. Statements are separated by virtue of being written on separate lines.
2. Alphabetical characters in each statement appear to be exclusively upper case.
3. A statement may have a label. If so, the label precedes the statement, and is separated from it in some way, apparently by some blank space.
4. The assignment symbol is the equal sign (=), an unfortunate state of affairs, since a left pointing arrow would have been our choice!
5. The symbol for exponentiation is the double asterisk (**), and we appear to have lost the ability to use a symbol like $\sqrt{\ }$, having now to use SQRTF which is a curious abbreviation.
6. Certain numbers, specifically 100 and 101, which appear following the words READ and PRINT, have no apparent counterpart in the flow chart.

Now, before taking a more methodical look at input, output and assignment statements and rules for forming these correctly, we show in Figure F2-4. a complete FORTRAN program as it might be written on a coding sheet preliminary to being punched on cards, one card per line of FORTRAN coding.

We can now notice a number of features. Some have to do with layout of the cards. Others relate to the language of FORTRAN itself.

Card layout

A statement label will be punched in card Columns 1 through 5. The

statement proper will be placed in card Columns 7 through 72. We can add comments to identify our program or to help make it self-explanatory: A comment will be recognized if we place the letter C in card Column 1. So the title,

EVALUATION OF D

is treated as a comment instead of a procedural statement, because the letter C will be punched in Column 1 of this card. We don't expect the computer to pay any attention to comments. They are there mainly to help us document what we are doing!

By common convention, Columns 72 - 80 are not considered part of the program. You might say the computer would ignore any part of a program which happens to be punched in these columns.

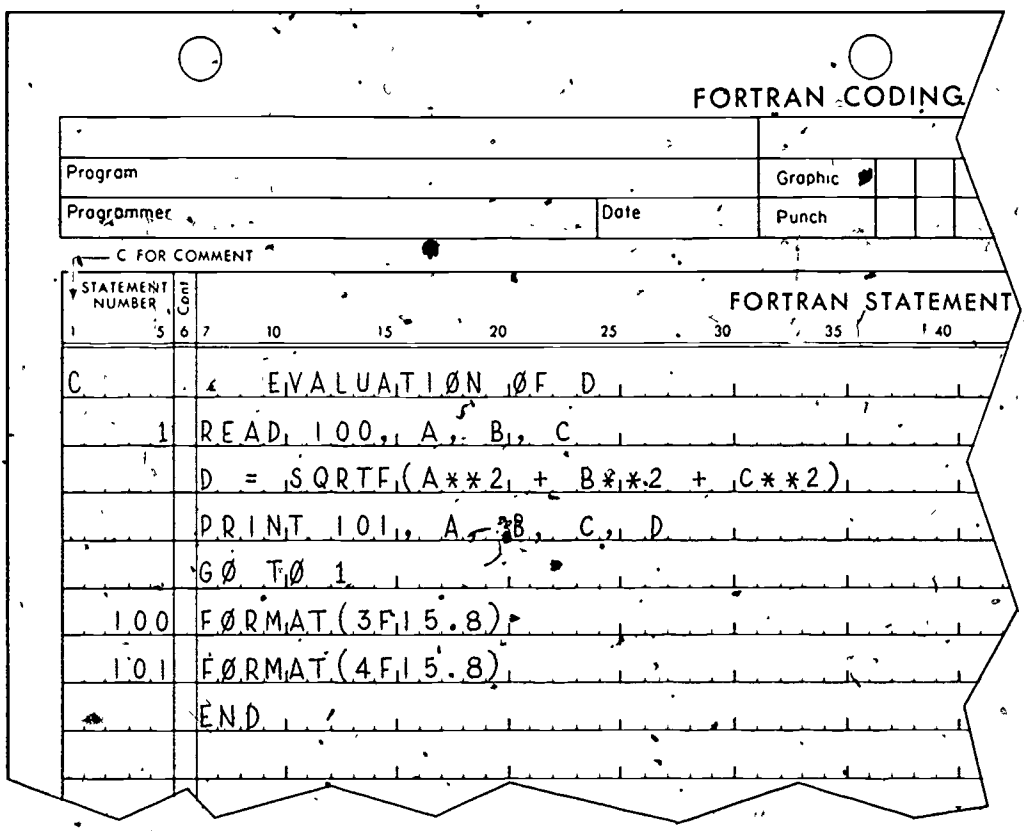


Figure F2-4. Complete program written on a FORTRAN coding sheet

Column 6, although not utilized in the example given in Figure F2-4, will have a special significance to be explained later.

Language

Notice that a statement

END

is placed at the end of the program to mark the end. This is characteristic of all FORTRAN programs. They are not complete without this terminal statement.

There are two new statements shown labeled 100 and 101. Statement 100 provides a format required by the READ statement. By format we mean certain coded information pertaining to the layout of data as it might be punched on one or more data cards or as it might appear printed or typed out on one or more lines of paper. Likewise statement 101 provides a format required by the PRINT statement. We will defer further discussion of format codes until Section F2-3. (If you are wondering what special significance the format numbers 100 and 101 have--don't bother. You will see later that these are chosen quite arbitrarily.)

So far, then, we see that a FORTRAN program appears to consist of a group of statements and possibly comments ending with END. Each statement is punched on a card of standard layout so coding forms like that shown in Figure F2-4 may prove helpful. A picture of the cards produced from this coding form is shown in Figure F2-5.

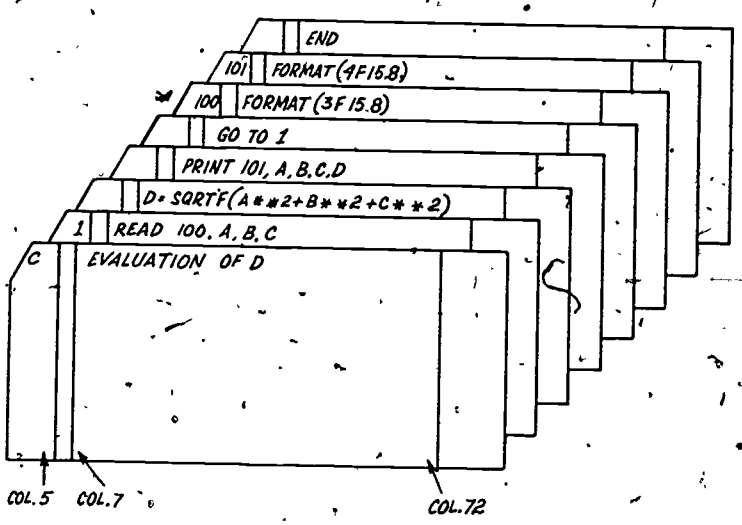


Figure F2-5. The Program as a "deck" of cards

F2-2 FORTRAN language elements

In this section we shall deal with the character set or "alphabet" of FORTRAN and their use in constructing some of the simple components of statements, specifically the construction of numerals (constants and labels), variables, names for functions, and operators.

The character set

The characters which are used in FORTRAN statements are limited to those shown in Table F2-1. Notice the absence of lower case letters.

Table F2-1

The FORTRAN Character Set

(a) Letters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
(b) Digits	0 1 2 3 4 5 6 7 8 9
(c) Special Characters	+ - * / ( ) = , . ' $

Constructing numerals (numerical constants) and labels

To understand how numerals are written in a FORTRAN statement it helps to recall the two schemes we discuss in Chapter One for representing numbers internally, i.e., in a word of memory. There was the integer scheme and the floating point scheme. Externally, i.e., in a FORTRAN statement, there are two types of numerals, called integer and real.

For example, the numerals

4, +14, -15, 1001

are all examples of integers in FORTRAN.

The connection between the external and the internal representation is this: If a numeral, 4, for instance, is written as a FORTRAN integer (externally), then it will be represented internally in the integer scheme.

Expressions involving FORTRAN integers are then evaluated in integer arithmetic which is, of course, characteristically different from real arithmetic as explained in Chapter One.

If a numeral, 4., for instance, is written as a FORTRAN real, then it will be represented internally in the floating point scheme. Expressions which involve FORTRAN reals are then evaluated using real arithmetic.†

We see then, that the way we write numerals as numerical constants in FORTRAN statements (i.e., with or without the decimal point) determines by inspection, and without ambiguity, which type of internal representation the number is to have during computation.

There is one more way we can write a FORTRAN real. This is to use the "E" notation mentioned in Chapter One. You will recall that "E", as it is used in a numeral, is merely a convenient symbol whose meaning is "times ten to the power of". Some examples are:

Number	Method 1 (Without "E")	Method 2 (With "E" notation)
52×10^{-8}	.00000052	52.E-8
5.2×10^{-8}	.000000052	5.2E-8
-2	-2.	-2.E0 or -.2E1 or -.2E+1 or -20.E-1
6.023×10^{23}	602300000000000000000000.	6.023E23 (Avogadro's number, the number of molecules in a gram mole of gas at standard temperature and pressure.)
1.587404×10^{10}	15874040000.	1.587404E10, (1 light year in miles.)

The "E" notation provides us with some flexibility. It is especially useful in expressing numbers which are very small or very large in magnitude.

---

†Strictly speaking, the arithmetic used is floating point which, you recall from Chapter One, was said to be a way of approximating true real arithmetic. We take the liberty of calling this type of arithmetic "real".

Moreover, any one real number can be expressed in a variety of ways. The decimal point in the first part of the numeral can be shifted at will without changing the value of the number, provided a corresponding change (in the opposite sense) is made to the exponent. For example, moving the decimal point to the left is equivalent to multiplying the number by a power of ten, so we have to subtract the same power from the exponent.

Exercises F2-2 Set A

1. Show three ways (other than those shown above) to write -2. in FORTRAN.

2. Use "E" notation to show 1 light year in centimeters. Use a precision of 4 decimal places.

3. Write in customary notation the values represented by the following FORTRAN constants:

   5.96E0, .002E+8, 88.E-1, -522.4, 5.E5

4. Write real constants using "E" notation for the numerals given:

   3.91×10^{-5}

   9.09

   -6.67×10^{3}

   176.4×10^{-9}

5. Do all the FORTRAN constants given below on the left correctly express the suggested values shown on the right? Correct those FORTRAN constants that do not.

179E-4	1.79×10^{-3}
6179.E-7	6179×10^{-2}
16.79E	16.79

Statement labels

In FORTRAN, statement labels are special forms of FORTRAN integers, namely, unsigned integer constants other than zero. Thus 14, 1000, 393 would be perfectly good labels, but -4, 0, 4T, or S1 are not acceptable.

Constructing variables and names for functions

A variable is formed according to this rule:

A letter, or a letter followed by a limited sequence of letters or digits. The total number of characters in the name is limited. Most implementations (FORTRAN processors) permit variables to have up to six characters. We will assume this limit of six characters for our text.

Examples

HARRY TEMP X T46
IKE COLUMN A15AA

Type of number represented

What type of number is represented by a given variable? In the FORTRAN expressions like

A + B or A / B

do you think values for A and B are represented internally in the integer or in the floating point scheme? It makes quite a difference, because the kind of arithmetic which will be performed in evaluating A + B or A / B depends on the internal representation of A and B.

Consequently, a rule for determining the mode or type of internal representation is required. It is this:

All variables which begin with the letters I, J, K, L, M or N correspond to numerical values which are stored in the integer scheme. We shall refer to such variables as integer variables.

All other variable names then correspond to variables stored in the floating point scheme and can be referred to as real variables.

Examples of integer variable names are

ISTEP3, I1, ITEM, KT1, NUM, KCOUNT

Examples of real variable names are

TEMP, X2, FNUMBR, SMELL, ABEL, BAKER

Exercises F2-2 Set B

1. Which of the following inscriptions (or character strings) are invalid for variables?

   ZZFFZ, 1976S, S1976, 19S76, NN,
   ITIF, VAR, T15TT, MAR.Y, MAR*Y

2. Which of the following character strings are valid for integer variables?

   I15Z., N-G, J97, G2IT, ABI,
   0 37, N 37

Names of predefined functions

Several kinds of functions may be named in FORTRAN. In this chapter we shall use a group of predefined functions. The rule for naming these functions is similar to the rule for naming variables except that all names for predefined functions end in the letter F. For example:

ABSF for absolute value function

and SQRTF for square root function.

Other functions of this kind are given in Table F2-2. Like variable names, these function names must begin with a letter and must be followed by a limited sequence of letters (ending in F). The total number of characters in the function name is not entirely standard among the various processors. For this book we shall assume a predefined function name consists of from 4 to 7 characters including the terminal F. The letter F, by itself, is not a legal name for a function.

In order to avoid confusion between these function names and variable names, we shall adopt this arbitrary, but very practical rule:

> Never end a variable name with the letter F, if the name is to consist of four or more characters. Thus, TIFF, TUFF, and JEFF would be avoided, but TIF, TUF, JEF and F are perfectly "OK" names for variables.

Table F2-2

Predefined Mathematical Functions

Name	Meaning
ABSF	absolute value
SQRTF	square root
LOGF	logarithm to the base e
EXPF	powers of e or exponential
SINF	sine of an angle whose measure is given in radians
COSF	cosine of an angle whose measure is given in radians
ATANF	arctangent or principal angle in radians of a given tangent value. That is, ATAN(X) gives a value in radians corresponding to the principal angle whose tangent is X.

Most of the functions listed in Table F2-2 should already be familiar to you. When we use one of these special predefined functions in a FORTRAN statement, the resulting machine code automatically carries out the evaluation of the specified function. The arguments we use with the functions listed in Table F2-2 are all real and the function values that are developed are also all real values.

Operators

To write arithmetic expressions and assignment statements we need operator symbols. Considering the limited character set we have with FORTRAN, it should not be surprising to find some compromises with conventional mathematical notation. Table F2-3 shows a list of the symbols we shall be using for the various arithmetic operators. For convenience in later discussion they are given in hierarchical order, that is, in descending order of precedence which is the same in FORTRAN as it is in our flow chart language.

We have also included the assignment symbol in this group. It is a binary operator, but, of course, not really arithmetic in nature.

Table F2-3

FORTRAN Operator Symbols

	FORTRAN Symbol	Meaning	Example
	**	exponentiation raising to a power	A**B means A^B
same level of precedence	*	multiplication	A*B
	/	division	A/B
same level of precedence	+	addition	A + B
	-	subtraction	A - B
	=	assignment	A =B means A ←B

F2-3 Input-output statements

Now that you have become somewhat accustomed to the appearance of FORTRAN characters and to their use in the elementary components of the language, like numerals, variables, function names, and operators, you are ready to study the three important statement types: input, output and assignment.

You have already seen examples of input and output statements, namely:

```
READ    100, A, B, C
```

and

```
PRINT   101, A, B, C, D
```

The statements are simple in form. They always begin with the key word

```
READ
or  PRINT
```

Following the key word we provide an unsigned integer constant which is the label of another FORTRAN statement. This label tells where to get some needed format information. We call this label a format number. A comma follows next. Then we write the list of variables whose values are either to be read or printed. This list is simply a set of variables, separated by commas, if there is more than one.

We will not limit the number of list elements of an input or output list, but we will prohibit certain things from being on the list, a constant for example.

Thus,

```
READ    100, 2.5, A
or PRINT    150,  A, B, 2.5,  C, 152,  D
```

are invalid, because constants are prohibited as list elements. In the same vein any expressions which require evaluation, such as A + B, and such forms as (A), (-A), +A, or -A, are prohibited.

Thus in the two statements that follow

```
READ    150,  A + B,   -A,   +B
or PRINT    151,  A*A + B*B
```

each list element is invalid.

In this way we arrive at the general form of an input or output statement, i.e.,

```
READ   format number, input list
or     PRINT  format number, output list
```

For a READ statement, the list elements are the variables whose values are to be assigned from the input data, while for a PRINT statement the list elements are the variables whose currently assigned values are to be printed.

Executing a READ statement

In this discussion we shall assume, as in the flow chart text that input data originate on punched cards. The effect of executing a READ statement is as follows.

1. First we assume a card is in position to be read by the computer's input device. If not, the execution of the program ceases immediately. In some but not all systems the computer might then print some message like

   "YOU HAVE RUN OUT OF DATA"

   or "ALL INPUT DATA HAVE BEEN PROCESSED"

2. The format number of the READ statement is then used to consult (look up) the format code stored in memory which has this format number. The format number is simply a way of identifying the particular format code you want among several different formats that may be employed in the same program.

   Format code, when properly interpreted, will, in effect, tell the computer where on the card to find the desired values. The code itself is written in a special language which will be described momentarily.

3. The content of the card that is ready to be read is then transported to the computer memory, where it is examined one character at a time beginning with Column 1. The format code which was found in Step 2 is then used to decide which numerals (from the card) go with which variable (of the input list). The purpose is to achieve a one-to-one matching between the numerals on the card and the variables of the input list with which they are to become associated.

At this point some examples should help us see how this matching process is carried out.

Example 1

Study Figure F2-6 where you see a READ statement, an associated FORMAT statement, and a picture of a card that might possibly be read as a result of executing the given READ statement.

The format number is 22. The associated format code is (I15, I15, I15). The code is interpreted to mean--in this case--that: the data are arranged on the card in three groups, each fifteen columns in width. The letter, I, for integer, tells us that each group of 15 columns is expected to contain an integer which can be stored in memory at a place associated with an integer variable.

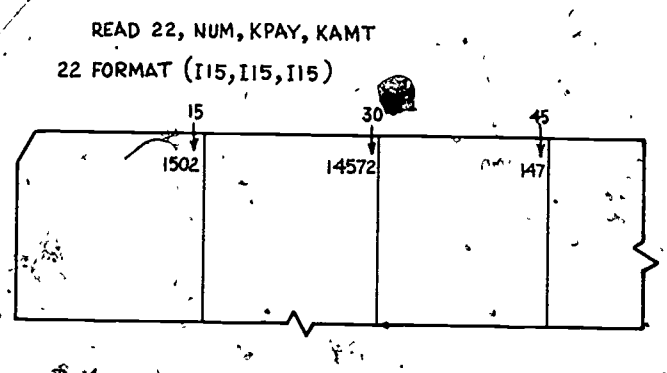


Figure F2-6. A data card with three integers punched on it, together with related READ and FORMAT statements.

We shall call each such group of columns a field. Codes like I15 are then called field codes because they tell the type of number, I, (for integer) in this case, and the width of the field in which the numeral can be found.

You may wonder if there is any special reason for choosing a field width of 15 columns for these integers. We want a field which is, of course, wide enough to contain the largest expected integer value. But it doesn't hurt to use a wider field if room is available on the data card. The main thing to remember, however, is that the units portion of the data value should be punched in the rightmost position of the card field. This keeps you out of trouble because, in many computer systems, any blanks inside the field and to the right of the numeral will be interpreted as trailing zeros. Thus, if the integer 1502 were punched in columns 9-12 instead of 12-15, on the card shown in Figure F2-6, it would be erroneously interpreted as 1502000.

As the scan of the data from the card proceeds, column by column, the information is picked off one field at a time to be matched with the next item in the input list. When a match is achieved, then that numeral from the card is converted to its internal form as a sequence of bits representing that number in the appropriate scheme. This is done by a special program

provided by the processor. For I-fields the conversion is made to the integer scheme.

Several general remarks are always true concerning format statements.

1. Every format statement in a program must have a (unique) statement number.
2. Every format statement has the form FORMAT (format code).

Example 2

Now study Figure F2-7 where you see another READ statement along with an associated FORMAT statement. Here we show a data card with numerals for three real numbers punched on it. The field codes, which show where these numerals are to be found and how they are to be converted to their internal form, are each F15.8.

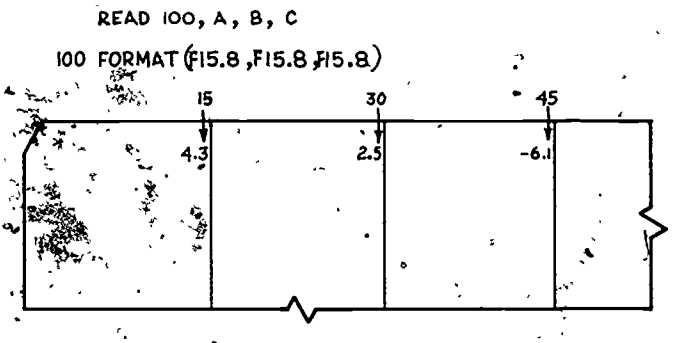


Figure F2-7. A data card with three values punched on it, together with related READ and FORMAT statements.

The letter F (for floating point) tells us that the group of 15 columns is expected to contain the numeral for a real number, and when converted to internal form, it should be represented in the floating point scheme.

In Figure F2-7 each numeral on the data card contains a decimal point. It's easy and quite natural to punch the decimal point with each numeral, and when it is punched, the position of the decimal point on the data card will decide its representation in the floating point scheme. There is provision, however, to omit the punching of the decimal point and have the computer figure out where it ought to be in each field; that is, locate the place in the numeral where the decimal point is intended to be. Use of this feature requires a little more care than is probably warranted at the beginning. However, it is worth knowing about for several reasons. It will help you, for one thing, to understand the meaning of the 8 in the field code F15.8.

Properly interpreted this code means: The numeral found in the next 15 columns of the card is to be converted for storage in the floating-point scheme. If the numeral does not have a decimal point, actually punched on the card, then a decimal point is located by counting 8 columns to the left from the 15th, or rightmost, column of the field. The decimal point is intended to go immediately to the left of this eighth column. You can see this illustrated in the card picture found in Figure F2-8.

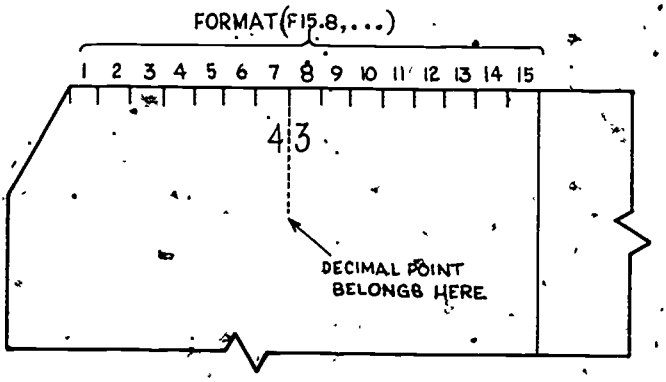


Figure F2-8. F-type data without punching the decimal point. The numeral 43, which is punched in Columns 7 and 8, will be interpreted as 4.3.

The numeral 43 is punched in Columns 7 and 8. We assume that no other punches are present in the first fifteen columns. Because no decimal point is shown on the data card, the numeral will be interpreted as 4.3 when it is finally stored in memory under "control" of a field code F15.8.

We see that F-field codes, which are used to identify numerals representing real numbers punched on data cards, are somewhat more complicated than I-field codes. An easy way to view the form of these codes is suggested in Figure F2-9.

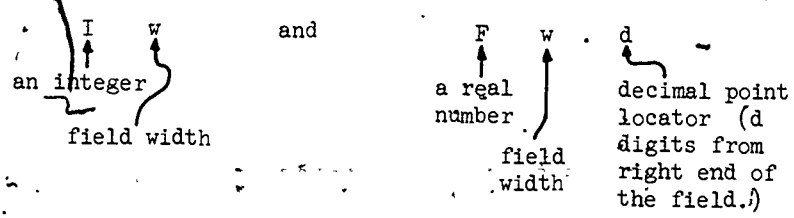


Figure F2-9. General forms for I- and F-field codes

Strictly speaking a field can be of any width, w, from 1 to 80, the width of the card, not just 15 columns as we have been illustrating. Similarly, the decimal point locator d can vary in value, and not just equal 8 as illustrated thus far. For the time being, however, in order not to have to keep so many new ideas in our head at once it will be simpler to imagine the field widths are fixed at 15 and a value of d fixed at 8.

Another point about writing field codes--and this one is worth remembering--is that you don't have to write a series of identical codes. You can group them and use a repetition indicator to tell how many are intended in sequence. Thus in Figure F2-6 we could write

```
22 FORMAT (3I15)
```

instead of

```
22 FORMAT (I15, I15, I15)
```

Similarly, in Figure F2-7 we could write

```
100 FORMAT (3F15.8)
```

in place of the more tedious

```
100 FORMAT (F15.8, F15.8, F15.8)
```

Example 3

A card can contain any combination of I- and F-fields desired and in any sequence as may be seen in Figure F2-10. Notice that the one-to-one match is properly achieved employing format number 15. The first and fourth list elements in the READ statement are real variables and the second and third items are integer variables (according to the leading letters in their names). Correspondingly, the first and fourth field codes of format number 15 are F-fields while the second and third are I-fields. This guarantees that assigned values for the real variables will be stored in the appropriate floating-point scheme, while assigned values for the integer variables are stored in the desired integer scheme.

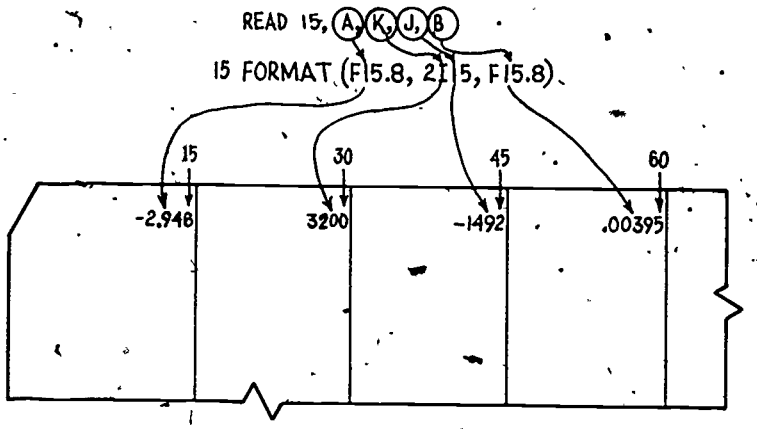


Figure F2-10. Two field types on the same data card

Exercises F2-3 Set A

We imagine a class of very simple problems to be solved in the computer. Let the flow chart for each of these have a structure identical with the one in Figure F2-3. In the following exercises you are given Box 2. Your job in each case is to:

A. Decide what should be in Box 1.

B. Write an appropriate READ statement.

C. Write a companion FORMAT statement.

D. Draw a picture of a typical data card which could be read as a result of executing the READ statement which you have written.

1. $Z \leftarrow 2.5 + T$ (2)

2. $\ell \leftarrow n \times (i - 1) + j$ (2)

3. $Z \leftarrow ((a \times x + b) \times x + c) \times x + d$ (2)

4. $Q \leftarrow \sqrt{(u - \frac{2}{2}) \times v}$ (2)

5. $X \leftarrow 2/(Y + \frac{A}{Y})$ (2)

6. $$\text{AREA} \leftarrow \frac{3.14159}{2} \times r^2 - (s \times \sqrt{r^2 - s^2} + r^2 \times \text{PHI})$$

Executing a PRINT statement

Printing under control of a given format is analogous to reading, but in a reversed sense. The value that is currently assigned to each element of the output list will appear typed or printed across the page by the output device. When each number is copied from its place in memory it is converted for printing to the external form that is dictated by the matching field code. The spacing of the numerals on the printed line is set automatically by the widths of the corresponding field codes. As you can see, the field codes, in a sense, control or govern the appearance of the printed line. Several examples will help you to see this process.

Example 1

Examine Figure F2-11 where you will see an example of a PRINT statement and what it might accomplish when executed. Notice the format is essentially the same as that used in an earlier example for a READ statement. This illustrates the fact that the same format can be used in connection with any READ, PRINT, or group of such statements, so long as each refers to the format by the same number.

For most simple problems, it is helpful to obtain a computer-produced printed copy of every piece of data that is read in to the computer. This process is sometimes referred to as echo checking the data. Of course it isn't essential to print the data values in precisely the same order as they may have appeared on the data card. Thus, a program may begin with statements like

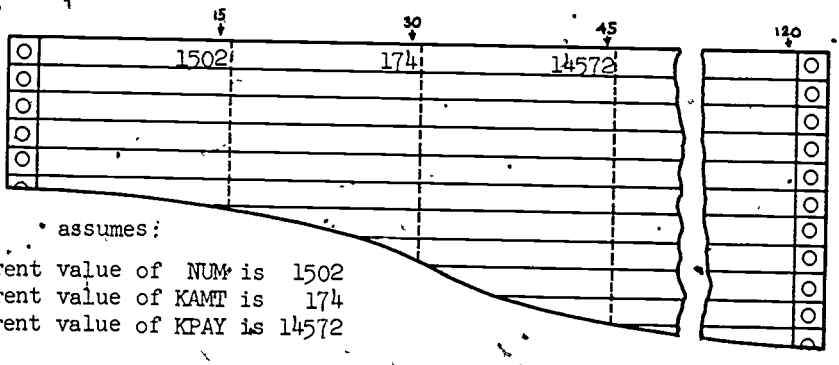
```
1    READ 22, NUM, KPAY, KAMT
2    PRINT 22, NUM, KAMT, KPAY
22   FORMAT (3I15)
```

Data read as a result of executing statement 1 will be printed as a result of executing statement 2. The echoed data can then be checked visually to see if the computer got the right information. Both statements refer to the same format.

Now since statement 22 has a passive or reference role, rather than an active or "executable" role, its position in the sequence of statements

which forms a program isn't really critical. To emphasize this distinction, we shall henceforth call non-executable statements like the format, declarations, because, in a sense, they provide "declarative", or descriptive information only. In this way we can reserve the word statement for the class of action or executable steps of a computer program. FORTRAN has several other declarative statement types. You will be introduced to these in later chapters. You might now check Figure F2-4 and notice how the FORMAT declarations in that program are placed at the very end (just ahead of END). They could just as well have been placed ahead of statement number 1. Many people prefer to keep FORMAT declarations near the input or output statement which refers to it (or near the first of these).

```
      PRINT 22, NUM, KAMT, KPAY
   22 FORMAT (3I15)
```



assumes:

current value of NUM is 1502
current value of KAMT is 174
current value of KPAY is 14572

Figure F2-11. Example of a PRINT statement, its referenced FORMAT, and a possible line of printed results caused by the execution of the PRINT statement

Maximum line widths

You may have wondered how many numerals can be printed on one line, and, if more than one line is needed, how the format can be written for use in controlling such a process. Unlike the punched card which has a fixed number of columns, the maximum width of a printed line is not so well standardized. For a given class of printing equipment there is some degree of uniformity, however. Thus 120 characters or 132 characters are common "standards". For our text, we will assume a maximum line width of 120 columns or characters. Hence, if we use fixed field widths of 15 columns, we will be able to print up to eight numerals per line.

Example 2

Now suppose we want to print, say ten values, as a result of executing a single PRINT statement. Since many printers use type spacing of 10 characters to the horizontal inch, the 120 character width line is 12 inches across. For many purposes this is wider than we actually need. To keep our page width narrow enough to fit in our notebook or in an $8\frac{1}{2} \times 11$ report, let's suppose we agree to print not more than four numerals on any one line. Figure F2-12 illustrates one way this might be done. We use as our format (4F15.8) which is interpreted to mean: up to four items of the form F15.8 (per line). The numerals on each printed line are to account for up to four list elements of the PRINT statement. When the PRINT statement is executed, values are copied out of memory one at a time from positions associated with A, B, C, etc. Each copied value is then converted for printing in the desired external real form, as illustrated.

```
      PRINT 31, A, B, C, D, E, F, G, H, P, Q
31    FORMAT (4F15.8)
```

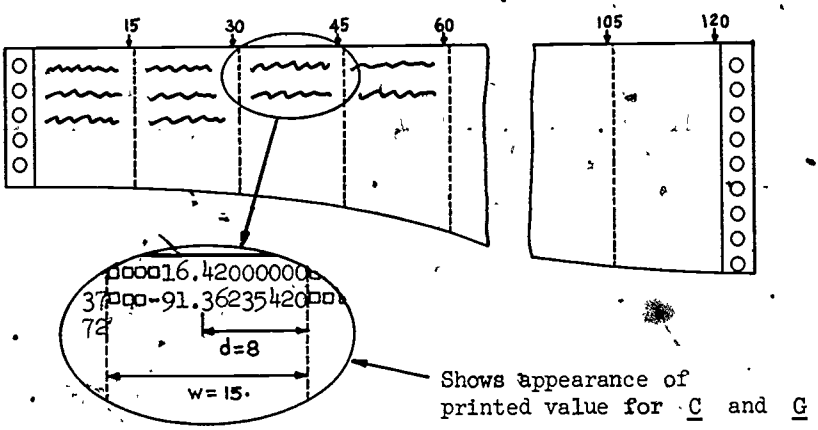


Figure F2-12. Showing printing of up to four items per line, each item under control of an F15.8 code

If a line printer device is used, then, when four such numbers have been converted, a complete line is printed at once. However, if a typewriter device is used, each number is typed as soon as it has been converted to output form. This difference in behavior is no vital consequence as the net effect is the same. In any case, when four items have been dispensed with, the format, (4F15.8), is reused and the next four numbers are matched with the four F15.8's, converted and printed. The format is again reused. This time the last two list elements are matched with the first two of the F15.8's, converted and printed. Since the last of the ten values has now been copied out, and the number properly converted under control of its matching field code, the list is discovered to be exhausted, i.e., no more items remain to be copied. This discovery then signals the end of the process. Execution of the PRINT statement is terminated and the computer is free to execute whatever statement happens to be next in the program.

Exercises F2-3 Set B

For the following output lists, write PRINT statements, each with a suitable FORMAT declaration to go along with it. Use only I15 and F15.8 field codes. Assume a maximum line width of 90 columns, i.e., six items.

1. A, B, J, K, and L

2. A, B, C, J, K, L, A1, B1, C1

3. IKE, JAK, BAKER, CHARLY, D, B

Exercises F2-3 Set C

1 - 6. In these six exercises, you will continue with the development begun earlier, in Section F2-3, Set A. Your job now in each exercise is to:

a. Decide what should go in Box 3 of the flow chart.

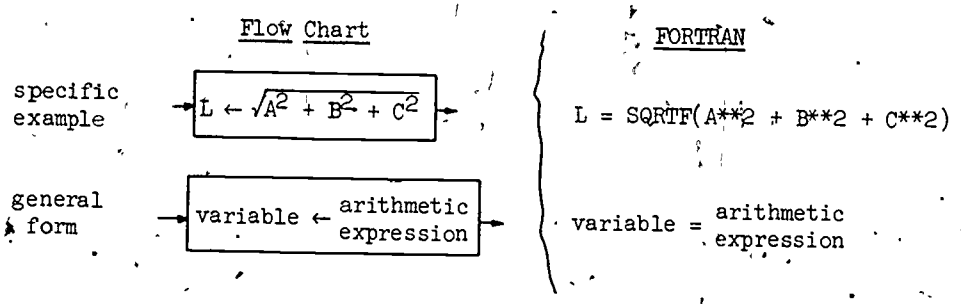
b. Write a PRINT statement which, if executed, would carry out the intent of Box 3.

c. Show an accompanying FORMAT declaration.

d. For the data you used in the first set of exercises, compute the result which would be printed and show its expected formatted value as it would appear on the printed page.

F2-4 Assignment statements

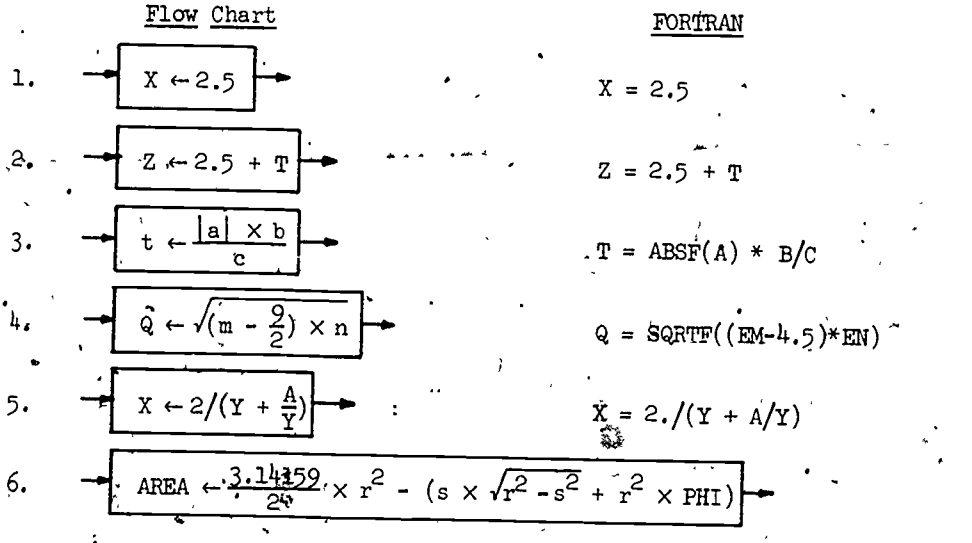
We shall first examine the parallel existing between our previously developed flow chart concepts of assignment steps and those of FORTRAN statements.



The FORTRAN variable is a character string built up of letters and digits as described in Section F2-2.

The arithmetic expression in FORTRAN corresponds to any meaningful computational rule which uniquely defines a single numerical value. There will be a few important restrictions to observe in writing such expressions correctly. Before considering these, let us see how several flow chart examples are rewritten in FORTRAN.

Examples



We have already become acquainted with three of the basic components of arithmetic expressions; namely, constants, variables, and operators.

Two other important components are parentheses and function references.

In FORTRAN we shall use parentheses in several different ways. If you look at Example 5, you will see the parentheses used to form subexpressions that force a desired ordering to the computation. This is the use which is perhaps most familiar to us.

In Example 3 you see the parentheses used in a new way--to enclose the argument of a function. A reference to a function value, or function reference, for short, consists of the name of the function followed by a pair of parentheses enclosing the argument. This convention, in FORTRAN, which is used for all functions, is often used in mathematical notation. For example,

$$f(x)$$

may mean "the function f evaluated with argument x", or simply, "f of x".

Since some functions are given special marks or symbols, like $|\ |$, $\sqrt{\ }$, $\sqrt[3]{\ }$, which are simply not available in the FORTRAN character set, it makes good sense to use the parentheses. We then get a uniform way for writing function references.

How are the parentheses used in Example 4? There are two pairs used here, one nested within another. You can probably see easily that the outermost parentheses serve to enclose the argument of the SQRTF function. This argument is the value which will be obtained upon evaluation of the expression

$$(EM - 4.5) * EN$$

Here the parentheses are employed for ordering computation.

As you can see, a function argument may itself be a FORTRAN expression of decided complexity. A similar example is given in Example 6.

Number types in arithmetic expressions

In general, an expression in FORTRAN must be homogeneous. By this we mean the variables, constants and function references which are joined by arithmetic operators in one expression, all must be of the same type--either all real or all integer. (There is one exception which is treated a bit later.)

For example:

```
A + 5.
6. * T
F * G
6. * SQRTF(X)
6./ABSF(T - P)
```

are examples of homogeneous real expressions.

```
K + 4
4 * K**IT
J/K * 3
J * (J - 2 + K/M)
```

are examples of homogeneous integer expressions.

In the case of real expressions, each arithmetic operation is carried out in real arithmetic. The value resulting in any of these expression evaluations is developed internally in floating point representation.

In the case of integer expressions, each arithmetic operation is conducted using integer arithmetic and the resulting value is developed internally in integer representation.

Target code produced from an arithmetic expression is generally more efficient when the expression is homogeneous. Moreover, the results of each operation are easier to define. For these reasons, and other purely pragmatic ones, the original FORTRAN languages and compilers were designed and constructed to reject, as illegal, any expression that was not homogeneous.

Many of the later versions of the FORTRAN language retained this restriction. As a result, expressions like

```
I + 1.
I + A
A + 4
7/T + 5. * U
(M - 4.5) * N
```

are illegal in FORTRAN! You can now see why, in Example 4, names like EM, in place of M, and EN, in place of N, were used in constructing a homogeneous real expression as the desired argument for SQRTF.

Because in customary mathematical notation we do not normally impose this restriction against "mixing modes" of the operands, you may find yourself very frequently writing illegal FORTRAN without meaning to. If you fail to make the necessary changes to correct the error, fear not! When your program is read by the computer, under control of the processor program, it will certainly be rejected for this reason. In many cases, in fact, the processor will find most or all such illegal expressions the first time the program is examined. Not only will you receive a printed rejection slip, but all such violations which have been found will be identified rather clearly. So, it will be very easy to correct this type of error!

Special case of exponentiation with integral powers

You may have wondered if indeed all homogeneous expressions are illegal in FORTRAN. Are there no exceptions? Some of you may recall the discussion in the main text on exponentiation. In order to distinguish between

a^3 with the meaning $a \times a \times a$

and

a^3 with the meaning $e^{3\ell na}$

we agreed that two different ways to write a^3 were needed. In FORTRAN the notation

```
A ** 3
```

is permitted so that we can imply the computation $a \times a \times a$ and distinguish it from

```
A ** 3.
```

which is an order to carry out the computation $e^{3\ell na}$. Note that a more cumbersome expression equivalent to A ** 3., can be written in FORTRAN. It is: EXPF(3. * LOFG(A)) which also means $e^{3\ell na}$.

In short, raising a real expression (like A) to an integral power (like 3) is the one and only form of nonhomogeneous expression which is permitted.†

†In more advanced versions of FORTRAN like FORTRAN IV, there are other forms of nonhomogeneity which are permitted.

Which of the two expressions,

A ** 3 or A ** 3.

do you think is easier to carry out manually? By computer? The obvious answer is A ** 3 in each case.

To compute $e^{3\ell na}$ requires the determination of a logarithm and the raising of e to a power. These operations are carried out in a computer with the use of the LOGF and the EXPF functions, which are separate programs. Though automatically supplied when expressions like A ** 3. are used, each ordinarily involves from 10 to 100 times as much computer time as a simple multiplication. We can conclude therefore that, when a choice is available, expressions like A ** 3 are to be preferred to A ** 3., because a more efficient target program will result.

Integer division and its relationship within the greatest integer function

Integer division in FORTRAN plays an important role in algorithmic processes because it is related to the TRUNK function which we defined in Section 2-5 and which is in turn related to the logically powerful greatest integer or "bracket" ([ ]) function. Specifically, the FORTRAN expression I/J is equivalent to the mathematical expression $TRUNK(\frac{I}{J})$ for all integers I and J.

	FORTRAN Expression	Computed Value	Mathematical Equivalent
1.	9/10	0	[9/10]
2.	-10/(-10)	1	[10/10]
3.	11/10	1	[11/10]
4.	10/1	10	[10/1]
5.	-5/10	0	-[5/10]
6.	-15/10	-1	-[15/10]
7.	10/(-1)	-10	-[10/1]
8.	1/(-10)	0	-[1/10]

Notice that when I and J are both positive or are both negative, the FORTRAN expression

I/J,

is equivalent to the mathematical expression, $[\frac{I}{J}]$. On the other hand, if the sign of either I or J, but not of both, is negative, the FORTRAN expression

I/J

is equivalent to the mathematical expression, $-[|\frac{I}{J}|]$.

Special note on the exponentiation operation (**)

You may have wondered how we would write expressions like

$$A^{B^E}$$

in FORTRAN. Either we mean

(1) A ** (B ** E), i.e., $A^{(B^E)}$

or we mean

(2) (A ** B) ** E, i.e., $(A^B)^E$ which is really $A^{(B \times E)}$.

As you can see, they'll only be the same when B^E is the same as $B \times E$.

We must therefore conclude that the ** operator is not associative, and in FORTRAN it is invalid to write

A ** B ** E

because it is considered ambiguous. One of the two unambiguous forms (1) or (2), whichever is desired, must be explicitly written.

Functions which have function values as their arguments

In some of the expressions you will see in the next exercise, the argument of a function is expressed in terms of the value of another function. This is perfectly permissible in FORTRAN as there is no restriction on the complexity of an arithmetic expression when used as an argument of a function. Another example of this which we saw earlier was the expression

EXPF(3. * LOGF(A))

argument of EXPF

Exercises F2-4 Set A

1. We would like to express $A^{3/2}$ in FORTRAN, where $A \geq 0$. Keep in mind that, $A^{3/2}$ is, in this case, the same as $\sqrt{A^3}$, or $(A^3)^{1/2}$. All of the following correctly express $A^{3/2}$ in FORTRAN. Some are awkward. Some have superfluous operations. Comment on each and choose the one which appears to be the most efficient computationally.

   a. ABSF(A ** 1.5)
   b. (A ** 3) ** 0.5
   c. SQRTF(A ** 3)
   d. ABSF(SQRTF(A ** 3))
   e. (A ** 1.5)
   f. ABSF(A) ** 1.5
   g. SQRTF(ABSF(A ** 3))

2. If A can have negative values, which of the seven FORTRAN expressions given in the preceding exercise correctly expresses $|A|^{3/2}$? If more than one, which is simpler computationally? Explain.

Unary minus

In Section 2-4 of the flow chart text you learned to distinguish between the unary and number-naming minus, on the one hand, and the binary minus, on the other hand. If a minus sign appears at the very beginning of an expression, or immediately following a left-parenthesis, it is either a unary minus or a number-naming minus. It cannot be a binary minus.

Here are some examples in FORTRAN statements.

```
1.  Q = -5.
2.  Q = -(A + B)
3.  Q = -A ** (-C)
4.  Q = (-4.) ** 2
5.  Q = -(4. ** 2)
6.  Q = -4. ** (2).
7.  Q = -4. ** 2
8.  Q = SINF(-A + B * (-COSF(C)))
```

Note that (4) assigns a value of +16 to Q. Remembering from Table 2-4 of the flow chart text that exponentiation takes precedence over unary minus,

we see that (5), (6) and (7) each assign -16 to Q. Similarly, -A ** (-C) is the same as -(A ** (-C)).

Of course, no two operators may be written side-by-side. Thus, A × -B is invalid. We must write instead A * (-B) or perhaps -A * B. Similarly, A ** + C or A ** - C are both invalid.

Exercises F2-4 Set B

Correct the following three invalid FORTRAN statements. What problems arise, if any, in correcting the second and third statements?

1. T = T * -A
2. F = C/-3 + 4
3. G = A + B * (C * -F/D)

F2-5 The order of computation in a FORTRAN expression

We have not deferred this question to this late point because there is something special that must be said here about FORTRAN that is different from what we have already said in Chapter 2 of the flow chart text. On the contrary, the rules for interpreting the order of computation are precisely the same. If you don't recall these, close the book now and try to reconstruct them. Then compare them with the following.

1. When parentheses are used to nest one subexpression inside another, evaluate these nested subexpressions in the order from the innermost to the outermost.

2. Within any one subexpression evaluate in descending order of precedence:

Highest	function references
↓	** (exponentiation)
↓	*, /
Lowest	+, -

3. In case of a tie in precedence level (within any subexpression), perform those operations in the tie from left to right.

F2-6 Meaning of assignment when the variable on the left is of different type from the expression on the right

Is it possible to convert a number from integer to real representation or vice versa? Based on our discussion so far, it would seem not. To answer this question properly, you should notice that until now all the assignment statements we have illustrated were homogeneous in the sense that the variable on the left of the (=) sign and the expression on the right of it were both of the same type (real or integer). There are two other obvious possibilities in FORTRAN. They are not only both legal, but highly useful.

In short, we have four cases:

(a) real variable = real expression

or

(b) integer variable = integer expression

(c) real variable = integer expression

(d) integer variable = real expression.

We have nothing further to say about cases (a) and (b). It is (c) and (d) we are interested in because such statements can be used to convert integers to reals and vice versa.

Let's first consider case (c). The number assigned to the real variable simply has no fractional part.

Example

```
I = 4
J = 3 * 6
T = I + J
```

Observe that this sequence of FORTRAN statements will lead to a real value for T equal to 22.0. In other words, the integer sum of I + J results in the real value for T having a zero fractional part.

Now consider case (d). Here the integer value assigned to the variable is truncated in the sense described in Chapter 2, Section 2-5. In other words, when the expression is real and the variable is integer, then

variable = expression

really means

variable = sign of expression × [|expression|]

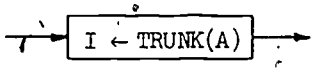
[absolute value marks]

[greatest integer]

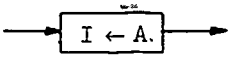
What this boils down to is that a FORTRAN statement like

I = A

is the equivalent of the flow chart assignment:



which is by no means the same as



Why? Because the flow chart variables I and A do not have specific digital representations associated with them. Hence, no rounding (lopping-off) can be implied in the simple flow chart assignment.

Example 1

```
DIAM  = 5.9
ICIRC = 3.14159 * DIAM
```

This sequence of FORTRAN statements leads to a value of 18 for ICIRC, and not 18.535 which would be the actual circumference of a circle with a diameter of 5.9.

Example 2

```
BALNCE  = 52.51
WITHDR  = 92.49
IOWE = BALNCE - WITHDR
```

Assuming BALNCE and WITHDR refer to bank balance and withdrawal, the overdraw is $39.98. The fractional part, .98, is lost when storing -39 in IOWE.

Exercises F2-6

1. Which of the following statements are invalid in FORTRAN? Explain.

   (a) T1 = T1 * VAR3/4

   (b) A = EXPF(A4 * Z ** 3. + A3 * Z ** 2.)

   (c) Y = LOGF(SINF(F)) + 22

   (d) J = J + 1.

   (e) I = I/K

2. Assuming 1(b) above is valid, what changes would you propose in the interest of efficiency?

In each of the following write a sequence of one or more FORTRAN assignment statements to accomplish the indicated task.

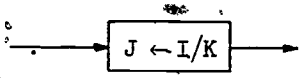
3. Assign to IPART the integral part of the real value now assigned to V.

4. Assign to FPART the fractional part of the real value currently assigned to V.

5. Provide an alternative real representation for the integer value currently assigned to the variable called INTV. Call this new representation RINTV.

6. Compare the following FORTRAN statement with the accompanying flow chart box

J = I/K



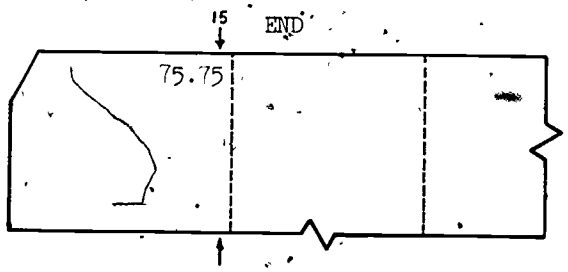
Are they the same? If not, change the flow chart box to conform with the FORTRAN statement.

F2-7 Writing complete programs

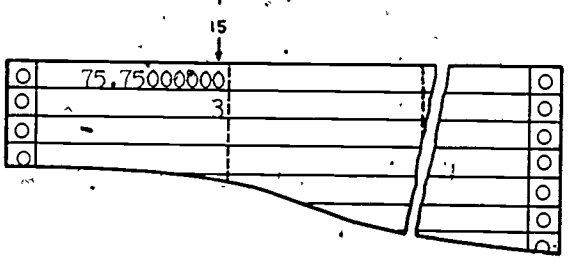
Remember the $20 bill problem? You will now be able to write FORTRAN statements, indeed a complete FORTRAN program, which will show how to compute the most $20 bills obtainable from the given (real) PRICE of Dad's Jersey cow. Before looking at the version given below in Figure F2-13, write your own first and then compare the two.

FORTRAN PROGRAM

Label	Statement or declaration
	C WHAT THE JERSEY COW WILL BRING
	READ 101, PRICE
	C PRICE IS GIVEN IN
	C DOLLARS AND CENTS
	PRINT 101, PRICE
101	FORMAT (F15.8)
	C NOW CONVERT TO PENNIES
	IPRICE = PRICE * 100.
	C COMPUTE NUMBER OF 20'S
	NUM20 = IPRICE/2000
	C NOW PRINT ANSWER, I.E.,
	C VALUE OF NUM20, AND STOP
	PRINT 102, NUM20
102	FORMAT (I15)
	STOP
	END



Picture of a Data Card



Picture of Results

Figure F2-13. The Jersey Cow - programmed in FORTRAN

Exercises F2-7

1 - 6. In the exercises of Section F2-3, Set A and C, you worked out the input and output details needed for six FORTRAN programs each having the simple loop structure shown in Figure F2-3. You're to finish the job now by writing out on a coding sheet each of these six simple FORTRAN programs.

7. What single assignment statement can replace the two that are used in the program given in Figure F2-13?

8. Recall the problem (in Section 2-5 of the main text) to simulate winning points on a carnival roulette wheel. It's presumed you have already drawn a flow chart for this situation. Now write a complete FORTRAN program which is equivalent to your flow chart.

F2-8 Some clerical details

This section introduces you to some additional details concerning the preparation of punched cards for FORTRAN programs.

Length of a statement

A FORTRAN statement (or declaration) can be, for all practical purposes, as long as necessary. If it is necessary to continue any one statement on a series of lines, provision is provided in the coding form to indicate continuation. This is the purpose of Column 6 on the coding form shown in Figure F2-4. Thus, suppose we choose to write the statement

```
T = SQRTF(A ** 2 + B ** 2 + C ** 2)
```

on two lines instead of one. Its appearance on a coding form might then be modified as shown in Figure F2-14.

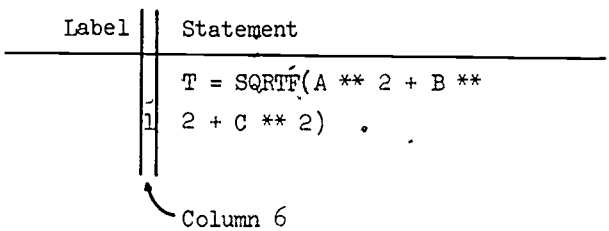


Figure F2-14. Showing use of the continuation code on a coding form

The digit 1 is used here on the continuation line. Any digit is satisfactory except zero. Notice we leave Column 6 blank on the first line. The cards punched from these instructions are pictured in Figure F2-15.

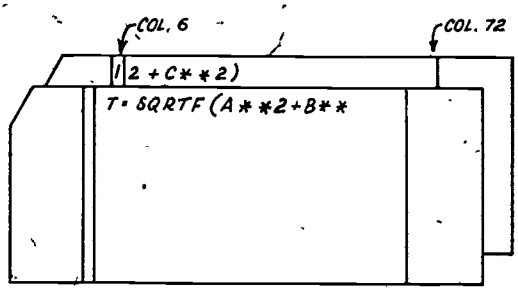


Figure F2-15. Showing use of the continuation code on the punched card

Blank spaces (columns) in a statement

A blank column in any statement or declaration punched on a card is, with one exception, always ignored in FORTRAN. To illustrate this point we can let a ō represent a blank card column. Then the statement

AōBōEōLō=ōBōAōKōEōRō+ō1ōōō.ō4

will be treated in the same way as

ABEL=BAKER+10.4

So blanks can appear anywhere and will be ignored. Moreover, the A in ABEL need not be punched in Column 7. A statement may be punched beginning in any column which one chooses, between 7 and 72 inclusive. The same principles apply to READ, PRINT and FORMAT. - The exception, where blank columns do have significance occurs in H-field codes used in FORMAT declarations. This subject will be treated in the next chapter.

F2-9 The printer carriage†

The carriage of a line printer, which holds and rolls the paper, must receive instructions as to how much to roll or feed the paper forward prior to the printing of the next line. Note this is analogous to an electric typewriter receiving a carriage return impulse which causes the paper to move forward one unit or line in the vertical direction. Also note that vertical paper movement on either a typewriter or on a line printer is forward only: If we are going to space the printed material so as to get various vertical arrangements, we will want to have the computer issue spacing commands to the printer carriage telling how many lines are to be moved, preferably before each printing action.

How is this control achieved in the FORTRAN language? To see how this is done, we should first visualize a string of characters which has been developed for printing as one line. Call this a line image. These characters are, in general, the numbers converted to their output appearance governed by one or more field codes. The printer which receives this line image is so wired that it can (and does) "shunt off" the leading character (i.e., the left-most in the string) and interpret it in a special way. Instead of being treated as the first of a series of characters to be printed, this character is received as a coded signal which then activates carriage movement. Depending on the character, the carriage moves various distances (i.e., feeds the paper forward various amounts). For this reason we call the leading character of a line image the carriage control code. It is never printed.

The three codes which all printers are wired to understand, and their resulting effect are:

Carriage control code	Effect
□ (blank)	Rolls paper forward 1 space.
0 (zero)	Rolls paper forward 2 spaces.
1 (one)	Rolls paper forward so as to position it at the top of the next page leaving a one-inch margin at the top. A "page" on a continuous printer form roughly corresponds to a single ticket on a roll of movie theatre tickets or a single "square" on a roll of paper towels. In other words, a page consists of the paper between two horizontal perforations. In the case of the printer we shall assume these perforations occur every eleven inches.

†This section can be omitted if you communicate with your computer via a typewriter rather than by punch card for input and by line printer for output.

The next question is, what technique can we use to force the character, blank, zero, or one, as desired, into the leading position of each line image which is to be printed?

For the present suppose we limit ourselves to single space printing. For this we will only need to force a blank into the leading position.

You have probably noticed that upon printing a number under control of an I- or F-field, if the numeral is small and does not fill the field, positions to the left of the leading digit print as blanks. This is our clue. A blank will therefore automatically be present at the left end of the first field of a line image if

1. this field is an I- for F-field, and if
2. the field width is wider than necessary for printing the desired number and its sign.

Now if you check the examples used so far in this text, you will find that with a field width of 15 columns, most of the simple numbers we have dealt with are in this category.

For a field coded as

F15.8

what is the largest number which can be printed and still have a blank at the left-most position?

To answer this, interpret the X's in Figure F2-16 as digits.

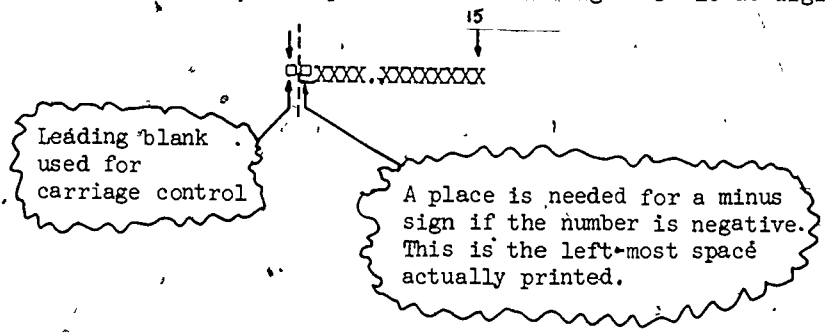


Figure F2-16. Printed form of a leading numeral under F15.8 control

Clearly, a number as large as

± 9999.99999999

can be printed.

For printing integers there is even less problem with a field coded I15.

F2-10 Input and output of alphanumeric data

Computers can store and manipulate alphanumeric data (letters, digits and special characters). This idea was introduced to you earlier in the flow chart text (Figure 2-17). You may have wondered how, in FORTRAN, we can describe the input, output, and manipulation of such data.

Since FORTRAN was originally designed to describe algorithms for computation with numerical data, it should not surprise you that its scope is limited when it comes to describing processes for handling alphanumeric information.

Nonetheless, we can perform a few elementary alphanumeric processes via some FORTRAN statements. These actions, though simple, will permit us to carry out some surprisingly complicated information processing. We can easily describe in FORTRAN how to input alphanumeric data (to memory), as we shall see shortly. Assignment statements can be used to move alphanumeric data to new locations in memory. Thus, if a variable T, for instance, has acquired an alphanumeric value via input, then subsequent execution of the statement,

```
S = T
```

would assign to S the alphanumeric value of T. Of course, we cannot perform any meaningful arithmetic operations on alphanumeric data; but, as we shall see in Chapter 5, it will be possible to compare two alphanumeric quantities for equality. Finally, we can express in FORTRAN how to print out alphanumeric data from memory.

Storage of alphanumeric data

Up to now we imagined FORTRAN permitting only two classes of variables; real and integer numbers. But alphanumeric data is neither real nor integer. How, then, can alphanumeric data be stored in memory? FORTRAN II processors permit you to use storage locations that are assigned to numeric variables for other purposes.

We shall in the following discussions imagine a computer whose word length in bits is such that one word stores six whole characters. (If you have forgotten how the characters may be coded, each as a group of binary bits, you should review Section 1-4.)

Example

Suppose the instructor who posed the original problem to compute

$$D = \sqrt{A^2 + B^2 + C^2}$$

has since been introduced to computer programming and has some inkling as to the power of computers. He now poses the problem this way:

"Imagine that several different sophomore geometry students have given you values of A, B, and C corresponding to the edges of a rectangular prism. You are to compute for them the distance D, which represents the length of a diagonal, according to the formula suggested in Figure 2-1. Write a program which prints values for A, B, C, and the computed value for D, and then also prints for identification purposes the full name of the student, his room number, and seat location; like

BOB JOHNSON, ROOM 342, ROW C, SEAT 4."

If identification of this sort were punched on a card, it might look like that shown in Figure F2-17.

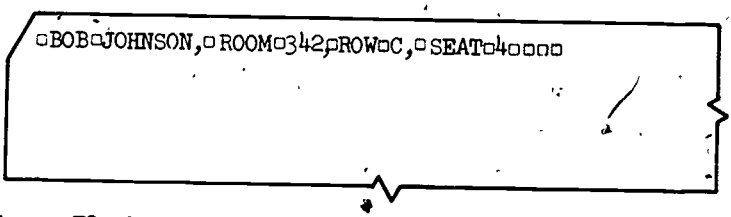


Figure F2-17. Alphanumeric identification (I.D.) card

With respect to the structure of the algorithm first flow charted in Figure 2-1, little has changed when we add, as we now must, the steps for reading and printing the alphanumeric identification, as shown in Figure F2-18.

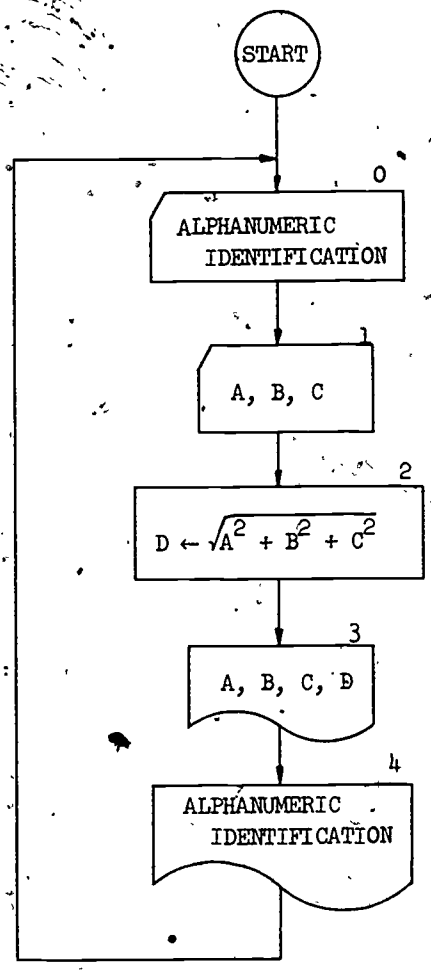


Figure F2-18. Flow chart with provision for reading and printing alphanumeric identification

The question is, how do we write Boxes 0 and 4 in FORTRAN?

To write Box 0 in FORTRAN we must first imagine the information on the I.D. card divided into six-column fields. Now, recall that we are supposing each group of six characters can be stored in a single memory word, say, one that is normally used for storing real numbers. (Any other character packing capacity would not change what we are saying here in principle.) Then, any real variables we choose--you name it--such as R, S, T, U, V, W and X or R1, R2, R3, etc., will be suitable as elements of the input list. So, in effect,

we want to think of alphanumeric information that has been punched in the card as being partitioned into groups of six characters each, for storing into locations associated with some group of variables. This idea is suggested in Figure F2-19.

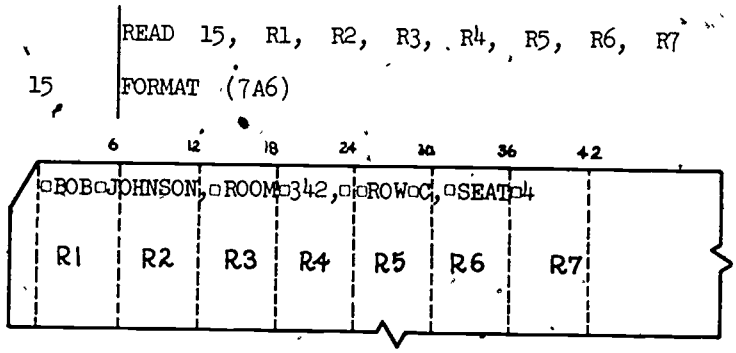
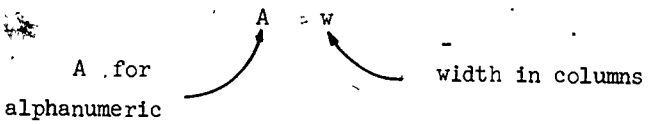


Figure F2-19. Illustrating use of Alphabetic, or A-field codes

The internal representation of alphanumeric data follows a scheme, which is different from either the integer or the floating point scheme. So, a special conversion procedure is needed and hence a special field code is also needed to call for this conversion. The field code that is used has the form



During execution of a READ statement, if an A-field code like A6, is encountered when consulting the referenced format, the characters matched to this code are converted for storage in the alphanumeric scheme. So, executing a READ statement when linked to a format, as one shown in Figure F2-19, will prove suitable for inputting alphanumeric data.

Many other possibilities, some even more subtle, would also work. We shall consider one of these alternatives presently. Right now, we want to get an idea how we might write Box 4, the output box, in FORTRAN. We can take advantage of the natural symmetry of input with output. The statements

```
      PRINT 15, R1, R2, R3, R4, R5, R6, R7
15    FORMAT (7A6)
```

will be suitable.

In executing this PRINT statement the A6 fields encountered in the consulted format will force the reverse conversion of the alphanumeric data from the internal scheme, to the external scheme and printing will then be accomplished in groups of six characters per field.

The complete program requested by the instructor might appear as shown in Figure F2-20.

Label	Statement or declaration
C	EVALUATION OF D
C	EACH SET OF DATA, A, B, AND C IS PUNCHED
C	ON ONE CARD, BUT IS PRECEDED BY A CARD
C	CONTAINING ALPHANUMERIC IDENTIFICATION
1	READ 15, R1, R2, R3, R4, R5, R6, R7
15	FORMAT (7A6)
	READ 101, A, B, C
101	FORMAT (4F15.8)
	D = SQRTF(A**2 + B**2 + C**2)
	PRINT 101, A, B, C, D
	PRINT 15, R1, R2, R3, R4, R5, R6, R7
	GO TO 1
	END

Figure F2-20. Program showing facility for alphanumeric input and output

In examining Figure F2-19, you may have wondered about the form of the information on the I.D. card. Except for the fact that the information will be grouped into "words" of six columns each for storage, the name, room number and seat location is in free form. That is, the name, room number, etc., don't have to be punched in certain fields on the card. Moreover, with the exception of a blank in Column 1, which will be explained in a moment, no special attention is given to the number of blank columns between words.

How then would our program (Figure F2-20) handle the I.D. card for Algernon Thistlewhaite whose I.D. card is shown in Figure F2-21?

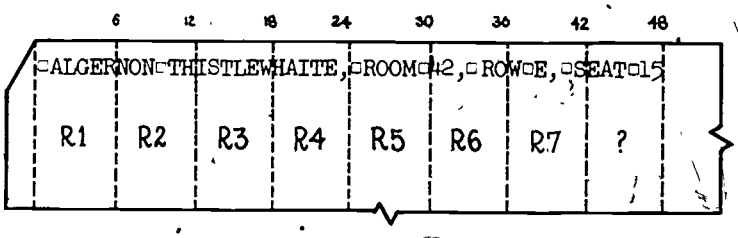


Figure F2-21. I.D. card with a longer name

The last six characters on this card will not be read by the computer because provision was made for storing information from only the first forty-two columns of each I.D. card (7A6). If we want to be safe, we had better make provision to store all 80 columns of the card. This means 13 full words of six characters each plus a partially filled word containing the last two characters from the card (Columns 79 and 80). In other words, we need an input list of 14 items with a governing format code like

13A6, A2

To revise our program so it will both read and print the full I.D. card, we can replace the affected READ, PRINT, and FORMAT statements with those shown here:

```
      |  READ 15, R1, R2, R3, R4, R5, R6, R7, R8, R9,
     1|      R10, R11, R12, R13, R14
   15 |  FORMAT (13A6,A2)
      |
      |
      |  PRINT 15, R1, R2, R3, R4, R5, R6, R7, R8,
     1|       R9, R10, R11, R12, R13, R14
```

Now to clear up the question of Column 1 on the I.D. card, and why we suggested it should be blank. The mystery is easily solved if we recall our discussion of printer carriage control from Section F2-9. If our output comes from a line printer we need to ensure that the first character of an output line is shunted-off and used as a carriage control code instead of being printed. By insisting on a blank in Column 1 of each I.D. card, we guarantee single spacing of each printed line of I.D. information. If you use a typewriter for output--no harm--the first position of the typed line will be blank.

Chapter F3

BRANCHING AND SUBSCRIPTED VARIABLES

F3-1 Conditional statements

In Section 3-1 of your flow chart text you studied techniques for branching by means of a two-exit condition box. In Section 3-3 you will study multi-exit condition boxes. Branching instructions may be written in FORTRAN by means of an IF statement. The FORTRAN IF statement can be used as either a two-exit or three-exit conditional statement. In practice, however, the IF statement is most frequently used for two-way branching and less frequently for three-way branching. Figure F3-1 shows both uses of the IF statement.

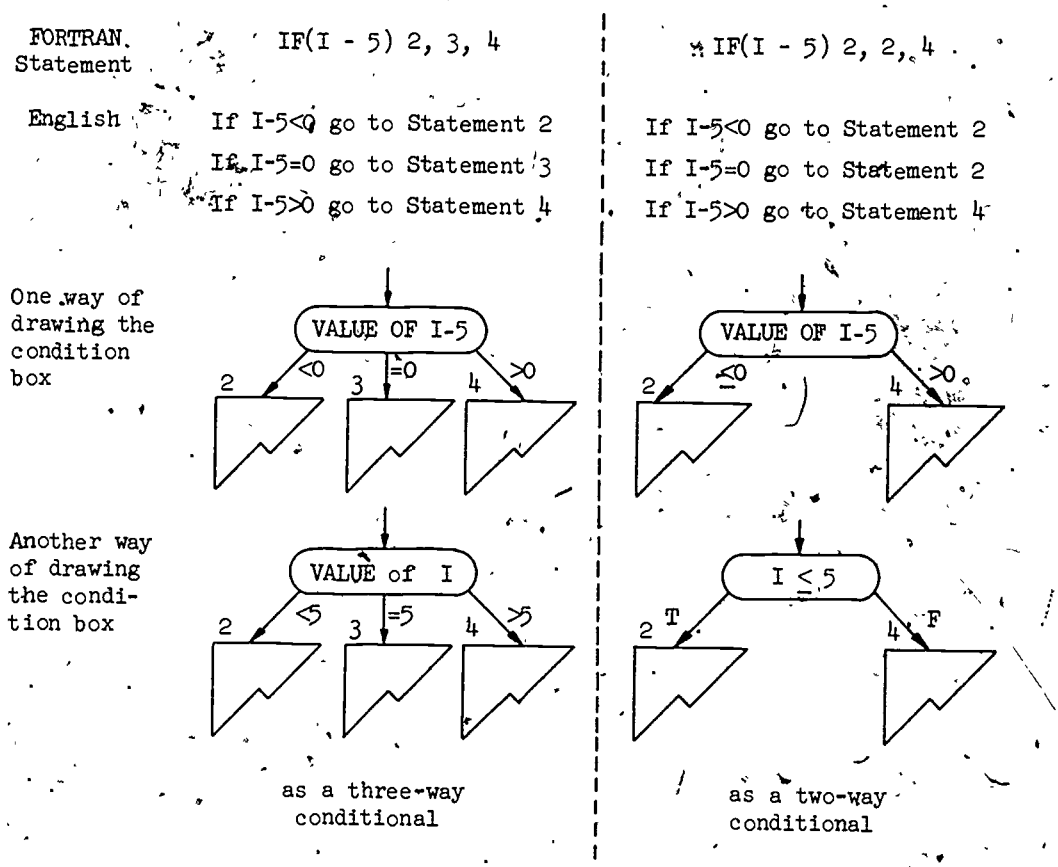


Figure F3-1. Examples of FORTRAN IF statements

An IF statement consists of the word

IF

followed by a FORTRAN arithmetic expression enclosed in parentheses, for example,

(I - 5)

followed by three statement numbers, for example,

2, 3, 4

The complete statement is

IF(I - 5) 2, 3, 4

It tells the computer to evaluate the expression

I - 5

and choose among the following alternatives:

if I - 5 < 0 go to Statement Number 2;
if I - 5 = 0 go to Statement Number 3;
if I - 5 > 0 go to Statement Number 4.

Thus the IF statement is a test of whether the value of a given expression is negative, zero, or positive. The three statement numbers specify where to go in each case.

The second example in Figure F3-1 shows how the three-exit IF statement can be used as a two-way branch. There is really no requirement that the three specified statement numbers in the IF statement be all different. Two of them can be the same, and when two are the same, we get a two-way branch. Thus, we can test whether the value of I - 5 is non-positive ($I - 5 \leq 0$) by writing

IF(I - 5) 2, 2, 4

It is important to observe that this gives the same effect as if we ask whether $I \leq 5$ is true or false.

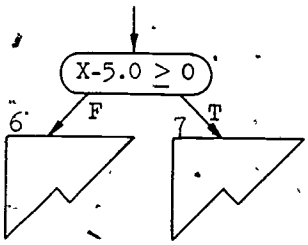
Examples: Consider the statement IF(X - 5.0) 6, 7, 7

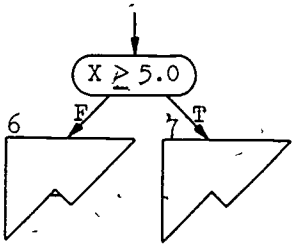
1. What will be the next statement executed if X = 4.7? (Answer: Statement Number 6, since 4.7 - 5.0 = -0.3 is negative.)
2. What will be the next statement executed if X = 5.4? (Answer: Statement Number 7, since 5.4 - 5.0 = +0.4 is positive.)

3. What will be the next statement executed if X = 5.0? (Answer: Statement Number 7, since 5.0 - 5.0 equals zero.)

4. Based on the above IF statement, complete the following: If X ___ 5.0, the next statement executed will be Statement Number 6; but if X ___ 5.0, the next statement executed will be Statement Number 7. (Answer: If $X < 5.0$, the next statement executed will be Statement Number 6; but if $X \geq 5.0$, the next statement executed will be Statement Number 7.)

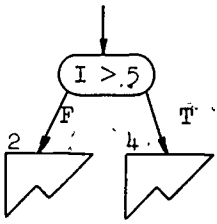
5. Draw a two-exit condition box with exit arrows labelled "T" and "F" corresponding to the IF statement given above. Give two alternate solutions. Answer:





Which is easier to understand at a glance? Answer: The author thinks it will be the one on the right in most instances.

6. Refer to the lower right hand condition box in Figure F3-1. Redraw the box without changing its intent so that the T exit now leads to Box 4 while the F exit leads to Box 2. Answer:



How has the corresponding IF statement changed? Answer: It hasn't.

Exercises F3-1 Set A

In Exercises 1 - 5 use the following IF statement:

IF(K - 8) 20, 30, 20

1. What will be the next statement executed if K = 4?
2. What will be the next statement executed if K = 8?
3. What will be the next statement executed if K = 9?
4. Complete the following: The next statement executed will be Statement Number 30 if and only if K ___ 8.
5. Complete the following: The next statement executed will be Statement Number 20 if and only if K ___ 8.
6. Write an IF statement that will go to Statement Number 9 if X is positive and to Statement Number 10 if X is non-positive.

In Exercises 7 - 12, transform each of the given equations or inequalities into an equivalent form having "0" as one member:

7. $K > 7$ (Answer: $K - 7 > 0$)
8. $X \geq 8.4$
9. $Y < 4.2$
10. $A \leq B$
11. $X = Y$
12. $X + 5 \neq Y$

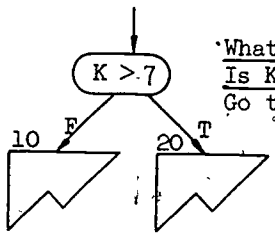
Converting to a three-way algebraic test from any one of six 2-way relations.

Figure F3-2 shows how one might go about writing IF statements corresponding to two-exit condition boxes involving one of the six relational symbols

$$< \quad > \quad \leq \quad \geq \quad = \quad \neq$$

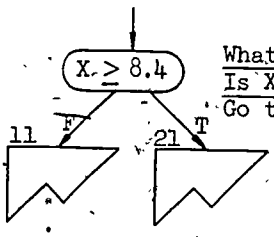
In every case illustrated we convert a relation whose form is A relational symbol B, to a difference whose form is A - B, where A and B are the expressions in the condition box that lie on either side of the relational symbol.

Flow Chart | Reasoning | Fortran



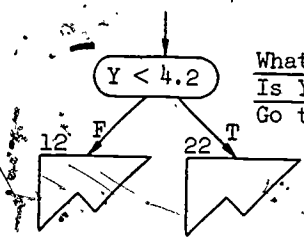
What is True?	K-7 < 0	K-7 = 0	K-7 > 0
Is K-7 > 0 T or F	F	F	T
Go to Statement #	10	10	20

IF(K-7)10,10,20



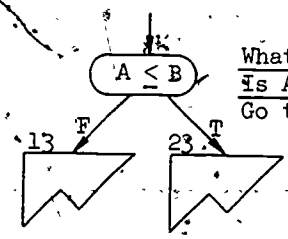
What is True?	X-8.4< 0	X-8.4= 0	X-8.4> 0
Is X-8.4≥0 T or F	F	T	T
Go to Statement #	11	11	21

IF(X-8.4)11,11,21



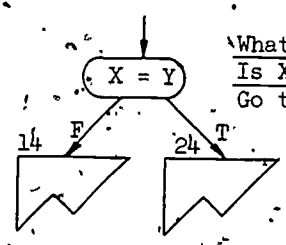
What is True?	Y-4.2<0	Y-4.2 =0	Y-4.2>0
Is Y-4.2<0 T or F	T	F	F
Go to Statement #	22	12	12

IF(Y-4.2)22,12,12



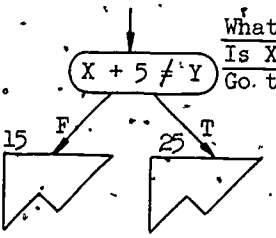
What is True?	A-B < 0	A-B = 0	A-B > 0
Is A-B≤0 T or F	T	T	F
Go to Statement #	23	23	13

IF(A-B)23,23,13



What is True?	X-Y < 0	X-Y = 0	X-Y > 0
Is X-Y=0 T or F	F	T	F
Go to Statement #	14	24	14

IF(X-Y)14,24,14



What is True?	X+5-Y <0	X+5-Y = 0	X+5-Y >0
Is X+5-Y≠0 T or F	T	F	T
Go to Statement #	25	15	25

IF(X+5-Y)25,15,25

Figure F3-2. Six variations on the IF theme for two branches

Table F3-1 shows the six patterns possible in using the IF statement as a two-way branch.

Table F3-1

The six patterns for statement numbers in two-way IF's
going to Statement 1 if the relation is true.
going to Statement 2 if the relation is false

Form of Relation	IF Statement
$A > B$	IF(A - B) 2, 2, 1
$A \geq B$	IF(A - B) 2, 1, 1
$A < B$	IF(A - B) 1, 2, 2
$A \leq B$	IF(A - B) 1, 1, 2
$A = B$	IF(A - B) 2, 1, 2
$A \neq B$	IF(A - B) 1, 2, 1

Among the important things to remember about the form of the IF statement are the following:

(a) The arithmetic expression to be tested is enclosed in parentheses following "IF".

(b) The three exits are negative, zero, and positive in that order.

(c) A statement number must be written for each of the three exits in the order indicated in (b), even though one of the conditions may never occur.

(d) There must be commas between the statement numbers but none between the right parenthesis and the first statement number.

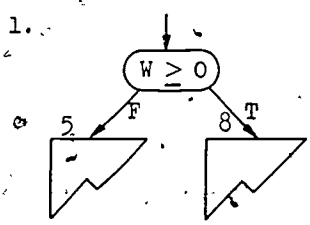
You may recall the Ruritanian Postal Regulations problem. Compare the executable steps of the following FORTRAN program with the flow chart in Figure 3-1 and verify that the two are equivalent.

Label	Statement or declaration
1	READ 100, N, A, B, C
	D = SQRTF(A**2 + B**2 + C**2)
	IF(D - 29.0) 4, 4, 1
4	PRINT 101, N
	GO TO 1
100	FORMAT (I6, 3F6.2)
101	FORMAT (I6)
	END

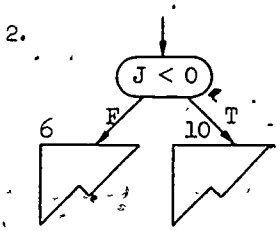
Exercises F3-1 Set B

Write a FORTRAN IF statement for each of the following condition boxes:

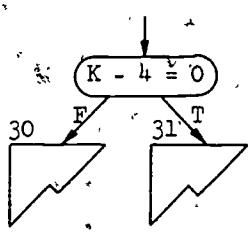
1.



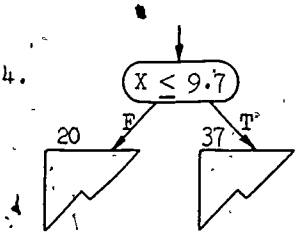
2.



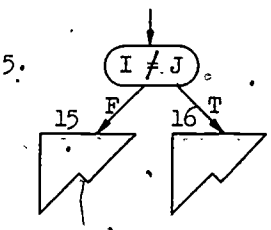
3.



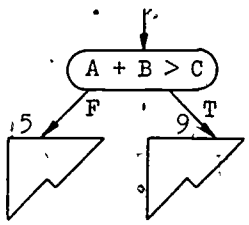
4.



5.

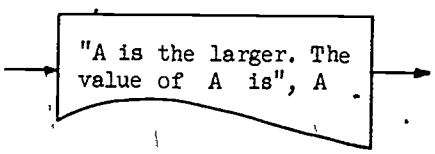


6.



Identifying remarks in FORTRAN output

In Section 3-1 you worked with output boxes which included identifying remarks as elements of the output list. For example:



Note that an identifying remark is a constant, more precisely an alpha-numberic constant, as defined in Section 2-6 of the Main Text. There is no way that the remark itself can be changed in value as a result of carrying out the steps of the flow chart.

In FORTRAN an identifying remark such as

"A is the larger. The value of A is"

does not appear directly as an item in the output list. Constants are not legal elements of an output list. Instead, we insert the remark in the FORMAT declaration. Thus far, you have encountered the following types of field specifications in FORMAT declarations: I, F, and A. To these we now add the H-field.

The H-field (Hollerith field) may contain the text of any remark, message or heading, i.e., any alphanumeric constant. Instead of enclosing the text in quotation marks, we precede it by the letter H (identifying the type of field) and precede that letter by an unsigned integer indicating the length (number of characters) of the text that immediately follows the letter H. Figure F3-3 gives several illustrations.

Flow Chart	FORTRAN and the resulting printed line
J, "IS THE LARGER"	PRINT 100, J 100 FORMAT (I6; 14H□IS□THE□LARGER) XXXXXX IS THE LARGER
"THE LARGER IS J=", J	PRINT 102, J 102 FORMAT (19H□THE□LARGER□IS□J□=□, I6) THE LARGER IS J = XXXXXX
X, "IS GREATER. THAN", Y	PRINT 101, X, Y 101 FORMAT (F10.3, 17H□IS□GREATER□THAN□, F10.3) XXXXXX.XXX IS GREATER THAN XXXXXX.XXX
"THE LARGER IS J=", J	PRINT 103, J 103 FORMAT (19H1THE□LARGER□IS□J□=□, I6) THE LARGER IS J = XXXXXX

Figure F3-3. Use of H-fields for identifying remarks

Note the difference between FORMAT statements 102 and 103. In 102 the first character of output for the line is a blank, while in 103 the first character is the numeral "1". If you recall the discussion of printer carriage control in Chapter 2, you will remember that the blank initial character calls for a single space before printing, while the "1" calls for a skip to the top of the next page before printing. If you begin a line with an H-field, and if you do your printing on a line printer instead of a typewriter, you must remember to put in this first carriage control character for each line. If the first field would not normally be an H-field and you want to be certain that the leading character will signal the carriage control of your choice, you can use a one-character H-field at the beginning of a FORMAT statement.

Carriage Control Desired	Initial H-field
single space	1H□
double space	1H0
skip to top of next page	1H1

For example, we might rewrite the first format in Figure F3-3 as

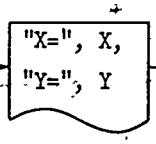
```
100  FORMAT (1H□, I6, 14H□IS□THE□LARGER)
```

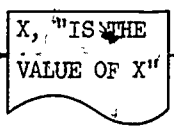
in order to guarantee a blank as the leading character for use in carriage control.

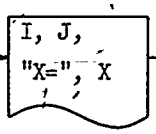
Exercises F3-1 Set C

Write FORTRAN PRINT and FORMAT statements for each of the following exercises. Assume that flow chart variables beginning with any letter I through N are integer variables (and use I6 fields). Assume that all other variables are floating point variables (and use F10.3 fields).

Insert a one-character H-field at the beginning of each output record to provide carriage control on the assumption you are using a line printer for output. In Exercises 1 and 2, single space before printing. In 3 double space. In 4 skip to the next page before printing. The choice of Format numbers is up to you.

1. 

2. 

3. 

4. 

Example 1

The second form of the flow chart in Figure 3-5(b) could be programmed in FORTRAN as follows:

Label	Statement or Declaration
	READ 100, A, B, C
	ALRGST = A
	IF(ALRGST - B) 4, 5, 5
4	ALRGST = B
5	IF(ALRGST - C) 6, 7, 7
6	ALRGST = C
7	PRINT 101, ALRGST
	STOP
100	FORMAT (3F10.3)
101	FORMAT (22H□THE□LARGEST□VALUE□IS□, F10.3)
	END

Example 2

The flow chart in Figure 3-6 could be programmed in FORTRAN as follows:

L bel	Statement or Declaration
	READ 100, A, B, C
	IF(A - B) 4, 3, 3
3	IF(A - C) 6, 5, 5
5	PRINT 101, A
8	STOP
6	PRINT 102, C
	GO TO 8
4	IF(B - C) 6, 7, 7
7	PRINT 103, B
	GO TO 8
100	FORMAT (3F10.3)
101	FORMAT (32H□A□IS□THE□LARGEST.□ITS□VALUE□IS□, F10.3)
102	FORMAT (32H□C□IS□THE□LARGEST.□ITS□VALUE□IS□, F10.3)
103	FORMAT (32H□B□IS□THE□LARGEST.□ITS□VALUE□IS□, F10.3)
	END

Example 3

The tallying problem in Figure 3-7 could be programmed in FORTRAN as follows:

Label	Statement or Declaration
	READ 100, N
	KCOUNT = 0
	LOW = 0
	MID = 0
	KHIGH = 0
3	READ 101, T
	IF(T - 50.0) 9, 9, 5
9	LOW = LOW + 1
	GO TO 7
5	IF(T - 80.0) 10, 10, 6
10	MID = MID + 1
	GO TO 7
6	KHIGH = KHIGH + 1
7	KCOUNT = KCOUNT + 1
	IF(KCOUNT - N) 3,11, 11
11	PRINT 102, KCOUNT, LOW, MID, KHIGH
	STOP
100	FORMAT (I6)
101	FORMAT (F8.3)
102	FORMAT (43H□VALUES□OF□KCOUNT,□LOW,□MID,□AND□KHIGH
1	ARE□, 4I6)
	END

Exercises F3-1 Set D

1 - 5. Refer to Exercises 7 - 11, Section 3-1 Set A in your flow chart text. For each of these five exercises you prepared a flow chart of a simple algorithm. Now write a FORTRAN program corresponding to each chart. Choose statement numbers where needed, to correspond to the box numbers used in the flow charts. Specifically, let 1 be the label for a statement that corresponds to Box 1, 2 for Box 2, etc.

Exercises F3-1 Set E

1 - 6. Refer to Exercises 1- 6, Section 3-1 Set B. For each of these six exercises you prepared a flow chart of a simple summing algorithm. Now write a FORTRAN program corresponding to each flow chart. Choose statement numbers, where needed, to correspond to the box numbers used in the flow charts. Assume a suitable input format code for the data is F10.5.

F3-2 Auxiliary variables

The use of auxiliary variables in FORTRAN programs mirrors what you have already learned in the flow chart text.

Exercises F3-2 Set A

1 - 5. Refer to Problems 2 through 6, Section 3-2 Set A in your flow chart text. Write FORTRAN programs for each of the flow charts you have constructed for these exercises. Choose statement numbers which correspond to the box numbers used in the flow charts.

You might like to see how the flow chart for the Euclidean Algorithm given in Figure 3-14 might work out in FORTRAN:

Label	Statement or Declaration
	READ 100, KA, KB
	PRINT 101, KA, KB
	IF(KA-KB) 5, 5, 4
4	KR = KA
	KA = KB
	KB = KR
5	IF(KA) 6, 7, 6
6	KR = KB - KB/KA*KA
	KB = KA
	KA = KR
	GO TO 5
7	PRINT 102, KB
	STOP
100	FORMAT(2I6)
101	FORMAT(12H□THE□GCD□OF□, I6, 5H□AND□, I6, 4H□IS.)
102	FORMAT(1H□, I6)
	END

Exercise F3-2 Set B

Write a FORTRAN program corresponding to the flow chart that you constructed in Problem 2 of Exercise 3-2 Set B of your flow chart text.

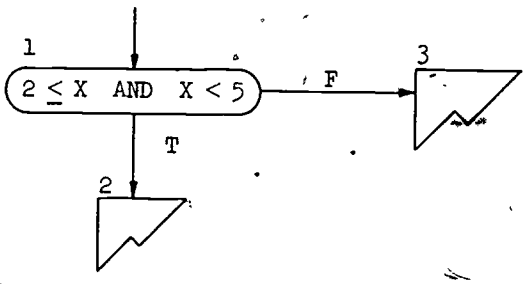
Exercise F3-2 Set C

1 - 8. Refer to Problems 1 through 8, Section 3-2 Set C in your flow chart text. For each of these eight exercises you have prepared a flow chart of a simple algorithm related to coordinates of points on a straight line. Now write a FORTRAN program corresponding to each flow chart. Choose statement numbers which correspond to the box numbers used in the flow charts.

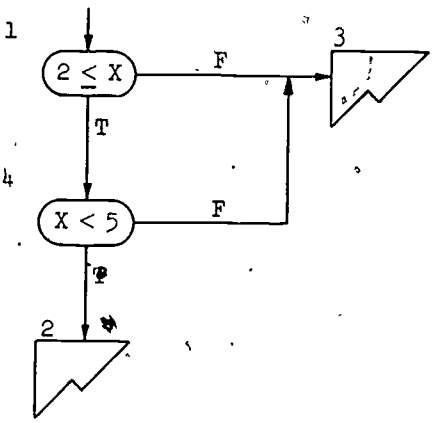
F3-3 Compound conditions and multiple branching

Writing compound conditions using IF statements

In Section 3-3 you encountered condition boxes involving more than one decision, e.g.,



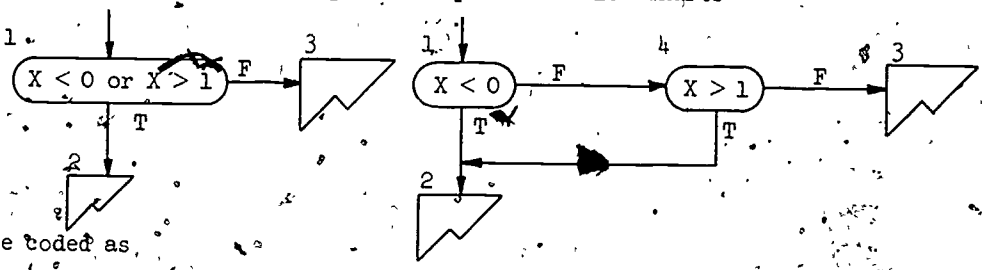
and you saw that this single decision box was equivalent to a pair of boxes:



This pair of boxes can be coded in FORTRAN as

```
    |  IF(2.0 - X) 4, 4, 3
  4 |  IF(X - 5.0) 2, 3, 3
```

Similarly, the following pair of equivalent flow charts



can be coded as

```
    |  IF(X) 2, 4, 4
  4 |  IF(X - 1) 3, 3, 2
```

Exercises F3-3. Set A

1 - 7. Refer to Exercises 1 - 7, Section 3-3 in your flow chart text. For each of these exercises write FORTRAN statements equivalent to the flow charts you prepared. For example, the flow chart example would probably be coded in FORTRAN as:

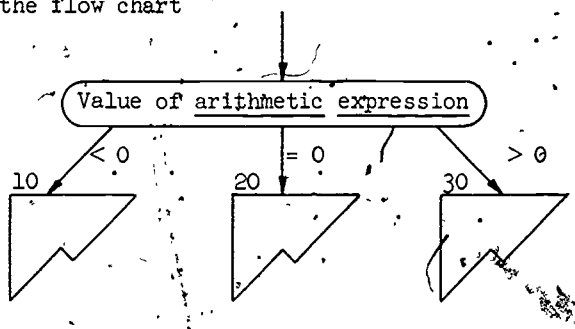
Label	Statement or Declaration
	IF(X1 - X2) 2, 30, 30
2	IF(P - G) 3, 3, 20
3	IF(T - S) 30, 20, 30

Writing multiple branching instructions using IF statements

In Section F3-1 you learned that the FORTRAN IF statement was a kind of three-exit (i.e., three-branch) conditional statement. That is

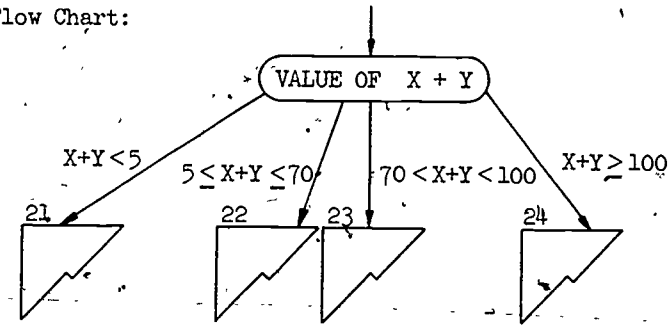
IF(arithmetic expression) 10, 20, 30

is equivalent to the flow chart



In certain situations you will find it useful to use the IF statement this way--as a three-way branch. Frequently, however, when you need to do a three-way branch, you will find that your problem does not neatly fit the pattern of a single IF statement. Usually you will find it necessary to write two or more IF statements corresponding to a multiple branch condition box.

Example: Flow Chart:

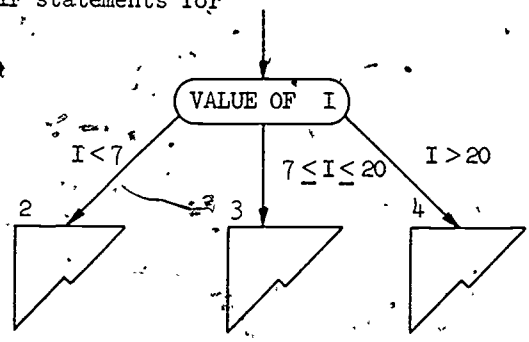


FORTRAN:

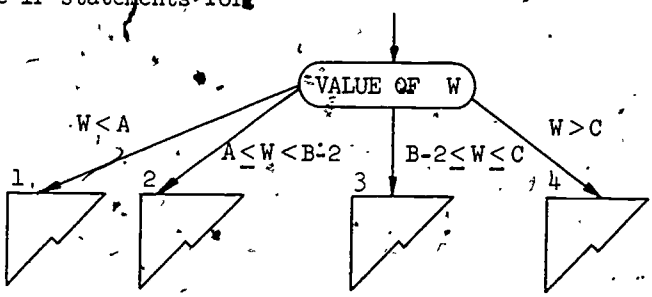
```
    IF(X + Y - 5.0) 21, 40, 40
40  IF(X + Y - 70.0) 22, 22, 41
41  IF(X + Y - 100.0) 23, 24, 24
```

Exercises F3-3 Set B

1. Write FORTRAN IF statements for



2. Write IF statements for

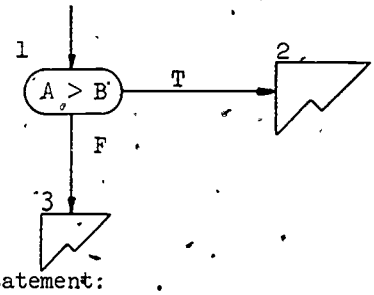


3. Write a FORTRAN program corresponding to the flow chart you prepared for Problem 10, Exercises 3-3.

4. Write a FORTRAN program corresponding to the new flow chart which you prepared for the carnival wheel problem as the answer to Problem 11, Exercises 3-3.

†F3-4 Logical expressions

It would be nice if it were possible in FORTRAN to code the two-way condition boxes of our flow charts in a more straightforward fashion. To do this one would need to add relational operators (and symbols for them) to the set of allowed operations. For example, it would be nice if a condition box like



could be coded as the single statement:

meaning: go to 2 if true
meaning: go to 3 if false

```
IF(A > B) 2, 3
```

or possibly in some other way, like

```
      IF(A > B)  GO TO  2
     3
```

where here the IF statement shows what to do if the condition inside the parentheses is true, implying that we should otherwise proceed to the next statement if false.

You may have surmised or have learned from other sources that FORTRAN is more a set of similar dialects than a single language. The dialect FORTRAN II which we are studying, has no relational operators and hence it has no "logical" IF statement like the ones suggested above. It has only the "arithmetic" IF statement. As a matter of fact, there are FORTRAN dialects, notably a group referred to as FORTRAN IV which do exhibit logical IF statements like those shown above. You can guess that coding a flow chart into one of these dialects of FORTRAN is an easier job.

---

†This section can be skipped without loss of continuity.

F3-5 Subscripted variables

Representation of subscripted variables in FORTRAN

Figure F3-4 shows how subscripted variables are represented in FORTRAN.

Flow Chart Form	FORTRAN Form
X_4	X(4)
X_N	X(N)
B_{I+2}	B(I+2)
JOE_{MOE-6}	JOE(MOE-6)
$CHAR_{5\times I+4}$	CHAR(5*I+4)
$ALPHA_{22\times J+15}$	ALPHA(22*J+15)
$BETA_{17\times JAY-9}$	BETA(17*JAY-9)

Figure F3-4. Representation of subscripted variables in FORTRAN

As you can see from the figure, subscripted variables are represented in FORTRAN by enclosing the subscript in parentheses and writing it following the variable to which the subscript is affixed. This is another example of a notation that enables FORTRAN code to be written "on the line." Other examples you have seen include "A**2" for "A^2" and "SQRTF(X)" for $\sqrt{X}$. Since implied multiplication (omission of the multiplication sign) is strictly forbidden, there is no danger than Y_N will be confused with $Y \times N$--the latter is written using an asterisk between the variables. The only source of confusion in the notation is that a similar notation is used for function references. You will see later in this section how this potential ambiguity is resolved.

In your flow chart text, just about any integer-valued arithmetic expression was permitted as a subscript. In the version of FORTRAN you are studying, the rules are much more strict:

(a) A subscript may take on only positive integer values (zero, negative numbers, and numbers other than integers may not occur as subscript values).

(b) The most general forms permitted are

constant1 * variable + constant2

and

constant1 * variable - constant2

where constant1 and constant2 are unsigned integer constants and variable is a non-subscripted integer variable. Other permitted forms are

constant
variable
variable + constant
variable - constant
constant * variable

where constant is again an unsigned integer constant and variable is also of non-subscripted integer type.

Exercises F3-5 Set A

Examine each of the following flow chart subscripts. If the subscript is legal in FORTRAN, write the subscripted variable in FORTRAN form. Otherwise point out the defect.

1. X_5
2. X_J
3. $CHAR_i$
4. B_{I+2}
5. I_X
6. $Z_{5\times I+2}$
7. Z_{I+J}
8. $ALPHA_{-4}$
9. XYZ_{IXY}
10. $ABC_{-5\times J}$

Allocation of memory storage for arrays

As you know, the computer must have a storage location available corresponding to each variable in a given FORTRAN program. If all variables were of the form

A, X, CHAR, X_4, etc.

this would be a simple problem indeed. The processor could merely assign a

storage location to each variable occurring in your program.

But what about X_I? How can the processor know for what values of I to assign storage locations? It cannot tell merely by looking at the occurrences of X_I in your program. But if it were to wait until the program was being executed, it might find that it needed locations for X_1, X_2, X_3, ..., X_{25} and had assigned locations only for X_1, X_2, ..., X_{10}. Since storage of arrays such as X_1, X_2, ..., X_{25} in consecutive locations is of overwhelming importance in efficiency of program execution in most computers, we would like to have advance knowledge of the range of values possible for a subscript before we start executing the program in which that subscript occurs. More precisely, we require at least knowledge of the maximum range so that we will allow enough locations.

In FORTRAN this problem is solved by means of a DIMENSION declaration. It is used to specify an upper limit for the subscript value for a particular subscripted variable. Figure F3-5 shows the form and use of the DIMENSION declaration. These declarations must precede all executable statements of a program.

Now we can explain how the computer can distinguish between such things as CHAR(J) meaning the J-th element of the array CHAR and CHAR(J) meaning a function reference consisting of the function name CHAR followed by the argument expression "J". If a variable CHAR appears in a DIMENSION declaration, then a subsequent occurrence of CHAR(J) will be interpreted as a subscripted variable. If CHAR does not appear in a DIMENSION declaration, then CHAR(J) will be interpreted as a reference to a function named CHAR. (A common source of error in writing FORTRAN programs is to omit a necessary DIMENSION declaration. With this omission the processor will dutifully interpret all occurrences of CHAR(J) as function references. The result is usually a nasty but misleading error message telling you about your misuse of the function CHAR.)

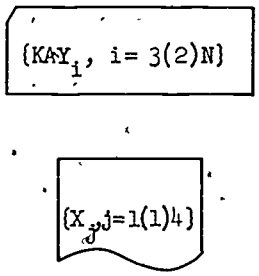
Declaration	Resulting Storage Allocation
DIMENSION X(5)	five locations: one each for X_1, X_2, X_3, X_4, and X_5
DIMENSION CHAR(100)	one hundred locations: one each for $CHAR_1$, $CHAR_2$, ..., $CHAR_{100}$
DIMENSION X(3), Y(4)	one location each for X_1, X_2, X_3, Y_1, Y_2, Y_3, and Y_4
DIMENSION I(2)	two locations: one for I_1 and one for I_2

Figure F3-5. Storage allocations for FORTRAN DIMENSION declarations

Input and output of arrays

The form of the input and output lists for arrays in FORTRAN is quite similar to that in your flow chart language, with a few small, but important differences in detail. Figure F3-6 shows how arrays can be input and output in FORTRAN.

Flow Chart Box



FORTRAN Program

```
Label     Statement or Declaration
          READ 100, N, (KAY(I), I = 3,N,2)
          .....
  100     FORMAT(12I6)

          PRINT 101, (X(J), J = 1, 4)
          .....
  101     FORMAT (1HO, 4F12.6)
```

Figure F3-6. Input and output of arrays in FORTRAN

You should note these points concerning the form of our notation in FORTRAN:

(a) The increment for the subscript is placed after the upper bound instead of between the lower and upper bounds, as practiced in the flow chart notation.

(b) When the increment is equal to 1, it may be omitted.

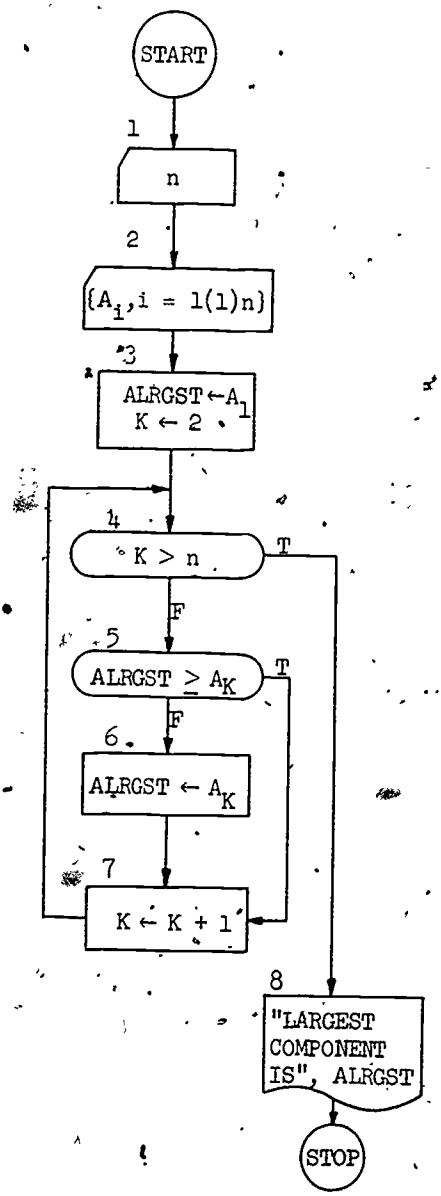
(c) Parentheses rather than braces are used.

(d) The upper bound for the subscript is set off by commas rather than by parentheses.

Example Figure F3-7

Draw a flow chart and write a FORTRAN program to find the largest component of an n-component vector ($n \leq 100$).

Flow Chart



FORTRAN

```
      DIMENSION A(100)
      READ 102, N
      READ 100, (A(I), I = 1,N)
      ALRGST = A(1)
      K = 2
4     IF(K - N) 5, 5, 8
5     IF(ALRGST - A(K)) 6, 7, 7
6     ALRGST = A(K)
7     K = K + 1
      GO TO 4
8     PRINT 101, ALRGST
      STOP
100   FORMAT(4F16.8)
101   FORMAT(22H LARGEST COMPONENT IS, F16.8)
102   FORMAT(I3)
      END
```

Figure F3-7. Find the largest number problem

Exercises F3-5 Set B

In Exercises 1 - 3 write the necessary DIMENSION statements and READ statements to input the arrays indicated.

1. $\{A_i,\ i = 1(1)k\}$ Assume $k \leq 50$

2. $\{B_j,\ j = 5(2)n\}$ Assume $n \leq 125$

3. $\{A_i,\ i = 1(10)n\},\ \{B_i,\ i = 10(2)n\}$ Assume $n \leq 50$

Exercises F3-5 Set C

1. Write a FORTRAN program corresponding to the flow chart in Figure 3-24(b) (The carnival wheel problem using subscripts).

2. Write a FORTRAN program that corresponds to Figure 3-25. Choose your format codes using I5 and F10.5 fields as needed. Assume values for B are punched on data cards with up to seven values per card.

3. Write a FORTRAN program that corresponds to the flow chart you drew for Problem 8 of Section 3-5 Set A. Assume the value of n will never be greater than 50. You will have to write the program so that subscripts for the coefficients range from 1 instead of from zero, because zero subscripts are not allowed in FORTRAN. For format codes use I5 and F10.5 fields as needed for input. Assume polynomial coefficient values are punched in order on data cards having up to five values per card.

Exercises F3-5 Set D

Write a FORTRAN program for the flow chart you drew for Part C of the Exercise 3-5 Set C in the Main Text. Assume by "any size orchestra" we mean one that will never exceed 500 players. Assume the ages of the players are punched as integer data (I5 fields, up to 10 per card).

F3-6 Double subscripts

Representation of double subscripts in FORTRAN

In Section F3-5 you learned that a subscripted variable like

$$x_i$$

could be written in FORTRAN as

X(I)

Now, in Section 3-6, you have been introduced in your flow chart language to doubly-subscripted variables such as

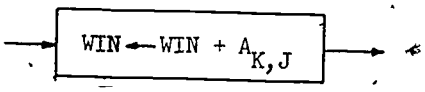
$$x_{i,j}$$

The FORTRAN representation for such a doubly-subscripted variable is just about what you would expect it to be:

X(I,J)

The two subscripts are separated by a comma.

Thus the assignment box.



could be written in FORTRAN as

WIN = WIN + A(K,J)

Allocation of storage for doubly-subscripted arrays

A DIMENSION declaration is required to allocate space for doubly-subscripted variables. For example

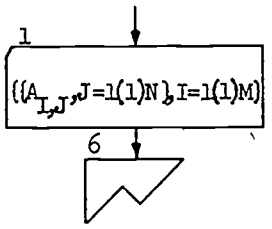
DIMENSION A(3,4)

This particular declaration allocates 12 spaces for a matrix having 3 rows and 4 columns. The subscripted variables of this matrix are:

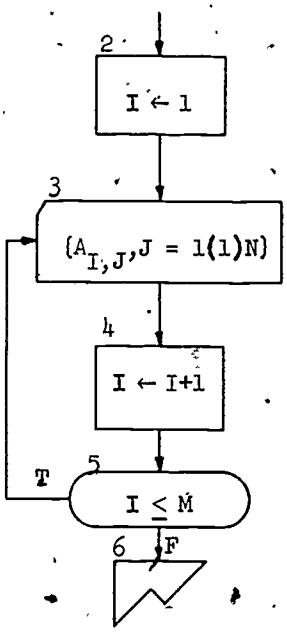
$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$
$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$

Thus the acceptable values for the first subscript are 1, 2, and 3 and for the second subscript they are 1, 2, 3, and 4. No other subscript values have significance with the above DIMENSION declaration.

To input or output an entire matrix, we can employ either of two approaches suggested by the flow charts in Figure F3-9.



Method (a)



Method (b)

Figure F3-9. Two ways to input values of a matrix having M rows and N columns.

The FORTRAN coding for each method is given below for comparison. In each case it is assumed that data are punched on cards, four values per card using F10.5 field codes.

Method (a)

```
      READ 50, ((A(I,J), J = 1, N), I = 1,M)
50    FORMAT (4F10.5)
```

Method (b)

```
      I = 1
3     READ 50, (A(I,J), J = 1,N)
50    FORMAT (4F10.5)
      I = I + 1
      IF(I - M) 3, 3, 6
```

In Method (b), Box 3 is an order to input one entire row of the matrix (the I^{th} row). The other boxes define a range of values for I from 1 to M inclusive, under which Box 3 is repeated. In Method (a) Box 1 is an order to input the entire matrix row-by-row.

It would certainly seem that Method (a) might always be preferred because it is easier to write. In the next chapter we will show situations where the approach used in Method (a) has certain practical advantages.

The FORTRAN program equivalent to the game flow-charted in Figure 3-34 is given below. The one-row-at-a-time approach for input of the 6 × 6 matrix (Method (b)) is chosen.

```
      |  DIMENSION A(6,6)
      |  I = 1
    1 |  READ 100, (A(I,J), J = 1,6)
      |  I = I + 1
      |  IF(I - 6) 1, 1, 2
    2 |  READ 101, K, L
      |  WIN = 0.0
      |  ALOSE = 0.0
      |  I = 1
      |  J = 1
    4 |  IF(J - 6) 5, 5, 6
    5 |  WIN = WIN + A(K,J)
      |  J = J + 1
      |  GO TO 4
    6 |  IF(I - 6) 7, 7, 8
    7 |  ALOSE = ALOSE + A(I,L)
      |  I = I + 1
      |  GO TO 6
    8 |  ANET = WIN - ALOSE
      |  PRINT 102, ANET
      |  STOP
  100 |  FORMAT (6F8.3)
  101 |  FORMAT (2I2)
  102 |  FORMAT (1H□, F8.3)
      |  END
```

Exercises F3-6

1 - 5. Refer to Exercises 1 - 5, Section 3-6. For each of these exercises you have drawn flow charts describing certain "row" or "column operations" on a matrix. For each of these flow charts your job is to write the equivalent FORTRAN statements preceded by the necessary DIMENSION declarations. No FORMAT declarations will be required in your answers.

Chapter F4

LOOPING

F4-1 The DO statement

It should come as no unpleasant surprise that the little wonder box we discovered and called the "iteration box" has an almost perfect parallel in FORTRAN. This shorthand is called the DO statement. An example is shown in Figure F4-1.

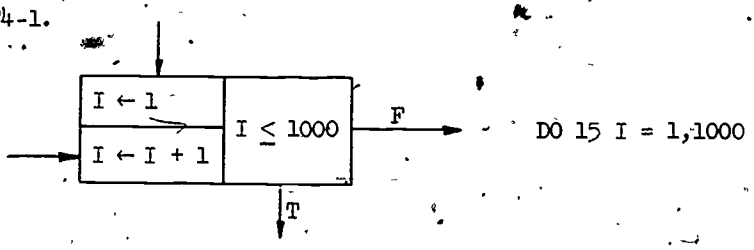


Figure F4-1. The iteration box and an equivalent DO statement

Admittedly, the parallel is not yet obvious, but we promise it will be after we show each in the larger context of a loop.

To display a parallel between a flow chart and a FORTRAN "DO loop" we present Figure F4-2. The algorithm used, you will recognize, is the Fibonacci sequence generator (Figure 4-6).

Our next discussion will focus on the loop counter that is used in a DO statement. It's "personality" will now be described. You may find this description somewhat long and detailed. Don't be discouraged. Read it once and then we will look at some examples.

1. It's always a non-subscripted integer variable, like I, IKE, IBALL, J, M, etc., and never like J(5), INT(K), etc.
2. As in the iteration box, the loop counter must always be given an initial value (≥ 0).

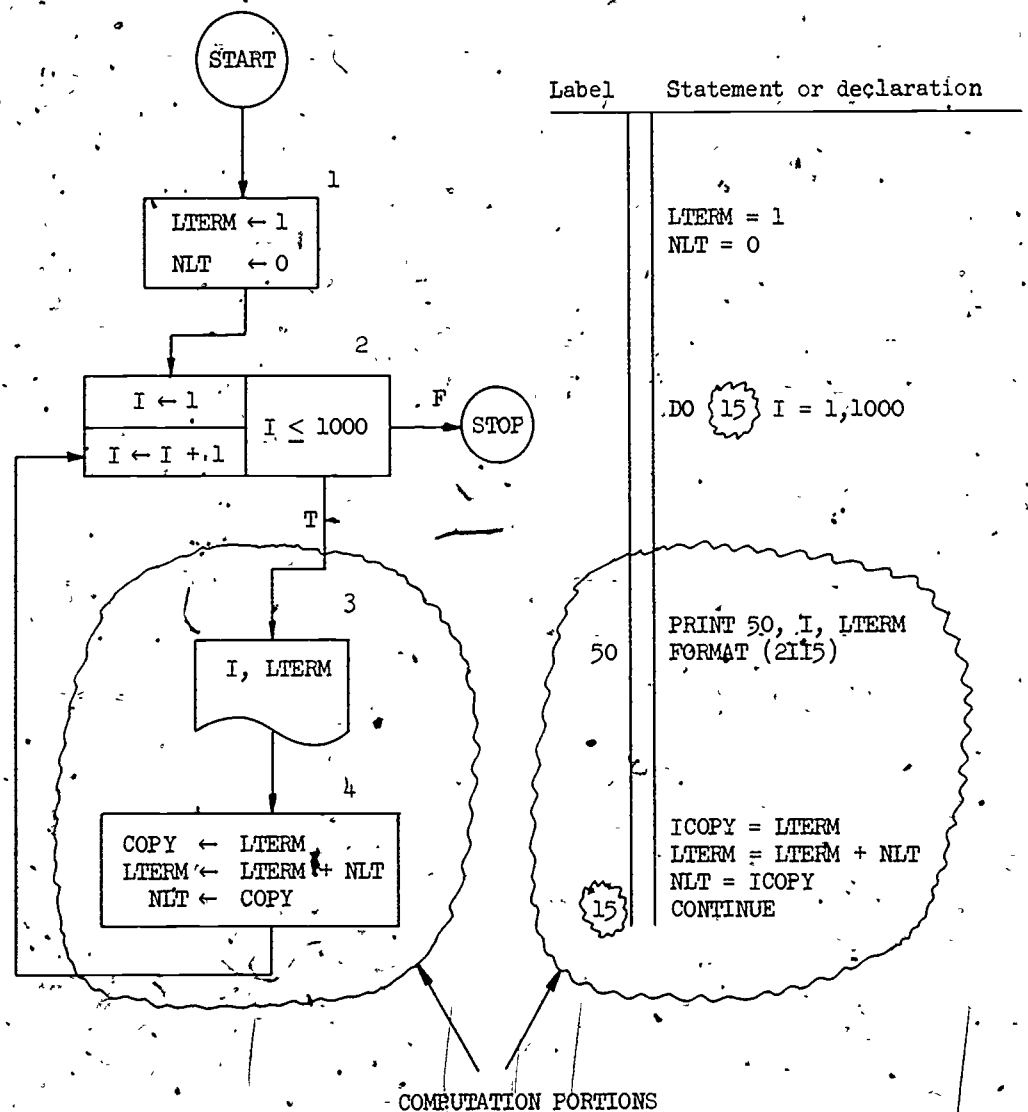


Figure F4-2. Parallel between a flow chart and a FORTRAN loop

The DO statement is an order to repeat a certain task, once for each value of the counter I in the range 1 to 1000 inclusive. The task which is to be repeated begins with the first statement that follows the DO statement and terminates after executing the statement labeled 15. To mark the termination of a "DO loop" we can use a special "dummy" statement or marker for this purpose. It is simply the word

CONTINUE

(Although we frequently use the CONTINUE statement to mark the terminus, an assignment, READ or PRINT statement would also be suitable.) We now see that the statement

DO 15 I = 1,1000

should be read this way:

"Repeatedly execute (or do) all statements which follow this one, down to and including the (CONTINUE) statement that is labeled 15. Do this once for each value of I in the range 1 to 1000."

3. A highest value for the counter's range must always be supplied.

4. An increment must be supplied if it is other than one. We shall see examples presently.

In supplying Items 2, 3, and 4 in the DO statement, we are privileged to give each either as an unsigned integer constant or as a non-subscripted integer variable. The rules are strict on this point. If you forget and use some other form, don't worry, the processor will politely remind you by printing an error message for each poorly formed DO statement.

Examples

1. DO 51 I = 1, N, 2

means: repeat the execution of the immediately following statements through the one labeled 51, once for each value of the loop counter I, in steps of 2 until I > N. Thus if N had a value of 7, the loop would be repeated (4 times) for values of I of 1, 3, 5 and 7. When the counter is incremented again its value exceeds 7 and we exit from the loop by proceeding to execute the first statement that follows the one labeled 51.

2. DO 51 K = 3, 8, 2

means: repeat through 51 once for each value of K until K > 8. K starts with a value of 3 and is incremented each time by 2. The loop is repeated for K = 3, 5 and 7 (three times).

3. DO 51 J = I, L, K

loop counter — for initial value — for loop limit — for counter increment

means: repeat through Statement 51, once for each value of the counter J until J > L. J is given a starting value equal to that currently assigned to I. Prior to each proposed repetition J is incremented by K.

4. DO 51 J = I, L

loop counter — for initial value — for loop limit

means: same as Example 3, except that no increment is supplied. In such an event we (and the computer that must read this statement) always assume the increment is one.

Exercises F4-1

Problems 1 - 5: For each of the flow charts you drew in answering the exercises 1 through 5 Section 4-1, now write an equivalent FORTRAN program. In order to write a complete FORTRAN program, use the information supplied below for each of the exercises.

1. Assume ID is a 6-digit integer, and that values of A, B, and C can be punched on cards using F10.5 fields.

2. Use an I10 field-code for ID and F10.5 codes for A, B, C, and D. Assume N a number less than 10000.

3. Choose a wide I-field like I15 for printing values of I, LTERM and S. (Is S a good name for our purposes in the FORTRAN program?)

4. Use I15 fields for all integers. Let the four values of P be punched on a single card.

5. The Timekeeper records his data in hours and hundredths of an hour. Personnel keep data on rates in dollars and cents per hour.

F4-2 Illustrative examples

There are a number of simple examples of loops in Section 4-2 of your flow chart text which can be easily transliterated into FORTRAN code with the aid of the DO loop. We shall use these to further illustrate some of the details in the proper use of the DO statement. Figure F4-3 shows FORTRAN coding equivalent to the flow charts in Figure 4-8.

Label	Statement
	DO 4 I = 1, N
	Y = X(I)**3
	PRINT 51, X(I), Y
4	CONTINUE

(a)

Label	Statement
	DO 5 J = 1, N
	READ 51, X
	Y = X**3
	PRINT 52, X, Y
5	CONTINUE

(b)

Figure F4-3. FORTRAN for flow charts in Figure 4-8.

Similarly we see in Figure F4-4 a FORTRAN equivalent of the flow chart in Figure 4-9(a).

Label	Statement
	SUM = 0.
	DO 10 J = 1, N
	SUM = A(J) + SUM
10	CONTINUE
4	PRINT 50, SUM

Figure F4-4. FORTRAN for Figure 4-9(a).

Before looking at our next transliteration, please try your hand at writing the FORTRAN for Figure 4-9(b). Now compare your code with that found in Figure F4-5.

Label	Statement
	FMAX = ABSF(A(1))
2	DO 10 J = 2, N
3	IF(ABSF(A(J)) - FMAX) 10, 10, 4
4	FMAX = ABSF(A(J))
10	CONTINUE
5	PRINT 50, FMAX

Figure F4-5. FORTRAN for Figure 4-9(b)

There are several lessons to be learned here. If you look at the flow chart you see a line returning from the false side of Box 3 to the incrementation portion of Box 2.

How do we express this return line in FORTRAN?

You may have been tempted to write code something like that in Figure F4-6. Woe is you if you did!

Label	Statement
	FMAX = ABSF(A(1))
2	DO 10 J = 2, N
	IF(ABSF(A(J) - FMAX) 2, 2, 4
4	FMAX = ABSF(A(J))
10	CONTINUE
5	PRINT 50, FMAX

Figure F4-6. A mistake!

In that case you have fallen into a great big FORTRAN trap! Unfortunately, when we order a return to Statement 2 it is equivalent to returning to Box 2 of the flow chart at the initialization compartment and not at the incrementation portion! This means we start the loop all over again with J = 2 instead of continuing with the counting and testing process. In fact the only way to say in FORTRAN "go back to the incrementation portion of the DO statement" is to send control forward to the statement which is named as the terminus in the DO statement. You can see now why the CONTINUE statement is so handy to use as a terminus for a DO loop. By giving the CONTINUE the label 10 in this case and "funneling" all flow to it, we guarantee a return to the incrementation portion of the DO loop.

With this idea in mind you should have no trouble writing the FORTRAN for the flow charts in Figure 4-10 and 4-11.

Exercises F4-2 Set A

In each of the following four exercises, we present FORTRAN code for the flow charts in Figures 4-10(a), 4-10(b), 4-11(a) and 4-11(b), respectively. Your job is to indicate what errors, if any, have been made in the coding process. The necessary declarations are being disregarded here.

1. For Figure 4-10(a)

Label	Statement
	FMAX = ABSF(A(1))
	INDEX = 1
2	DO J = 2, N
	IF(ABSF(A(J)) - FMAX) 4, 4, 3
	FMAX = ABSF(A(J))
	INDEX = J
4	CONTINUE
5	PRINT 51, INDEX, FMAX

2. For Figure 4-10(b)

Label	Statement
	MAX = ABSF(A(2))
	DO 10, J = 4, N, 2
	IF(ABSF(A(J)) - MAX) 10, 10, 4
4	MAX = ABSF(A(J))
10	CONTINUE
	PRINT 5, MAX

3. For Figure 4-11(a)

Label	Statement
	FACT = 1
	DO 10 K = 1, 1, N
	FACT = K * FACT
	PRINT 5, K, FACT
10	CONTINUE

4. For Figure 4-11(b)

Label	Statement
	LTERM = 1
	NLT = 0
	DO 10 L = 1, N
	COPY = LTERM
	LTERM = LTERM + NLT
	NLT = COPY
	PRINT 5, K, LTERM
10	CONTINUE

There is one more point that is worthy of note about the DO statement. Many FORTRAN II processors interpret the DO statement slightly differently than the iteration box would imply. The iteration box implies that the test for termination is made immediately after the counter is initialized. If the test happens to fail this first time, then we never do execute the computation portion of the loop--not even once. Not so with DO statement in many FORTRAN II processors. After initializing the counter, the test is bypassed and we do enter the computation portion. It's only after completing an incrementation of the loop counter that the test is made. Hence the computation portion of a DO loop (for these processors) is always executed at least once. Ask your teacher if the processor you are using in the laboratory behaves this way.

Supposing for the moment your FORTRAN processor is of the type just described, under what circumstances would the FORTRAN code in Figure F4-5 fail to agree with our flow chart in Figure 4-9(b)?

The answer is--if by some chance the value of N were 1, then the printed value for MAX at Box 5 (Boxes 3 and 4 not executed) would be $|A_1|$ (unequivocably). On the other hand, in the FORTRAN case, if $|A_2|$ were larger than $|A_1|$, then the printed value for FMAX would be $|A_2|$, since Statements 3 and 4 are executed once.

Well, then, what must we do to bring about strict equivalence between the flow chart and the FORTRAN under these circumstances? One way this can be done is to precede the DO statement with the necessary test to permit skipping over the DO loop. Thus we might write in this case

```
      FMAX = ABSF(A(1))
      IF(2 - N) 2, 2, 5      <- this is the extra step
    2 DO 10 J = 2, N
      .
      .
      .
   10 CONTINUE
    5 PRINT 50, FMAX
```

To add this step is cumbersome and totally unnecessary in most instances. Your judgment is relied on here to decide in these matters. Obviously in this case, you are not going to be looking for the largest of 1 numbers with a DO statement.

Exercises F4-2 Set B

1 - 17. For each of the flow charts you drew in answering the exercises of Section 4-2, Set A now write the equivalent FORTRAN statements as partial programs only. Do not bother to write declarations unless it helps you to see what is going on. Hint--the tricky ones where you should be wide awake are: 4, 5, 6, 7, 12, 13 and 17.

You may be interested in seeing how the flow chart of the factors-of-N algorithm (Figure 4-14) is coded in FORTRAN. Here it is in Figure F4-7.

You will notice that all the data of this problem are integers, but the SQRTF function requires a real argument and produces a real result. So the value of N is converted to a real value by

FN = N

and FN is then used as the argument of SQRTF in the next statement. The rest of the program should be easy to follow. You may recall that [N/K] can be written in FORTRAN as simply N/K, because for positive values integer division produces the same result as the bracket function.

Label	Statement or declaration
C	FACTORS OF N
1	READ 50, N
50	FORMAT (I10)
	FN = N
	IBOUND = SQRTF(FN)
	PRINT 51, N
51	FORMAT(16H□THE□FACTORS□OF□,I15,4H□ARE)
	DO 10 K = 1, IBOUND
	IF(N - K * (N/K)) 10, 6, 10
6	L = N/K
	PRINT 52, K, L
	FORMAT(1H□, 2I15)
10	CONTINUE
	STOP
	END

Figure F4-7. Algorithm for factors of N equivalent to Figure 4-14.

The FORTRAN coding of the polynomial evaluation algorithm Figure 4-17 presents an interesting problem. In the flow chart we show the polynomial coefficients as components of a vector A beginning at A_0. In FORTRAN we

cannot have a zero subscript. The smallest value is 1. Consequently in the equivalent FORTRAN program you will see how we have had to position the coefficients beginning at A_1. To do this a slight change in the algorithm is required, as you will see when you study Figure F4-8.

Label	Statement or declaration
C	POLYNOMIAL
	DIMENSION B(4)
	READ 50, (B(I), I = 1, 4)
50	FORMAT (4F15.5)
	READ 50, X
	VALUE = B(1)
	DO 10 K = 2, 4
	VALUE = VALUE * X + B(K)
10	CONTINUE
	PRINT 51, VALUE
51	FORMAT(14H␣THE␣VALUE␣IS␣,F15.5)
	STOP
	END

Figure F4-8. FORTRAN equivalent of the polynomial evaluation (Figure 4-17)

Exercises F4-2 Set C

1 - 3. For each of the flow charts you drew in answering the exercises of Section 4-2 Set B, now write the equivalent FORTRAN--(full programs). Note special information supplied below:

(1) Assume that N will never exceed 50. Assume the values for X and A may be input using F10.5 field codes and that up to 5 values of X may be punched on a single data card.

(2) Make same assumptions about the data you made in the preceding exercises.

(3) If you continue to use integer variables, especially for your winnings, you may need to know how to call for the absolute value of an integer expression. In FORTRAN II we use the ABSF function when the argument is real but we use the XABSF function when the argument is integer. The resulting value of say XABSF(K) is an integer.

F4-3 Table-look-up

In this section we shall do two things:

1. Verify our ability to write the FORTRAN equivalent to the fairly involved table-look-up (Figure 4-24) using the bisection method. Figure F4-9 shows the program.
2. Learn a little more about input-output statements and format code.

The assumptions we have made in order to write the program given in Figure F4-9 are:

1. The table to be stored will never contain more than 200 X's and 200 Y's.
2. All input data can be formatted using I5 fields for the integers and F10.5 fields for the reals.
3. A data card read at Box 2 contains only one pair of values for the X-Y table.

```
Label    Statement or declaration
         DIMENSION X(200),Y(200)
         READ 50, N
   50    FORMAT(I5)
    2    READ 51, (X(K), Y(K), K = 1, N)
   51    FORMAT (2F10.5)
         READ 52,A
   52    FORMAT (F10.5)
         IF (X(1)-A)41, 41, 11
   41    IF(A-X(N))5, 5, 11
   11    PRINT 53,A
   53    FORMAT(1H ,F10.5,
        1      26H□IS□NOT□IN□RANGE□OF□TABLE.)
         STOP
    5    LOW = 1
         IHIGH = N
    6    IF (IHIGH-LOW-1)7, 12, 7
   12    PRINT 54, XLOW, YLOW, A, XHIGH, YHIGH
   54    FORMAT(1H□ 5F10.5)
         STOP
    7    MID = (LOW + IHIGH)/2
         IF (A-X(MID))9, 9, 10
    9    IHIGH = MID
         GO TO 6
   10    LOW = MID
         GO TO 6
         END
```

Figure F4-9. FORTRAN equivalent of table-look-up using bisection (Figure 4-24)

Implied DO loop as list elements

In studying the program in Figure F4-9, and in comparing it with its corresponding flow chart you may have noticed that in Statement 2 we used the input list element

$$(X(K), Y(K), K = 1, N)$$

This is the FORTRAN equivalent of our new flow chart notation

$$(X_K, Y_K, K = 1(1)N)$$

Executing the statement:

```
      READ 51(X(K), Y(K), K = 1, N)
51    FORMAT(2F10.5)
```

is a shorthand way of calling for execution of the statements:

```
      DO 10 K = 1, N
      READ 51, X(K), Y(K)
51    FORMAT(2F10.5)
10    CONTINUE
```

You can now see why we frequently refer to a notation like

$$(X(K), Y(K), K = 1, N)$$

as "implied DO loop" notation.

In fact, if we look at such notation in this way, we can perhaps begin to understand it for the first time. Other examples which should help are given in Table F4-1. Study these carefully.

Table F4-1

Implied DO loop notation

Example	"Shorthand"	Equivalent "longhand"
1.	READ 51,(A(I), I = 1, N) 51 FORMAT(2F10.5)	DO 10 I = 1, N, 2 READ 51, A(I), A(I+1) 51 FORMAT(2F10.5) 10 CONTINUE

Input values for all A_i from A_1 through A_N inclusive. Data card contains up to two values per card. If the value of N is 7, for example, then 4 cards will be read, the last card containing the seventh value.

Table F4-1 (continued)

Example	"Shorthand"	Equivalent "longhand"
2.	READ 51, (A(I), I = 1, N, 2) 51 FORMAT (2F10.5)	DO 10 I = I, N, 4 READ 51, A(I), A(I+2) 51 FORMAT (2F10.5) 10 CONTINUE

Input values for odd-subscripted values of A_i, beginning with A_1 and up to or including A_N (depending on whether N is even or odd, respectively). The first card contains values for A_1 and A_3, the second card has values for A_5 and A_7, etc.

3.	READ 51, (A(I), I = 5, K, 3) 51 FORMAT (3F10.5)	DO 10 I = 5, K, 9 READ 51, A(I), A(I+3), A(I+6) 51 FORMAT (3F10.5) 10 CONTINUE

If K has the value 15, for instance, then values will be input for A_5, A_8, A_{11}, and A_{14}. The first three values will be punched on the first card that is read and the fourth value will be on the second card.

4.	READ 51, (A(I), B(I), C(I), I=1, N) 51 FORMAT (3F10.5)	DO 10 I = 1, N READ 51, A(I), B(I), C(I) FORMAT (3F10.5) 10 CONTINUE

Inputs values for A_1, B_1, C_1, A_2, B_2, C_2, A_3, B_3, C_3, etc., up to and including A_N, B_N, C_N. The values for A_i, B_i, C_i for any one value of i are punched on a single card.

In the next section we will look at implied DO loops once again.

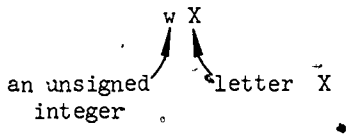
The X-field

We would now like to give you a tidbit about format code which might occasionally prove useful to you. It is the so-called "X-field". There are several interesting uses for it. Normally, however, you can get by without using them.

For an input format, the X-field is useful for skipping over (or ignoring) the data which may be punched in certain columns of a data card.

For an output format the X-field allows you to insert any number of blank spaces between numbers or comments that are to be printed on a line. When an X-field appears at the very beginning of a format code, it guarantees a blank which can be used for single-line spacing in carriage control.

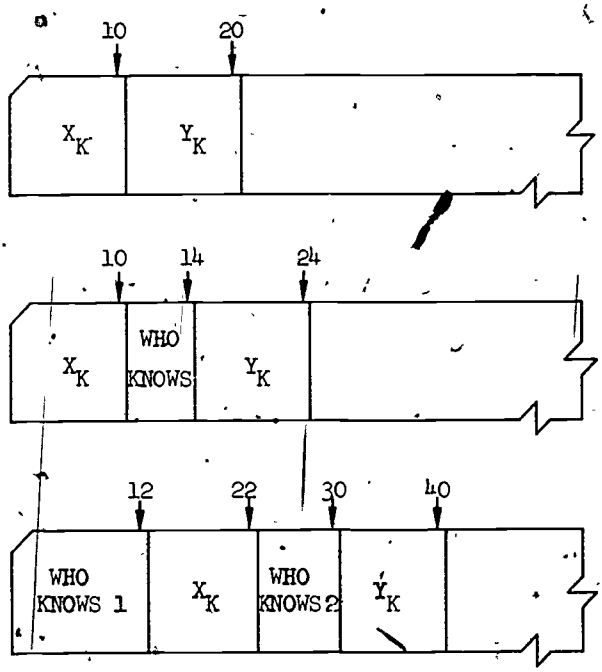
We write an X-field code using the form:



For example, 6X or 12X or 1X (Never do we write X6 or X12 or X1 !)

We now return our attention to the table-look-up problem and again consider that the deck of data cards containing the X_K and Y_K (Figure F4-10) shows four possible "card designs."

Example	Format Code
1.	(2F10.5)
2.	(F10.5,4X,F10.5)
3.	(12X,F10.5,8X,F10.5)



Example Format Code

4. On Sunday:
(2F10.5)
On Monday:
(F10.5,10X,F10.5)
On Tuesday:
(F10.5,20X,F10.5)

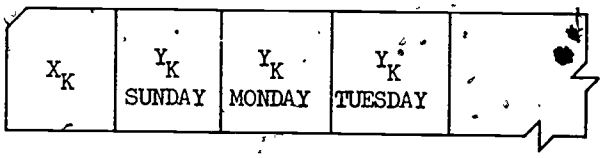


Figure F4-10. Various card designs illustrate possible uses of the X-field

1. If we were punching our own data cards for the table of function values, we might naturally choose the first design as the easiest. One never knows how many different uses such data might be put to.

2. For the benefit of a human reader, however, you may have decided to separate the two fields by, say, 4 columns. Of course, we would want to tell the computer which also must read this card that we have left a space of 4 columns so we now must write the format code as either

(F10.5, F14.5)

or as

(F10.5, 4X, F10.5)

In the first way we merely treat columns 11 - 14 as the left (blank) portion of the second field. In the second approach we tell the computer to completely ignore these columns. If, sometime later, we decide to punch something in the four intermediate columns, like a sequence number, then the use of the X-field becomes mandatory. That is why we have called columns 11 - 14 on the second card a WHO KNOWS. It is certainly safer to use the X-field.

3. Here is another card design, this time with two WHO KNOWS fields. You see how we can "sprinkle" the X-fields in and around the F-fields (or around I-fields if we had them here) in any way we may choose.

4. The final example suggests that several different tabulated functions for the same values of X_K may be punched on one card. On Sunday we might want to do a table-look-up using values of Y in Columns 11 - 20, on Monday perhaps with the values of Columns 21 - 30, and so forth. The same FORTRAN program could be used each time, provided that each time we used the program, we change FORMAT number 51 in the program (Figure F4-9) in the manner shown (Example 4 of Figure F4-10).

Now that we've begun to appreciate ways in which the X-field can be used for input format, let's see how it may be used for output.

In the first place we can see that for FORMAT 54 of the program (Figure F4-9) an alternative to

(1H□,5F10.5)

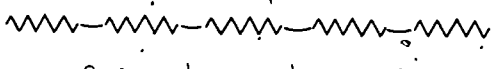
would be

(1X,5F10.5)

On output an X-field inserts blanks. So 1X at the beginning inserts the blank needed as a carriage control signal. We can also insert blanks to spread out the printed values. Thus, if we want the 5 values to be printed with arbitrary amounts of space in between each number, we could write

(1X, F10.5, 2X, F10.5, 4X, F10.5, 4X, F10.5, 2X, F10.5)

This format would group X_{LOW}, Y_{LOW} pair and the X_{HIGH}, Y_{HIGH} pair each flanking the printed value of A in order to help the eye do the grouping as suggested by the wiggles below



2 4 4 2

Of course, the same "spread" could be achieved by simply lengthening each F-field the right amount such as:

(1X, F10.5, F12.5, F14.5, F14.5, F12.5)

Take your choice.

The X-field could also be used to help in the spacing of headings across the top of tabulated data.

Exercises F4-3 Set A

Go back to Chapter 2 of your Flow Chart text. See Figure 2-7 there which shows tabulated results for our friendly loop, Figure 2-8. Your job is to write a FORTRAN program which would print a heading across the top of the table to "name" the values below, as suggested in Figure F4-11.

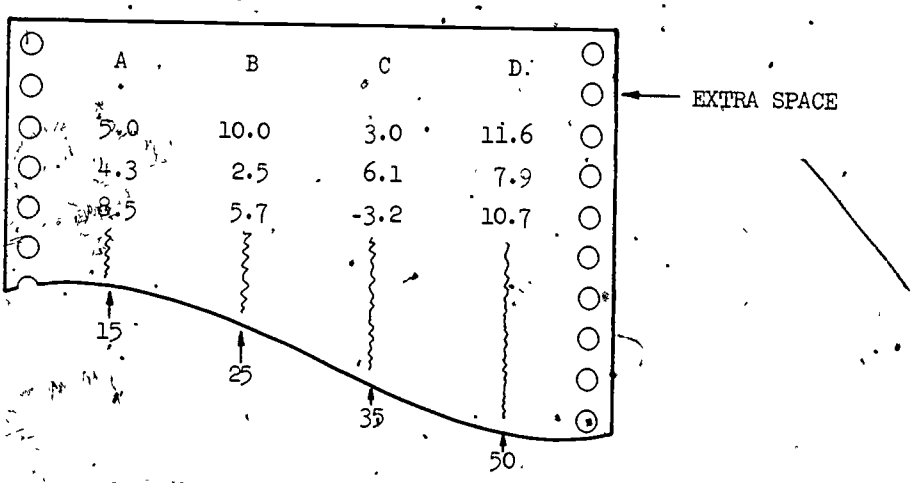


Figure. A table with printed heading

Print the table heading at the top of a page and figure out a way, if you can, to skip a line before the first line of numbers is printed using single spacing thereafter. You will probably have to revise the flow chart of Figure 2-8 first.

F4-4 Nested DO loops

Just as we can have one loop, with its iteration box, form part of the computation portion of another loop, we can have the one DO loop become part of another DO loop.

Examine the flow chart in Figure 4-29, beginning with Box 4. We now show the equivalent FORTRAN code in Figure F4-11.

```
      DO 20 I = 1, M
         SUM(I) = 0.0
         DO 10 J = 1, N
            SUM(I) = A(I,J) + SUM(I)
   10       CONTINUE
    7    TOTAL = TOTAL + SUM(I)
   20    CONTINUE
    8  PRINT 55, TOTAL
```

Figure F4-11. Nested DO-loops

We have deliberately indented the statements after each DO-loop to suggest the idea of nesting. Suppose we call those statements which are repeated under control of a DO-statement the "range" or "scope" of that DO statement or DO loop. Using this terminology, we can say that the scope of the "outer" or first DO loop includes another DO statement and its scope. If you again look at Figure F4-11 you see that the doubly-indented statements constitute the scope of the inner DO statement, while all those which are at least singly-indented belong to the scope of the outer DO statement.

When executing these FORTRAN statements, the computer can keep track of what loop it is in at all times. When the

```
DO 20 I = 1, M
```

is executed, the computer attempts to repeat the scope of this DO M times. In doing each repetition, it encounters the inner DO statement which means that the scope of this loop must be repeated N times before moving on to Statement 7 and completing the scope of the outer loop.

Each time we emerge from a loop by succeeding on a loop-limit test we say "a DO loop has been satisfied."

Using this terminology, we might say that, "in executing the scope of the outer DO loop we must repeat an inner DO loop until it is satisfied."

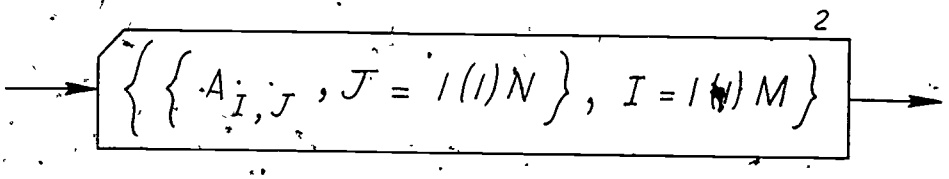
Now let's look at the whole program for summing the matrix entries. Here it is in Figure F4-12. We have assumed the matrix can have up to 50 rows and 50 columns. Data for the matrix entries are assumed to be punched up to 4 items per card, in row by row order.

```
Label   Statement or declaration
C       SUMMING A MATRIX
        DIMENSION A(50, 50)
        READ 50, M, N
   50   FORMAT(2I5)
    2   READ 51, ((A(I,J), J = 1, N), I = 1,M)
   51   FORMAT (4F10.5)
        TOTAL = 0.0
        DO 20 I = 1, M
           SUM(I) = 0.0
           DO 10 J = 1, N
              SUM(I) = A(I,J) + SUM(I)
   10         CONTINUE
           TOTAL = TOTAL + SUM(I)
   20      CONTINUE
    8   PRINT 55, TOTAL
   55   FORMAT (14H␣THE␣TOTAL␣IS␣, F10.5)
        STOP
        END
```

Figure F4-12. The whole show

Nested implied DO loops

Let's take a moment here to digress again on the subject of implied DO loop notation. In Chapter 3 we introduced you to the flow chart notation like the one on Box 2 (Figure 4-29).



The FORTRAN way is to write something very similar--namely,

```
2   READ 1, ((A(I,J), J = 1, N), I = 1, M)
```

inner implied DO-loop

outer implied DO-loop

You can now begin to see the method in this notational madness--if you want to take the time.

This is seen to be just a slick shorthand within a shorthand to represent a simpler READ statement under control of 2 (nested) DO-loops, Figure F4-13.

```
      DO 20 I = 1, M
         DO 10 J = 1, N
            READ 70, A(I,J)
70          FORMAT (F10.5)
10          CONTINUE
20       CONTINUE
```

Figure F4-13. The "long way", but not quite equivalent to Statement 2

In this case we show here the equivalence with Statement 2 is not quite complete because only one matrix entry can be put on one card whereas in Figure F4-12 we were able to specify 4 items per card using FORMAT. We can correct this by having 4 list elements in the READ statement as shown in Figure F4-14.

```
      DO 20 I = 1, M
         DO 10 J = 1, N, 4
            READ 51, A(I,J), A(I,J+1), A(I,J+2), A(I,J+3)
51          FORMAT (4F10.5)
10          CONTINUE
20       CONTINUE
```

Figure F4-14. "The long way", but equivalent to Statement 2

There is still another and even better "long way" as shown in Figure F4-15. We would recommend it over any of the other methods. It has both the short and the long look.

```
      DO 20 I = 1, M
         READ 51, (A(I,J), J = 1, N)
51       FORMAT (4F10.5)
20          CONTINUE
```

Figure F4-15. The "long-short" way, equivalent to Statement 2

You see that we have used a single implied DO-loop in the READ statement and we repeat the read statement under control of a DO statement. This

method has the following virtue:

The READ statement amounts to an order to read a whole row of the matrix with up to four entries per card. If the last entries of a row don't quite fill out a card--no matter--we can still start the entries for the next row--fresh on the next card in sequence. This way if you have a fairly big matrix to put on cards, it's easy to verify you've punched entries for each row correctly. You don't get this flexibility when you buy the doubly-implied DO loop that is used in Figure F4-12; there entries for each new row begin, if necessary, on the same card with the last entry or entries of the preceding card.

Exercises F4-4 Set A

1 - 8. For each of the flow charts you constructed for the exercises in Section 4-4, Set A, write the equivalent FORTRAN statements. Don't bother writing declarations unless you feel they add to your understanding of the transliteration problem.

Triply nested DO loops are just as easy to write as doubly nested ones. Figure F4-16 shows how the stickler in Figure 4-31 would be coded in FORTRAN.

```
Label   Statement
        DO 30 IH = 1, 9
        DO 20 IT = 1, 10
        DO 10 IU = 1, 10
           ITM1 = IT - 1
           IUM1 = IU - 1
        IF (100 * IH + 10 * ITM1 + IUM1)
       1     -(IH**3 + ITM1**3 + IUM1**3)
       2     10, 5, 10
      5 PRINT 50, IH, ITM1, IUM1
     50 FORMAT (3I15)
     10 CONTINUE
     20 CONTINUE
     30 CONTINUE
        STOP
        END
```

Figure F4-16. The stickler in FORTRAN

Since a DO-loop counter may not begin with a zero value we have had to be a bit inventive to accomplish the same objectives of the stickler algorithm. As you can see in Figure F4-16, we let IT and IU range from 1 to 10 instead of from 0 to 9. In the IF statement expression we use ITM1 (defined as IT - 1) and IUM1 (defined as IU - 1) in place of IT and IU, respectively.

Exercise F4-4 Set B

1. Write FORTRAN programs for the flow chart solutions you obtained for Problem 7, Section 4-4, Set B of the Main Text.

Exercises F4-4 Set C

1. Write a complete FORTRAN program for the Complete Factorization Algorithm, Figure 4-32. Assume the input value for N will not be as large as 10000. In converting to FORTRAN, you should be on the lookout for two interesting features. First, take aim on the test portion of the iteration box--the upper limit must be an integer. Second notice that the scope of the DO loop includes a complete loop--but without an iteration box.

2. Write a complete FORTRAN program for the shuttle-interchange sorting algorithm, Figure 4-34. Assume you may wish to sort up to 500 numbers, each of which can be input using an F10.5 field--say up to four values per card. Be on the lookout for occasional difficulties in converting the flow chart to the FORTRAN code--especially in modeling Box 6 which shows decreasing counter set initially to an expression.

3. Write a complete FORTRAN program for the sort algorithm shown in Figure 4-35. Make the same assumptions in this program that you are asked to make in the preceding exercise.

4. Write a complete FORTRAN program for finding the largest decreasing subsequence. Base your program on the flow chart in Figure 4-39. Assume the given sequence will not exceed 100 values in all, and that the values themselves are to be punched on data cards, four values per card, governed by F10.5 field codes.

Chapter F5

SUBPROGRAMS

F5-1 FORTRAN subprograms

FORTRAN programs that correspond to reference flow charts are called FORTRAN subprograms. As you might expect, these subprograms look very much like other FORTRAN programs except for special statements at the beginning and end corresponding to the funnel and return box of the flow chart. In addition to these special statements there are a number of conventions to be observed which will be explained in what follows.

A subprogram is a self-contained unit which can be compiled separately, and then used by a number of programs or other subprograms. One of the main advantages of separate compilation is that one can in this way establish and develop a library of subprograms, which is available for later use.

Subprograms that evaluate a function and report a single value to the main program are called function subprograms. (A second type of subprogram will be encountered in Section F5-4.) For a function subprogram, the flow chart funnel corresponds to the first statement of the subprogram. This statement begins with the word FUNCTION, followed by the name of the function and its argument in parentheses. For example, Figure 5-4 corresponds to

FUNCTION SQROOT(Y)

Since special symbols like $\sqrt{\ }$ are not available in FORTRAN, we naturally replace such a symbol with an alphabetic name for the function (in this case SQROOT). Names of function subprograms are to be chosen with some care, observing the following conventions. The first character in the subprogram name must be I, J, K, L, M, or N if and only if the value reported is in the integer mode. The last character must not be F if the name has more than 3 characters. Otherwise the name can have 1 to 6 characters, the first being alphabetic.

You are already familiar with the use of predefined mathematical functions like SQRTF, SINF, ABSF, etc. (See Table F2-2.) FORTRAN subprograms are different from these predefined mathematical functions even though our first example, SQROOT, serves the same purpose as SQRTF. The difference is not just in the way the names are spelled (the conventions are different for subprograms and predefined functions), but is mainly in the

fact that the predefined mathematical functions are part of the compiler system. The techniques for adding to the list of predefined mathematical functions are outside of the scope of this book but function subprograms provide a way that you can use to develop whatever set of reference programs you want.

Just as reference flow charts must sooner or later reach a return box, a subprogram must eventually reach a RETURN statement. This statement, corresponding to Figure 5-5, consists just of the word "RETURN". This statement need not appear in the last line of the subprogram (more than one RETURN statement may be used in a single subprogram), but it must be the last step in the execution of the subprogram.

Since the RETURN statement of FORTRAN does not indicate what variable is to be reported to the main program, another convention is needed to identify the value to be reported. The convention is that the name of the function subprogram itself must appear at least once on the left side of an assignment statement and the value assigned is, upon return, the value reported to the main program. For this purpose only, the function name is to be thought of as a variable.

In all other respects, a FORTRAN subprogram must conform to the requirements of any FORTRAN program. In particular, the last line of a subprogram must be followed by an END statement, and DIMENSION information must be given for each subscripted variable in the subprogram.

The use of a function subprogram by a main program is shown in Figure F5-1.

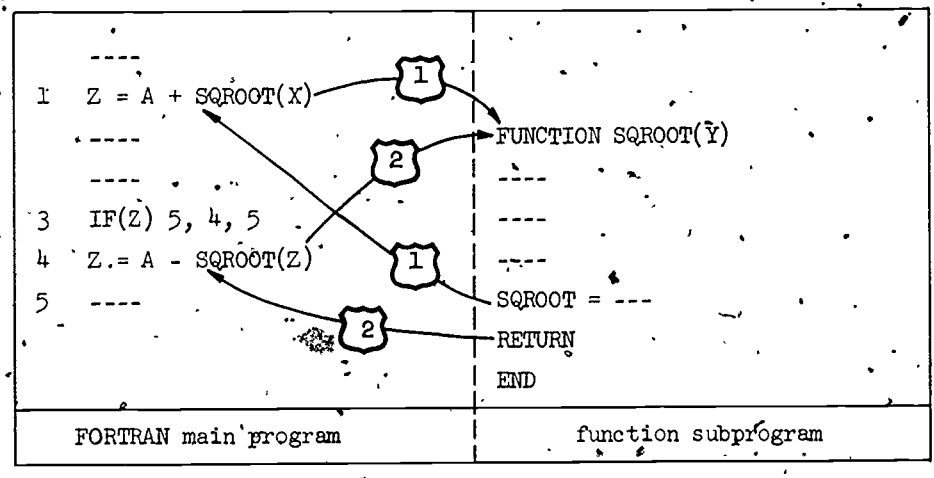


Figure F5-1. Use of a FORTRAN function subprogram

Figure F5-1 is intended to correspond to Figure 5-6 of the flow chart text and to illustrate only those features related to subprogram use. The first time the subprogram is required (in Statement Number 1 of the main program), we go to the FUNCTION statement via Route 1. This statement directs that the value of X be assigned to Y. It is essential that the name of the argument in the FUNCTION statement and the name of the argument where the subprogram is requested be of the same mode. That is, since Y represents a real variable, a request for SQROOT(N) would be in error since N represents an integer variable.

When the subprogram has been executed a value has been assigned to SQROOT and the return to Statement 1 of the main program is by Route 1. Upon return to the main program, the value assigned to SQROOT is added to A and the result assigned to Z. Where the subprogram is again required (in Statement Number 4) we are to go to the FUNCTION statement by Route 2, assign Z to the Y in the subprogram (note they agree in mode), execute the subprogram, and return via Route 2 to Statement 4 with the result of the subprogram assigned to SQROOT.

An actual function subprogram for the square root can be prepared with reference to Figure 5-7, as shown in Figure F5-2.

```
C       SQUARE ROOT FUNCTION SUBPROGRAM
        FUNCTION SQROOT(Y)
        G = 1.0
    2   H = 0.5 * (G + Y/G)
        IF(ABSF(H - G) - .0001) 5, 10, 10
   10   G = H
        GO TO  2
    5   SQROOT = H
        RETURN
        END
```

Figure F5-2. Function subprogram for square roots

As you inspect this first function subprogram we take the opportunity to warn you away from something you're not likely to do anyhow. That is, don't try to assign anything to an argument (in this case, Y) of a function subprogram. It is never necessary to do this anyway. With some FORTRAN compilers this will produce very strange results--so let the buyer beware! (You will be given one more reminder of this danger area in Section F5-3--then you will be on your own.)

Exercises F5-1

1 - 3. Write FORTRAN function subprograms for the flow charts prepared in Exercises 5-1 main text.

F5-2 Functions and FORTRAN

The flow chart text tells us that any flow chart which, when given a value will produce another value, can be viewed as the evaluation of some function. The statement is true for FORTRAN programs as well as for flow charts. Mathematical functions exist which cannot be evaluated with a flow chart or by a FORTRAN program, but the common usage of the word function in computing is strictly limited to those which can be evaluated with a flow chart. Moreover, we do not usually think of expressions which can be precisely evaluated solely by one of the four basic arithmetic operations as functions although, in fact, they are. In computing, then, a function is commonly thought of as a relationship for which a reference flow chart is used.

The domain of a function, in computing, is the set of values that the argument in the funnel of the flow chart can take on. The range of a function is the set of values that can be reported to the main flow chart. In FORTRAN, the domain can be either a set of real numbers representable in a computer or a set of integers representable in a computer. Which set of numbers is meant is indicated, in the usual way, by the first letter of the name of the argument. The range, too, can be either from the set of integers or from the set of real numbers. Which is meant is indicated, in the usual way, by the spelling of the name of the function.

F5-3 FORTRAN functions with more than one argument

FORTRAN function subprograms can have as many arguments as are necessary. The min function provides an example. The subprogram in Figure F5-3 corresponds to Figure 5-14 of the flow chart text.

```
C
C       MINIMUM OF TWO ARGUMENTS
C         FUNCTION SUBPROGRAM
        FUNCTION FMIN(B, C)
        IF(B - C) 2, 2, 3
    2   Z = B
        GO TO 4
    3   Z = C
    4   FMIN = Z
        RETURN
        END
```

Figure F5-3. A function subprogram of two arguments

Notice that we have changed the name of the function subprogram so that it does not begin with the letter M. That is, the function subprogram expects to receive two real values and to report a real value. We do not need to introduce the variable Z since FMIN can serve the same role. For this reason, Figure F5-3 can be replaced by Figure F5-4. Should Figure F5-2 be changed in a similar way?

```
C
C       MINIMUM OF TWO ARGUMENTS
C         FUNCTION SUBPROGRAM
        FUNCTION FMIN (B, C)
        IF(B - C) 2, 2, 3
    2   FMIN = B
        GO TO 4
    3   FMIN = C
    4   RETURN
        END
```

Figure F5-4. Improved FMIN subprogram

The parameter list of a FORTRAN subprogram can contain integer variables and constants, real variables and constants, variables containing alphanumeric information, the names of vectors, and the names of matrices. In every case there must be a one-to-one correspondence in the number and types of parameters between those of the parameter list where the subprogram is requested and those of the parameter list in the FUNCTION statement. Confusion would

reign if we tried to request the subprogram of Figure F5-4 by writing something like

T = FMIN (A, B, C)

or T = FMIN (M, P)

A classification of variables

The distinction between local and nonlocal variables is the same in FORTRAN as is described in the flow chart text. Thus in the two subprograms for FMIN given earlier, the arguments B and C are nonlocal variables. In the first subprogram Z is a local variable.

Some compilers are written on the assumption you will never try to reassign values to nonlocal variables of a FUNCTION subprogram. Such compilers take advantage of this fact in assembling efficient target code for your program. Hence, an attempt to "out-fox" the compiler in this respect, just to save a step in the program, or to save a storage cell or two, could lead to trouble. Unless you are absolutely sure about what can happen you should avoid changing nonlocal variables in a subprogram.

An appreciation of the distinction between local and nonlocal variables can be had by recalling that subprograms can be compiled separately from programs that make use of them. During compilation a local variable is associated with a specific location in memory (except for an additive constant needed to account for possible shifting of the whole subprogram from one place in memory to another.) On the other hand, there is no way a subprogram can know prior to compilation of the main program, where nonlocal variables may be located when the subprogram is executed.

Independence of statement numbers among programs

The realization that compilation of a subprogram is an entirely separate process from the compilation of a main program also brings out the fact that statement numbers in a subprogram are entirely different references from statement numbers in another subprogram or in a main program even if the same numeral has been used.

Composition of function references

The situation with respect to composition of function references is exactly as described in the flow chart text. This is just what you have learned as composition of functions. That is, given function subprograms

defining F1(X) and F2(X), one can write

$$Y = F1(F2(X))$$

so long as the range of F2 is a subset of the domain of F1. Correspondingly, the following function references are entirely proper:

```
Y = FMIN(ABSF(A + B), 5.4)
```

or `Y = FMIN(FMIN(F, ABSF(T)), Q)`

or `Y = FMIN(SQROOT(B * B - 4.0 * A * C), -B)`

or `Y = SQROOT(FMIN(X,Y))`

Exercises F5-3 Set A

1 - 7. Write FORTRAN programs and function subprograms for the flow charts prepared in Exercises 5-3, Set A, Main Text.

Exercises F5-3 Set B

1. Write a FORTRAN function subprogram for the GCD functional reference whose flow chart you prepared in Problem 1 of Set B in the main text. Call this subprogram KGCD. Why must the name be changed for FORTRAN?

2. Write a FORTRAN function subprogram called KGCF corresponding to the flow chart you drew for the GCF algorithm (Problem 2, Set B, main text).

3. Write a FORTRAN program that corresponds to the flow chart for determining

   (a) the number of similar triangles

   (b) the total perimeter of similar triangles

   corresponding to the flow charts you prepared in Problem 3, Set B, main text.

*4. Write a FORTRAN program that corresponds to the algorithm for Problem 4, Set B, main text. Try to estimate how much computation will be involved. Computation can be measured in terms of the number of additions, subtractions and comparisons that must be made, counting each as 1.

---

*These problems are quite difficult. The student will be able to solve them only with considerable time and effort.

F5-4 FORTRAN procedures

FORTRAN subprograms which correspond to reference flow charts for procedures are called SUBROUTINE subprograms. Corresponding to the funnel of the reference flow chart is the SUBROUTINE statement which takes the form such as:

```
SUBROUTINE SORT(N, V)
```

The subroutine statement consists of the word SUBROUTINE followed by the name of the procedure you are defining and a parameter list in parentheses. In the SORT example we see that an entire vector is identified solely by its name, V, in the parameter list. No attempt is made to subscript V in the parameter list but the subprogram will have to provide DIMENSION information for this vector.

Since a subroutine subprogram does not report a value in the same way that a function subprogram does, the name of the procedure will not appear in the body of the subprogram. Moreover, since this name will not be used as though it were a local variable, its spelling (i.e., its first letter) has no special significance. The name of a procedure must not end in F if it is more than three characters long and all other conventions are as they are for function subprograms.

A subroutine subprogram corresponding to Figure 5-16 is given in Figure F5-5. The dimension of V has been given as 100 but, for each use, N is the actual number of components in the vector to be sorted. Thus, this subprogram is usable for any vector having 100 or fewer components.

```
C       SUBROUTINE SUBPROGRAM FOR SORT
        SUBROUTINE SORT(N, V)
        DIMENSION V(100)
        K = N - 1
        DO 40 I = 1, K
        M = I + 1
        DO 40 J = M, N
        IF(V(I) - V(J)) 40, 40, 4
    4   B = V(J)
        V(J) = V(I)
        V(I) = B
   40   CONTINUE
        RETURN
        END
```

Figure F5-5. Subprogram for sorting

Look at Figures 5-16 and F5-5 side by side. See how the FORTRAN statements correspond almost exactly to the flow chart boxes. Notice also that values can be intentionally assigned to parameters of a subroutine subprogram. We warned you not to do this in function subprograms but this is how a subroutine subprogram produces its output.

Use of a SUBROUTINE subprogram is accomplished with a CALL statement, analogous to the execute box of the flow chart text. The CALL statement consists of the word CALL, followed by the name of the subroutine subprogram and a parameter list in parentheses, for example:

```
CALL SORT(88, B)
```

where B is dimensioned to have at least 88 components in the main program. In FORTRAN, the process of referring to a subroutine is called "calling the subroutine".

A main program calling SORT, corresponding to Figure 5-23, is shown in Figure F5-6. This program

```
C       A PROGRAM TO ILLUSTRATE CALL STATEMENTS
        DIMENSION B(100), C(100)
        READ 101, K
  101   FORMAT (I3)
        READ 102, (B(I), I = 1, K)
  102   FORMAT(5F15.8)
        READ 102, (C(I), I = 1, K)
        CALL SORT (K, B)
        CALL SORT (K, C)
        PRINT 103, (B(I), C(I), I = 1, K)
  103   FORMAT (1X, 2F 15.8)
        STOP
        END
```

Figure F5-6. A main program calling the sort procedure

assumes that K is a three digit integer in the first three columns of the first data card; that the components of B and C are punched five per card in F15.8 format. It is also assumed that K will not exceed 100.

Exercises F5-4 Set A

1 - 5. For the flow charts prepared in Exercises 5-4, Set A main text, write FORTRAN calling programs and subroutine subprograms.

Exercises F5-4 Set B

1 - 3. For the flow charts prepared in Exercises 5-4, Set B main text for Problems 1, 2, and 3, write the FORTRAN function and subroutine subprograms.

Exercises F5-4 Set C

1 - 4. For the flow charts prepared in Exercises 5-4, Set C main text for Problems 1 through 4, write the FORTRAN procedures and programs. In doing the program for the subroutine DEGREE, keep in mind that you must shift the indexes for the coefficients, i.e., the coefficient a_0 must be associated with A(1), etc. The same problem must be handled in a similar fashion in coding the flow chart for SIMPLIFY and REDUCEMOD. (spelling O.K. for FORTRAN?) in Problems 2 and 3.

F5-5. Alternate exits and procedures for branching

Provision for alternate exits and branching from subroutine subprograms in FORTRAN mirrors the initial discussion in the flow chart text. A parameter is provided to indicate the result of tests performed by the subprogram. Figures F5-7 and F5-8 present programs corresponding to the flow charts of Figure 5-26 and Figure 5-27.

```
C       SUBPROGRAM TO TEST EQUALITY OF
C         TWO COMPLEX NUMBERS
        SUBROUTINE COMPEQ(A, B, C, D, N)
        IF(A - C) 4, 2, 4
    2   IF(B - D) 4, 3, 4
    3   N = 0
        GO TO 5
    4   N = 1
    5   RETURN
        END
```

Figure F5-7. Equality of complex numbers in FORTRAN

```
C       PROGRAM SEGMENT SHOWING USE OF COMPEQ
          ---
        CALL COMPEQ(X, Y, U, V, K)
        IF(K) 3, 4, 3
C       STATEMENT 3 IF UNEQUAL
C       STATEMENT 4 IF EQUAL
          ---
```

Figure F5-8. A program segment to test equality of complex numbers

Statement labels and function names as arguments

FORTRAN II was not designed with such types of arguments in mind. Hence, few if any processors have this capability. More advanced languages do, however, permit full freedom in the use of such procedure arguments.

Exercises F5-5

1. Construct a flow chart similar to that of Problem 1, Exercises 5-5, Main Text, and then write a FORTRAN subroutine subprogram to solve two equations in two unknowns.

2. Use the appropriate flow charts prepared for Problem 2, Exercises 5-5, to write a FORTRAN program and subroutine subprogram which will compute the real roots of a quadratic equation. Use the program to solve the following equations:

$$2x^2 - 3x + y = 0$$

$$3.14x^2 - 6.2x - 14.23 = 0$$

3. Write a FORTRAN subroutine subprogram and calling program statements corresponding to one of the techniques you used for solving Problem 3, Exercises 5-5.

F5-6 Symbol manipulation in FORTRAN

In Section F2-10 we discussed the input and output of alphanumeric characters. Now we want to find out how alphanumeric data can be processed so that we will be able to alter such input data as

THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.

or 3.14159

or $r + s(t + u(v + w))$.

Since we will want to be able to refer to each individual element in such character strings, we will associate a separate variable with each element of a string. This means that a card, let us say, would be read with a FORMAT code of 80A1.

We are now ready to code a subroutine subprogram, corresponding to Figure 5-33, for CHEKCH. This program is shown in Figure F5-9. We notice that there is a DIMENSION statement giving the maximum length of S as 200. Other than for the DIMENSION statement the program

```
C       A SUBPROGRAM FOR CHEKCH
        SUBROUTINE CHEKCH (N, S, M, C, IP)
        DIMENSION S(200)
        DO 2 I = M, N
        IF(S(I) - C) 2, 3, 2
    2   CONTINUE
        IP = 0
        GO TO 5
    3   IP = I
    5   RETURN
        END
```

Figure F5-9. A FORTRAN subprogram for CHEKCH

could accomodate any length string. As presented in Figure F5-9, 200 is simply a maximum length and could be replaced by a larger integer subject only to the amount of memory available. The alphanumeric data is associated with real variables, S and C, as in Section F2-10. Finally, we see that two characters like the values of S(I) and C can be tested for identity by subtracting one from the other as if they were numbers, and testing for a zero difference.

A FORTRAN subprogram corresponding to Figure 5-34 is given in Figure F5-10. Here we assume the string, S, has 200 or fewer elements and the substring, C, has 20 or fewer elements.

```
C       A SUBPROGRAM FOR CHEKST
        SUBROUTINE CHEKST (N, S, M, K, C, IP)
        DIMENSION S(200), C(20)
        L = M
    2   IF(L - N + K - 1) 3, 3, 11
    3   CALL CHEKCH (N, S, L, C(1), IP)
        IF(IP) 5, 11, 5
    5   IF(IP - N + K - 1) 6, 6, 11
    6   IR = IP + 1
        DO 9 J = 2, K
        IF(S(IR) - C(J)) 10, 9, 10
    9   IR = IR + 1
        GO TO 12
   10   L = IP + 1
        GO TO 2
   11   IP = 0
   12   RETURN
        END
```

Figure F5-10. A FORTRAN subprogram for CHEKST

As the flow chart text has pointed out, the ability of a subroutine subprogram to call another subprogram has far-reaching significance. It is this ability that permits the construction of subprograms of increasingly greater complexity from simpler "building blocks". Chapter 8 and all larger problems will make heavy use of this building block ability.

Exercises F5-6

1 - 4. For the flow charts prepared in Exercises 5-6, main text, write FORTRAN programs and subprograms.

One further point should be made here. It is often very convenient to consider the length of a string, and the string itself as being a single entity. One way to do this is to use an array, say STR, with the property that its first component STR(1) is equal to the length of the string, while its remaining components STR(2), STR(3), and so on, are the characters themselves.

If we denote the character for which the search of CHEKCH is being made by CHAR, and its position (if found) by KP, the subprogram is shown in Figure F5-11.

```
C       MODIFIED SUBPROGRAM FOR CHEKCH
        SUBROUTINE CHEKCH (STR, M, CHAR, KP)
        DIMENSION STR(100)
        MM = M + 1
        N = STR(1) + 1.0
        DO 2  I = MM, N
        IF(STR(I) - CHAR) 2, 1, 2
    1   KP = I - 1
        GO TO 3
    2   CONTINUE
        KP = 0
    3   RETURN
        END
```

Figure F5-11. A modified subprogram for CHEKCH

The only complication is in adding 1 in two places, and subtracting in another. These steps are required because now the I^{th} character in the string is the $(I + 1)^{th}$ component of the array STR.

Chapter F7

SOME MATHEMATICAL APPLICATIONS

F7-1 Root of an equation by bisection

Now let us write a program to find the root of an equation $y = f(x)$ in the interval $x_1 = A$ and $x_2 = B$. We shall write the program, corresponding to the flow chart of Figure 7-5, in the form of a subroutine subprogram called ZERO so that it can be used with many different main programs. Then we shall write a FORTRAN program which calls ZERO.

The given function f(x), we shall call FUNCT(X); it will need to be defined as a FORTRAN function subprogram. The desired accuracy of the result is given as EPSI; so the bisection process will be terminated when the length of the interval is less than the value of EPSI. Inside the subroutine the value of the root of the equation will be assigned to the variable ROOT. The final value of ROOT is printed before returning to the calling program.

Figure F7-1 shows one way to code the ZERO subroutine. Only three dummy arguments are used, A, B, and EPSI. The first thing that's done by the subprogram is to reassign the values of A and B to auxiliary variables X1 and X2 as a safeguard to protect the values of arguments that match A and B in the calling program. In case the bisection method is inapplicable we let the subroutine print the message.

The value of ROOT could, of course, be carried back to the calling program by making ROOT a dummy variable and adding it to the parameter list in the opening declaration. In this event the CALL statement would also show a parameter list of four actual arguments, the last one being a variable that accepts the computed value of the root.

```
      SUBROUTINE ZERO(A,B,EPSI)
      X1 = A
      X2 = B
      Y1 = FUNCT(X1)
      IF(Y1 - FUNCT(X2)) 6, 5, 4
    5 IF(Y1) 8, 7, 8
    7 ROOT = X1
      GO TO 12
    8 ROOT = X2
   12 PRINT 61, ROOT
   61 FORMAT(E20.8)
      RETURN
    6 XM = 0.5 * (X1 + X2)
      IF(ABSF(X1 - X2) - EPSI) 11, 10, 10
   10 ROOT = XM
      GO TO 12
   11 IF(Y1 * FUNCT(XM)) 13, 10, 14
   13 X2 = XM
      GO TO 6
   14 X1 = XM
      GO TO 6
    4 PRINT 62
   62 FORMAT(23HbMETHODbISbINAPPLICABLE)
      RETURN
      END
```

Figure F7-1

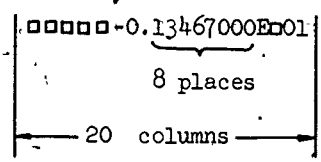
E-fields

There are many things you probably want to look at carefully in the ZERO subprogram. We might, however, digress temporarily to explain the strange field code that is used in format number 61.

The "E-field" is in many respects similar to the F-field for input and output of real data values. E-fields are used to describe the input or output of real data values written in the "E" notation like

0.5253E-5

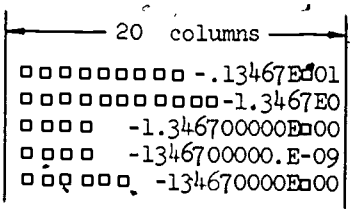
A number printed under control of an E20.8 field would occupy 20 columns. If the value in memory is -1.3467 this number would appear in print as:



The right-most four columns would be used for printing the exponent, the first of which is for the letter E, the second for a minus sign or blank, and the third and fourth for a two digit exponent (power of 10). There is a zero to the left of the decimal point of the precision part preceded by a minus sign if the number is negative. The leading digit on the right of the decimal point is always made non-zero and the exponent is adjusted accordingly.

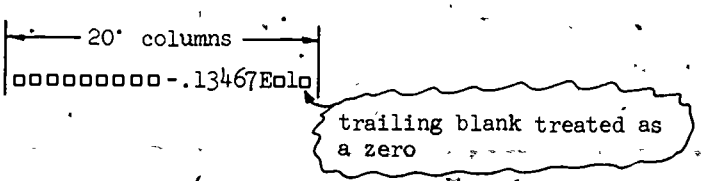
A number that is read into memory under control of an E-field like E20.8 need not have the decimal point actually punched on the card. If it is punched, the 8 in E20.8 is ignored just like in F-fields. Thus the number -1.3467 might be punched in several different ways on input using the E20.8 and still enter memory with the same value. Here are some examples.

Different ways to punch the same value on 20 columns of a card.



Only in the last example where no decimal point is punched will the 8 in E20.8 be used to tell where the decimal point should be. In all other cases the computer takes the number just as it is punched and interprets it the same way it would any FORTRAN constant which is written the same way.

One word of caution--always be sure--if you use this type of input, to have the exponent part of the number occupy the right-most four or fewer columns. Leave no trailing blanks after the exponent as they are generally interpreted as zeros. For example:



Here most FORTRAN implementations would interpret this value as if it were

-.13467E10 , instead of

-.13467E1 as intended.

One time use of ZERO

Now suppose we wanted to use ZERO, to find the root of the equation $3x^3 - 7x - 2 = 0$ which lies between 1 and 2. Then we should need to define a function subprogram FUNCT which calculates the value of $3x^3 - 7x - 2$. If we choose EPSI = 10^{-4} we would need to call ZERO by the statement:

```
CALL ZERO (1.0, 2.0, 1.0 E - 4, ROOT)
```

The FORTRAN program could be written as shown in Figure F7-2.

```
      CALL ZERO (1., 2., 1.0E-4)
      STOP
      END
C - PLACE THE DEFINITION OF SUBROUTINE ZERO HERE
      FUNCTION FUNCT(X)
      FUNCT = (3.0*X*X-7.0)*X-2.0
      RETURN
      END
      END
```

Figure F7-2.

Using ZERO on several different functions

Ideally we would prefer to have written ZERO with a list of dummy arguments like those shown in Figure 7-6 of the main text. Unfortunately FORTRAN II processors do not permit statement label arguments, and only a few permit function name arguments. This cuts down the flexibility we would like, but we can still find good uses for procedures like ZERO after some slight modification.

Suppose we wish to use the ZERO procedure with a series of functions. Though desirable, this would be difficult with ZERO as written. As things stand now, we would have to reproduce the ZERO and "package" it with each function separately. What we need is some means by which to identify each of a series of functions so that we can communicate to ZERO which of the functions it is to use when called. Although we must always call our function FUNCT, we will now consider FUNCT to be a function of two variables, x and k, where k is really just an index to identify the particular function.

The revised FORTRAN program which we will call ZEROK follows in Figure F7-3. Notice the fourth argument is K, an index that is used to identify the particular function to be called on.

```
      SUBROUTINE ZEROK(A,B,EPSI,K)
      X1 = A
      X2 = B
      Y1 = FUNCT(X1,K)
      IF(Y1 - FUNCT(X2,K)) 6, 5, 4
    5 IF(Y1) 8, 7, 8
    7 ROOT = X1



   10 ROOT = XM
      GO TO 12
   11 IF(Y1*FUNCT(XM,K)) 13, 10, 14


      RETURN
      END
```

Figure F7-3. The subroutine ZEROK. It's identical with that of Figure F7-1 except for the opening declaration and statements that contain references to FUNCT.

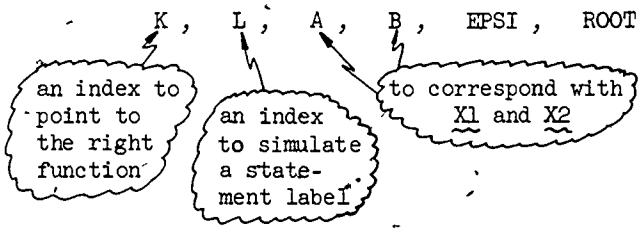
Figure F7-4 shows how we might use ZEROK to find the root of $3x^3 - 7x - 2 = 0$ between 1 and 2, the root of $x^5 - 4x^4 + 7x^3 - x + 3 = 0$ between -1 and 0 and the root of $x = \cos(x)$ between 0 and 1. We choose EPSI = 10^{-4}.

```
      CALL ZEROK (1., 2., 1.0E-4, 1)
      CALL ZEROK (-1., 0., 1.0E-4, 2)
      CALL ZEROK (0., 1., 1.0E-4, 3)
      STOP
      END
C     PLACE THE DEFINITION OF SUBROUTINE ZEROK HERE
      FUNCTION FUNCT(X,K)
      IF(K - 2) 1, 2, 3
    1 FUNCT = (3.*X*X-7.)*X-2.
      RETURN
    2 FUNCT = (((X-4.)*X+7.)*X*X-1.)*X+3.
      RETURN
    3 FUNCT = COSF(X)
      RETURN
      END
      END
```

Figure F7-4.

The final step in the development of ZERO as a procedure is to give it an alternate exit if the method proves inapplicable. We shall use the technique illustrated in Figures 5-26 and 5-27 of the main text. That is, we shall employ another output argument, L, which will be assigned an index value in the procedure. Upon return to the calling program the value of L will be inspected to see if an alternate exit has been implied. Once we add this feature it will no longer be necessary to have the procedure do any printing. We can add ROOT as an output argument and carry its value back to the calling program. The calling program can, in turn, take all responsibility for printing results or diagnostic messages.

We shall call the new bisect procedure ZEROKL. To complete the analogy with the parameter list in the funnel of the zero procedure in Figure 7-6, we let the parameters for ZEROKL be:



The subprogram is given in Figure F7-5. The argument L is set to 0 to indicate a normal exit and is set to 1 to indicate the alternate exit.

```
      SUBROUTINE ZEROKL(K,L,A,B,EPSI,ROOT)
      X1 = A
      X2 = B
      Y1 = FUNCT(X1, K)
      IF(Y1 - FUNCT(X2,K))6,5,4
    4 L = 1
      RETURN
    5 IF(Y1) 8, 7, 8
    7 ROOT = X1
   12 L = 0
      RETURN
    8 ROOT = X2
      GO TO 12
    6 XM = 0.5*(X1 + X2)
      IF(ABSF(X1 - X2) - EPSI) 10, 11, 11
   10 ROOT = XM
      GO TO 12
   11 IF(Y1*FUNCT(XM,K)) 13, 10, 14
   13 X2 = XM
      GO TO 6
   14 X1 = XM
      GO TO 6
      END
```

Figure F7-5. A FORTRAN analog to Figure 7-6

Exercises F7-1

1. Write a FORTRAN program to solve all of the equations given in Exercises 7-1, Set C, Problem 1 of the main text. Use the indicated intervals and the indicated error tolerances. Use the subprogram ZEROK given in Figure F7-3. Run the program and compare your results with the hand calculated ones. Also run the program for 3 with EPSI = 10^{-4}.

2. Write and run a FORTRAN program to carry out the function evaluations needed in drawing the graphs in Exercises 7-1, Set A of the main text. You can use some of the same FUNCTION subprograms needed in Problem 1.

3. Write and run a FORTRAN program to solve the alley problem (Number 6) in Exercises 7-1, Set D of the main text. Then solve the problem to the nearest hundredth of a foot if the ladders are 25.83 and 19.14 feet long, and the crossover point is 7.17 feet above the ground.

4 - 5. For each of the flow chart solutions you prepared for Problems 1 and 4, Exercises 7-1, Set D, employ the ZEROKL subroutine developed in Figure F7-5 and write the necessary companion FORTRAN programs (main programs and FUNCTION subprograms).

F7-2 The area under a curve: An example, $y = 1/x$ between $x = 1$ and $x = 2$

Since the area under the curve $y = 1/x$ is of interest in defining logarithms we begin by writing a simple FORTRAN program for the calculation of the approximate area under this curve between $x = 1$ and $x = 2$. This calculation will provide an approximation to $\ell n 2$. We assume that an error tolerance EPSI is read in from a card and that calculation of the approximate area is to be carried out by doubling the number of subdivisions each time and terminating the calculation when the absolute value of the difference of two successive approximations is less than EPSI. The program in Figure F7-6 follows closely the flow chart of Figure 7-16. Remember that $f(x) = 1/x$.

```
      |DIMENSION T(101)
      |READ 105, EPSI
  105 |FORMAT(E20.8)
      |T(1) = 0.5*(1.0 + 0.5)
      |N = 1
    2 |S = 0
      |NL = 2**N-1
      |FNP = 2**N
      |DO 30K = 1, NL,2
      |FK = K
      |S = S+1.0/(1.0+FK/FNP)
   30 |CONTINUE
      |T(N+1) = 0.5*T(N) + S/FNP
      |IF(ABS(T(N+1) - T(N)) - EPSI) 9, 8, 8
    8 |N = N+1
      |GO TO 2
    9 |AREA = T(N)
      |PRINT 106, AREA
  106 |FORMAT(8H□AREA□=□,E20.8)
      |STOP
      |END
```

Figure F7-6

Note that we had to introduce FK as a floating point value of K and FNP as a floating point value of the number 2^N, in order to be able to calculate the desired terms in the calculation of S since FORTRAN does not permit us to mix modes in arithmetic expression.

Exercises F7-2

1. In the above program it is implicitly assumed that the calculation will terminate before N exceeds 100.

   (a) Is it possible for N to exceed 100?

   (b) What would happen if it failed to terminate before N exceeds 100?

   (c) Add some statements to the above program to protect against this undesirable event, even if the error tolerance is not satisfied. Print out a message in this case indicating failure to satisfy the error tolerance.

2. Criticize the above program for inefficiency. Revise it to make it more efficient by following the flow chart of Figure 7-17. Also include a "safety" termination if N exceeds 100. Run your revised program using first 0.01 and then 0.001 as values for EPSI.

3. Instead of terminating the calculation of the approximate area when two successive approximations differ in absolute value by less than EPSI, we could terminate the calculation after a fixed finite number of approximations have been calculated. Revise the program of the previous problem to read in an upper limit for the number of iterations to be carried out and then to terminate when this is reached. Run your program for N = 15.

4. Tell how to revise the program given in this section so that the calculation could be repeated for a series of values of EPSI each of which is read in from a card.

5. Write a FORTRAN program for the calculation described in Exercise 7-2, Set C, Problem 6, main text. Use $f(x) = 1/x$. Run your program and compare results using n = 5, 25, 75, 125, 200.

F7-3 Area under curve: the general case

We now consider the general case of finding an approximation to the area under a curve $y = F(x)$, above the x-axis and between the vertical lines $x = A$ and $x = B$. In order to make the program as useful as possible we could write it in the form of a function or a subroutine subprogram. We choose the former here. The function $F(x)$ is assumed to be defined as a FORTRAN function called FUNCT(X). An error tolerance EPSI is given and we terminate the calculation when the absolute value of the difference of two successive approximations is less than EPSI. We follow the flow chart of Figure 7-20.

```
      FUNCTION AREA(A,B,EPSI)
      M = 1
      H = B-A
      OLAREA = .5*H*(FUNCT(A)+FUNCT(B))
    3 M = 2*M
      H = H/2.0
      S = 0.
      DO 40 K = 1, M, 2
      FK = K
      S = S + FUNCT(A + FK*H)
   40 CONTINUE
      AREA = .5*OLAREA + H*S
      IF(ABSF(AREA - OLAREA) - EPSI) 9, 8, 8
    9 RETURN
    8 OLAREA = AREA
      GO TO 3
      END
```

Figure F7-7

If we want to use this function subprogram to calculate and print the approximate area under the curve $y = 1/x$, above the x-axis, and between the lines $x = 1$ and $x = 2$ we might use a tolerance of $EPSI = 10^{-4}$ and then we could write the following program:

```
      Z = AREA(1.0, 2.0, 1.0E-4)
      PRINT 4,Z
    4 FORMAT (7H AREA = E20.8)
      STOP
      END
C     PLACE THE DEFINITION OF FUNCTION AREA HERE
      FUNCTION FUNCT(X)
      FUNCT = 1./X
      RETURN
      END
```

Exercises F7-3

1. (a) Write a FORTRAN function subprogram AREA2(A,B,N) which calculates an approximation to the area under the curve $y = F(x)$, above the x-axis, and between the lines $x = A$ and $x = B$ and which uses a subdivision of the interval (A,B) into N equal parts. Follow the flow chart drawn in Exercises 7-3, Problem 1, of the main text. Test your program for $y = \sin x$ between $x = 0$ and $x = \pi$ with $N = 5000$. (How does your result compare with the area of a semi-circle of diameter π?)

   (b) Use function AREA2 to print out a table of natural logarithm values from 1 through 51 in intervals of 5.

2. Tell how the function subprogram AREA(A,B,EPSI) of this section may be adapted to protect against the possibility of an endless loop by causing termination of the calculation if the number of subdivisions exceeds N. If the calculation is terminated in this manner without satisfying the accuracy criterion, a message should be printed in addition to giving the approximation to the area.

3. Write a FORTRAN program which includes the subprogram AREA of this section and the subprogram AREA2 of Problem 1. Then use these subprograms to calculate approximations to the areas described below. First use 1, 2, 4 equal subdivisions of the interval and then use an error tolerance of $EPSI = 10^{-3}$. Of course you will need to supply the necessary FORTRAN function subprograms to define the functions:

   (a) Below $y = .43429/x$, above x-axis, between $x = 1$ and $x = 3$. (True area is $\log 3$)

   (b) Below $y = 3x^2 + 2x + 1$, above x-axis, between $x = -2$ and $x = 2$.

   (c) Below $y = x^3$, above $y = x^2$, between $x = 1$ and $x = 4$.

4. Write FORTRAN programs that may be used in calling on AREA(A,B,EPSI) to compute an approximate value of π to four decimal places. (See Problem 6, in Exercises 7-3 of the main text.)

F7-4 Simultaneous linear equations: Developing a systematic method of solution

In this section of the main text we explained carefully how to solve systems of two and three simultaneous equations. Exercises F7-4 provided examples of the method. You should now be ready to write a simple FORTRAN program for the solution of two simultaneous equations in two unknowns.

Exercises F7-4

1. Follow the flow chart drawn in Exercises 7-4, Set B of the main text, and write a corresponding FORTRAN program for the solution of two simultaneous equations in two unknowns:

$$a_{11}x_1 + a_{12}x_2 = b_1$$

$$a_{21}x_1 + a_{22}x_2 = b_2$$

2. Use the program of Problem 1 to solve the following systems of equations on the computer. Make a hand-calculated check of your computer results. For systems (f) and (g) slide rule accuracy is sufficient.

(a) $4x - 2y = 5$
$2x + y = 4$

(b) $4x + 3y = 5$
$2x - 4y = 7$

(c) $3x - 4y = 12$
$4x + 6y = 3$

(d) $2x + 4y = -7$
$3x + y = 2$

(e) $5x + y = 2$
$3x - 4y = 7$

(f) $3.124x_1 + 5.375x_2 = -1.234$
$10.246x_1 - 5.214x_2 = 3.714$

(g) $5.128x_1 - 3.874x_2 = 12.42$
$3.817x_1 + 15.157x_2 = 3.784$

F7-5 Simultaneous linear equations: Gauss algorithm

In describing the solution of three equations in three unknowns we described each of the essential operations in turn and drew a flow chart for each. It will be instructive to build up the FORTRAN program in the same gradual fashion. We begin by dividing the first equation through by a_{11} as described in Figure 7-24. The corresponding FORTRAN statements would be

```
      DO 100, J = 2, 3
      A(1,J) = A(1,J)/A(1,1)
100   CONTINUE
      B(1) = B(1)/A(1,1)
```

The elimination of x_1 from the i^{th} row, $i = 2, 3$ is described in Figure 7-25 and the corresponding FORTRAN statements would be

```
      DO 200, J = 2, 3
      A(I,J) = A(I,J) - A(I,1) * A(1,J)
200   CONTINUE
      B(I) = B(I) - A(I,1) * B(1)
```

Next we have to divide the new second equation by a_{22} and then eliminate x_2 from the third equation. Following these simple examples, you should have little trouble writing the FORTRAN that's equivalent to Figures 7-27 through 7-30.

Exercises F7-5 Set A

1. Write the FORTRAN statements corresponding to the flow chart of Figure 7-27.

2. Note the similarity between the statements of Problem 1 and those corresponding to Figure 7-24. Write a single set of FORTRAN statements to cover both cases by following Figure 7-28. (Remember that $k = 1$ or 2.)

3. Write the FORTRAN statements for the flow chart of Figure 7-30. Be sure to take account of the possibility that k may exceed 2.

4. Now write the FORTRAN that's equivalent to Figure 7-33.

Next we want to carry out the back solution in order to obtain x_3, x_2, x_1 in turn. This is described in the flow charts of Figures 7-34 and 7-35. The FORTRAN statements corresponding to the latter flow chart, Figure 7-35 might be:

```
      DO 900 I = 1, 3
      I1 = 4-I
      X(I1) = B(I1)
      IF(I - 1) 11, 900, 11
      I1M1 = I1 -1
      DO 1100 J = 1, I1M1
      J1 = 4 - J
      X(I1) = X(I1) - A(I1, J1) * X(J1)
 1100 CONTINUE
  900 CONTINUE
```

This set of statements appears to be somewhat more complicated than the flow chart of Figure 7-35. One reason is that in the flow chart the controlled variables i and j are decreasing whereas FORTRAN requires that controlled variables always increase. We take care of this by making the substitutions I1 = 4 - I and J1 = 4 - J. Another reason is that in most versions of FORTRAN a DO loop is always executed at least once. To avoid this difficulty we bypass the inner DO loop if I = 1.

Now just as the complete flow chart of Figure 7-33 was built up from partial flow charts, so we can build up the complete FORTRAN program corresponding to Figure 7-36 from the partial FORTRAN programs which we have just discussed and which you have written in Exercise F7-5, Set A.

Exercises F7-5 Set B

1. Write a complete FORTRAN program for the Gauss Algorithm given in Figure 7-36.

2. Run the above program on your machine and use the program to solve the systems of simultaneous linear equations represented by the following arrays.

(a) $3x + 4y + z = -7$
$2x + 4y + z = 3$
$3x - 5y + 3z = 7$

(b) $x + 2y - z = 4$
$3x - 2y + 4z = 1$
$x - 3y - 2z = 7$

(c) $4x - 2y - 3z = 7$
$3x - 5y + 2z = 1$
$2x + y + 2z = 1$

(d) $2x - y + 6z = 3$
$3x - 4y + 4z = 1$
$x + 2y - 5z = 7$

3. Now use the above program on your machine to solve these systems of equations

(a) $3.147x_1 + 2.419x_2 - 3.479x_3 = 4.219$

$6.241x_1 - 5.678x_2 + 4.271x_3 = -52.17$

$3.841x_1 + 5.761x_2 + 34.314x_3 = 27.14$

(b) $27.147x_1 - 3.417x_2 - 3.479x_3 = 5.617$

$31.468x_1 + 3.428x_2 + 4.719x_3 = 31.421$

$11.121x_1 - 3.171x_2 + 5.314x_3 = -17.121$

Solution of n equations in n unknowns

The generalization to n equations is quite easy if we follow exactly the pattern we just used for 3 equations.

Exercise F7-5 Set C

Revise your FORTRAN program for the Gauss Algorithm to handle n equations and n unknowns according to the procedure flow chart you prepared for Problem 2, 7-5 Set A of the main text. Test the procedure (subroutine subprogram) using the 4 by 4 system given in Problem 3 of 7-5 Set B in the main text. The procedure should be capable of handling up to 15 equations in 15 unknowns.

---

*Exercises F7-5 Set D

1. In Exercises 7-5 Set C of the main text you were asked to insert "partial pivoting" as a capability of your flow chart for the Gauss procedure. Show the corresponding changes necessary to the FORTRAN subroutine called GAUSS which you prepared in the previous exercise.

2. Complete the modification of GAUSS and use the new subroutine to solve the following systems of simultaneous linear equations. Compare results with and without partial pivoting.

(a) $3x_2 - 4x_3 = -4$

$3x_1 - 2x_2 + 4x_3 = 7$

$5x_1 + 15x_2 - 3x_3 = -4$

(b) $2x_1 - 3x_2 + 4x_3 = 7$

$4x_1 - 6x_2 + 13x_3 = 11$

$2x_1 - 7x_2 - 12x_3 = 1$

INDEX