ED 183 361 SE 029 526

AUTHOR	Fisher, D. D.; And Others
TITLE	An Introduction to Fortran Programming: An IPI Approach.
INSTITUTION	Oklahoma State Univ., Stillwater.
SPONS AGENCY	National Science Foundation, Washington, D.C.
PUB DATE	71
GRANT	NSF-GY-9310(EN)
NOTE	382p.
EDRS PRICE	MF01/PC16 Plus Postage.
DESCRIPTORS	Computer Oriented Programs; Computer Science; *Computer Science Education; Flow Charts; Higher Education; *Individualized Instruction; Input Output; *Pacing; *Programing; Programing Languages; Programing Problems
IDENTIFIERS	*FORTRAN Programing Language

ABSTRACT

This text is designed to give individually paced instruction in Fortran Programing. The text contains fifteen units. Unit titles include: Flowcharts, Input and Output, Loops, and Debugging. Also included is an extensive set of appendices. These were designed to contain a great deal of practical information necessary to the course. These appendices include instructions for operating card readers, listers, printers and terminals, as well as a computer science glossary. (MK)

P/PI

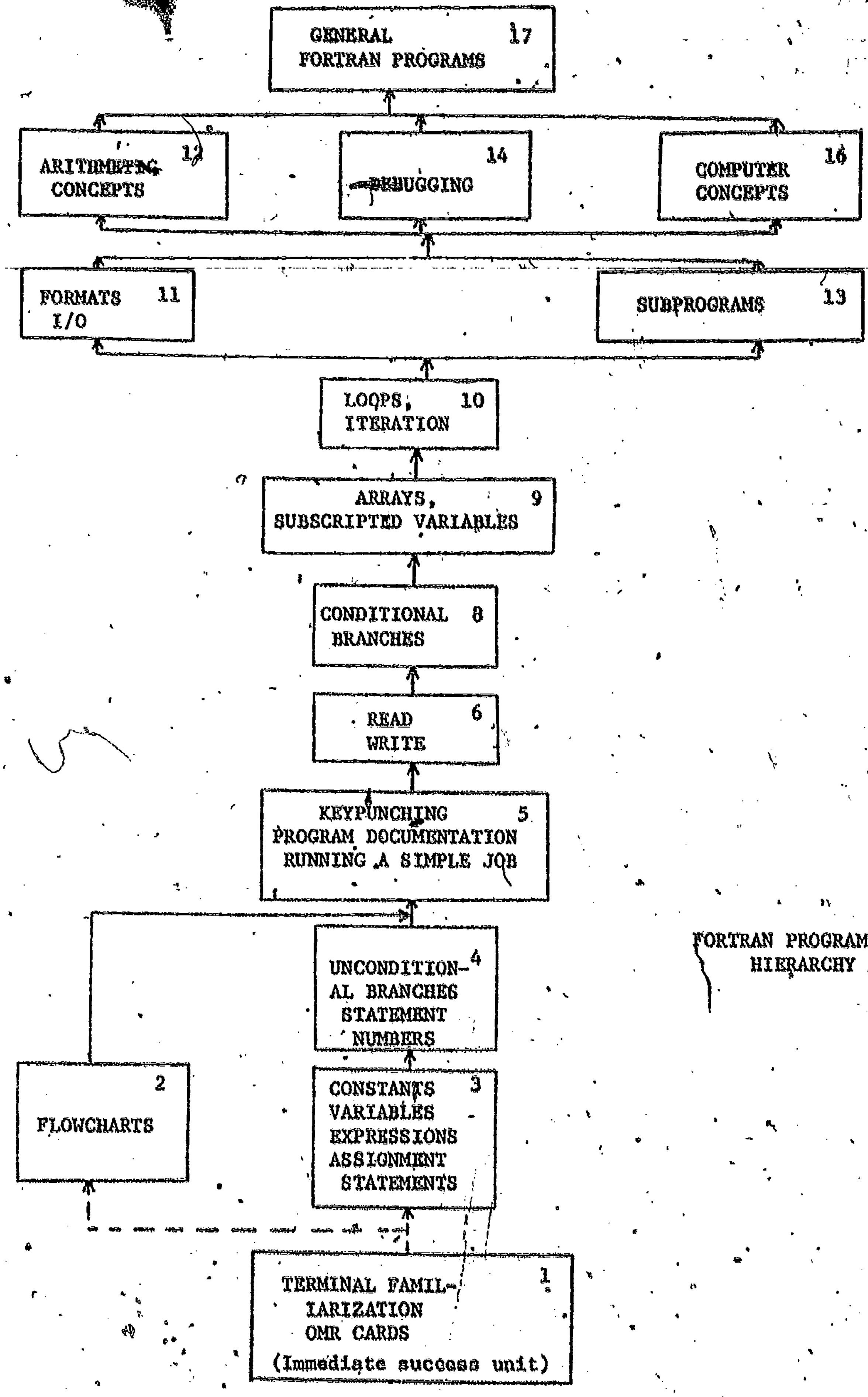
GY-9310 (EN)

Okla State U.
McCollom

INTRODUCTION TO FORTRAN PROGRAMMING IPI

AN INDIVIDUALLY PACED INSTRUCTION APPROACH

Oklahoma State University
January, 1972
Department of Computing and Information Sciences



GENERAL FORTRAN PROGRAMS 17
ARITHMETIC CONCEPTS 12
DEBUGGING 14
COMPUTER CONCEPTS 16
FORMATS I/O 11
SUBPROGRAMS 13
LOOPS, ITERATION 10
ARRAYS, SUBSCRIPTED VARIABLES 9
CONDITIONAL BRANCHES 8
READ WRITE 6
KEYPUNCHING PROGRAM DOCUMENTATION RUNNING A SIMPLE JOB 5
UNCONDITION-AL BRANCHES STATEMENT NUMBERS 4
FLOWCHARTS 2
CONSTANTS VARIABLES EXPRESSIONS ASSIGNMENT STATEMENTS 3
TERMINAL FAMIL-IARIZATION OMR CARDS (Immediate success unit) 1

FORTRAN PROGRAMMING HIERARCHY

An Introduction to Fortran Programming
An IPI Approach

D. D. Fisher
T. E. Bailey

D. H. Seeger

Oklahoma State University

Composed and Printed by the

Division of Engineering

Oklahoma State University

Stillwater, Oklahoma 74074

(Revised December, 1971)

PREFACE

The main objective of the Fortran programming units is to develop the necessary skills to

(i) translate a problem statement to a programming problem statement;

(ii) develop a debugged program that obtains a "workable" solution for the original problem;

(iii) prepare a description of the program so that others may make use of the program, or the author may use the program at a later date with a minimum investment in time;

(iv) accomplish the previous three skills with reasonable degrees of efficiency.

To accomplish (i) and (ii) it is necessary to learn a computer programming language and to practice the translation phase for a variety of problems. Preparing program descriptions and developing efficiency will be emphasized at each step. Several programs will be written that may be used as problem solving tools in subsequent courses.

A programming language is a language used to describe an algorithm, that is, a procedure for solving a problem, in such a way that a computer can interpret the algorithm and can carry out each of the steps prescribed by the algorithm. Programming languages vary in complexity. In general the languages that are easiest for the machine to use require considerable knowledge of the computer on the programmer's part. On the other hand, languages that are easiest for the programmer to use generally require the computer to translate the program from the source language to machine language. The translation process from source language to machine or object language is a task that computers do reasonably well; consequently, most of the programs written today are written in so-called "higher level" programming languages, which invoke the computer to translate as well as to compute.

Fortran (an acronym obtained from <u>formula</u> <u>translator</u>) is an example of a higher level programming language. It is the most widely used programming language for scientific applications. There are many variations or dialects of Fortran. You should be aware that different computer manufacturers support different dialects, one manufacturer may support different dialects, and users may create their own dialects. In order to be able to move programs from one computer installation to another, it is necessary to have a set of standards. Appendix I gives comparisons between USA standard Fortran and other available Fortrans. The USAS BASIC Fortran is a subset of USAS FULL Fortran. It is likely that you will discover useful features of some of the compilers that are not included in USAS Fortran. If you use those features, you do so "at your own risk." Some features not included in USAS Fortran are very desirable; consequently, if it is unlikely that the program will have to be run on different computers, it is worthwhile to make use of those features. On the other hand, if a program is to be run at several different computer installations, only those features listed in USAS Fortran should be used.

This is an example of one of many possible choices you may make while completing the computer science units. Other decisions will be required to get a program to "work," to transform a working program to an efficient program, to "tune" a program, etc. An aspect of computing many find challenging is that, in general, there is no best way to attack a new problem. You will find this to be true in some of the programs you attempt to develop.

Other higher level programming languages in widespread use are listed below.

Cobol	a business language
RPG	a report generator used primarily for business applications
PL/I	combines features of Algol, Fortran and Cobol
Algol	a scientific language
Basic	a version of Fortran used in conjunction with typewriter like terminals
APL	a mathematical language
Assembler	a form of machine language

Several undergraduate and graduate courses make use of these and other programming languages. Translators of varying sophistication are required to convert programs from these source languages to machine language.

Take some time to examine the organization of this book. The instructional materials are organized into units in the text. Each unit contains a descriptive title, a rationale explaining the purpose for including the material in the unit, the behavioral objective that states what you will be able to perform after you have mastered the material in the unit, prerequisites for the unit, a series of activities designed to point you in the direction of the objective and help you attain it, some self evaluation materials so that you can assess for yourself whether you have attained the objective, assessment tasks with which you demonstrate to your instructor that you have attained the objective, and finally instructions telling you what to do next.

The book also contains a rather extensive set of appendices. Read the titles of the appendices in the Table of Contents, and thumb through them. They contain a great deal of practical information that you will need throughout the course. Particularly at this time be aware of the appendix containing the glossary of terms. If terms which you do not understand appear in the text, look for them in the glossary. If there are omitted terms which you think should be included in the glossary, write them down and give them to your instructor.

The page numbers in the text are designed to be used for quick referencing. The first number is the unit number, and the second number numbers the page in sequence in that unit. For example, page 8.3 is page 3 or Unit 8. Appendices are numbered similarly except that Roman numerals are used. For example, page V.2 is page 2 of Appendix V.

The cover of the book shows the sequence of units for the course in the form of a <u>flow chart</u> (discussed further in Unit 2). This flow chart will help you to assess your progress toward the terminal unit.

Unit 1 requires a card reader with optical mark read (OMR) capability. If this feature is not available, then Unit 1 may be omitted without disturbing the sequence of the material.

TABLE OF CONTENTS

UNIT #1 (COMSC)

TITLE: Becoming Acquainted With the Computer

RATIONALE: If you're going to use a computer, you must know something about running jobs on the computer. You must familiarize yourself with some buttons and lights and noises. You must overcome any fears that the computer is out to get you or that it will harm you. You must also get some confidence and not be afraid that you will harm the computer.

But all these reasons are less important than the real rationale for this unit. We want you to have some fun playing with the computer! We want you to enjoy pushing the buttons, seeing the lights, and hearing the noises! Relax, go through this unit, and have a great time!

OBJECTIVE: At the end of this unit you will be able to demonstrate that you can run a job through the computer, including operating the card reader and line printer, using control cards correctly, and marking OMR cards.

PREREQUISITES: A willingness to try, a soft lead pencil, and some confidence.

ACTIVITIES:

A. OMR Cards

We'll begin with OMR computer cards. OMR stands for the Optical Mark Read feature available on the computer card reader. The OMR Fortran card is shown in Figure 1.1.

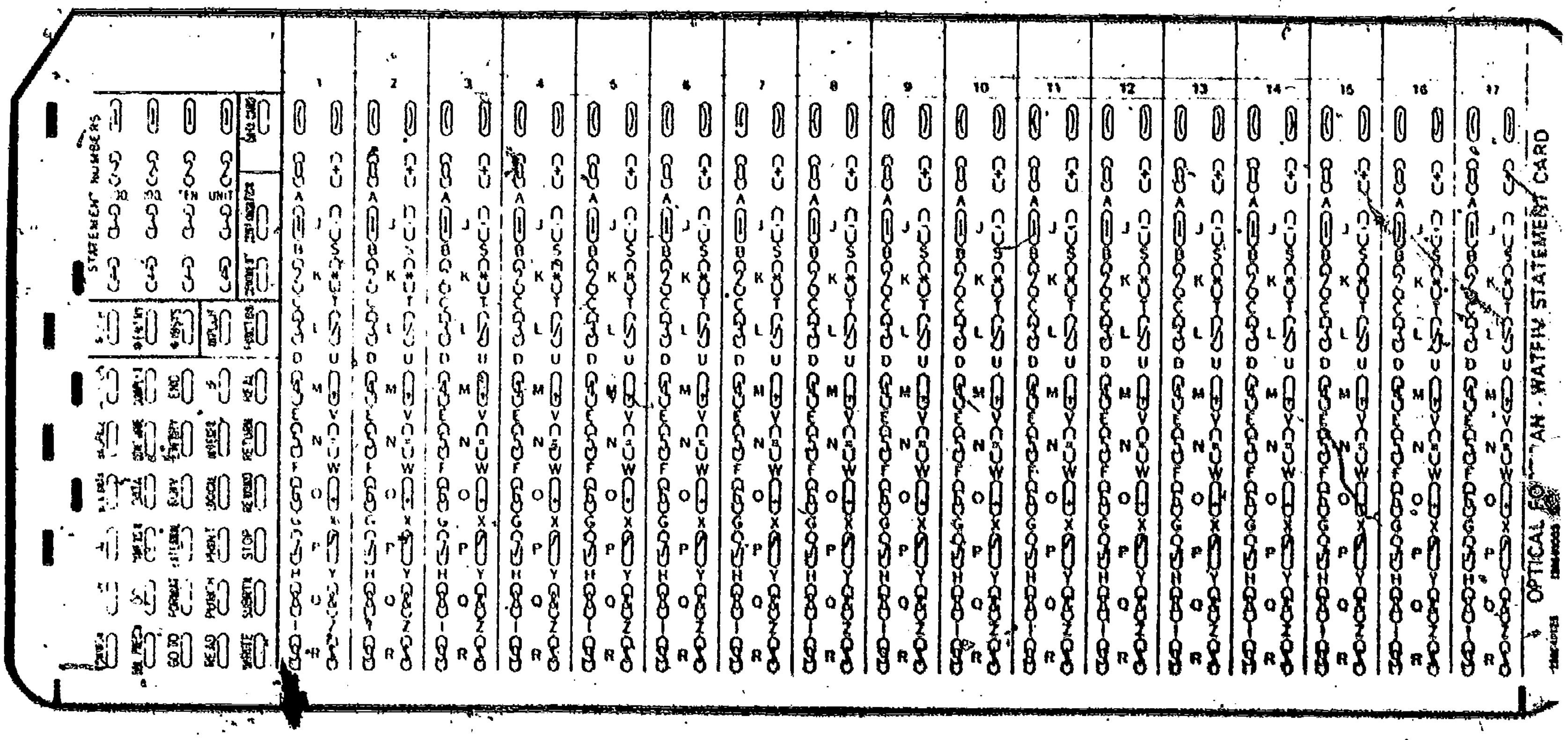


Figure 1.1 OMR Fortran Card

The left-hand side of the card is enlarged and is shown in Figure 1.2.

The card is designed so that most key words -- such as WRITE, FORMAT, STOP, and END -- can be obtained by a single pencil mark in the box associated with each word in the lower left-hand corner of the card. Statement numbers are marked in the upper left-hand corner of the

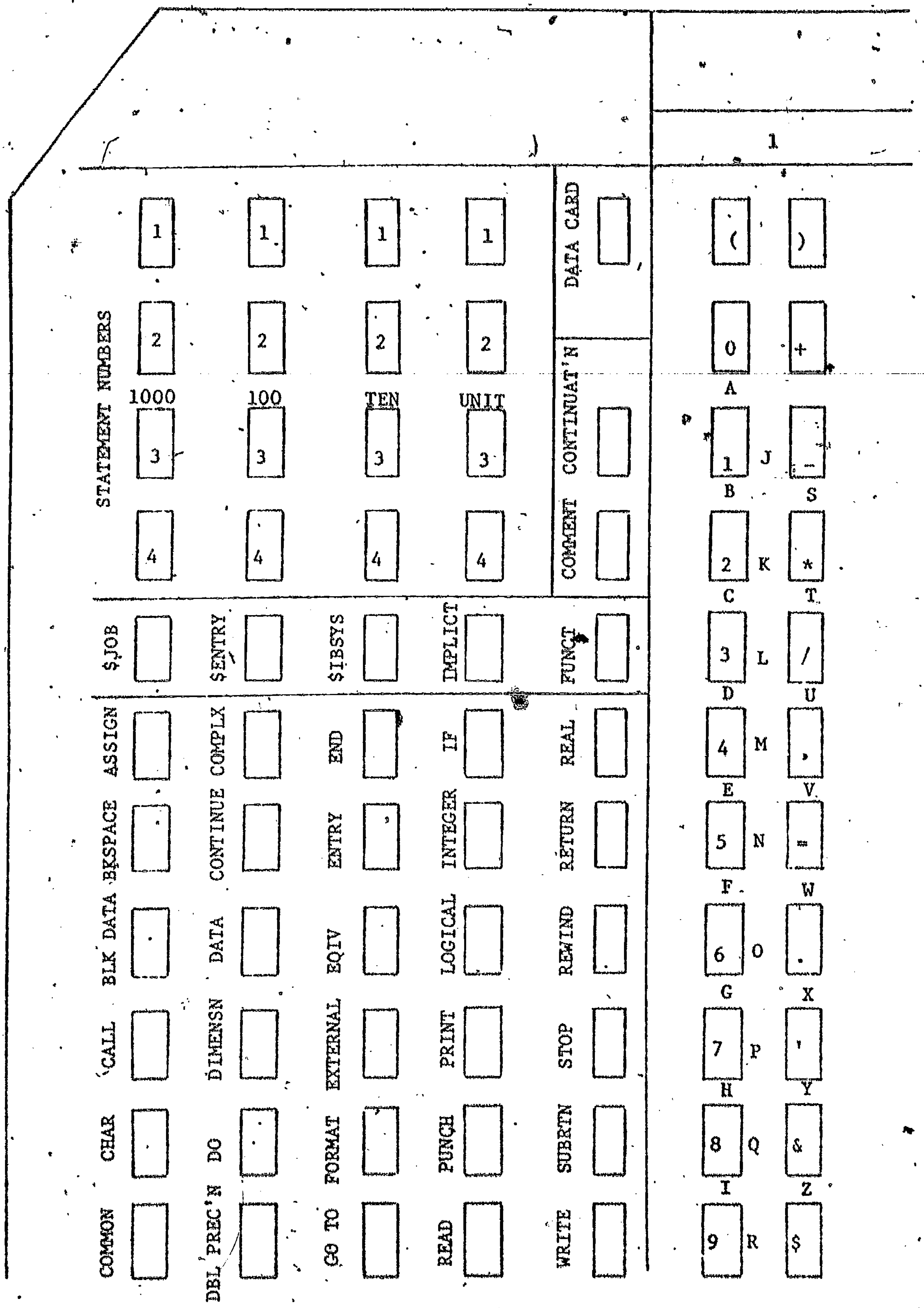


Figure 1.2 Enlargement of the left end of the OMR Fortran card

card. Comments (not actually a part of the Fortran program itself) may be inserted anywhere in the program by marking the COMMENT box and then marking the message on the rest of the card.

The main portion of the card is used for various Fortran statements. Here numerals and special characters require only a single mark. The letters all require two marks, either in positions above and below the letter or on both sides of the letter. The card is designed so that you may write in the boxes along the top of the card the characters marked on the card, thus simplifying marking the card and providing the information marked for later reference.

It should be emphasized that ordinarily a single pencil mark in the box with a soft lead pencil (No. 2, for example) is sufficient. It is not necessary to black in the box completely. Sometimes marks that are too heavy will be misread by the card reader. Ball point pen marks are ignored, as are marks from most other writing instruments. Pencil marks may be erased, provided that they are erased very thoroughly without damaging the card. Be sure to brush the eraser fragments off the card; it must be clean; else it may clog up the card reader.

B. Examples of marked OMR cards

Examine the card in Figure 1.3. What is marked in the card is written along the top: Column 1 contains the character "A"; column 2 contains "="; column 3 contains "B"; etc.

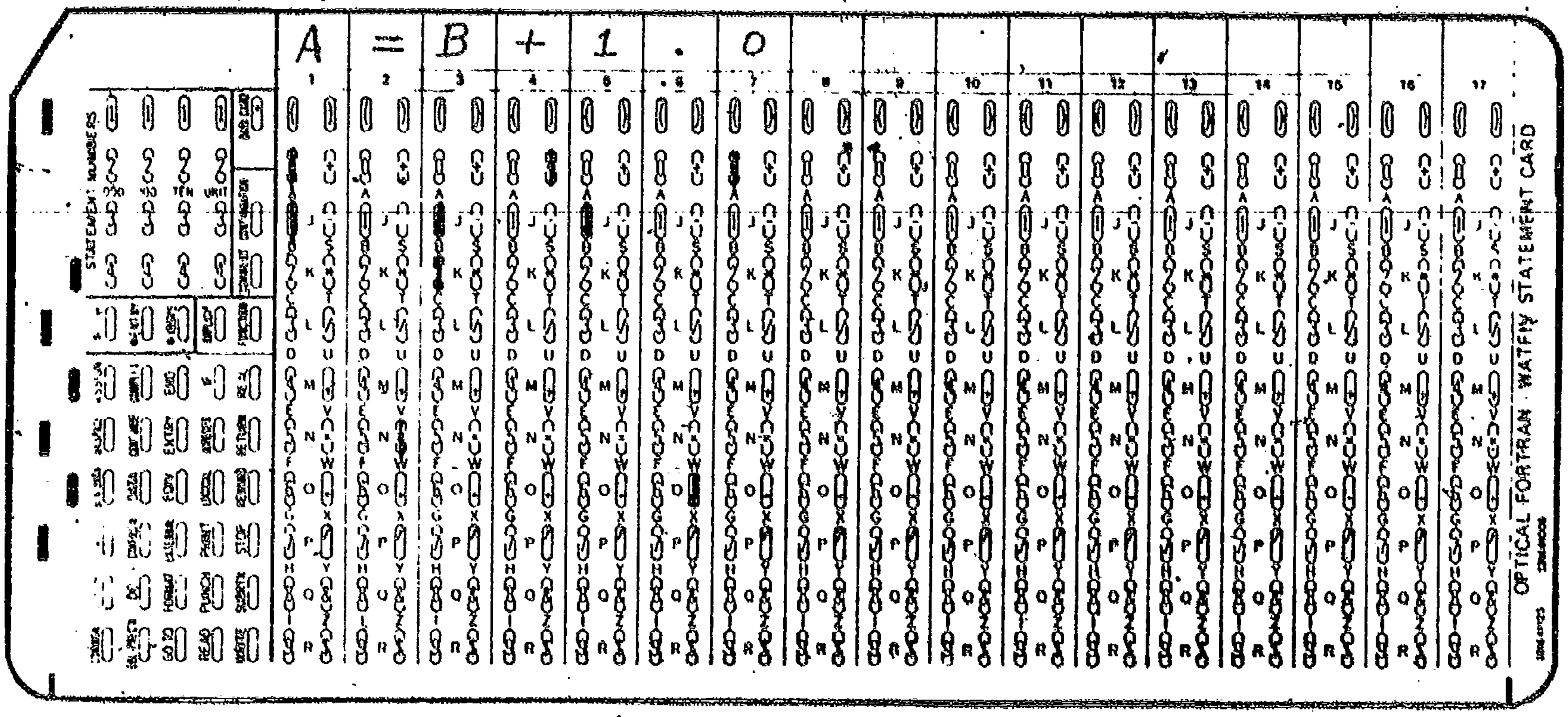


Figure 1.3 Example of a marked OMR card

Examine the card in Figure 1.4. Again, what is marked in the card is written along the top. Notice that the keyword "WRITE" is marked in the lower left corner. Then column 1 contains "("; column 2 contains "6"; column 3 contains ","; etc.

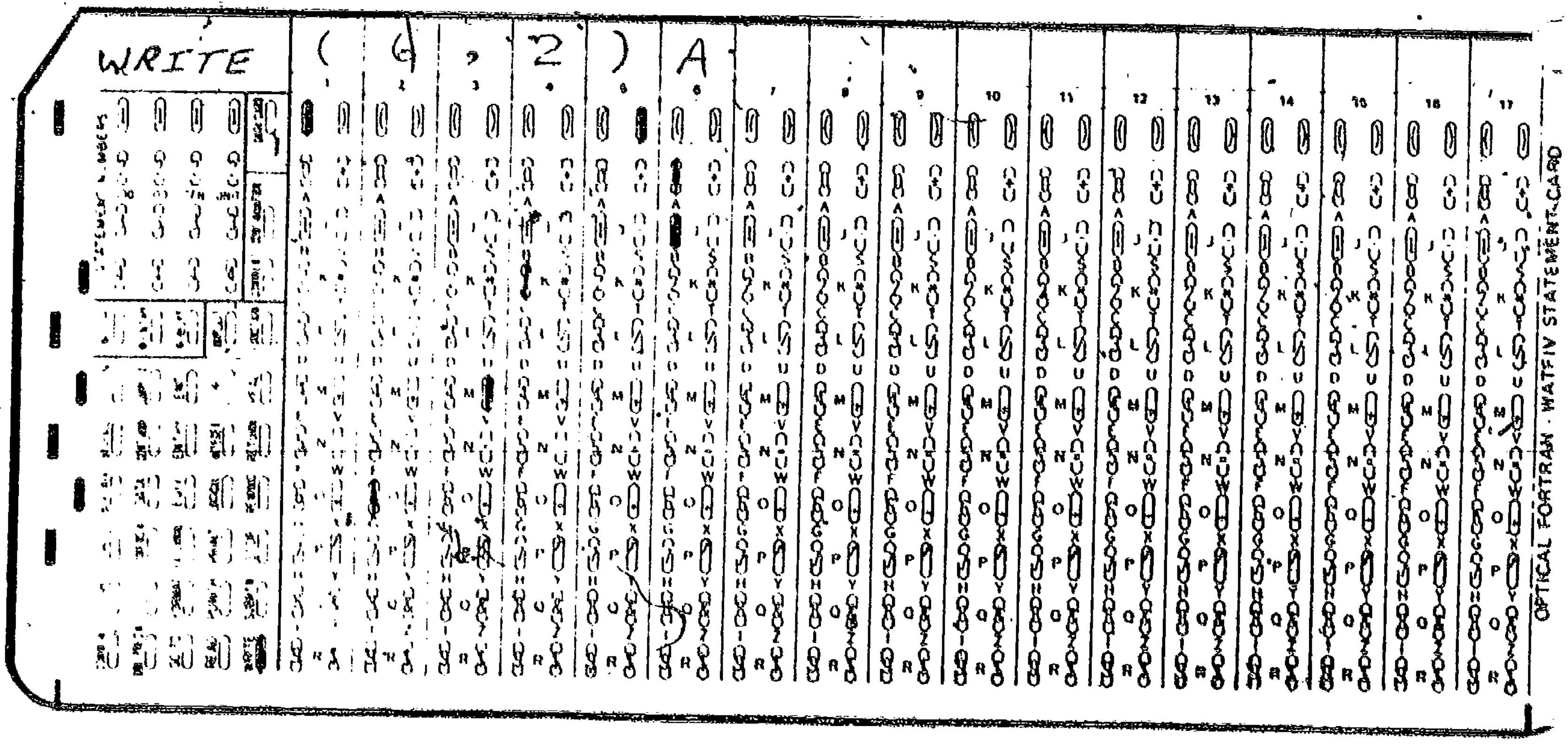
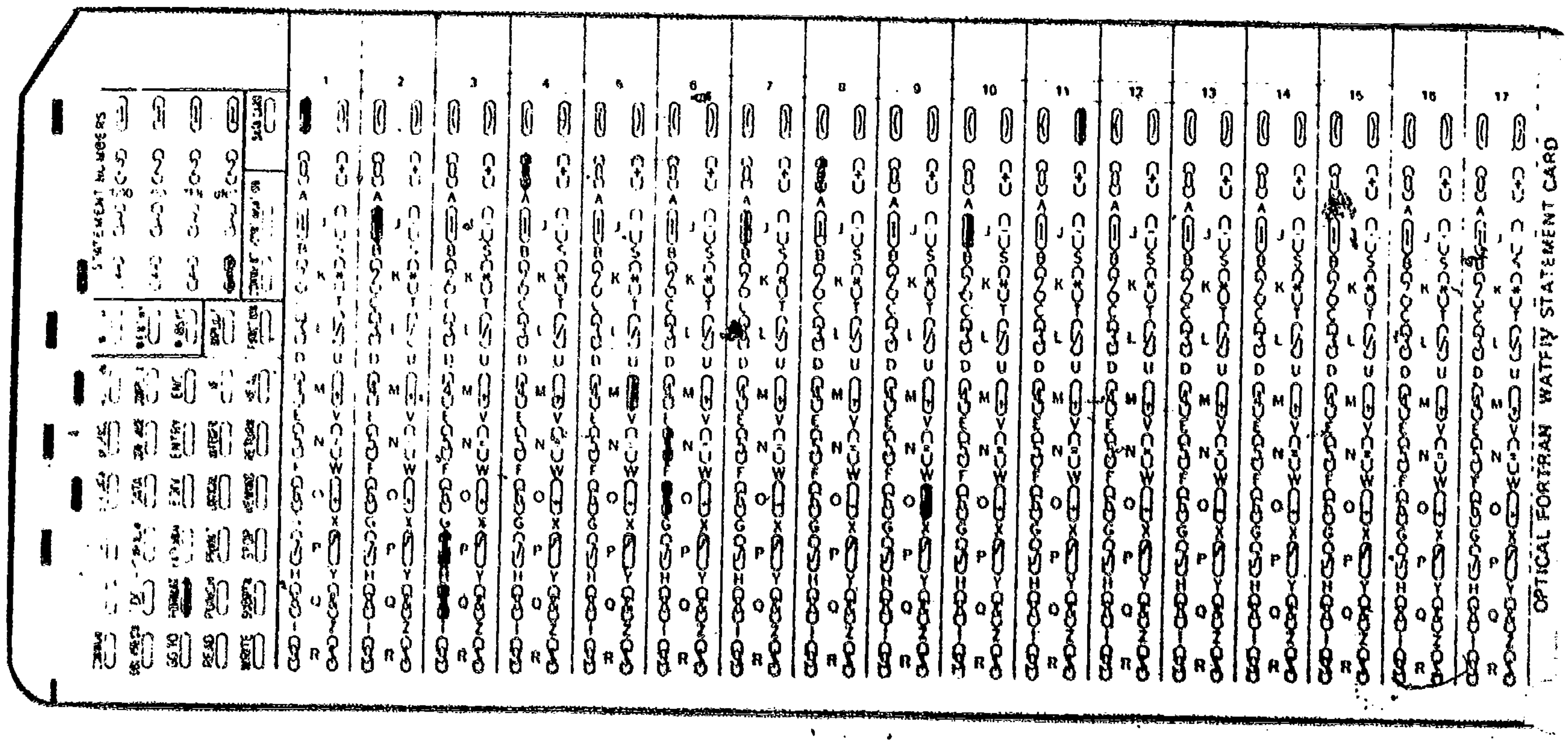


Figure 1.4 Sample OMR card with keyword WRITE marked

SELF EVALUATION: Look at the card below and write in the boxes at the top of the card what is marked in the card.



The following statement is marked on the card:

```
4 FORMAT(1H0,F10.1)
```

The "4" is marked in the block of statement numbers in the upper left-hand corner; it is marked in the "Units" column, meaning that it is 4, rather than 40 (tens column) or 400 (hundreds column) or 4000 (thousands column). The keyword FORMAT is marked in the keyword block in the lower left-hand corner of the card. The rest of the statement is marked in columns 1-11 of the card: "(" in column 1, "1" in column 2, "H" in column 3, etc.

ASSESSMENT TASK: You are to mark a simple Fortran program on OMR cards and run it on the computer. The program and directions are given in the UNIT #1 ACTIVITIES TABLE, APPENDIX IX. When you finish this task, take your program deck and the printed output produced by your program deck to your instructor for evaluation. You will not be allowed any errors on the printed output. Don't worry about this seemingly stiff requirement; you can meet it.

WHAT NEXT? You may proceed with either UNIT #2 or UNIT #3, or both.

UNIT #2 (COMSC)

TITLE: Flowcharts

RATIONALE: A flowchart (also spelled as two words, flow chart) is a plan by which a particular problem is to be solved. We stress the use of a flowchart to describe a plan for organizing a computer based algorithm, or procedure for solving a problem, because for most useful algorithms there are many different ways to accomplish the necessary computations. A flowchart pins down one of these many possibilities. A flowchart gives us a record of what we had planned to do. Before we decide to yield to the temptation to make a change in the solution procedure, we will be able to make a comparison of the proposed modification with the original. In some cases a change will be desirable, in other cases a catastrophe. By spending more time in the planning stages, one usually reduces the debugging time and the overall project time.

OBJECTIVE: When you finish this unit, you will be able to construct a plan or procedure for solving problems of a general type, using both a boxed flowchart and a line by line flowchart.

PREREQUISITES: UNIT #1, if required.

ACTIVITIES: We'll begin with a fairly detailed discussion of flowcharts, and then you can try your hand at constructing some yourself.

A. What is a flowchart?

The concept of a flowchart is completely general. A flowchart is simply a step by step plan or procedure, usually given in some graphic form, for accomplishing a goal or set of goals. A "treasure hunt" map is a flowchart. A game board like the one used in Monopoly or Aggravation is a type of flowchart. A set of directions for assemblying a model car or airplane is a flowchart. Even the procedure that you follow in your early morning getting up routine is a flowchart, though you probably don't have a copy of it in a graphic form.

Do you get the idea? Any multistep procedure can readily be generated from a flowchart, whether it be traveling from place A to place B (the flowchart might be a map), writing a paper for an English course (the flowchart might be called an outline in this case), carrying out a laboratory experiment (called a procedure) or writing a computer program.

The level of detail required in a flowchart varies with the situation and procedure being flowcharted. In some cases, it may be necessary to present every step in detail; in other cases, many steps may be summarized in a single line or statement.

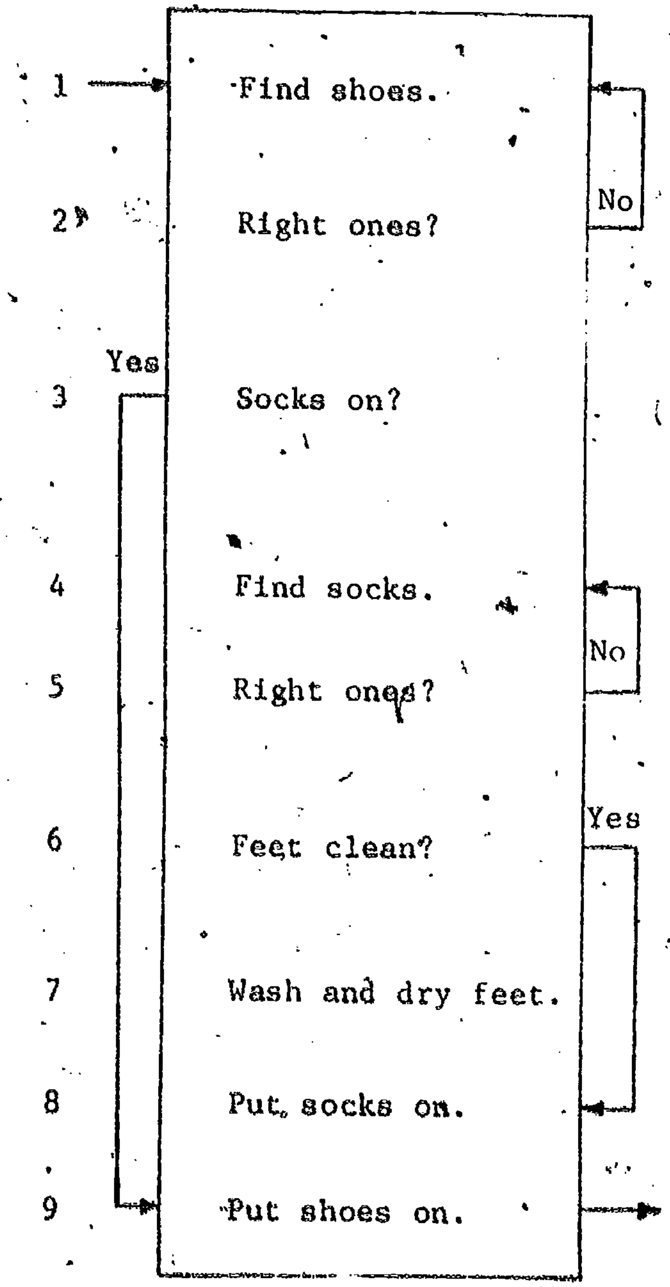
Now that you have a general notion of flowcharts, let's go on to some specifics of flowcharting as used in a simple example.

B. Two types of flowcharts.

Although many notations exist for flowcharting, only two will be used here. One method consists of placing the steps in various kinds of boxes; the other consists of a line by line representation. The basic reason for using a flowchart is to indicate the possible alternatives considered by the solution procedure and the conditions under which these alternatives are pursued.

First, consider a rather trivial, but practical example -- the procedure for putting on your shoes and socks (assuming, of course, that you do wear shoes and socks). Examine Figure 2.1 where two flowcharts are given which describe a procedure for shoeing your feet.

You can probably read and interpret the "flow" of the procedure without any trouble. You'll learn about the meanings of the symbols later. If you are sure that you understand how to read the two flowcharts and you understand what the procedure



Begin where arrow comes into box on line by line flowchart or at "Start" on boxed flowchart. Find shoes.

If they aren't the right ones, keep looking until you find the right ones.

When you've found the right ones, check your socks. If they're on, then put your shoes on.

If your socks aren't on, then find them.

If they aren't the right ones, keep looking until you find the right ones.

When you've found the right ones, check your feet. If they're clean, put your socks on, then your shoes.

If they're not clean, wash and dry your feet.

Put your socks on.

Put your shoes on.

Stop when arrow goes out of the box on the line by line flowchart or with "Stop" on the boxed flowchart.

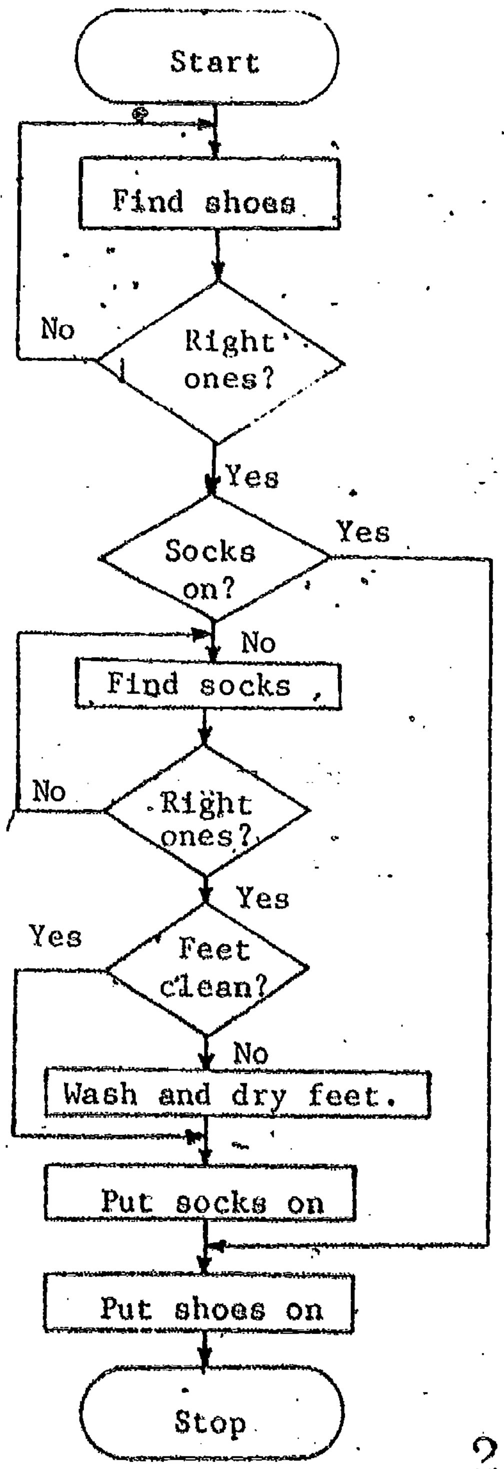


Figure 2.1 A line by line flowchart (left) and a boxed flowchart (right) describing a procedure (center) for putting on shoes and socks.

is for showing your feet as described by the flowcharts, then proceed with Activity C.

C. A numeric example.

Let's take an example more nearly related to computer programming -- finding the sum and average of N values, $Y_1, Y_2, Y_3, \ldots, Y_N$.

The sum is defined by

$$SUM = Y_1 + Y_2 + Y_3 + \ldots + Y_N$$

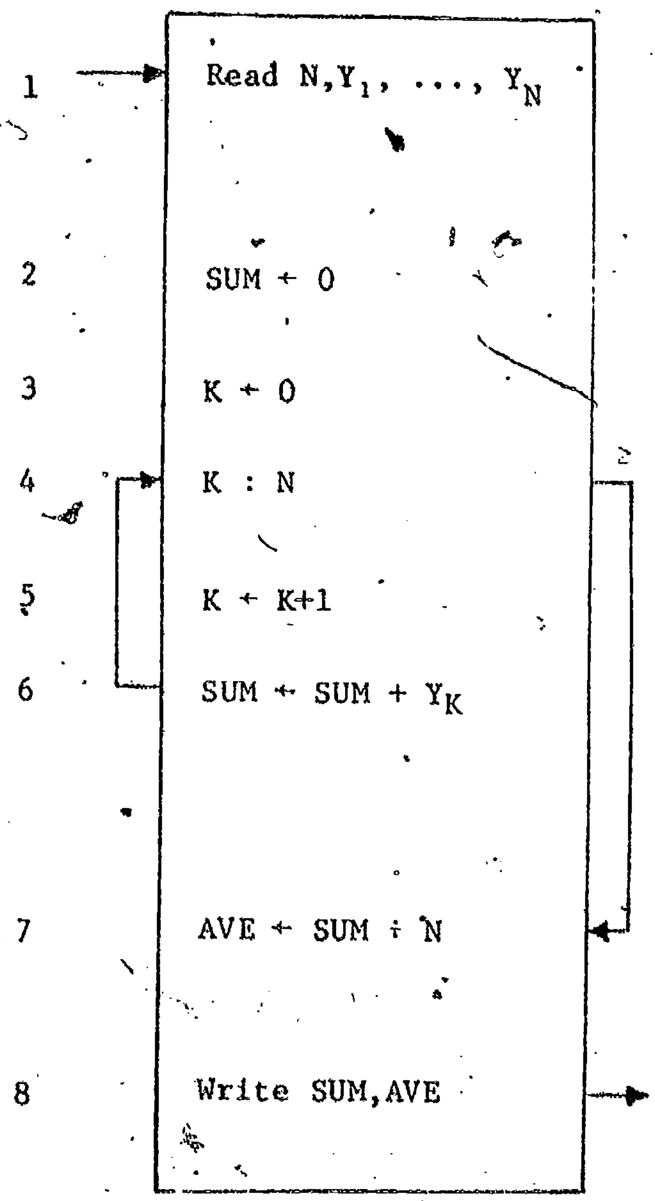
and the average is defined by

$$AVE = SUM \div N.$$

A procedure for performing these calculations is described by the flowcharts in Figure 2.2 along with an explanation of the operations represented by the flowcharts. After you have examined the flowcharts, then we'll discuss each one in greater detail.

D. Discussion of the line by line flowchart.

In the "line by line" flowchart, the flow of computation is from one line to the next, unless a branch occurs, which is indicated by an arrow emanating from the line. An unconditional branch is indicated by an arrow without conditional indicators. A conditional branch is indicated by an arrow or arrows with conditional indicators, such



Obtain the numbers that will be used in the computations.

Set the summer to zero. (← means "is replaced by") This is analogous to clearing the dials on a desk calculator, resetting them to zeros.

A counter K is set so that the N values of Y can be added one at a time.

K is compared to N. If $K \geq N$, then all the values of Y have been added, and the calculation of the average is next.
When $K \ngeq N$, then the counter is incremented,

and the next value of Y is added, after which the counter is checked again. As long as $K \ngeq N$, repetition of the adding operation is continued; this is called <u>looping</u>.

When the answer to the question $K \geq N$? is yes, then the average is calculated.

The results of the calculations are recorded for later use, and that's all there is to do!

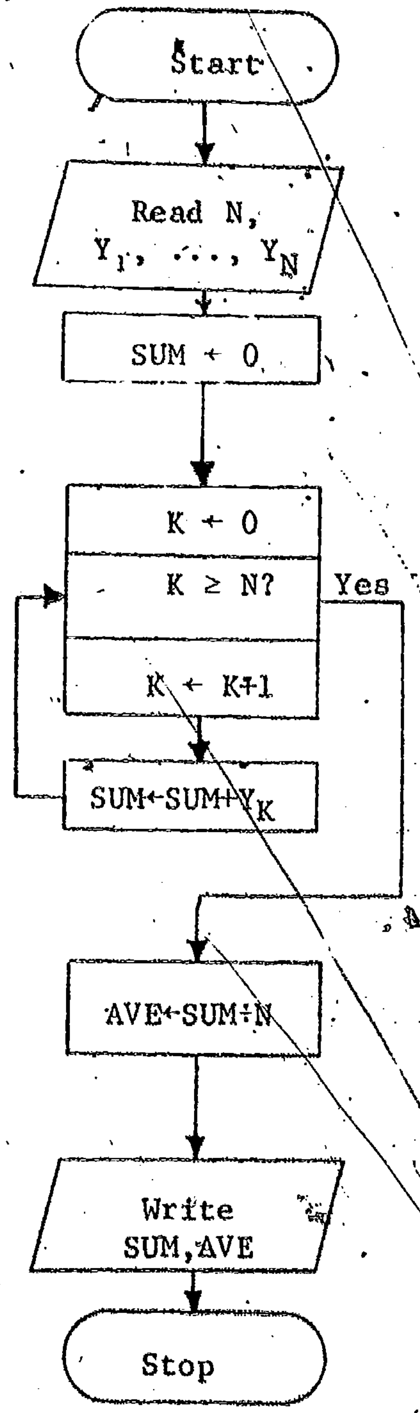


Figure 2.2 Line by line (left) and boxed (right) flowcharts describing a procedure (center) for calculating the average of a set of N numbers.

as $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, emanating from a statement of the form

expression1 : expression2

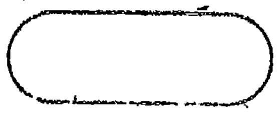
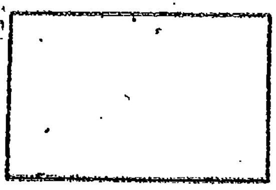
At line 4 the statement which follows the statement

K : N

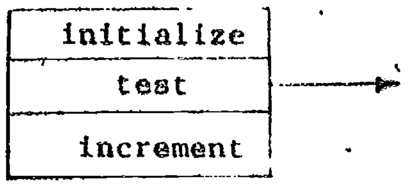
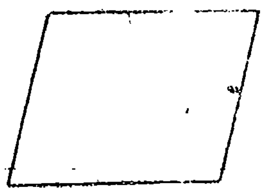
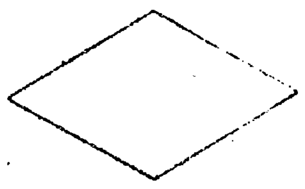
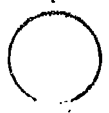
as long as $K < N$ is on line 5. When $K \geq N$, the statement at line 7 follows the one at line 4. Entry to the routine is indicated by the unattached arrow at line 1; exit from the routine is indicated by the arrow at line 8.

E. Discussion of boxed flowchart.

In the boxed type flowchart, each statement or idea is placed in one of six types of boxes. The shape of the box identifies a specific function that is to be performed. Box types and their functions follow.

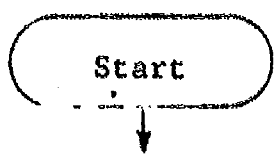
BOX TYPE	FUNCTION
	initiate or terminate a program segment or algorithm
	execute a process or perform a calculation

2.8

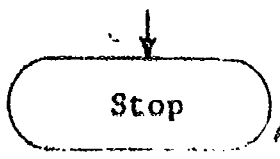
BOX TYPE	FUNCTION
	perform loop process .
	perform input or output
	make a decision
	connect to another sequence

The sequence of steps, or flow, in the flowchart is indicated by arrows which connect boxes. Unlabeled arrows are called <u>unconditional branches</u>. Arrows with labels designate <u>conditional branches</u>, allowing the flow to proceed along alternate pathways, the conditions being stated by the labels on the arrows.

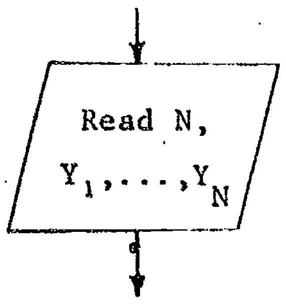
Now, let's take each block of the boxed flowchart separately.



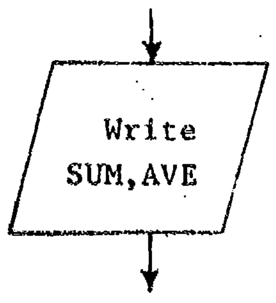
Obviously, this tells you where to start in the flowchart.



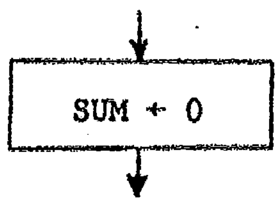
It's also obvious that this box tells you where to stop or terminate the procedure. This box must be at the logical end of the flow through the flowchart.



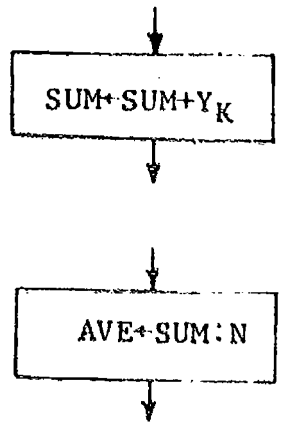
This box indicates that we want the computer to get the value of N and all the Y's from some device which can supply the computer with information or data. Our concern at this time is not how this is accomplished by what kind of device; rather the general notion of putting information into the system from some external source is the main concern. (This is analogous to a person's putting data into a calculator through the keyboard, the person himself being the external source.)



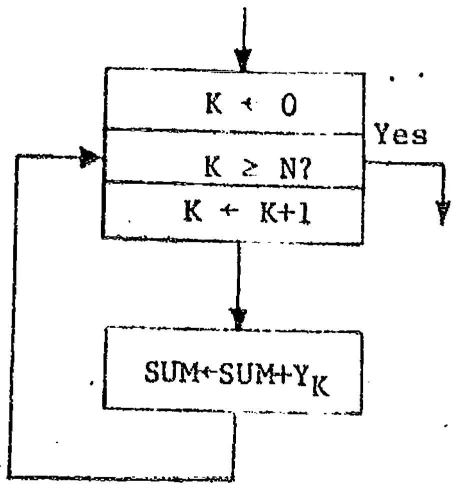
Similarly, this block instructs the computer to write the results on some external output device. As in the case of the read box, the main concern now is not how this is done; just the general notion that it is done is important. (This is analogous to the printing of results on paper tape by a desk calculator or cash register.)



All of these boxes represent some process, including calculations in two cases. The



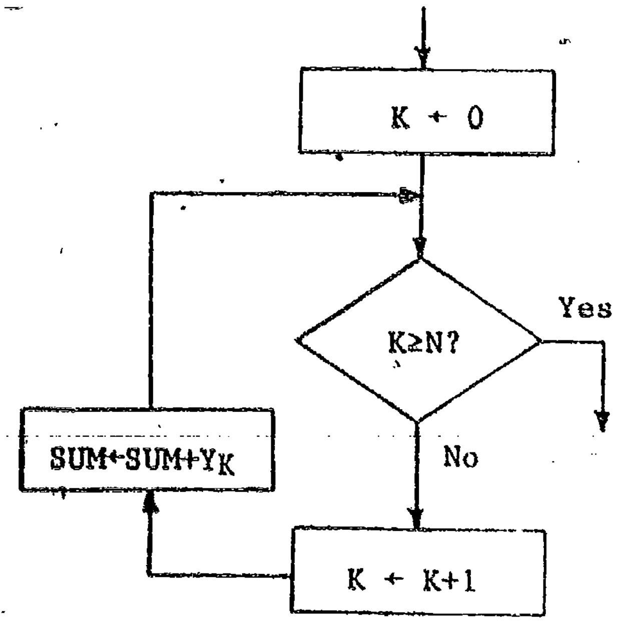
first box says, "Replace what is in SUM by zero." The second box says, "Replace SUM with the current value of SUM added to Y_K." The third box says, "Replace AVE with SUM divided by N."



This box sets up and controls a loop process. The summing box is to be repeated N times. The counter for the loop is given an initial value of zero. The next statement in sequence is $K \geq N$? As long as the answer is "no," the next statement in sequence, $K \leftarrow K+1$, is executed, followed by the summing process. Then the value of K is compared to N again, and the process is repeated as long as K is not $\geq$ N. But, when K is $\geq$ N, then the looping will be terminated and the next box in sequence will be executed -- in this case, $AVE \leftarrow SUM \div N$.

Notice that the loop process box has more than one arrow coming out of it; one, the one labeled "yes," is a conditional branch. Notice also that the box contains three distinct steps: _initializing_ K, _testing_ K against N, and _incrementing_ K.

The loop process could also be represented by the symbols shown on the next page with



the three steps -- initializing, testing, and incrementing -- shown separately. Here the decision box is used. Like the loop process box, the decision box has more than one exit from the box. Two conditional branches occur in this case, one labeled "yes" and one labeled "no." Sequences which set an initial value and test and increment the value occur so frequently, however, that they are usually combined into one loop process block. Generally you should expect to use the loop process box for initialize-test-increment sequences.

F. Some additional examples of boxed flowcharts.

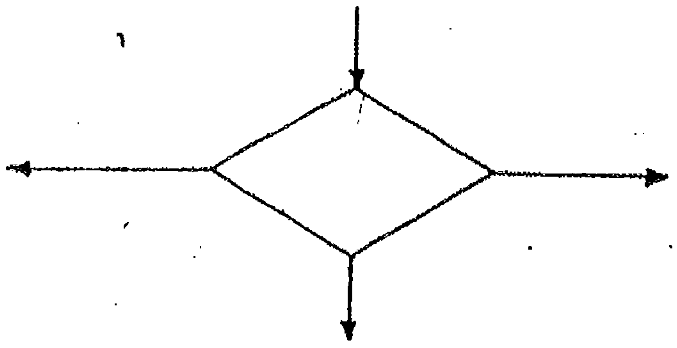


Figure 2.3
Decision Box

The box shown in Figure 2.3 may be used to indicate branching alternatives such as

(i) less than zero (<0), equal to zero (=0), greater than zero (>0) (three way branch)

(ii) less than or equal to zero (≤0),

greater than zero (>0) (two way branch)

(iii) etc.

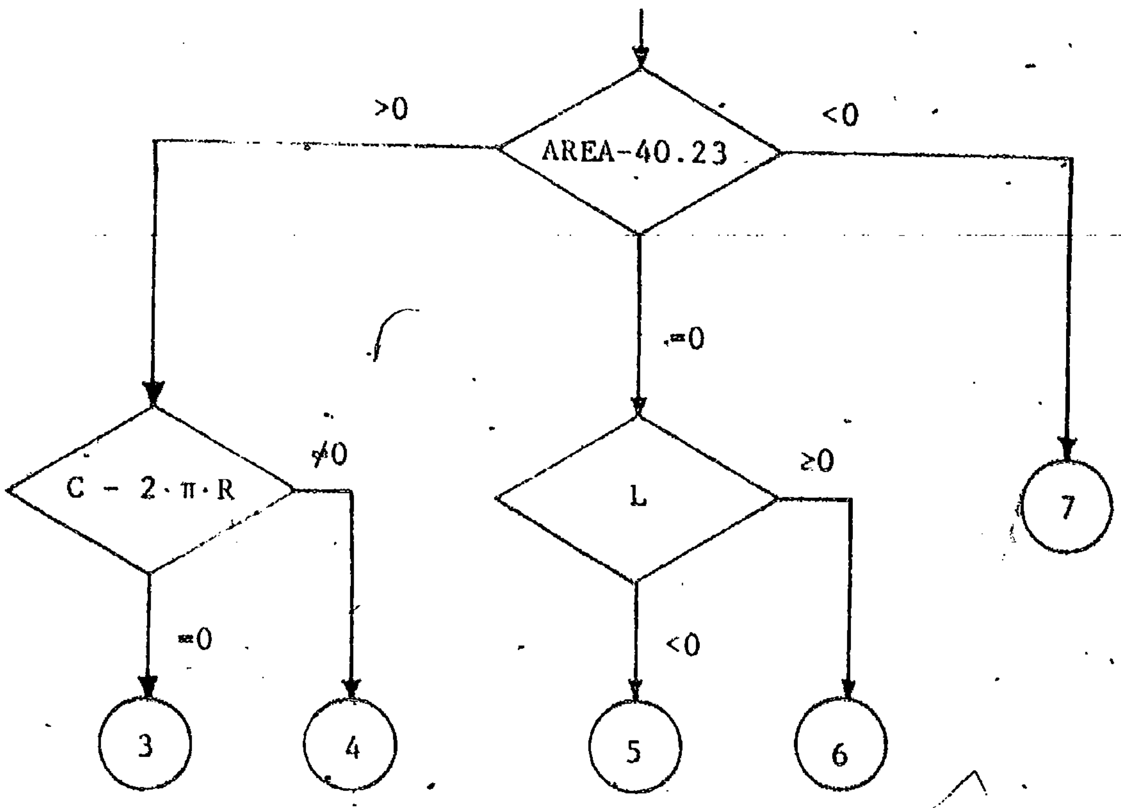


Figure 2.4
Decision Box Examples

In Figure 2.4 examples of two way branches and three way branches are given. If

$$AREA - 40.23 < 0$$

the next box in sequence is the one indicated by (7). Note that

$$AREA - 40.23 < 0$$

and

$$AREA < 40.23$$

are two different ways to state that the quantity called AREA is less than 40.23. The other conditions require additional tests before the next box in sequence is determined. These choices are illustrated in Table 2.1.

FIRST CONDITION	SECOND CONDITION	NEXT BOX
$AREA - 40.23 < 0$		7
$AREA - 40.23 > 0$	$C - 2\pi R = 0$	3
$AREA - 40.23 > 0$	$C - 2\pi R \neq 0$	4
$AREA - 40.23 = 0$	$L < 0$	5
$AREA - 40.23 = 0$	$L \geq 0$	6

Table 2.1
Branching Alternatives for Figure 2.4

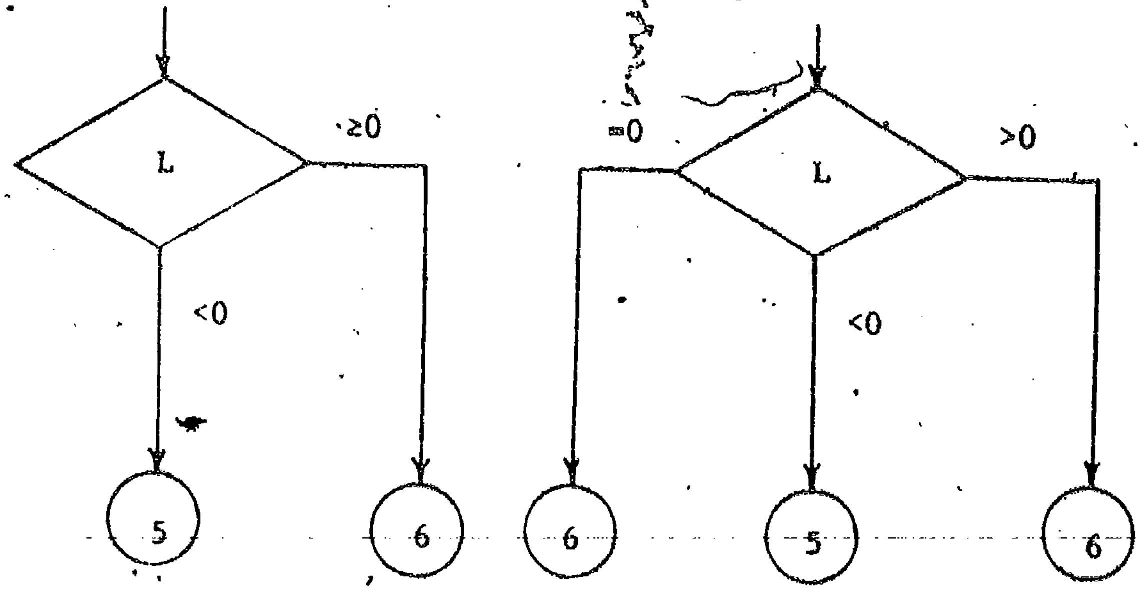


Figure 2.5
Alternate Representations of Two Way Branches

Figure 2.5 illustrates different ways to represent the same alternatives.

For two way branches, it is possible to state the question in terms of logical relations rather than arithmetic relations. Figure 2.6 illustrates this slightly different notation.

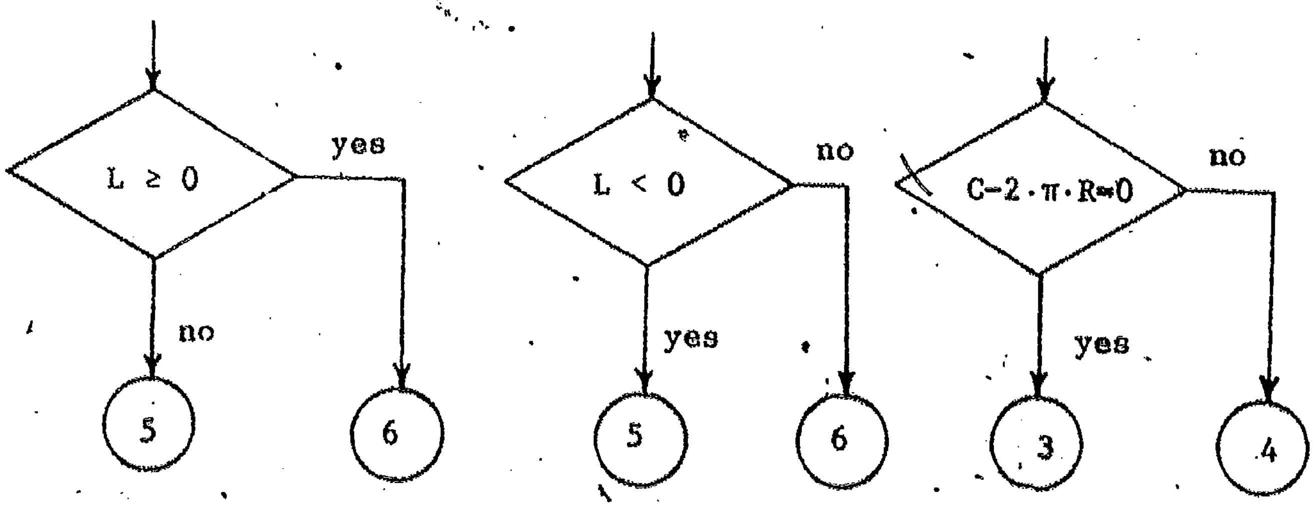


Figure 2.6
Logical Representation of Two Way Branches

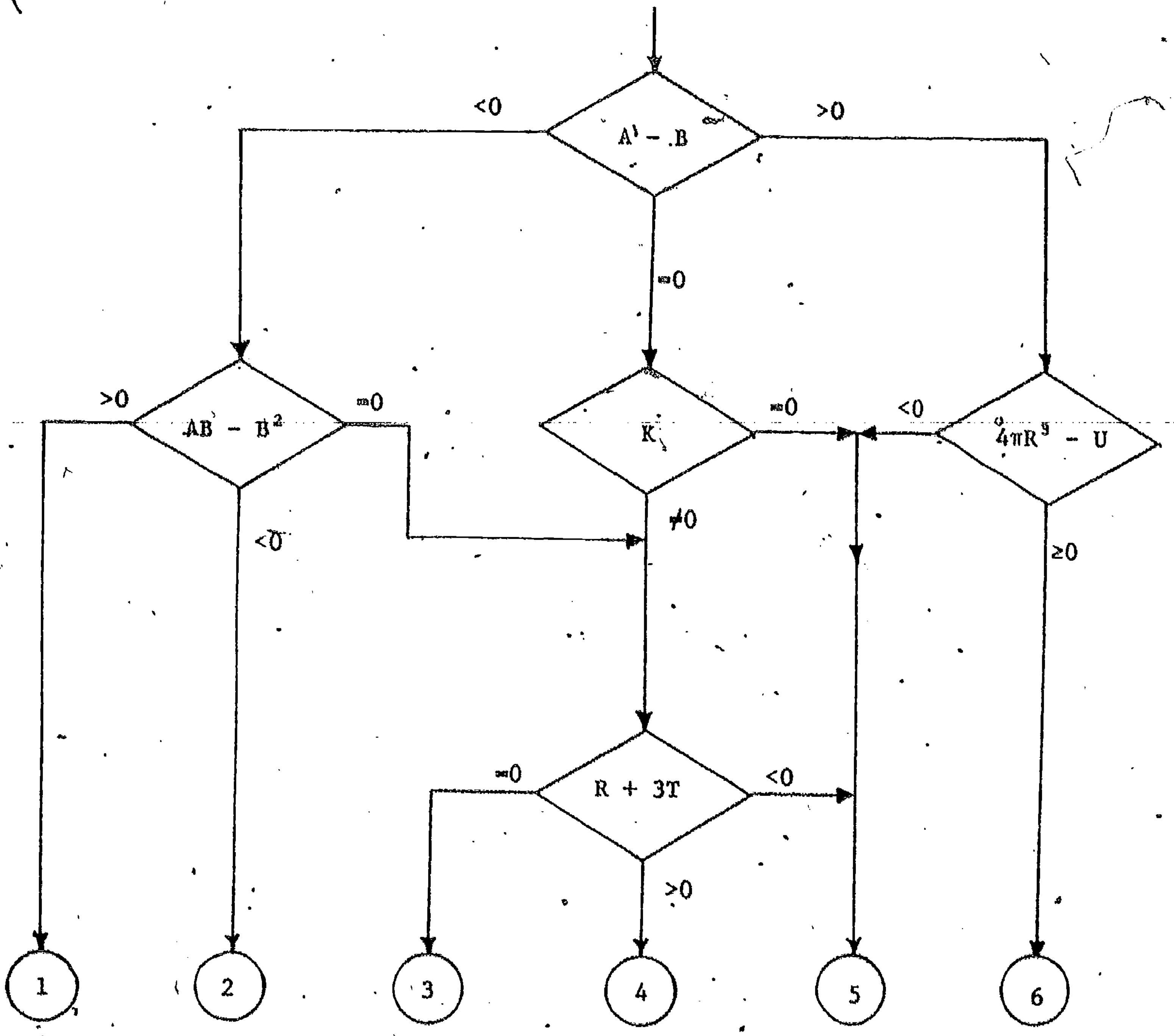


Figure 2.7
A Branching Maze

Fill in Table 2.2 to correspond to the branching alternatives for Figure 2.7.

FIRST CONDITION	SECOND CONDITION	THIRD CONDITION	NEXT BOX

Table 2.2
Branching Alternatives for Figure 2.7

G. Your textbook may have additional information that is helpful. Do Activity 1 in the UNIT #2 ACTIVITIES TABLE.

SELF ASSESSMENT:

A. Construct a line by line flowchart for Figure 2.4.

B. Construct a line by line flowchart for Figure 2.7.

C. Construct a boxed flowchart containing of at least three decisions describing how you spend Mondays during the current semester.

D. Construct a line by line flowchart containing at least three decisions describing your enrollment in the University this semester.

E. Construct a boxed flowchart containing at least two decisions describing how to change a flat tire -- or how to get help, if you don't know how to change it.

F. Construct a line by line flowchart consisting of as few operations as possible to determine which of eight objects was of different weight if seven of the eight had identical weights and the eighth was of different weight, using only a beam balance. Your solution must also determine whether the object was heavier or lighter than the other seven.

G. If you are enrolled in the "PIPI Package," then construct a line by line flowchart to describe the techniques of preparing a report as described in Communications Unit 5.

See your instructor and show him some of your flowcharts. He will help you with any errors that have come up or with anything that you don't understand.

ASSESSMENT TASKS: See your instructor. You will need pencil and paper, and you may use your books. You will be required to write both boxed and line by line flowcharts.

WHAT NEXT? You may proceed with UNIT #5 if you have done UNITS #3 and #4 already. If you haven't done 3 and 4, then go to UNIT #3. If you have done 3, but not 4, then do UNIT #4.

UNIT #3 (COMSC)

TITLE: Variables, Constants, Expressions and Assignment Statements

RATIONALE: Fortran is a procedure-oriented language developed specifically to handle algebraic expressions. The most basic elements of algebra and Fortran are variables and constants and the expressions constructed from them by connecting them with arithmetic operators. In order to program in Fortran you must have a thorough grasp of these concepts in both algebra and Fortran. Your grasp of the algebra part is assumed; the Fortran part is contained in this unit.

OBJECTIVE: When you complete this unit, you will be able to

(i) identify and construct integer and real constants and variables;

(ii) identify, construct, and evaluate Fortran expressions;

(iii) identify and construct assignment statements, and describe the results when they are executed.

PREREQUISITES: UNIT #1, if required, and a working knowledge of elementary algebra.

ACTIVITIES:

Part I. Fortran variables and constants.

1. The one most noticeable difference between algebra and arithmetic is the use in algebra of alphabetic letters to represent unknown values. As in the case of algebra, this is a particularly important characteristic of Fortran. Just as in algebra we solve or reduce expressions in which unknowns (variables) appear, so we will manipulate expressions in Fortran, even though at the time of writing the program the values of the variables are unknown.

2. Refer to UNIT #3, ACTIVITIES TABLE, Activity 1.

3. In Fortran, the mode or type of the number being used is designated through the initial letter in the variable name. You must be very careful to insure that the name you use for a variable correctly reflects the use that will be made of the variable. One of the most common errors made by new and old programmers alike is to use an integer name for a value that is not a whole number, or to use a real name for a value that is a whole number. If we attempt to place the value 3.4 in the location labeled IX, IX would contain only the value 3. The .4 would be dropped and lost.

SELF EVALUATION FOR PART I:

The answers to these exercises are given on the page following them.

1. Identify the following as variable names or constants

and as real or integer if they are valid. If they are invalid, state why they are invalid.

a. AKTION

b. X-RAY

c. INTEREST

d. 5.34

e. SUM3

f. 10,000

g. KING

h. 2AIJ

i. 5.78E6

j. 25

k. DISTANCE

l. I*J

m. 10,365.

n. RATE

o. C

p. K8J3

q. STOP

2. Take the number 256 and write it as an integer constant, a real constant, and a real constant in exponential form.

3. Take the number 25.6 and write it as an integer constant, a real constant, and a real constant in exponential form.

4. Write five different integer variable names.

5. Write five different real variable names.

Answers:

1. a. Valid real variable
   b. Invalid: special character -
   c. Invalid: more than six characters
   d. Valid real constant
   e. Valid real variable
   f. Invalid: imbedded comma
   g. Valid integer variable
   h. Invalid: does not begin with alphabetic character
   i. Valid real (exponential) constant
   j. Valid integer constant
   k. Invalid: to more than six characters
   l. Invalid: special character *
   m. Invalid: imbedded comma
   n. Valid real variable
   o. Valid real variable
   p. Valid integer variable
   q. Valid real variable

2. Integer, 256; real, 256. or 256.0; real exponential, 256.E0 or 256.0E0 or 2.56E2 and other variations.

3. Integer, 25 (the .6 is truncated); real, 25.6; real exponential, 25.6E0 or 2.56E1 or 256.E-1 and other variations.

4. Any combinations of up to and including six alphabetic and numeric characters, beginning with one of the letters I-N, is acceptable.

5. Any combination of up to and including six alphabetic and numeric characters, beginning with one of the letters A-H and O-Z, is acceptable.

Note: IBM 1130 Fortran allows a maximum of only five characters in variable names.

You must be sure that you can recognize the difference between integer and real constants and variables names; proceed with Part II when you feel that you are ready.

Part II. Fortran Expressions.

1. Refer to UNIT #3 ACTIVITIES TABLE, Activity 2.
2. There are several very important points in this section which need emphasizing. Reread the reference cited in the ACTIVITIES TABLE with special emphasis on
   a. the use of parentheses in expressions;
   b. the hierarchy of operations in an expression;
   c. valid types of exponentiation;
   d. the mode or type of the value of an expression;
   e. problems of accuracy and precision;
   f. and integer division.
3. The exponentiation operator requires more discussion. Either a real or an integer quantity may be raised to an integer power. For example, it is correct to use

```
    A**K
or
    J**K
```

When an integer exponent is used, the exponentiation is actually performed by successive multiplications. For example, A**4 is evaluated by the computer as A*A*A*A. Thus, A may be positive, negative, or zero. Furthermore, K may be negative or zero (if A≠0.0). If K has a value of -4, for example, then A**K is evaluated as

$$\frac{1.0}{A} * \frac{1.0}{A} * \frac{1.0}{A} * \frac{1.0}{A}$$

or

$$\frac{1.0}{A*A*A*A}$$

In summary, there are no restrictions on A or K for the case A**K, that is, for a real quantity raised to an integer power (except that A cannot be zero when K≤0).

For the case J**K, there is one restriction, however. Since by definition no fractional parts are available with integers, K cannot be negative. In other words, J**K cannot be evaluated since 1/J contains no fractional part. (Also J cannot be zero when K is zero.)

Real exponents may be used. For example,

A**E

is a valid expression. When real exponents are used, the expression is evaluated with logarithms;

antilog(e·logA)

or in Fortran,

EXP(E*ALOG(A))

Since the logarithm function is undefined for A≤0.0, A must always be greater than zero.

The expression

J**E

is allowed with some compilers. When it is allowed, J is converted to a real quantity before the expression is evaluated; the result is real, not integer.

The results and restrictions of the types of exponentiation are summarized in Table 3.1. Certain suggestions become apparent upon examination of the concepts in the table.

(i) Usually, if the power to which you are raising a number is a small integer, it may be better simply to multiply it out rather than to use the exponentiation operator. For example,

X**2 is better written X*X.

(ii) Since the use of an integer power is less restrictive, use an integer power whenever possible, unless the power is large, in which case the execution time can become excessive.

(iii) When the power is large, it is better to use a real exponent for shorter execution time, provided, of course, that the number being raised to the power is greater than zero.

Type of exponentiation	Restrictions	Evaluation procedure	Result
J**K	K≥0*	J*J*...*J	Integer
A**K	Generally none—most general case*	A*A*...*A	Real

J**E	Not allowed with some compilers; if allowed, J>0	XJ ← J
		EXP(E*ALOG(XJ))
		or
		antilog(e·logXJ)
			Real

A**E	A>0.0	EXP(E*ALOG(A))
		or
		antilog(e·logA)
			Real

* There are additional restrictions when J or A is zero. The following cases are undefined:

0**K, K≤0

0.0**K, K≤0

Table 3.1. Results and restrictions on various types of exponentiation. J and K represent integer quantities; A, E, and XJ represent real quantities.

4. The hierarchy of operations can be summarized as follows:

   (i) Parentheses, innermost first, from left to right.

   (ii) Functions from left to right.

   (iii) Exponentiation from left to right.

   (iv) Multiplication and division from left to right.

   (v) Addition and subtraction from left to right.

SELF EVALUATION FOR PART II:

Answers are on the page following the exercises.

1. Identify the following expressions as real or integer if they are valid. If they are invalid, state why they are invalid.

   a. J*2.0+K

   b. A**4+2.0*B-C

   c. 2.0(A+B)-C

   d. J**I/K**N

   e. A/-B+C

2. Construct Fortran expressions for the following.

   a. $\frac{a+b}{c+d}$

   c. $\frac{a+b}{c+\frac{d}{e}}$

   e. $\left[p\left(\frac{r}{s}\right)\right]^{t-1}$

   b. $\frac{a\cdot b}{c+10}$

   d. $\frac{a}{b}+\frac{c\cdot d}{e\cdot f\cdot g}$

Answers.

1. a. Invalid (generally) - mixed mode
   b. Valid, real
   c. Invalid - missing operator after 2.0
   d. Valid, integer
   e. Invalid - two operators (/ and -) together

2. a. (A+B)/(C+D)
   b. (A*B)/(C+10.0) or A*B/(C+10.0)
   c. (A+B)/(C+D/E)
   d. A/B+(C*D)/(E*F*G) or A/B+C*D/(E*F*G)
   e. (P*(R/S))**(T-1.0) or (P*R/S)**(T-1.0)

   If P*R/S is negative, an error will result since the exponent is real and the expression is evaluated by logarithms. This problem is overcome by writing the expression with an integer exponent.

   (P*R/S)**(IT-1)

Part III. Assignment Statements.

1. Refer to UNIT #3 ACTIVITIES TABLE, Activity 3.
2. In Fortran assignment statements, the "equals" symbol does not actually mean equality. Rather it should be thought of as a storage operator or a replacement operator. The value of the expression on the right of "=" is stored in the storage location identified on the left of "="; or saying it another way, the value on the left of "=" is replaced by the value of the expression on the right.

3. Notice that the value of an integer expression on the right can be stored as a real value on the left, and vice versa. For example, J=2.6 gives an integer value of 2 stored in J. (The fractional part is truncated.) A=2 gives a real value of 2.0 stored in A.

SELF EVALUATION FOR PART III.

Answers are on the page following the exercises.

1. Identify the following assignment statements as valid or invalid.

   a. J*K=I

   b. A=2.0*B

   c. -B=C/D+E

   d. X=SIN(Y)

2. Construct Fortran assignment statements for the following.

   a. $x = \cos(y) + x \cdot \sin(z)$

   b. $a = \left(- \frac{-x+y+27}{z^2} \right)^4$

   c. $r = (2.0+x^2)^{1/2}$

3. State the numeric value of J that will be transferred to memory by the following arithmetic assignment statements.

   a. J = 5*5/7

   b. J = 5/7*5

c. J = 2.0/3.0 + 2.0/3.0

d. J = 5*7/5

e. J = 7/5*5

4. State the numeric value of X that will be transferred to memory by the following arithmetic assignment statements.

a. X = 5*5/7	d. X = 5.0*4.0/2.0
b. X = 7/5*5	e. X = 4.0/2.0*5.0
d. X = 4*3**2	f. X = 5.0/3.0+3.0/3.0+5.0/3.0

5. Refer to UNIT #3 ACTIVITIES TABLE, Activity 4.

Answers.

1. a. Invalid - single name must appear on left
   b. Valid
   c. Invalid - -B incorrect
   d. Valid

2. a. X = COS(Y) + X*SIN(Z)
   b. A = (-((-X+Y+27.0)/Z**2))**4
      or A = (-(-X+Y+27.0)/Z**2)**4
   c. R = (2.0+X**2)**0.5 or R = SQRT(2.0+X**2)

3. a. 3
   b. 0
   c. 1
   d. 7
   e. 5

4. a. 3.0
   b. 5.0
   c. 36.0
   d. 10.0
   e. 10.0
   f. Approximately 4.33333 or 13/3

5. Refer to UNIT #3 ACTIVITIES TABLE, Activity 4.

ASSESSMENT TASK: Please see your instructor. You will be required to identify and construct correctly written constants, variable names, expressions, and assignment statements. You will also be required to evaluate Fortran expressions and to describe the results when Fortran assignment statement are executed.

WHAT NEXT? You may go ahead with UNIT #2 or with UNIT #4.

UNIT #4 (COMSC)

TITLE: STATEMENT NUMBERS AND UNCONDITIONAL BRANCHES

RATIONALE: One of the important characteristics of the modern computer is its ability to execute repeatedly a series of instructions automatically. This unit is the first of several that will help you learn to utilize this ability.

OBJECTIVES: When you complete this unit, you will be able to construct and identify statement numbers and unconditional branches that will utilize the statement numbers.

PREREQUISITES: UNIT #3 (COMSC).

ACTIVITIES:

1. Normally program steps are executed sequentially in the same order in which they appear. In Figure 4.1 the first statement executed would put 5.0 in the storage location named A.

```
      A = 5.0
      B = A + 1.0
      C = B + A * 3.0
```

Figure 4.1. A sample program segment, illustrating the normal order of execution from top to bottom.

The next one executed would put a 6.0 in location B. The last one would put 21.0 in location C. Normally this order of execution from top to bottom is desirable, since usually we do want the statements executed in the same order in which they are written. There are, however, important exceptions. Frequently we may want to repeat the execution of some statements or group of statements. Without the ability to repeat the execution of a series of program steps, the programs we write would be too long to be practical.

For example, suppose we want to compute the volume of 25 boxes. The first box has dimensions of 1 unit, 4 units, and 5 units. Each succeeding box has dimensions each 1 unit larger than the preceding box. For such a job we might write the program shown in Figure 4.2.

```
A = 1.0
B = 4.0
C = 5.0
VOL = A * B * C
A = 2.0
B = 5.0
C = 6.0
VOL = A * B * C
A = 3.0
B = 6.0
C = 7.0
VOL = A * B * C

     etc.
```

Figure 4.2. Sample program segment for calculating the volume of 25 boxes with dimensions incremented by one each time.

This series would continue until we had computed the volume of 25 boxes. How many statements would we have written? __________ Notice that there are 4 statements for the computation of VOL for each box.

If we take advantage of the iterative capability of a computer, we can write the program in a much shorter way, shown in Figure 4.3. The next to the last line is not a valid Fortran statement; it is a substitution for a Fortran statement that will be covered later.

```
      A = 1.0
      B = 4.0
      C = 5.0
  5   VOL = A * B * C
      A = A + 1.0
      B = B + 1.0
      C = C + 1.0
      if A is equal 26 stop
      GO TO 5
```

Figure 4.3. Shortened program segment for calculating the volume of 25 boxes with dimensions incremented by one each time

There are some things in this series that need explaining, but one thing should be clear. From a series of 100 statements, we have cut down to only 9 statements.

Take a look at the fourth statement in Figure 4.3. There is a number that appears in front of the statement. This number is referred to as a statement number. The sole purpose of a statement number is to identify the statement for later reference. We give it a unique number so that we can refer to that statement from other parts of the program. (Note that this is not a sequence number. Statement number 5 is not necessarily the fifth statement. If you have a hangup for numbers, you might name three children "Ten," "Two," and "Twenty." These are valid names and do not necessarily imply that you have twenty children, nor are they necessarily named in numerical order.)

The statement numbers in Figure 4.4 are perfectly valid.

```
   77 || A = B
  105 || A = B + 15.0 + A
    6 || B = B * A
  999 || B = B + B
```

Figure 4.4. Examples of statements with statement numbers.

Statement numbers must be positive integers from 1 to 99999, though the maximum number allowed may be less for some computer systems.

2. Refer to UNIT #4 ACTIVITIES TABLE, Activity 1. An example of a Fortran coding form, which is available in the bookstore, is shown on page 4.5. You are encouraged to use coding forms for writing programs to be punched on computer cards, which you'll take up in UNIT #5, since the format of the coding form is the same as the format of the Fortran computer card.

Notice once more that statement numbers are placed anywhere in columns 1-5 and Fortran statements are placed

OKLAHOMA STATE UNIVERSITY
FORTRAN CODING FORM

PROBLEM TITLE	COURSE NO. SECTION NO.	PROBLEM NUMBER	PROGRAMMERS NAME	PAGE ____ OF ____

STATEMENT NO. (C)	CONT.	FORTRAN STATEMENT	IDENTIFICATION ONLY
1 2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80

anywhere in columns 6-72.

3. The last statement in Figure 4.3 is called an unconditional branch. Up to this point, each statement has been executed sequentially. This last statement, however, changes the order of execution. It tells the computer to execute next the statement identified by statement number "5" and continue sequentially from that point. The next statement executed after statement number 5 is the statement

```
        A = A + 1.0
```

In an unconditional GO TO statement, the GO TO is always followed by an integer constant which is the unique statement number of the statement to which transfer is to take place.

4. Refer to UNIT #4 ACTIVITIES TABLE, Activity 2.

SELF EVALUATION:

1. Which of the following are legal statement numbers?

a.	13	d.	123456
b.	9B	e.	2/3
c.	3456	f.	2-1

2. Write a statement that will cause a branch from the last statement of the following routine to the second statement. Add statement numbers if necessary.

```
3 A = 1.0
  B = A + 3.0
  A = A + A
  H = 0.5 * A ** 2
```

3. Write a set of Fortran statements (a program segment) that will count by fives. Set the counter to zero; use integers. Then add five to the counter, and go back to the statement that adds five, etc. (Refer to UNIT #4 ACTIVITIES TABLE, Activity 3, for help.)

Answers:

1. Legal statement numbers are a and c. The number in d has more than 5 digits. The ones contained in b, e, and f contain characters that are not numerals.

2.

```
   3 || A = 1.0
  50 || B = A + 3.0
     || A = A + A
     || H = 0.5 * A ** 2
     || GO TO 50
```

3.

```
     || N = 0
  10 || N = N + 5
     || GO TO 10
```

ASSESSMENT TASK: Please see your instructor. You will be required to identify valid statement numbers and to construct one or more short program segments using unconditional GO TO statements and the material in UNIT #3.

WHAT NEXT? If you have completed UNIT #2, you are ready to proceed with UNIT #5. If you have not done UNIT #2, then do it, after which you may go to UNIT #5.

UNIT #5 (COMSC)

TITLE: Preparing a Job for Running on the Computer

RATIONALE: In the next unit, UNIT #6, you will be writing your first Fortran program. Before you can run your program on the computer, you must know about keypunch machines and punched cards; you must know how to run jobs on the computer; and you need to know how to document a program.

The first part of this unit deals with punched cards. Since the punched card is still one of the prime means of input to the computer, you need to know how to punch and interpret computer cards. In order to punch cards, you make use of a card punch or keypunch machine, which you will find out how to use.

The second part of the unit describes documentation of programs and why it is important.

The third part tells you how to run a job on the computer. If you were required to do UNIT #1, then most of this procedure will be a review.

OBJECTIVE: When you have completed this unit, you will be able to demonstrate that you can punch computer cards and interpret them, document and punch a Fortran program that is given to you, and run the program on the computer.

PREREQUISITES: UNITS #2 and #4.

ACTIVITIES:

PART I. Punching and interpreting computer cards.

A. "Do not fold, mutilate, or spindle." "The punched hole will add itself to something else, subtract itself from something else, multiply itself by something else, divide itself by something else, list itself, reproduce itself, classify itself, select itself, print itself on a card, produce an automatic balance forward, file itself, post itself, cause a total to be printed, compare itself to something else, reproduce and print itself on the end of a card, cause a form to feed to a predetermined position, or to be rejected automatically, or to space the form from one position to another."[1]

How can something that is nothing do all of this? Simply by punching rectangular shaped holes into an IBM card (punched card) as codes for letters and numbers, we can use the card as input to data processing equipment which in turn performs these functions.

A standard IBM punched card measures 7 3/8 by 3 1/4 inches, is 0.007 inches thick, and is made of

[1]Punched Cards, Donald A. C. McGill, McGraw-Hill Book Company, page 30.

special paper which withstands the effects of handling by man and machine. A larger card punched with round holes is used on Sperry Rand Corporation (UNIVAC) equipment; however we will consider only the IBM card, since it is the most common, being used on most modern equipment, even by UNIVAC since 1966.

An example of an IBM computer card is shown in Figure 5.1. Usually the upper corner is cut at a 60° angle with the long edge of the card, although cards will be found with uncut corners. The corner cut has no effect on the operation of the computer and is only to enable the operator to make a quick visual check that all the cards are facing the same way and are right-side up. Mixed corner cuts and mixed colors can be used if it is important to be able to distinguish different card types visually.

The card is divided into 80 columns numbered 1 thru 80, from left to right, and into 12 rows numbered 12, 11, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, from the top of the card to the bottom.

B. The Hollerith code used for punched cards.

The code used in punching data into these cards was patented by Herman Hollerith, a statistician for the Bureau of The Census. This code enabled the Census of 1890 to be processed by automated equipment.

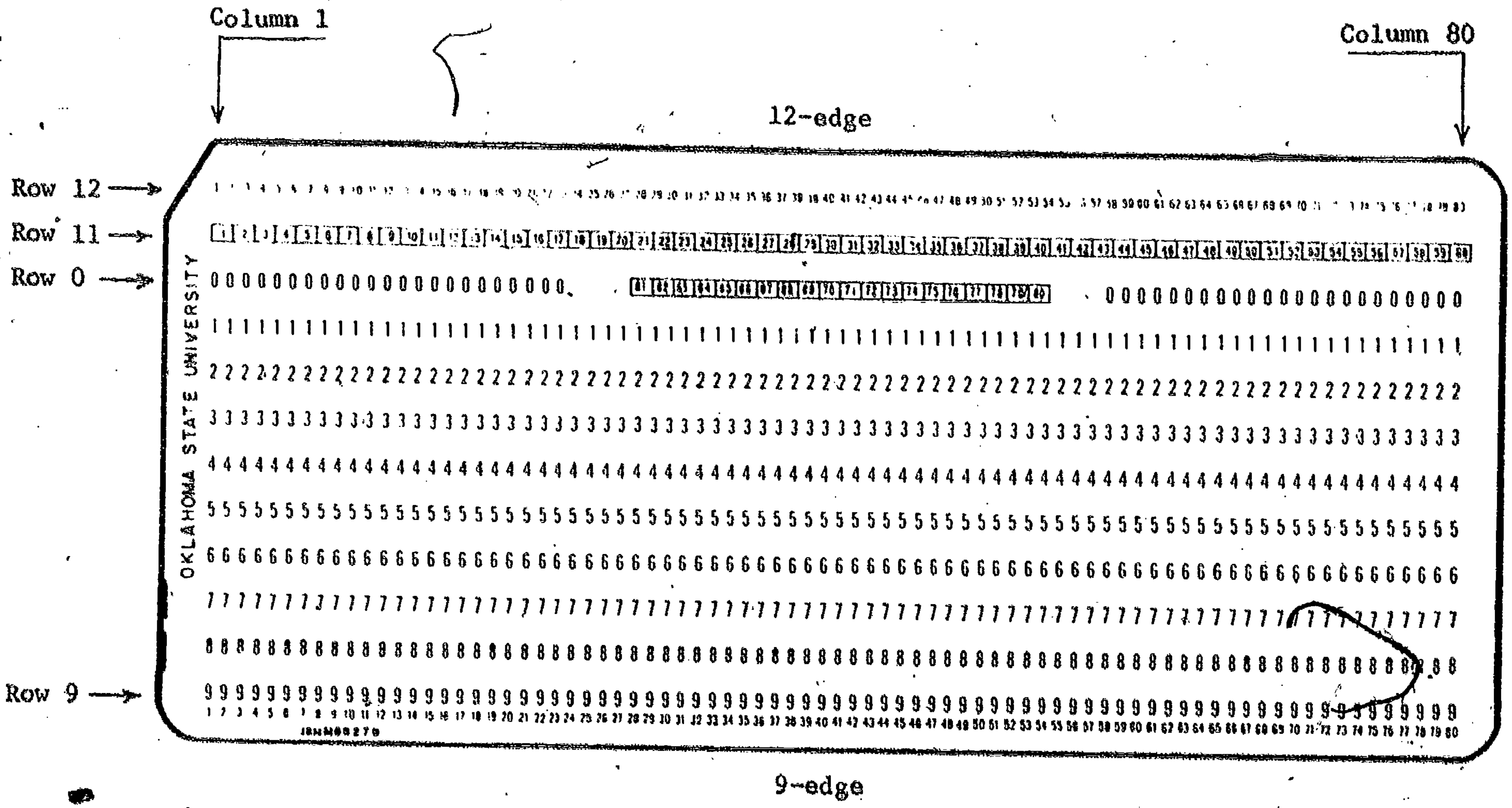


Figure 5.1. An example of an IBM computer card. The 80 columns are numbered by small numbers at the top and at the bottom. The card also contains 12 rows, 10 of which are numbered 0-9 and two of which are not numbered on the card. Row 12 is at the top of the card, and row 11 is between row 12 and row 0. The top edge of the card is called the 12-edge, while the bottom edge is called the 9-edge.

Dr. Hollerith in 1903 left the Bureau of the Census to found the Computing Tabulating Recording Company, which later was the nucleus of the International Business Machine Corporation.

An example of a punched computer card is shown in Figure 5.2. Careful examination of the card will reveal that the Hollerith Code uses a single punch in rows 0 through 9 to represent the digits 0 through 9, respectively. A punch in these rows is called a digit punch, or a numeric punch. Alphabetic characters or letters are represented by two punches in the same column. One of these punches is in one of the rows 12, 11, or 0 and is called a zone punch, and the other is a digit punch in one of the rows 1 thru 9. Note that row 0 is called a zone punch for alphabetic characters and is called a digit punch for numbers. The codes for the alphabetic and numeric characters are depicted in Table 5.1.

Note that the zone 0 digit 1 punch is not used for the letter S as you might expect but it is used as the special character / or slash. S is zone 0 digit 2.

There are also codes for special characters and/or punctuation marks. These consist of two digit punches, a single zone punch, or a zone punch and one or two digit punches. The special characters that are used in Fortran IV are given in Table 5.2.

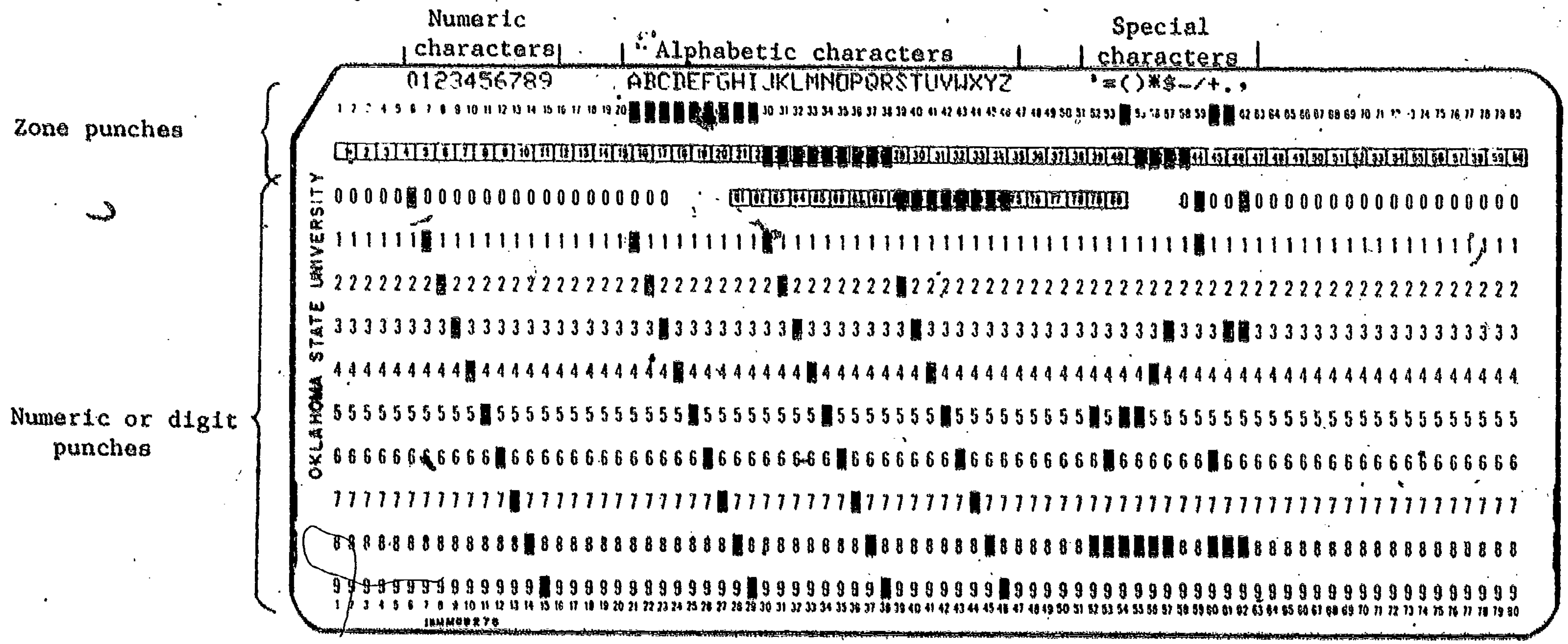


Figure 5.2. A punched computer card, showing punches for numeric, alphabetic, and some special characters used in Fortran.

CHARACTER	ZONE PUNCH	DIGIT PUNCH	CHARACTER	ZONE PUNCH	DIGIT PUNCH
0		0			
1		1	J	11	1
2		2	K	11	2
3		3	L	11	3
4		4	M	11	4
5		5	N	11	5
6		6	O	11	6
7		7	P	11	7
8		8	Q	11	8
9		9	R	11	9
			S	0	2
A	12	1	T	0	3
B	12	2	U	0	4
C	12	3	V	0	5
D	12	4	W	0	6
E	12	5	X	0	7
F	12	6	Y	0	8
G	12	7	Z	0	9
H	12	8			
I	12	9			

Table 5.1
Punched card codes

CHARACTER		ZONE PUNCH	DIGIT PUNCH(ES)
.	Period or Decimal Point	12	3 - 8
(	Left Parenthesis	12	5 - 8
+	Plus	12	6 - 8
-	Minus or Dash	11	
$	Dollar	11	3 - 8
*	Asterisk	11	4 - 8
)	Right Parenthesis	11	5 - 8
/	Virgule or Slash	0	1
,	Comma	0	3 - 8
=	Equal		6 - 8

Table 5.2
Punched card codes

The special characters in Table 5.3 are used in other programming languages.

CHARACTER		ZONE PUNCH	DIGIT PUNCHES
&	Ampersand	12	
¢	Cent	12	2 - 8
<	Less Than	12	4 - 8
|	Vertical Line	12	7 - 8
!	Exclamation	11	2 - 8
;	Semicolon	11	6 - 8
	Not	11	7 - 8
%	Percent	0	4 - 8
_	Break or Underscore	0	5 - 8
>	Greater Than	0	6 - 8
?	Question Mark	0	7 - 8
:	Colon		2 - 8
#	Number or Pound		3 - 8
@	Commercial At Sign		4 - 8
'	Single Quote or Apostrophe		5 - 8
"	Double Quote		7 - 8

Table 5.3
Punched card codes

C. A pause for self evaluation.

Take a short "time out" and see how you're doing so far.

1. Interpret the punched card shown in Figure 5.3.

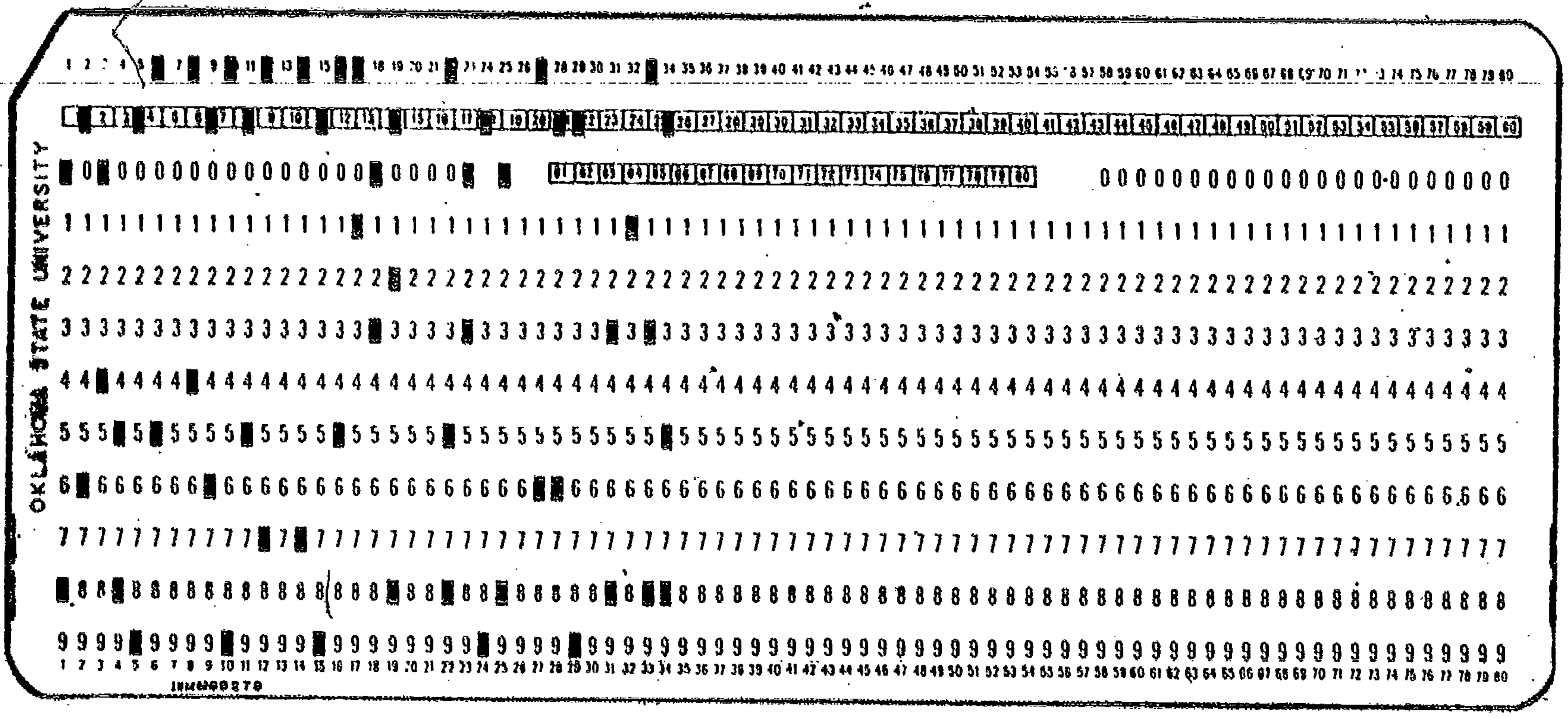


Figure 5.3. Interpret the punched card.

2. Using the blank card shown in Figure 5.4 and a pencil, mark the punched card codes for the letters of your name and the characters of your Social Security number (including the dashes).

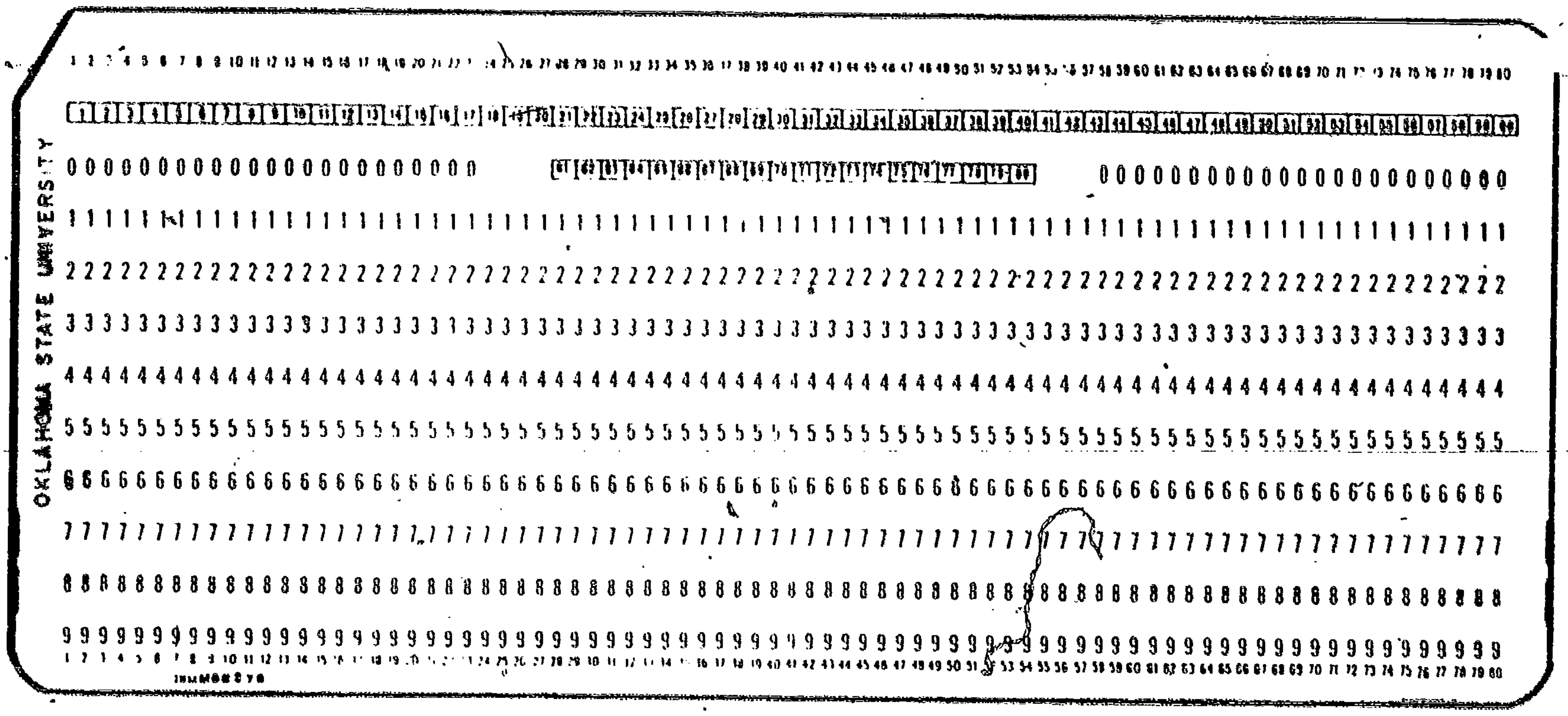


Figure 5.4
Blank card for marking your name and Social Security number

If you feel that you're ready, then go ahead. If not, then back up and review the material.

D. The Fortran statement card.

There is a card printed especially for Fortran shown in Figure 5.5. (Of course, the computer doesn't know one card from another; the various types and colors of cards are of consequence only to the user for his convenience.) The card has various blocks of columns labeled to aid you in punching Fortran programs properly. Column 1 must have a "C" punched in it for comments. Statement numbers are placed anywhere in columns 1-5. Fortran statements

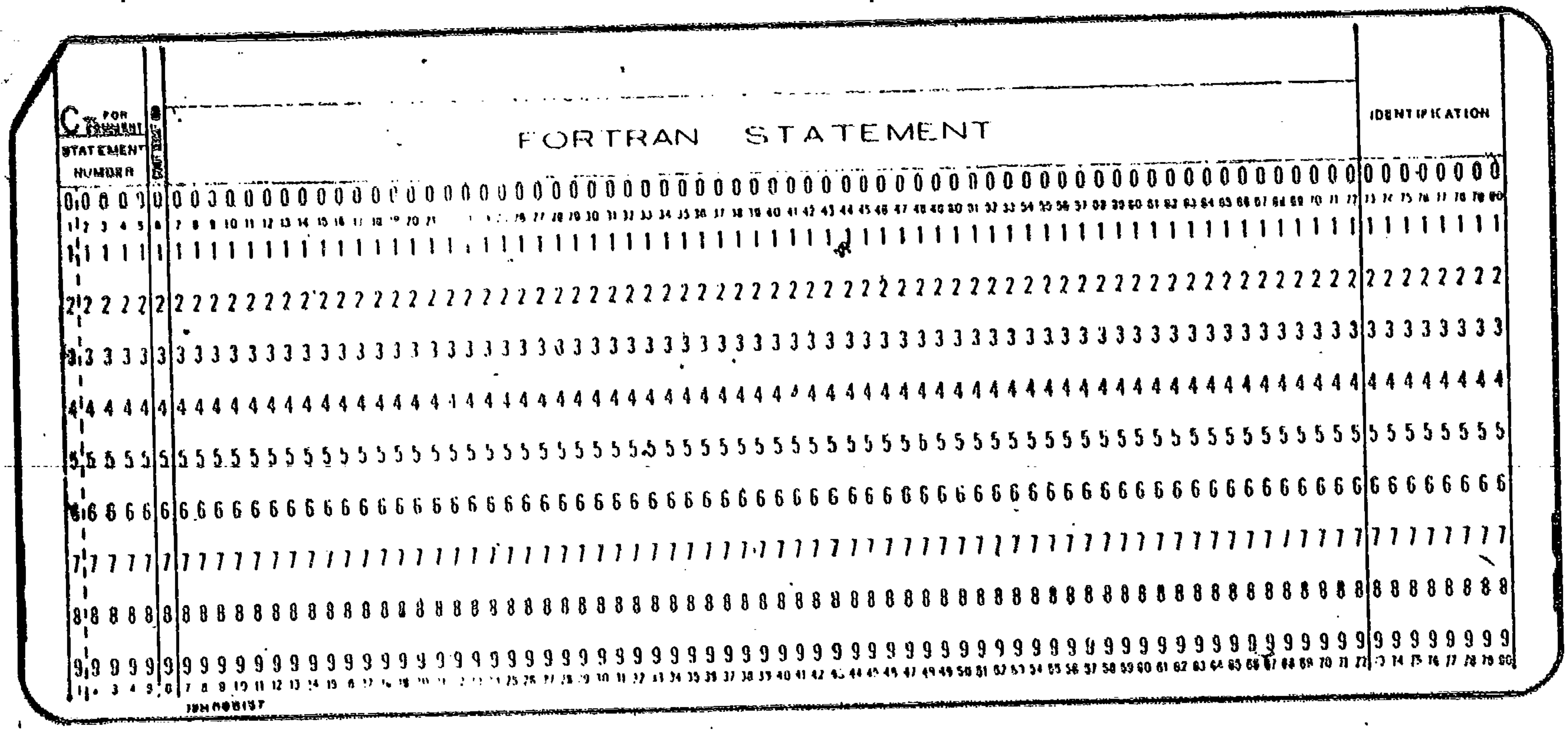


Figure 5.5. A Fortran statement card

are placed anywhere in columns 7-72. Columns 73-80 may be used in any way for purposes of identification, such as program name, programmer's name or initials, or sequence numbers. Column 6 is used for indicating that a card is a continuation of the previous card.

You will get some practice using Fortran cards later, but now you should be ready to take on the keypunch machine.

E. Punching computer cards.

The following instructions will introduce you to the IBM model 29 keypunch machine. Go through the instructions carefully until you can load cards, punch something into some cards (for example, your name, the date, your social security number, etc.) duplicate a card, and clear the machine.

Notice: Sometimes pressing certain keys will cause the machine to "lock up." When this happens, press the REL key to release the machine.

1. Keypunches are located in MS 04, are a light grey color, and are identified by a 29 on the name plate on the upper right-hand front of the machine. Cards are located in bins placed among the machines.

2. Turn on the main switch located in front of your right knee as you sit at the keyboard.

3. Behind the name plate on the upper right is the card hopper. The cards are held in place by a spring-loaded plate. Place cards neatly and securely into the hopper in the upright position.

4. Three buttons or keys on the right side of the keyboard, REL (release), FEED, and REG (register), and one switch on the left center of the panel of switches just above the keyboard, AUTO FEED, control the feeding of cards into the card track. There are two ways to use these controls:

   a. AUTO FEED switch "OFF." Depress the REL, FEED, and REG keys in that order. This procedure allows the passage of one card at a time through the machine and is useful for beginning users of the keypunch.

b. AUTO FEED switch "ON." Depress REL key twice just after you have loaded the card hopper, once thereafter. Use only the REL key for automatic operation.

5. When a card is in place and is ready to be punched, characters may be placed in the card by use of the keyboard, which is similar to a standard typewriter keyboard. The column on the card currently being punched is indicated by a pointer and a scale on a drum directly in front of the operator, upper center behind the window. In order to punch alphabetic characters and other characters on the lower portion of the keys, simply depress the proper key. In order to punch numbers and other characters on the upper portion of the keys, depress and hold the NUM (numeric shift) key, lower left of keyboard, and then depress the proper key. .

6. When a card is being punched, it is in place at the punch station. When that card is released by pressing the release button (REL), it moves from the punch station to the read or duplicate station to the left of the punch station. When the DUP (duplicate) key, located in the middle

of the top row, is depressed and held, the information punched in the card in the read position is transferred to the card in the punch position. An entire card may be duplicated; or part of it may be duplicated by releasing the DUP button, allowing corrections to be made.

A card also may be inserted by hand into the card track at the read station. There are two slots in the middle of the card track through which a card may be pushed until it is against the stop of the read station. Both the card to be duplicated and the blank card to be punched may be moved into position simultaneously with the REG button. The easiest way to do this is to have the card track empty, insert the card to be duplicated, depress the FEED button to feed a blank card into the punch station, depress the REG button, and then duplicate.

7. At the completion of the job, CLEAR ALL CARDS FROM THE MACHINE AND THE DESK TOP and turn the main switch to "OFF." (The card track may be cleared automatically by flipping the CLEAR switch located at the extreme right on the panel of switches above the keyboard to "ON.")

Using these instructions, practice punching cards, duplicating cards, and correcting errors.

SELF EVALUATION FOR PART I.

1. Punch your name and social security number into a card. Begin your name in column 5 of the card and begin your social security number in column 50.
2. Duplicate exactly the card you just punched.
3. Punch OKLAHONA STATE.UNIVERSITY in columns 11-35. Then correct the error punched in column 17 by duplicating the part that is correct and repunching column 17.
4. Find some Fortran statement cards in the bins beside some of the keypunches. Punch the Fortran statement

```
4 XTAP=2.3*AVG/(21.0+B)
```

in the proper columns of the card. The statement number 4 goes anywhere in columns 1-5; the Fortran statement goes anywhere in columns 7-72.

5. Punch the Fortran program shown below exactly as it appears, putting one line per card and punching the characters in the columns as shown. Do not expect to understand the program; just punch it for now. The two lines of numbers above the program are the column numbers. For example, $\begin{smallmatrix}1\\4\end{smallmatrix}$ indicates column 14 of the punched card; an "R" is punched in this column of the first card. The "O's" in the program are slashed (Ø) in order to distinguish them from zeros. Be sure to punch zero when you mean zero and "O" when you mean "O."

```
         11111111112222222222333333333344444444445555555555666666666677777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890
CØMPUTER PRØGRAM WRITTEN IN FØRTRAN IV
      TRY1=2.1
      TRY2=2.6
      TRY3=0.6
      ANS=3.0*TRY1+TRY2/(TRY1-TRY3)
      WRITE(6,4)ANS
    4 FØRMAT(1H0,F10.1)
      STØP
      END
```

Check the cards carefully for errors and make corrections as necessary. Save the cards; you'll be using them soon.

PART II. Documentation of a program.

The purpose of documentation is summed up in the following limeric:

Johnny found a program
One very useless day.
Exactly what that program did
It simply didn't say!

In order for a program to be of use to any user, including the programmer himself, it must be accompanied by a fairly detailed description. Some of the short programs that you will write for this course will seem not to require any documentation; but you need to develop good habits early, so documentation of all programs will be required.

F. What should be included in documentation?

Program documentation consists of two parts:

(i) program comments

(ii) program description

Comments in a program should convey to the reader the essential facts of the program and should include at the beginning of the program

(a) the student's name, problem title, and date submitted;

(b) a description of the problem;

(c) a description of the program;

(d) special or exceptional conditions;

(e) definitions and formats of input variables;

(f) definitions and formats of output variables;

(g) definitions of other key variables;

(h) error messages;

(i) key comments interspersed throughout the program.

The accompanying program description should include, when appropriate,

(a) the problem title, the student's name, the date submitted, the unit number;

(b) a brief description of the problem and the solution methods employed;

(c) a description of the limitations of the program;

(d) a description of major variables and of all dimensional variables;

(e) a description of possible errors and associated error messages;

(f) a complete program listing including control cards, input and output;

(g) operating instructions;

(h) a flowchart at a level of detail necessary to convey the essential information about the program;

(i) a description of key points in the flow diagram.

The sample program on pages 5.20 to 5.22 illustrates many of the points described. Notice that one line in the program listing corresponds to one punched card. The C which appears to the left of the first several lines is punched in column 1 of a Fortran statement card. Such cards are called COMMENT cards. COMMENT cards are printed on the program output listing, but are ignored by the compiler. Fortran statements start in any column after column 7 of a statement card and may be punched from columns 7 to 72. Statement numbers are punched in columns 1 to 5; column 6, the continuation indicator column, is punched if the statement is too long for the previous card and has to be continued on the current card.

Sequential line numbers appear on the far left of the printed listing. Comments do not have a line number. For example, line number 0006 is associated with the statement

```
INDX = 1
```

which places the value 1 in the location assigned to variable called INDX.

Note that the comments at the beginning consist of the author's name, the date, a brief description of the program, input card description, output card description, special operating instructions, and a description of the principal variables.

The comment cards interspersed throughout the program gives an indication of what the program is supposed to do. This program generates the glossary listing that appears in Appendix II. The comment cards at the beginning of the program would have been easier to read if they had been set up like the terms and the definitions of the terms in the glossary. Comment cards with a series of special characters such as * and blank comment cards may be used to advantage. For example, one might have punched the comment cards in the following format.

```
C
C     INPUT
C        CARDS ARE PUNCHED AS FOLLOWS:
C          COLUMNS           CONTENTS
C           1-11             KEY
C          13-14             SEQUENCE NUMBERS
C          16-80             DEFINITION LINE
C        - ONE DEFINITION IS LIMITED TO 50 CARDS.
C     *****  *****  *****  *****
```

Comments may be punched anywhere in columns 2-72. (Actually, columns 73-80 may also be used, but generally

```
C  AUTHOR:  CHARLES ELLIS
C  CLASS:  COMSC NEW STUDENTS
C  DATE:  18 AUG 1971
C  BRIEF DESCRIPTION:  THIS PROGRAM USES CARD INPUT CONSISTING
C  OF LINES IN A DEFINITION OF A WORD.  THE CARDS ARE KEYED AND
C  SEQUENCED.  THE CARDS ARE CHECKED FOR KEY AND SEQUENCE AND STORED
C  IN AN ARRAY.  WHEN THE KEY CHANGES THE WORD DEFINITION IS PRINTED
C  AND THE PROCESS IS REPEATED UNTIL ALL GLOSSARY ENTRIES HAVE BEEN
C  WRITTEN.  OUT OF SEQUENCE ENTRIES ARE REPORTED BY KEY ON AN ERROR
C  LISTING AND FLUSHED FROM THE INPUT, NOT APPEARING IN THE GLOSSARY.
C  INPUT:  CARDS OF THE FORM;
C  COL 1-11 KEY
C  COL 13-14 SEQUENCE NUMBER
C  COL 16-80 DEFINITION LINE
C  ONE DEFINITION IS LIMITED TO 50 CARDS
C  OUTPUT:  TWO 8.5*11 INCH LISTINGS.  THE FIRST IS THE GLOSSARY,
C  EACH PAGE TITLED AND NUMBERED.  THE SECOND IS AN ERROR REPORT.
C  OPERATING INSTRUCTIONS:  STANDARD BATCH FORTRAN G LEVEL DECK
C  SET-UP AND RUN PROCEDURES ARE USED WITH THE ADDITION OF THE CARD;
C  //GO.FT04F001 DD SYSOUT=A FOR THE ERROR LISTING.
C  PRINCIPAL VARIABLES:
C  KEY  11 CHARACTER ARRAY FOR THE CURRENT KEY
C  KEYSV  11 CHARACTER ARRAY FOR THE PREVIOUS KEY
C  LINE  65 CHARACTER FOR THE CURRENT DEFINITION LINE.
C  LINES  A 50 BY 65 ARRAY FOR THE COMPLETE DEFINITION
C  LNCNT  COUNTER FOR THE NUMBER OF LINES PRINTED PER PAGE
C  IBLK  A BLANK CHARACTER CONSTANT
C  LST  FLAG FOR END OF DATA
C  INDX  THE CURRENT SEQUENCE NUMBER
C  INDXSV  SEQUENCE NUMBER OF THE PREVIOUS LINE
C  DEFINE ARRAYS
      DIMENSION KEY(11),KEYSV(11),LINE(65)
      COMMON LINES(50,65)
C  INITIALIZE CONSTANTS AND COUNTERS
      DATA LNCNT/55/,IBLK/1H /,LST/0/
C  INITIALIZATION READ
      READ (5,1) KEY,INDX,(LINE(J),J=1,65)
    1 FORMAT (11A1,1X,I2,1X,65A1)
      INDX=1
      GO TO 11
C  MAIN DATA READ
    3 READ (5,1,END=99) KEY,INDX,(LINE(J),J=1,65)
C  CHECK FOR BLANK CARD, YES, REJECT IT
      IF (KEY(1).EQ.IBLK) GO TO 3
C  CHECK FOR END OF DEFINITION
      DO 5 I=1,11
      IF (KEYSV(I).NE.KEY(I)) GO TO 6
    5 CONTINUE
C  SEQUENCE CHECK
      IF ((INDXSV+1-INDX).NE.0.OR.INDX.EQ.0) GO TO 4
C  STORE CURRENT LINE IN DEFINITION ARRAY
   11 DO 7 I=1,65
    7 LINES(INDX,I)=LINE(I)
C  STORE CURRENT KEY AND INDX IN SAVE AREAS
      DO 12 I=1,11
   12 KEYSV(I)=KEY(I)
      INDXSV=INDX
      GO TO 3
```

```
C  CHECK FOR END OF PAGE IF SO WRITE NEW HEADING
    6 CALL PAGE(INDXSV,LNCNT,ISTRT)
C  WRITE DEFINITION ON LISTING
      DO 8 I=ISTRT,INDXSV
      WRITE (6,9) (LINES(I,J),J=1,65)
    9 FORMAT (8X,65A1)
    8 CONTINUE
C  CHECK FOR END OF DATA
      IF (LST.EQ.1) GO TO 100
C  KEEP LINE COUNT STRAIGHT, DON'T ALLOW PAGE OVERFLOW
      IF ((LNCNT+INDXSV+2).GT.55) GO TO 18
C  SPACE GLOSSARY LISTING SO ITS PRETTY
      WRITE (6,10)
   10 FORMAT (1H0)
C  INCREASE LINE COUNT
      LNCNT=LNCNT+INDXSV+2
      GO TO 17
   18 LNCNT=55
C  SEQUENCE CHECK
   17 IF (INDX.NE.1) GO TO 4
      GO TO 11
C  ERROR, REPORT IT
    4 WRITE (4,13) KEY
   13 FORMAT (1H ,'THE GLOSSARY ENTRY WITH KEY ',11A1,' IS OUT OF SEQUEN
     1CE')
C  FLUSH BAD DEFINITION
      DO 14 I=1,11
   14 KEYSV(I)=KEY(I)
   16 READ (5,1,END=100) KEY,INDX,(LINE(J),J=1,65)
      IF (KEY(1).EQ.IBLK) GO TO 16
      DO 15 I=1,11
      IF (KEYSV(I).NE.KEY(I)) GO TO 17
   15 CONTINUE
      GO TO 16
C  SET END OF DATA FLAG
   99 LST=1
      GO TO 6
  100 STOP
      END
```

5.22

```
      SUBROUTINE PAGE(INDX,LNCNT,ISTRT)
C     THIS SUBROUTINE WRITES PAGE HEADINGS AND KEEPS LINE COUNT STRAIGHT
      COMMON LINES(50,65)
      DATA IPAGE/1/
      ISTRT=1
C     IS THERE ROOM FOR ALL OF THE DEFINITION ON THIS PAGE
      IF((INDX+LNCNT)  .LE.55) RETURN
C     YES, RETURN AND PRINT IT
C     NO,PRINT AS MUCH AS YOU CAN
      LNPOSS=55-LNCNT
      IF (LNPOSS.LE.2) GO TO 1
      DO 2 I=1,LNPOSS
      WRITE (6,3) (LINES(I,J),J=1,65)
    3 FORMAT (8X,65A1)
    2 CONTINUE
C     SET STARING POINT FOR REST OF DEFINITION
      ISTRT=LNPOSS+1
C     WRITE NEW HEADINGS
    1 WRITE (6,4) IPAGE
    4 FORMAT (1H1,33X,'COMSC GLCSSARY'/,37X,'PAGE ',I3//)
C     GET LINE COUNT STRAIGHT
      LNCNT=0
      IF(LNPOSS.GT.2) LNCNT=-LNPOSS
C     UPDATE PAGE NUMBER
      IPAGE=IPAGE+1
C     GO BACK
      RETURN
      END
```

it is best to leave these columns for other uses.) With this in mind, you may want to make your comments stand out on the output listing. All kinds of variations may be used to accomplish this, limited only by your imagination and creativity. (You may even wish to put graphic illustrations in your program with comment cards.) Two examples are shown below.

```
COMMENT     *****     *****     *****
C                                   *
C                                   *
C         PLACE YOUR MESSAGE        *
C         FOR POSTERITY HERE.       *
C                                   *
C                                   *
C *****     *****     *****     *****
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C**************************************************************************C
C*                                                                        *C
C*  THE OUTPUT FORMAT FOR THE CORRECT ANSWERS IS CHANGED FOR EXACTLY      *C
C*  TWENTY QUESTIONS PER TEST.                                            *C
C*                                                                        *C
C**************************************************************************C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

SELF EVALUATION FOR PART II.

1. Describe a method of punching the first few comment cards of the program on page 5.20 so that each category stands out on the page.
2. Write a set of comment cards giving your name, this course number, the date, and the objective of this unit.
3. Punch the comment cards that you wrote in 2. Place them

with the Fortran program that you punched for Part I. Save all these cards. You are gradually producing a job that you can run on the computer.

PART III. Running the Job on the Computer.

If you have correctly performed the activities and self evaluations in Parts I and II of this unit, then you have a documented Fortran program ready to be run on a computer. Refer to the ACTIVITIES TABLE for UNIT #5 for the procedure for doing this.

ASSESSMENT TASK:

Actually, you have already completed the assessment task by running your program on the computer. The documentation must be correct, the computer output must be correct, and you must have an 80/80 listing of your program. Take the program deck (less control cards), the printer output, and the listing to your instructor for his approval.

WHAT NEXT? You're nearly ready now to write your first program, but first you must learn how to read and write -- with the computer, that is. UNIT #6 tells you about that.

UNIT #6 (COMSC)

TITLE: INPUT AND OUTPUT

RATIONALE: In order for a computer to perform useful work on data, it is necessary to "read" those data into the computer. Similarly, in order to obtain the results, it is necessary for the computer to "write" those results on some form of output medium. In this unit, you will discover how to construct rather simple input/output commands in Fortran.

OBJECTIVES: When you finish this unit, you will be able to construct input/output Fortran statements to

(i) read data from data cards,

(ii) write results on the output printer,

(iii) carry out (i) and (ii) by means of FORMAT statements.

You will also demonstrate your ability to construct a Fortran program with documentation and run it on a computer.

PREREQUISITES: UNIT #5

ACTIVITIES: The term input/output will be used throughout this unit and will often be abbreviated I/O. The term is applied to any and all instructions and processes related to putting information into a computer and getting information out of a computer.

1. Introduction

In this unit you will learn to construct Fortran READ, WRITE, and FORMAT statements. The ideas covered are implied by the following questions.

What to do?	(READ, WRITE)
Where to do it?	(on an I/O device)
How to do it?	(by an appropriate FORMAT)

For a given computer configuration, it may be possible to obtain input from several different media, such as cards, magnetic tape, magnetic disk, a typewriter, another computer, paper tape, etc. Similarly, it may be possible to place the results on several different media, such as paper, cards, paper tape, magnetic tape, magnetic disk, typewriter, another computer, etc. For the present we will make use of only one type of input medium and one type of output medium, namely

punched cards for input

and

the printed page for output.

Figure 6.1 illustrates a computer with a card reader for input and a printer for output.

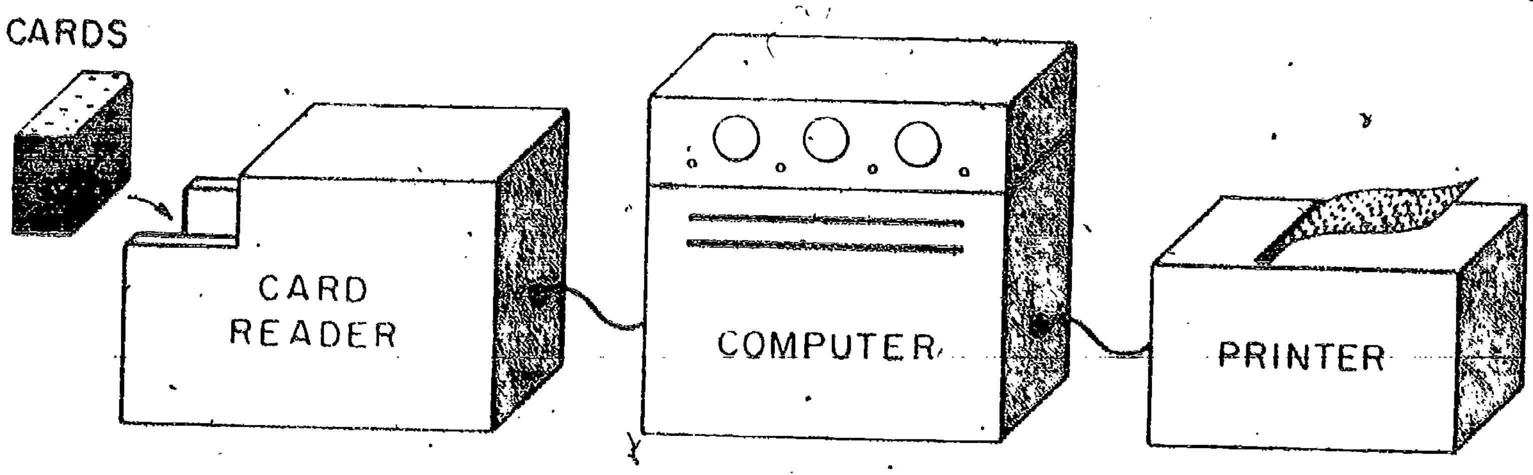


Figure 6.1. Schematic representation of a computer with its I/O units.

2. READ and WRITE statements.

To read a data card, Fortran statements of the form

READ(u,f) list

are used. To write a line on the line printer, Fortran statements of the form

WRITE(u,f)

and

WRITE(u,f) list

are used. The parameter u designates an I/O unit number or an integer variable name which takes on the value of the I/O unit number. The parameter f is the statement number of the corresponding FORMAT statement, and list refers to a variable name or to several variable names separated by commas. Each of these parameters will be discussed in detail in the following paragraphs.

The I/O unit number u is determined by the computer system being used. Table 6.1 identifies the unit numbers that you will need for running programs.

u	WATFIV FORTRAN	360 FORTRAN IV	1130 FORTRAN IV
READ (card reader)	5	5	2
WRITE (line printer)	6	6	1

Table 6.1. Fortran I/O Unit Numbers

Examples of WATFIV Fortran READ and WRITE statements are shown in Figure 6.2.

```
      READ(5,123) A,B,C
      WRITE(5,321) A,B,C,D,INO
      WRITE(6,1971)
1971  FORMAT(you will find out what goes here in the pages ahead)
 123  FORMAT(you will find out what goes here in the pages ahead)
 321  FORMAT(you will find out what goes here in the pages ahead)
```

Figure 6.2. Examples of WATFIV READ and WRITE statements.

Examples of 1130 Fortran READ and WRITE statements are shown in Figure 6.3.

```
      READ(2,444) I,A,Y
      WRITE(1,13) A,B
444   FORMAT(you will find out what goes here in the pages ahead)
 13   FORMAT(you will find out what goes here in the pages ahead)
```

Figure 6.3. Examples of 1130 Fortran READ and WRITE statements.

By using the variable I/O unit number in the READ and WRITE statements, conversion from one computer system to another is effected with minimal effort. In Figure 6.4 is shown the program segment taken from Figure 6.3 in a form that can be used on either WATFIV or 1130 Fortran simply by changing the first two assignment statements.

```
C  1130 FORTRAN                          C  WATFIV
      IN=2                                  IN=5
      IOUT=1                  <---->        IOUT=6
      READ(IN,444)I,A,Y                     READ(IN,444)I,A,Y
      WRITE(IOUT,13)A,B                     WRITE(IOUT,13)A,B
```

Figure 6.4. Use of variable I/O unit numbers to facilitate changing from one computer system to another.

An even better way is shown in Figure 6.5, using the DATA initialization statement, a nonexecutable statement that initializes variables at the time the program is compiled. The DATA statement generally reduces execution and compile time and also conserves storage in the computer, since machine instructions for IN=2 and IOUT=1 do not have to be set up, stored, and executed. Use the DATA statement for initializing constants in a program whenever possible.

```
C  1130 FORTRAN                          C  WATFIV
   DATA IN,IOUT/2,1/        }  <-->  {      DATA IN,IOUT/5,6/
   READ(IN,444)I,A,Y                        READ(IN,444)I,A,Y
   WRITE(IOUT,13)A,B                        WRITE(IOUT,13)A,B
```

Figure 6.5. Using the DATA initialization statement for initializing variable I/O unit numbers.

The DATA statement can be written in other ways. For example,

```
DATA IN/2/,IOUT/1/
```

is equally acceptable. Notice that in both ways of writing the statement the constants are always to the right of their respective variable names and are contained within slashes. There must be a one-to-one correspondence between variable names and constants.

The list of a READ or WRITE statement is of arbitrary length; however, it usually is easier for the novice programmer to manage a program that has several READ or WRITE statements with short lists and several simple FORMAT statements, rather than one READ or WRITE with a long list and a complex FORMAT statement. For this reason, several READ or WRITE statements with short lists may be preferred to one READ or WRITE statement with a long list. With additional experience in writing and debugging (correcting, finding the errors in) input/output statements, you will develop a style that is effective for you.

The parameter f refers to a FORMAT statement which gives the form of the I/O record. For our purposes, an input record is a computer card, and an output record is a line of print on the line printer. In other words, the FORMAT statement describes the format or layout of a record.

3. FORMAT Statements

The FORMAT statement describes the detailed layout of either data on data cards or the output on the line printer. When reading data, it indicates

(i) when to get a new data card,

(ii) which columns of the data card are associated with each variable,

(iii) which columns of the data card are to be skipped, and

(iv) the field descriptor to be associated with the data field.

When printing on the line printer, the FORMAT statement indicates

(i) when to begin printing on a new page,

(ii) when to double space before printing,

(iii) when to single space before printing,

(iv) when to print headings and what headings to print,

(v) where to print the values on a line,

(vi) the field descriptor to be associated with the values, and

(vii) the number of print positions to use to print a value.

The field descriptor is used as a template or mapping to translate between an internal machine respresentation of the data (usually in the binary number system) to an external "people compatible" representation of the data (usually in terms of alphabetic characters and the base ten number system). The field descriptors and their corresponding actions used in this unit are given in Table 6.2

Field Descriptor	Action
Iw	Perform integer conversion from or to a field whose width is w columns.
Fw.d	Perform real conversion from or to a field whose width is w columns with d places to the right of the decimal point.
Ew.d	Same as Fw.d except for exponential form.
nX	Omit next n columns from a data card or insert n blank characters into an output line.
nH	Print the n characters (including blanks) immediately following the H in the next n positions on the output line.

Table 6.2 Field Descriptors and Actions

EXAMPLES

Given the input data card (where ɓ represents a blank),

```
                                   11111111112222
card columns:             12345678901234567890123
punched values:           ɓɓ492ɓɓɓ30.72ɓɓ12345678
```

the READ statement and its associated FORMAT statement

```
    |  READ(5,131) I,A,B,K,Y
131 |  FORMAT(I5,F8.2,F5.1,I2,F3.0)
```

set the variables in the READ statement to the respective values

I:	492	col. 1-5
A:	30.72	col. 6-13
B:	12.3	col. 14-18
K:	45	col. 19-20
Y:	678.	col. 21-23

For the same data card the READ statement and its associated FORMAT statement

```
    |  READ(5,11) I,A,B,K,Y
 11 |  FORMAT(I2,F3.1,3X,F5.1,2X,I2,2X,F2.0)
```

set the variables to the respective values

I:	00	col. 1-2	blank data columns are converted as zeros

A:	49.2	col. 3-5	
B:	30.72	col. 9-13	a decimal point punched in a data card takes precedence over that indicated in the FORMAT
K:	12	col. 16-17	
Y:	56.	col. 20-21	the remainder of the card is ignored

The WRITE statement with its associated FORMAT statement (ɓ means blank)

```
     WRITE(6,9876)
9876 FORMAT(1H1,5X,25HIDENTIFICATIONɓORɓHEADING)
```

prints

IDENTIFICATION OR HEADING

in print positions 6 through 30 at the top of a new page. The same may be accomplished by the statements

```
   WRITE(6,43)
43 FORMAT(31H1ɓɓɓɓɓIDENTIFICATIONɓORɓHEADING)
```

The leftmost character in a print line is not actually printed, but controls the spacing between lines of print, if printing is on a line printer. (This statement is not true for the typewriter/printer used on the 1130, for example.)

You must designate what goes into that character; otherwise, chaos may result. The first field descriptor type encountered in a FORMAT statement, scanning from left to right, must be an H. The first character to the right of the first H controls the spacing between lines of print. Table 6.3 gives the commonly used control characters.

CARRIAGE CONTROL CHARACTER	ACTION
1	Skip to the top of the next page before printing.
0	Double space before printing.
blank	Single space before printing.
+	Print on the current line.

Table 6.3. Carriage control characters for the line printer.

Given the FORMAT statements

```
121     FORMAT(1H1,20X,22HCALENDARƀFORƀTHEƀMONTH)
123     FORMAT(1H0,26X,9HSEPTEMBER)
125     FORMAT(1H ,21X,1HS,2X,1HM,2X,1HT,2X,1HW,2X,1HT,
     1         2X,1HF,2X,1HS,/)
```

the WRITE instructions

```
      WRITE(6,121)
      WRITE(6,123)
      WRITE(6,125)
```

instruct the line printer to print the following headings at the top of the next page. (NOTE: FORMAT statement 125 is punched on more than one card. To signal that a card is to be a continuation of a previous card, a punch is placed in column 6, called the continuation column. Although any punch except zero in column 6 signifies continuation, the digit punches 1, 2, 3, . . . simplify keeping track of the number of continuation cards. APPENDIX I gives the number of continuation cards allowed.)

```
|<-- 20 blanks --->CALENDAR FOR THE MONTH
|
|                      SEPTEMBER
|
|                   S  M  T  W  T  F  S
|
|
|<-- leftmost print position
```

Let's look in detail at these three statements.

```
WRITE(6,121)
```

1H1: carriage control positions the paper so that printing will start at the top of the next page.

20X: places 20 blank characters in the next 20 print positions.

22H: places the 22 characters following the H into the next 22 print positions.

Be sure the character count preceding the H is correct. In this example the 22H indicates that the 22 characters immediately following the H are to be printed. If the character count is too small, the compiler doesn't approve; and if the character count is too large, the next field descriptor or a right parenthesis might get swallowed. Count carefully!

In FORMAT 121, if 20H had been used instead of 22H, the compiler would not know how to interpret the characters TH (in MONTH) and would indicate an error. Also, if 26H had been used, the right parenthesis would have been included in the string of characters to be printed. Then, as the compiler continued to scan to the right, it would not find a right parenthesis and would signal that an error had been committed.

WRITE(6,123)

1H0: carriage control spaces the paper up two lines for printing (double-space printing).

26X: places 26 blank characters in the next 26 print positions.

9H: places the next 9 characters following the H intø the next 9 print positions.

WRITE(6,125)

1Hb̸: carriage control spaces the paper up one line for printing (single-space printing).

21X: places 21 blank characters into the next 21 print positions.

1HS: places S into the next print position.

2X: places 2 blank characters into the next 2 print positions.

etc.

1HS: places S into the next print position.

/: inserts a blank line.

A slash or end of record indicator tells the printer to skip to the next record indicated by the carriage control following it. A slash followed by a right-hand parenthesis actually defines a blank record. Since a blank record contains a blank carriage control, then single spacing results, producing one blank line.

A FORMAT of the general form

```
n FORMAT(/1H␢, . . . )
```

introduces a blank line before printing (double spacing), while

```
n FORMAT(/1H0, . . . )
```

introduces two blank lines before printing (triple spacing), which can also be accomplished by

```
n FORMAT(//1H␢, . . . )
```

Notice that in an output FORMAT a carriage control is required after a slash, unless there is a right parenthesis or another slash (both of which indicate a blank record or line) immediately following the slash.

Similarly,

```
n FORMAT(1H␢, . . . /)
```

introduces a blank line after printing, and

```
n FORMAT(1H␢, . . . //)
```

introduces two blank lines after printing.

```
n FORMAT(1H␢, . . . /1H␢, . . . )
```

introduces no blank lines, simply skipping to the next single spaced record.

```
        n FORMAT(1H␢, . . . //1H␢, . . . )
                      and
        n FORMAT(1H␢, . . . /1H0, . . . )
```

both introduce a blank line (double spacing) between the two printed lines.

```
        n FORMAT(1H␢, . . . ///1H␢, . . . )
                      and
        n FORMAT(1H␢, . . . //1H0, . . . )
```

both introduce two blank lines or triple spacing.

Vertical spacing on the page with various combinations of slashes and carriage controls is limited only by your ingenuity and your needs.

Slashes may also be used in input FORMAT statements, meaning to skip to the next card.

A systematic way of numbering FORMAT statements is desirable. By placing all FORMAT statements at the beginning or end of a program and setting aside a set of statement numbers for FORMAT statements, it is easy to add, delete, or use a previous FORMAT statement. Such a convention is extremely useful for debugging purposes.

More discussion of I/O statements is contained in UNIT #11, but you know enough now to construct and use simple I/O instructions.

4. Refer to UNIT #6 ACTIVITIES TABLE, Activity 1.

5. Before you can construct a complete Fortran program, you must be able to tell the computer when to stop compiling the program and when to stop the execution of the program.

The END statement terminates the end of the compile or translation phase and must go at the physical end of your program. The END statement tells the compiler program that there are no more Fortran statements to compile or translate.

The STOP or CALL EXIT statement terminates the execution phase and must be at the logical end of your program -- that is, at the end of the flow through the flowchart of the program. It is the last Fortran statement executed.

While the statements CALL EXIT and STOP both terminate the execution of the program, their functions may differ slightly on different machines.

Basically the intent of the CALL EXIT statement is to terminate execution of the program and to return control of the computer back to the monitor program that is "in charge" of the overall operation, allowing the computer to receive another job. STOP, on the other hand, not only terminates the execution of the program, but may also arrest the total operation of the computer so that it must be restarted with the START button before another job can be processed.

On the IBM 1130 computer (MS 214) you should preferably use the CALL EXIT statement; otherwise, the computer must be restarted.

On the IBM System 360, you may use either CALL EXIT or STOP, since the compiler is programmed to interpret both statements as returning control back to the monitor program. On WATFIV, however, it is possible to receive a diagnostic warning if the CALL EXIT statement is immediately followed by END. (The reasons for this are probably too sophisticated for you right now. If you really want to know, ask an instructor.) Your program is <u>not</u> incorrect if this warning message appears.

In summary, use STOP or CALL EXIT (warning possible) on the 360; preferably use CALL EXIT on the 1130.

SELF EVALUATION: You should now be ready to construct Fortran programs using I/O statements and run them on a computer. Refer to UNIT #6 ACTIVITIES TABLE, Activities 2 and 3.

ASSESSMENT TASK: Please see your instructor. You will be required to construct a Fortran program using I/O statements and run it on both the 360 and the 1130.

WHAT NEXT? You are now ready to tackle serious programming, making use of the computer's decision making capability. Continue with Unit 8.

UNIT #8 (COMSC)

TITLE: CONDITIONAL BRANCHING OR TRANSFER STATEMENTS

RATIONALE: One of the powerful capabilities of a computer is its ability to make certain logical "decisions" - that is, its ability to do a certain set of operations under a certain condition and to do some alternate set of operations if some alternate condition prevails. In this unit you will learn how to use Fortran for decision-making.

OBJECTIVE: At the end of this unit you will be able to construct the four conditional transfer statements in Fortran - the computed GO TO, the assigned GO TO, the arithmetic IF, and the logical IF - and to construct and follow the logic of program segments that make use of these statements.

PREREQUISITES: UNIT #6

ACTIVITIES: The first set of activities (A-D) is intended to acquaint you with the forms of the four types of conditional transfer statements and how these statements operate. Activities E-G are intended to show you how and when the four statements may be used in actual problem solving situations; a sample problem is given, and five sample programs are used to illustrate the statements.

A. Computed GO TO.

Already you have been introduced to the unconditional GO TO statement. In this unit you will learn about two more GO TO statements in which the transfer is conditional, rather than unconditional. The first of these is the computed GO TO, and the second is the assigned GO TO.

Refer to the UNIT #8 ACTIVITIES TABLE, Activity 1.

Several things about computed GO TO statements need special emphasis:

1. In the general form of the computed GO TO,

   GO TO (n_1, n_2, ..., n_k),i

   i is called the index and must be an integer name. It must also be preceded by a comma.

2. The relationship between the index and the statement numbers in the list, n_1, n_2, ..., n_k, is positional. That is, when the index is one, the first statement number is used; when the index is two, the second statement number is used; etc.

3. Normally the value of the index should not be allowed to exceed the number of statement numbers in the list. ($1 \leq i \leq k$) In IBM System/360 FORTRAN, however, if the value of the index does exceed the number of statement numbers in the list, then the

first executable statement following the computed GO TO is executed.

4. The value of the index is never allowed to be zero or negative.

B. Assigned GO TO.

The assigned GO TO is probably less often used then the computed GO TO, and it is not available on all compilers. (The assigned GO TO is not available in 1130 Fortran.)

Refer to UNIT #8 ACTIVITIES TABLE, Activity 2.

When the assigned GO TO is used, the index, which is i in the general form

GO TO i, $(n_1, n_2, \ldots, n_3)$

must have previously been assigned one of the statement numbers contained in the list by use of an ASSIGN statement. The index is not related to the positions of the statement numbers, as in the case of the computed GO TO; but transfer is to the statement number in the list which has been assigned to the index.

In general, the computed GO TO can be used to accomplish anything that the assigned GO TO can do, as you'll see in the example later in this unit.

C. Arithmetic IF.

The arithmetic IF is an especially important conditional transfer statement. While the computed GO TO and the assigned GO TO allow any number of possible transfers, the arithmetic IF provides branching only for the conditions of negative, zero, and positive.

Refer to UNIT #8 ACTIVITIES TABLE, Activity 3.

The arithmetic expression e in the general form

$$\text{IF }(e)\ n_1,n_2,n_3$$

can be any valid integer or real arithmetic expression. For example,

```
IF (X**4+3.0/X) 5,6,20
```

is a valid IF statement, as is

```
IF(N)10,30,5
```

Two of the conditions may cause transfer to the same statement. In other words, a statement number may be used twice. For example,

```
IF(Z)5,5,10
```

causes transfer to statement number 5 if the argument is negative or zero and to statement number 10 if the argument is positive.

D. Logical IF

Another type of IF statement, the logical IF, is available with many computer systems. (The logical IF is not available in IBM 1130 Fortran.) The logical IF branches on one of two conditions, depending upon whether the argument is true or false.

Refer to UNIT #8 ACTIVITIES TABLE, Activity 4.

There are three main pitfalls in using the logical IF:

1. If the argument cannot be determined to be true or false, an error will result. Whenever you write a logical IF, ask yourself the question, "Is the argument true or false?"

   For example,

   ```
   IF(5*N)GO TO 6
   ```

   is not correct. Is 5*N true or is it false? Actually, it's nonsense even to ask the question!

   On the other hand,

   ```
   IF(5.GT.N)GO TO 6
   ```

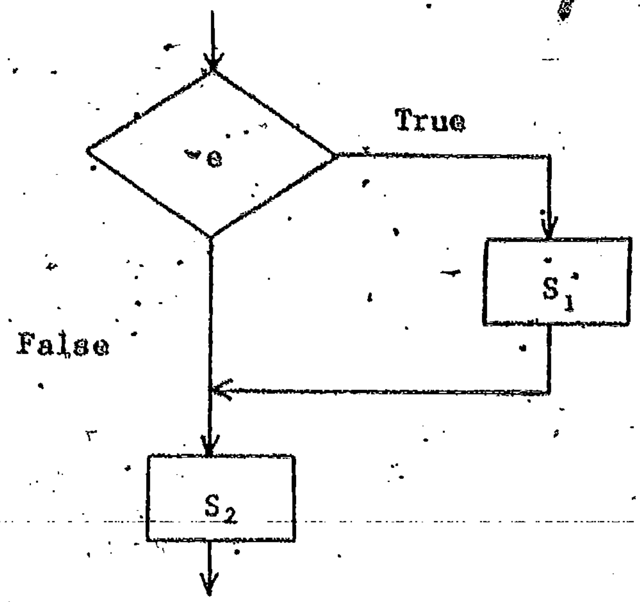
   is correct. Whether 5 is greater than N (the "true" case), or not (the "false" case), can be determined immediately when the value of N is known.

2. Lack of understanding of the operation of the logical IF can cause incorrect results in a program. Fix in mind firmly the paths of execution in the logical IF.

   Suppose S_1 and S_2 represent two executable Fortran statements. Here is a flowchart for the general form of the logical IF

   $IF(e)S_1$
   S_2

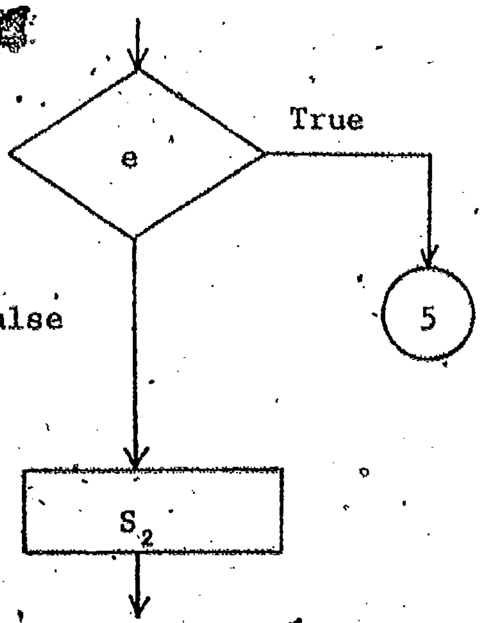
   provided that S_1 is not a transfer statement:



Notice that only S_2 is executed if the argument is false, but that both S_1 and S_2 (in that order) are executed if the argument is true.

Of course, if S_1 is a transfer statement, then only S_2 will be executed if the argument is false, and only S_1 will be executed if the argument is true. Here is a flowchart for the general form

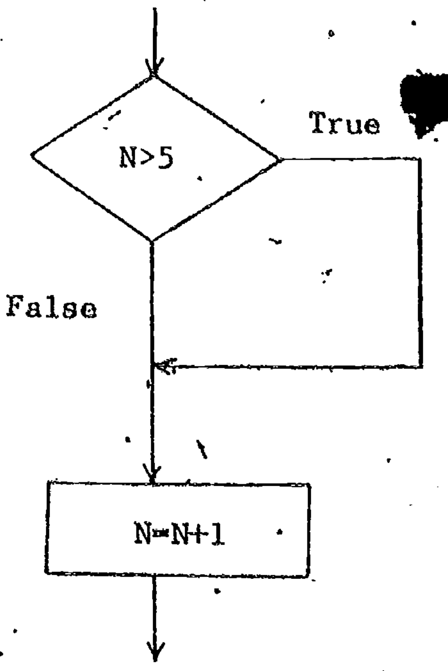
```
IF(e)GO TO 5
S2
```



3. Do you see anything wrong with the following statements?

```
      IF(N.GT.5)GO TO 2
    2 N=N+1
```

You should! Look at a flowchart for the statements.



The statement N=N+1 is executed if the argument is true; it is also executed if the argument is false! Nonsense! The IF statement might as well have been left out!

E. Putting it all together.

How do you decide what type of conditional transfer statement to use? The answer to that question is determined by the problem itself and in part by your preference.

In general, assuming that the logical IF is available on the computer you are using, logical IF's and arithmetic IF's can be used interchangeably. This is particularly true if only two branches from the arithmetic IF are being used. For example,

```
          IF(X)5,5,7
```

could be used interchangeably with

```
          IF(X.GT.0.0)GO TO 7
           Statement #5 goes here.

                  or

          IF(X.LE.0.0)GO TO 5
           Statement #7 goes here.
```

If all three branches of the arithmetic IF are being used, however, then two logical IF statements are required. For example,

```
          IF(X)5,6,7

             and

          IF(X.GT.0.0)GO TO 7
          IF(X.EQ.0.0)GO TO 6
           Statement #5 goes here.
```

could be used interchangeably.

The computed GO TO is particularly useful when several branches are needed, especially if the condition of transfer is based on consecutive integers. For example, suppose that you wish to make a count of students who are freshmen (coded 1), sophomores (coded 2), juniors (coded 3), and seniors (coded 4). One computed GO TO can test for all four cases and transfer to the proper counter. This is much simpler than using three logical IF's or two arithmetic IF's which would be required to accomplish the same task.

The assigned GO TO is very general, since statement number assignments are determined by whatever conditions the programmer wishes. It does not depend upon consecutive integers as the computed GO TO does; it does not check just for positive, negative, or zero or just for true or false. Any set of predetermined conditions may be used for statement number assignments for the index of the assigned GO TO. As already pointed out, however, the same thing can be accomplished by assigning values to the index of the computed GO TO.

F. Simple counters.

In a moment we'll look at an example that illustrates the conditional transfer statements. But before we do that, you need to know about simple integer counters. Refer to UNIT #8 ACTIVITIES TABLE, Activity 5.

G. A sample program.

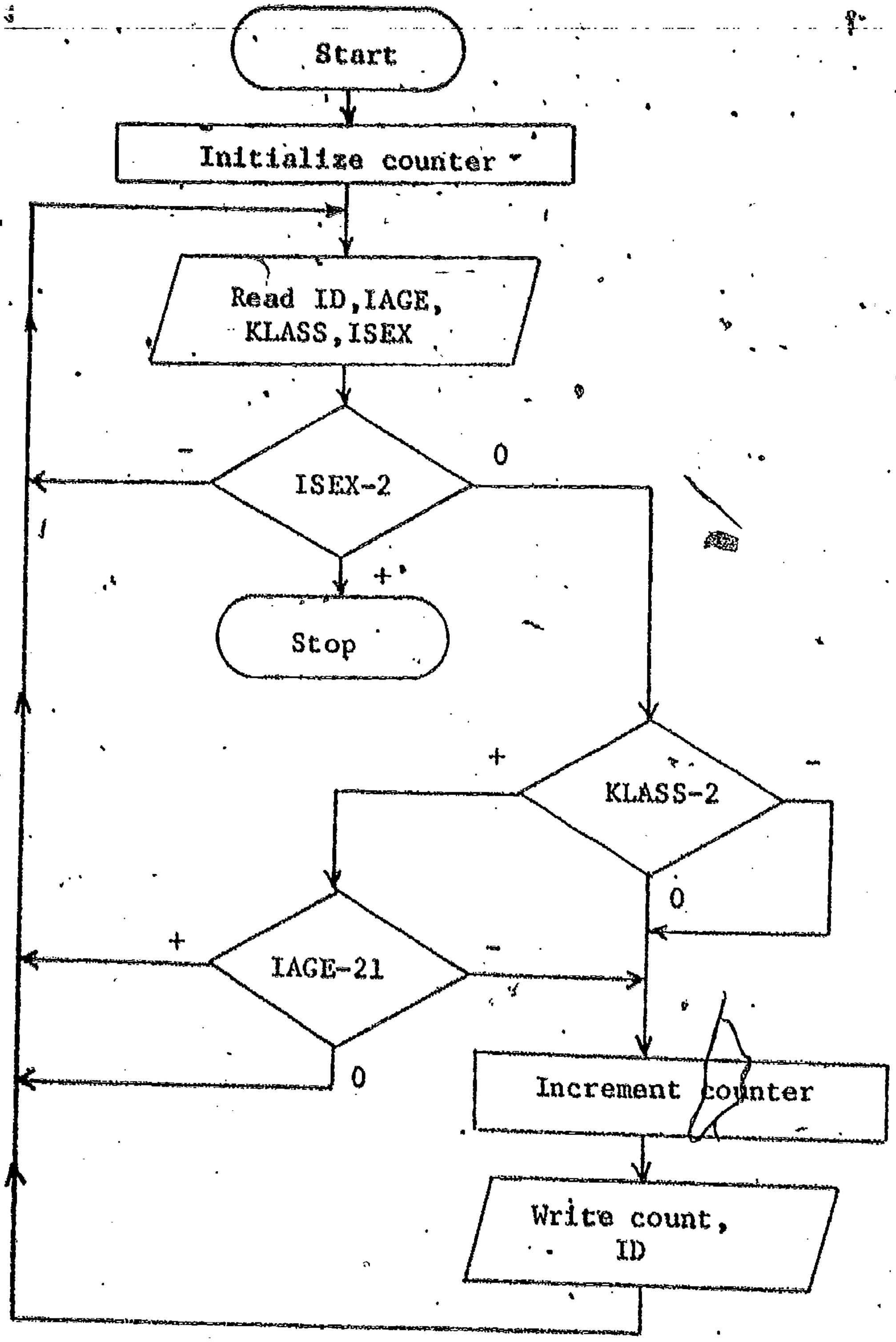
Now let's look at some programs which make use of IF, computed GO TO, and assigned GO TO statements. These programs will also illustrate the use of a simple integer counter. Here is a statement of the problem:

Suppose we have a set of data cards, each of which has a student's name and ID number, and his age, sex, and classification punched in the columns shown below.

Columns 1 - 20:	Name
Columns 22 - 25:	ID number
Columns 26 - 27:	Age
Column 28:	1 for freshman
	2 for sophomore
	3 for junior
	4 for senior
Column 29:	1 for female
	2 for male
	3 for last card

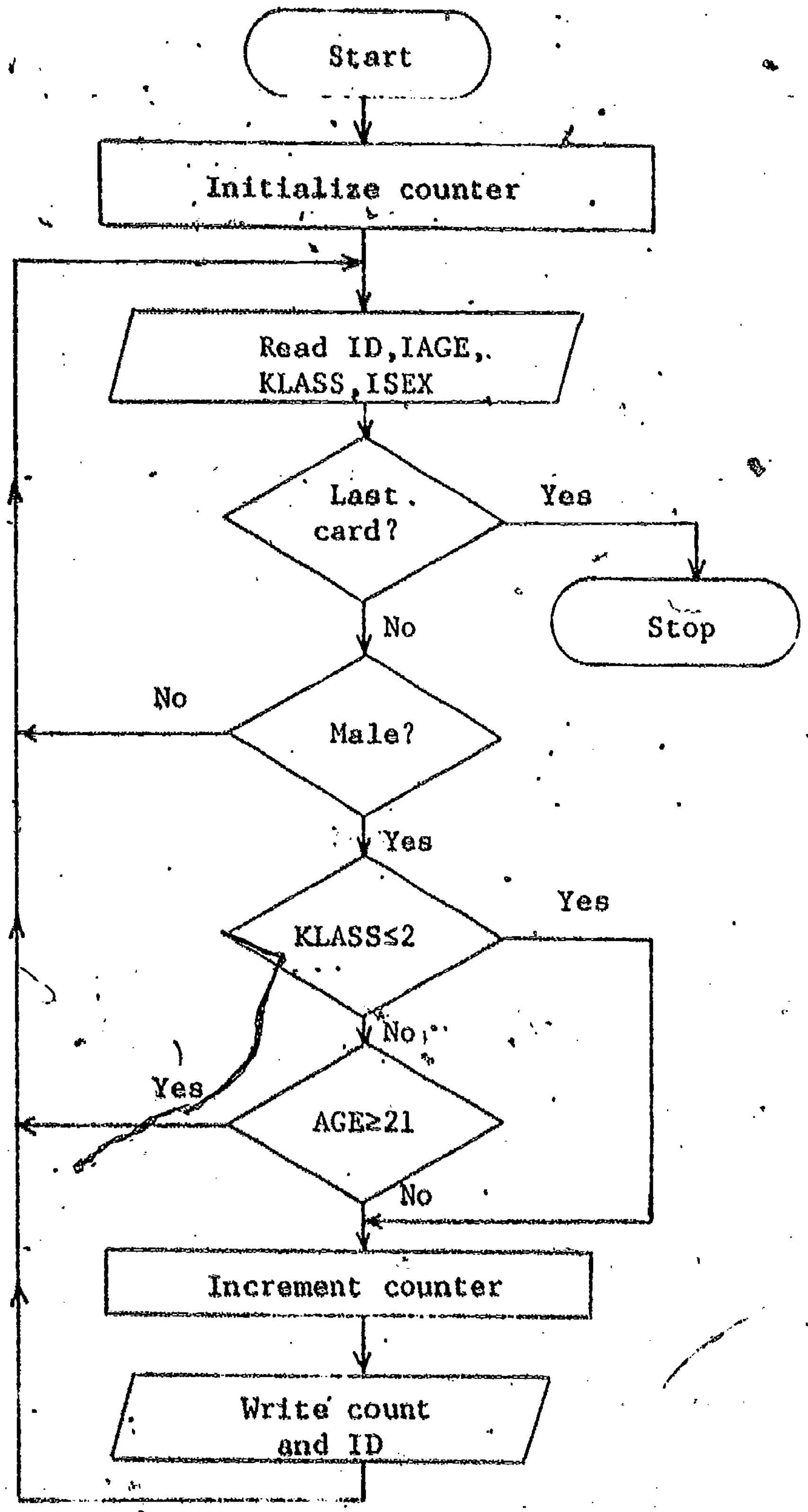
We want a list of ID numbers and a head count of male students who are freshmen or sophomores <u>or</u> who are under 21.

There are a number of ways to program the solution of this problem; we will examine five programs with their flowcharts. Figure 8.1 shows a solution that uses only arithmetic IF statements for decisions, and Figure 8.2 contains the same program using only logical IF statements. Figure 8.3 shows a program using arithmetic IF and computed GO TO statements for decisions, while Figure 8.4 shows a program using logical IF and computed GO TO statements for decisions. Finally, in Figure 8.5 decisions are made with logical IF, computed GO TO, and assigned GO TO statements.



```
      DATA IN,IOUT/5,6/,KOUNT/0/
   10 READ(IN,1)ID,IAGE,KLASS,ISEX
      IF(ISEX-2)10,11,12
   12 STOP
   11 IF(KLASS-2)8,8,13
   13 IF(IAGE-21)8,10,10
    8 KOUNT=KOUNT+1
      WRITE(IOUT,2)KOUNT,ID
      GO TO 10
    1 FORMAT(21X,I4,I2,2I1)
    2 FORMAT(1H ,I5,1H.,2X,I4)
      END
```

Figure 8.1. Program using only arithmetic IF statement for decisions.

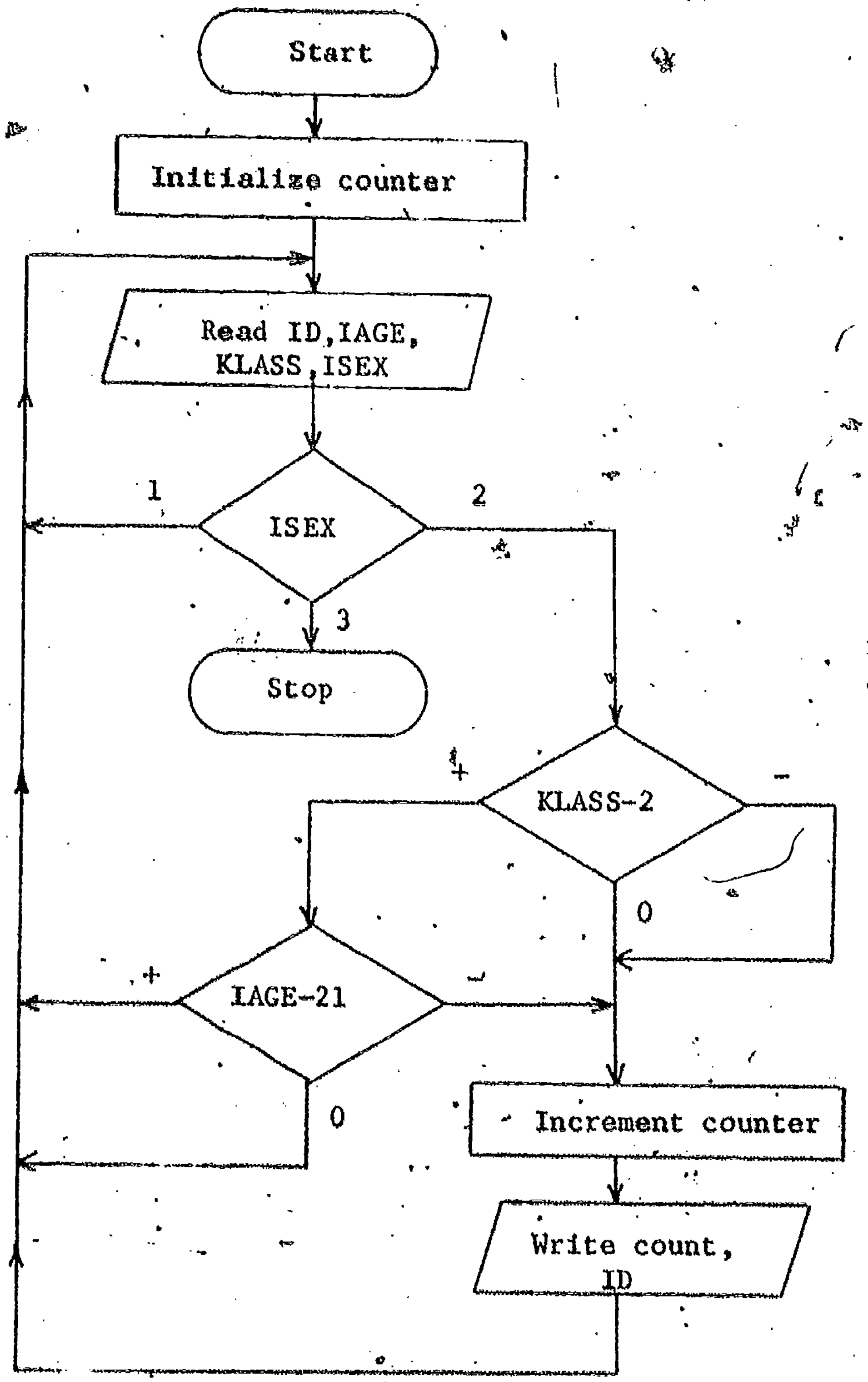


```
      DATA IN,IOUT/5,6/,KOUNT/0/
   10 READ(IN,1)ID,IAGE,KLASS,ISEX
      IF(ISEX.EQ.3)STOP
      IF(ISEX.NE.2)GO TO 10
      IF(KLASS.LE.2)GO TO 8
      IF(IAGE.GE.21)GO TO 10
    8 KOUNT=KOUNT+1
      WRITE(IOUT,2)KOUNT,ID
      GO TO 10
    1 FORMAT(21X,I4,I2,2I1)
    2 FORMAT(1H ,I5,1H.,2X,I4)
      END
```

The program can also be written with only two IF statements.

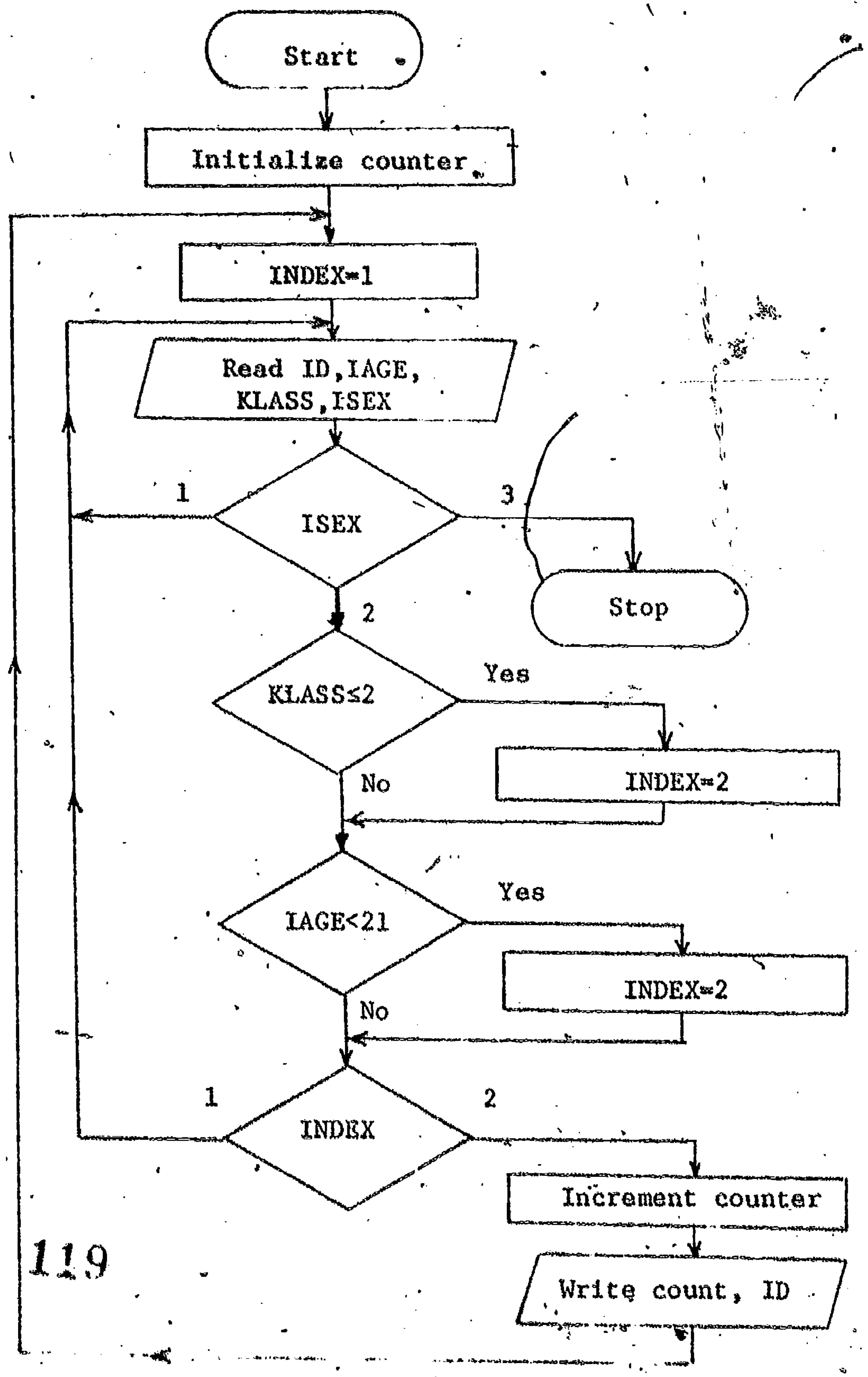
```
      DATA IN,IOUT/5,6/,KOUNT/0/
   10 READ(IN,1)ID,IAGE,KLASS,ISEX
      IF(ISEX.EQ.3)STOP
      IF(ISEX.NE.2.OR.KLASS.GT.2.AND.IAGE.GE.21)
     $     GO TO 10
      KOUNT=KOUNT+1
      WRITE(IOUT,2)KOUNT,ID
      GO TO 10
    1 FORMAT(21X,I4,I2,2I1)
    2 FORMAT(1H ,I5,1H.,2X,I4)
      END
```

Figure 8.2. Programs using only logical IF statements for decisions. (These programs cannot be run on the IBM 1130 computer because of the logical IF statements.)



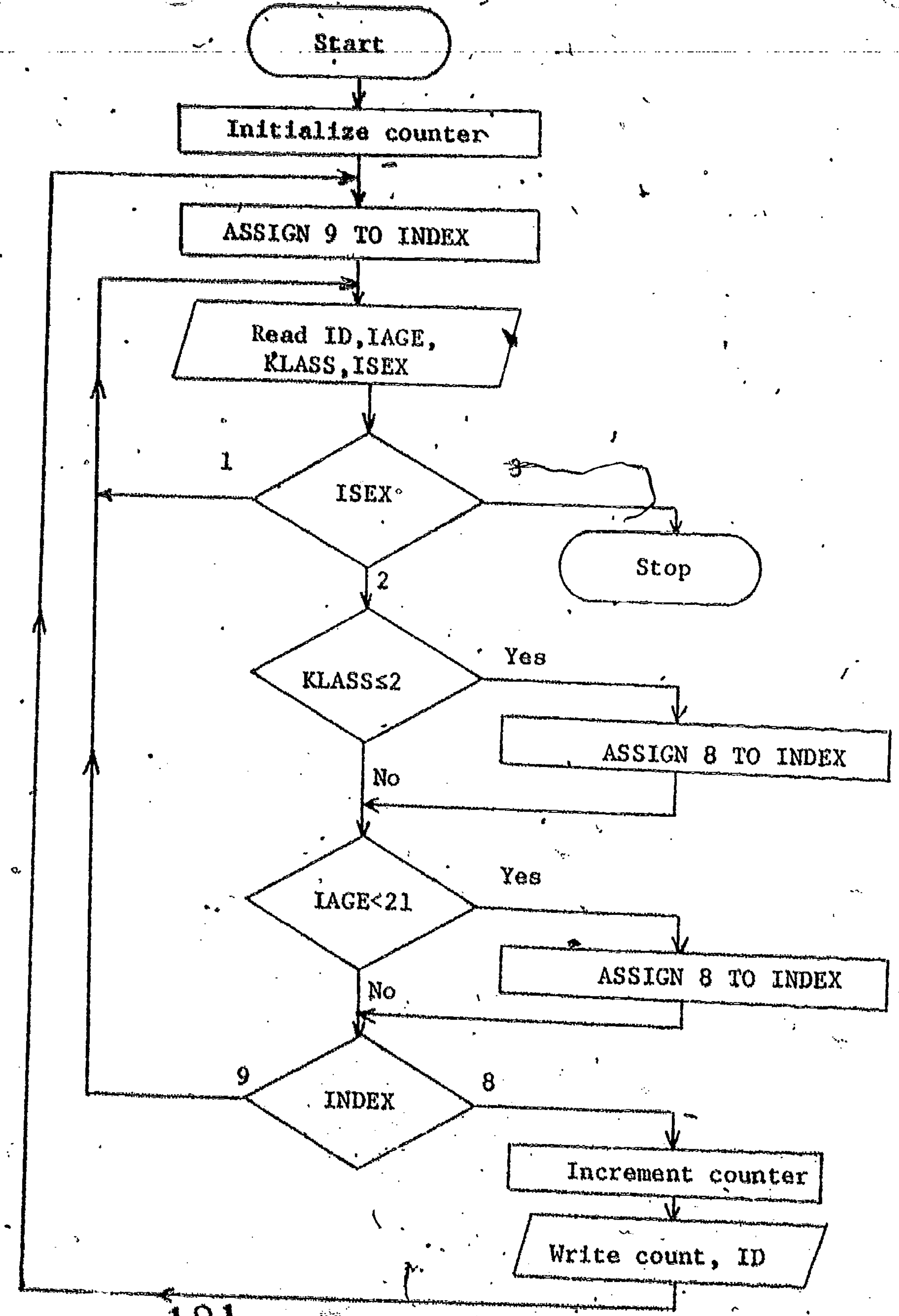
```
      DATA IN,IOUT/5,6/,KOUNT/0/
   10 READ(IN,1)ID,IAGE,KLASS,ISEX
      GO TO (10,11,12),ISEX
   12 STOP
   11 IF(KLASS-2)8,8,13
   13 IF(IAGE-21)8,10,10
    8 KOUNT=KOUNT+1
      WRITE(IOUT,2)KOUNT,ID
      GO TO 10
    1 FORMAT(21X,I4,I2,2I1)
    2 FORMAT(1H ,I5,1H.,2X,I4)
      END
```

Figure 8.3. Program using computed GO TO and arithmetic IF statements for decisions.



```
      DATA IN,IOUT/5,6/,KOUNT/0/
   10 INDEX=1
    9 READ(IN,1)ID,IAGE,KLASS,ISEX
      GO TO (9,11,12),ISEX
   12 STOP
   11 IF(KLASS.LE.2)INDEX=2
      IF(IAGE.LT.21)INDEX=2
      GO TO (9,8),INDEX
    8 KOUNT=KOUNT+1
      WRITE(IOUT,2)KOUNT,ID
      GO TO 10
    1 FORMAT(21X,I4,I2,2I1)
    2 FORMAT(1H ,I5,1H.,2X,I4)
      END
```

Figure 8.4. Program using computed GO TO and logical IF statements for decisions. (This program cannot be run on the IBM 1130 computer because of the logical IF statements.)



```
      DATA IN,IOUT,KOUNT/5,6,0/
   10 ASSIGN 9 TO INDEX
    9 READ(IN,1)ID,IAGE,KLASS,ISEX
      GO TO(9,11,12),ISEX
   12 STOP
   11 IF(KLASS.LE.2)ASSIGN 8 TO INDEX
      IF(IAGE.LT.2)ASSIGN 8 TO INDEX
      GO TO INDEX,(9,8)
    8 KOUNT=KOUNT+1
      WRITE(IOUT,2)KOUNT,ID
      GO TO 10
    1 FORMAT(21X,I4,I2,2I1)
    2 FORMAT(1H ,I5,1H.,2X,I4)
      END
```

Figure 8.5. Program using computed GO TO, assigned GO TO, and logical IF statements. (This program cannot be run on the IBM 1130 computer because of the logical IF and assigned GO TO statements.)

Which of these programs cannot be run on the 1130 computer and why?

SELF EVALUATION: Write a Fortran program for printing the Dean's List. The data input will consist of a set of data cards, each containing a student's ID number (columns 1-5), the student's classification code (column 6, explained in the table below), the number of hours in which he is enrolled (columns 7 and 8), and his grade point average (columns 9-13, decimal point punched with three digits to the right of the decimal point).

Classification	Code
Freshman	1
Sophomore	2
Junior	3
Senior	4
Special	5

In order to qualify for the Dean's List, suppose that a student must be a freshman, sophomore, junior, or senior; must be enrolled in twelve or more hours; and must have a grade point average of 3.50 or better.

The last card will have only a 6 punched in column 6 where the classification code is punched.

The program is to make a list of ID numbers and grade point averages of students who qualify for the Dean's List. Terminate execution when the last card is encountered.

Use at least one computed GO TO, at least one logical IF, and at least one arithmetic IF in your program.

Run your program on the 360 computer until it is correct. Punch the set of test data shown below for trying out your program.

```
         11111111112
12345678901234567890

111114163.678
222221152.678
333332123.500
444443104.000
555555163.850
666663181.875
777772133.750
     6
```

ASSESSMENT TASK: Turn in to your instructor the printer output and the program deck for the problem you worked in the self evaluation section.

8.18

Your instructor will give you additional assessment tasks. You will be required to construct one or more Fortran programs and/or program segments making use of conditional transfer statements.

WHAT NEXT? Go ahead to UNIT #9.

UNIT #9 (COMSC)

TITLE: ARRAYS AND SUBSCRIPTED VARIABLES

RATIONALE: In order to handle large groups of data which share some common characteristic, such as points on a curve or grade point averages of students enrolled in a university, some means of grouping these data together under a single variable name, recognizing that the data share a common characteristic, and some means of referring to specific data items, recognizing the uniqueness of each data item, are needed. In Fortran, arrays and subscripts are used for this purpose.

OBJECTIVE: At the end of this lesson you will be able to construct a Fortran program that makes use of one-dimensional and/or two-dimensional arrays.

PREREQUISITES: UNIT #8.

ACTIVITIES: Activities A, B, and C are designed for students who cannot identify the terms subscript and element as used in a set (or an array) of items. If a set of values of x containing n elements in the notation

$$x_1, x_2, x_3, \ldots, x_n$$

and if a set of values of x containing n rows and m columns in the notation

$$\begin{array}{lllll} x_{1,1} & x_{1,2} & x_{1,3} & \cdots & x_{1,m} \\ x_{2,1} & x_{2,2} & x_{2,3} & \cdots & x_{2,m} \\ x_{3,1} & x_{3,2} & x_{3,3} & \cdots & x_{3,m} \\ & & \cdots & & \\ x_{n,1} & x_{n,2} & x_{n,3} & \cdots & x_{n,m} \end{array}$$

are meaningful to you, then skip to Activity D, page 9.6 If these notations are not meaningful to you, then go through Activities A-C. Activity A describes arrays; Activity B describes subscripts for arrays of one dimension; and Activity C describes arrays of two and three dimensions.

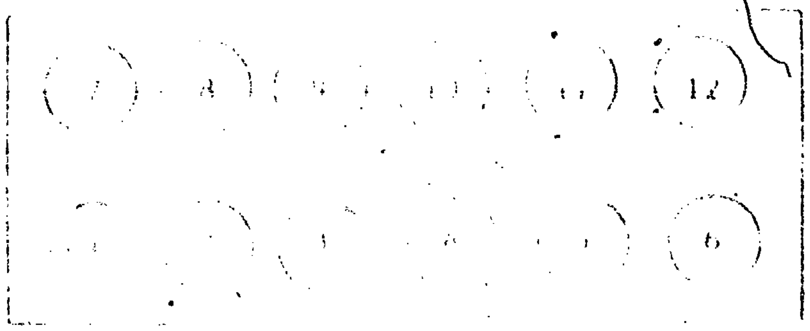
A. What is an <u>array</u>? An array is simply a group of itmes with some common property. A dozen eggs is an array of eggs, sharing the common property of "eggness." A group of red things is an array of red things, sharing the common property of "redness." People can be grouped into arrays in many ways: an array of males and an array of females; an array of tall people and an array of short people, provided that <u>tall</u> and <u>short</u> are properly defined; an array of blue-eyed people and an array of brown-eyed people; an array of red-haired people, an array of blond-haired people, and an array of all the rest; or simply an array of people.

In Fortran, arrays are usually composed of numbers; but they may also be composed of alphameric data, alphameric and numeric data [illegible], and other types of data. For example, if we were considering an array containing the grade point averages of students, we would have an array of real numbers.

The individual data items in an array are referred to as elements of the array.

B. What is a subscript? A subscript is an integer number that describes uniquely the position in the array of an individual element. It is important to remember that a subscript is a positional reference. Let's expand the notion of a subscript with some examples.

Example 1. Consider a dozen eggs in a carton, represented by [illegible]



We can number the eggs in any arbitrary manner; but some ordered, systematic numbering, as indicated in the diagram, is certainly neater and more convenient.

Suppose that you are told to point to egg number 3. You certainly have no difficulty doing that, do you? Egg number 3 is uniquely specified by being in the third position (as numbered from the reference point that was arbitrarily chosen). Similarly, all the other eggs are uniquely specified by the number of the position in which each egg is located.

Considering these eggs as an array of eggs, we can refer to the positional numbers as subscripts. We could refer to the egg in the ~~third position~~ as egg$_3$, read as egg sub-three.

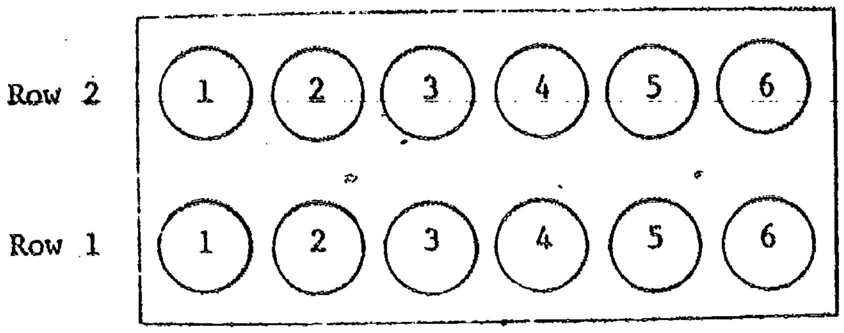
Example 2. Consider the days of the year. They too form an array, and each day is uniquely specified by a number. For example, the birthday of Abraham Lincoln is the 43rd day of the year. (It's also February 12, of course; we'll consider that notation later.) In array jargon, we could say that Abe was born on day$_{43}$ or day-sub-forty-three.

Example 3. Consider the seats in an auditorium as an array of seats. Suppose that you are assigned seat number 108, or seat-sub-one-hundred-eight. That seat number defines uniquely a position in the auditorium in which you may sit.

Do you now have a firm grasp on the notion of subscripts as positional numbers? If not, ponder the examples again. If so, then consider this question: Does it make sense to speak of position 5 1/2 in the egg carton or seat 75.32 in the auditorium? Hopefully you answered, "Certainly not!" In the definition of subscripts at the beginning of Activity 2, a subscript was said to be integer; that restriction should make sense to you now.

The arrays considered in this section are said to be one-dimensional arrays because they have one subscript or one positional number associated with each position in the arrays. There are other ways of looking at the positions in arrays, however; one of these will be considered in the next section.

C. Let's go back to the eggs and number them in a different way.



Clearly, there are two rows of eggs, each containing six eggs. We can specify uniquely any egg in the carton now by stating two positional numbers; for example, the third egg on row one is egg$_{1,3}$ or egg-sub-one-three. (The row number is arbitrarily stated first.)

Consider again the year as an array of days. The year can also be thought of as being divided into groups of days, or months. Thus, the 43rd day of the year may also be referred to by saying "the 12th day of the 2nd month," or day$_{2,12}$, having two positional numbers.

Or consider the seats of an auditorium. They may be numbered by row and by seat - row 5, seat 4, for example, or seat$_{5,4}$.

In these examples, two positional numbers or subscripts were used in order to identify uniquely each position in the array. Such a system of numbering is said to be two-dimensional, and the arrays are two-dimensional arrays.

Whether an array is one-dimensional or two-dimensional is really a matter of preference. Certain types of problems may, however, be more easily accommodated by a one-dimensional array, while for others a two-dimensional array may be better One-dimensional arrays do require less execution time in the computer.

What about the eggs in the carton? Both one- and two-dimensional arrays can conveniently be used. In other words, to think of the eggs as twelve eggs or as two rows of six eggs each makes little difference probably. But with the year considered as an array of days, we usually prefer to think in terms of a two-dimensional array -- that is, months and days.

In the case of the seats in an auditorium, it's definitely easier to find a seat if you are given the row and the seat number; rather than just a single number for the seat.

There are also arrays of more than two dimensions. For example, we could think of the year in terms of months, weeks, and days. The third day of the second week of the sixth month would be designated $\text{day}_{6,2,3}$, having three subscripts. Or the seats of the auditorium might be divided into sections, rows, and seats. We might have section 1, row 5, seat 3, designated $\text{seat}_{1,5,3}$. These are examples of three-dimensional arrays.

D. Refer to UNIT #9 ACTIVITIES TABLE, Activity 1.

SELF EVALUATION: Refer to UNIT #9 ACTIVITIES TABLE, Activities 2 and 3.

ASSESSMENT TASK: Please see your instructor. You will be required to write a Fortran program making use of arrays and run it on a computer.

WHAT NEXT? Handling arrays in the problems in this unit has been rather cumbersome. In UNIT #10 you will learn a simpler way of handling arrays.

UNIT #10 (COMSC)

TITLE: LOOPS

RATIONALE: One of the great assets of a computer is the ability to repeat a set of operations time and time again. Many applications in data processing and problem solving require iterative procedures, that is, procedures requiring repetition of all or parts of a program. The term usually applied to such repetitive operations is looping or loops. In this unit you will learn more about looping in Fortran. You will also get a better handle on making use of arrays in programs.

OBJECTIVE: At the end of this unit you will be able to construct Fortran program loops, using the DO statement.

PREREQUISITES: UNIT #9

ACTIVITIES:

A. Suppose that you were given the task of writing a program segment that would find the sum of the elements of a one-dimensional array. One solution to the problem is shown in Figure 10.1. A flowchart is shown in Figure 10.2.

Notice that a logical IF might be used instead of the arithmetic IF in line 5. What logical IF would you use?

____

```
C***********************************************
C*****  PROGRAM SEGMENT FOR SUMMING        *****
C*****    THE N ELEMENTS OF THE ARRAY A,  *****
C*****    USING K FOR THE COUNTER.        *****
C***********************************************
C*****  INITIALIZE SUM AND COUNTER.
      SUM=0.0
      K=1
C*****  PERFORM THE SUMMING.
    2   SUM=SUM+A(K)
C*****  INCREMENT COUNTER AND TEST IT.
        K=K+1
        IF(K-N)2,2,3
C*****  SUMMING COMPLETED.  WRITE RESULTS.
    3 WRITE(6,10)SUM
```

Figure 10.1. Program for summing elements of an array.

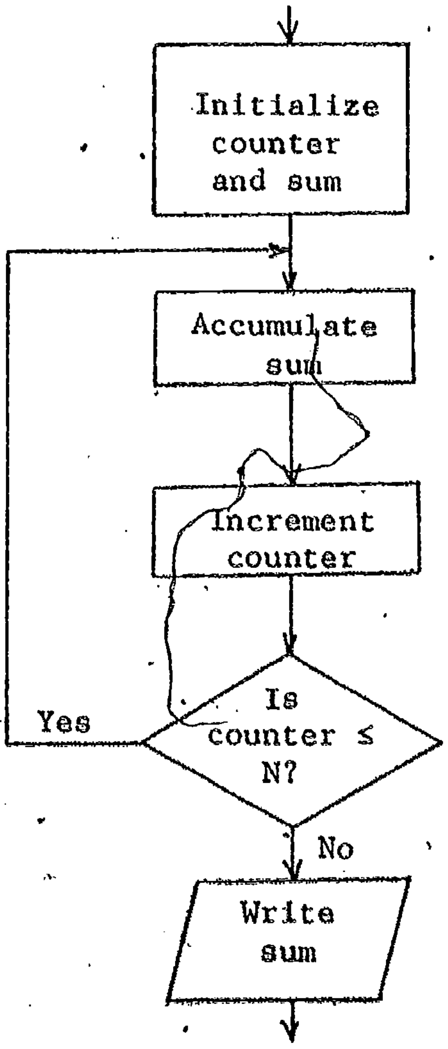


Figure 10.2. Flowchart for summing elements of an array showing loop.

Let's analyze the program segment carefully, using the line numbers in the left margin. Lines 3 - 5 form a program loop, that is, a set of statements that are repeated. In this case, the loop is to be executed N times, since there are N elements in the array to be added into the sum. After the N elements have been added, the looping is discontinued; and; in this particular program segment, the WRITE statement at line 6 is executed.

There are three statements altogether that control the looping: lines 2, 4, and 5 of the program segment. The three statements initialize, increment, and test the counter.

There are three keywords here: initialize, increment and test. Don't forget them, because you will use them later.

Initialize, increment, test.

Fortran provides a special statement for looping that handles the details of loop control for the programmer. This statement is the DO statement. The three control statements in the program segment (lines 2, 4, and 5) that initialize, increment, and test the counter can be replaced with a single DO statement illustrated in Figure 10.3. A flowchart is shown in Figure 10.4. In the DO statement, "K=1,N,1" contains the three control functions: "K=1" initializes the counter to 1; "N" serves as the test value (that is, as long as K≤N, looping will continue); the last "1" serves as the increment value

```
C  PROGRAM SEGMENT FOR SUMMING
C     THE N ELEMENTS OF AN ARRAY,
C     USING A DO STATEMENT.
       SUM=0.0
       DO 2 K=1,N,1
          SUM=SUM+A(K)
     2 CONTINUE
       WRITE(6,10)SUM
```

Figure 10.3. Program segment with DO statement for looping.

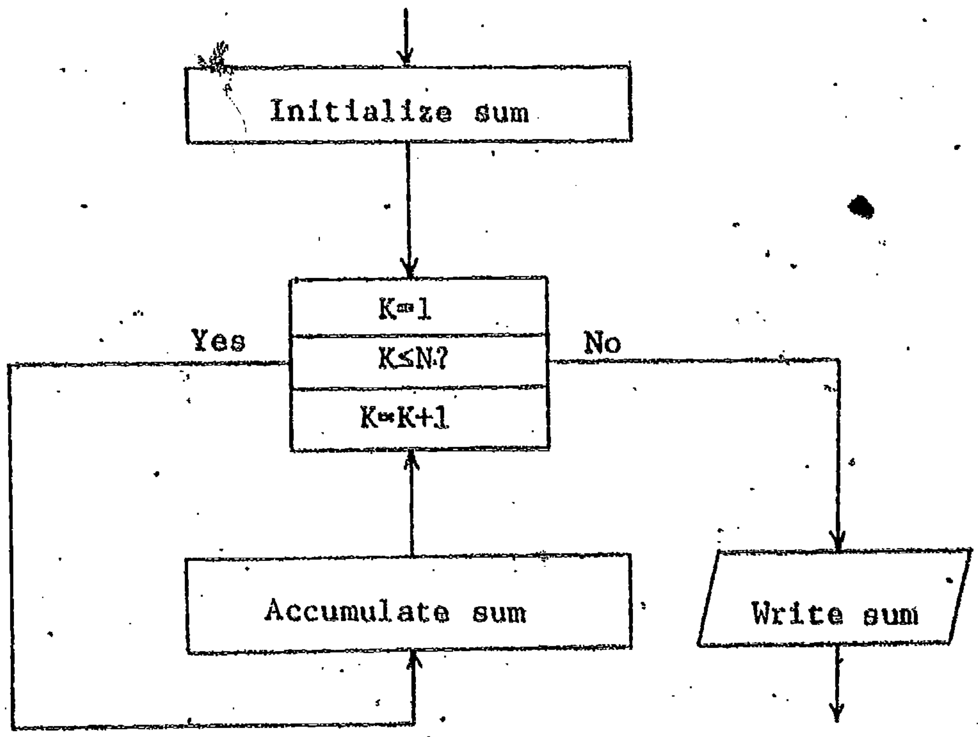


Figure 10.4. Flowchart for summing N elements of an array, using a DO loop.

(that is, K is incremented by one each time). The first part of the DO statement, "DO 2," defines the range of the loop, consisting of the statements immediately following the DO statement down to and including statement number 2. In this particular case, the range consists of two statements. (The CONTINUE statement is actually optional in this case; we could have placed the statement number 2 on SUM=SUM+A(K) and omitted the CONTINUE statement. The purpose of CONTINUE will be discussed later.)

When the looping is completed, that is, when K is no longer less than or equal to N (K is greater than N, in other words), then the first executable statement following the last statement in the range of the DO is executed. In the example, the WRITE statement would be executed.

The CONTINUE statement is a useful way to delimit the range of a DO. It also provides a way of visually delimiting DO ranges by allowing the "body" of a DO range to be indented from the DO and the CONTINUE statements.

Place the following program segment in parenthesized or indented format. Place a CONTINUE statement at the end of each DO loop. (See the last page of this unit for the solution.)

```
      DO 17 I=1,N
      DO 13 K=1,N
      SUM=0.
      DO 15 J=1,N
   15 SUM=SUM+A(I,J)*B(J,K)
   13 C(I,J)=SUM
   17 CONTINUE
```

B. Refer to Activity 1 of UNIT #10 ACTIVITIES TABLE.

C. Looping can be accomplished in other ways besides that of using the DO statement. You have seen one way already in the sample program in Figure 10.1, in which a counter and an IF statement were used.

Some loops are handled more easily with DO statements, such as in the program segment in Figure 10.3. But some loops are handled more easily by using a single conditional transfer statement -- an IF statement or a computed GO TO statement, for example. Let us consider a problem illustrating looping without a DO statement.

Suppose that we are to use a computer for compiling the Dean's List, and suppose that a student must have a grade point average of 3.0 or better in order to be on the Dean's List. Also suppose that there is one data card per student, each of which contains a five-digit ID number in columns 1-5, the student's classification in column 6 (1 for freshman, 2 for sophomore, 3 for junior, and 4 for senior), and the grade point average in columns 7-11 (decimal point punched). If the enrollment is very large, it would be inconvenient to count the cards so that the exact number would be known; it would be more convenient to place a "sentinal" card or a "trailer" card at the end of the data cards and program the computer to recognize that card as the last card. One way to do this would be to place a card with a 5 punched in column 6, the classification column, at the end of the data. Since only 1, 2, 3, and 4 are legitimate classification codes, a 5 can be used to terminate the loop. A computed GO TO is very useful for this purpose.

A program to prepare the Dean's List is shown in Figure 10.5. The flowchart is shown in Figure 10.6. This example is similar to the problem for self evaluation in UNIT #8.

```
C   PROGRAM FOR COMPILING THE DEAN'S LIST
   20 READ(5,1)ID,KLASS,GPA
    1 FORMAT(I5,I1,F5.3)
      GO TO(10,10,10,10,11),KLASS
   11 STOP
   10 IF(GPA-3.0)20,21,21
   21 WRITE(6,2)ID,GPA
    2 FORMAT(1H ,I5,2X,F5.3)
      GO TO 20
      END
```

Figure 10.5. Program looping controlled by a computed GO TO.

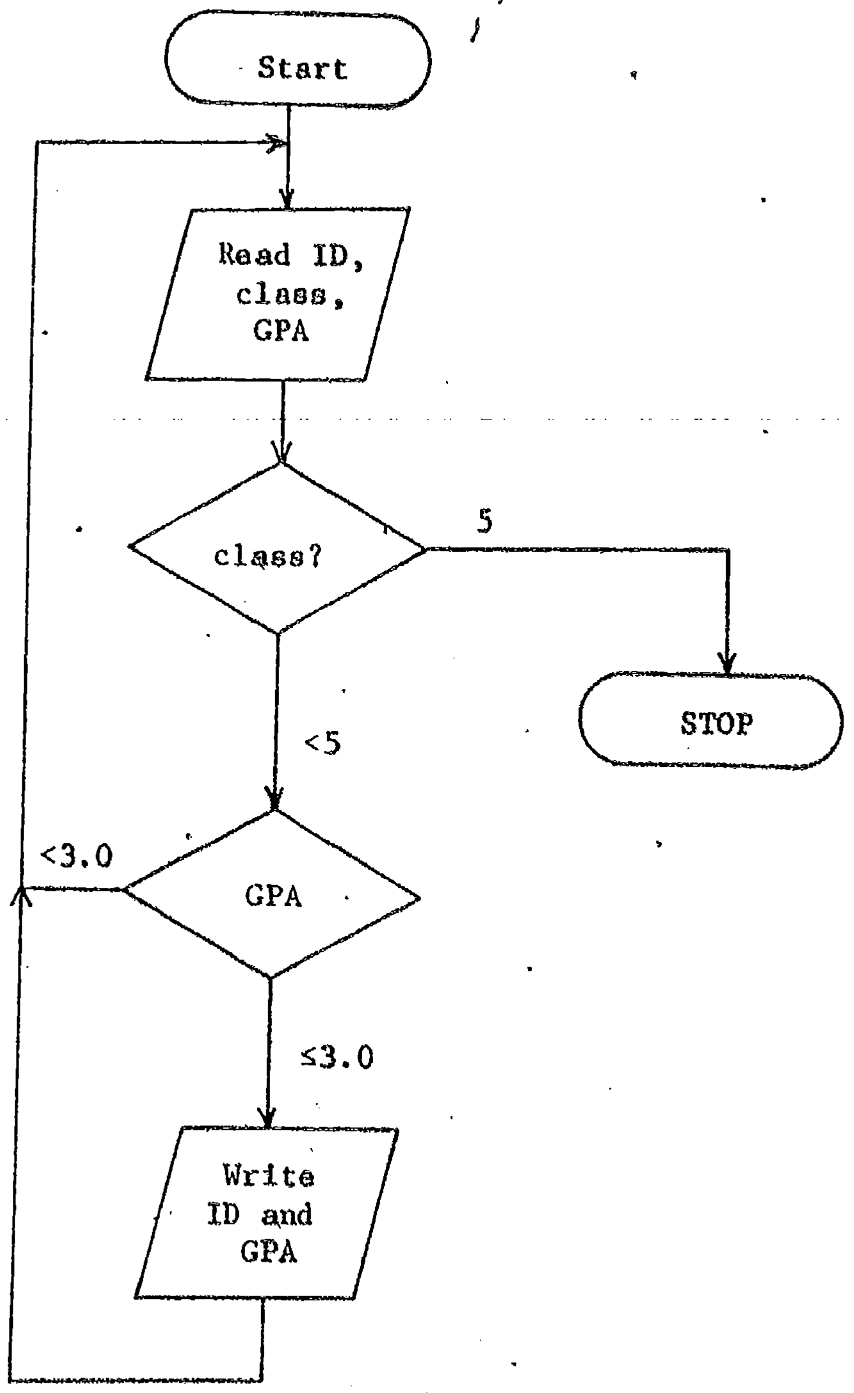


Figure 10.6. Flowchart for Dean's List problem.

Notice in the computed GO TO that execution is terminated whenever the classification is 5, but that looping continues as long as the classification is 1, 2, 3, or 4.

The computed GO TO could, of course, be replaced with an IF statement. For example, the arithmetic IF

```
IF(KLASS-5)10,11,11
```

or the logical IF

```
IF(KLASS.GT.4)STOP
```

could be used.

There are some alternate ways of checking for the last card in this example. We could have used a blank card for the trailer card and checked for a classification of zero or for an ID number of zero. (Remember that blank numeric fields are read as zeros when the computer reads the data cards.) Another way would be to place a "nonsense" number, such as 9.0, in the field containing the grade point average and check for that.

The point of this example is simply this: looping may be accomplished more easily in some cases without the use of DO statements.

How do you decide whether to loop with a DO or to use some other means of looping? Here is a "simple rule of thumb" to help you decide:

If the loop makes use of a counter, then probably you should use a DO.

If the looping is controlled by some condition based on the input data or some condition set by the program, usually tested by some conditional transfer statement, then probably you should not use a DO.

This rule is certainly not absolute, however, and should be applied only as a kind of starting point.

SELF EVALUATION:

A. Refer to UNIT #10 ACTIVITIES TABLE, Activity 2.

B. Refer to UNIT #10 ACTIVITIES TABLE, Activity 3.

C. Write a program segment that will find the sum of the elements of a two-dimensional array containing N rows and M columns. (Answer on next page.)

ASSESSMENT TASKS: Please see your instructor. You will be given a problem for which you are to construct a Fortran program, making ~~use of~~ one or more DO loops, and run it on a computer.

WHAT NEXT? You may go to UNIT #11 or to UNIT #13.

Answers to problems in the text of UNIT #10

```
C  INDENTING THE "BODY" OF
C  THE RANGE OF A DO LOOP.
C
      DØ 17 I=1,N
        DØ 13 K=1,N
          SUM=0.
          DØ 15 J=1,N
            SUM=SUM+A(I,J)*B(J,K)
   15     CØNTINUE
          C(I,J)=SUM
   13   CØNTINUE
   17 CØNTINUE
```

```
C  SELF EVALUATION C.
C  ASSUME THAT N<11 AND THAT M<21.
C
      DIMENSIØN A(10,20)
C  SUMMING CAN BE DONE BY ROWS OR BY COLUMNS.
C  EXAMPLE OF SUMMING ACROSS ROWS.
      SUM=0.0
      DØ 10 I=1,N
        DØ 11 J=1,M
          SUM=SUM+A(I,J)
   11   CØNTINUE
   10 CØNTINUE
C
C  EXAMPLE OF SUMMING DOWN COLUMNS.
      SUM=0.0
      DØ 12 J=1,M
        DØ 13 I=1,N
          SUM=SUM+A(I,J)
   13   CØNTINUE
   12 CØNTINUE
```

UNIT #11 (COMSC)

TITLE: INPUT, OUTPUT, FORMATS

RATIONALE: Although the input/output discussions in UNIT #6 (COMSC) should provide the essentials for reading data and printing results on the line printer, there are additional features that provide simplifications or capabilities that are useful. As in UNIT #6 (COMSC), the card reader will be used for input and the line printer will be used for output.

When you complete this unit you are well on your way toward being able to use Fortran as an effective computational tool. Many of the programs which you develop in the remainder of this course should be useful to you throughout your college career. We feel that being able to understand and write computer programs is one of the most important skills that you can learn. It is almost of the same level of importance as the ability to communicate effectively both in oral and written form.

There is an appendix that summarizes FORMAT statements at the end of this unit.

OBJECTIVES: When you finish this unit, you will be able to construct Fortran IV READ and WRITE statements and their associated FORMAT statements to handle a variety of input/output situations.

PREREQUISITES: UNIT #10 (COMSC).

ACTIVITIES: You should make use of a variety of READ, WRITE, and FORMAT statements in the programs required by the other COMSC units. This will help you to become proficient in performing the tasks of reading and writing. It also will help you to organize and present results in a readable form. Furthermore, by inserting temporary WRITE statements at key points in your program, you should be able to debug the program much faster than by trusting to luck.

Format field descriptors covered in UNIT #6 (COMSC) were

F, H, I, X.

Additional format field descriptors covered in this unit are

A, D, E, G.

Other format field descriptors (L, T, Z) exist, but will not be covered in this IPI sequence. Also covered in this unit are input and output of array data.

NOTE: THE MOST SIGNIFICANT DIFFERENCES IN FORTRAN IMPLEMENTATIONS FOR VARIOUS MACHINES ARE MOST LIKELY TO OCCUR IN INPUT/OUTPUT PROVISIONS. In other words, FORMAT statements that work on one compiler may not necessarily work on another.

Recall that input/output statements are of the form

$$\left\{\begin{array}{l}\text{READ}\\ \text{WRITE}\end{array}\right\} \quad \text{(u,f) list}$$

where "u" is an I/O unit number, "f" is a format statement number, and "list" may be empty or contain a list of variable names. Each variable name in the list requires an associated field descriptor of type A, I, D, E, F, or G to be present in the FORMAT statement specified by "f." Literal, H, and X field descriptors are not associated with variable names in the list.

How are the field descriptors and the items in the I/O list coordinated? Each action of format control depends on information jointly provided by the next element of the input/output list, if one exists, and the next field descriptor obtained from the FORMAT statement. If there is an input/output list, at least one field descriptor other than 'literal', H, or X must exist. Stated more simply, if you tell the computer to write or read the value associated with a variable name, then you must provide a field descriptor for that value in the FORMAT statement. As an example, consider the statements written in symbolic form in the following example.

```
        READ(5,111)ε1,ε2,ε3,ε4,ε5,ε6
111     FORMAT(δ1,5X,δ2,δ3/δ4,4(10X,δ5))
        WRITE(6,113)λ1,λ2,λ3,λ4,λ5,λ6,λ7
113     FORMAT(1H0,4X,2HA=,δ1,//2(δ2,δ3))
```

Assume

ϵ_i represents a list element,

λ_i represents a list element,

δ_i represents a field descriptor.

Once the read statement is invoked, the joint action proceeds as follows:

action sequence	list element	field descriptor
1	ϵ_1	δ_1
2		5X (5 columns skipped)
3	ϵ_2	δ_2
4	ϵ_3	δ_3
5		/ (read next card)
6	ϵ_4	δ_4
7		10X (10 columns skipped)
8	ϵ_5	δ_5
9		10X
10	ϵ_6	δ_6
11	input list exhausted	

Similarly, when the write statement is invoked, the action proceeds as follows:

action sequence	list element	field descriptor
1		1H0 (double space before print)
2		4X (blanks placed in next 4 print positions)
3		2HA= (inserts A= in next two print positions)
4	λ_1	δ_1
5		// (double space before print)
6	λ_2	δ_2 (watch this; the left most character to be printed is a carriage control character and must be a blank)
7	λ_3	δ_3
8	λ_4	δ_2
9	λ_5	δ_3
10	λ_6	δ_2
11	λ_7	δ_3
12	list exhausted	

NOTE: A field descriptor must agree with the data type of its associated variable in the I/O list. For example, the descriptor I must be used with an integer variable name.

To be more specific, consider the statements

```
121   READ(5,121) W,Y,K,J,A
      FORMAT(F10.3,4X,F6.4,2(3X,I5)/F10.6,F6.3,I4)
```

The corresponding action sequence follows:

action sequence	list element	field descriptor
1	W	F10.3
2		4X
3	Y	F6.4
4		3X
5	K	I5
6		3X
7	J	I5
8		/ (advance to next data card)
9	A	F10.6
10	list exhausted	(the rest of the field descriptors are ignored)

NOTE: The field descriptors

2(3X,I5)

may also be written as

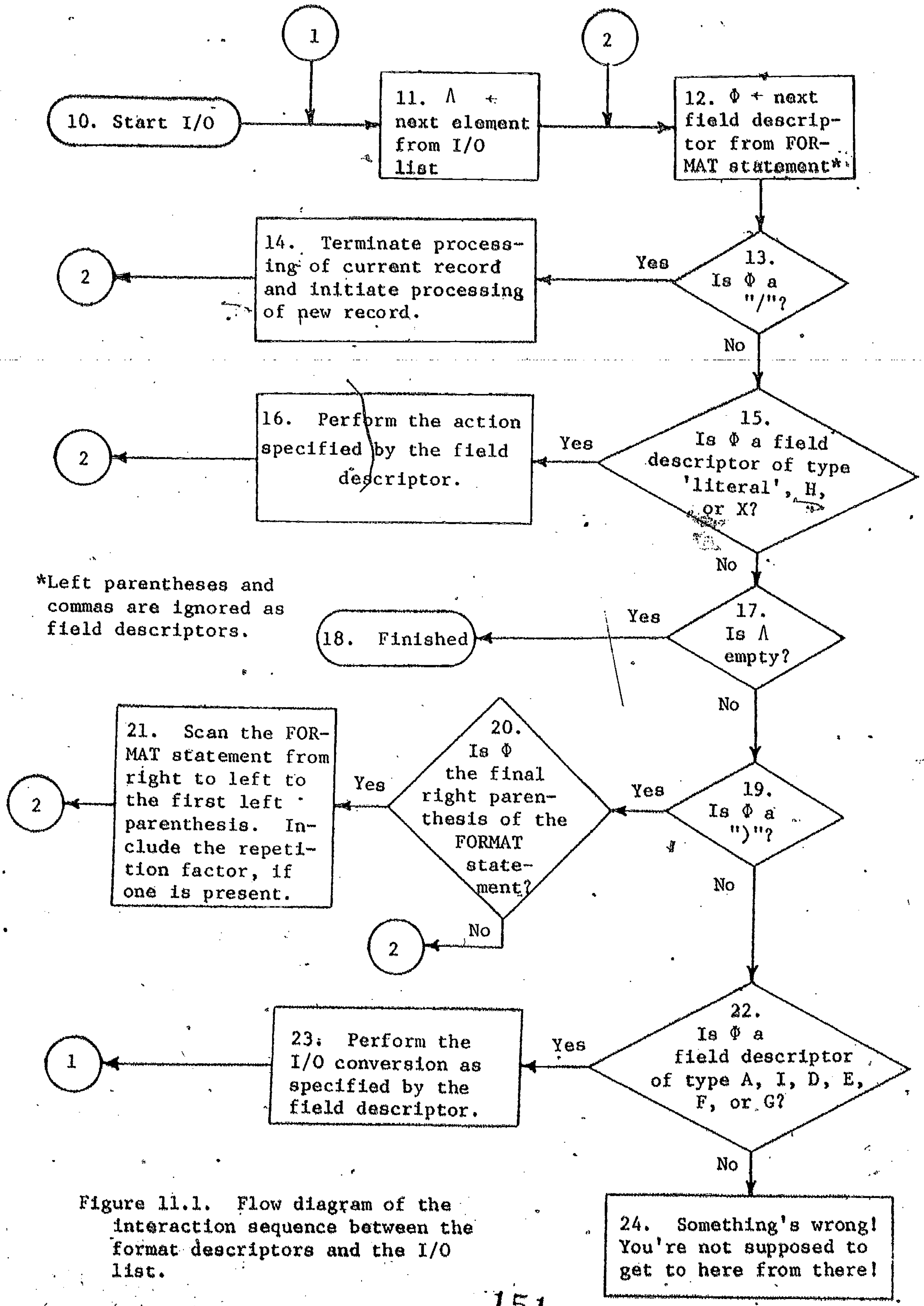
3X,I5,3X,I5

Complete the action sequence for the following Fortran statements.

```
      WRITE(6,196)I,J,A,X,P
  196 FØRMAT(1H1,39X,16HQUARTERLY REPØRT/
     1        13H0INCREMENT = ,I5,3X,8HINDEX = ,I4,
     2        11HAMT, EXT. = ,2F10.2,5X,9HPRØFIT = ,
     3        F12.2)
```

action sequence	list element	field descriptor
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

A simplified flow diagram of the interaction sequence between the format descriptors and the I/O list appears in Figure 11.1. Refer to the flow diagram and trace through the action sequences for some I/O statements. Notice that the blocks in the diagram are num-



1
2
10. Start I/O
11. Λ ← next element from I/O list
12. Φ ← next field descriptor from FORMAT statement*
13. Is Φ a "/"?
Yes
No
14. Terminate processing of current record and initiate processing of new record.
2
15. Is Φ a field descriptor of type 'literal', H, or X?
Yes
No
16. Perform the action specified by the field descriptor.
2
17. Is Λ empty?
Yes
No
18. Finished
19. Is Φ a ")"?
Yes
No
20. Is Φ the final right parenthesis of the FORMAT statement?
Yes
No
2
21. Scan the FORMAT statement from right to left to the first left parenthesis. Include the repetition factor, if one is present.
2
22. Is Φ a field descriptor of type A, I, D, E, F, or G?
Yes
No
23. Perform the I/O conversion as specified by the field descriptor.
1
24. Something's wrong! You're not supposed to get to here from there!
*Left parentheses and commas are ignored as field descriptors.

Figure 11.1. Flow diagram of the interaction sequence between the format descriptors and the I/O list.

bered, so you can indicate an action sequence with a series of numbers. Here's an example:

```
   5  WRITE(6,5)A
      FORMAT(1H0,F5.1)
```

The action sequence is

```
10      12      11
11      13      12
12      15      13
13      17      15
15      19      17
16      22      18
        23
```

Here's another example:

```
   6  WRITE(6,6) A,B
      FORMAT('1TABLE 5'/(1H0,F5.1))
```

The action sequence is

```
10   12   12   12   11   12   12   12   11
11   13   13   13   12   13   13   13   12
12   14   15   15   13   15   15   15   13
13        16   17   15   17   16   17   15
15             19   17   19        19   17
16             22   19   20        22   18
               23   20   21        23
```

Now, you try some. Complete the action sequence for each of the following three examples, using the flow diagram in Figure 11.1.

1.

```
      WRITE(6,7) A
    7 FORMAT(1H0,F5.1,1H%)
```

2.

```
      WRITE(6,8) A
    8 FORMAT(1H0,F5.1//'0THAT IS ALL.')
```

3.

```
        READ(5,9)A,B,C,D
  9     FORMAT(2(5X,F5.1),F11.4)
```

Correct action sequences are shown below:

1.	10	12	11	12
	11	13	12	13
	12	15	13	15
	13	17	15	17
	15	19	16	18
	16	22		
		23		

2.	10	12	11	12	12	12
	11	13	12	13	13	13
	12	15	13	14	15	15
	13	17	14		16	17
	15	19				18
	16	22				
		23				

3.	10	12	11	12	11	11	12	12	11	12
	11	13	12	13	12	12	13	13	12	13
	12	15	13	15	13	13	15	15	13	15
	13	17	15	17	15	15	16	17	15	17
	15	19	16	19	17	17		19	16	18
	16	22		22	19	19		22		
		23		23	22	20		23		
					23	21				

NOTE: In this case, the 2(5X,F5.1) in the FORMAT was treated simply as 5X,F5.1,5X,F5.1, thus deleting the inner parentheses.

Array input/output is handled in much the same way as nonarray input/output. It is possible to reduce the number of entries in the input/output list by using "implied DO's"; however, this can be time consuming so that simplifications in writing the I/O statement must be weighed against the accompanying increase in computer execution time. An example of a list with an implied DO follows.

```
READ(5,111)K,L,((A(I,J),I=1,K,2),M(J,3),J=1,L)
```

In this example the elements updated by the READ are

```
K,L, A(1,1),A(3,1),A(5,1), ..., A(k,1),M(1,3),
     A(1,2),A(3,2),A(5,2), ..., A(k,2),M(2,3),
                  . . .
     A(1,L),A(3,L),A(5,L), ..., A(k,L),M(L,3)
```

where k is either K or K-1, whichever is odd. The FORMAT statement associated with this READ statement must be such that the field descriptors agree with the data type of corresponding variables in the I/O list.

NOTE: Index variables for implied DO's (I & J in the example above) are modified just as index variables of DO statements. Consequently the value appearing in an implied DO index variable will be different after the I/O statement is executed from what it was before the statement was executed.

QUESTION: What is an appropriate FORMAT statement for the above READ statement if K, L, M are of integer type and A is real type? Assume that K=5 and L=3.

One correct FORMAT using arbitrary field widths is shown below:

```
  111 || FORMAT(2I5/(3F5.1,I6))
```

In this FORMAT, the values of K and L must be punched on the first data card; and three values of the array A and one value of the array M are placed on each of the subsequent data cards.

Refer to Activity 1, UNIT #11 ACTIVITIES TABLE.

SELF EVALUATION:

A. Write a program segment that will read in the array punched on these data cards, using only one READ statement and one FORMAT statement. (See pages 11.16 and 11.17 for solutions.)

```
         11111111112222222222333333333344444444445555555555666666666677777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890
20
  0.3  1.2 -3.6 51.3  1.1-81.2 20.0 49.9 -5.6 21.3 78.0-31.1 30.5-19.2  4.7 16.0
-17.0 18.0  0.0-20.0
```

B. Write a program segment that will write out the array described in 1 above five elements per line on every other line (double-spaced).

C. Write a program segment that will write out the array in 1 above one element per line single-spaced in reverse order in which they were read in.

D. The students in a particular course took an examination on which the scores were based on 100 points. Write a Fortran program that can handle up to 1000 scores and will count the number of students who made scores greater than the average. (The average is found by dividing the total of the scores by the number of scores.)

The scores are punched in 16 fields of five columns each with decimal points punched. The first data card contains only the number of students who took the test, right-justified in columns 1-5. Use only one READ statement in the program.

Use a DATA initialization statement for initializing running sums and counters.

The output is to be placed at the top of a new page, must look exactly like that shown below, and must be produced with only one WRITE statement.

(Small x's indicate numeric fields.)

```
          1111111111222222222233333333334444444444 5
 1234567890123456789012345678901234567890123456789 0
AVERAGE SCORE IS xxx.x

NUMBER OF SCORES ABOVE AVERAGE IS xxxx

SCORES
 xxx.
 xxx.
 xxx.
 etc.
```

E. Refer to Activity 2, UNIT #11 ACTIVITIES TABLE.

ASSESSMENT TASK: You will be asked to construct, debug, and document a Fortran program. Contact the instructor when you feel ready.

WHAT NEXT? If you haven't completed UNIT #13 (COMSC), you should do so. You should start UNIT #14 (COMSC) as soon as you feel that you have a good grasp of UNITS #11 and 13. You may want to try UNIT #12 or 16 (COMSC) concurrently with UNIT #14.

When you have completed UNITS #11 and #13, you may elect to stop, in which case you are not eligible for a grade higher than "B." If you wish to try for an "A," then you must complete the remaining units. Discuss this with your instructor if you wish.

Solutions to problems:

A.

```
    |READ(5,1)N,(A(I),I=1,N)
  1 |FORMAT(I2/(16F5.1))
```

B.

```
    |WRITE(6,2)(A(I),I=1,N)
  2 |FORMAT(1H0,5F6.1)
```

C.

```
    |DO 3 I=1,N
    |   L=N-I+1
    |   WRITE(6,5)A(L)
  3 |   CONTINUE
  5 |   FORMAT(1H ,F6.1)
```

On some computer systems the following program segment will work, but notice that the form of the subscript is nonstandard:

```
    |DO 3 I=1,N
    |   WRITE(6,5)A(N-I+1)
  3 |   CONTINUE
```

D.

```
C     K IS THE COUNTER FOR THE SCORES ABOVE THE AVERAGE.
C     SUM IS THE CUMULATIVE SUM OF THE SCORES.
C     S IS THE ARRAY OF SCORES.
      DIMENSION S(1000)
      DATA SUM,K/0.0,0/
C     INPUT SECTION
      READ(5,1)N,(S(J),J=1,N)
    1 FORMAT(I5/(16F5.1))
C     FIND THE AVERAGE.
      DO 2 J=1,N
        SUM=SUM+S(J)
    2   CONTINUE
      AVG=SUM/N
C     COUNT THE SCORES GREATER THAN THE AVERAGE.
      DO 3 J=1,N
        IF(S(J)-AVG)3,3,10
   10     K=K+1
    3   CONTINUE
C     OUTPUT SECTION
      WRITE(6,4)AVG,K,(S(J),J=1,N)
    4 FORMAT(18H1AVERAGE SCORE IS ,F5.1/
     1       35H0NUMBER OF SCORES ABOVE AVERAGE IS ,I4/
     2        7H0SCORES/(2X,F4.0))
      CALL EXIT
      END
```

UNIT #11 (COMSC) APPENDIX

FORMATS

GENERAL FORM:	nnnnn FORMAT ($\delta_1, \delta_2, \ldots, \delta_k$)
where	nnnnn is a statement number (1 through 4 or 5 digits)
	$\delta_1, \ldots, \delta_k$ are field descriptors

THE FIELD DESCRIPTORS AND DATA TYPES TO WHICH THEY APPLY ARE:

rAw	character data fields
rIw	integer data fields
prDw.d	real data fields
prEw.d	real data fields
prFw.d	real data fields
prGw.s	integer, real, logical or complex data fields
'literal'	transmits literal data (output)
wH	transmits literal data (output)
wX	field skip on input or insert blanks on output
r(...)	group format specification

where

d is an unsigned integer constant specifying the number of decimal places to the right of the decimal point, i.e., the fractional portion. The d must be specified in D, E, and F field descriptors even if it is zero: Furthermore w must be greater than d. For E type format $w \geq d+7$. In this case the field width w must include in addition to d, a position for a sign, a digit, a period, an E, an exponent sign, and a two digit exponent such as

±0.xxxxxxE±ee

p is optional and represents a scale factor designator of the form nP where n is an unsigned or negatively signed integer constant.

r is optional and is an unsigned integer constant used to denote the number of times the format field descriptor is to be used. If r is omitted, the field descriptor is used only once.

s is an unsigned integer constant specifying the number of significant digits.

w is an unsigned nonzero integer constant that specifies the width of the field.

UNIT #12 (COMSC)

TITLE: ARITHMETIC CONCEPTS .

RATIONALE: It is safe to say that more arithmetic is done in an hour on the installed computers today than has been done by all of mankind since man began to count. Yet the type of arithmetic that we study in school satisfies quite different properties from the type of arithmetic performed by computers. The results of a computer computation are usually close to those of a hand computation; however, in a specific case in which an author reported, "The numerical integration in this study took about one and one half years with twenty working hours every week with a considerable amount of work and endurances," the resulting hand calculation turned out to be totally incorrect! In this unit different types of arithmetic computation will be investigated.

OBJECTIVES: Determine the number of significant digits in an expression, given the number of significant digits in each element of the expression.

Describe integer arithmetic, floating point arithmetic, and fixed point arithmetic as implemented on digital computers.

Describe the associative, commutative, and distributive laws of arithmetic and their relationships to arithmetic performed on digital computers.

Describe errors in arithmetic due to rounding, truncation, loss of significant digits, conversion from decimal to binary, and conversion from binary to decimal.

PREREQUISITES: UNITS 11 and 13, Math units on associative, commutative, and distributive laws of arithmetic.

ACTIVITIES: Read the following material and perform the tasks indicated. Answers are at the end of the unit.

Accuracy is measured in terms of the number of significant digits which a number contains. Any one of the digits 1, 2, 3, 4, 5, 6, 7, 8, 9 is a significant digit. The digit 0 is sometimes, but not always significant. Three cases are possible for the digit 0.

(a) Zeros are to the left of all other non-zero digits. Here the zero is used to indicate the decimal point and is not significant. For example, in .00123, the leftmost two zeros are not significant; rather they serve as position holders for the decimal point.

(b) Zeros are between significant digits. In this case the zeros are considered to be significant. In the number 1002, the two zeros are significant.

(c) Zeros lie to the right of non-zero digits of a number as, for example, in the number 123400. The zeros may or may not be significant. The recommended notation if the digits are not significant is to write the number as

$$1234 \times 10^2.$$

If the first zero were significant this should be written as

$$12340 \times 10^1.$$

The number of significant digits in each of the following numbers is four: 1234, 1002, .01234, 3210×10^3, 1000×10^4.

How many significant digits are in each of the following numbers?

(12.1)	a. 1.234	______	f. 123400×10^6	______
	b. 1.005	______	g. 1000×10^{-3}	______
	c. 60200	______	h. 0	______
	d. .000012	______		
	e. .000100	______		

A number is said to be correct to n significant digits if its value is correct to within one half unit (of the given base) in the least significant position. For example, if the number 1234 is correct to four significant digits, it is understood that the number lies between 1233.5 and 1234.5. An alternate expression for this is 1234 ± 0.5. If 780×10^5 is correct to three significant digits, it lies between 77950000 and 78050000.

Frequently, numbers which are known to many significant digits must be reduced to a suitable length for computational purposes. A process called rounding is used to reduce such a number to n significant digits. On a computer this operation is performed by adding (subtracting) one half of the base in the position (n+1) if the number is positive (negative) and then retaining the resultant n significant digits. The following examples illustrate this procedure for base 10 numbers rounded to 4 significant digits.

```
  1 2 . 3 4 5 6
+ 0 0 . 0 0 5
  1 2 . 3 5|0 6        ≡  12.35 correct to four significant
                                digits


  . 9 8 7 6 4
  . 0 0 0 0 5
  . 9 8 7 6|9          ≡  .9876 correct to four significant
                                digits
```

```
- 1 2 . 3 4 5 6
- 0 0 . 0 0 5
- 1 2 . 3 5|0 6      ≡  -12.35 correct to four signif-
                              icant digits

- . 9 8 7 6 4
- . 0 0 0 0 5
- . 9 8 7 6|9        ≡  -.9876 correct to four signif-
                              icant digits
```

This procedure produces slightly different distribution of results relative to the manual procedure of rounding. The manual procedure does not round if the digit in the nth significant position is even and the digit in the position (n+1) is a 5.

Examples of the manual procedure are

```
12.345     ≡  12.34 correct to four significant
                       digits

12.355     ≡  12.36 correct to four significant
                       digits
```

Round by both the manual and computer based methods each of the following numbers to 5 significant digits.

			manual	computer based
(12.2)	a.	3.1415926	________	________
	b.	2.71828	________	________
	c.	4679.25	________	________
	d.	4679.35	________	________
	e.	-4679.25	________	________
	f.	-4679.35	________	________

To determine the number of significant digits resulting from the evaluation of an arithmetic expression, given the number of significant digits in each element of the expression, it is necessary to establish some rules governing significant digit arithmetic. These are given under the headings addition, subtraction, multiplication, and division below.

<u>Addition</u>. When adding two positive or two negative numbers, there is no loss of significance when unrounded operands are used. In some cases the result may have one more significant digit than either of the operands. For example, suppose 1536.2 and 7428.9 were rounded to 1536 and 7429 respectively. The additions of the rounded and unrounded numbers are

rounded	unrounded
4536	4536.2
7429	7428.9
11965	11965.1

both of which are correct to 5 significant digits. The following case illustrates a loss of significance when rounded operands are used:

rounded	unrounded
1232	1231.5
6746	6745.5
7978	7977.0

since the result is correct to three significant digits instead of four significant digits which each operand contained.

How frequently is such a loss likely to occur?

(12.3) ____________________

When one of the operands contains fewer significant digits than the other, the operand with the greater number of places to the right of the decimal must be rounded to conform to the other operand.

For example, in order to add the two numbers 12.345 and 2345.6, it is meaningless to retain more than one digit to the right of the decimal point because the second number is accurate only one place to the right of the decimal point. The sum is given by

$$\begin{array}{r} 12.3 \\ \underline{2345.6} \\ 2357.9 \end{array}$$

Subtraction. When a subtraction occurs (by either adding a positive number to a negative number or by subtracting a positive number from a positive number), a complete loss of significance may occur. Usually each case must be considered independently. As an example consider the numbers 12345.6 and 12345.4 which are rounded to 12346 and 12345. The results of rounded and unrounded subtraction are

```
   rounded           unrounded

    12346             12345.6
  - 12345           - 12345.4
    00001             00000.2
```

Although the two are rounded correctly to 5 significant digits, the result has no significant digits, i.e., the 1 is off by a full unit. It is more likely that a loss of significance occurs, but not a complete loss of significance. The best strategy is to try to arrange the computation so that subtractions do not occur. As discussed earlier in UNIT #11 (COMSC), one real root of the quadratic equation should be constructed as

$$x_1 = \frac{-b + \sqrt{b^2 - 4\cdot a\cdot c}}{2a} \qquad \text{if } -b \text{ is positive}$$

or as

$$x_1 = \frac{-b - \sqrt{b^2 - 4\cdot a\cdot c}}{2a} \qquad \text{if } -b \text{ is negative.}$$

The other root is determined by recognizing that

$$x_1 x_2 = c/a$$

and hence

$$x_2 = \frac{c}{a\cdot x_1} \quad .$$

Thus there is no subtraction in determining the root. If both a and c are positive, a loss of significance definitely will occur in the computation of the discriminant. This loss may be reduced if $b^2 - 4ac$ is computed to as high a number of significant digits as possible. This is called double or multiple precision and will be discussed subsequently. Double precision is another strategy for reducing the loss of significance.

Multiplication. Up to two significant digits may be lost in a multiplication. The product of 921.2 and 102.4 each rounded to three significant digits compared with the unrounded numbers provides an example.

```
   rounded          unrounded

     921              921.2
     102              102.4
    1842              36848
   9210              18414
   93942            92120
                    94329.88
```

Although the multiplier and multiplicand are correct to three significant digits, the product differs by 6 in the third position from the left or by 0.6 in the second position from the left. That is, there is only one significant digit in the product, even though both the multiplier and multiplicand were correct to three significant digits.

Division. Up to two significant digits may be lost in a division. Consider the quotient of 1763.4 by 1761.6 with both dividend and divisor rounded to four significant digits each. How many digits are significant in the rounded quotient?

(12.4) ____________

How many significant digits result if each operand in the expression

R = (A-B+C)*D*E

is rounded to 3 significant digits if

A = 432.7	D = 123.4
B = 432.4	E = 987.6
C = 341.62	

(12.5) $R_{rounded}$ = ________ $R_{unrounded}$ = ________

number of significant digits ________

12.12

Integer arithmetic as implemented on digital computers assumes that the radix (decimal, binary, octal, hexadecimal, etc.) point is on the right of a fixed size number. The fixed number is called a <u>word</u> and might be 10 decimal digits or 32 binary digits or 36 binary digits or some other size. Addition and subtraction are performed using one word. Multiplication and division require two words, but are subsequently reduced to one word. Consider a decimal machine in which a word consists of two digits.

$$\begin{array}{rr} & \boxed{2}\boxed{3} \\ + & \boxed{4}\boxed{7} \\ \hline & \boxed{7}\boxed{0} \end{array}$$

The sum or difference is generated in one word.

$$\begin{array}{rr} & \boxed{2}\boxed{3} \\ \times & \boxed{4}\boxed{7} \\ \hline & \boxed{1}\boxed{0}\boxed{8}\boxed{1} \end{array}$$

The product requires two words.

$$\boxed{0}\boxed{1}\boxed{6}\boxed{9} \div \boxed{1}\boxed{2} \rightarrow \underbrace{\boxed{1}\boxed{4}}_{\text{quotient}}\,\underbrace{\boxed{0}\boxed{1}}_{\text{remainder}}$$

The dividend requires two words, the divisor requires one word. The quotient is in the leftmost word, the remainder is in the rightmost word after division.

The result after division is not rounded. The remainder occurs in one of the words and requires special manipulation if it is to be utilized further in the representation of the quotient. Instructions on the computer make it easy to select the rightmost word or the left-most word of such a pair of words.

In floating point arithmetic, part of the word refers to the exponent (or power of the base) and the rest of the word consists of significant digits. Suppose our computer has 4-digit decimal words with a plus or minus sign. Suppose floating point numbers have the form

±	e	d_1	d_2	d_3

where e stands for the exponent and the numbers in d_1, d_2, and d_3 make up the <u>mantissa</u> or the actual digits of the number. Exponents may range between 0 and 9 in a one digit field. If the number 5 is used to represent an exponent of 0, a one digit field can be used to represent exponents between -5 and 4. Such a representation is termed an 'excess 5' representation. It allows positive numbers to represent negative exponents. The location of the decimal point relative to the digits ddd (the subscript has been dropped temporarily) is given by the following table.

e	exponent	decimal point
9	4	ddd0.
8	3	ddd.
7	2	dd.d
6	1	d.dd
5	0	.ddd
4	-1	.0ddd
3	-2	.00ddd
2	-3	.000ddd
1	-4	.0000ddd
0	-5	.00000ddd

We require that after all floating point computations d_1 is non-zero, unless e, d_1, d_2, d_3 are all zero.

The exponent e is a power of 10. To multiply two numbers, one multiplies the mantissas, and subtracts 5 from the sum of the two exponents, perhaps making a correction to force d_1 of the result to be non-zero. For example, the computation of 3 × 2 in this floating point format is carried out as

$$\begin{array}{|c|c|c|c|}\hline 6&3&0&0\\\hline\end{array} \times \begin{array}{|c|c|c|c|}\hline 6&2&0&0\\\hline\end{array} \rightarrow \overset{e\,=\,6+6-5}{\begin{array}{|c|c|c|c|}\hline 7&0&6&0\\\hline\end{array}} \rightarrow \begin{array}{|c|c|c|c|}\hline 6&6&0&0\\\hline\end{array}$$

In this example, d_1 is zero; consequently, an automatic left shift of 1 occurs. A left shift of 1 is equivalent to multiplying by 10; thus, the exponent must be reduced by 1 to retain the proper value.

The computation of 8 × 7 does not require a fixup shift to place a non-zero digit into the d_1 position.

$$\begin{array}{|c|c|c|c|}\hline 6&8&0&0\\\hline\end{array} \times \begin{array}{|c|c|c|c|}\hline 6&7&0&0\\\hline\end{array} \rightarrow \begin{array}{|c|c|c|c|}\hline 7&5&6&0\\\hline\end{array}$$

Since two three digit operands can generate a six digit product, either two or three of the six digits are lost in a single precision multiply. No rounding occurs. In a double precision multiply, all six digits would be retained.

Write each of the following numbers as a floating point number in the format just described.

(12.6) a. 4.67 ________ d. -36.2 ________

b. .000987 ________ e. .02 ________

c. 104000 ________ f. -.0000763 ________

Indicate what the results would be in single precision floating point format for

(12.7) a. 4.67 × (-36.2) ________

b. 104000 × (-.0000763) ________

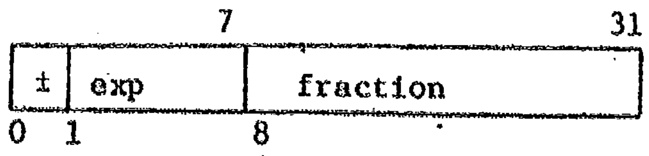
c. -36.2 + 4.67 ________

d. .02 + .000987 ________

e. 1040.00 + .02 ________

f. .000987 × (-.0000763) ________

On the 360/65, floating point arithmetic is used for arithmetic of type REAL. Two formats exist, single precision and double precision. A single precision floating point number is 32 bits in length and consists of a sign bit, 7 bits for the exponent, and 24 bits for the fraction.



Single precision floating point representation

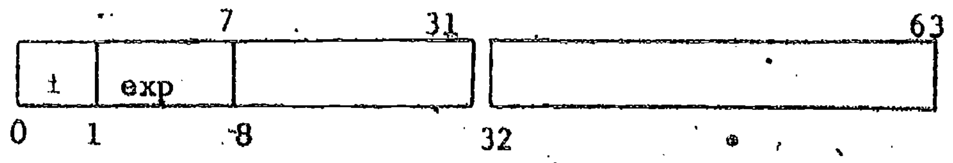
The fraction is in binary; the exponent represents powers of 16. A bias of 64_{10} serves the same purpose as the bias of 5 in the discussions above. "Excess 64" is another designation for the bias of 64. The binary representation of 64 is

$$1000000_2 \equiv 64_{10} \equiv 40_{16} .$$

An exponent of 65_{10}

$$1000001_2 \equiv 65_{10} \equiv 41_{16}$$

indicates that the fractional part is to be multiplied by 16^1. An exponent of 62 indicates that the fractional part is to be multiplied by 16^{-2}, etc. Double precision floating point numbers on the 360/65 use two words with the fraction part being 56 bits in length.



Double precision floating point representation

Although it is not difficult, we shall forego calculating in floating point arithmetic for the 360.

In mathematics a great deal of time is spent discussing associative, commutative, and distributive properties of arithmetic. To remind you of these properties we list them below.

Associative property (grouping property)

Addition	$(a+b)+c = a+(b+c)$
Multiplication	$(a\cdot b)\cdot c = a\cdot(b\cdot c)$

Commutative property (ordering property)

Addition	$a+b = b+a$
Multiplication	$a\cdot b = b\cdot a$

Distributive property

$$(a+b)\cdot c = a\cdot c + b\cdot c$$

$$a\cdot(b+c) = a\cdot b + a\cdot c$$

On a digital computer, the commutative property holds (any truncation that occurs will occur independent of the order of the operands); however, the associative and distributive properties need not hold. The three digit floating point representation used earlier illustrates this well.

$\pm$	e	$d_1d_2d_3$

Single precision floating point format

If the values a, b, and c were

$$a = 5670.$$

$$b = -5670.$$

$$c = .000987$$

then

$$(a+b)+c = .000987 \equiv \boxed{+}\boxed{2}\boxed{9\ 8\ 7}$$

and

$$a+(b+c) = 000 \equiv \boxed{+}\boxed{0}\boxed{0\ 0\ 0}$$

Thus

$$(a+b)+c \neq a+(b+c).$$

Determine numbers a, b, and c for which

$$(a+b)\cdot c \neq a\cdot c + b\cdot c$$

if the arithmetic is done in our digit floating point representation.

(12.8) a. a = ______________ d. $(a+b)\cdot c$ = ______________

b. b = ______________ e. $a\cdot c + b\cdot c$ = ______________

c. c = ______________

Determine numbers a, b, and c for which

$$a\cdot(b\cdot c) \neq (a\cdot b)\cdot c$$

(12.9) a.	a = __________	d.	$a\cdot(b\cdot c)$ = __________
b.	b = __________	e.	$(a\cdot b)\cdot c$ = __________
c.	c = __________		

if the arithmetic is done in the 3 digit floating point representation.

NOTE: 99.999% OF THE ARITHMETIC DONE TODAY DOES NOT SATISFY THE ASSOCIATIVE OR DISTRIBUTIVE PROPERTIES.

It should be obvious from these examples that the order of operands may produce some error in a computation.

Before you get the idea that things are really bad, the computers are almost associative (aa) and almost distributive (ad) enough so that reasonable results occur. It is unlikely that computers ever will be associative or distributive. (Why?) Consequently, in order to establish a degree of harmony with mathematics an aa-ad system must be defined mathematically. Who knows, you just might get involved in such a project.

Minor numerical discrepancies may take place during input/output. Numbers represented in base 10 must be converted to binary before computations occur and results must be converted from binary to decimal for printing. Exact fractional representations in decimal do not necessarily have exact fractional representations in binary; consequently, an error occurs when a converted value is truncated to the word size of the machine. For example,

$$(.1)_{10} = (.0001000100010001000100001 \ldots)_2$$

That is, 1/10 does not have a terminating binary fraction expansion and hence has to be truncated to the number of bits in a word. The fraction 1/2, on the other hand, has an exact binary representation, namely, $(.1)_2$. The fraction 1/3 has both a non-terminating decimal expansion and a non-terminating binary expansion. Errors which occur because of the requirement for different bases are usually not significant. Infrequently special programming may be required to avoid a truncation problem in the conversion from one base to another - but only infrequently.

Conclusion. In the preceding discussion we have avoided actual details of how various computers handle arithmetic operations. Computers may differ markedly in this regard. The same program run on different computers may possibly produce slightly different numeric results, especially if several significant figures are required in the results.

The exercises in this unit are intended to demonstrate to you that numbers are not necessarily what they appear to be and that whether numbers are or are not rounded before calculations generally affects results.

Generally computers do not round results of arithmetic operations, but rather truncate the results to the specified number of digits that the computer is designed to handle. An example of how this might affect results is found in a calculation that involves several division steps, in which the remainder is lost each time. The absolute value of the final result will be expected to be smaller than if rounding based on the magnitude of the remainder could have been performed. For example, suppose we wanted to evaluate

$$\frac{25.3}{3} + \frac{36.5}{7} + \frac{4.01}{2.6} + \frac{8.42}{3.7}$$

to three significant digits with truncation of digits beyond three. The result is

$$8.43 + 5.21 + 1.54 + 2.27 = 17.45 \rightarrow 17.4 .$$

Rounding gives

$$8.43 + 5.21 + 1.54 + 2.28 = 17.46 \rightarrow 17.5.$$

If the fractions containing the remainders are included, then the result is

$$17.47 \rightarrow 17.5.$$

You can see that the error accumulates as more operations are performed.

What conclusions can we draw? There are many possible; here are a few:

a. In general, you should organize calculations in a program so that the number of arithmetic operations that are performed is minimized.

b. Avoid situations in a program in which two quantities of similar magnitude are subtracted, resulting in loss of significant digits.

c. In general, you should arrange operations that are especially prone to loss of significant digits (subtraction and division) so that they occur last or near the end of a series of calculations.

d. Be sure that results given by the computer are reasonable. Check them by hand, if possible, with several sets of trial data. Additional care is also required because a program will not necessarily produce correct results with all data just because it does for some data.

SELF ASSESSMENT: If you have worked through the examples in the material, then you should have assessed your progress already. Check your answers with those given on the next page.

ASSESSMENT TASK: Please see your instructor. You will be required to apply the rules on rounding, the types of arithmetic, etc., as specified in the objectives.

WHAT NEXT? You should complete UNITS 14 and 16 before you can begin UNIT 17.

Solutions to problems in UNIT #12:

(12.1)	a.	4	f.	6
	b.	4	g.	4
	c.	5	h.	1
	d.	2		
	e.	3		

(12.2)		manual	computer based
	a.	3.1416	3.1416
	b.	2.7183	2.7183
	c.	4679.2	4679.3
	d.	4679.4	4679.4
	e.	-4679.2	-4679.3
	f.	-4679.4	-4679.4

(12.3) The digit in the position involved in the rounding must contain 5 for each operand. Assuming that the digits 0 to 9 are equally likely candidates for this digit position, there is 1 chance in 10 for each operand or 1 chance in 100 that such a loss will occur.

(12.4) 3

(12.5) $R_{rounded} = 41682732$ $R_{unrounded} = 41669735.6928$

number of significant digits = 2

(12.6)	a.	+6467	d.	-7362
	b.	+2987	e.	+4200
	c.	cannot be represented	f.	-1763

(12.7) a. -8169

b. cannot be represented

c. -7315

d. +4210

e. +9104

f. cannot be represented

(12.8) a. 1000.

b. -999.

c. 10.

d. 10

e. cannot be represented since a·c is too large

(12.9) a. 100.

b. 100.

c. .01

d. 100 · (1) = 100

e. cannot be represented since a·b is too large

UNIT #13 (COMSC)

TITLE: SUBPROGRAMS

RATIONALE: The writing and debugging of long programs may become quite frustrating. But, if the programs can be divided into shorter, simpler parts, then both the writing and debugging can be simplified. By using subprograms in Fortran, you can divide programs into shorter and simpler segments, write them, and debug them quite easily. This capability alone provides an important rationale for learning to use subprograms.

Another useful feature of subprograms is that they may be saved for future use with other programs. A subprogram is an independent unit which can be placed with any other program (or subprogram) and can be used with that program, provided it is properly called. This obviates your having to write more than one time a program that handles a particular type of task.

Subprograms are a very powerful part of Fortran and are used widely. Certainly your skills in the use of Fortran would be incomplete without an ability to write and use subprograms.

OBJECTIVE: At the end of this unit you will be able to construct any type of subprogram and its corresponding calling statement and to construct a program using any type of subprogram.

PREREQUISITES: UNIT #10 (COMSC).

SUBOBJECTIVE I: Identify the four elements of the concept of subprograms.

ACTIVITIES:

A. Here are six numbers: 2.5, 8.1, -5.2, 0.0, 4.3, -1.6. Find the average of these numbers and write it in this blank: ______________. In order to work the problem you had to go aside to the margin or to scratch paper for a work area. You, in effect, transferred the numbers to the work area, performed the calculations that were required, and then transferred the answer back to the blank line where you were told to place the answer. What you just did is analogous to what happens when a subprogram is used in Fortran. Let's investigate further what you did.

In the problem given to you above, "Here are six numbers. Find the average ... ." the key word is <u>average</u>. This word tells you which procedure to pull from the stored information in your mind. In other words, the word average elicits the procedure that says to add the numbers together and divide by the number of items. The procedure has a name. Fortran subprograms also have names. In order to use a subprogram, you must refer to it by its name in a calling statement, about which you will learn more shortly.

The name average tells you what computational procedure to follow, but you cannot find the average of some numbers unless you have those numbers. So the second thing you did was to transfer the numbers whose average you were to find to a work area. Similarly, a Fortran calling statement sends the data items specified in the calling statement to the subprogram (analogous to the work area). The items sent are called arguments.

Once the numbers were in the work area, then you could actually execute the "average" procedure and find the average. In the same way, a Fortran subprogram is executed, performing whatever task it is programmed to do.

Finally, when you had calculated the average in the work area, then you transferred the result to the place where you were instructed to put it. Then you proceeded through the material. Similarly, a Fortran subprogram sends the result (or results) to the main program and returns control to the main program, which allows the computer to proceed through the rest of the program.

Let's review the four elements of the concept of subprograms and the corresponding elements of solving the problem to find the average of six numbers.

Average Problem	Fortran Subprograms
1. The name "average" tells the student which of his mental procedures to use. Invoking the name calls for the procedure.	1. The name of the subprogram tells the computer which subprogram to use. Invoking the name calls the subprogram.
2. The student transfers the six numbers to the work area (scratch paper or margin) so that he will have numbers for calculating.	2. The data to be used in the subprogram are sent to the subprogram through the argument list.
3. The actual calculation of the average is performed.	3. The instructions in the subprogram are executed.
4. The results of the calculation are transferred back to the blank line on the page.	4. The results of the subprogram that has been executed are returned to the main program.

B. Refer to UNIT #13 ACTIVITIES TABLE, Activity 1.

SELF EVALUATION:

A. Identify the four elements of the concept of subprograms in the following analogy.

A student is working a problem and comes to a step in which he must take the square root of a number. He refers to a square root table in a mathematical handbook, writes the square root of the number on the paper, and proceeds with the problem.

B. The Fortran compiler has a "built-in" subprogram for calculating the square root; the name of the subprogram is SQRT. Suppose that the statement Y=SQRT(X) occurs in a Fortran program. Identify the four elements of the concept of subprograms when that statement is executed.

SUBOBJECTIVE II: Construct single statement functions and calling statements for them; construct complete a program using them.

ACTIVITIES:

C. Refer to UNIT #13 ACTIVITIES TABLE, Activity 2.

D. The quadratic formula for finding the roots of a quadratic equation of the form

$$ax^2 + bx + c = 0$$

is

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

A simple program for finding the roots of a quadratic equation is shown below; the program assumes that $b^2 - 4ac \geq 0$, that is, that the equation has two real roots.

```
      READ(5,1)A,B,C
 1    FORMAT(3F10.1)
      ROOT1=(-B+SQRT(B*B-4.0*A*C))/(2.0*A)
      ROOT2=(-B-SQRT(B*B-4.0*A*C))/(2.0*A)
      WRITE(6,2)A,B,C,ROOT1,ROOT2
 2    FORMAT(1H0,5F10.2)
      STOP
      END
```

This program can be improved, however, in the accuracy of the calculation. In UNIT #12 (COMSC) you learn something about the loss of significant digits in arithmetic operations. An example given in UNIT #11 illustrating this condition is found in the quadratic formula. If it turns out that the absolute value of b and the quantity $\sqrt{b^2 - 4ac}$ are of nearly equal magnitude and the two are subtracted, significant digits may be lost. In order to prevent the occurrence of this possibility, we can find the roots using a slightly different approach.

Use

$$r_1 = \frac{-b+\sqrt{b^2 - 4ac}}{2a} \qquad \text{when } -b>0$$

and use

$$r_1 = \frac{-b-\sqrt{b^2 - 4ac}}{2a} \qquad \text{when } -b<0.$$

The second root can be found from

$$r_2 = \frac{c}{ar_1}$$

since it is true that

$$r_1 r_2 = \frac{c}{a}.$$

(Try it, if you don't believe it!)

The program can be rewritten as follows:

```
       READ(5,1)A,B,C
  1    FORMAT(3F10.1)
       IF( -B)10,11,11
 10    ROOT1=(-B-SQRT(B*B-4.0*A*C))/(2.0*A)
       GO TO 12
 11    ROOT1=(-B+SQRT(B*B-4.0*A*C))/(2.0*A)
 12    ROOT2=C/(A*ROOT1)
       WRITE(6,2)A,B,C,ROOT1,ROOT2
  2    FORMAT(1H0,5F10.2)
       CALL EXIT
       END
```

NOTE: In the arithmetic IF statement, transfer can be either to statement number 10 or to statement number 11 when (-B) is zero. The results are the same either way.

Construct three single statement functions for finding the roots of a quadratic equation. (These will be for statements 10, 11, and 12 of the previous program.)

There are many ways to construct the functions correctly. One set of correct functions is shown below:

```
R1NEG(A,B,C)=(-B-SQRT(B*B-4.0*A*C))/(2.0*A)
R1POS(A,B,C)=(-B+SQRT(B*B-4.0*A*C))/(2.0*A)
R2(X,Z,R)=Z/(X*R)
```

(Notice that the names chosen for the functions must be real in this case, since the results are returned to the calling statements as the names of the subprograms. Real names return real results; integer names return integer results.)

E. Now construct three calling statements for the functions you have written.

Again there are many possible correct calling statements. For the statements above, the calling statements could be

```
ROOT1=R1NEG(A,B,C)
ROOT1=R1POS(A,B,C)
ROOT2=R2(A,C,ROOT1)
```

The function names must match exactly; the arguments in the corresponding calling and function statements must match in number (three in this case), type (real to real and integer to integer), and order (first argument in calling statement to first argument in function statement, second to second, etc.).

SELF EVALUATION:

C. Reconstruct the entire program that finds the roots of a quadratic equation, making use of the functions that you have constructed.

There are many ways to construct the program correctly. One solution is shown below.

```
      R1NEG(A,B,C)=(-B-SQRT(B*B)-4.0*A*C))/(2.0*A)
      R1POS(A,B,C)=(-B+SQRT(B*B)-4.0*A*C))/(2.0*A)
      R2(X,Z,R)=Z/(X*R)
      READ(5,1)A,B,C
    1 FORMAT(3F10.1)
      IF(-B)10,11,11
   10 ROOT1=R1NEG(A,B,C)
      GO TO 12
   11 ROOT1=R1POS(A,B,C)
   12 ROOT2=R2(A,C,ROOT1)
      WRITE(6,2)A,B,C,ROOT1,ROOT2
    2 FORMAT(1H0,5F10.2)
      CALL EXIT
      END
```

Notice that the arguments in the function statements are dummy arguments and need not be named the same as they are in the calling statement. Any valid names of the proper types may be used as arguments in the function statement -- provided that the same names are used on the right side of the =, of course.

D. Identify the four elements of the concept of subprograms for the program you have written.

SUBOBJECTIVE III: Construct a FUNCTION subprogram and calling statements; construct a main program that calls the subprogram.

ACTIVITIES:

F. Refer to UNIT #13 ACTIVITIES TABLE, Activity 3.

G. Below is a simple program for finding the maximum element in an array of real numbers.

```
      DIMENSION T(31)
      READ(5,1)N,(T(J),J=1,N)
    1 FORMAT(I3/(10F8.1))
      HIGH=T(1)
      DO 10 J=1,N
      IF(HIGH-T(J))11,10,10
   11 HIGH=T(J)
   10 CONTINUE
      WRITE(6,2)HIGH
    2 FORMAT(1H0,F5.2)
      STOP
      END
```

Take the algorithm for finding the maximum and construct a FUNCTION subprogram that will find the maximum.

One correct way to construct the subprogram is shown below.

```
      FUNCTION HIGH(T,N)
      DIMENSION T(31)
      HIGH=T(1)
      DO 10 J=1,N
      IF(HIGH-T(J))11,10,10
11    HIGH=T(J)
10    CONTINUE
      RETURN
      END
```

Notice that the name of the subprogram is real, since we have an array of real numbers and will want a real number returned to the main program. Notice also that the name of the subprogram appears (and must appear) at least once to the left of =; this is done so that the result obtained by the subprogram (the maximum element in this case) can be transferred back to the main program as the name of the subprogram in the calling statement.

It is also correct — and actually preferable — to write the DIMENSION statement DIMENSION T(N) (except on the IBM 1130). More will be said of this later.

H. Now construct a calling statement that could be used for calling the subprogram. Any correct calling statement will do.

Several examples of correct calling statements are shown below.

```
X=HIGH(X,J)
BIG=HIGH(T,N)
BIG=ABS(HIGH(T,N))+5.0
Y=HIGH(T,10)
B=HIGH(T,J-1)
```

I. Now construct a main program from the simple program given earlier that will call the subprogram that you wrote.

One correct way to construct the main program is shown below.

```
C  MAIN PROGRAM
      DIMENSION A(31)
      READ(5,1)N,(A(J),J=1,N)
    1 FORMAT(I3/(10F8.1))
      BIG=HIGH(A,N)
      WRITE(6,2)BIG
    2 FORMAT(1H0,F5.2)
      STOP
      END
```

SELF EVALUATION:

E. Refer to UNIT #13 ACTIVITIES TABLE, Activity 4.

SUBOBJECTIVE IV: Construct a SUBROUTINE subprogram and CALL statements; construct a main program that calls the subprogram.

ACTIVITIES:

J. Refer to UNIT #13 ACTIVITIES TABLE, Activity 5.

K. Construct a SUBROUTINE subprogram that will find the maximum element of an array. (You may use the same algorithm used in the previous section.)

One correct way to construct a SUBROUTINE subprogram to find the maximum element of an array is shown below.

```
      SUBROUTINE HIGH(T,N,BIG)
      DIMENSION T(31)
      BIG=T(1)
      DO 1 J=1,N
      IF(BIG-T(J))11,10,10
   11 BIG=T(J)
   10 CONTINUE
      RETURN
      END
```

L. Construct a CALL statement that may be used to call the SUBROUTINE subprogram.

Some examples of correct CALL statements are given below.

```
CALL HIGH(T,N,BIG)
CALL HIGH(A,M,X)
CALL HIGH(A,15,X)
```

M. Write a main program that will read an array of N elements from data cards, will call a SUBROUTINE subprogram to find the largest element in the array, and will write the largest element.

One solution to the problem follows:

```
      DIMENSION A(31)
      READ(5,1)N,(A(J),J=1,N)
    1 FORMAT(I3/(10F8.1))
      CALL HIGH(A,N,BIG)
      WRITE(6,2)BIG
    2 FORMAT(1H0,F10.2)
      STOP
      END
```

NOTE: The DIMENSION statement in the subprogram could be -- in general, should be -- written DIMENSION T(N), except on the IBM 1130. You will learn more about this in the next activity.

N. Refer to UNIT #13 ACTIVITIES TABLE, Activity 6.

SELF EVALUATION:

F. Refer to UNIT #13 ACTIVITIES TABLE, Activity 7.

ASSESSMENT TASK: Please see your instructor. You will be given a problem for which you are to construct a program and run it on a computer. Then you will be required to construct some short programs or program segments that make use of the various kinds of subprograms.

WHAT NEXT? If you haven't completed UNIT #11, you should do so. You should start UNIT #14 as soon as you feel that you have a good grasp of UNITS #11 and #13. You may want to try UNIT #12 or #16 concurrently with UNIT #14.

When you have completed UNITS #11 and #13, you may elect to stop, in which case you are not eligible for a grade higher than "B." If you wish to try for an "A," then you must complete the remaining units. Discuss this with your instructor if you wish.

UNIT #14 (COMSC)

TITLE: DEBUGGING

RATIONALE: If you haven't discovered by now that it is rather easy to make a programming error, then either you are an exceptional programmer or you haven't been doing the programs. (Which do you think is the more likely?). People involved in generating computer programs usually admit to making programming errors. They also have developed debugging techniques which help them to keep the errors to a minimum. In this unit you will be exposed to methods that others have found useful. You will also have an opportunity to help those behind you in the Fortran programming hierarchy with their programming and debugging problems. You should strive to prevent programming errors, but you also should be able to correct errors once they occur.

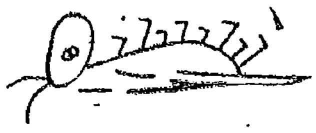
OBJECTIVES: To prevent programming and logical errors. To diagnose and correct errors, if they do occur

PREREQUISITES: UNITS #11 and 13 (COMSC).

At this point you should carry out the activities in the UNIT #14 (COMSC) ACTIVITIES TABLE.

Now you are on your way to becoming ~~an~~ MBE or

MASTER BUG ERADICATOR



Several levels of debugging exist. Some are easier to use than others; some are used under one system, but not another; and so on. What one looks for is an effective method for the particular system that is to be used.

Here are some general suggestions that you should use for preventing errors when you are constructing programs so that debugging will be simplified:

1. Plan your program carefully. Is the logic correct? Does it handle all possible cases? Draw a flowchart in enough detail to handle the sticky parts of the program before you actually start to write the program. If necessary, break up the program into subroutines. Include check point and diagnostic printouts in the planning stages, rather than after the

errors occur. Check your logic one more time.

2. You should be in the habit of writing an initial comments section at the beginning of the program. This comments section should include your name, date, problem name or title, a brief description of the problem, a list of variables, definitions and formats of both input and output variables, their function in the program and their dimensions, a description of special or exceptional conditions, and possible error conditions.

3. Place all of the type declaration statements at the beginning of the program immediately after the initial comments section. These declarations include COMMON, DIMENSION, INTEGER, REAL, DATA, etc.

4. Reserve a block of statement numbers for FORMAT statement numbers - for example, 200-299. Place all FORMAT statements immediately after the type declaration statements. In this way it is possible to check easily the existing FORMAT statements to determine whether it is necessary to add another FORMAT statement or whether a current one will do. As a rule you will be adding and deleting write statements during the debugging phase. A block of reserved FORMAT statement numbers will simplify the process.

5. Reserve another block of statement numbers as FORMAT statement numbers for temporary diagnostic type printouts. Place the FORMAT statement right next to the WRITE state-

ment. If both are punched on opposite cut cards from the rest of the program (on the back side of a FORTRAN card, for example), it is easy to remove both when you are finished with them.

6. Keep your READ, WRITE, and FORMAT statements as simple as possible. The more you try to include in these statements, the more likely you are to have trouble.

7. Use a CONTINUE statement as the last statement in the range of a DO. Use separate CONTINUE statements for the ranges of nested DO's. Use a CONTINUE statement as a reference point for GO TO statements. Then it is easy to insert or delete other statements without having to repunch the statement number and shuffle the cards.

8. Parenthesize your program by indenting parts of the program that are in the same logical block. In particular parenthesize the statements that are in the range of a DO statement.

Example

```
      DO 37 I = 1,N
         T = B*B-4.0*A*C
         DO 35 J = 1,M
            S(I,J) = T*X(I)
            R(I,J) = T*FLOAT(J)*Y(I,J)
 35      CONTINUE
 37   CONTINUE
```

9. Use variables rather than constants if there is a remote chance of having to make a change in the value of the con-

stant. For example, if you have several statements of the form

```
DO ~ I = 1,10
```

it is easier to make a single change of the form

```
IUP = 15
```

provided all the DO's originally were punched as

```
DO ~ I = 1,IUP
```

than to change each DO individually.

After you have tried the program on the computer and found that it contains errors, then you should begin by correcting compile errors, that is, errors which prevented the compiler from making sense out of what you wrote. Generally the compiler will list such errors. Some compilers (WATFIV, for example) list most of the error diagnostic messages immediately following the statements containing the errors; some (1130, for example) list all of the diagnostic messages at the end of the program listing. Often one error may produce several diagnostic messages, some of which may not appear to be related to the error. Usually, in such cases, however, at least one of the messages will be meaningful.

When there are no compile errors, the program may still contain execution errors, errors which make it impossible for the computer to perform the instructions given to it. Some compilers (WATFIV, for example) give diagnostic messages for execution errors; some (1130, for example) give no error diagnostics.

Here are some general suggestions for removing compile and execution errors:

10. Check the punched deck for mispunched characters, words spelled incorrectly, interchanged letters, etc. Also check to see that the correct columns are used -- statement number in 1 to 5, continuation in 6, Fortran statement in 7 to 72. Make sure that all comment cards have a "C" punched in column 1.
11. Check control cards to see that they are in the proper order and are correctly punched.
12. Use the error messages to see which lines cause compile errors. Remember that the error messages may not always diagnose the error clearly, but they do inform you which line is in error. Check the correct form of the statement with the textbook for commas, parentheses, mandatory use of integer numbers, etc.
13. Use the WATFIV diagnostic messages to find execution errors. The messages tell in which lines the errors are. Again be aware that the error message may not always diagnose the fault exactly. Check the form of the statement; check to be sure

that the type of the variables is consistent - real numbers read with F-type formats, characters compared with characters in an IF statement, etc. Check for keypunch errors involving misspelling in variable names. Also check for reversal of row and column subscripts.

14. If there are no error messages, but the answers are wrong, three main methods are used:

    a) Intermediate results should be printed out if any long calculations are performed; write statements may be inserted to check on the order of execution of the statements. (Unformatted I/O in WATFIV is useful here. See "Format-free I/O" under "Language Extensions," page VIII.7 of Appendix VIII.)

    b) Go thru the program statement by statement, performing all operations by hand and keeping track of the current value of the variables.

    c) Break up the program into segments or subprograms, running each individually to pinpoint where the error occurs.

There is additional information in Appendix VIII for debugging using the WATFIV compiler. See particularly pages VIII.17 to VIII.19.

SELF EVALUATION: You have already been debugging programs now for some time. You probably have some notion of your success or lack of it in debugging your own programs.

What's it like to debug someone else's program--one you didn't write? Your success at this task is a real measure of your debugging ability. Report to your instructor; he will give you a program listing that contains errors for you to debug.

ASSESSMENT TASK: Now that you are saturated with "bug killer" techniques, you are ready for the acid test. It is now your turn to climb over to the other side and help those farther down the ladder with their debugging problems. Be gentle, be tactful, ask penetrating questions, try to help the neophytes learn how to debug. If all else fails and you are in dire need to convince the neophyte that he can improve his technique, you may ask in a controlled manner, "Why in thunderation did you do that, stupid?"

When you are ready, report to your instructor and he will set up a schedule for you to be in diagnostic lab for several hours in the next 2-3 weeks. Keep a detailed log of your activities by filling out the form on the next page. You will be supervised by the diagnostician regularly assigned to the diagnostic lab. Be sure to get his signature and his evaluation when either you or he must leave the lab. When you have "served your time," then take your log sheet to your instructor for his approval.

Log for Diagnostic Lab

Your name ______________________

Date	Time in	Time out	Time spent	Supervisor's Name	Supervisor's Evaluation*	Total helped	Tally of visitors helped

* "S" for satisfactory; "U" for unsatisfactory. Place remarks below or on back.

WHAT NEXT? At this point the remaining units probably are 12 and 16. You may elect either one of these. If you have completed them, then you are ready to do UNIT 17, which is the last one.

UNIT #15 (COMSC)

This unit has been omitted from this manual.

UNIT #16 (COMSC)

TITLE: COMPUTER CONCEPTS

RATIONALE: After proceeding this far in the COMSC sequence you should be wondering what is really going on in that computer. How does it put programs, subprograms, instructions, and data together in a way that meaningful results occur? This unit should give you some insight into basic hardware elements of a computer and a feeling for the relationships that exist between hardware and programs.

OBJECTIVES: Describe hardware features of two different machines. Describe the interrelationships between hardware features and various features used in setting up a Fortran program.

PREREQUISITES: UNITS #11, 13 (COMSC)

ACTIVITIES: We shall take some time out here to describe hardware aspects of several different computers. A spectrum of references exists on different machines. The most comprehensive to date is the book by Bell and Newell which discusses in detail many existing computers. The book by Flores discusses briefly the organization of several different computers, while the book by Iverson introduces an elegant language that can be used to describe computer instruction sets concisely. The book by Struble refers specifically to the IBM/360 series and the book by Louden refers to the IBM 1800 and 1130 machines. Both of these books deal with assembly language programming for the respective machines.

G. G. Bell and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, 1971.

I. Flores, Computer Organization, Prentice Hall, 1969.

K. Iverson, A Programming Language, Wiley, 1962.

R. K. Louden, Programming the IBM 1130 and 1800, Prentice Hall, 1967.

G. Struble, Assembler Language Programming the IBM System/360, Addison Wesley, 1969.

Experience with assembler language programming is necessary to get a real feeling for computer hardware and its relationship to higher level programming. If you find the concepts described here and in the readings of interest, you might want to get some experience with assembler language programming followed by additional experience with compiler writing and studies in computer organization. One of the best ways to get an understanding of computer hardware concepts is to get hands on experience on a small computer such as an IBM 1130, a Digital Equipment Corporation PDP-8 or PDP-11, a Hewlett Packard hp 2100, or some similar machine.

Computers can be described in terms of processors (P), memory (M), switches (S), control lines (K), and input/output devices. Bell and Newell refer to such a description as a PMS description and have created a special language to facilitate the discussion of a PMS description. Computers can be described also in terms of instruction sets, data representations and registers. Iverson created an elegant language for the description of instruction sets. Bell and Newell created a different language for the description of instruction sets. Although these languages are well designed to describe machine characteristics, we shall not use them in this unit since it takes some study to become proficient with their use.

The memory of a computer stores data and instructions. Just as Fortran statements are arranged in cards in sequence, machine instructions are arranged in memory in sequence. Data also are arranged in memory in sequence. The only distinction between data and instructions is that instructions are interpreted and executed by the computer whereas data are not. At times, instructions may be operated on as data. Present day computer memories are made of magnetic cores. New technologies are under investigation for faster cheaper memories. Most of the memories operate in a two state or binary mode, that is in an "on-off" or "0 - 1" mode. Two state devices are fast, reliable, stable and reasonably cheap. Devices with more states currently lack one or more of these required characteristics.

Because of the use of two state technology, computers normally operate in a binary arithmetic mode. We shall discuss binary, octal, and hexadecimal arithmetic later. Memory is organized in a hierarchy of bit patterns. Two of the fundamental bit patterns are used to represent characters and groups of numbers. Some computers use 6 bits to represent a character resulting in 2^6 or 64 different unique characters. Recent computers use 8 bits to represent a character. How many unique characters are possible with an 8 bit representation? ______________________

Sample character codes are shown in Table 16.1.

	ASCII		IBM	
character	8-bit	6-bit(TTY)	8-bit	6-bit
blank	1000 0000	10 0000	0100 0000	11 0000
A	1010 0001	00 0001	1100 0001	01 0001
B	1010 0010	00 0010	1100 0010	01 0010
I	1010 1001	00 1001	1100 1001	01 1001
J	1010 1010	00 1010	1101 0001	10 0001
K	1010 1011	00 1011	1101 0010	10 0010
R	1011 0010	01 0010	1101 1001	10 1001
S	1011 0011	01 0011	1110 0010	11 0010
T	1011 0100	01 0100	1110 0011	11 0011
Z	1011 1010	01 1010	1110 1001	11 1001
0	0101 0000	11 0000	1111 0000	00 0000
1	0101 0001	11 0001	1111 0001	00 0001
9	0101 1001	11 1001	1111 1001	00 1001
(	0100 1000	10 1000	0100 1101	11 1100
)	0100 1001	10 1001	0101 1101	01 1100
+	0100 1011	10 1011	0100 1110	01 0000
-	0100 1101	10 1101	0110 0000	10 0000
.	0100 1110	10 1110	0100 1011	01 1011
=	0101 1101	11 1101	0111 1110	00 1011

TABLE 16.1
SEVERAL DIFFERENT CHARACTER CODES.

These character codes are recognized by different organizations and different computers. The ASCII code (American Standard Code for Information Interchange) is the result of an attempt to standardize character codes. The codes implemented by computer manufacturers differed considerably from the codes used by the common carriers (telephone, telegraph). Problems arose when computers were attached to common carrier lines. The ASCII code helps to provide standards for two previously independent industries.

The extended binary coded decimal interchange code (EBCDIC) is used by IBM 360 and 370 computers. By replacing a single bit under program control in a 360 or 370, that machine will recognize ASCII code. However, IBM, at present, does not support software to use the ASCII code.

There are two places where the character codes play important functions. One of these has been discussed earlier, namely the number of bits used to represent characters. The more bits per character, the greater the number of unique characters that can be represented by the code. The choice of the bit codes to represent characters is of lasting importance to the programmers. Since a character is represented by a string of bits, that character code also can be thought of as a number. When all the characters are arranged in sequence by their numeric codes the resulting sequence is called the collating sequence. Arrange the characters presented in the previous table in increasing sequence according to their 8 bit codes:

IBM code: blank, ., ______________________

ASCII code: ______________________

Are these sequences the same? ______________

If a list of peoples' names were placed in increasing alphabetic sequence according to the ASCII 8 bit code, would that list also be in increasing alphabetic sequence according to the IBM 8 bit code? ______________

What do you think the 8 bit IBM code and 8 bit ASCII for C, H, L, Q and V should be?

	8 bit IBM	8 bit ASCII
C		
H		
L		
Q		

	8 bit IBM	8 bit ASCII
V	____________	____________

Both character codes arrange the alphabet in increasing sequence. The IBM character code has 'holes' in it between I and J and between R and S whereas the ASCII code assigns 26 consecutive numbers to the 26 characters of the alphabet. If a file is placed in ascending sequence according to the ASCII code and then printed, would the listing be the same as if the file were placed in ascending sequence according to IBM code and then listed? The answer depends on what was in the file as follows:

characters in file	printed listings the same	
alphabetic only	yes	
numeric only	yes	
alphabetic and numeric	no	Why? ____________
alphabetic and special characters ?	____	Why? ____________
numeric and special characters ?	____	Why? ____________

The fact that different character codes exist can be of concern to you in applications where the interpretation of the code is dependent on the specific code used.

Since any group of bits could be called a character, if one is interested in processing characters, the computer should have an easy way to manipulate characters. Such machines are referred to as data processing machines. Other applications require the computer to perform a large number of computations quickly. Those machines are called scientific computers or "number crunchers." Several computers bridge the gap and provide both character manipulation and scientific computing capability so that the dichotomy no longer exists for those computers.

The arithmetic data representations also are important in describing a computer. Binary is usually the base. Integers are represented in terms of binary integers. Real numbers are represented in terms of "floating point" numbers. A floating point number consists of three parts: a sign, an exponent and a mantissa. The exponent may be in terms of base 2 or base 16 on a binary machine. In order to represent both positive and negative exponents and positive and negative values with a single sign position, the sign of the exponent part is determined by its magnitude compared with the largest magnitude allowed for an exponent. For example on the 360, 7 bits represent the exponent (a total of 128 different exponent values). Those exponent values that are greater than 64 are considered as positive exponent values, whereas those exponent values that are less than 64 are considered as negative exponent values. Such notation is given the name "excess 64" representation. On the 360 the base for the exponent is 16 rather than 2, while the base for the value is 2. Given the numbers

exponent	value
65	xxxxxxx
68	xxxxxxx

where xxxxxxx stands for the same mantissa in both cases, the second number is

$$16^3 = (2^4)^3 = 2^{12}$$

larger in magnitude than the first number.

Numeric data are said to occupy a "word" or group of words on a computer. Examples of word sizes are

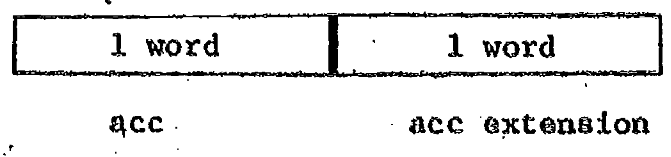
computer	word size
PDP-8	12 bits
HP-2115	16 bits
IBM-1130	16 bits
IBM 360	32 bits
IBM 7090	36 bits
CDC 3600	48 bits
CDC 6600	60 bits

Other numeric representations are possible. Half word or double word representations provide for more economic use of storage space in the first instance and for more accuracy in the second.

What other features should a computer have? (Your experience with Fortran should help you with this one.) ______________________

A computer should be able to perform calculations and tests. It should be able to initiate an input and an output. Furthermore, it must be able to exert control over all of these functions.

Calculations are performed in registers on most of the scientific machines. A register contains the same number of bits as a word. Registers in which computations are performed usually are called the accumulator and the accumulator extension. The extension is the least



significant part of the accumulator-accumulator extension pair. Addition and subtraction take place in the accumulator, whereas multiplication and division take place in the pair of registers. Tests can be performed to determine whether the previous result were zero, negative or positive. What Fortran statement performs this test?

____

Suppose the Fortran statement

```
IF (A - 7.2) 12,6,29
```

were executed and A had the value 7.19. What would be the next statement to be executed? ____________

The computer might translate this Fortran statement to a series of machine instructions similar to the following:

```
load the accumulator with A
subtract 7.2
branch if acc < 0 to statement 12
branch if acc = 0 to statement 6
branch if acc > 0 to statement 29
```

The IBM/360 has 16 general purpose registers for performing integer arithmetic instead of a single accumulator and a single extension. It also has 4 floating point registers for performing floating point arithmetic. The IBM 1130 has an accumulator and extension for integer arithmetic. Floating point arithmetic is simulated by software rather than hardware.

The machine instructions are executed sequentially similar to Fortran statements. The sequence is altered if a branch condition occurs.

Input/output occurs between memory and some external device. The list of different external devices that can be attached to a computer is an ever expanding one. Examples of such devices are

magnetic disk	teletype keyboard
magnetic drum	typewriter keyboard
magnetic tape	character display
magnetic strip	vector display
punched card reader	incremental plotter
magnetic card reader	line printer
optical card reader	character printer
punched paper tape reader	analog interface
card punch	communications interface

paper tape punch   another computer

Frequently a small computer is used to control the input/output. Such a device is called an I/O processor. The CDC 6600 has 10 peripheral processors attached to the main processor. Each of these peripheral processors is given a specific type of task to perform. Other computers have special purpose computers which control input/output operations for the faster devices. Such special purpose computers, or control units, have a limited instruction set which specifically relates to input/output.

A simplified diagram of a computer appears in FIGURE 16.1.

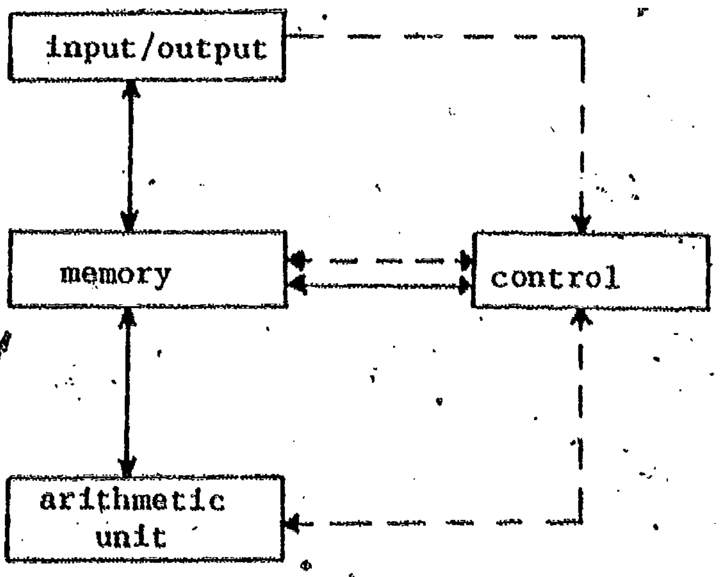


FIGURE 16.1
THE MAIN COMPUTER COMPONENTS.

In this example data paths are solid lines, control paths are broken lines.

Draw a simplified diagram of a computer which includes a secondary memory such as disk which has a data path to memory and a control path to the control unit.

Consider the Fortran expression

R = A + B * C

What components of the computer described in the previous diagram are activated at each stage of this computation?

1.	load B	control, memory, arithmetic unit
2.	multiply by C	control, memory, arithmetic unit
3.	add A	control, memory, arithmetic unit
4.	store result in R	control, arithmetic unit, memory

What components of the computer described in the diagram are activated by the instruction

READ(5,111) A,B ?

1. format 111 ____________________
2. read A ____________________
3. format 111 ____________________
4. read B ____________________
5. format 111 ____________________

What components are activated by the instruction

WRITE(6,113) C,D ?

1. ________ ____________________
2. ________ ____________________
3. ________ ____________________
4. ________ ____________________
5. ________ ____________________

What components are activated by the instruction

IF (A-B) 12,13,12 ?

1. load A ____________________
2. subtract B ____________________
3. branch ____________________

Refer to UNIT #16 ACTIVITIES TABLE for more information.

UNIT #10 (COMSC) was concerned with loops and methods for executing loops in Fortran. The example

```
      SUM=0.0
      DO 17 I=1,N
        SUM=SUM+X(I)
   17 CONTINUE
```

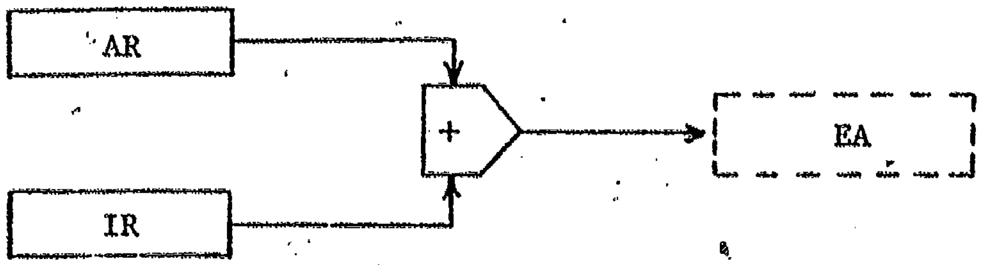
illustrates a loop in which the addresses associated with the array variable X are modified successively. Recall that X(1) is the address of the first element of the X array, X(2) is the address of the second element of the X array, etc. A computer must have the ability to compute addresses of array elements easily. One method of calculating addresses is to make use of a special register (or set of registers) called an index register. The address of the array X is held in the instruction; the index is held in the index register; the address of X and the value in the index register are added to produce an "effective address" that is used to reference the location X(I) in memory. If a memory reference instruction is to use an index register, an indicator must be set in the instruction to that effect sometime prior to the execution of the instruction. For the statement

SUM=SUM+X(I)

a schematic of effective address calculation, assuming index register 3 contains the index, appears as follows:

instruction	instruction address	index register 3	effective address
load accumulator from SUM	SUM	i	SUM
add to accumulator from X modified by index register 3	X	i	X+i
store accumulator into SUM	SUM	i	SUM

The increment step of the DO loop modifies index register 3 and also tests index register 3 to determine whether the loop has been completed. Symbolically the effective address computation can be represented as follows:



where

AR	is the address register,
IR	is the index register,
EA	is the effective address.

The EA is enclosed in a dashed box to indicate that it does not appear in storage; rather it is created, used, and discarded unceremoniously in fractions of a microsecond.

The Fortran statements

```
      DO 57 J=1,N
        X(J) = Y(J)+A(J)
   57 CONTINUE
```

illustrate more graphically the usefulness of an index register in effective address computation. Suppose index register 2 contains the index J. Fill in the following table that simulates effective address calculation.

instruction address	index register 2	effective address
Y	j	
A	j	
X	j	

In this case index register 2 is used in three effective address computations.

How many index registers does a computer have? That depends on the specific computer. Some computers do not have index registers; others have three, others have eight, still others have fifteen.

The 1130 has 3 index registers; the 360 has 15 registers that may be used as index registers. One must be able to set an index register to an initial value, increment (or decrement) an index register, test an index register, branch based on the test, and store an index register. Such capabilities are implemented differently on different machines.

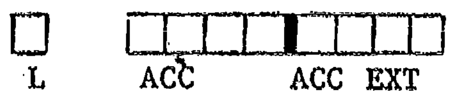
Other hardware features include base registers, indirect addressing, floating point registers, interrupt facilities, input/output channels, multiplexor channels, control units, peripheral processors, memory locks and keys (also called memory protect), and relocatability. Many of these topics are covered in a course in assembly language programming; others are covered in topics in computer organization.

The types of data a computer allows are an important key to its character. The set of operations which a computer performs is another basic indication of its character. The IBM 1130 has a basic instruction set of 29 instructions, while the IBM 360/65 has over 140 instructions and the IBM 370/165 has over 160 instructions. Subsets of the instruction set pertain to specific hardware functions. Several of these functions are listed below.

function	instruction types
arithmetic	loading a register, storing a register, addition, subtraction, multiplication, division in binary integer, floating point, and decimal, round, compare
index register	load index, modify index, store index, test index, branch on index
bit manipulation	test bit, insert bit, shift register left, shift register right, rotate register, store bit, and, or, exclusive or, mask, test
character manipulation	move character, insert character, test character, translate character, edit character
input/output	initiate I/O, test I/O, test channel, stop I/O
miscellaneous	interrupt processing, storage protect, error recovery

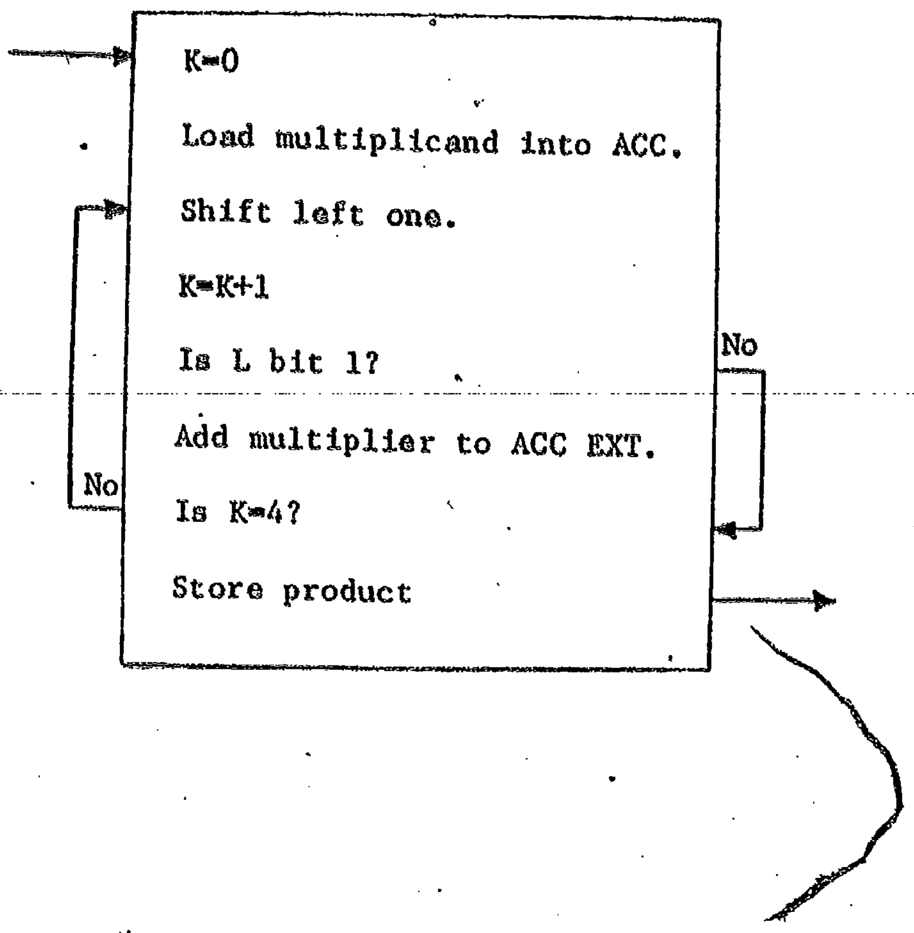
Since computer instruction sets vary widely, not all computers will contain all of these functions. If a computer does not have floating point hardware, floating point arithmetic may be simulated by a program. The resulting simulation is a great deal slower in execution speed than hardware would be, however. Most of the functions listed above may be simulated by programs if they do not exist on the particular computer. Exceptions are input/output and interrupt processing. These may be quite difficult or impossible to simulate if the hardware does not exist on the machine. Many of the hardware features on today's computers were simulated by programs on earlier computers. In fact, multiply and divide are simulated to this day on some computers by making use of repeated additions and shifts in the first case and by repeated subtractions, tests, and shifts in the second case! Those software simulated features that proved useful were incorporated into the hardware of later machines. Floating point, index registers, character manipulation, storage protect, and interrupt processing are a few examples of software simulated features that have been incorporated into hardware.

SELF EVALUATION: Suppose a computer as a 4 bit accumulator and a four bit accumulator extension. Suppose



the contents of both ACC and ACC EXT can be shifted left by 1. Suppose the leftmost bit of ACC replaces the bit in L after the shift. Suppose the L bit can be tested for 0 or 1. Draw a flowchart to simulate multiplication by addition by loading the multiplicand in ACC, shifting left the ACC and ACC EXT one position, testing the L bit. If the L bit is 1, the multiplier should be added into ACC EXT. The product appears in ACC and ACC EXT.

Answer:



ASSESSMENT TASKS: Report to your instructor and discuss the assessment tasks with him.

WHAT NEXT? You may do either UNIT #12 or UNIT #14. If you have completed 12 and 14, then you should be ready for UNIT #17, the last one.

UNIT #17 (COMSC)

TITLE: General Fortran Programs

RATIONALE: Constructing general, efficient, accurate and skillfully done computer programs for solving problems is a worthwhile goal for any programmer. That goal also happens to be the overall objective of this course.

You should by this time have developed considerable expertise in constructing Fortran programs and running them on the computer. Now you should put that skill to use in a fairly sophisticated problem solving situation.

OBJECTIVES: At the end of this unit you will demonstrate your ability to solve problems efficiently and accurately using Fortran programs that you construct and a computer.

PREREQUISITES: If you have chosen the option of trying for an "A," then UNITS 12, 14, and 16 are required. If you have chosen to try for a "B," then only UNIT 12 is required.

ACTIVITIES:

A. Some general considerations

A Fortran program of optimum worth is one that is accurate and efficient. Some of the considerations regarding accuracy were discussed in UNIT 12 and will not be discussed again in this unit. Rather we will focus on efficiency.

An efficient program is one that uses the computer's capabilities to the greatest advantage. Efficiency suggests in particular that the best possible use be made of the computer's storage and time. Execution time, compile time, and required storage should all be minimized with respect to each other. Since these variables are all interdependent, minimizing one may increase another; some compromises will be necessary. Let's notice some of the factors involved in efficient programming.

1. Length of program. A program that is longer than necessary will probably require more compile time, more execution time, and more storage for the object code. As the program is shortened, these variables will tend to decrease. If, however, the program is made much shorter by combining statements and operations as much as possible, the compile time will tend to increase -- in some cases, drastically; the required storage and execution time may also be increased. It is important to realize then that the shortest program is not necessarily the best.
2. Sequence of operations. In general, the more operations the computer has to perform, the greater will be the execution time. Unnecessary operations-- such as transfer to another transfer statement and then to another statement, rather than a direct transfer to the last statement mentioned -- should

always be avoided. Arithmetic expressions should, in general, be arranged so that unnecessary operations are not performed. (Not only do unnecessary operations require extra execution time, but also accuracy is reduced, as discussed in UNIT 12.) The same is true for assignment statements. (For example, X=0 requires integer zero to be converted to real zero before it is stored into X. X=0.0 requires no unnecessary operations.)

3. Input and output. In general, any kind of I/O requires more execution time than internal machine operations. A program may become "I/O bound," that is, the computer is having to use much of its execution time for I/O to occur. It is very easy, for example, to cause the 1130 with a typewriter-printer to be I/O bound, since the printing is very slow. The 360 handles I/O much more efficiently by a process in which I/O is performed through intermediate magnetic disc storage which has a very small access time. Therefore, I/O does not slow the 360 very much at all. Even so, excessive I/O should be avoided.

4. Storage for variables and arrays. When reserving storage for arrays, you should use only as much space as necessary. If you can avoid the use of arrays, that may be even better. Use as few variables as possible in a program. For example,

use one variable for the index of all unnested DO loops. For another example, consider the program segments in Figure 17.1, which find the average of an array of numbers. The one on the right makes better use of storage by having less variables than the one on the left. Reuse variables whenever possible.

```
      SUM=0.0                              AVG=0.0
      DO 1 J=1,N                           DO 1 J=1,N
        SUM=SUM+A(J)                         AVG=AVG+A(J)
    1 CONTINUE                           1 CONTINUE
      XN=N                                 XN=N
      AVG=SUM/XN                           AVG=AVG/XN
```

Figure 17.1. A program segment for calculating the average of a set of numbers, illustrating how variables may be reused.

5. Mixed mode. If mixed mode arithmetic is available, usually it is slower than arithmetic on a single mode. Since mixed mode arithmetic requires extra operations for changing all operands to the same type, in general it should be avoided, especially if the expression is in a loop. Examine the program segment in Figure 17.2, which generates a table of x and y values for the equation

$$y = 3x^3 - 2x^2 + 5x - 2.$$

Since the calculation of y occurs in a loop, the integer constants in the polynomial must be converted to real constants with each iteration, increasing execution time. Mixed mode should not be used in this expression. In fact, there is never any reason to use mixed mode operations in which one or more of the operands is a constant written in a different mode from that of the rest of the expression.

```
          DO 2 J=1,50
            READ(5,10)X
            Y = 3 * X**3 - 2 * X**2 + 5 * X - 2
C
C             THE EXPRESSION SHOULD BE WRITTEN
C           Y = 3.* X**3 - 2.* X**2 + 5.* X - 2.
C
            WRITE(6,10)X,Y
        2 CONTINUE
       10 FORMAT(1X,2F10.2)
```

Figure 17.2. Program segment illustrating an inefficient mixed mode expression. (Notice that X**3 is not mixed mode.)

There are cases in which mixed mode is definitely advantageous. Refer to the program segment in Figure 17.1. In the calculation of the average where AVG/XN is evaluated, compile time, execution time, and storage requirements for variables and the object code would probably be reduced by using the mixed mode expression AVG/N, since the statement XN=N is absent and does not have to be compiled, stored, and executed. Besides that, storage for one less variable (XN) is required. Mixed mode may be useful, then, when both operands are variables and the expression is evaluated only once (or very few times) in the program. If, however, N, for example, were to be used several times as an arithmetic operand in one or several mixed mode expressions then, probably addition of the statement XN=N and subsequent use of XN would be better.

Mixed mode is useful when used with discretion.

6. Integer arithmetic. Integer (fixed point) arithmetic is considerably faster than real (floating point) arithmetic, since there is no decimal point to keep track of. Integer arithmetic should be used whenever possible, and especially all counters should be in integer mode.

7. DO loops. DO loops may or may not require extra compile time and execution time, depending on the compiler. But, as a general rule, as few DO statements as possible should be used in a program. For example, instead of using two short DO loops, combine the two into a single loop, if possible. In Figure 17.3 there are two program segments for finding the sum and the sum of the squares of the elements of an array. The one on the left uses two DO loops, making two passes through the array, while the one on the right uses only one loop and makes one pass through the array. The efficiency

has also been improved slightly by using the multiplication operator instead of the exponentiation operator and by eliminating the CONTINUE statement.

```
C  INEFFICIENT PROGRAM
      S=0.0
      DO 3 K=1,N
        S=S+A(K)
    3 CONTINUE
      SSQ=0.0
      DO 4 K=1,N
        SSQ=SSQ+A(K)**2
    4 CONTINUE
```

```
C  MORE EFFICIENT PROGRAM
      S=0.0
      SSQ=0.0
      DO 3 K=1,N
        S=S+A(K)
    3   SSQ=SSQ+A(K)*A(K)
```

Figure 17.3. Program segments for finding the sum and the sum of the squares of the elements of an array.

Another important rule is that you should never put unnecessary operations inside loops. Figure 17.4 illustrates an unnecessary step, XN=N, that must be performed needlessly N-1 times, having a need to be performed only once.

```
C  UNNECESSARY STEP
C  INSIDE A DO LOOP
      S=0.0
      DO 6 K=1,N
        XN=N
        S=S+A(K)
    6 CONTINUE
      AV=S/XN
```

Figure 17.4. Program segment for finding the average of the elements of an array.

8. Storage of character data. Character strings should be stored so that the maximum number of characters is placed into the storage locations. For the 360, in single precision a maximum of four characters is allowed (eight in double precision), while on the 1130 it may be four or two, depending upon whether the storage location is associated with a real type or an integer type, respectively. If you wish, however, to make use of single character data in a program, then you will have to store just one character per storage location. (There are ways around this problem, but they are beyond the scope of this course.)

There are many other factors to be considered in various situations. But our main purpose here is just to give you a feel for some of the problems involved. Be aware of these problems as you construct programs; try to minimize all three variables -- compile time, execution time, and required storage -- simultaneously, realizing that compromises will be necessary.

The intended uses to which a program will be put also dictate what additional compromises may be necessary. For example, a program that is designed to be run on any computer cannot make use of "short cut" options available on some computers. Unless a program is definitely intended to do a specific "one shot" kind of task, it should be constructed to handle general cases rather than specific cases. General programs probably will be longer than specific programs, however; and they usually become longer as they become more general. Several compromises will be necessary, therefore, as a program is generalized for broader application to a wide range of computers and similar problems. Furthermore, compile and execution times as well as storage requirements for a given program may vary markedly with different computer systems and with different compilers used on the same computer.

B. Some specifics.

Refer to UNIT 17 ACTIVITIES TABLE.

C. Someone else's ideas.

The following material is reprinted from the Waterloo University Newsletter, September, 1970.

The following is a list of hints on how to optimize FORTRAN coding in order to achieve better accuracy in calculations and to increase the speed of execution of programs in general..

1. Think carefully about the problem before programming. If you are unsure of the techniques involved, make a point of seeing someone at the Information Desk. In particular, make sure your program checks out on the following points:

    a) The program does exactly the job you want it to do.

    b) Input and output are in the most convenient format.

    c) Today's results will be understandable in six months' time without having to re-run the program?

    d) The program structure naturally reflects the problem structure, thereby being easier to code and debug..

    e) The program can easily be extended to cope with an extended version of the problem.

2. Use DOUBLE PRECISION arithmetic in critical calculations wherever space allows.
3. Use a minimum of mixed-mode arithmetic. The extra coding generated can in some cases take more time to execute than the arithmetic itself.
4. Avoid using SUBROUTINES and FUNCTIONS for small repeated tasks.
5. Arrange the program logic to avoid branches whenever possible.
6. Make the most probable result of all LOGICAL IF statements a simple drop through instead of a branch.
7. Use LOGICAL IF's instead of ARITHMETIC IF's.
8. Choose variable types to avoid conversions (i.e. mixed modes) whenever possible.
9. Reduce input-output to the minimum necessary.
10. Avoid implied DO's in input-output where possible.
11. Align all COMMON and EQUIVALENCE statements with variables in decreasing order of storage space (i.e. COMPLEX*16 before REAL*8 before ... LOGICAL*1).
12. Calculate all quantities which are constant through a program at the beginning, and calculate all quantities constant through a loop outside the loop.

```
      DO 20 I=1,450
      C(I+3;2*I+1)=D(I+2,2*K)+E(2*L+1)
   20 CONTINUE
```

should be written:

```
      M=2*K
      N=2*L+1
      DO 20 J=3,452
      C(J+1,2*J-3)=D(J,M)+E(N)
   20 CONTINUE
```

This modification saves 898 multiplications and 899 additions.

13. Store any ARRAY element used more than once, in a loop in a temporary SCALAR variable.

14. Use as few subscripts as possible on arrays (i.e. use A(720) instead of A(12,6,10)).

15. a) Make all on-off switches, flags, etc. LOGICAL*1.

e.g.

```
      DIMENSION X (500)
      LOGICAL*1 OUTPUT
      READ3,OUTPUT,N
    3 FORMAT(L1,I5)
      -
      -
      -
      IF(OUTPUT)PRINT9,(X(I),I=1,N)
      -
      -
      -
```

b) Make all test variables (3-way or more) INTEGER*2.

e.g.

```
      INTEGER*2 BRANCH
      READ3,BRANCH
    3 FORMAT(I3)
      IF(BRANCH)7,304,82
      -
      -
```

Here, only 16 bits are tested in the IF statement, whereas the use of a 4-byte integer variable would necessitate the testing of 32 bits.

16. Use assigned GO TO's instead of computed GO TO's.

17. a) Use LOGICAL IF's instead of 2-way GO TO's.

```
e.g.    Inefficient                      Efficient

        IF(DOG)14,23,14                  IF(DOG.EQ.0)GO TO 23
     14 X=X+1                            X=X+1
        -                                -
        -                                -
        -                                -
     23 X=X-1                         23 X=X-1
```

Using a logical IF here is more efficient because it generates less coding and executes faster than the arithmetic IF.

b) Use ARITHMETIC IF's instead of 3-way GO TO's.

```
e.g.    Inefficient                      Efficient

        IF(TEST.EQ.0)GO TO 1             IF(TEST)3,1,5
        IF(TEST.GT.0)GO TO 5           3 STOP
        STOP                           1 X=X+1
      1 X=X+1                          5 PRINT2,X
      5 PRINT2,X                       2 FORMAT(F10.3)
      2 FORMAT(F10.3)                    -
        -                                -
        -                                -
        -
```

For the same reasons as in (a), it is more efficient to use an arithmetic IF here.

18. Using IF statements to determine conditional branches is less efficient than using assigned or computed GO TO's. The computed GO TO uses more overhead time and space than the assigned GO but it still is better than an IF statement.

19. Where possible, pass variables to SUBROUTINES through COMMON instead of using parameter lists; this saves much time because addresses do not have to be passed down to the subroutine for the variables in the calling sequence..

20. Do not test for equality using floating-point variables, because of roundoff error in low-order bits. Use .GE. or .LE.

21. Use SQRT instead of **.5, since the SQRT routine is faster than the logarithms routine used to evaluate expressions of the form X**R.

22. For small powers, use A*A*A ... or A**I with I=R instead of A**R, where R is a floating point integer; values raised to integer powers are computed by repetitive multiplications, whereas values raised to real powers are computed by using logarithms.

23. Use unformatted I/O for scratch units; FORMATs waste time and space.

24. Always debug programs under WATFIV; the compilation time

is faster and the error-messages are useful.

25. Production-type jobs (i.e. those large-core and time-consuming jobs which are run frequently without changing the source program - or perhaps with changing only one subroutine) should not be run under WATFIV. Any unchanging routines in such programs should be compiled into object decks under FORTRAN H. Running under FORTRAN H will decrease the execution time.

SELF EVALUATION: Pages 17.18 - 17.21 contain programs that produce the same end results, but they vary considerably in how they obtain the results.

Here is the problem to be solved: make a list of the names and ages of all people who are between the ages of 17 and 21 and find their average age. Assume that you have a set of data cards with the following information punched in each card:

Name, columns 1-20

Social security number, columns 21-31

Age, columns 32-34, right justified

Place a blank card at the end of the punched cards as a signal that all the data have been processed.

Examine both programs very carefully and determine what advantages and disadvantages may exist regarding the efficiency of each program.

Record your observations in the spaces provided. On the pages following the programs some possible observations are recorded so that you can check yourself. If you made observations that are not listed, then discuss them with your instructor to see whether you are correct.

The programs were run on 360 WATFIV. Compare the execution and compile times and the storage requirements of the two programs and harmonize them with your observations, if possible. (Your observations may be correct regardless.)

Notice in particular that these programs were executed with just a few data cards (actually 10, 5 of which did not qualify for listing). If the programs were used with 500 cards, or 50,000 cards, would your observations still hold? Does the number of data cards, in fact, make any difference in compile time? In execution time? In storage requirements?

What about the compatability of the two programs with the 1130? Just how general are the two programs?

These are some of the questions that you need to answer as you make your observations.

The following pages contain the programs for the SELF EVALUATION section.

17.18'

```
$JOB
C
C      ----- PROGRAM 1 -----
C
C  THIS PROGRAM MAKES A LISTING OF PEOPLE WHOSE AGES ARE 18-20
C  AND ALSO CALCULATES THEIR AVERAGE AGE.
C
C            ***********************************************
C            *                                             *
C            *  LIST OF VARIABLES                          *
C            *    DATA   - ARRAY CONTAINING NAMES IN FIRST *
C            *             FIVE COLUMNS AND AGES IN COLUMN *
C            *             SIX                             *
C            *    KSUM   - USED FOR SUMMING AND CALCULATING*
C            *             AVERAGE                         *
C            *    SUM    - REAL EQUIVALENT OF KSUM         *
C            *    N      - COUNTER FOR NUMBER OF CARDS READ*
C            *    K      - COUNTER FOR NUMBER OF NAMES ON  *
C            *             THE LIST                        *
C            *    IN     - CARD READER LOGICAL UNIT NUMBER *
C            *    LP     - LINE PRINTER LOGICAL UNIT NUMBER*
C            *                                             *
C            ***********************************************
C
      INTEGER DATA(500,6)
      DATA IN,LP/5,6/
      DATA N,K,KSUM/3*0/
C--------- WRITE HEADINGS.
        WRITE(LP,1)
C--------- PLACE DATA INTO ARRAY.
      DO 90 I=1,500
         READ(IN,2)(DATA(I,J),J=1,6)
C--------- CHECK FOR LAST CARD.
         IF(DATA(I,6).EQ.0)GO TO 10
         N=N+1
   90 CONTINUE
C--------- SCAN ARRAY FOR AGES IN THE RANGE 18-20.
   10 DO 91 I=1,N
         IF(DATA(I,6).GT.20.OR.DATA(I,6).LT.18)GO TO 91
C--------- SUM AND COUNT FOR FINDING THE AVERAGE.
         K=K+1
         KSUM=KSUM+DATA(I,6)
         WRITE(LP,3)(DATA(I,J),J=1,6)
   91 CONTINUE
C--------- CHANGE INTEGER SUM TO REAL FOR CALCULATING AVERAGE.
C             THEN CALCULATE AVERAGE AND ROUND TO NEAREST INTEGER.
        SUM=KSUM
        KSUM=SUM/K+0.5
C--------- WRITE AVERAGE AGE.
        WRITE(LP,4)K,KSUM
      STOP
C------------------------------------------------------------------------
C       FORMAT STATEMENTS.
    1 0    FORMAT(/' LIST OF PEOPLE WHOSE AGES ARE 18-20'//
     1      '                    NAME          AGE'     /)
    2      FORMAT(5A4,11X,I3)
    3      FORMAT(6X,5A4,3X,I2)
    4      FORMAT(/' THE AVERAGE AGE OF ',I5,' PEOPLE IS ',I2,'.')
C------------------------------------------------------------------------
      END
```

```
$ENTRY

LIST OF PEOPLE WHOSE AGES ARE 18-20

              NAME            AGE

     DDDDDDDDDDDDDDDDDDDD     20
     FFFFFFFFFFFFFFFFFFFF     18
     GGGGGGGGGGGGGGGGGGGG     19
     HHHHHHHHHHHHHHHHHHHH     18
     IIIIIIIIIIIIIIIIIIII     20

THE AVERAGE AGE OF      5 PEOPLE IS 19.

CORE USAGE       OBJECT CODE=     5736 BYTES,ARRAY AREA=    12000

COMPILE TIME=      0.83 SEC,EXECUTION TIME=      0.14 SEC,  WATFIV
```

Possible good points	Possible bad points

17.20

```
      $JOB
      C
      C        ----- PROGRAM 2 -----
      C
      C   THIS PROGRAM MAKES A LISTING OF PEOPLE WHOSE AGES ARE 18-20
      C   AND ALSO CALCULATES THEIR AVERAGE AGE.
      C
      C               *********************************************
      C               *                                           *
      C               *  LIST OF VARIABLES                        *
      C               *    NAME  - PERSON'S NAME                  *
      C               *    SSN   - PERSON'S SOCIAL SECURITY NUMBER *
      C               *    AGE   - PERSON'S AGE (INTEGER)         *
      C               *    N     - COUNTER FOR NAMES ON THE LIST  *
      C               *    XN    - THE REAL EQUIVALENT OF N       *
      C               *    IN    - CARD READER LOGICAL UNIT NUMBER *
      C               *    LP    - LINE PRINTER LOGICAL UNIT NUMBER *
      C               *    AVG   - USED FOR SUMMING AND CALCULATING *
      C               *            AVERAGE                        *
      C               *                                           *
      C               *********************************************
      C
            INTEGER AGE
            DIMENSION NAME(10),SSN(11)
            DATA IN,LP/5,6/,AVG/0.0/,N/0/
      C--------- PUT OUT HEADINGS.
               GO TO 100
      C--------- READ DATA CARD.
         10    READ(IN,2)NAME,SSN,AGE
      C--------- CHECK FOR LAST CARD.  AGE IS ZERO ON LAST CARD.
                IF(AGE)20,20,13
      C--------- ON ENCOUNTERING LAST CARD, CALCULATE AVERAGE.
         20       AVG=AVG/N
      C--------- ROUND AVERAGE TO NEAREST INTEGER.
                  AGE=AVG+0.5
                  GO TO 102
      C--------- PROCESS CARD.  CHECK FOR AGES IN THE RANGE 18-20.
         13     IF(AGE-21)11,10,10
         11     IF(AGE-17)10,10,12
      C--------- SUM FOR AVERAGE AND COUNT.
         12       AVG=AVG+AGE
                  N=N+1
                  GO TO 101
      C------------------------------------------------------------------
      C     OUTPUT STATEMENTS.
      C
      C         HEADINGS.
        100        WRITE(LP,1)
                   GO TO 10
      C         LINE OF LIST.
        101        WRITE(LP,3)NAME,AGE
                   GO TO 10
      C         AVERAGE AGE.
        102        WRITE(LP,4)N,AGE
                   STOP
      C------------------------------------------------------------------
      C     FORMAT STATEMENTS.
      C       INPUT.
          2      FORMAT(10A2,11A1,I3)
      C       OUTPUT.
```

```
        1 0     FORMAT(/' LIST OF PEOPLE WHOSE AGES ARE 18-20'//              17.21
          1     '                    NAME            AGE'      /)
        3       FORMAT(6X,10A2,3X,I2)
        4       FORMAT(/' THE AVERAGE AGE OF ',I5,' PEOPLE IS ',I2,'.')
      C-----------------------------------------------------------------------
             END

      $ENTRY
 LIST OF PEOPLE WHOSE AGES ARE 18-20

             NAME            AGE

     DDDDDDDDDDDDDDDDDDDD     20
     FFFFFFFFFFFFFFFFFFFF     18
     GGGGGGGGGGGGGGGGGGGG     19
     HHHHHHHHHHHHHHHHHHHH     18
     IIIIIIIIIIIIIIIIIIII     20

 THE AVERAGE AGE OF     5 PEOPLE IS 19.

 CORE USAGE        OBJECT CODE=     5560 BYTES,ARRAY AREA=       84 BYTES,TOTAL AR

 COMPILE TIME=     0.74 SEC,EXECUTION TIME=     0.14 SEC,  WATFIV - VERSION 1 L
```

Possible good points	Possible bad points

Some Observations on Program 1

Good points.

Line 8: Integer counter used.

Lines 2 and 3: DATA used for initializing constants rather than assignment statements.

Line 7: Logical IF is faster than arithmetic IF.

Line 13: Integer arithmetic used for summing.

Lines 13 and 17: Previously used variable KSUM reused, conserving storage.

Line 21: Maximum number of characters stored per storage location by using A4 fields.

Line 17: Good use of mixed mode.

Bad points.

Line 1: Using a two-dimensional array makes program less general and uses a great deal of storage. A large number of cards could not be processed using this program.

Lines 5 and 10: The two DO loops could be combined into one, making the use of a two dimensional array unnecessary.

Line 11: Compound argument in logical IF may be slower than two simple logical IF's.

Lines 11 and 13: A new nonsubscripted variable should be set up to replace DATA(I,6) in these statements since referencing the subscripted variable three times is time consuming.

Lines 9 and 15: Removal of CONTINUE's would reduce the number of executable statements.

Program is not compatable with 1130, if that matters.

Line 8: Counter not needed; index of DO should be used rather than separate counter.

Some Observations on Program 2

Good points.

Organization of program is convenient, having all output statements together.

Program can handle an unlimited number of data cards.

Can be used on the 1130 by changing I/O unit numbers in DATA statement.

Lines 7 and 12: Uses AVG for storing both the sum and the average, reducing the number of variables.

Line 7: Mixed mode is better than using

```
XN=N
AVG=AVG/XN
```

assuming mixed mode capability, of course.

Lines 5 and 17: Program requires a great deal of I/O in the loop and will probably be I/O bound on the 1130. This may not be bad on the 360, however. The alternative, placing the data into an array, may require large amounts of storage.

Line 3: DATA statements used for initializing constants.

Bad points.

Novel organization requires excessive number of transfer statements.

Line 12: Mixed mode expression in sum is inefficient. Besides, the summing should be in integer mode.

Line 5: The social security number is never used and shouldn't be read.

Lines 2 and 21: Storage of characters is not efficient since only two characters per storage location are used for NAME and only one for SSN. Since, however, NAME is integer and a maximum of two characters is allowed on the 1130, this makes the program compatable with the 1130. It would be better to use REAL NAME(5) and 5A4 on both machines.

Lines 6, 10, and 11: Arithmetic IF statements are slower than logical IF's for two-way branching; but, if compatability with the 1130 is important, then that is a necessary compromise.

ASSESSMENT TASKS: If you are trying for a "B" in the course, then you will be required to construct a Fortran program to solve one problem given to you by your instructor.

If you are trying for an "A," then you will be required to construct programs to solve two problems given to you by your instructor.

In either case, see your instructor.

WHAT NEXT? You're on your own!

ENRICHMENT OPPORTUNITIES: If you are so excited about UNIT 17 that you would like to work on additional problems, see your instructor. You may wish to work on a project of your own, but get the instructor's approval before you do.

There are more advanced courses which you may wish to take. A partial list with a brief description of each course is given below:

COMSC 2123: Intermediate Programming. More on algorithmic problem solving with the computer. Use of files. IBM System/360 Fortran language extensions and Job Control Language (JCL).

COMSC 3223: Digital Computer Methods. Digital computer approximate solutions of algebraic and transcendental equations, solutions of linear and non-linear equations, functional approximations, least squares curve-fitting and allied topics. Practical programming experience in the applications of these techniques.

COMSC 3333: Procedures and Algorithmic Processes. The description and programming of numeric and non-numeric problems. The concept of an algorithm. PL/I language.

See your instructor if you want additional information.

APPENDIX I

COMPARISONS OF VARIOUS FORTRAN IMPLEMENTATIONS

y ≡ yes
n ≡ no
NR ≡ no ruling

	USA STANDARD FULL	USA STANDARD BASIC	WATFIV	360	360 DOS BASIC FORTRAN IV	1130 ~~(monitor version 2)~~
character set						
A···Z 0···9 blank=+-*/(),	y	y	y	y	y	y
'(apostrophe)	n	n	y	y	y	y
statement continuation lines	19	5	5	19	19	5
numeric statement label (decimal digits)	1 to 5	1 to 4	1 to 5	1 to 5	1 to 5	1 to 5
variable name (characters)	1 to 6	1 to 5	1 to 6	1 to 6	1 to 6	1 to 5
data types						
integer	y	y	y	y	y	y
real	y	y	y	y	y	y
double precision	y	n	y	y	y	n
complex	y	n	y	y	n	n
logical	y	n	y	y	n	n
Hollerith	y	n	y	y	y	n
integer range	NR	NR	$\pm(2^{31}-1)$	$\pm(2^{31}-1)$	$\pm(2^{31}-1)$	$\pm(2^{15}-1)$
real constant						
basic real constant	y	y	y	y	y	y
real exponent range	NR	NR	±75	±75	±75	-39 to +38
real single precision (# of digits)	NR	NR	7	7	7	7
integer constant followed by a decimal exponent	y	n		y	n	n
double precision constant						
real double precision (# of digits)	NR	NR	16	16	16	10
real constant with 'D' in place of 'E'	y	n	y	y	n	n
number of array dimensions	3	2	7	7	3	3
subscript form given by (integer) constant*variable ±constant	y	y	y	y	y	y

	USA STANDARD FULL	USA STANDARD BASIC	WATFIV	360	360 DOS BASIC FORTRAN I	1130 (monitor version 2)
H	y	y	y	y	y	y
' ' (literal)	n	n	y	y	y	y
I	y	y	y	y	y	y
L	y	n	y	y	n	n
T	n	n	y	y	y	n
X	y	y	y	y	y	y
Z	n	n		y	n	n
A format maximum field width	NR	NR		8	4	6
format (parens levels)	2	1	3		2	1
scale factor	y	n		y	n	n
blanks in numeric conversions						
high-order	zero	zero	zero	zero	zero	zero
within the field	zero	error	zero	zero	error	zero
real conversions						
integer plus exponent						
E type exponent	y	y	y	y	y	y
D type exponent	y	n	y	y	n	n
format during execution	y	n	y	y	n	n
statement functions must precede the 1st executable statement and follow the specification statements	y	y	y	y	y	y
type specification in a function statement	y	n	y	y	y	y
function may define or redefine its arguments	y	n	y	y	n	n
transmit in a CALL						
Hollerith arguments	y	n		y	y	n
external subprogram names	y	n		y	y	y
block data subprograms	y	n	y	y	n	n
specification statements precede 1st executable statement	y	y	y	y	y	y
DIMENSION, COMMON, EQUIVALENCE must be ordered	n	y	n	n	y	y
external function may alter variable in COMMON	y	n		y	n	n
mixed mode arithmetic			y	y	y	y

270

	USA STANDARD FULL	USA STANDARD BASIC	WATFIV	360	360 DOS BASIC FORTRAN IV	1130 (monitor version 2)
relational expressions	y	n	y	y	n	n
logical operators	y	n	y	y	n	n
assigned GO TO	y	n	y	y	y	n
logical IF	y	n	y	y	n	n
DO extended range	y	n	n		n	y
READ and WRITE						
formatted	y	y	y	y	y	y
unformatted	y	y	y	y	y	y
REWIND	y	y	y	y	y	y
BACKSPACE	y	y	y	y	y	y
ENDFILE	y	y	y	y	y	y
formatted records						
1st character printed	n	n	n	n	n	n*
space before printing						
blank 1 line	y	n	y	y	y	y*
0 2 lines	y	n	y	y	y	y*
1 1st line, new page	y	n	y	y	y	y*
+ suppress space	y	n	y	y	y	y*
adjustable dimension	y	n	y	y	n	n
common						
blank	y	y	y	y	y	y
named	y	n	y	y		n
array size declared	y	n	y	y	y	y
external statement	y	n		y	y	y
type statement	y	n	y	y	n	y
dimension information	y	n	y	y	n	y
data statement	y	n	y	y	y	y
format types						
A	y	n	y	y	y	y
D	y	n	y	y	y	n
E	y	y	y	y	y	y
F	y	y	y	y	y	y
G	y	n	y	y	n	n

*Applies to line printer not console typewriter

APPENDIX II

COMPUTER SCIENCE GLOSSARY

ABEND
ABNORMAL END.

ABSOLUTE ADDRESS
AN ADDRESS THAT IS PERMANENTLY ASSIGNED BY THE MACHINE DESIGNER TO A STORAGE LOCATION.

ACCESS TIME
THE PERIOD OF TIME NECESSARY TO LOCATE AND TRANSFER THE CONTENTS OF A SPECIFIED MEMORY LOCATION INTO A WORKING REGISTER OR, CONVERSELY, TO TRANSFER THE CONTENTS OF A WORKING REGISTER TO A SPECIFIED MEMORY LOCATION.

ACCUMULATOR
A REGISTER IN WHICH THE RESULT OF AN ARITHMETIC OR LOGIC OPERATION IS STORED.

ACM
ASSOCIATION FOR COMPUTING MACHINERY.

ACRONYM
A WORD FORMED FROM THE FIRST LETTER OR LETTERS OF THE SUCCESSIVE WORDS OF A MULTIPLE WORD TERM.

ADAPTIVE SYSTEMS
SYSTEMS DISPLAYING THE ABILITY TO LEARN, CHANGE THEIR STATE, OR OTHERWISE REACT TO A STIMULUS. ANY SYSTEM CAPABLE OF ADAPTING ITSELF TO CHANGES IN ITS ENVIRONMENT.

ADDRESS
A LABEL SUCH AS AN INTEGER OR OTHER SET OF CHARACTERS WHICH IDENTIFIES A LOCATION IN WHICH INFORMATION IS STORED.

ADDRESS INDEXING
THE PROCESS OF CHANGING AN ADDRESS IN A MACHINE-LANGUAGE COMPUTER INSTRUCTION BY ADDITION OF A QUANTITY HELD IN A SPECIAL REGISTER (INDEX REGISTER). THIS CHANGE IS DONE AUTOMATICALLY IN THE EXECUTION OF AN INSTRUCTION.

ALGOL
ALGORITHMIC LANGUAGE. A DATA PROCESSING LANGUAGE UTILIZING ALGEBRAIC SYMBOLS TO EXPRESS PROBLEM-SOLVING FORMULAE FOR MACHINE SOLUTION.

ALGORITHM
A DEFINITE STEP-BY-STEP RULE FOR CONSTRUCTING THE SOLUTION TO A PROBLEM OR FOR EVALUATING A FUNCTION. ALGORITHMS USUALLY LEAD TO A SOLUTION IF ONE IS POSSIBLE, ALTHOUGH TIME REQUIRED MAY BE LONG OR SHORT. SEE "HEURISTIC."

ALPHAMERIC
ALSO ALPHANUMERIC PERTAINING TO A CHARACTER SET THAT CONTAINS BOTH LETTERS AND NUMERALS, AND USUALLY OTHER CHARACTERS.

ALPHABETICALLY ORIENTED MACHINES
COMPUTERS HAVING INSTRUCTIONS AND MEMORIES ORGANIZED ESPECIALLY TO MANIPULATE ALPHABETIC CHARACTERS (INCLUDING NUMERALS).

ANALOGUE
THE USE OF PHYSICAL VARIABLES, SUCH AS DISTANCE OR ROTATION OR VOLTAGE, TO REPRESENT AND CORRESPOND WITH NUMERICAL VARIABLES THAT OCCUR IN A COMPUTATION; CONTRASTED WITH "DIGITAL."

ANALOG COMPUTER
A COMPUTER WHICH REPRESENTS NUMERICAL QUANTITIES AS ELECTRICAL AND PHYSICAL VARIABLES.

ANALYSIS
A SEPARATING OR BREAKING-UP OF A WHOLE INTO ITS PARTS SO AS TO FIND OUT THEIR NATURE, PROPORTION, FUNCTION, RELATIONSHIP, ETC.

APPLICATIONS PROGRAMMING
THE PREPARATION OF PROGRAMS FOR APPLICATION TO SPECIFIC PROBLEMS IN ORDER TO FIND SOLUTIONS; CONTRASTED WITH "SYSTEMS PROGRAMMING."

ARRAY
A SERIES OF ITEMS ARRANGED IN AN ORDERED, MEANINGFUL PATTERN.

ARITHMETIC UNIT
THE UNIT OF A COMPUTING SYSTEM THAT CONTAINS THE CIRCUITS THAT PERFORM ARITHMETIC OPERATIONS.

ARTIFICIAL INTELLIGENCE
REFERS TO THE PERFORMANCE BY A COMPUTER OF TASKS THAT HAVE HITHERTO REQUIRED THE APPLICATION OF HUMAN INTELLIGENCE.

ASSEMBLE
TO PREPARE A MACHINE-LANGUAGE PROGRAM FROM A SYMBOLIC PROGRAM BY SUBSTITUTING MACHINE CODES FOR SYMBOLIC CODES.

ASSEMBLER
ALSO ASSEMBLY PROGRAM A COMPUTER PROGRAM THAT OPERATES ON A SYMBOLIC LANGUAGE AS INPUT DATA TO PRODUCE MACHINE LANGUAGE INSTRUCTIONS WHICH CAN BE PROCESSED DIRECTLY BY THE COMPUTER.

ASSEMBLIES
COLLECTIONS OF PARTS INTO UNITS BY RELATING EACH PART TO ALL OTHERS IN THE UNIT ACCORDING TO SOME PLAN.

ASSEMBLER LANGUAGE
SEE ASSEMBLY LANGUAGE.

ASSEMBLY LANGUAGE
THE SYMBOLIC LANGUAGE WHICH IS THE INPUT TO AN ASSEMBLY PROGRAM.

ASSOCIATIVE MEMORY
A COMPUTER MEMORY IN WHICH INFORMATION IS LOCATED BY THE CONTENT OF SOME PART OF THE MEMORY. A "TAG" MIGHT BE ATTACHED TO EACH ITEM OF INFORMATION TO IDENTIFY IT. SYNONOMOUS WITH "CONTENT-ADDRESSABLE MEMORY."

AUTOMATIC COMPUTER
SEE "COMPUTER."

AUXILIARY (PERIPHERAL) EQUIPMENT
EQUIPMENT NOT ACTIVELY INVOLVED DURING THE PROCESSING OF DATA, SUCH AS INPUT/OUTPUT EQUIPMENT AND AUXILIARY STORAGE UTILIZING PUNCHED CARDS, MAGNETIC TAPES, DISKS OR DRUMS.

BCD
SEE BINARY CODED DECIMAL.

BUG
ANY MECHANICAL, ELECTRICAL, ELECTRONIC, OR PROGRAMMING DEFECT THAT INTERFERES WITH THE OPERATION OF THE COMPUTER OR THE SUCCESSFUL RUNNING OF A PROGRAM. USED SYNONYMOUSLY WITH ERROR AND MALFUNCTION.

BLOCK
A COLLECTION OR GROUP OF WORDS, RECORDS, OR CHARACTERS WHICH ARE HANDLED AS A SINGLE UNIT, ESPECIALLY WITH REFERENCE TO INPUT AND OUTPUT. A FILE STORAGE BLOCK IS OFTEN CALLED A PHYSICAL RECORD. ALSO, THE SET OF LOCATIONS OR TAPE POSITIONS IN WHICH A BLOCK OF WORDS, AS DEFINED ABOVE, IS STORED OR RECORDED.

BLOCK DIAGRAM
A DIAGRAM OF A SYSTEM, INSTRUMENT, COMPUTER OR PROGRAM IN WHICH SELECTED PORTIONS ARE REPRESENTEC BY ANNOTATED BOXES AND INTERCONNECTING LINES.

BOOLEAN ALGEBRA
THE SCIENCE OF SYMBOLS DENOTING LOGICAL PROPOSITIONS AND THEIR COMBINATIONS ACCORDING TO CERTAIN RULES WHICH CORRESPOND TC THE LAWS OF LOGIC. NAMED AFTER THE ENGLISH MATHEMATICIAN GEORGE BOOLE (1815-1864).

BRANCH (NOUN)
A SEQUENCE OF INSTRUCTIONS EXECUTED AS A RESULT OF A DECISION INSTRUCTION.
BRANCH (VERB)
TO DEPART FROM THE USUAL SEQUENCE OF EXECUTING INSTRUCTIONS IN A COMPUTER; SYNONYMOUS WITH JUMP OR TRANSFER.

BINARY
1. THE NUMBER REPRESENTATION SYSTEM WITH BASE OF TWO.
2. A CHARACTERISTIC OR PROPERTY INVOLVING A SELECTION, CHCICE OR CONDITION IN WHICH THERE ARE ONLY TWO POSSIBILITIES.

BINARY MACHINES
DIGITAL COMPUTERS IN WHICH NUMBERS ARE REPRESENTED IN THE BINARY (BASE 2) NUMBER SYSTEM. INFORMATION OTHER THAN NUMERIC IS ALSO REPRESENTED AS COMBINATIONS OF "BITS."

BINARY CODED DECIMAL
A TYPE OF NOTATION IN WHICH EACH DECIMAL DIGIT IS IDENTIFIED BY A GROUP OF BINARY ONES AND ZEROS.

BRANCHING
THE ACT OF EXECUTING A CONDITIONAL CHANGE OF ADDRESS.

BASE ADDRESS
A GIVEN ADDRESS FROM WHICH AN ABSOLUTE ADDRESS IS DERIVED BY COMBINATION WITH A RELATIVE ADDRESS.

BATCH PROCESSING
A SYSTEM APPROACH TO PROCESSING WHERE SIMILAR INPUT ITEMS ARE GROUPED FOR PROCESSING DURING THE MACHINE RUN.

BOOTSTRAP
A TECHNIQUE OR DEVICE DESIGNED TO BRING ITSELF INTO A DESIRED STATE BY MEANS OF ITS OWN ACTION, E.G., A MACHINE ROUTINE WHOSE FIRST FEW INSTRUCTIONS ARE SUFFICIENT TO BRING THE REST OF ITSELF INTO THE COMPUTER FROM AN INPUT DEVICE.

BIT
THE MOST BASIC UNIT OF INFORMATION, REPRESENTING A CHOICE BETWEEN TWO ALTERNATIVES, THAT CAN BE STORED OR HANDLED. A BINARY DIGIT; A ONE (1) OR ZERO(0), OR THE ELECTRICAL, MECHANICAL MAGNETIC, OR CHEMICAL REPRESENTATION OF EITHER IN AN AUTOMATIC COMPUTER.

BUFFER
A TEMPORARY STORAGE DEVICE USED TO COMPENSATE FOR A DIFFERENCE IN THE SPEED OF DATA FLOW OR THE OCCURRENCE OF EVENTS WHEN DATA IS BEING MOVED FROM ONE DEVICE TO ANOTHER.

BYTE
A CONTIGUOUS SET OF BINARY DIGITS OPERATED UPON AS A UNIT.

CACM
COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY.

CALL
THE TRANSFERRING OF CONTROL TO A SPECIFIED CLOSED SUBROUTINE.

CARD HOPPER
A DEVICE THAT HOLDS CARDS AND MAKES THEM AVAILABLE TO A CARD FEED MECHANISM. SYNONYMOUS WITH INPUT MAGAZINE. CONTRAST WITH CARD STACKER.

CARD IMAGE
A ONE-TO-ONE REPRESENTATION OF THE CONTENTS OF A PUNCHED CARD, E.G., A CARD IMAGE ON MAGNETIC TAPE.

CARD READER
A DEVICE WHICH SENSES AND TRANSLATES INTO INTERNAL FORM THE HOLES IN PUNCHED CARDS.

CARD STACKER
AN OUTPUT DEVICE THAT ACCUMULATES CARDS IN A DECK. CONTRAST WITH CARD HOPPER.

CARDS TO TAPE
PERTAINING TO EQUIPMENT OR METHODS THAT TRANSMIT DATA FROM PUNCHED CARDS TO MAGNETIC TAPE.

CARRIAGE RETURN
THE OPERATION THAT CAUSES THE NEXT CHARACTER TO BE PRINTED AT THE LEFT MARGIN.

CATALOG
THE DATA SET CONTAINING THE NAMES AND VOLUME IDENTIFICATION OF SELECTED DATA SETS, USED BY THE SYSTEM TO LOCATE DATA SETS SPECIFIED BY NAME ONLY.

CELL
A STORAGE CELL OF 1 BINARY DIGIT CAPACITY, E.G., A SINGLE BIT REGISTER.

CHANNEL
A PATH ALONG WHICH SIGNALS CAN BE SENT, E.G., DATA CHANNEL, OUTPUT CHANNEL.

CHARACTER SET
A LIST OF CHARACTERS ACCEPTABLE FOR CODING TO A SPECIFIC COMPUTER OR INPUT/OUTPUT DEVICE.

CHECK BIT
A BINARY CHECK DIGIT, OFTEN A PARITY BIT.

CHECK DIGIT
ONE OR MORE DIGITS CARRIED IN A SYMBOL OR A WORD DEPENDENT UPON THE REMAINING DIGITS IN SUCH A FASHION THAT IF A SINGLE ERROR OCCURS EXCLUDING COMPENSATING ERRORS, THE ERROR WILL BE REPORTED.

CLOSED SUBROUTINE
A SUBROUTINE THAT CAN BE STORED AT ONE PLACE AND CAN BE CONNECTED TO A ROUTINE BY LINKAGES AT ONE OR MORE LOCATIONS. CONTRAST WITH OPEN SUBROUTINE.

CODE
'1 A SYSTEM OF SYMBOLS FOR REPRESENTING DATA OR INSTRUCTIONS IN A COMPUTER OR A TABULATING MACHINE OR, 2 TO WRITE A PROGRAM OR PART OF A PROGRAM FOR THE SOLUTION OF A PROBLEM BY A COMPUTER.

COLD START
NO PROGRAMS STORED ON SPOOL PACK.

COLLATING SEQUENCE
THE SEQUENCE INTO WHICH THE ALLOWABLE CHARACTERS OF A COMPUTER ARE RANKED OR ORDERED.

COLUMN
'1 A VERTICAL ARRANGEMENT OF CHARACTERS OR OTHER EXPRESSIONS, OR 2 LOOSELY, A DIGIT PLACE.

COLUMN BINARY
PERTAINING TO THE BINARY REPRESENTATION OF DATA ON PUNCHED CARDS IN WHICH ADJACENT POSITIONS IN A COLUMN CORRESPOND TO ADJACENT BITS OF DATA, E.G., EACH COLUMN IN A 12-ROW CARD MAY BE USED TO REPRESENT 12 CONSECUTIVE BITS OF A 36-BIT WORD.

COMMAND
A CONTROL SIGNAL.

COMPILE
TO PREPARE A MACHINE LANGUAGE PROGRAM FROM A COMPUTER PROGRAM WRITTEN IN ANOTHER PROGRAMMING LANGUAGE.

CONTROL CHARACTER
A CHARACTER WHOSE OCCURRENCE IN A PARTICULAR CONTEXT INITIATES, MODIFIES, OR STOPS A CONTROL OPERATION, E.G., A CHARACTER TO CONTROL CARRIAGE RETURN.

CONTROL FIELD
A CONSTANT LOCATION WHERE INFORMATION FOR CONTROL PURPOSES IS PLACED.

CONTROL UNIT
THE PORTION OF A COMPUTER WHICH DIRECTS THE SEQUENCE OF OPERATIONS, INTERPRETS THE CODED INSTRUCTIONS, AND INITIATES THE PROPER COMMANDS TO THE COMPUTER CIRCUITS PREPARATORY TO EXECUTION.

CONVERSATIONAL MODE

A MODE OF COMPUTER OPERATION WHERE TWO-WAY COMMUNICATION IS MAINTAINED BETWEEN THE USER AND THE MACHINE, AS OPPOSED TO A TYPE OF PROCESSING WHERE THE COMPUTER IS TOLD IN ADVANCE PRECISELY WHAT IS TO BE DONE.

CONVERT

TO CHANGE THE REPRESENTATION OF DATA FROM ONE FORM TO ANOTHER, E.G., TO CHANGE NUMERICAL DATA FROM BINARY TO DECIMAL OR TRANSFER INFORMATION FROM CARDS TO TAPE.

CENTRAL PROCESSING UNIT (CPU)

THE UNIT OF A COMPUTING SYSTEM THAT CONTAINS THE CIRCUITS THAT CALCULATE AND PERFORM LOGIC DECISIONS BASED ON A MAN-MADE PROGRAM OF OPERATING INSTRUCTIONS.

CHARACTER

ONE OF A SET OF ELEMENTARY SYMBOLS ACCEPTABLE TO A DATA PROCESSING SYSTEM FOR READING, WRITING, OR STORING.

CHARACTER RECOGNITION

THE TECHNIQUE OF READING, IDENTIFYING AND ENCODING A PRINTED CHARACTER BY OPTICAL MEANS.

CITATION INDEX

AN INDEX USING FOOTNOTE REFERENCES IN DOCUMENTS AS COUPLING MECHANISMS AMONG RELATED PAPERS. USED PARTICULARLY IN LAW, E.G. SHEPARD'S CITATIONS.

CLEAR

TO PUT A STORAGE OR MEMORY DEVICE INTO A STATE DENOTING ZERO OR BLANK.

CLOSED-SHOP OPERATION

THAT MODE OF OPERATING A COMPUTING CENTER IN WHICH ALL MACHINE OPERATION IS DONE BY MEMBERS OF A SPECIALIZED GROUP WHOSE ONLY PROFESSIONAL CONCERN IS THE CONTROL AND MANIPULATION OF COMPUTERS. SEE "OPEN-SHOP PROGRAMMING."

COBOL*

COMMON BUSINESS ORIENTED LANGUAGE. A DATA PROCESSING LANGUAGE THAT RESEMBLES BUSINESS ENGLISH.

CODING

THE ACT OF SPECIFYING A PROBLEM-SOLVING PROCEDURE BY A SEQUENCE OF INSTRUCTIONS FOR THE OPERATIONS TO BE PERFORMED BY

THE COMPUTER. SUCH INSTRUCTIONS MAY BE IN MACHINE LANGUAGE OR MAY BE COMPILED INTO MACHINE LANGUAGE. SEE "PROGRAMMING."

COLLATOR
A DEVICE TO COLLATE OR MERGE SETS OF CARDS INTO A NEW SEQUENCE.

COMMAND LANGUAGE
A LANGUAGE BY MEANS OF WHICH CONTROL IS EXERCISED OVER A COMPLEX SYSTEM OF EQUIPMENT AND PROGRAMS. SEE "EXECUTIVE SYSTEMS."

COMPILER
A PROGRAM ENABLING THE COMPUTER FOR WHICH IT IS DESIGNED TO ACCEPT PROGRAMS IN PROCEDURE-ORIENTED OR PROBLEM-ORIENTED LANGUAGE AND TO TRANSFORM THEM INTO MACHINE-LANGUAGE PROGRAMS.

COMPILING
THE PROCESS OF TRANSFORMING A PROGRAM IN A SOURCE LANGUAGE INTO A PROGRAM IN MACHINE LANGUAGE.

COMPUTER
ANY MACHINE CAPABLE OF ACCEPTING INFORMATION, PERFORMING NUMERICAL AND LOGICAL MANIPULATIONS, AND DISPLAYING THE RESULTS; AN AUTOMATIC COMPUTER IS ONE WHICH PERFORMS SEQUENCES OF OPERATIONS ON THE BASIS OF INITIALLY STORED INSTRUCTIONS. THROUGHOUT THIS REPORT, "COMPUTER" IS ALWAYS USED IN THE SPECIAL SENSE OF "DIGITAL COMPUTER." SEE "DIGITAL COMPUTER."

CONSOLE
THE UNIT OF EQUIPMENT USED FOR COMMUNICATION BETWEEN THE OPERATOR OR SERVICE ENGINEER AND THE COMPUTER.

CONTENT-ADDRESSABLE MEMORY
A COMPUTER MEMORY IN WHICH INFORMATION IS LOCATED BY THE CONTENT OF SOME PART OF THE MEMORY. FOR EXAMPLE, SUCH A MEMORY MIGHT CONTAIN A TABLE OF THE NAMES OF QUANTITIES TOGETHER WITH ADDRESSES SPECIFYING WHERE THE VALUES OF THESE QUANTITIES ARE TO BE FOUND. SEE "ASSOCIATIVE MEMORY."

CORE STORAGE
A FORM OF MAGNETIC STORAGE THAT PERMITS HIGH-SPEED ACCESS TO INFORMATION WITHIN THE COMPUTER. SEE MAGNETIC CORE.

COUPLING

AN INTERACTION BETWEEN SYSTEMS OR BETWEEN PROPERTIES OF A SYSTEM.

CRITICAL PATH METHOD

A MANAGEMENT-CONTROL TOOL FOR EVALUATING A SEQUENCED PLAN FOR ACHIEVING AN OBJECTIVE. EMPHASIS IS PLACED ON THE "CRITICAL PATH" WHICH IS THAT SEQUENCE WHICH ACTS AS A BOTTLENECK AND DELIMITS THE OPPORTUNITY TO ACHIEVE THE OBJECTIVE. SEE "PERT."

CRYOGENIC DEVICE

ANY DEVICE CAPABLE OF HIGH-SPEED SWITCHING, BY VIRTUE OF SUPERCONDUCTIVITY AND VERY LOW THERMAL NOISE AT TEMPERATURES NEAR ABSCLUTE ZERO.

CRYOTRON

AN ELECTRIC-SWITCHING AND BINARY MEMORY-STORAGE DEVICE UTILIZING THE FACT THAT A MAGNETIC FIELD CAN CAUSE A SUPERCONDUCTING ELEMENT TO BE IN EITHER A STATE OF LOW OR HIGH RESISTANCE.

CYBERNETICS

THE THEORY OF CONTROL AND COMMUNICATION IN THE MACHINE AND THE ANIMAL.

DASDI

DIRECT ACCESS DEVICE INITIATION.

DATA CELL

THE STORAGE FOR ONE UNIT OF INFORMATION, USUALLY ONE CHARACTER OR ONE WORD.

DATA DEFINITION

A JOB CONTROL STATEMENT THAT DESCRIBES A DATA SET ASSOCIATED WITH A PARTICULAR JOB STEP.

DCB

DATA CONTROL BLOCK. A SYSTEM CONTROL BLOCK THROUGH WHICH THE INFORMATION REQUIRED BY ACCESS ROUTINES TO STORE AND RETRIVE DATA IS COMMUNICATED TO THEM.

DEBE

DATA EXTENDED BINARY CODED DECIMAL INTERCHANGE CODE.

DECK
A COLLECTION OF PUNCHED CARDS.

DECOLLATOR
A DEVICE USED FOR THE AUTOMATIC REMOVAL OF CARBON PAPER FROM PRINTED FORMS.

DELIMITER
A FLAG THAT SEPARATES AND ORGANIZES ITEMS OF DATA. SYNONYMOUS WITH SEPARATOR.

DIGIT
A CHARACTER USED TO REPRESENT ONE OF THE NON-NEGATIVE INTEGERS SMALLER THAN THE RADIX, E.G., IN DECIMAL NOTATION, ONE OF THE CHARACTFRS 0 TO 9.

DISPLAY
A VISUAL PRESENTATION OF DATA.

DUMMY
PERTAINING TO THE CHARACTERISTIC OF HAVING THE APPEARANCE OF A SPECIFIED THING BUT NOT HAVING THE CAPACITY TO FUNCTION AS SUCH.

DATA PROCESSING SYSTEM
A NETWORK OF MACHINE COMPONENTS CAPABLE OF ACCEPTING INFORMATION, PROCESSING IT ACCORDING TO MAN-MADF INSTRUCTIONS, AND PRODUCING THE COMPUTED RESULTS.

DEBUGGING
THE PROCESS OF ISOLATING AND REMOVING MALFUNCTIONS FROM A COMPUTER OR FROM A COMPUTER PROGRAM; A PROCEDURE TO ESTABLISH PROGRAM ACCURACY BY RUNNING THE PROGRAM WITH SELECTIVE DATA TO FIND LOGICAL OR CLERICAL "BUGS" OR ERRORS.

DEBUGGING SYSTEMS
PROGRAMS OR OTHER SYSTEMS WHICH ASSIST THE PROGRAMMER IN DETECTING AND CORRECTING ERRORS. SEE "DEBUGGING."

DIAGNOSTIC PROGRAMS
COMPUTER PROGRAMS USED IN THE DETECTION AND ISOLATION OF EITHER HARDWARE OR SOFTWARE DIFFICULTIES.

DIGITAL
THE QUALITY OF USING NUMBERS EXPRESSED IN DIGITS AND IN A SCALE OF NOTATION.

DIGITAL COMPUTER
A COMPUTER THAT PERFORMS MATHEMATICAL AND LOGICAL OPERATIONS WITH INFORMATION, NUMERICAL OR OTHERWISE, REPRESENTED IN DIGITAL FORM.

DIGITAL DATA
INFORMATION EXPRESSED IN DISCRETE SYMBOLS.

DIGITIZER
A DEVICE TO CONVERT INFORMATION FROM ANALOGUE FORM TO DIGITAL FORM.

DIODE
AN ELECTRONIC DEVICE USED TO PERMIT CURRENT FLOW IN ONE DIRECTION AND TO INHIBIT CURRENT FLOW IN THE OPPOSITE DIRECTION.

DIRECT ACCESS
SEE RANDOM ACCESS.

DISK STORAGE
A METHOD OF STORING INFORMATION IN CODE, MAGNETICALLY, IN QUICKLY ACCESSIBLE SEGMENTS ON FLAT ROTATING DISKS.

DOWNTIME
THE ELAPSED TIME WHEN A COMPUTER IS NOT OPERATING CORRECTLY BECAUSE OF MACHINE OR PROGRAM MALFUNCTION.

DRUM STORAGE
A METHOD OF STORING INFORMATION IN CODE, MAGNETICALLY, ON THE SURFACE OF A ROTATING CYLINDER.

DUMP
TO COPY THE CONTENTS OF ALL OR PART OF A STORAGE, USUALLY FROM A CENTRAL PROCESSING UNIT INTO AN EXTERNAL STORAGE.

EBCDIC
EXTENDED BCD INTERCHANGE CODE, THE PRIMARY CHARACTER SET USED BY THE 360.

EDIT
TO MODIFY THE FORM OR FORMAT OF DATA, E.G., TO INSERT OR DELETE CHARACTERS SUCH AS PAGE NUMBERS OR DECIMAL POINTS.

ENTRY POINT
IN A ROUTINE, ANY PLACE TO WHICH CONTROL CAN BE PASSED.

ELECTRONIC RECORDING AND MACHINE ACCOUNTING
A SYSTEM DEVELOPED AT STANFORD RESEARCH INSTITUTE WHICH ESTABLISHED THE FEASIBILITY OF MAGNETIC-INK CHARACTER RECOGNITION AND THE USE OF COMPUTERS IN BANKING INSTITUTIONS WHERE THEY PERFORM ALL ROUTINE BOOKKEEPING FUNCTIONS FOR CHECKING ACCOUNTS.

EXECUTE ✓
TO PERFORM A DATA PROCESSING ROUTINE OR PROGRAM, BASED ON MACHINE-LANGUAGE INSTRUCTIONS.

EXECUTIVE SYSTEMS
SYSTEMS OF COMPUTER PROGRAMS DESIGNED TO PROCESS AND TO CONTROL THE EXECUTION OF OTHER PROGRAMS.

EXTRAPOLATION
AN ESTIMATE OR INFERENCE OF A VALUE BEYOND THE KNOWN RANGE FROM WHICH THE ESTIMATED VALUE IS ASSUMED TO FOLLOW.

FIELD
IN A RECORD, A SPECIFIED AREA USED FOR A PARTICULAR CATEGORY OF DATA, E.G., A GROUP OF CARD COLUMNS OR A SET OF BIT LOCATIONS IN A COMPUTER WORD.

FIXED POINT
PERTAINING TO A NUMERATION SYSTEM IN WHICH THE POSITION OF THE POINT IS FIXED WITH RESPECT TO ONE END OF THE NUMERALS, ACCORDING TO SOME CONVENTION.

FLAG
1 ANY OF VARIOUS TYPES OF INDICATORS USED FOR IDENTIFICATION, OR 2 A CHARACTER THAT SIGNALS THE OCCURRENCE OF SOME CONDITION.

FORMAT
FORM, USUALLY REFERRING TO THE ARRANGEMENT OF INPUT OR OUTPUT DATA OR LINES OF PRINT. SEE FORMAT STATEMENTS IN LANGUAGE MANUALS.

FERRITES
A CLASS OF NON-METALLIC SUBSTANCES CONTAINING IRON, OXYGEN, AND OTHER METALS. THESE MATERIALS HAVE FERROMAGNETIC PROPERTIES AND ARE POOR CONDUCTORS OF ELECTRICITY. THIS MAKES THEM USEFUL IN MANY APPLICATIONS WHERE ORDINARY FERROMAGNETIC MATERIALS (WHICH ARE GOOD ELECTRICAL CONDUCTORS) WOULD CAUSE TOO MUCH LOSS OF ELECTRICAL ENERGY.

FILE
A COLLECTION OF RELATED RECORDS; E.G., IN INVENTORY CONTROL, ONE LINE OF AN INVOICE FORMS AN ITEM, A COMPLETE INVOICE FORMS A RECORD, AND THE COMPLETE SET OF SUCH RECORDS FORMS A FILE.

FILE MAINTENANCE
THE PROCESSING OF INFORMATION IN A FILE TO KEEP IT UP TO DATE.

FLIP-FLOP
A CIRCUIT OR DEVICE CONTAINING ACTIVE ELEMENTS CAPABLE OF ASSUMING EITHER ONE OF TWO STABLE STATES AT A GIVEN TIME.

FLOATING-POINT ARITHMETIC
A TECHNIQUE PERMITTING ARITHMETIC OPERATION ON NUMBERS IN WHICH THE LOCATIONS OF THE DECIMAL POINTS ARE NOT UNIFORM.

FLOW CHART
A DIAGRAMMATIC REPRESENTATION OF THE SEQUENCE OF CHOICES AND ACTIONS IN A COMPLICATED ACTIVITY.

FORMAL
MECHANICAL, METHODICAL, OR DETERMINISTIC IN CHARACTER.

FORMAL LANGUAGE
A SYSTEM CONSISTING OF A WELL-DEFINED, USUALLY FINITE, SET OF CHARACTERS AND RULES FOR COMBINING CHARACTERS WITH ONE ANOTHER TO FORM WORDS OR OTHER EXPRESSIONS BUT WITHOUT ASSIGNMENT OF PERMANENT MEANING TO SUCH WORDS OR EXPRESSIONS.

FORTRAN
FORMULA TRANSLATING SYSTEM. A DATA PROCESSING LANGUAGE THAT CLOSELY RESEMBLES ALGEBRAIC NOTATION.

GENERATE
TO PRODUCE A PROGRAM BY SELECTION OF SUBSETS FROM A SET OF SKELETAL CODING UNDER THE CONTROL OF PARAMETERS.

GENERAL-PURPOSE
BEING APPLICABLE TO A WIDE VARIETY OF USES WITHOUT ESSENTIAL MODIFICATION. CONTRASTED WITH "SPECIAL-PURPOSE."

HASP
HOUSTON AUXILARY SPOOLING PRIORITY, A JOB SCHEDULING PROGRAM.

HOLLERITH CODE
AN ALPHA-NUMERIC PUNCHED-CARD CODE INVENTED BY DR. HERMAN HOLLERITH IN 1889, IN WHICH THE TOP THREE POSITIONS IN A COLUMN ARE CALLED ZONE PUNCHES 12, 11, AND 0 FROM THE TOP DOWNWARD, AND ARE COMBINED WITH THE REMAINING DIGIT PUNCHES 1 THROUGH 9 TO REPRESENT ALPHABETIC, NUMERIC, AND SPECIAL CHARACTERS.

HARD COPY
A PRINTED COPY OF MACHINE OUTPUT, E.G., PRINTED REPORTS, LISTINGS, DOCUMENTS, ETC.

HARDWARE
THE MECHANICAL, MAGNETIC, ELECTRICAL, AND ELECTRONIC DEVICES FROM WHICH A COMPUTER IS CONSTRUCTED.

HEURISTIC
A TECHNIQUE BY MEANS OF WHICH AN INDIVIDUAL (OR MACHINE) CAN BE ORGANIZED TO SOLVE PROBLEMS. WHEN APPLICABLE, IT MAY PROVIDE A SHORTCUT TO THE GOAL BUT CANNOT GUARANTEE A SOLUTION OR AN OPTIMAL SOLUTION. SEE "ALGORITHM."

HOUSEKEEPING
OPERATIONS IN A ROUTINE WHICH DO NOT CONTRIBUTE DIRECTLY TO THE SOLUTION OF A PROBLEM BUT DO CONTRIBUTE DIRECTLY TO THE EXECUTION OF A PROGRAM BY THE COMPUTER.

IDENTIFIER
A SYMBOL WHOSE PURPOSE IS TO IDENTIFY, INDICATE, OR NAME A BODY OF DATA.

INDEX
AN ORDERED REFERENCE LIST OF THE CONTENTS OF A FILE OR DOCUMENT, TOGETHER WITH KEYS OR REFERENCE NOTATIONS FOR IDENTIFICATION OR LOCATION OF THOSE CONTENTS, OR 2 A SYMBOL OR A NUMBER USED TO IDENTIFY A PARTICULAR QUANTITY IN AN ARRAY OF SIMILAR QUANTITIES.

INDEXED SEQUENTIAL
A DATA SET WHOSE RECORDS ARE ORGANIZED ON THE BASIS OF A COLLATING SEQUENCE DETERMINED BY KEYS THAT PRECEDE EACH BLOCK OF DATA.

INITIALIZE
TO SET COUNTERS, SWITCHES, AND ADDRESSES TO ZERO OR OTHER STARTING VALUES AT THE BEGINNING OF, OR AT PRESCRIBED POINTS IN, A COMPUTER ROUTINE.

INTERFACE
A SHARED BOUNDARY.

INTERPRETER
A DEVICE THAT PRINTS ON A PUNCHED CARD THE DATA ALREADY PUNCHED IN THE CARD.

INTERRUPT ROUTINE
A SECONDARY PROGRAM THAT TAKES ACTION FOLLOWING A TRANSFER FROM THE MAIN PROGRAM.

ITEM
A COLLECTION OF RELATED CHARACTERS, TREATED AS A UNIT. CONTRAST WITH FILE.

IDLE TIME
THE TIME THAT A COMPUTER IS AVAILABLE FOR USE, BUT IS NOT IN OPERATION.

INDEX REGISTER
A REGISTER CONTAINING A QUANTITY WHICH MAY BE USED FOR MODIFYING THE ADDRESS PART OF AN INSTRUCTION, FOR COUNTING, OR FOR OTHER PURPOSES AS DIRECTED BY THE PROGRAM.

INFORMATION PROCESSING
THE STORAGE, RETRIEVAL, ARRANGEMENT, SELECTION OR TRANSFORMATION OF INFORMATION. GENERALLY APPLIED TO CARRYING OUT SUCH OPERATIONS WITH THE AID OF A MODERN DIGITAL COMPUTER.

INFORMATION RETRIEVAL
THE LOCATION AND SELECTION, ON DEMAND, OF DOCUMENTATION OR GRAPHIC RECORDS RELEVANT TO A GIVEN INFORMATION REQUIREMENT FROM A FILE OF SUCH MATERIAL.

INFORMATION STORAGE AND RETRIEVAL
THE ARRANGEMENT OF DOCUMENTATION OR RECORDS IN A SYSTEMATIC WAY TOGETHER WITH THEIR SUBSEQUENT LOCATION AND SELECTION ON DEMAND. SEE "INFORMATTON RETRIEVAL."

INPUT-OUTPUT
INPUT: INFORMATION READ INTO THE COMPUTER FROM THE "OUTSIDE WORLD." OUTPUT: INFORMATION TRANSFERRED FROM THE COMPUTER TO THE "OUTSIDE WORLD." AS AN ADJECTIVE, "INPUT" PERTAINS TO THE DEVICES WHICH BRING INFORMATION INTO THE COMPUTER, AND "OUTPUT" ANALOGOUSLY.

INQUIRY
A REQUEST FOR INFORMATION FROM STORAGE, E.G., A REQUEST FOR THE NUMBER OF AVAILABLE AIRLINE SEATS.

INSTRUCTION
A STATEMENT THAT CALLS FOR A SPECIFIC COMPUTER OPERATION.

INTERLEAVE
TO INSERT SEGMENTS OF ONE PROGRAM INTO ANOTHER PROGRAM SO THAT DURING PROCESSING DELAYS IN ONE PROGRAM, PROCESSING CAN CONTINUE ON SEGMENTS OF ANOTHER PROGRAM; A TECHNIQUE USED IN MULTIPROGRAMMING.

INTERRUPT
A BREAK IN THE NORMAL FLOW OF PROCESSING. THE NORMAL JOB FLOW CAN BE RESUMED FROM THAT POINT AT A LATER TIME. AN INTERRUPT IS USUALLY CAUSED BY A SIGNAL FROM AN EXTERNAL SOURCE, E.G., A TERMINAL UNIT.

ITERATE
TO REPEAT, AUTOMATICALLY, UNDER PROGRAM CONTROL, THE SAME SERIES OF PROCESSING STEPS UNTIL A PREDETERMINED STOP OR BRANCH CONDITION IS REACHED; TO LOOP.

JACM
JOURNAL OF THE ASSOCIATION FOR COMPUTING MACHINERY.

JOB
A UNIT OF COMPUTER DATA-PROCESSING WORK. USUALLY DEFINED AS THE PROCESSING OF A SINGLE USER S PROGRAM.

JOB CONTROL LANGUAGE
STATEMENTS OF A SPECIFIC FORMAT WHICH AFFECT PROCESSING USUALLY IN THE FORM OF PROGRAM CONTROL CARDS.

KEYPUNCH
A KEYBOARD-ACTUATED DEVICE THAT PUNCHES HOLES IN A CARD TO REPRESENT DATA.

KILOMEGACYCLE
A BILLION CYCLES PER SECOND. A REPETITION RATE IN WHICH AN EVENT IS REPEATED A BILLION TIMES PER SECOND.

KINETICS
THE SCIENCE OF THE RATE OF CHEMICAL REACTIONS. THE BRANCH OF DYNAMICS DEALING WITH THE CHANGES OF MOTION PRODUCED BY FORCES.

LABEL
A KEY ATTACHED TO THE ITEM OF DATA THAT IT IDENTIFIES.

LCS
LARGE CORE STORAGE. A LOW COST AUXILIARY BULK STORAGE UNIT DESIGNED FOR USE WITH THE IBM SYSTEM/360.

LIBRARY
A COLLECTION OF ORGANIZED INFORMATION USED FOR STUDY AND REFERENCE. SEE PROGRAM LIBRARY.

LINKAGE
THE MEANS BY WHICH COMMUNICATION IS EFFECTED BETWEEN TWO ROUTINES OR CONTROL SECTIONS.

LANGUAGE
A MEANS OF COMMUNICATION BY MEANS OF EXPRESSIONS. SPECIFICALLY, ANY MEANS OF COMMUNICATING INSTRUCTIONS AND DATA TO AND FROM A COMPUTER USING SYMBOLS OR PATTERNS PERCEPTIBLE TO BOTH THE HUMAN AND THE MACHINE.

LIBRARY ROUTINE
A SPECIAL-PURPOSE PROGRAM WHICH MAY BE MAINTAINED IN STORAGE FOR USE WHEN NEEDED.

LIST PROCESSING
THE PROCESSING OF INFORMATION ORGANIZED IN LISTS (I.E., ORGANIZATIONS IN WHICH EACH ELEMENT IDENTIFIES ONE OR MORE SUCCEEDING ELEMENTS).

LIST STRUCTURES

AN ELEMENT OF A LIST MAY, ITSELF BE THE NAME OF A LIST. THE MORE COMPLEX ORGANIZATIONS THAT CAN BE CONSTRUCTED BY HAVING THE NAMES OF LISTS AS ELEMENTS OF OTHER LISTS ARE CALLED LIST STRUCTURES. SEE "LIST PROCESSING."

LOAD

TO PLACE DATA INTO MAIN CORE STORAGE.

LOOP

SEE ITERATE.

MACHINE INSTRUCTION

AN INSTRUCTION THAT A MACHINE CAN RECOGNIZE AND EXECUTE.

MATRAN

MATRIX TRANSLATION. A TYPE OF MATHEMATICAL PROGRAMMING LANGUAGE.

MATRIX

A RECTANGULAR ARRAY OF NUMBERS SUBJECT TO MATHEMATICAL OPERATIONS, SUCH AS ADDITION, MULTIPLICATION, AND INVERSION, ACCORDING TO SPECIFIED RULES. BY EXTENSION, AN ARRAY OF ANY NUMBER OF DIMENSIONS.

MEMBER

AN ENTITY WITHIN A DIRECTORIED DATA SET, INDEXED IN THE DATA SET S DIRECTORY AND HAVING DATA CONTENT.

MFT

MULTIPROGRAMMING WITH FIXED NUMBER OF TASKS.

MODULE

1 A SEGMENT OF CORE STORAGE, OR 2 A PIECE OF PERIPHERAL EQUIPMENT WITH A SPECIFIC STORAGE CAPACITY, E.G., A DISK MODULE.

MULTIPLEX

TO INTERLEAVE OR SIMULTANEOUSLY TRANSMIT TWO OR MORE MESSAGES ON A SINGLE CHANNEL.

MVT

MULTIPROGRAMMING WITH VARIABLE NUMBER OF TASKS.

MACHINE-INDEPENDENT
HAVING A FORM WHICH DOES NOT DEPEND ON THE PECULIARITIES OF ANY COMPUTER OR CLASS OF COMPUTERS.

MACHINE LANGUAGE
A VOCABULARY OF "WORDS" MEANINGFUL TO A COMPUTER; A STRING OF DIGITS ACCEPTABLE TO AND MANIPULATABLE BY MACHINE CIRCUITS; TRAINS OF ELECTRICAL PULSES SETTING AND RESETTING COMPUTER CIRCUITS OR MEMORY.

MACHINE-READABLE FORM
A FORM IN WHICH INFORMATION IS ACCEPTABLE TO A MACHINE. FOR EXAMPLE, PUNCHED CARDS OR MAGNETIC TAPE CAN CONTAIN INFORMATION IN MACHINE-READABLE FORM, WHEREAS HANDWRITING USUALLY DOES NOT.

MACRO INSTRUCTION
A SINGLE INSTRUCTION THAT CAUSES THE COMPUTER TO EXECUTE A PREDETERMINED SEQUENCE OF MACHINE INSTRUCTIONS.

MAGNETIC
OF, PRODUCING, CAUSED BY, OR OPERATED BY MAGNETISM.

MAGNETIC CORE
A DOUGHNUT-SHAPED PIECE OF FERRITE WHICH CAN BE MAGNETIZED IN EITHER A POSITIVE (CLOCKWISE) OR NEGATIVE (COUNTER-CLOCKWISE) SENSE AND SO CAN RECORD A "BIT." RECTANGULAR ARRAYS OF MAGNETIC CORES SITUATED ON THE INTERSECTIONS OF HORIZONTAL AND VERTICAL SETS OF WIRES FORM A "CORE PLANE." THE MAGNETIC STATE OF AN INDIVIDUAL CORE CAN BE CONTROLLED AND TESTED BY SELECTING THE HORIZONTAL WIRE AND THE VERTICAL WIRE THAT INTERSECT AT THAT CORE.

MAGNETIC DISKS
THIN, FLAT, CIRCULAR OBJECTS COATED WITH MAGNETIZABLE MATERIAL SO THAT DIGITAL RECORDINGS CAN BE MADE THEREON. CHARACTERISTICALLY SUCH DISKS PROVIDE HIGH DENSITY OF RECORDING PER UNIT VOLUME WITH RELATIVELY SHORT ACCESS TIMES TO THE INFORMATION RECORDED.

MAGNETIC-INK CHARACTER RECOGNITION
THE PROCESS OF MECHANICALLY RECOGNIZING CHARACTERS WHICH ARE RECORDED IN MAGNETIZABLE INK. PRINTED CHARACTERS, WHEN MAGNETIZED, CAN BE RECOGNIZED BY THE UNIQUE PATTERNS OF MAGNETIC INDUCTION CREATED AS THE PRINTED PATTERNS PASS A MAGNETIC-READING HEAD.

MAGNETIC-READING HEAD
AN ELECTROMAGNET USED FOR CONVERTING ELECTRICAL SIGNALS INTO A MAGNETIC RECORDING, CONVERTING A MAGNETIC RECORDING INTO ELECTRICAL SIGNALS, OR ERASING A MAGNETIC RECORDING; FOR INSTANCE ON A MAGNETIC DISK.

MAGNETIC RESONANCE
THE PHENOMENON IN WHICH A MOVEMENT OF A PARTICLE OR SYSTEM OF PARTICLES IS COUPLED RESONANTLY TO AN EXTERNAL MAGNETIC FIELD.

MAGNETIC TAPE
A PLASTIC TAPE WITH A MAGNETIC SURFACE ON WHICH DATA CAN BE STORED IN A CODE OF MAGNETIZED SPOTS.

MAGNETIC THIN-FILM
A LOGIC OR STORAGE ELEMENT COATED WITH AN EXTREMELY THIN LAYER OF MAGNETIC MATERIAL, USUALLY LESS THAN ONE MICRON THICK (ABOUT FOUR HUNDRED-THOUSANDTHS OF AN INCH).

MARK-SENSE
TO MARK A POSITION ON A CARD OR PAPER FORM WITH A PENCIL. THE MARKS ARE THEN INTERPRETED ELECTRICALLY FOR MACHINE PROCESSING.

MATHEMATICAL MODEL
A SET OF MATHEMATICAL EXPRESSIONS THAT DESCRIBES SYMBOLICALLY THE OPERATION OF A PROCESS, DEVICE OR CONCEPT.

MEMORY
THE TERM "STORAGE" IS PREFERRED BY ALL ANTIANTHROPOMORPHISTS BUT "MEMORY" PERSISTS. IT REFERS TO THE CAPACITY OF A COMPUTER TO STORE INFORMATION SUBJECT TO RECALL, OR TO THE COMPONENT OF THE COMPUTER SYSTEM IN WHICH SUCH INFORMATION IS STORED.

MEMCRY PROTECTION
A SYSTEM OR DEVICE WHICH ASSURES THAT INFORMATION RECORDED CANNOT BE REPLACED, EITHER INADVERTENTLY OR INTENTIONALLY, BY INFORMATION OTHER THAN THAT INTENDED BY AN EXECUTIVE SYSTEM.

MICROELECTRONICS
THAT FIELD OF ELECTRONICS DEALING WITH THE MINIATURIZATION OF CIRCUITS BY THE COMBINATION OF A NUMBER OF ELEMENTARY CIRCUITS INTO A COMPOSITE.

MICROSECOND
A MILLIONTH PART OF A SECOND.

MILLISECOND
A THOUSANDTH PART OF A SECOND.

MNEMONIC CODE
ASSEMBLY LANGUAGE CODE WHICH IS EASY FOR THE PROGRAMMER TO REMEMBER BECAUSE OF ITS MNEMONIC NATURE, E.G., MPY FOR MULTIPLY AND ACC FOR ACCUMULATOR.

MONITOR
A SYSTEM THAT REMINDS, CAUTIONS, OR WARNS ONE OF SITUATIONS THAT CAN INTERFERE WITH THE PROPER EXECUTION OF INTENDED ACTIVITIES.

MONTE CARLO METHODS
METHODS OF COMPUTATION BASED ON PROBABILITY THEORY THAT USE RANDOM NUMBERS AND STATISTICAL METHODS TO FIND SOLUTIONS TO VARIOUS TYPES OF PROBLEMS.

MULTI-PROCESSING
TO PROCESS MULTIPLE REQUIREMENTS CONCURRENTLY ON A SYSTEM SO THAT EACH REQUIREMENT IS SATISFIED SEPARATELY.

MULTIPROCESSOR
A MACHINE WITH MULTIPLE ARITHMETIC, LOGIC AND MAIN STORAGE UNITS THAT CAN BE USED SIMULTANEOUSLY ON MORE THAN ONE PROBLEM.

MULTIPROGRAMMING
SEE INTERLEAVE.

NONDESTRUCTIVE READ
A READ PROCESS THAT DOES NOT ERASE THE DATA IN THE SOURCE.

NUMBER SYSTEM
A SYSTEM FOR THE REPRESENTATION OF NUMBERS, E.G., THE DECIMAL SYSTEM, THE ROMAN NUMERAL SYSTEM, THE BINARY SYSTEM.

NANCSECOND
A BILLIONTH PART OF A SECOND.

NETS
SYSTEMS OF INTERCONNECTED POINTS TO WHICH FORMAL RELATIONSHIPS CAN BE APPLIED.

NUMERICAL ANALYSIS
THAT BRANCH OF MATHEMATICAL ANALYSIS WHICH DEALS WITH THE CONVERSION OF MATHEMATICAL PROCESSES INTO OPERATIONS WITH NUMBERS.

OBJECT LANGUAGE
THE RESULT OF A TRANSLATION PROCESS STARTING WITH A SOURCE LANGUAGE. USUALLY, SYNONYMOUS WITH MACHINE LANGUAGE.

OCTAL
PERTAINING TO THE NUMBER SYSTEM WITH A BASE OF EIGHT.

OFF-LINE
PERTAINING TO EQUIPMENT OR DEVICES NOT UNDER DIRECT CONTROL OF THE CENTRAL PROCESSING UNIT.

OP CODE
OPERATION CODE. A COMPUTER INSTRUCTION CODE.

OPEN SUBROUTINE
A SUBROUTINE THAT MUST BE RELOCATED AND INSERTED INTO A ROUTINE AT EACH PLACE IT IS USED. SYNONYMOUS WITH DIRECT INSERT SUBROUTINE. CONTRAST WITH CLOSED SUBROUTINE.

OPERATOR
A PERSON WHO OPERATES A MACHINE.

OS/360
OPERATING SYSTEM/360. SUPERVISOR FOR NON TIME-SHARED 360 SYSTEMS.

OVERLAP
PROCESSING AND INPUT/OUTPUT TO DO SOMETHING AT THE SAME TIME SOMETHING ELSE IS BEING DONE, FOR EXAMPLE, TO PERFORM INPUT/OUTPUT OPERATIONS WHILE INSTRUCTIONS ARE BEING EXECUTED BY THE CENTRAL PROCESSING UNIT.

OVERLAY
THE TECHNIQUE OF REPEATEDLY USING THE SAME BLOCKS OF INTERNAL STORAGE DURING DIFFERENT STAGES OF A PROBLEM. WHEN ONE ROUTINE IS NO LONGER NEEDED IN STORAGE, ANOTHER ROUTINE CAN REPLACE ALL OR PART OF IT.

OBJECT PROGRAM
THE RESULT OF TRANSLATING A PROGRAM FROM ITS ORIGINAL FORM INTO A MACHINE-READABLE FORM; THE ACTUAL RUNNING PROGRAM.

ON-LINE
OPERATION UNDER DIRECT CONTROL OF THE COMPUTER; TASKS PERFORMED UNDER DIRECT COMPUTER CONTROL.

OPEN-SHOP PROGRAMMING
A BASIS FOR ORGANIZING WORK IN A COMPUTING CENTER IN WHICH THE PERSON WITH THE PROBLEM TO SOLVE DOES HIS OWN PROGRAMMING WITH OR WITHOUT HELP FROM PERSONNEL ATTACHED TO THE CENTER.

OPERATING SYSTEM
AN INTEGRATED COLLECTION OF COMPUTER INSTRUCTIONS THAT HANDLE SELECTION, MOVEMENT AND PROCESSING OF PROGRAMS AND DATA NEEDED TO SOLVE PROBLEMS.

OPTICAL READER
A DEVICE USED FOR MACHINE RECOGNITION OF CHARACTERS BY IDENTIFICATION OF THEIR SHAPES.

OPTICAL SCANNING
A PROCESS IN WHICH A LIGHT BEAM REFLECTED FROM (OR TRANSMITTED THROUGH) A SOURCE DOCUMENT IS ANALYZED TO IDENTIFY THE SYMBOLS ON THAT DOCUMENT. THE LIGHT BEAM IS CONTROLLED TO SCAN THE DOCUMENT IN SOME PREDETERMINED WAY.

OUTPUT
1. THE FINAL RESULTS AFTER DATA IS PROCESSED IN A COMPUTER.
2. THE DEVICE OR SET OF DEVICES USED FOR TAKING DATA OUT OF A COMPUTER SYSTEM AND PRESENTING THEM TO THE USER IN THE FORM HE DESIRES.

PARALLEL
PERTAINING TO THE SIMULTANEITY OF TWO OR MORE PROCESSES, OR 2 PERTAINING TO THE SIMULTANEITY OF TWO OR MORE SIMILAR OR IDENTICAL PROCESSES.

PARAMETER
A VARIABLE THAT IS GIVEN A CONSTANT VALUE FOR A SPECIFIC PURPOSE OR PROCESS.

PARITY
AN ERROR DETECTING TECHNIQUE IN WHICH A REDUNDANT BIT IS USED WITH AN ARRAY OF BITS SO THAT THE SUM OF EACH GROUP OF BITS

IS ALWAYS ODD OR ALWAYS EVEN.

PARITY BIT
A BINARY DIGIT APPENDED TO AN ARRAY OF BITS TO MAKE THE SUM OF ALL THE BITS ALWAYS ODD OR ALWAYS EVEN.

PARITY CHECK
A CHECK THAT TESTS WHETHER THE NUMBER OF ONES OR ZEROS IN AN ARRAY OF BINARY DIGITS IS ODD OR EVEN.

PATCH
TO MODIFY A ROUTINE IN A ROUGH OR EXPEDIENT WAY.

PCP ENVIRONMENT
PRIMARY CONTROL PROGRAM.

PDS
PARTITIONED DATA SET.

PL/1
PROGRAMMING LANGUAGE, LEVEL 1

PRESET
TO ESTABLISH AN INITIAL CONDITION.

PROBLEM PROGRAM
ANY OF THE CLASS OF ROUTINES THAT PERFORMS PROCESSING OF THE TYPE FOR WHICH A COMPUTING SYSTEM IS INTENDED.

PROC
PROCEDURE.

PROCESS CONTROL BY COMPUTERS
PERTAINING TO SYSTEMS WHOSE PURPOSE IS TO PROVIDE AUTOMATION OF CONTINUOUS OPERATIONS. THIS IS CONTRASTED WITH NUMERICAL CONTROL WHICH PROVIDES AUTOMATION OF DISCRETE OPERATIONS.

PROCLIB
PROCEDURE LIBRARY, A LIBRARY OF JCL STATEMENTS CALLABLE BY PROGRAMMERS.

PROGRAM LIBRARY
A COLLECTION OF AVAILABLE COMPUTER PROGRAMS AND ROUTINES.

PAPER-TAPE READER
A DEVICE WHICH SENSES AND TRANSLATES THE HOLES IN A ROLL OF PERFORATED PAPER TAPE INTO MACHINE-PROCESSABLE FORM.

PARALLEL PROCESSING
TO PROCESS SIMULTANEOUSLY WITH SEPARATE EQUIPMENT.

PATTERN RECOGNITION
THE PROCESS OF LOCATING AND IDENTIFYING A PATTERN SUCH AS THOSE MADE BY PRINTED LETTERS, BUBBLE-CHAMBER PHOTOGRAPHS, ASTRONOMICAL PHOTOGRAPHS AND SPECTRA, X-RAY PHOTOGRAPHS, AND CLOUD-COVER PHOTOGRAPHS.

PRINTER
A DEVICE WHICH PRINTS RESULTS FROM A COMPUTER ON PAPER.

PROBLEM-ORIENTED PROGRAMMING LANGUAGE
AN ARTIFICIAL LANGUAGE (VOCABULARY AND RULES) CONVENIENTLY EXPRESSING RELATIONSHIPS BETWEEN A PARTICULAR PROBLEM OR CLASS OF PROBLEMS AND THE METHOD OF SOLUTION. SEE ALSO "FORMAL LANGUAGE," "PROCEDURE-ORIENTED LANGUAGE," "MACHINE LANGUAGE," AND "LANGUAGE."

PROCEDURE-ORIENTED PROGRAMMING LANGUAGE
A LANGUAGE FOR WRITING COMPUTER PROGRAMS THAT CONVENIENTLY EXPRESSES CERTAIN PROBLEM-SOLVING PROCEDURES. SUCH LANGUAGES SHOULD BE DISTINGUISHED FROM PROBLEM-ORIENTED PROGRAMMING LANGUAGES WHICH ARE DESIGNED TO FACILITATE THE SOLUTION OF A TYPE OF PROBLEM.

PROCESSOR
A GENERIC TERM WHICH INCLUDES ASSEMBLY, COMPILING, GENERATION, ETC.

PROGRAM (NOUN)
A PLAN FOR THE SOLUTION OF A PROBLEM. OFTEN USED INTERCHANGEABLY WITH "ROUTINE" TO SPECIFY THE PRECISE SEQUENCE OF INSTRUCTIONS ENABLING A COMPUTER TO SOLVE A PROBLEM.
PROGRAM (VERB)
TO MAKE A PROGRAM, INCLUDING INVESTIGATIONS OF SOLUTION METHOD, NUMERICAL ANALYSIS, APPROPRIATE PARAMETER CHOICES, AND SO ON. THE WRITING OF A SEQUENCE OF INSTRUCTIONS IS ONLY PART OF PROGRAMMING ALTHOUGH OFTEN THE TERMS ARE USED INTERCHANGEABLY.

PROGRAM DECK
A SET OF PUNCHED CARDS (DECK) CONTAINING INSTRUCTIONS THAT MAKE UP A COMPUTER PROGRAM.

PROGRAM INTERRUPT
A SIGNAL CAUSING A COMPUTER TO STOP EXECUTION OF THE CURRENT PROGRAM BUT TO SAVE THE STATUS OF THE MACHINE SO THAT THAT PROGRAM WILL BE ABLE TO CONTINUE AFTER THE INTERRUPTING PROGRAM IS FINISHED. ALSO THE CORRESPONDING ACTION.

PROGRAM MAINTENANCE
COMPUTER PROGRAMS REQUIRE PERIODIC MAINTENANCE TO REMOVE ERRORS AND DISCREPANCIES WHICH MAY BE DISCOVERED AFTER LONG PERIODS OF USE, TO CORRECT ADDITIONS OR DELETIONS WHICH MAY HAVE BEEN INADVERTENTLY MADE, TO IMPROVE AND MODERNIZE THE PROCEDURES USED, AND TO ADAPT THEM TO USE NEW UNITS OF EQUIPMENT WHICH MAY BE ADDED TO THE COMPUTER.

PRCGRAMMING
THE PROCEDURES CONTRIBUTING TO THE DEVELOPMENT OF A SEQUENCE OF INSTRUCTIONS FOR COMPUTER SOLUTION OF A PROBLEM; INCLUDES PROBLEM ANALYSIS, PROGRAM DESIGN, CODING, AND TESTING.

PROGRAMMING LANGUAGES
THOSE LANGUAGES DEFINED AND USED FOR THE PROGRAMMING OF DIGITAL COMPUTERS.

PRCGRAMMING SYSTEMS
PROGRAMS AND PROCEDURES DESIGNED AND USED TO ASSIST IN THE PREPARATION OF DIGITAL-COMPUTER PROGRAMS. AMONG SUCH AIDS ARE COMPILERS, DIAGNOSTIC PROGRAMS, AND PROGRAMS TO PRODUCE FLOW CHARTS.

PRCJECT EVALUATION AND REVIEW TECHNIQUE
A MANAGEMENT-CONTROL TOOL FOR DEFINING, INTEGRATING, AND INTERRELATING WHAT MUST BE DONE TO ACCOMPLISH DESIRED OBJECTIVES ON TIME. A COMPUTER IS USED TO COMPARE CURRENT PROGRESS AGAINST PLANNED OBJECTIVES AND TO GIVE MANAGEMENT INFORMATION FOR PLANNING AND DECISION-MAKING. SEE "CPM."

PUNCHED CARD
1. A CARD PUNCHED WITH A PATTERN OF HOLES TO REPRESENT DATA.
2. A CARD AS IN 1, BEFORE BEING PUNCHED (SLANG).

QUANTITATE
TO MEASURE OR ESTIMATE THE QUANTITY OF; TO EXPRESS IN QUANTITATIVE TERMS.

QUANTUM CHEMISTRY
THAT PORTION OF CHEMISTRY BASED ON THE THEORY THAT ENERGY IS NOT ABSORBED OR RADIATED CONTINUOUSLY BUT DISCONTINUOUSLY, IN DEFINITE UNITS CALLED QUANTA.

RADIX
THE BASE OF A NUMBER SYSTEM TEN FOR THE DECIMAL, TWO FOR THE BINARY, EIGHT FOR THE OCTAL, ETC.

RANGE
THE SET OF VALUES THAT A QUANTITY OR FUNCTION MAY ASSUME.

READ
TO ACQUIRE DATA FROM A SOURCE.

RECORD
A SET OF DATA.

RELATIVE ADDRESS
THE NUMBER THAT SPECIFIES THE DIFFERENCE BETWEEN THE ABSOLUTE ADDRESS AND THE BASE ADDRESS.

REMOTE TERMINAL
ANY DEVICE CAPABLE OF SENDING AND/OR RECEIVING INFORMATION AT A DISTANT LOCATION OVER A COMMUNICATION CHANNEL.

RESET
TO RESTORE A STORAGE DEVICE TO A PRESCRIBED INITIAL STATE, NOT NECESSARILY THAT DENOTING ZERO.

RESTART
TO RETURN TO A PREVIOUS POINT IN A PROGRAM AND RESUME OPERATION FROM THAT POINT.

RJE
REMOTE JOB ENTRY.

RPG
REPORT PROGRAM GENERATOR. A HIGH LEVEL BUSINESS-ORIENTED PROGRAMMING LANGUAGE FOR CREATING REPORTS.

RUN
A SINGLE, CONTINUOUS PERFORMANCE OF A COMPUTER ROUTINE.

RANDOM-ACCESS MEMORIES
COMPUTER MEMORIES IN WHICH THE TIME REQUIRED TO LOCATE THE NEXT POSITION FROM WHICH INFORMATION IS TO BE OBTAINED IS IN NO WAY DEPENDENT ON THE POSITION LAST LOCATED.

REACTION TIME
THE TIME FROM THE APPLICATION OF A STIMULUS TO THE RESPONSE TO THAT STIMULUS. THE TIME FROM THE SUBMISSION OF A JOB BY AN INVESTIGATOR TO ITS RETURN TO THE INVESTIGATOR.

REAL TIME
THE TECHNIQUE OF COMPUTING WHILE A PROCESS TAKES PLACE SO THAT RESULTS CAN BE USED TO GUIDE OPERATION OF THE PROCESS.

REGISTER
A SPECIAL DEVICE THAT HOLDS INFORMATION READY FOR MANIPULATION. IT HOLDS ONLY A PART (SUCH AS A WORD) OF THE TOTAL INFORMATION IN A DIGITAL COMPUTER.

RELOCATABLE PROGRAM
A DIGITAL-COMPUTER PROGRAM WHICH CAN BE PLACED IN ANY PORTION OF THE COMPUTER MEMORY. THUS A PROGRAM INDEPENDENT OF LOCATION.

RESONANCE
THE REINFORCED VIBRATION OF A BODY EXPOSED TO THE VIBRATION, AT ABOUT THE SAME FREQUENCY, OF ANOTHER BODY OR PHYSICAL MAGNITUDE.

ROUTINE
A SEQUENCE OF MACHINE INSTRUCTIONS WHICH CARRY OUT A SPECIFIC PROCESSING FUNCTION.

SCAN
TO EXAMINE SEQUENTIALLY, PART BY PART.

SEMANTICS
THE RELATIONSHIPS BETWEEN SYMBOLS AND THEIR MEANINGS.

SEQUENTIAL ACCESS
OBTAINING DATA FROM AN INPUT/OUTPUT DEVICE IN A SERIAL MANNER ONLY.

SEQUENTIAL CONTROL
A MODE OF COMPUTER OPERATION IN WHICH INSTRUCTIONS ARE EXECUTED CONSECUTIVELY UNLESS SPECIFIED OTHERWISE BY A TRANSFER OF CONTROL.

SEQUENTIAL OPERATION
PERTAINING TO A PERFORMANCE OF OPERATIONS ONE AFTER THE OTHER.

SNAPSHOT
SNAP A DUMP USUALLY OF A SELECTED AREA OF STORAGE TAKEN DURING PROCESSING AT SPECIFIED TIMES PROVIDING A TIME HISTORY OF THE SPECIFIED STORAGE AREA FOR DEBUGGING PURPOSES.

SORTER
A DEVICE OR COMPUTER ROUTINE THAT SORTS.

SPECIAL CHARACTER
IN A CHARACTER SET, A CHARACTER THAT IS NEITHER A NUMERAL, A LETTER, NOR A BLANK, E.G., ASTERISK, DOLLAR SIGN, EQUALS SIGN, COMMA, PERIOD.

SPOOL
SIMULTANEOUS PERIPHERAL OPERATION, ON LINE.

STATEMENT
IN COMPUTER PROGRAMMING, A MEANINGFUL EXPRESSION OR GENERALIZED INSTRUCTION IN A SOURCE LANGUAGE.

STEP
ONE OPERATION IN A COMPUTER ROUTINE.

STORAGE CAPACITY
THE AMOUNT OF DATA THAT CAN BE CONTAINED IN A STORAGE DEVICE.

SUBPROGRAM
A PROGRAM THAT IS A PART OF ANOTHER PROGRAM, USUALLY SYNONYMOUS WITH SUBROUTINE.

SYMBOL
A REPRESENTATION OF SOMETHING BY MEANS OF RELATIONSHIP, ASSOCIATION, OR CONVENTION.

SYNTAX
1 THE STRUCTURE OF EXPRESSIONS IN A LANGUAGE, OR 2 THE RULES GOVERNING THE STRUCTURE OF A LANGUAGE.

SYSGEN
SYSTEM GENERATION, THE PROCESS BY WHICH A NEW OPERATING SYSTEM AND SUPPORTING SOFTWARE E.G. COMPILERS AND UTILITY PROGRAMS IS CREATED IN A 360.

SELF-ORGANIZING ADAPTIVE SYSTEMS
ANY SYSTEM WHICH CAN CONTROL ITS OWN STRUCTURE SO THAT IT CAN ADAPT TO CHANGES IN ITS ENVIRONMENT.

SELF-ORGANIZING SYSTEMS
SAME AS SELF-ORGANIZING ADAPTIVE SYSTEMS.

SENSORS
DEVICES TO DETECT AND MEASURE PHYSICAL PHENOMENA, SUCH AS TEMPERATURE, STRESS, HEARTBEAT, AND ACCELERATION.

SIMULATION
REPRESENTATION OF THE ESSENTIAL ELEMENTS OF SOME OBJECT, PHENOMENON, SYSTEM, OR ENVIRONMENT THAT FACILITATES ITS CONTROL AND STUDY (OFTEN BY OR INVOLVING AN AUTOMATIC COMPUTER).

SLOW POTENTIAL
A LOW-FREQUENCY COMPONENT (APPROXIMATELY 4-5 CYCLES/SEC) OF THE ELECTROENCEPHALOGRAPH.

SOFTWARE
COMPUTER PROGRAMS AND COLLECTIONS THEREOF, INCLUDING COMPILERS AND ASSEMBLERS WHICH CAN BE USED TO GENERATE OTHER PROGRAMS. ALSO INCLUDES EXECUTIVE AND DIAGNOSTIC PROGRAMS WHICH CAN BE USED TO SCHEDULE AND TEST OTHER PROGRAMS.

SOLID-STATE
REFERS TO THOSE DEVICES WHICH UTILIZE THE ELECTRIC, MAGNETIC, OR PHOTIC PROPERTIES OF SOLID MATERIALS-E.G., TRANSISTORS, MAGNETIC CORES, ETC.

SORT
TO ARRANGE DATA IN AN ORDERED SEQUENCE.

SOURCE LANGUAGE
THE LANGUAGE IN WHICH A PROGRAM IS ORIGINALLY WRITTEN. USED TO INDICATE THAT CONVERSION TO A MACHINE LANGUAGE IS REQUIRED.

SOURCE-LANGUAGE DEBUGGING
THE DETECTION AND CORRECTION OF ERRORS (BUGS) USING ONLY THE SOURCE LANGUAGE.

SOURCE PROGRAM
A PROGRAM IN ITS ORIGINAL FORM BEFORE BEING PROCESSED BY A COMPUTER. USUALLY REFERS TO PROGRAMS WRITTEN IN A PROCEDURE-ORIENTED LANGUAGE AS OPPOSED TO MACHINE LANGUAGE.

SPECIAL-PURPOSE
BEING APPLICABLE TO A LIMITED CLASS OF USES WITHOUT ESSENTIAL MODIFICATION. CONSTRASTED WITH "GENERAL-PURPOSE."

SPECTRAL ANALYSIS
SEPARATION OF A SERIES OF VALUES TO IDENTIFY THEIR SIGNIFICANCE TO THE PROBLEM IN QUESTION.

STORAGE
PERTAINING TO A DEVICE IN WHICH DATA CAN BE ENTERED AND STORED AND FROM WHICH IT CAN BE RETRIEVED AT A LATER TIME.

STORAGE ALLOCATION
THE ASSIGNMENT OF STORAGE LOCATIONS TO MAIN ROUTINES AND SUBROUTINES THEREBY FIXING THE OPERATING VALUES OF ADDRESSES IN RELOCATABLE PROGRAMS.

STORED PROGRAM
A PROGRAM IN THE INTERNAL STORAGE SECTION WHICH CONTROLS THE BEHAVIOR OF A COMPUTER OR OTHER DEVICE. THE COMPUTER THUS HAS ACCESS TO AND CAN CHANGE ITS OWN PROGRAM.

STORED-PROGRAM COMPUTER
A DIGITAL COMPUTER THAT STORES INSTRUCTIONS IN MAIN CORE AND CAN BE PROGRAMMED TO ALTER ITS OWN INSTRUCTIONS AS THOUGH THEY WERE DATA AND CAN SUBSEQUENTLY EXECUTE THESE ALTERED INSTRUCTIONS.

SUBROUTINE
A PROGRAM SO ARRANGED THAT CONTROL MAY BE TRANSFERRED TO IT FROM A MAIN PROGRAM, AND, AT THE CONCLUSION OF THE SUBROUTINE, CONTROL REVERTS TO THE APPROPRIATE POINT IN THE MAIN PROGRAM.

THIS AVOIDS REPEATING THE SAME SEQUENCE OF INSTRUCTIONS AT DIFFERENT PLACES IN THE PRINCIPAL PROGRAMS.

SUPERCONDUCTIVITY

SOME METALS AND A GREAT NUMBER OF ALLOYS LOSE ALL THEIR ELECTRICAL RESISTANCE AT VERY LOW TEMPERATURES. THE TEMPERATURE AT WHICH THIS OCCURS MAY VARY FROM A FRACTION OF A DEGREE TO A HIGH OF APPROXIMATELY 9 K FOR NIOBIUM. THESE METALS ARE CALLED SUPERCONDUCTORS, AND THE TEMPERATURE AT WHICH THE TRANSITION TO SUPERCONDUCTIVITY TAKES PLACE IS KNOWN AS THE CRITICAL TEMPERATURE. VERY HIGH MAGNETIC FIELDS WILL CAUSE A SUPERCONDUCTING MATERIAL TO TRANSFORM TO THE NORMAL STATE. THUS FAR, APPROXIMATELY 23 ELEMENTS HAVE BEEN FOUND TO BECOME SUPERCONDUCTORS IF TAKEN TO SUFFICIENTLY LOW TEMPERATURES.

SWITCHING

THE CONNECTION OF TWO POINTS OF A NETWORK AT CONTROLLABLE INSTANTS OF TIME.

SYMBOLIC LANGUAGE

SEE MNEMONIC CODE.

SYSTEM

AN ASSEMBLY OF UNITS, DEVICES, OR MACHINES UNITED BY SOME FORM OF REGULATED INTERACTION TO FORM AN ORGANIZED WHOLE. OR: A COLLECTION OF OPERATIONS AND PROCEDURES, MEN AND MACHINES, BY WHICH AN ACTIVITY IS CARRIED ON.

SYSTEMS ANALYSIS

THE STUDY OF ARRANGEMENTS OF TERMS OR ENTITIES MAKING UP A SYSTEM, ESPECIALLY ARRANGEMENTS THAT COMPOSE A LARGER AGGREGATE.

SYSTEMS PROGRAMMING

THE DEVELOPMENT OF PROGRAMS WHICH FORM OPERATING SYSTEMS FOR COMPUTERS. SUCH PROGRAMS INCLUDE COMPILERS, TRANSLATORS, MONITORS, GENERATORS, ETC.

TABLE

A COLLECTION OF DATA, EACH ITEM BEING UNIQUELY IDENTIFIED EITHER BY SOME LABEL OR BY ITS RELATIVE POSITION.

TABULATE

1 TO FORM DATA INTO A TABLE, OR 2 TO PRINT TOTALS.

TAPE

SEE MAGNETIC TAPE.

TAPE TO CARDS
PERTAINING TO EQUIPMENT OR METHODS THAT TRANSMIT DATA FROM EITHER MAGNETIC TAPE OR PUNCHED TAPE TO PUNCHED CARDS.

TAPE UNIT
A DEVICE CONTAINING A TAPE DRIVE, TOGETHER WITH READING AND WRITING HEADS AND ASSOCIATED CONTROLS.

TELEPROCESSING
A FORM OF INFORMATION HANDLING IN WHICH A DATA PROCESSING SYSTEM UTILIZES COMMUNICATION FACILITIES.

TEMPORARY STORAGE
IN PROGRAMMING, STORAGE LOCATIONS RESERVED FOR INTERMEDIATE RESULTS. SYNONYMOUS WITH WORKING STORAGE.

TERMINAL
A POINT IN A SYSTEM OR COMMUNICATION NETWORK AT WHICH DATA CAN EITHER ENTER OR LEAVE.

TESTING
THE PROCESS FOLLOWING DEBUGGING OF A COMPUTER ROUTINE, TO VERIFY THAT THE SOFTWARE AND/OR THE HARDWARE IS FUNCTIONING PROPERLY.

TRANSCEIVING
A PROCEDURE INVOLVING THE SENDING AND/OR RECEIVING OF DATA VIA A REMOTE TERMINAL.

TRANSLATE
TO CONVERT FROM ONE LANGUAGE TO ANOTHER LANGUAGE, E.G., FROM FORTRAN TO MACHINE LANGUAGE.

TELEMETERED EXPERIMENTAL DATA
INFORMATION WHICH HAS BEEN MEASURED AT A DISTANCE BY THE TRANSMISSION OF A SUITABLE SIGNAL BY TELEGRAPH, TELEPHONE, OR RADIO.

TERMINAL UNIT
A DEVICE, SUCH AS A KEY-DRIVEN OR VISUAL DISPLAY TERMINAL, WHICH CAN BE CONNECTED TO A COMPUTER OVER A COMMUNICATIONS CIRCUIT AND WHICH MAY BE USED FOR EITHER INPUT OR OUTPUT FROM A LOCATION EITHER NEAR OR FAR REMOVED FROM THE COMPUTER.

THIN-FILM MEMORY
A MEMORY ELEMENT MADE BY DEPOSITING MAGNETIC ALLOYS IN LAYERS SO THIN THAT DIRECTION OF MAGNETIZATION CAN BE SWITCHED EXTREMELY RAPIDLY.

TIME SHARING
A TECHNIQUE ALLOWING EXECUTION OF TWO OR MORE FUNCTIONS ESSENTIALLY AT THE SAME TIME, BY ALLOCATING (IN ROTATION, FOR INSTANCE) SMALL DIVISIONS OF THE TOTAL TIME FOR THE PERFORMANCE OF EACH FUNCTION. A SYSTEM BY WHICH SEVERAL CONSOLES ARE CONNECTED TO A LARGE CENTRAL COMPUTER WHICH IS PROGRAMMED SO THAT, ON CALL, IT CAN GIVE SHORT BURSTS OF TIME INTERMITTENTLY TO EACH CONSOLE.

TOPOLOGY
A BRANCH OF MATHEMATICS CONCERNED WITH THE RELATIONS OF GEOMETRIC FORMS WITHOUT REGARD TO THEIR SIZE OR MEASURE.

TRANSDUCER
A DEVICE WHICH CONVERTS ENERGY FROM ONE FORM TO ANOTHER, AS A HI-FI PICKUP CARTRIDGE CONVERTS MECHANICAL TO ELECTRICAL ENERGY.

TUNNEL DIODE
THE TUNNEL DIODE IS A SPECIAL TYPE OF P-N JUNCTION DIODE. AS ONE INCREASES THE VOLTAGE ACROSS THIS DIODE, THE CURRENT FIRST INCREASES AND THEN DECREASES, AND THEN INCREASES AGAIN. THE REGION WHERE THE CURRENT FALLS AS THE VOLTAGE RISES IS CALLED A "NEGATIVE-RESISTANCE" REGION. THIS NEGATIVE-RESISTANCE REGION GIVES THE DIODE MANY PRACTICAL USES. THE NAME "TUNNEL DIODE" COMES FROM A QUANTUM MECHANICAL EFFECT ON WHICH THE DEVICE IS BASED. IN THE TUNNEL EFFECT IT IS FOUND THAT THE WAVE NATURE OF ATOMIC PARTICLES SOMETIMES ENABLES THEM TO GET TO THE OTHER SIDE OF A BARRIER DESPITE THE FACT THAT THEY DO NOT HAVE ENOUGH ENERGY TO GET OVER THE TOP OF THE BARRIER. THE PROCESS IS ONE OF PENETRATION OF THE BARRIER AND HENCE THE NAME "TUNNEL EFFECT."

TRANSISTOR
A TRANSISTOR IS BASICALLY A DEVICE MADE BY ATTACHING THREE WIRES TO A SMALL WAFER OF SEMICONDUCTING MATERIAL. THE SEMICONDUCTING MATERIAL IS A SINGLE CRYSTAL WHICH HAS BEEN SPECIALLY TREATED SO THAT ITS PROPERTIES ARE DIFFERENT AT THE POINT WHERE EACH OF THE WIRES IS ATTACHED. THE THREE WIRES ARE USUALLY CALLED THE EMITTER, BASE, AND COLLECTOR, AND THEY PERFORM FUNCTIONS SIMILAR TO THOSE OF THE CATHODE, GRID, AND PLATE OF A VACUUM TUBE (IN THE SAME ORDER). THE TRANSISTOR IS USUALLY WIRED INTO A CIRCUIT IN SUCH A WAY THAT A SMALL CURRENT TO BE AMPLIFIED IS SENT INTO THE BASE AND PRODUCES A CORRESPONDINGLY LARGER CURRENT IN THE COLLECTOR.

TURN-AROUND TIME
THE TIME ELAPSED BETWEEN THE SUBMISSION OF A COMPUTER RUN BY AN INVESTIGATOR AND THE RETURN TO HIM OF THE RESULTS OF THE RUN. WITH THE CLOSED-SHOP OPERATION OF A LARGE COMPUTER, THIS INTERVAL MAY BE FROM AN HOUR TO MORE THAN A DAY. TWO OR THREE HOURS IS USUALLY CONSIDERED TO BE A SHORT TURN-AROUND TIME.

TRUTH TABLE
A TABLE THAT DESCRIBES A LOGIC FUNCTION BY LISTING ALL POSSIBLE COMBINATIONS OF INPUT VALUES AND INDICATING ALL THE LOGICALLY TRUE OUTPUT VALUES.

UNDERFLOW
PERTAINING TO THE CONDITION THAT ARISES WHEN A MACHINE COMPUTATION YIELDS A NONZERO RESULT THAT IS SMALLER THAN THE SMALLEST NONZERO QUANTITY THAT THE INTENDED UNIT OF STORAGE IS CAPABLE OF STORING.

UNIT
1 A DEVICE HAVING A SPECIAL FUNCTION, OR 2 A BASIC ELEMENT.

UNIT RECORD EQUIPMENT
ELECTROMECHANICAL MACHINES USED TO PROCESS DATA RECORDED ON PUNCHED CARDS. OFTEN USED AS INPUT/OUTPUT DEVICES CONNECTED TO AN ELECTRONIC STORED-PROGRAM COMPUTER.

VARIABLE
A QUANTITY THAT CAN ASSUME ANY OF A GIVEN SET OF VALUES.

VERIFIER
A DEVICE SIMILAR TO A CARD PUNCH, TO CHECK THE INSCRIBING OF DATA BY REKEYING.

VERIFY
TO CHECK THE RESULTS OF KEYPUNCHING.

VARIABLE-CAPACITY DIODE
A SILICON SEMICONDUCTOR DIODE IN WHICH THE CAPACITANCE, VARYING AS A FUNCTION OF THE BIAS VOLTAGE, IS USED AS A CIRCUIT ELEMENT, ALLOWING THE DEVICE TO BE USED AS A VARIABLE-REACTANCE CONTROL DEVICE OR AMPLIFIER.

WARM START
ONE OR MORE PROGRAMS STORED ON SPOOL PACK.

WRITE
TO DELIVER DATA TO A MEDIUM SUCH AS STORAGE.

WORD
A SET OF CHARACTERS WHICH HAVE ONE ADDRESSABLE LOCATION AND ARE TREATED AS ONE UNIT.

WORD SIZE
THE NUMBER OF CHARACTERS IN A WORD. SYNONOMOUS WITH "WORD LENGTH."

APPENDIX III

Running a WATFIV Job on the IBM System/360 Model 65 User Terminal

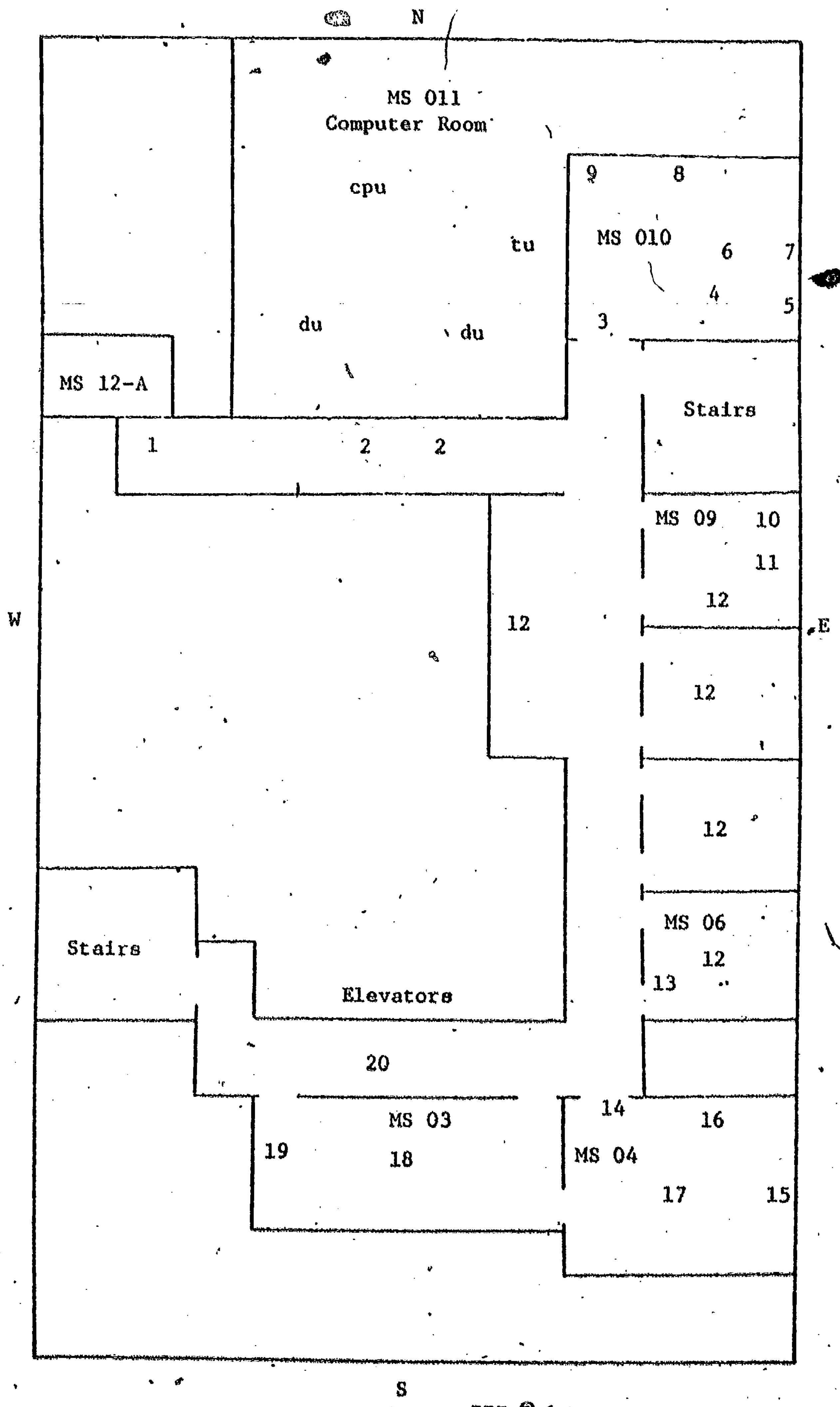
Section A of this appendix contains a self-guided tour of the University Computer Center facilities. Section B describes the control cards that are used for running WATFIV jobs. A procedure for actually running a job is given in Section C.

Section A

The University Computer Center-Facilities

You need to know your way around the University Computer Center before you attempt to run any jobs on the computer. The following pages contain a map and a guide to the user area of the Computer Center in the basement of the Mathematical Sciences Building. Take the self-guided tour, referring to the map and the guide following the map.

Map of the University Computer Center, MS Basement



Guide to the Map of the University Computer Center, MS Basement

The numbers below correspond to the numbers on the map of the MS basement. Start at "1" and check off each item as you go.

1. _____ This is the supervisor's office. Report any malfunctions or difficulties with any equipment (keypunch machines, card reader, line printer, etc.) to the supervisor.

2. _____ The console with the blinking lights is part of the central processing unit (cpu) of the IBM System/360 Model 65 computer. This unit is the heart of the computing system. Behind the console is the memory unit, containing 1,572,864 addressable storage locations (or bytes).

_____ In the foreground along the wall to your left and to your right is a line of disc drive units (du) for storing and retrieving information using magnetic discs. (You will probably see some disc pack covers sitting around the area. They look like plastic cake containers.)

_____ To the right and in front of the central processing unit is a line of magnetic tape drive units (tu) for storing and retrieving information. (Data transfer with magnetic tape units is slower than with disc units by several orders of magnitude.)

_____ Most of the other units in the room are control units for the various input/output devices.

3. _____ This room contains the user terminal of the computer.

_____ This area, like all other areas, must be kept clean. Count the number of waste containers in this room. _____

4. _____ Find the name of this unit on the nameplate. _____

_____ Acquaint yourself with any signs and instructions taped to the machine. You will need them later.

_____ Find two sets of "START" and "STOP" buttons.

5. _____ Please read and observe the "No Smoking" sign on the wall.

6. _____ Find the name of this unit on the nameplate. _____

_____ Acquaint yourself with any signs and instructions taped to this machine. You will need them later.

_____ Locate the "END OF FILE" and "START" buttons.

7. _____ On the wall are the instructions for using the card reader. At the time when you are ready to use the card reader, read these directions carefully.

8. _____ Along this wall is the input/output area for jobs that are run "inside" on the computer by an operator. Jobs are submitted at the window to your left and are picked up from the bins. <u>You will not be using this service in this course</u>.

9. _____ In the window where jobs are submitted is a tray containing control cards for running jobs on the computer. The only control cards you will need for this course are $JOB, $ENTRY, and $IBSYS, some of which should be available in the tray. (You don't need to take the cards now.)

_____ At the back of the tray is a supply of rubber bands for putting around decks of computer cards.

_____ Just to the right and above the window is a button for calling an operator whenever you need to report difficulties with equipment.

10. _____ This is a cathode ray tube (CRT) display unit that shows the status of jobs being run in the computer. You will not need to use this unit in this course, but it's sort of fun to watch.

11. _____ Beside the CRT unit is a bin of OMR cards (computer cards that can be marked with a pencil).

12. _____ These are work areas for your convenience. PLEASE! KEEP THESE AREAS CLEAN!

13. _____ A diagnostic lab is operated in this area for students enrolled in basic computer science programming courses. This service provides help for you when you do not understand diagnostic messages on your computer output.

14. _____ This is the user keypunch area. How many waste containers can you find in this room? _____ PLEASE USE THEM! KEEP THIS AREA CLEAN!

15. _____ Please read and observe the "No Smoking, No Drinks" sign.

16. _____ The dark grey keypunch machines are model 26 machines. They are used primarily by students enrolled in COMSC 2112.

17. _____ The light grey keypunches are model 29 machines.

_____ Please read and observe the instructions taped to some of the desk tops of the machines. PLEASE CLEAN UP YOUR MESS!

_____ Notice the card bins containing blank (unpunched) computer cards placed among the keypunch machines.

18. _____ This room contains unit record equipment, which you will not need for this course.

19. _____ There are keypunches for making corrections only located here for your convenience.

_____ Notice that there is a time limit on the use of these machines, which must be observed.

20. _____ You're on your own to find your way out.

Section B

WATFIV Control Cards

Three control cards are required for all WATFIV jobs: $JOB, $ENTRY, and $IBSYS.

The $JOB card initiates the accounting routine used by the Computer Center for charges and records; it also initiates the compile phase during which time the Fortran program is compiled or translated.

The general form for the $JOB card that you need for this course is shown below:

```
$JOB vvvvv,xxx-yy-zzzz Your name
```

vvvvv - the project number assigned to this course by the
        Computer Center, used for accounting purposes.

xxx-yy-zzzz - an assigned identification number used for
        the keeping of records in the course.

Your name must be preceded by at least one blank.

The $JOB card shown above is used with jobs that are punched on model 29 keypunch machines. If you use a model 26 keypunch, then the general form of the $JOB card includes a KP parameter:

```
$JOB vvvvv,xxx-yy-zzzz,KP=026 Your name
```

The $JOB card may contain additional parameters and options that are beyond the scope of this course. Refer to the section titled "WATFIV Job Card" in APPENDIX VIII if you need more information.

The $ENTRY card initiates the execution phase of the job during which time the instructions generated by the compiler are executed. The control information, $ENTRY, is punched in columns 1-6 of the card.

The $IBSYS card terminates the job, recording the necessary accounting information. The control information, $IBSYS, is punched in columns 1-6 of the card.

Place your cards in the following order when you run a job:

```
        $JOB
                  } Program deck
        $ENTRY
                  } Data cards, if any
        $IBSYS
```

Section C

Running a WATFIV Job

Now that you know your way around, try running a job on the user terminal in room 010.

1. Take your punched program deck and place the proper control cards with the program deck as described in Section B.
2. Read the sign "Instructions for Using Card Reader" on the wall in room 010.
3. Read the instructions attached to the card reader.
4. In your mind go through a dry run of putting your cards onto the card reader and starting it, but don't do an actual run yet.
5. Read "Additional Instructions for Operating the IBM 2501 Card Reader" contained in APPENDIX IV.
6. Read "Instructions for Using the IBM 1403 Printer" contained in APPENDIX V.
7. Once more referring to the sign on the wall, try running your job on the card reader, and get your output from the line printer.
8. Be sure that you pick up your deck (Careful! Not someone else's!) from the card reader.
9. Move out! Don't loiter in the terminal room.

APPENDIX IV

Additional Instructions for Operating the IBM 2501 Card Reader

The control panel of the card reader contains several lights and buttons. The top row contains signal lights and the bottom row contains push buttons.

VALIDITY CHECK (yellow)	END OF FILE (yellow)	READ CHECK (yellow)	READY (green)	FEED CHECK (yellow)	POWER ON (white)
	OMR CHECK (yellow) UNDEFINED	END OF FILE (blue)	START (green)	NPRO (blue)	STOP (red)

The following three lights indicate normal operating conditions.

1. The white POWER ON light indicates that the machine is on.
2. The green READY light will be on when the reader is reading cards and everything is "go." Even though the reader may stop, as long as the READY light is on, it is still "go." The reader is probably waiting for the computer or the line printer if it stops in the READY condition.
3. The yellow END OF FILE light comes on whenever the END OF FILE button is pushed and goes off whenever the READY light goes off. This light signals that the last card through the reader will be put into the stacker; otherwise, the last card will remain inside the reader.

The following four lights indicate abnormal conditions which cause the card reader to stop.

1. The yellow VALIDITY CHECK light signals that a card contains an invalid punch. There are at least three common causes:

   a. one or more cards placed upside down or backward in the feed hopper;

   b. a card with more than one character punched in a column of the card;

c. a card punched off center.

2. The yellow READ CHECK light indicates that the card was misread. Each card is read twice by the card reader, and the two readings are compared. If they are not identical, a "read check" stoppage occurs. Usually this condition is caused by foreign matter inside the holes of the card or inside the machine itself, but may also be caused by a defective card.

3. The yellow FEED CHECK light indicates a problem in the card handling mechanism. Usually this condition is caused by a bad card, especially a card with a blunted bottom edge (the 9-edge) or a card that is not flat.

4. The yellow OMR CHECK light indicates an invalid OMR card. This condition should not occur if you use the OMR cards provided by the Computer Center, but it could occur if the OMR feature is not functioning properly. The UNDEFINED light is disconnected and should never come on.

The buttons on the bottom row of the control panel control the operation of the card reader.

1. END OF FILE (blue) must be pushed in order for the last card processed by the reader to be ejected into the stacker. Otherwise, the last card remains inside the reader. Always push this button whenever you are using the card reader. (If you "lose" your $IBSYS card in the card reader, you forgot to push the END OF FILE button.)

2. START (green) must be pushed in order for the reader to operate. Pushing this button causes the READY light to come on, assuming that operation is normal.

3. NPRO (blue) is used for ejecting cards from inside the reader without processing them. This button is used primarily whenever an abnormal stoppage occurs (indicated by a FEED CHECK, OMR CHECK READ CHECK, or VALIDITY CHECK light) for getting cards out of the reader and into the stacker. This button is not functional if cards are in the feed hopper or if the STOP button has not been pushed.

4. STOP (red) is used for stopping the operation of the card reader and for releasing the reader from the control of the computer.

WHAT TO DO IN CASE OF STOPPAGE OF CARD READER

Light on	Procedure
READY	Check the PRINT READY light on the line printer. If the light is off, push the line printer START button. Wait. Operation should resume momentarily. If operation does not resume, then there may be a problem in the computer system. You may wish to remove your deck and return later. **How to remove your deck from the reader** Push STOP. Remove cards from the feed hopper (or just lift up on the cards and hold them above the bottom of the hopper). Push and hold NPRO momentarily until all cards have been ejected into the stacker. Remove your cards from the stacker (and from the feed hopper, if you have not done so).
VALIDITY CHECK	The last card ejected into the stacker before stoppage is the offending card. Is the offending card backward or upside down? NO. Remove your deck from the reader using the instructions above under "READY." Check the card for an invalid punch (usually more than one character punched in a column of the card) and for off center punches. Repunch the card and try again. YES. You can recycle the card through the reader as follows: Push STOP. Lift up and hold the cards in the feed hopper. Push and hold NPRO momentarily until all cards have been ejected into the stacker. Take the offending card and the ejected cards and place them properly in the feed hopper under the cards you are holding up. Be sure that the cards in the hopper are straight and neatly stacked; then push START.

Light on	Procedure
READ CHECK	The last card ejected into the stacker before stoppage is the offending card. Recycle the card through the reader as follows: Push STOP. Lift up and hold the cards in the feed hopper. Push and hold NPRO momentarily until all cards have been ejected into the stacker. Take the offending card and the ejected cards and place them properly in the feed hopper under the cards you are holding up. Be sure that the cards in the hopper are straight and neatly stacked; then push START. If read check occurs again, remove your cards using the instructions under "READY" above, repunch the offending card, and try again.
FEED CHECK	The bottom card in the feed hopper is probably the offending card. Is the weight on the cards in the feed hopper? NO. Inspect your cards in the hopper for damage. If they are OK, place cards and weight properly into the feed hopper, and push START. YES. Probably one or more of your cards in the feed hopper have blunt bottom edges or are not flat. Remove your cards from the reader using the instructions under "READY" above. Repunch damaged cards. (Sometimes you can repair blunt edges by placing the card on a flat surface and running the flat side of your thumbnail along the edges; the repair job is only temporary, however.) Sometimes a feed check means a "card jam" inside the reader. In such cases, secure the help of the Operations Supervisor (MS 12A) or an operator from the Computer Room.
OMR CHECK	Proceed exactly as in the case of a READ CHECK, replace the offending OMR card with a new one. If the condition prevails, report it to an operator or to the supervisor.
	When all else fails, secure help from the Operations Supervisor (MS 12A) or an operator from the Computer Room. DO NOT ATTEMPT TO REPAIR MALFUNCTIONING EQUIPMENT.

APPENDIX V

Instructions for Using the IBM 1403 Printer

The control panel for the 1403 Printer is on the front side in the upper left-hand corner. There are seven push buttons and five signal lights.

START (green)	CHECK RESET (blue)	STOP (red)
CARRIAGE SPACE (blue)	CARRIAGE RESTORE (blue)	SINGLE CYCLE (blue)

PRINT READY	PRINT CHECK
END OF FORMS	
FORMS CHECK	SYNC CHECK

CARRIAGE STOP (red)

The only buttons with which you will be concerned are START, STOP, and CARRIAGE RESTORE.

The green START button (there is also one on the back side in the upper right-hand corner) puts the printer in the "ready" state and turns on the white PRINT READY light. The line printer must be "ready" in order for it to operate and in order for the card reader to function.

The red STOP button (there is also one on the back side in the upper right-hand corner) stops the operation of the printer and releases it from the control of the computer. (Pushing the STOP button will usually also cause a normal, temporary stoppage of the card reader.)

The blue CARRIAGE RESTORE button causes the top of a new page of paper to be positioned in the print position. This button is not functional until the STOP button is pushed.

The END OF FORMS, FORMS CHECK, PRINT CHECK, and SYNC CHECK lights indicate abnormal conditions and cause stoppage of the printer. Secure the help of the Operations Supervisor (MS 12A) or an operator in the computer room if one of these lights comes on. DO NOT ATTEMPT TO "FIX" THE EQUIPMENT.

How To Use The Line Printer

1. Check the PRINT READY light. If it is not on, push the START button.

2. When the printer is finished with your job and if other people are running jobs behind yours, wait until your output comes into view at the top on the back side of the printer. Start a tear on one end of the perforation at the bottom of your last page, and then pull down on the paper sharply at a slight angle. The paper should tear cleanly if you do this quickly.

When the printer is finished with your job and no other people are running jobs behind yours, then push STOP, push CARRIAGE RESTORE three times only (more times just waste paper unnecessarily), and push START. The last page of your output will then be near the top on the back side of the printer. Then start a tear on one end of the perforation at bottom of your last page, and then pull down on the paper sharply at a slight angle. The paper should tear cleanly if you do this quickly.

3. It is always much easier to tear the paper when it is near the top of the printer on the back side. You can also do it more quickly near the top. Sometimes however, circumstances beyond your control require that you tear it off at the bottom of the printer; but ordinarily you can tear the paper near the top even though the printer is running.

4. If it becomes necessary to stop the printer while you are tearing off your output or for some other reason, then push the STOP button, correct the difficulty, and then push START.

APPENDIX VI

How to Get an 80/80 Listing With the System/360 User Terminal

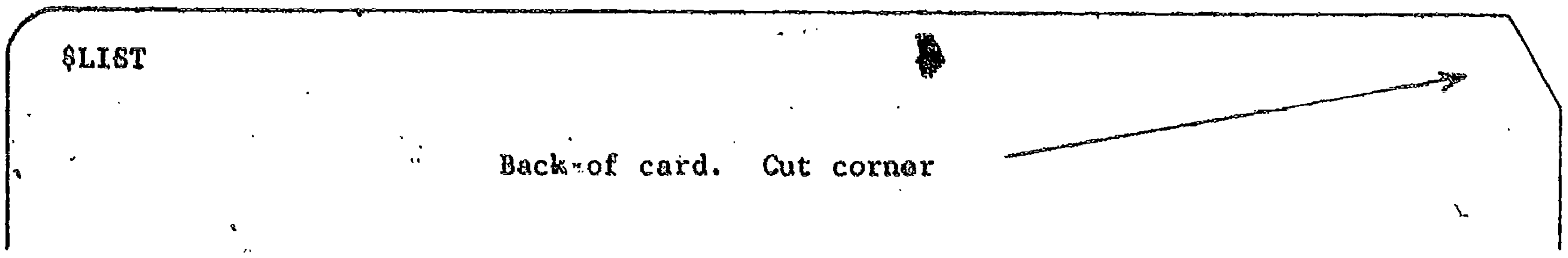
The 80/80 listing feature of the User Terminal produces a listing of punched or OMR cards on the line printer. This is a convenient way of checking cards for errors or for just reading what is punched or marked.

Two control cards are required:

```
123456789
$LIST
$ENDLIST
```

Place the $LIST card in front of the cards to be listed, and place the $ENDLIST card behind the cards to be listed. <u>Be sure to include the $ENDLIST</u> card; if you omit it, the terminal will remain in the list mode.

Usually there are $LIST and $ENDLIST cards in the card box beside the card reader or in the tray of control cards in the window where jobs are submitted. Replace the cards when you finish with them. If they are not available, then punch your own. It's better to punch them on the back side of the cards so that you can more easily identify your deck after it goes through the card reader by the top left corner of the two cards, since the top left corner is cut off the rest of the cards.


$LIST
Back of card. Cut corner

APPENDIX VII

Running Fortran Jobs on the IBM 1130 Computer

Section A of this appendix describes the control cards needed for 1130 Fortran jobs without user supplied subprograms. Use of FUNCTION and SUBROUTINE subprograms requires additional control cards which are described in Section B.

The 1130 computer is located in room MS 214. Normally the machine will be on, but occasionally you may find it off. If the machine is off, then you must "power up" and perform a "cold start." Section C tells you how to do this. If the machine is already on, then refer to Section D for instructions.

Section E outlines some major differences between 1130 Fortran and WATFIV.

Section F gives the error codes used with the 1130 Fortran compiler.

Section A

Control Cards

In order to run a Fortran job on the IBM 1130, the following control card setup is used.

```
         1111111111222222222233333333334444444444555555555566666666667777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890
// JOB     (or a "cold start" card, if doing a cold start)
// FOR
*IOCS(CARD,TYPEWRITER)
*LIST SOURCE PROGRAM
                                   }
                                   } Fortran source deck
                                   }
// XEQ
                                   }
                                   } Data cards, if any
                                   }
     One or two blank cards
```

The // JOB card prepares the computer for receiving and running a job. (It may be replaced by a "cold start" card when the cold start procedure is performed.)

The // FOR card loads the Fortran compiler and initiates the compile phase.

The *IOCS and *LIST SOURCE PROGRAM cards are control cards for the Fortran compiler. The *IOCS card prepares the computer to receive input from the card reader (logical unit number 2) and to put output on the typewriter/printer (logical unit number 1). The *LIST SOURCE PROGRAM card produces a listing of the program as it is compiled on the typewriter/printer. If you choose not to have a listing, this card may be omitted, greatly speeding up the compile phase.

The // XEQ card initiates the execution phase.

Since the card reader does not have an END OF FILE feature for reading the last card of the deck and ejecting it into the receiving hopper, one blank card insures that the last card of the deck will be read, but it will remain inside the card reader. Two blank cards will insure that the last card will be ejected into the hopper.

See Section B for information on control cards for running jobs with SUBROUTINE or FUNCTION subprograms.

Section B

Using Subprograms

On the 1130, subprograms must be compiled and stored on the disc before they can be called and used in the main program. For that reason, FUNCTION and SUBROUTINE subprograms must be placed before the main program in the program deck. Special control cards are also required. The setup of control cards and source decks is shown below.

```
         1111111111222222222233333333334444444444555555555566666666667777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890
// JOB T
// FOR
*LIST SOURCE PROGRAM
                                    } Subprogram source deck
// DUP
*STORE      WS  UA  Name
// FOR
*IOCS(CARD,TYPEWRITER)
*LIST SOURCE PROGRAM
                                    } Main program source deck
// XEQ
                                    } Data cards, if any
   One or two blank cards
```

The "T" on the // JOB card is necessary so that the storing of the subprograms on the disc will be temporary. If the "T" is omitted, then the subprograms will be stored permanently on the disc.

The *STORE card moves the compiled program from the work storage area (WS) of the disc to the user area (UA). The name of the subprogram must be punched beginning in column 21. (This is the same name used in the FUNCTION or SUBROUTINE statement, of course, and is not necessarily "Name" as shown on the card.)

If you have more than one subprogram, then set up control cards for each subprogram separately, as shown on the next page.

```
// FOR
*LIST SOURCE PROGRAM
                                  }
                                  } Subprogram source deck
                                  }
// DUP
*STORE      WS  UA  Name
```

Remember to place all subprograms before the main program.

Run jobs in the usual way, using instructions in Section C or Section D.

For additional information on control cards, see Section A.

Section C

Power Up and Cold Start

This procedure is to be used only if the computer is off--that is, the main POWER switch is in the OFF position and the POWER ON light on the card reader is off.

1. Power up.

   a. The POWER switch is on the operator's console in the upper right-hand corner. Position the switch to ON. The POWER ON light on the card reader should now be on.

   b. Open the door on the disc drive unit below and to the right of the console. (The entire front of the cabinet opens.) Find a switch labeled FILE. Move the switch to ON.

   c. Wait about 2 minutes, or until you hear a loud click. The computer should be ready for the start-up procedure then.

   d. Check the green FILE READY light above and to the left of the console keyboard. It should be on. If it isn't, get help.

2. Ready the card reader.

   a. Make sure that the card feed hopper on the card reader is empty.

   b. Push and hold momentarily the NPRO button on the reader; this will remove any cards that may be inside the reader.

   c. Place the proper control cards with your program (see Section A), and place a "cold start" card in place of the // JOB card. (There should be a cold start card in the box sitting on the card reader. If you are using a // JOB T card, place the cold start card in front of it.)

   d. Put your card deck into the card feed hopper, 9-edge away from you, face down, and push the START button on the card reader. The green READY light on the card reader should now be on.

3. Run the job.

   a. Push the PROGRAM STOP button on the console.

b. Push the IMM STOP button on the console.

c. Push the RESET button on the console.

d. Push the PROGRAM LOAD button on the console.

e. Check the green RUN light on the console; it should be on, and the computer should be operating. If it isn't, find help.

f. At the conclusion of the run, clear the card reader. Clear the feed hopper and press and hold momentarily the NPRO button to eject the last two cards that are inside the reader.

Section D

Running Jobs When the Computer is ON

<u>This procedure is to be used only if the computer is on -- that is, the main POWER switch is in the ON position, the POWER ON light on the card reader is on, and the FILE READY light on the console is on.</u>

1. Check the KB SELECT light on the left side of the console keyboard. If the light is on, then you must do the following before you can enter a job through the card reader:

    a. Using the keyboard, starting at the left margin, type

    //bJOB

    where "b" represents a blank.

    b. Push the EOF (end of file) button on the keyboard.

    c. Then type, starting at the left margin,

    //bTEND

    where "b" represents a blank.

    d. Push the EOF button on the keyboard again.

    e. Check the KB SELECT light. It should be off. If not, you probably didn't type the instructions correctly. Go back to step <u>a</u> and repeat the instructions carefully.

2. Ready the card reader.

    a. Be sure that the card feed hopper is empty.

    b. Push and hold momentarily the NPRO button on the card reader to remove any cards that may be left inside the reader.

    c. Place the proper control cards with your program deck (see Section A).

    d. Place your cards into the card feed hopper, 9-edge away from you, face down.

e. Push START on the card reader.

f. Check the READY light on the reader. It should be on. If not, try again or ask for help.

3. Run the job.

a. Push PROGRAM START on the console.

b. Check the green RUN light on the console. It should be on. If it is not, then either your control cards are incorrect or else you need to use a "cold start" card. Remove your cards from the feed hopper, and press and hold momentarily the NPRO button to eject the cards that are inside the reader. Check your control cards. If they are correct, then go to Section C, step 2c, and proceed to the end of Section C.

c. At the conclusion of the run, clear the card reader. Clear the feed hopper and push and hold momentarily the NPRO button to eject the last two cards that are inside the card reader.

Section E

1130 Fortran Compared to WATFIV

This list is far from exhaustive, being intended only to help you over some major hurdles.

1. Length of variable names.

   WATFIV: Maximum 6 characters.
   1130: Maximum 5 characters.

2. Logical IF and logical data.

   WATFIV: Logical capability available.
   1130: No logical capability available.

3. Alphameric or character data (single precision).

   WATFIV: Maximum of 4 characters associated with a variable name.
   1130: Maximum of 4 characters associated with a <u>real</u> variable name; maximum of 2 characters associated with an <u>integer</u> variable name.

4. Input/output unit numbers.

   WATFIV: Line printer, 6; card reader, 5.
   1130: Typewriter/printer, 1; card reader 2.

For compatibility between WATFIV and 1130 Fortran, use of variable I/O unit numbers is especially convenient. The program segments below illustrate this.

```
C  RUN ON WATFIV                 C  RUN ON 1130
      IN=5                             IN=2
      IOUT=6                           IOUT=1
      READ(IN,4)A                      READ(IN,4)A
      WRITE(IOUT,5)A                   WRITE(IOUT,5)A
```

A single DATA initialization statement may be used for initializing both unit numbers.

```
C  RUN ON WATFIV                    C  RUN ON 1130
     DATA IN,IOUT/5,6/                   DATA IN,IOUT/2,1/
     READ(IN,4)A                         READ(IN,4)A
     WRITE(IOUT,5)A                      WRITE(IOUT,5)A
```

5. Carriage controls.

WATFIV:	Blank, zero, +, 1, and /.
1130:	No carriage controls except /. Other carriage controls, if present, are simply printed, as any other character fields would be.

For example, below are two program segments illustrating what output is produced by each system.

```
C  WATFIV                           C  1130
       . . .                               . . .
     WRITE(6,1)                          WRITE(1,1)
   1 FØRMAT(7HOØUTPUT)                 1 FØRMAT(7HOØUTPUT)
       . . .                               . . .
$ENTRY                              // XEQ
                                    OØUTPUT
OUTPUT
```

Compatability between the two systems in the examples above can be achieved by using a / to get double spacing, as shown below.

```
C  WATFIV                           C  1130
       . . .                               . . .
     WRITE(6,1)                          WRITE(1,1)
   1 FØRMAT(/7H ØUTPUT)                1 FØRMAT(/7H ØUTPUT)
       . . .                               . . .
$ENTRY                              // XEQ

ØUTPUT                               ØUTPUT
```

Notice that the blank carriage control is "printed" on the 1130.

6. Variable dimensions in subprograms.

WATFIV:	Variable dimensions are permitted and recommended.
1130:	No variable dimensions are permitted. In general, using the same dimension in the main program and the subprogram is recommended.

7. Rounding of output.

WATFIV: Real numbers are rounded on output.
1130: Real numbers are not rounded, but are truncated on output.

The program segments below illustrate this.

```
C  WATFIV                          C  1130
      X=3.6789                           X=3.6789
      WRITE(6,1)X                        WRITE(1,1)X
    1 FORMAT(1H ,F4.2)                 1 FORMAT(1H ,F4.2)
         . . .                              . . .
$ENTRY                             // XEQ
3.68                                3.67
```

Section F

Error Codes

The error codes listed below are for the 1130 Disk Monitor System, Version 2, the 1800 Multiprogramming Executive Operating System (MPX), and the 1800 Time-Sharing Executive Operating System (TSX). Most of the error codes are the same for all three systems; where they differ in meaning, a separate definition is supplied for the system that is different from the others. In the table below, DM2 stands for the 1130 Disk Monitor System, MPX for the 1800 Multiprogramming Executive system, and TSX for the 1800 Time-Sharing Executive system. Some of the errors are caused by errors in control statements. For an explanation of these statements, refer to the appropriate manual: for DM2 -- Programming and Operator's Guide, Form C26-3717; for MPX -- Programmer's Guide, Form C26-3720; for TSX-Concepts and Techniques, Form C26-3703.

Error Code	Cause of Error
C01	Non-numeric character in statement number.
C02	More than five continuation cards, or continuation card out of sequence.
C03	Syntax error in CALL LINK or CALL EXIT statement, or, in TSX, CALL LINK or CALL EXIT in process program.
C04	Undeterminable, misspelled, or incorrectly formed statement.
C05	Statement out of sequence.
C06	Statement following STOP, RETURN, CALL LINK, CALL EXIT, GO TO, or IF statement does not have a statement number, or, in MPX or TSX, an MPX or TSX CALL statement does not have a statement number.
C07	Name longer than five characters, or name not starting with an alphabetic character.
C08	Incorrect or missing subscript within dimension information (DIMENSION, COMMON, REAL, or INTEGER).
C09	Duplicate statement number.
C10	Syntax error in COMMON statement.
C11	Duplicate name in COMMON statement.
C12	Syntax error in FUNCTION or SUBROUTINE statement.
C13	Parameter (dummy argument) appears in COMMON statement.
C14	Name appears twice as a parameter in SUBROUTINE or FUNCTION statement.
C15	DM2 and TSX: *IOCS control statement in a subprogram.
C16	Syntax error in DIMENSION statement.
C17	Subprogram name in DIMENSION statement.
C18	Name dimensioned more than once, or not dimensioned on first appearance of name.
C19	Syntax error in REAL, INTEGER, or EXTERNAL statement.
C20	Subprogram name in REAL or INTEGER statement, or, in DM2, a FUNCTION subprogram containing its own name in an EXTERNAL statement.
C21	Name in EXTERNAL that is also in a COMMON or DIMENSION statement.
C22	IFIX or FLOAT in EXTERNAL statement.
C23	Invalid real constant.
C24	Invalid integer constant.
C25	More than 15 dummy arguments, or duplicate dummy argument in statement function argument list.
C26	Right parenthesis missing from a subscript expression.
C27	Syntax error in FORMAT statement.
C28	FORMAT statement without statement number.
C29	DM2 and TSX: Field width specification greater than 145 columns. MPX: Field width specification greater than 155 columns.
C30	In a FORMAT statement specifying E or F conversion, w greater than 127, d greater than 31, or d greater than w, where w is an unsigned integer constant specifying the total field length of the data, and d is an unsigned integer constant specifying the number of decimal places to the right of the decimal point.
C31	Subscript error in EQUIVALENCE statement.
C32	Subscripted variable in a statement function.
C33	Incorrectly formed subscript expression.
C34	Undefined variable in subscript expression.
C35	DM2: Number of subscripts in a subscript expression, and/or the range of the subscript(s) does not agree with the dimension information. MPX and TSX: Number of subscripts in a subscript expression does not agree with the dimension information.
C36	Invalid arithmetic statement or variable; or, in a FUNCTION subprogram, the left side of an arithmetic statement is a dummy argument, or, in DM2 and TSX, is in COMMON.
C37	Syntax error in IF statement.
C38	Invalid expression in IF statement.
C39	Syntax error or invalid simple argument in CALL statement.
C40	Invalid expression in CALL statement.

Error Code	Cause of Error
C41	Invalid expression to the left of an equals sign in a statement function.
C42	Invalid expression to the right of an equals sign in a statement function.
C43	In an IF, GO TO, or DO statement, a statement number is missing, invalid, or incorrectly placed, or is the number of a FORMAT statement.
C44	Syntax error in READ, WRITE, or FIND statement.
C45	*IOCS record missing with a READ or WRITE statement (In DM2 and TSX mainline programs only).
C46	FORMAT statement number missing or incorrect in a READ or WRITE statement.
C47	Syntax error in input/output list; or an invalid list element; or, in a FUNCTION subprogram, the input list element is a dummy argument or is in COMMON.
C48	Syntax error in GO TO statement.
C49	Index of a computed GO TO is missing, invalid, or not preceded by a comma.
C50	*TRANSFER TRACE or *ARITHMETIC TRACE control record present, with no *IOCS control record in a mainline program.
C51	DO statements are incorrectly nested, or the terminal statement of the associated DO statement is a GO TO, IF, RETURN, FORMAT, STOP, PAUSE, or DO statement, or, in MPX or TSX, an MPX or TSX CALL statement.
C52	More than 25 nested DO statements.
C53	Syntax error in DO statement.
C54	Initial value in DO statement is zero.
C55	In a FUNCTION subprogram the index of DO is a dummy argument or is in COMMON
C56	Syntax error in BACKSPACE statement.
C57	Syntax error in REWIND statement.
C58	Syntax error in END FILE statement.
C59	DM2: Syntax error in STOP statement. MPX and TSX: Syntax error in STOP statement or STOP statement in process program.
C60	Syntax error in PAUSE statement.
C61	Integer constant in STOP or PAUSE statement is greater than 9999.
C62	Last executable statement before END statement is not a STOP, GO TO, IF, CALL LINK, CALL EXIT, or RETURN statement, or, in MPX or TSX, an MPX or TSX CALL statement.

Error Code	Cause of Error
C63	Statement contains more than 15 different subscript expressions.
C64	Statement too long to be scanned, because of Compiler expansion of subscript expressions or Compiler addition of generated temporary storage locations.
C65*	All variables in an EQUIVALENCE list are undefined.
C66*	Variable made equivalent to an element of an array in such a manner as to cause the array to extend beyond the origin of the COMMON area.
C67*	Two variables or array elements in COMMON are equated, or the relative locations of two variables or array elements are assigned more than once (directly or indirectly).
C68	Syntax error in an EQUIVALENCE statement; or an illegal variable name in an EQUIVALENCE list.
C69	Subprogram does not contain a RETURN statement, or, in TSX, a TSX CALL statement, or a mainline program contains a RETURN statement.
C70	No DEFINE FILE statement in a mainline program that has disk READ, WRITE, or FIND statements.
C71	Syntax error in DEFINE FILE statement.
C72	Duplicate DEFINE FILE statement, more than 75 DEFINE FILEs, or DEFINE FILE statement in subprogram.
C73	Syntax error in record number of disk READ, WRITE, or FIND statement.
C74	DM2: Defined file exceeds disk storage size. MPX and TSX: INSKEL COMMON referenced with two-word integer.
C75	Syntax error in DATA statement.
C76	Names and constants in a DATA statement not in a one to one correspondence.
C77	Mixed mode in DATA statement.
C78	Invalid Hollerith constant in a DATA statement.
C79	Invalid hexadecimal specification in a DATA statement.
C80	Variable in a DATA statement not used elsewhere in the program, or, in DM2, a dummy variable in DATA statement.
C81	COMMON variable loaded with a DATA specification.
C82	DATA statement too long to compile, because of internal buffering.
C83	TSX: TSX CALL statement appearing illegally.

*The detection of a code 65, 66, or 67 error prevents any subsequent detection of any of these three errors.

APPENDIX VIII

This appendix contains the "User Terminal WATFIV" section of the University Computer Center's System/360 User's Manual.

USER TERMINAL WATFIV

The present version of WATFIV which is being executed on the User's Terminal in MS 010, has the following capabilities and requirements:

(a) Compile time is unlimited.
(b) Execution time is limited to a maximum of 30 seconds, but can be cut down by use of the 'TIME' parameter on the $JOB card.
(c) Execution output pages is limited to 20.
(d) Core size is limited only by the amount of free low speed core available at any given time. This information can be obtained from the users display station.

Features

1. NAMELIST[1]
2. Direct-Access I/O[1]
3. CHARACTER variables
4. Debugging:
   - full sentence diagnostic messages instead of codes.
   - undefined array elements identified explicitly rather than by array name only.
   - more explicit error diagnostics.
   - other aids for localizing errors.
5. Compatibility with IBM's G, H Fortran compilers is increased:
   - computed GO TO's work as specified in C28-6515 when GO TO index is outside the allowable range.
   - array elements as arguments may be passed to subprogram parameters which are array names (see C28-6515 for rules).
   - variable dimensions work as specified by C28-6515.
   - character set conventions are compatible with G and H, i.e., treatment of $, &, ', @.
   - treatment of ENTRY points in function subprograms is as specified in C28-6515.
   - statement ordering conventions are followed; specification, statement function definitions, executable statements.
   - real constants of the form 1E2, i.e., without explicit decimal point, are now recognized.
6. A few more language extensions, e.g., multiple statements per card, are implemented.

WATFIV Control Cards

Three control cards - $JOB, $ENTRY, and $IBSYS - are required to run a program under WATFIV. The order of the cards is shown below which defines a WATFIV job.

```
$JOB - project number,social security number,parameters  name
       {FORTRAN program consisting of a main program and any number of
       {subprograms.
$ENTRY
       {any data required by the program
$IBSYS
```

[1]See IBM System/360 Fortran IV Language, Form C28-6515 for rules.

The control field $JOB is punched in columns 1 to 4 of the card, $ENTRY and $IBSYS in columns 1 to 6; column 5 and 7, respectively, must be blank. Accounting information and job parameters appear on the remainder of the $JOB card. Columns 8 to 80 of the $ENTRY and $IBSYS cards are ignored. The $ENTRY card is required to initiate execution of the compiled program even if no data is required.

<u>WATFIV JOB Card</u>

General Form:

$JOB XXXXX,SSS-SS-SSSS,TIME=t,PAGES=p,LINES=k,REGION=nK,

$$\text{KP=}\left\{\begin{matrix}\text{026}\\ \underline{\text{029}}\end{matrix}\right\},\left\{\begin{matrix}\underline{\text{CHECK}}\\ \text{NOCHECK}\\ \text{FREE}\end{matrix}\right\},\left\{\begin{matrix}\underline{\text{LIST}}\\ \text{NOLIST}\end{matrix}\right\},\left\{\begin{matrix}\underline{\text{NOLIBLIST}}\\ \text{LIBLIST}\end{matrix}\right\},\left\{\begin{matrix}\underline{\text{WARN}}\\ \text{NOWARN}\end{matrix}\right\},\left\{\text{NOSUBCHK}\right\}\ \text{name}$$

where:

XXXXX - a valid project number for the run

SSS-SS-SSSS - your Social Security number

t - an integer number representing the maximum number of <u>seconds</u> to allow for execution of the program. Default is TIME=5.

p - an integer number representing the maximum number of pages to allow the program to produce at execution time. Default is PAGES=20.

k - an integer number representing the number of lines printed per page. (The compiler uses 'k' to provide automatic page skipping at both compile and execution time.) Default is LINES=60.

n - an integer number representing the amount of working storage requested for the job. Default is REGION=20K.

$\left\{\begin{matrix}\text{026}\\ \underline{\text{029}}\end{matrix}\right\}$ - Choose 026 if the source program is punched on a model 026 (BCD) keypunch; choose 029 if punched on a model 029 (EBCDIC) keypunch.

$\left\{\begin{matrix}\text{CHECK}\\ \text{NOCHECK}\\ \text{FREE}\end{matrix}\right\}$ - The compiler will check, at execution time, for attempted uses of variables which have not been assigned a value (undefined variables) if CHECK is selected. Use of NOCHECK suppresses the check, resulting in a somewhat reduced execution time. Also, somewhat less object code is produced. RUN=FREE is the same as CHECK, but the compiler will initiate execution of the program even if it contains serious source errors. If an executable statement which contained a source error is subsequently encountered, execution is terminated.

$\left\{\begin{matrix}\underline{\text{LIST}}\\ \text{NOLIST}\end{matrix}\right\}$ - Choose LIST if the compiler is to produce a source listing of the program; NOLIST suppresses the listing.

{NOLIBLIST / LIBLIST} - Choose LIBLIST if the compiler is to produce a source listing of the subprograms automatically retrieved from a library; NOLIBLIST suppresses the listing of library routines. Note that the LIST/NOLIST and LIBLIST/NOLIBLIST parameters are independent.

{WARN / NOWARN} - Choose NOWARN if the compiler is to suppress all diagnostics of a severity less than a fatal error. (Error severities are discussed in the section on Diagnostics.) Choose WARN if the diagnostics are to appear in the source listing.

{NOSUBCHK} - Allows subscripts to take on any form as long as they do not exceed the space of the actual array. This will allow programs like the following to run. The SSP programs can be run in this manner.

```
      REAL A(10)
          .
          .
          .
      CALL ANY(A)
          .
          .
          .
      SUBROUTINE ANY(B)
      REAL B(1)
      DO 100 I=1,10
100   B(I)=0.
      RETURN
      END
```

NOTES:

1. The $JOB card should be punched on an orange card. The project number can begin in any column after column 5. The first blank encountered thereafter will terminate the card scan. One or more blanks must separate the user's name from the WATFIV parameters.

2. The parameters may be punched in any order, e.g.

   $JOB XXXXX,$SS-SS-SSSS,KP=026,TIME=10,NOLIST,PAGES=15 name

   Parameters may extend to column 80.

3. Default values (underlined) will be assumed for all parameters that are omitted from the $JOB Card.

4. If a parameter is mispunched, the scan for any remaining parameters is stopped, and default values will be assumed, e.g.,

   $JOB XXXXX,SSS-SS-SSSS,PAGES=10,NOWARN,TAME=20,KP=026,RUN=NOCHECK name

   The PAGES=10 and NOWARN parameters are recognized and used by the compiler, but defaults are assumed for all other parameters because of the misspelling of TIME.

5. If any parameter is specified more than once, the rightmost value is used, e.g.,

```
$JOB XXXXX,SSS-SS-SSSS,KP=026,TIME=15,LIBLIST,KP=029 name
```

The KP=029 parameter is recognized and used by the compiler.

Source Code Libraries for WATFIV

The capability for WATFIV to use a "source code" library for "unresolved" references is available on the terminal version of WATFIV. This means that users can put source code subroutines or function subprograms into this library and "CALL" them from WATFIV. It is optional to either print or not print this source code by using the LIBLIST/NOLIBLIST parameters on the $JOB Card. Should any user desire to put often used routines into this library and "CALL" them, he should contact the UCC librarian. A savings will result if FIVPAK compressed decks are stored rather than "one statement per card" Fortran source decks. See the subsection of FIVPAK and UNPACK.

Source Statement Compress and Unpack Routines (FIVPAK, UNPACK)

Purpose:

FIVPAK compresses "one statement per card" FORTRAN source decks into "multi-statements per card" decks useable in WATFIV. (UNPACK reverses the process).

This form of source input is efficient if programs are to be stored in source form in data sets on disks, since the results are:

(a) faster compile-time

(b) more cards in the same amount of disk space.[1]

Method:

Blanks are removed from all Fortran statements (including literal data using the H format code), except where they are imbedded with apostrophies. Therefore, a source deck written with the H format code should not be compressed. Comment cards are reproduced as read in.

```
          DATA A,B/2H *,' *'/
          X=5.0
       36 GO TO (3,8),I
```

is compressed into,

```
DATAA,B/2H*,' *'/;X=5.0;36:GOTO(3,8),I
```

The cards produced are sequence-numbered in increments of 10.

How to Use:

```
     CALL FIVPAK(NREAD,NPUNCH), or CALL UNPACK(NREAD,NPUNCH)
where    NREAD = unit number for input data
         NPUNCH = unit number for output data
```

Both programs must be called from a program run under WATFIV[2].

1. Approximate saving of 60% (typical example, 281 cards compressed to 111).
2. FIVPAK and UNPACK reside in WATFIV's source library WATFIV.WATLIB.

since they use CHARACTER variables; for example, to read cards from the reader and punch a new deck.

```
//JOBNAME JOB (XXXXX,SSS-SS-SSSS,time),'user-name'
// EXEC WATFIV
//WATFIV.SYSPUNCH DD SYSOUT=B
//WATFIV.SYSIN DD *
$JOB NOWARN name
     CALL  FIVPAK(5,7); STOP; END
$ENTRY
     one-statement-per-card deck to be compressed
$IBSYS
//
```

NOTE:

More than one program deck can be compressed using FIVPAK by placing a card with an asterisk (*) in column 1 between each complete deck. UNPACK does not require such a "separator" card. A 360 Special Run Submittal card indicating punched output must be included with the above job.

Control Cards to Edit Source Listings

Four new control cards have been added to WATFIV to be used for controlling the printing of compile-time listings. These are $PRINTON, $PRINTOFF, $SPACE and $EJECT.

When a $PRINTOFF card is placed in the source deck, the listing will terminate at that point. The $PRINTOFF card itself is not printed.

A $PRINTON card will allow the listing to be restarted if a previous $PRINTOFF card was used or a NOLIST option was punched on the job card. The $PRINTON card is printed.

The $EJECT and $SPACE cards cause the printer to skip to a new page and a new line respectively.

A source deck using these cards might be as follows:

```
$JOB XXXXX,SSS-SS-SSSS,NOLIST name
          cards not to be listed
$PRINTON
          cards to be listed
          END
$EJECT
          subprograms
$SPACE
          END
$ENTRY
          data cards
$IBSYS
```

These control cards can be used to advantage when a large program is being tested. By suppressing print in areas where code has not been changed, the user can save on machine printout time and thus have the job run more economically. Also, the $EJECT and $SPACE cards can be used for the final "production run" to make the output look more presentable.

Executing WATFIV in the OS Job Stream

When EXECuting WATFIV the following procedure should be followed:

1. Set up the deck as shown below and submit in MS 010.
2. The OS JOB (// JOB) card must be punched on an orange card, but the WATFIV $JOB card should not be on an orange card. Do not include the project number or Social Security number on the $JOB card.
3. A region of 110K plus the amount of working storage for the WATFIV job is required. For instance, if the user needs 40K for his job, then 150K would be specified in the region parameter. CLASS=B is required for region sizes greater than 127K. An example follows:

```
//JOBNAME JOB (XXXXX,SSS-SS-SSSS,2),'user-name',REGION=150K,CLASS=B
// EXEC WATFIV
//WATFIV.SYSIN DD *
$JOB TIME=20,KP=026,NOWARN,NOSUBCHK name
        WATFIV source deck
$ENTRY
        input data
$IBSYS
//
```

Language Extensions

WATFIV attempts to support the language described in the IBM publication "IBM System/360 FORTRAN IV Language", form C28-6515, subject to the subsection on "RESTRICTIONS". In addition, WATFIV supports a number of extensions to the language, which are described below. Uses of the language extensions, except for 1, 2, 12, and 13 are flagged with *EXTENSIONS* messages. These mean that the program is acceptable to WATFIV but will not likely compile on other compilers. The messages can be suppressed by use of the NOWARN parameter on the $JOB card.

1. Format-free I/O. This allows the programmer to do I/O without reference to a FORMAT statement. For example, the statement PRINT, A,B will cause the values of A and B to be printed with a standard format. Format-free I/O statements may have one of the following forms:

```
READ, list
PRINT, list
READ(unit,*,END=m1,ERR=m2) list
WRITE(unit,*) list
```

The I/O for the first two forms is done on the standard reader and printer units, i.e., 5 and 6 respectively. The asterisks in the last two forms imply format-free I/O, and "unit" may be a constant or variable unit number. The END and ERR returns are optional, as with the conventional READ statement.

Note that the two statements

```
READ,list
READ(5,*)list
```

are equivalent, as are

```
PRINT,list
WRITE(6,*)list
```

Some examples follow:

```
    READ,A,B, (X(I),I=1,N)
    PRINT,(J ,Z(J) ,J=N,K,L),I,P
99  WRITE(6,*)'DEBUG OUTPUT',99,X,Y,Z+3.5
    READ(I,*,END=27) (X(J),J=1,N)
```

Free input data items may be punched one per card, or many per card; in the latter case, the data items must be separated by a comma and/or one or more blanks. The first data item on a card need not start in column 1. A data item may not be continued across two cards, i.e., the end of a card acts as a delimiter.

Successive cards are read until enough items have been found to satisfy the requirements of the "list" part of the statement. Any items remaining on the last card read for a particular READ statement will be ignored since the next READ statement executed will cause a new card to be read.

It is perfectly valid to use format-free READ statements and conventional READ statements in the same program.

The forms of data items which may be used for the various types of FORTRAN variables are:

Integer	- signed or unsigned integer constant
Real	- signed or unsigned real constant in F,E or D forms
Complex	- 2 real numbers enclosed in parentheses and separated by a comma, e.g., (1.2,-3.8).
Logical	- a string of characters containing at least one T or F. The first T or F encountered determines the logical value.
Character	- a string of characters enclosed by quotes. If a quote is required as input, two successive quotes should be punched.

The type of a data item must match the type of the variable it is being read into.

A duplication factor may be given to avoid punching the same constant many times. For example, if we have

```
DIMENSION A(25)
READ,A
```

the data for the READ statement could be punched as

```
15*0.,10*-3.8
```

Examples:

(i)	source statement	READ,X,I,Y,J
	typical data	2.5 3,-7.9,-41
(ii)	source statements	COMPLEX Z(5) READ, (Z(I),I=1,3)
	typical data	(5.2,-16.0) 2*(0.,.5E-3)
(iii)	source statements	LOGICAL L1,L2,L3 READ,L1,L2,L3
	typical data	T .FALSE. , CAT
(iv)	source statements	CHARACTER A*1, B*3 READ,A,B
		'A','DOG'

For free output data items the compiler supplies formatting for list items output by format-free statements. Line overflow is automatically accounted for, i.e., several records may result from one output statement.

The formats used are:

Integer	-I12
Real*4	-E16.7
Real*8	-D28.16
Complex*8	- '(' E16.7 ',' E16.7 ')'
Complex*16	- '(' D28.16 ',' D28.16 ')'
Logical	-L8
Character*n	-An

2. CHARACTER Variable. This is a new type of variable which allows the manipulation of data in the form of character strings. A simple example of the use of a CHARACTER variable follows:

```
CHARACTER A*7
     .
     .
     .
A='FINALLY'
     .
     .
     .
```

The standard and optional length specifications which determine the number of characters that are reserved for each character variable are:

Variable Type	Standard	Optional
CHARACTER	1	$1 \le n \le 255$

A programmer may declare a variable to be of the character type by use of the:

(i) IMPLICIT specification statement.
(ii) Form of the explicit specification statement: CHARACTER

IMPLICIT Statement:

The type CHARACTER is permitted in the IMPLICIT statement with a specified length. If length is omitted, the standard length of 1 is assumed.

Example:

```
IMPLICIT CHARACTER*80 (A-D),CHARACTER ($,Z)
```

Explanation:

All variables beginning with the characters A through D are declared as CHARACTER type, each variable or array element 80 characters in size. All variables beginning with the characters $ and Z are declared as CHARACTER. Since no length specification was explicitly given, 1 character (the standard length for CHARACTER) is allocated for each variable.

CHARACTER Statement:

GENERAL FORM
CHARACTER*<u>s</u> <u>a</u>*s1(<u>k</u>$_1$)/<u>x</u>$_1$/,<u>b</u>*<u>s</u>$_2$(<u>k</u>$_2$)/<u>x</u>$_2$/,...,<u>z</u>*<u>s</u>$_n$(<u>k</u>$_n$)/<u>x</u>$_n$/
Where: *<u>s</u>,*<u>s</u>$_1$,*<u>s</u>$_2$,...,*<u>s</u>$_n$ are optional. Each s represents one of the permissible length specifications.
<u>a</u>,<u>b</u>,...,<u>z</u> represent variable or array names
(<u>k</u>$_1$),(<u>k</u>$_2$),...,(<u>k</u>$_n$) are optional. Each <u>k</u> is composed of of 1 through 7 unsigned integer constants separated by commas, representing the maximum value of each subscript in the array. Each <u>k</u> may be an unsigned integer variable only when the CHARACTER statement in which it appears is in a subprogram.
/<u>x</u>$_1$/,/<u>x</u>$_2$/,...,/<u>x</u>$_n$/ are optional and represent initial data values.

The information necessary to allocate storage for arrays (dimension information) may be included within the statement. However, if this information does not appear in a CHARACTER statement, it must appear in a DIMENSION or COMMON statement.

Initial data values may be assigned to variables or arrays by use of /<u>x</u>$_n$/ where <u>x</u>$_n$ is a constant or list of constants separated by commas.

This set of constants may be in the form "r* constant", where r is an unsigned integer, called the repeat constant. The initial data values may only be literal constants and must be the same length as or shorter than the corresponding variable or array element. Initial data values will be truncated from the right and diagnosed if too long, and they will be padded with blanks on the right if too short (see "Example 2" below).

An initially defined variable or a variable of an array may not be in . la common. In a labeled common block they may be initially defined only in a BLOCK DATA subprogram.

The CHARACTER statement overrides the IMPLICIT statement. If the length specification is omitted (i.e., *s), the standard length of 1 is assumed. If an array is used in a subprogram and is not in a COMMON, the size of this array may be specified implicitly by an integer variable of length 4 which can appear explicitly in the SUBROUTINE statement or implicitly in COMMON (adjustable dimensions).

Example 1:

```
     CHARACTER*80 CARDS (10), LINES*132(56,2),TCARD
```

Explanation:

This statement declares that the variable TCARD and the arrays named CARDS and LINES are of type CHARACTER. In addition, it declares the size of the array CARDS to be 10 and array LINES to be 112 (2 groups of 56 each). Each element of the array LINES is assigned 132 characters for a total of 14,784 (112 times 132) for the array.

Each element of the array CARDS and the variable TCARD is assigned 80 characters (the length associated with the type). The array CARDS is assigned a total of 800 characters.

Example 2:

```
     CHARACTER X*3(4)/'ABC','DEFG','HI','JKL'/
```

Explanation:

This statement declares that the array of four elements of three characters each named X has initial values:

X(1)	ABC
X(2)	DEF
X(3)	HI
X(4)	JKL

The statement is incorrectly written, and the value specified for X(2) has been altered by truncation.

Multiple Assignment Statements.

Statements of the form

$v_1 = v_2 = \dots = v_n =$ expression

are allowed, where v_1, [illegible], etc., represent variable names or array elements. The effect is that of the sequence of statements

```
v  = expression
 n
v    = v
 n-1    n
  .
  .
  .
v  = v
 1    2
```

E.g., A = B(5) = C = 1.5

. Expressions in Output Lists.

Expressions may be placed in output statements, e.g.,

```
WRITE(6,2) SIN(X)**2,A*X+(B-C)/2
```

The expression may not, however, start with a left parenthesis becau he compiler uses this as a signal that an implied DO follows in the list. For example:

```
PRINT, (A+B)/2
```

would result in an error message. However, the equivalent

```
PRINT, +(A+B)/2
```

is acceptable.

Note that CHARACTER constants are forms of expressions acceptable in output statements, e.g.,

```
PRINT,'VALUE OF X=',X
```

. Initializing of Blank Common.

Variables in blank common may be initialized in DATA or type statements, e.g.,

```
COMMON X
INTEGER X/3/
```

. Initializing Common Blocks.

Common blocks may be initialized in other than BLOCK DATA subprograms.

. Implied DO's in DATA Statements

Implied DO's are allowed in DATA statements, i.e., a statement of the form

```
DATA (C(I), I=1,5,2)/3*.25/
```

is valid.

In fact,

```
DATA (A(I), I=L,M,N)/ constant list/
```

is acceptable if L,M,N have been previously initialized and at least $\left[\frac{M-L}{N}\right]+1$ constants are present in the constant list.

. Subscripts in Statement Function Definitions.

Subscripts may be used on the right-hand side of statement function definitions, e.g.,

```
F(X) = A(I)+X + B(I)
```

Subscripts may be logical, complex, or character values.

The real part of a complex value is converted to an integer, and this value is used for indexing into the array. For example, if Z is complex, and A is an array, then A(Z) is equivalent to A(INT(REAL(Z))).

- Transfer statements as object of a DO.

A logical IF statement used as the last statement (object) of a DO loop may contain a GOTO of any form, a PAUSE, STOP, RETURN, or arithmetic IF statement. E.g..

```
            DO 25 I=1,N
              .
              .
              .
         25   IF (X.EQ.A(I)) RETURN
```

- Exceeding the continuation card limit.

A statement may be continued on 10 continuation cards.

- Multiple statements per card.

WATFIV allows the programmer to punch more than one statement on a single card. This is particularly suitable for programs that are to be stored on libraries since less direct-access storage space is required, and fewer input operations are necessary to retrieve a subprogram.

The rules for this feature are:

(a) Only columns 7-72 may be used for statements.
(b) A semicolon is used to indicate the end of a statement.
(c) The normal continuation card rules are used for a statement which is to be continued beyond column 72.
(d) Statement numbers appear in columns 1-5, as usual, or following a semicolon and followed by a colon. They may not be split onto a a continuation card.
(e) Comment cards and FORMAT statements must be punched in the conventional manner.

```
              Column  6
E.g.,         25       A=B;C=D;39:PRINT, A,B,
                      * C,D;X=A+B*C+D
                       PRINT, X; 99: STOP;END
```

This could be punched in the conventional manner as

```
              25      A=B
                      C=D
              39      PRINT,A,B,C,D
                      X=A+B*C+D
                      PRINT,X
              99      STOP
                      END
```

- Comments on FORTRAN statements.

The compiler terminates the left-to-right scan of a particular card when a ǂ (pronounced 'zigamorph', and punched as a 12-11-0-7-8-9 multi-punch) is encountered. Effectively, this means comments may follow a FORTRAN statement on the same card if a ǂ is used to terminate the FORTRAN statement.

Note that a ǂ is unprintable, as well as being almost unpunchable.

```
        E.g.,    X=A+SIN(Y)ǂ     EVALUATE X
```

- The DUMPLIST and ON ERROR GO TO statements have been implemented in WATFIV as Debugging aids.

Restrictions

The user of WATFIV should take note of the following restrictions in language and facilities provided by the compiler.

1. The name of a common block must be unique, i.e., it may not also be used as the name of a variable, array, or statement function. This is in violation of C28-6515.
2. The concept of the extended range of a DO loop defined in C28-6515 is not supported.
3. The service subprograms DUMP and PDUMP defined in Appendix C of C28-6515 are not supported.
4. The Debug Facility described in Appendix E of C28-6515 is not supported.
5. There are no facilities in WATFIV which corresponds to the FORTRAN G/H options MAP, EDIT, XREF, OPT=, DECK, LOAD, NAME=, LIST.
6. The Extended Error Message facility is not supported.
7. No overlay facility is available; no 'module map' is produced.
8. The number of continuation cards, as well as the use of operator messages with STOP and PAUSE statements, are installation options.
9. No more than 255 DO statements are allowed in a program segment.
10. FORMAT(is a reserved character sequence when used as the first 7 characters of a statement. It is the only reserved character sequence. For example,

    ```
    FORMAT (I) = 3.5
    ```

    will result in FORMAT error messages, whereas

    ```
    X=FORMAT (I)
    ```

    is legal, assuming FORMAT to be an array or function name.
11. WATFIV is a 'one-pass' compiler, and requires several restrictions on statement ordering. These are:

    (a) Specification statements referring to variables used in NAMELIST or DEFINE FILE statements must preceed the NAMELIST or DEFINE FILE statements.

    (b) COMMON or EQUIVALENCE statements referring to variables used in DATA or initializing type statements must preceed the DATA or initializing type statements.

    ```
    E.g.,      REAL  I/5.2/
               COMMON I
    ```

    will produce error messages, whereas,

    ```
               COMMON I
               REAL I/5.2/
    ```

    is acceptable.

    (c) A variable may appear in an EQUIVALENCE statement and then in a subsequent explicit type statement only if the type statement does not declare the length of the variable to be different than could be assumed for it, based on the first letter of the variable name, at the time of its appearance in the EQUIVALENCE statement.

    For example,

    ```
    EQUIVALENCE  (A,B)
    REAL*8 B
    ```

    will produce an error message, whereas

    ```
    REAL*8 B
    EQUIVALENCE (A,B)
    ```

    will not. Note that

    ```
    EQUIVALENCE (A,B)
    INTEGER B
    ```

    is acceptable since the length of B is not changed by the type statement.

Incompatibilities with WATFOR

The most likely cause of difficulty is the use of arrays as subprogram arguments; this will be discussed last. Let's take care of the easy ones first.

1. WATFOR does not support the NAMELIST, Direct Access I/O, CHARACTER variable language features.
2. WATFOR does not support the LIST/NOLIST, LIBLIST/NOLIBLIST, WARN/NOWARN job options.
3. WATFIV issues warnings if the proper ordering to statements is not followed. The proper order is specification statements before statement function definitions before executable statements.
4. With WATFIV, DO-loops may be nested to any depth.
5. A half-word integer variable may not be used as a unit number in an I/O statement with WATFIV.
6. WATFOR does not accept source statements in compressed form, i.e., more than one statement per card.
7. If the index of a computed GOTO is negative or zero control transfers to the next executable statement with WATFIV; this follows the specifications of C28-6515. Under WATFOR, a terminating error message is given.
8. WATFOR gives special treatment to the $ in IMPLICIT statements. WATFIV assumes it follows Z in alphabetical order; this is the convention of C28-6515.
9. If a function subprogram has additional entry points, WATFOR does not 'equivalence' the variables which are the names of the function and its entry points. WATFIV does this, as prescribed by C28-6515.
10. The conventions, used by WATFIV, for intermixing EBCDIC and BCDIC characters in source programs are slightly different than those used by WATFOR.
    (a) WATFIV does not allow intermixing of the two quote marks in a program.
    (b) If KP=26 is specified, WATFIV uses '$' to denote a statement number argument; WATFOR uses a 12-8-6 multipunch (EBCDIC '+') for this.
11. WATFIV treats arguments passed to subprogram parameters which are arrays differently than does WATFOR.
    (a) WATFIV allows the actual argument to be an array element or a simple variable.
    (b) WATFIV uses the dimensions declared for the dummy array in the called subprogram. This ensures compatibility with FORTRAN G/H, and object time dimensions work as specified by C28-6515.

Under WATFOR, the dimensions for a dummy array are ignored at execution time. When an array is passed from subprogram to subprogram, the dimensions that are declared for it in the program segment in which it is actually allocated storage are implicitly passed as well. These dimensions are then used for subscript calculations.

Point (b) implies that the results will be different under WATFOR and WATFIV if the dimensions of the dummy array differ from those of the actual array passed.

Incompatibilities with FORTRAN's G/H

Note that the differences listed below do not include the language extensions and restrictions. Nor do they include differences which arise either because object programs compiled under G/H are freely allowed to violate the language rules defined by C28-6515 (e.g., passing an argument of type INTEGER to the SQRT subroutine), or because the G/H compilers accept syntax not defined in C28-6515, e.g.,

```
WRITE(6,2) (A(I), A(2))
```

The major causes of differences between WATFIV and FORTRAN's G/H are likely to be the treatment of FORTRAN-supplied functions and number conversions.

1. WATFIV provides execution-time page skipping, controlled by the LINES= job-parameter.
2. WATFIV allows any number of contiguous comments cards; comments cards may precede a continuation card.
3. WATFIV uses only the high-order byte of a logical quantity in logical operations. For example, if A and B are of type LOGICAL *4, execution of the statement

   ```
   A = B
   ```

   causes only one byte to be moved.
4. DO-loops may be nested to any depth in WATFIV.
5. WATFIV supports both EBCDIC and BCDIC '+' as a carriage control character.
6. WATFIV considers the program to be in error if it executes a RETURNi statement in which the value of 'i' is zero, negative, undefined, or greater than the number of statement number arguments which appeared in the argument list of the CALL statement which invoked the subprogram from which the return is being made.
7. WATFIV prints no message equivalent to the IHC210I ("old PSW is ...") message when interrupts occur.
8. With WATFIV, a use of T format which does a 'backward' tab in an output buffer does not cause existing characters in the buffer to be blanked out. For example, consider the statements:

   ```
      K= 9
      J= 1
      WRITE (6,7)K,J
    7 FORMAT (' $$$.00',T3,I2,T6,I2)
   ```

   With WATFIV, the line appears as:

   ```
   $$9.01
   ```

   With G/H, it appear as:

   ```
   $ 9. 1
   ```

   Actually, this is a consequence of the fact that WATFIV's formatting routines assume the buffer to be blanked before any filling of it occurs, i.e., only significant characters are moved into the buffer.
9. REAL*4 values are printed with a maximum of 7 significant digits. If the output format specification calls for more, i.e., E20.10, zeroes are supplied on the right.
10. WATFIV treats FORTRAN-supplied functions differently than G/H as follows:
    (a) The function's type must be explicitly declared if it is different than can be assumed from the implicit rules.
    (b) WATFIV makes no distinction between 'in-line' and 'out-of-line' functions; all functions are out-of-line.
    (c) WATFIV eveluates all functions that require complicated approximation formulae in double precision, i.e.,

    ```
    SQRT(X)
    ```

    is calculated as, essentially,

    ```
    SNGL(DSQRT(DBLE(X)))  .
    ```

11. WATFIV handles FORMAT statements differently than G and H as follows:
    (a) Commas are not required between format codes in WATFIV.
    (b) WATFIV allows more than the maximum number of continuation cards for FORMAT statements.
    (c) WATFIV does not allow group or field counts to be zero.
12. Execution-time data cards read on the standard card reader unit by WATFIV-compiled programs may not contain a $ in column 1.
13. With WATFIV a particular labeled COMMON block can be initialized in mor than one BLOCK DATA subprogram. This allows undetected violations of rul specified in C28-6515.

DIAGNOSTICS

WATFIV issues compile-time diagnostics at three levels of severity - EXTENSION, WARNING and ERROR. A diagnostic is generated in-line in the source listing, immediately below the statement in which the condition was detected.

An EXTENSION message results if you used an extension of the FORTRAN language allowed by WATFIV. The diagnostic is issued so that you can eliminate the problem should you ever wish to re-compile with IBM's G or H compilers.

A WARNING is issued for language violations for which the compiler can take some reasonable corrective action, e.g., truncating a name of more than 6 characters.

An ERROR is issued when a language violation severe enough to prevent execution is encountered. In this case, the compiler will normally inhibit execution of the program, unless you have specified RUN=FREE.

At execution time, all errors are fatal[1] in the sense that the compiler will terminate the current job and proceed to the next job in the batch. For execution-time errors, the compiler generates a diagnostic and a subprogram traceback in the printed output. This gives the line number of the statement in which the error occurred, the name of the subprogram in which the error occurred, the name of the subprogram which called it, etc., all the way back to the main program which is referred to as M/PROG. (The line number of each statement appears to the left of it in the source listing. This line number is compiler generated, and is distinct from and should not be confused with any FORTRAN statement number the programmer may have assigned to a statement.)

1. Exception: If an I/O error occurs and the programmer has specified an ERR= return in the affected I/O statement, an error message is given and execution proceeds at the statement specified by the ERR=.

Control Cards for Certain Diagnostics

Four Control cards have been added in the VIL2 WATFIV compiler. The $WARN and $NOWARN cards control the printing of compiler generated warning and extension messages; the $CHECK and $NOCHECK cards control the compiler's checking of undefined variables.

When a $NOWARN card is placed in the source deck, all warning and extension messages will be suppressed from that point on. A $WARN card will allow the warning and extension messages to be restarted if a $NOWARN card was used or the NOWARN option was punched on the job card.

1. When a $NOCHECK card is placed in the source deck, from that point on, the compiler bypasses the generation of object code to check for undefined variables at execution time. A $CHECK card causes the compiler to generate the checking code if a $NOCHECK card appeared previously, or NOCHECK was specified (or defaulted) on the $JOB card.

The source deck using these new cards might be as follows:

```
$JOB XXXXX,SSS-SS-SSSS,NOWARN name
        compile with "CHECK"
        no warning and extension messages
$WARN
        warning and extension messages may be printed
$NOCHECK
        compile  with "NOCHECK"
        END
$CHECK
        subprograms
        compile with "CHECK"
        END
$ENTRY
        data cards
$IBSYS
```

2. The $WARN/$NOWARN and $CHECK/$NOCHECK cards allow local control of their functions. This can be useful if a program is being debugged in stages, with routines being added or changed over a sequence of runs. If a $NOCHECK card (or the NOCHECK job option) can be used because a segment of a program is known to be free of undefined variables, several advantages can result:

-less object code is generated; thus, a somewhat larger program can be compiled for a given amount of available memory.
-the program will run somewhat faster since the checking code is not executed.

Additional WATFIV Debugging Aids

Some new debugging aids have been added in the VIL2 version of WATFIV. They are the DUMPLIST statement, the ON ERROR GOTO statement, and a statement trace facility.

1) The DUMPLIST statement is designed especially as a program debugging aid; it is used as follows:

(i) A DUMPLIST statement is essentially a NAMELIST statement, except that the work DUMPLIST replaces the word NAMELIST. The usual rules for NAMELIST statements apply. Sample statements are:

```
DUMPLIST /XXX/A,XYZ,APE/LOK/XX,NEXT
DUMPLIST/THIS/N,TWO,SIX,OLD
```

(ii) A DUMPLIST list name need never appear in a READ or WRITE statement.

(iii) A DUMPLIST statement has no effect unless the program in which it appears is terminated because of an error condition; then WATFIV will automatically generate NAMELIST - like output of all DUMPLIST lists appearing in program segments which have been entered. The values printed are those which the variables had when the program was terminated.

To avoid producing too much output, only a few key variables should be placed in DUMPLIST statements.

2) The ON ERROR GOTO statement allows a program which has an error to recover and take some alternate and possibly corrective action, such as giving diagnosis. This feature can only be executed once in a program (to prevent infinite loops) however, any number of ON ERROR GOTO statement may appear in the source program. The last ON ERROR GOTO statement encountered before an error occurs is the one which is executed.

A program using this feature might be as follows:

```
$JOB XXXXX,SSS-SS-SSSS name
       ON ERROR GOTO 50
       I=0
5      READ(5,*,END=40)A
       I=I+1
       PRINT,A
       GO TO 5
50     PRINT,'CARD NUMBER', I, 'IS INVALID'
40     STOP
       END
$ENTRY
$IBSYS
```

The ON ERROR GOTO statement is not an executable statement; however, it can be placed anywhere in the program. It is not advisable to place an ON ERROR GOTO statement within the range of a DO-loop as no checking is performed to determine if the transfer at execution time will be valid (i.e., infinite looping may result).

3) An execution time statement trace of "ISN trace" feature is now available. The trace is turned on using a $ISNON card and is turned off using a $ISNOFF card. At least one executable statement must precede a ISNON. A sample program follows:

```
$JOB XXXXX,SSS-SS-SSSS name
       A=1
       J=3
$ISNON
       (statements to be traced)
$ISNOFF
       STOP
       END
$ENTRY
$IBSYS
```

INTERRUPTS

This section provides information on the treatment of interrupts that may occur during the execution of a FORTRAN program.

Normally, WATFIV terminates execution of the program at the first occurrence an exponent overflow, exponent underflow, fixed divide, or floating divide interrup.. However, a library subroutine, TRAPS, is provided to allow the programmer to accept more interrupts of the types just mentioned. Thus, with appropriate uses of subroutine DVCHK and OVERFL, a programmer may provide, to some extent, his own treatment of interrupts.

A call to TRAPS may have up to five integer valued arguments, and these correspond to the number of fixed overflows, exponent overflows, exponent underflows, fixed divide, and floating divide interrupts the programmer wishes. The arguments of TRAPS set up internal counters used by the compiler's interrupt routine. The latter routine decrements the appropriate counter by 1 when an interrupt occurs; when any counter reaches zero, the program is terminated.

TRAPS may be called (and subsequently recalled) at any point in the main program or a subprogram to set (or reset) the interrupt counters. Arguments of TRAPS are screened so that the absolute value of any negative argument is used as a positive count, and a zero value is taken to mean that the current value of the corresponding interrupt counter should be left unchanged.

EXAMPLES:

1. CALL TRAPS (0,5,7,-3,1)

   sets the interrupt counters so that the program will be kicked off on the occurrence of the first of the:

   -5th exponent overflow, or
   -7th exponent underflow, or
   -3rd fixed divide, or
   -1st floating divide exception following the execution of this call to TRAPS.

   The statement CALL TRAPS (0,5,7,3) has the same effect.

2.
```
LUNFLO = 100
LOVFLO = LUNFLO
CALL TRAPS (0, LUNFLO, LOVFLO)
```

   sets the counts to kick off the program on the occurrence of the first of the:

   -100th exponent overflow, or
   -100th exponent underflow, or
   -1st fixed divide, or
   -1st floating divide exception following the execution of this call.

3. CALL TRAPS (14)

   sets the fixed overflow counter to 14. Kickoff would occur at the 1st exponent overflow, underflow or divide exception or the 14th fixed overflow if the installation has activated this interrupt. NOTE that the distributed version of WATFIV operated with this interrupt masked off, and furthermore, that this is the normal mode of operation of FORTRAN G/H.

OVERFL, DVCHK

These routines function as follows:

CALL DVCHK (j)

j is an integer variable that is set to 1 if the (pseudo-) divide-check indicator was on, or to 2 if off. After testing, the indicator is turned off.

The indicator is set on when a fixed or floating divide exception occurs.

CALL OVERFL (j)

j is an integer variable that is set to reflect the most recent setting of a pseudo-indicator. The variable j is set to 1 if an exponent overflow was last to occur, to 2 if no exponent overflow or underflow condition exists, or to 3 if an exponent underflow was last to occur. After testing, the indicator is set for no condition, i.e., to 2.

NOTES:

1. The compiler interrupt routine loads the affected machine floating-point register with zero or the properly signed largest floating-point number for exponent underflow or overflow, respectively.
2. The five interrupt counters are initialized by the compiler to 1 at the start of each program. The divide-check and overflow indicator are not initialized; it is the programmer's reponsibility to do this, e.g., by dummy calls.
3. The terminating message is the only indication given by the compiler that interrupts have occurred. It is the programmer's responsibility to monitor these using OVERFL and DVCHK.
4. WATFIV operates with the fixed overflow and significance interrupts masked off entirely.
5. WATFIV automatically corrects for boundary alignment errors at execution time, but this is done with some overhead. Thus, programmers are advised to ensure that operands are aligned properly, where possible, by steps taken at the source level.

UNIT #1 ACTIVITIES TABLE

Assessment Task

1. Obtain about a dozen OMR cards. There is a bin with a sign "OMR Cards" above it in room MS 09 in the basement.

2. Mark OMR cards as directed below, one card per direction. The letter "O" is indicated by "Ø" to distinguish it from zero, which is just "0."

   Card 1: Mark the "CØMMENT" box on the card, and mark SAMPLE PRØGRAM beginning in column 1 of the card.

   Card 2: Mark the "CØMMENT" box, and mark your name beginning in column 1.

   Card 3: Mark A=2.1 beginning in column 1.

   Card 4: Mark B=2.6 beginning in column 1.

   Card 5: Mark C=0.6 beginning in column 1.

   Card 6: Mark AN=3.0*A+B/(A-C) beginning in column 1.

   Card 7: Mark WRITE in the keyword block of the card, and then mark (6,4)AN starting in column 1.

   Card 8: Mark a 4 in the "unit" column of the statement numbers block of the card; mark FØRMAT in the keyword block; and then mark (1HO,F10.1) starting in column 1.

   Card 9: Mark STØP in the keyword block.

   Card 10: Mark END in the keyword block.

3. You will need three control cards for running your job on the computer: $JOB, $ENTRY, and $IBSYS. The $JOB card is an orange color and will be given to you by your instructor. (APPENDIX III, Section B, contains a detailed discussion of the control cards, but probably you should save that till later.) Make your own $ENTRY and $IBSYS cards by marking the $ENTRY box on one card and the $IBSYS box on another card. These boxes are located in the upper left center of the OMR card.

4. Arrange your cards in the following order:

```
$JOB
CØMMENT  SAMPLE PRØGRAM
CØMMENT  Your name
               A=2.1
               B=2.6
               C=0.6
               AN=3.0*A+B/(A-C)
               WRITE(6,4)AN
             4 FØRMAT(1H0,F10.1)
               STØP
               END
$ENTRY
$IBSYS
```

5. Go to APPENDIX III, Section A, and take a self-guided tour of the Computer Center facilities.

6. Go to APPENDIX III, Section C, and follow the directions for running a job very carefully.

7. Now that you have mastered the User Terminal and have your printed output in hand, check the output for error or warning messages. If you have either or both, then probably you have marked your cards incorrectly or your cards are not in the proper order. Check the cards carefully, make corrections, and try again. If you need help, see your instructor.

8. When your computer output contains no errors and has 8.0 printed as the "answer" (after $ENTRY), then take it and your program deck (remove the control cards first) to your instructor. If he approves your work, then you have passed the first hurdle!

UNIT #2 ACTIVITIES TABLE

1. Read Section 4-1, pages 59-60, of Fortran IV Programming for Engineers and Scientists by Murrill and Smith.

UNIT #3 ACTIVITIES TABLE

All references are to FORTRAN IV Programming for Engineers and Scientists by Murrill and Smith.

1. Read the introduction to Chapter 2 and Sections 2-1 and 2-2, pages 17-21.
2. Read Sections 2-3, 2-4, and 2-5, pages 21-26.
3. Read Section 2-6, pages 26-28, and 2-8, pages 30-31.
4. Work as many of the Exercises, pages 31-35, as you feel a need for. (Solutions to Exercises marked with a dagger † are given in Appendix E, page 252.) If you need assistance, see your instructor.

UNIT #4 ACTIVITIES TABLE

References are to Fortran IV Programming for Engineers and Scientists by Murrill and Smith.

1. Read Section 2-7, pages 28-30.
2. Read Section 4-2, pages 60-63.
3. Section 2-6, pages 26-28 discusses statements of the form

   N = N + 1

   which is an example of a counter that counts by ones.

UNIT #5 ACTIVITIES TABLE

1. Read Chapter 1, pages 1-16, of Fortran IV Programming for Engineers and Scientists. You don't need to bother too much with all the details; try to get the general ideas. Pay particular attention, however, to the discussion and italicized terms in Sections 1-5 and 1-6. The concepts of compiling and execution are of great importance.

2. You will need three control cards for running your job on the computer: $JOB, $ENTRY, and $IBSYS. The $JOB card is an orange color and will be given to you by your instructor.
Read Appendix III, Section B, which tells you more about the control cards.
The simplest procedure at this point is for you to punch your own $ENTRY and $IBSYS cards, as described in Appendix III, Section B.

3. Arrange your cards in the following order:

```
      $JOB
                 } Comment cards
              TRY1=2.1
              TRY2=2.6
              TRY3=0.6
              ANS=3.0*TRY1+TRY2/(TRY1-TRY3)
              WRITE(6,4)ANS
            4 FØRMAT(1H0,F10.1)
              STØP
              END
      $ENTRY
      $IBSYS
```

4. Next get an "80/80 listing" of your program. This listing provides a convenient way of checking your program and control cards to make sure that the cards are in the proper order and that there are no keypunching errors. Read Appendix VI, which tells you how to get an 80/80 listing with the user terminal.

Since you have already used the card reader and the line printer in UNIT #1, you shouldn't have any trouble, if you read the instructions on the wall and do what they say. If you do run into problems, read Appendix IV on the card reader and Appendix V on using the line printer.

Once you get the 80/80 listing, check it for errors. If there are errors, then make the necessary corrections, and get a new listing.

Keep the listing of the correct program, since you will need it later.

5. Since you have already run a job on the user terminal in UNIT #1, then you may be able to go ahead and run your job without further ado. But, if you're not sure what to do, then do a and b below.

   a. Do you need to take the self-guided tour again? If so, then go to Appendix III, Section A.

   b. Go to Appendix III, Section C, and very carefully follow the directions for running a job.

6. When you have the printer output in hand, examine it for errors. If you have either error or warning messages, then you probably have not punched your cards correctly or else one or more cards

are out of order. Check the cards carefully, make corrections, and try again. If you need help, see your instructor.

7. When your computer output contains no errors, and has 8.0 printed as the "answer" (after $ENTRY), then you are finished, if you have an 80/80 listing of the correct program. Keep the printed output, the listing, and your program deck for the assessment task.

UNIT #6 (COMSC) ACTIVITIES TABLE

Unless otherwise indicated, all text references are to Fortran IV Programming for Engineers and Scientists by Murrill and Smith.

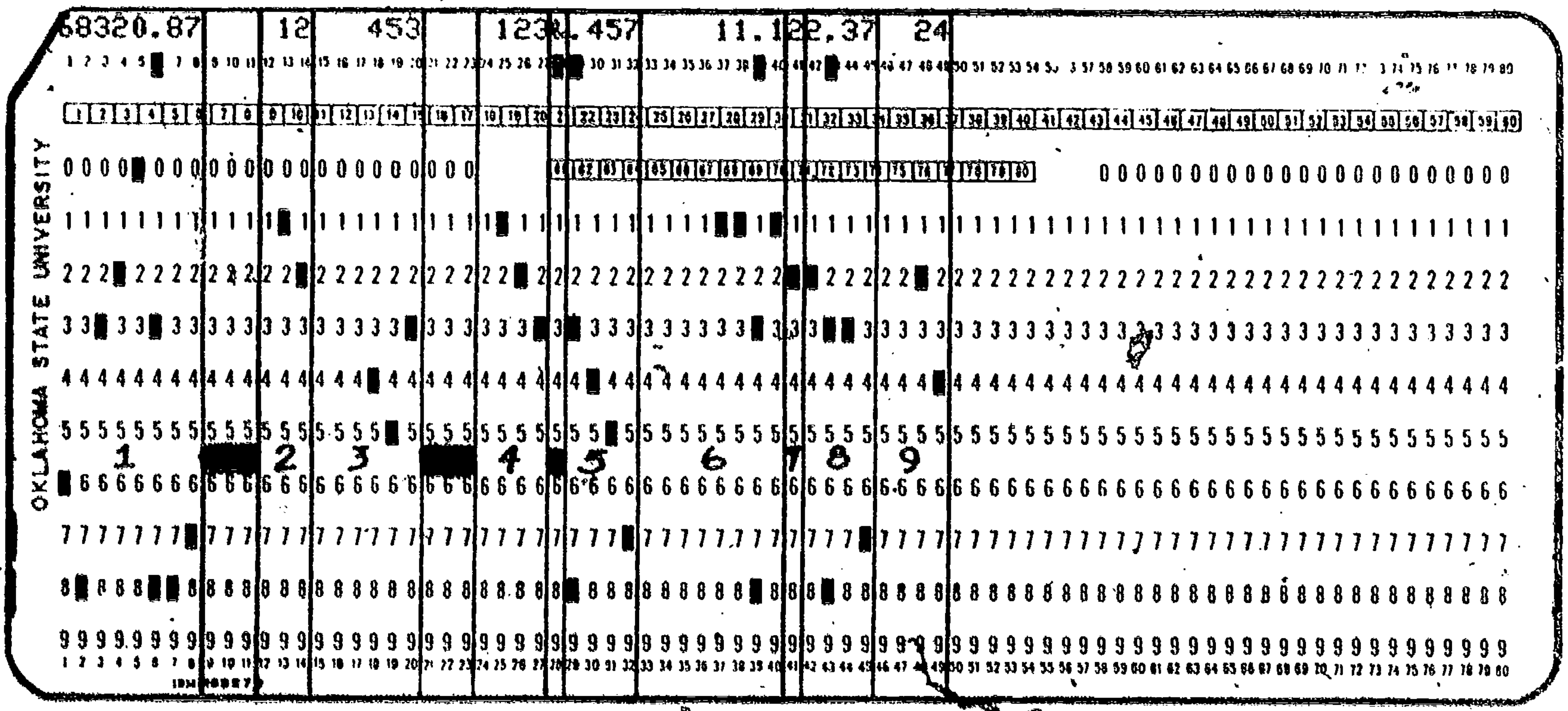
1. Read carefully chapter 3. Study the examples in Figures 3-2 and 3-3.

2. Complete as many of the exercises at the end of Chapter 3 (pages 55-58) as you feel are necessary to learn about input/output using I, F, X, H and 'literal' type conversions. In particular do exercises 3-15 and 3-18 so you can check your results. Be sure to include trial data cards as stressed in the italics on page 56.

   To be certain that you understand the materials properly, punch up a complete program deck including control cards and data for either exercise 3-15 or 3-18 and run it on the computer.

3. Nine fields of various widths are punched on the accompanying data card. These fields are indicated by the numbers 1, 2, 3, ..., etc. Decimal points are punched in all fields which contain numbers of type REAL. Fields which are to be regarded as integer fields do not contain a decimal point. Prepare a Fortran program (include documentation) to read and print the data. Write all the integer numbers first

under the heading THE INTEGER NUMBERS. Triple space and write the heading THE REAL NUMBERS and the real numbers under that heading. Separate all fields by two blank columns when printing. Punch a data card exactly like the one shown, and punch your program. Run the program first on the 360. When the output is correct, then run your program on the IBM 1130, referring to Appendix VII for instructions. Show the printed outputs of both computers to your instructor prior to requesting the assessment task.



UNIT #8 ACTIVITIES TABLE

All references are to <u>FORTRAN IV Programming for Engineers and Scientists</u> by Murrill and Smith.

1. Read Section 4-3, pages 63-66.
2. Read Section 4-4, page 66.
3. Read Section 4-5, pages 66-70.
4. Read Section 4-6, pages 70-72.
5. Read Sections 4-7 and 4-8, pages 73-77.

UNIT #9 ACTIVITIES TABLE

All references are to FORTRAN IV Programming for Engineers and Scientists by Murrill and Smith, unless stated otherwise.

1. Read Chapter 5, pages 80-97.
2. Work problem 5-2, page 98, and check your program by the one given on pages 264-265. Run it on the computer if you wish.
3. Work problem 5-6, page 98, and check your program by the one given on page 266. Run it on the computer if you wish.

UNIT #10 ACTIVITIES TABLE

All references are to Fortran IV Programming for Engineers and Scientists by Murrill and Smith.

1. Read Chapter 6, pages 102-118. Pay particular attention to the rules stated in Sections 6-3 and 6-4.
2. Work problem 6-1, page 118. Run your program on the computer. (One solution is shown on the next page.)
3. Work problem 6-2, page 118. In addition to printing b and f(b), print the coefficients in a clearly labeled format. Use the factored form of a polynomial for the calculation. The program must be general. Run your program on a computer and show your printer output to your instructor before you take the assessment task.

```
$JOB ******************     COMSG 2112
C  SELF EVALUATION, ACTIVITY 2.
C    CALCULATE N FACTORIAL.
C      IBM 360 WATFIV.
C
       DATA IN,LP/5,6/
       READ(IN,1)N
     1 FORMAT(I3)
       IF(N)10,11,12
C******* ERROR.  N IS NEGATIVE.
    10 WRITE(LP,2)N
     2 FORMAT(' N IS NEGATIVE.  N = ',I3)
       STOP
C******* N IS ZERO.  N FACTORIAL IS 1.
    11 NF=1
       GO TO 20
C******* N > ZERO.  CALCULATE N FACTORIAL.
    12 NF=1
       DO 13 J=1,N
         NF=NF*J
    13 CONTINUE
    20 WRITE(LP,3)N,NF
     3 FORMAT(' N = ',I3/' N FACTORIAL = ',I10)
       STOP
       END

$ENTRY
N =   8
N FACTORIAL =        40320

CORE USAGE        OBJECT CODE=     5152 BYTES,ARRAY AREA=        0 BY

COMPILE TIME=       0.73 SEC,EXECUTION TIME=       0.01 SEC,   WATFIV
```

UNIT #11 ACTIVITIES TABLE

References are to Fortran IV Programming for Engineers and Scientists by Murrill and Smith.

1. Read Chapter 7, Sections 1-10, pages 137-155.
2. Work problem 7-3, pages 160-161. Run the program on the computer. One solution to the program is given on pages 284-285.

UNIT #13 ACTIVITIES TABLE

Unless specified otherwise, all text references are to FORTRAN IV PROGRAMMING FOR ENGINEERS AND SCIENTISTS by Murrill and Smith.

Activity 1:	Read Section 2-5, pages 25-26.
Activity 2:	Read the introduction to Chapter 9 and Section 9-1, pages 188-190.
Activity 3:	Read Section 9-2, pages 190-194.
Activity 4:	Work problem 9-14, page 213. Check your results against the solution given on page 303.
Activity 5:	Read Section 9-3, pages 194-195. Read Section 9-4, pages 195-197.
Activity 6:	Read Section 9-5, pages 197-202.
Activity 7:	Work the first part of 9-23, page 214. Check your results against the solution given on page 306. (A vector is simply a single-dimensional array of numbers; multiply each element of the array by a single number, a scalar quantity.)

UNIT #14 ACTIVITIES TABLE

Read the following references in Fortran IV Programming for Engineers and Scientists by Murrill and Smith:

1. First paragraph of Section 3-5, page 49.
2. Section 3-6, pages 52-54.

UNIT #16 (COMSC) ACTIVITIES TABLE

1. Read pages 117 to 150 of the book Introduction to Computer Science by John K. Rice and John R. Rice, published by Holt, Rinehart and Winston.

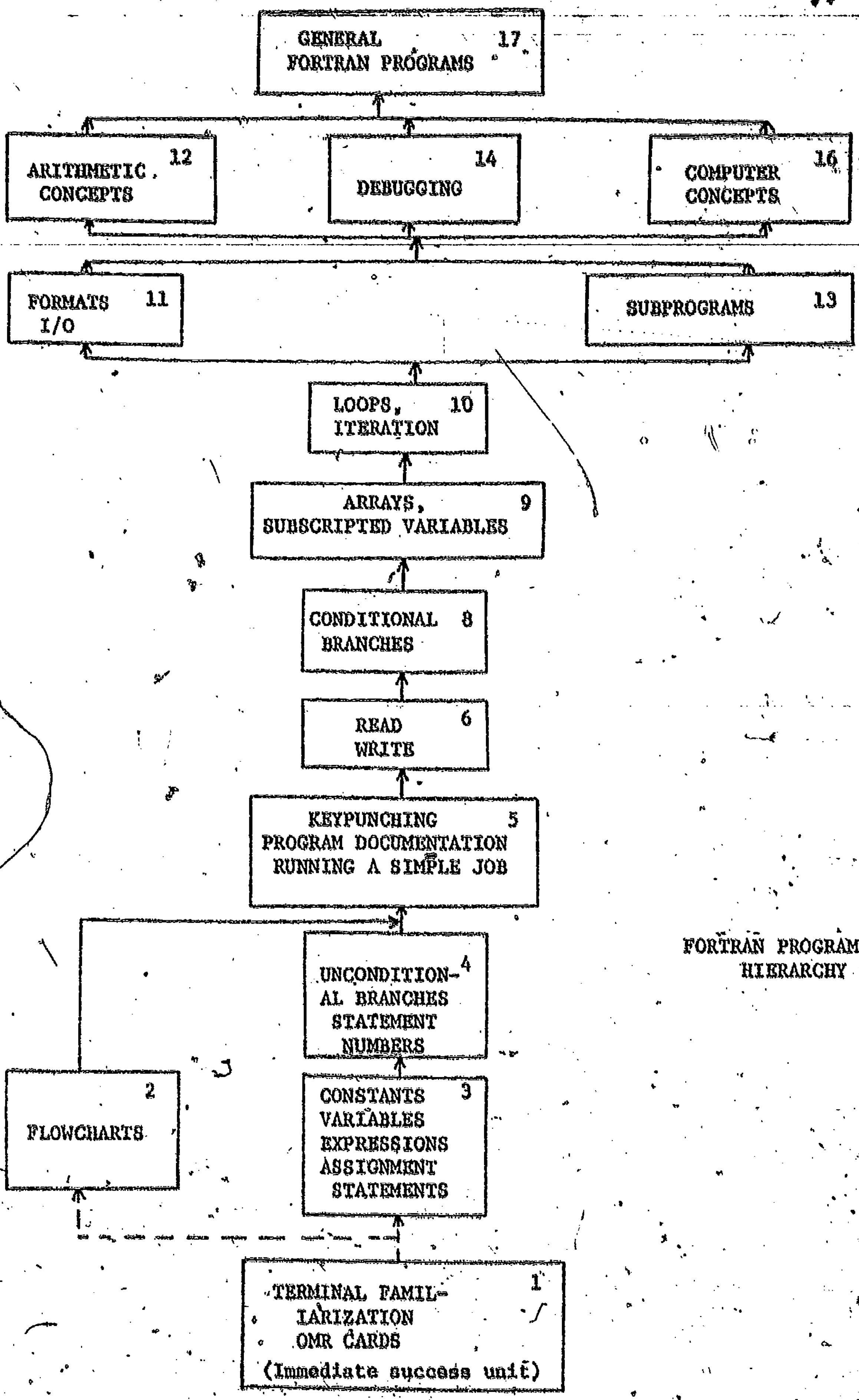
2. Reread Chapter 1, pages 1-16, of Fortran IV Programming for Engineers and Scientists.

UNIT 17 ACTIVITIES TABLE

Read Chapter 10, pages 216-225, in Fortran IV Programming for Engineers and Scientists by Murrill and Smith.

GEdwards.

GY-9310(EN)
PIPI
Okla State
McCollum


GENERAL FORTRAN PROGRAMS 17
ARITHMETIC CONCEPTS 12
DEBUGGING 14
COMPUTER CONCEPTS 16
FORMATS I/O 11
SUBPROGRAMS 13
LOOPS, ITERATION 10
ARRAYS, SUBSCRIPTED VARIABLES 9
CONDITIONAL BRANCHES 8
READ WRITE 6
KEYPUNCHING PROGRAM DOCUMENTATION RUNNING A SIMPLE JOB 5
UNCONDITION-AL BRANCHES STATEMENT NUMBERS 4
FLOWCHARTS 2
CONSTANTS VARIABLES EXPRESSIONS ASSIGNMENT STATEMENTS 3
TERMINAL FAMIL-IARIZATION OMR CARDS (Immediate success unit) 1

FORTRAN PROGRAMMING HIERARCHY