SPERRY UNIVAC

ATTN: CHARLIE GIBBS

01120
CAV208M45541 UP 7536 R1E

SPERRY UNIVAC UAS
SUITE 906
1177 WEST HASTINGS ST
VANCOUVER BC V6E 2K3

CAV

##

PUBLICATIONS UPDATE

General

Fundamentals of FORTRAN

Programmer Reference

UP-7536 Rev. 1–E

This Library Memo announces the release and availability of Updating Package E to "SPERRY UNIVAC Fundamentals of FORTRAN Programmer Reference", UP-7536 Rev. 1.

This update incorporates a minor change to the manual.

Copies of Updating Package E are now available for requisitioning. Either the updating package only or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive only the updating package, order UP-7536 Rev. 1–E. To receive the complete manual, order UP-7536 Rev. 1.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ and MZ	Mailing Lists A00,B00,A14,A15,B15,10,11,18,19,20,21, 28U,39,40,41,42,43,44,51,51D, 52,53,53D, 54,54D, 55, 55D,56,57,58,60,61,65,66,75,76,77 and 78 (Package E to UP-7536 Rev. 1, 5 pages plus Memo)	Library Memo for UP-7536 Rev. 1–E
		RELEASE DATE: August, 1982

SPERRY UNIVAC
COMPUTER SYSTEMS

PUBLICATIONS UPDATE

General

Fundamentals of FORTRAN

Programmer Reference

UP-7536 Rev. 1–C

This Library Memo announces the release and availability of Updating Package C to "SPERRY UNIVAC Fundamentals of FORTRAN Programmer Reference", UP-7536 Rev. 1.

This update incorporates minor corrections to the manual.

Copies of Updating Package C are now available for requisitioning. Either the updating package only, or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive the updating package only, order UP-7536 Rev. 1–C. To receive the complete manual, order UP-7536 Rev. 1.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ,CZ and MZ	Mailing Lists 10,11,18,19,20,21,28U,29U,39,40,41,42 43,44,51,51D,52,53,53D,54,54D,55,55D,56,57,58,60, 61,65,66,75,76,77 and 78 (Package C to UP-7536 Rev. 1, 5 pages plus Memo)	Library Memo for UP-7536 Rev.1–C
		RELEASE DATE: June, 1981

UNIVAC
Fundamentals of Fortran

Programmer Reference

SPERRY UNIVAC COMPUTER SYSTEMS

UP-7536 Rev. 1

This document contains the latest information available at the time of publication. However, Sperry Univac reserves the right to modify or revise its contents. To ensure that you have the most recent information, contact your local Sperry Univac representative.

UNIVAC is a registered trademark of the Sperry Rand Corporation.

Other trademarks of the Sperry Rand Corporation include:

FASTRAND

MATED-FILM

PAGEWRITER

UNISCOPE

UNISERVO

PRINTED IN U.S.A.

PAGE STATUS SUMMARY

ISSUE: Update E —UP-7536 Rev. 1

Part/Section	Page Number	Update Level
Cover/Disclaimer		A
PSS	1	E
Contents	1 thru 5	Orig.
1	1 thru 13	Orig.
2	1 thru 23 24 25 thru 27 28, 29	Orig. C Orig. D
3	1 thru 4 5 6 thru 14	Orig. E Orig.
4	1 thru 4	Orig.
5	1 thru 12 13 14, 15 16 17 thru 24	Orig. D Orig. A Orig.
6	1 thru 10 11 12 13 14 thru 22 23 24 25 26, 27 28 thru 32	Orig. D Orig. D Orig. A B Orig. D Orig.
7	1 thru 3 4 5 6 7	Orig. D Orig. D Orig.
8	1 thru 8 9 10 thru 22 23 24 thru 26 27 28 thru 32 33 34 thru 37	Orig. D Orig. A Orig. D Orig. A Orig.
9	1 thru 3	Orig.
Appendix A	1, 2	Orig.
Index	1 thru 7	Orig.
User Comment Sheet		

All the technical changes are denoted by an arrow (→) in the margin. A downward pointing arrow (↓) next to a line indicates that technical changes begin at this line and continue until an upward pointing arrow (↑) is found. A horizontal arrow (→) pointing to a line indicates a technical change in only that line. A horizontal arrow located between two consecutive lines indicates technical changes in both lines or deletions.

CONTENTS

TABLES

1. INTRODUCTION

1.1. GENERAL

FORTRAN (from FORmula TRANslator) is a programming language designed for extensive use in mathematical, scientific, and technological areas. The advantages of FORTRAN are minimum programming time and cost, and maximum interchangeability of FORTRAN programs on different FORTRAN processors.

FORTRAN statements resemble English statements and the equations of elementary algebra. Therefore, FORTRAN statements are self-documenting, that is, the intended operation is apparent from the statement itself. For example, to find the average of two numbers, the programmer can write a statement such as:

```
AVRGE = (A+B)/2.0
```

Since the FORTRAN programmer uses a programming language that resembles the language ordinarily used for the solution of problems, relatively little time is required to learn the language. As a result, programming effort can be devoted to the logic of the problem without being troubled by the intricacies of computer operation. This self-documenting feature of FORTRAN reduces debugging time and enables other programmers to readily grasp the logic of a program so that it can be modified or adapted to other purposes with minimal effort.

Because FORTRAN is the first programming language to be generally accepted as a standard by the data processing community, a FORTRAN program written for a particular FORTRAN processor can be accepted by many different FORTRAN processors with a minimum of change. The FORTRAN specifications described in this manual are those of the American National Standards Institute, Inc. (ANSI), formerly known as USA Standards Institute, Inc. (USASI), in FORTRAN, X3.9–1966. Another ANSI standard in current use is Basic FORTRAN, X3.10–1966, which is a subset of FORTRAN. Differences between the two standards are described in Appendix A.

Fundamentals of FORTRAN is designed to introduce FORTRAN to the novice programmer while providing sufficient depth of coverage for experienced FORTRAN programmers. The examples and descriptions of operation, therefore, will be particularly valuable for the novice.

1.2. PROGRAMMING LANGUAGES

The series of steps specified for the solution of a particular problem is called a *source program* and the notation that the programmer uses for specifying these steps is a *programming language*. Except in very few cases (where the programmer uses machine language) the computer cannot "understand" the programming language; the programming language must be translated into instructions that the computer can comprehend. Any programming language that resembles English or the language of mathematics must be translated into machine language before it can have any effect upon computer operation. The combination of the mechanism that accomplishes the translation from programming language to machine language and the data processing system is called the *processor*.

The following paragraphs trace the development of programming languages from machine language to assembly language to FORTRAN and show the application of each to the problem of evaluating Z in the following equation:

$$Z = \frac{(R + S - T)X}{Y}$$

where the values of R, S, T, X, and Y are known.

1.2.1. Machine Language

The fundamental unit of information handled by a data processing system is the *bit* (from binary digit). In the strictest sense, *machine language* is a combination of bits interpreted by a data processing system as an instruction. Each bit has two mutually exclusive states, represented by 0 and 1. However, almost every data processing system has built-in facilities for accepting an abbreviated form of machine language. In an "octal" computer, each of the octal digits, 0 through 7, represents a combination of three bits. For example, an octal 6 is the equivalent of binary 110. In a "hexadecimal" computer, the digits 0 through 9 and A, B, C, D, E, and F each represent four bits, e.g., the hexadecimal D represents the binary 1101. In a "decimal" computer, each of the digits, 0 through 9, represents four bits. For example, the decimal 91 may be represented internally as 1001 0001. The following descriptions of machine language refer to this abbreviated form, using a typical octal computer.

Each computer has its own unique set of instructions and machine language. Using machine language, the programmer must know the operation code (indicating the operation to be performed), the location (in main storage) of at least one operand, and the location which is to contain the result. Typical machine language coding for the problem described in 1.2 is indicated after the following assumptions:

The value of R is in location 10002.

The value of S is in location 10010.

The value of T is in location 10020.

The value of X is in location 10030.

The value of Y is in location 10040.

The result, Z, is to be stored in location 10050.

A typical machine language set of coded instructions for performing the evaluation is:

772110002 moves the value of R to an arithmetic *register*. (A register is a fixed location which stores an operand or the result of an arithmetic operation.)

770110010 adds the value of S to the value in the register and retains the result in the register, giving R + S.

770210020 subtracts the value of T from the value in the register and retains the result in the register, giving R + S – T.

770310030 multiplies the value in the register by the value of X and retains the result in the register, giving (R + S – T) X.

770510040 divides the value in the register by the value of Y and retains the result in the register, giving [(R + S – T)X] / Y.

772510050 moves the contents of the register to the location specified for Z.

The first four digits of each instruction in this example constitute the operation code; the rightmost five digits specify a memory location.

Machine coding is tedious; it requires time for learning (since it is unique), the programmer must keep track of all memory references, and it requires many lines of source coding for a relatively simple operation. However, machine coding requires the least machine time since no language conversion is involved.

1.2.2. Assembly Language

The next logical step from machine language is what is commonly known as assembly language. Assembly language requires a language translation program (an assembler). Using assembly language, the programmer is permitted to use symbolic references to memory and specified mnemonic codes instead of numeric codes to designate the operation to be performed. Typical assembly coding for the problem in 1.2 is:

ENT*R enters R into the register.

ADD*S adds S to R in the register.

SUB*T subtracts T from the contents of the A register and retains the result in the register.

MUL*X multiplies the contents of the register by X and retains the result in the register.

DIV*Y divides the contents of the register by Y and retains the result in the register.

STR*Z stores the contents of the register in Z.

Assembly language coding is less tedious than machine coding, and the operation code is more meaningful, making the language easier to learn. The assembler assigns memory locations to the symbolic memory references, relieving the programmer of that chore, thus making the coding even more meaningful. However, it is essentially a one-to-one source coding process (i.e., one source language instruction generates one machine language instruction) when compared to machine coding and still requires many lines of source coding for a simple problem.

1.2.3. FORTRAN Language

FORTRAN is one of many higher level languages than have evolved from assembly language. It is considered a higher level language because the translation of a FORTRAN statement may result in many machine language instructions. This conversion is performed by a program called the FORTRAN compiler. The design of a FORTRAN compiler is definitely machine-oriented and is not part of the FORTRAN language.

Offsetting this relative complexity (and consequent increased overall computer time), FORTRAN is a self-documenting language that cuts down the cost and time required for learning, writing, debugging, and maintenance. In standard FORTRAN there are approximately 30 statements to be remembered by the programmer; whereas, in assembly language there may be from 50 to 350 mnemonics. The compactness and self-documentation of FORTRAN are apparent in Figure 1–1. For instance, a program that requires from 10 to 20 pages of assembly coding may require only one page of FORTRAN coding.

FORTRAN	ASSEMBLER	MACHINE
Z = ((R+S–T)*X)/Y	ENT*R	772110002
	ADD*S	770110010
	SUB*T	770210020
	MUL*X	770310030
	DIV*Y	770510040
	STR*Z	772510050

Figure 1–1. FORTRAN–Assembler–Machine Coding

1.3. SOURCE AND OBJECT PROGRAMS

A FORTRAN program written by the programmer represents a series of logical steps for the solution of a particular problem. This program is the *source program.*

A source program must be translated to machine language for a particular data processing system. Translating the FORTRAN source program to machine language is generally accomplished by a prewritten program, the FORTRAN *compiler.* (The compiler is furnished with the data processing system and is not the responsibility of the programmer.) The output of this translation process (compilation) is the *object program.*

1.4. COMPILATION AND EXECUTION

The complete compilation process follows the steps below:

(1) The source program is keypunched onto cards to produce the source program deck.

(2) The FORTRAN compiler is read from auxiliary storage into the computer's memory (main storage) as shown in Figure 1–2.

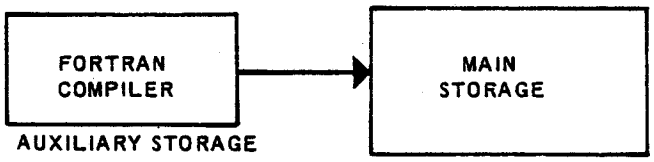


Figure 1–2. Compiler to Memory

(3) The source deck is then read into main storage and control is turned over to the compiler which determines the operations to be performed and generates the required machine language instructions to create the object program. This translation process includes the assignment of memory locations for variables and constants, and the utilization of routines stored in auxiliary storage when required. The compiler produces an object program deck of machine instructions and a listing of the source program as shown in Figure 1–3.

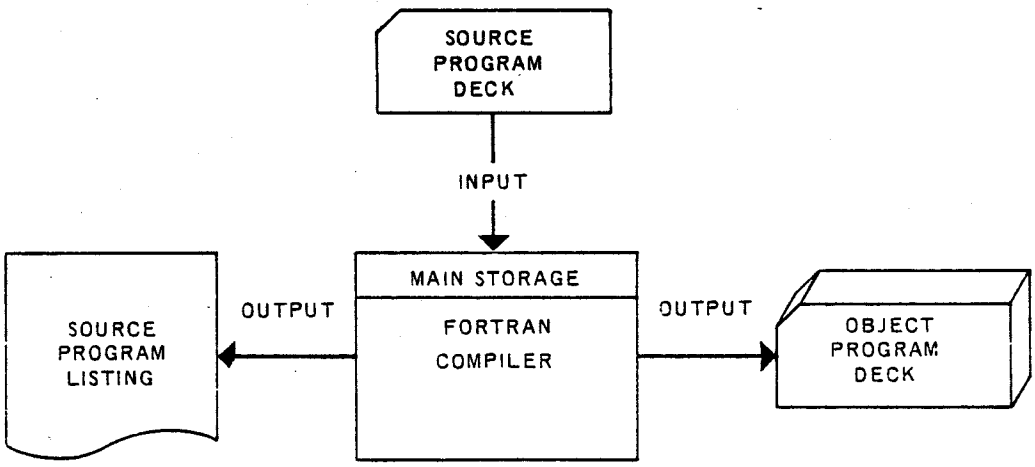


Figure 1–3. The Compilation Process

(4) The object program and internal data are entered into main storage. The data processing system can then execute the program: fetching data from input and auxiliary storage devices as required, operating on data as directed by the object program, and producing required output.

Steps 1 through 4 can be performed in a fairly rapid sequence referred to as "the compile and execute process" shown in Figure 1–4. The object program is placed in main storage as it is created by the compiler (rather than being transmitted in the form of punched cards as described in step 3) and immediately executed.

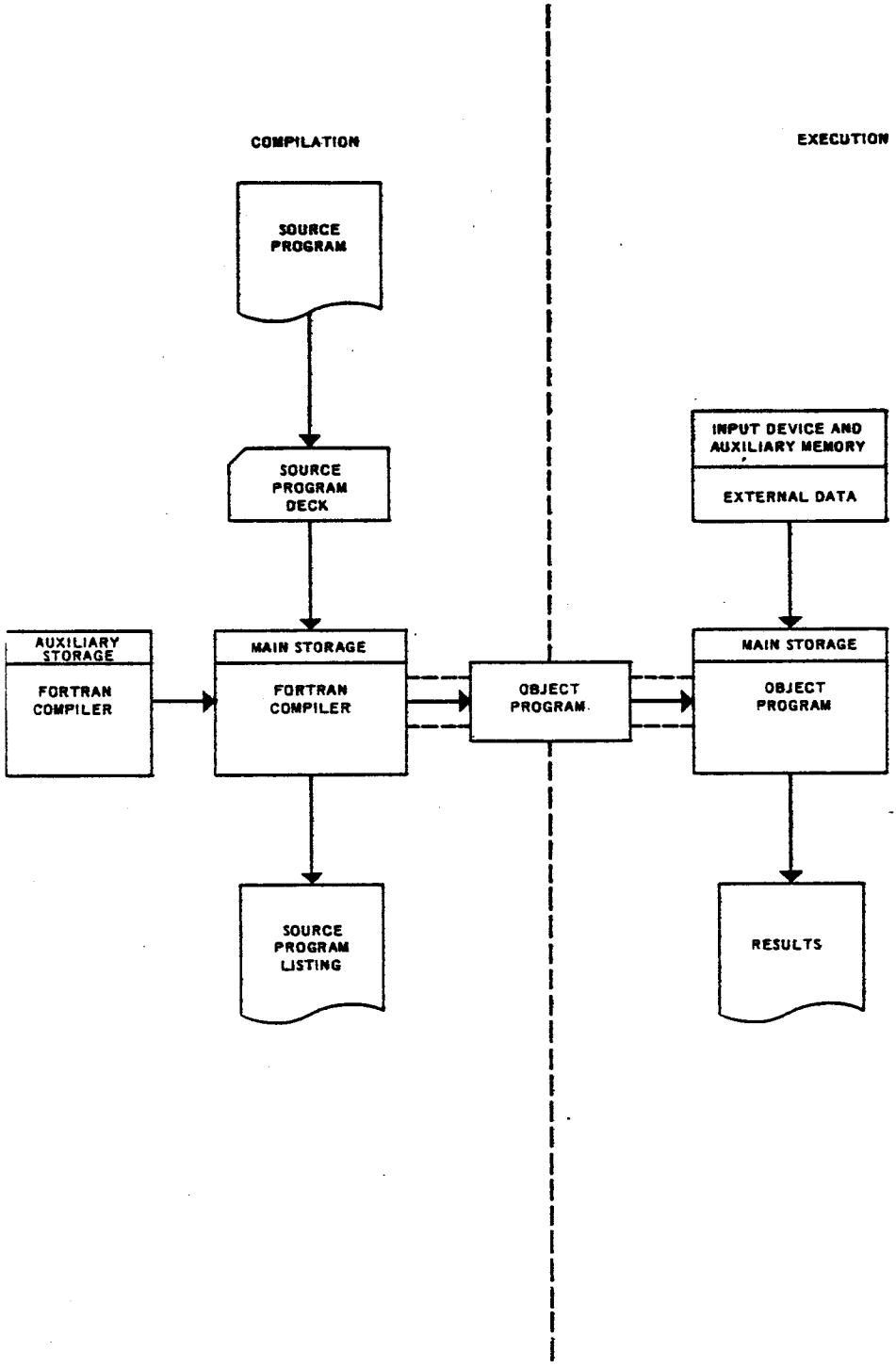


Figure 1–4. The Compile and Execute Process

1.5. THE COMPUTER SYSTEM

A computer system is made up of hardware components (equipment) and software (operating system). The operating system is a program usually furnished by the computer manufacturer made up of routines that coordinate hardware activity and furnish various services to the user.

1.5.1. Computer Hardware

A digital computer is a data processing system that processes data in accordance with a set of instructions (program) and produces useful results. The programmer may regard the computer as a complex of devices with the functions shown in Figure 1–5.

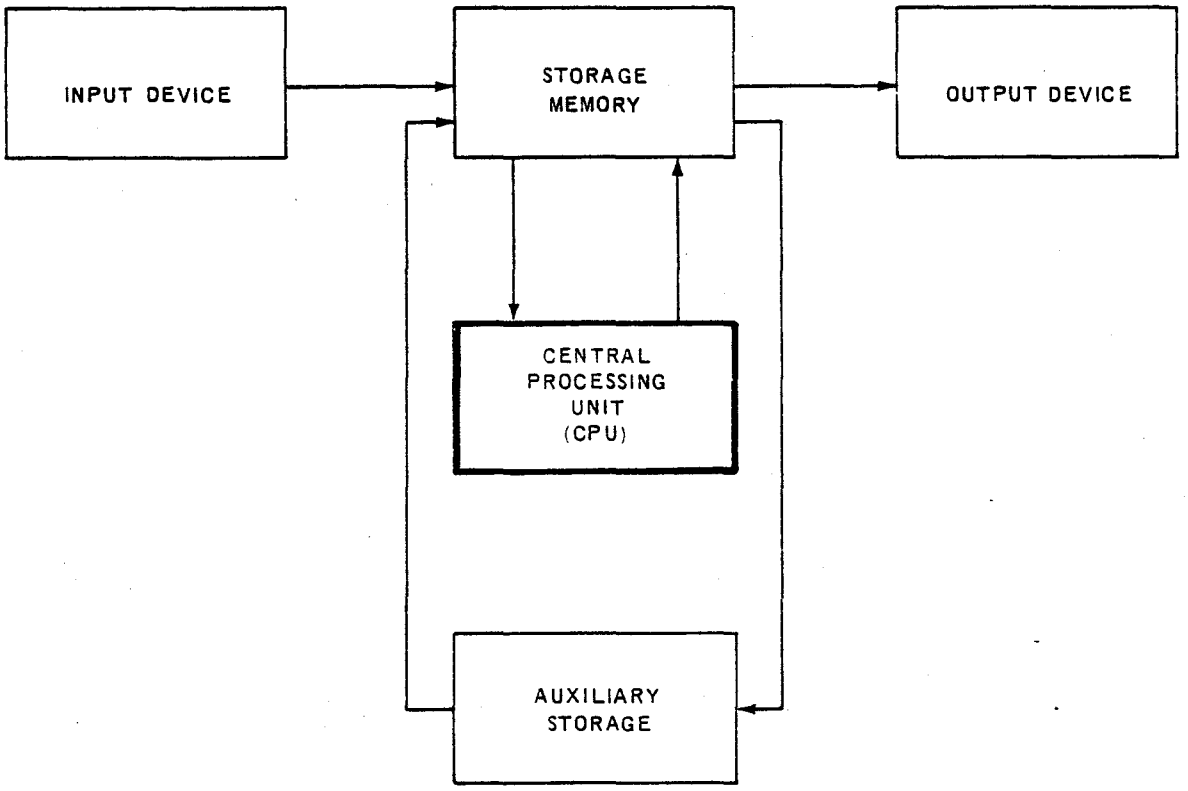


Figure 1–5. Elements of the Computer

1.5.1.1. Input Devices

An input device furnishes the program (and possibly data) to main storage. The central processing unit (CPU) fetches the program and data from main storage, processes the data, and stores the result in main storage from where it is sent to an output device. If the program requires, input may be obtained from auxiliary storage devices and the result stored in auxiliary storage for updating or for transmission to an output device at a later time. Similarly, the input program can be held in auxiliary storage to be processed on demand by the CPU.

Input devices are varied in nature. Currently, the most widely used device is the 80-column punched card reader and all descriptions presented here are oriented toward such a device. Other commonly used input devices are the console typewriter and punched paper tape readers.

1.5.1.2. Main Storage

Main storage retains the program, intermediate results, and output in addressable memory storage units. The contents of these storage units are transferred to the CPU for processing, and the storage units receive results from the CPU. Since processing in the CPU is generally much faster than the rate at which data is obtained from, and entered into, main storage, the processing speed of a computer is determined by the time required to enter and retrieve information into and from main storage.

1.5.1.3. Central Processing Unit

The central processing unit (CPU) decodes the machine language instruction, performs the arithmetic and logical processing functions indicated, and supplies the timing and control signals which synchronize the actions of the other elements of the computer. Each machine language instruction must indicate the operation to be performed and the location of the operands.

Internal representation of values depends upon the particular computer; however, number representation is divided into two classes: fixed-point representation and floating-point representation. All computers can represent fixed-point numbers; however, some may use preprogrammed routines for the representation of floating-point numbers.

1.5.1.3.1. Fixed-Point Representation

In fixed-point number representation, any external string of digits (optionally signed) is represented internally as a signed string of digits. A decimal point (radix point) is implied to be a *fixed* number of positions (this fixed number may be zero) from the rightmost position, hence the term "fixed point." When arithmetic operations are performed on fixed-point operands, it is the programmer's responsibility to keep track of the decimal point to ensure that both the operands and the result are correctly aligned. This is usually done by *scaling* the operands (multiplying by a power of 2 or 10). However, when many numbers of widely differing magnitudes are involved, floating-point numbers are used because fixed-point representation requires an extensive programming effort for the scaling operations. The advantage of fixed-point representation is that *any* external string of digits can be represented *exactly* provided it is within the limits of range prescribed by the computer. This range provides limited representation when compared to floating-point representation.

1.5.1.3.2. Floating-Point Representation

In floating-point representation, any number can be represented as a fixed-point number (mantissa) multiplied by an integral power (exponent) of 2 or 10. The exponent part is also a fixed-point number.

The problems that arise from the use of floating-point representation of values are truncations and roundoffs that result from arithmetic operations. For example, the division operation 1.0/3.0 has as its true result, 0.333...3. However, since the mantissa can only contain a limited number of significant digits, only an approximation to the true result can be stored. Because of this, a floating-point value actually represents an approximation to the true value of a number. These errors are propagated as successive operations are performed upon floating-point operands and their results. With some computers another approximation is required if the internal representation of the mantissa is in binary form, rather than in binary coded decimal, because a decimal fraction cannot always be expressed exactly in binary form. For these reasons, comparisons involving floating-point numbers require special attention from the programmer.

1.5.1.4. Auxiliary Storage

Auxiliary storage is storage in addition to main storage and includes devices such as tape (magnetic and paper), disc, drum, and card (magnetic and paper). The time required to access data contained in auxiliary storage is significantly longer than for main storage; however, the cost of auxiliary storage is lower and capacity is generally larger. Auxiliary storage devices may contain files of data required for a program, different modules of the operating system, and/or results of a program that will later be transcribed on an output device.

1.5.1.5. Output Devices

An output device usually furnishes a visual display of the results, as specified in the program. In addition, an output device can inform the operator of various conditions arising during operation. The most widely used devices are the console typewriter, printer, and CRT (cathode ray tube) display.

The auxiliary storage devices described in the previous paragraph can also be considered as output devices, depending on their end use.

1.5.2. Computer Software

The entire complex of software (programs furnished with the computer) is called the operating system. The operating system may contain programs which control scheduling, input/output, compilation, debugging, storage assignment, linking, loading, assembly, and other necessary functions depending upon the individual computer system.

1.6. SAMPLE PROGRAM

The following paragraphs present a simple *executable* program (a self-contained computing procedure) for the purpose of introducing FORTRAN programming and terminology. The concepts and terminology used in the description of the program are described in detail in other sections of the manual.

This program calculates the average of a series of numbers, each of which is supplied by a punched card. The program is general enough to calculate the average no matter how many values are involved. The last card of the data deck contains a value known to be outside the range of values expected. This card is used as an end data card. After the end data card is detected, the average is calculated and printed, together with explanatory text.

Figure 1-6 is a flowchart, with explanatory text, outlining the program. A flowchart should be constructed for any extensive program. It is a convenient means for detecting logical errors and provides documentation for other programmers who may be able to use or modify the same program. If the flowchart provides enough detail, the actual writing of the program is greatly simplified.

1. Initialize a running count and a running total to zero. The running count will indicate the number of values (one value per data card) to be averaged. Each time a value is obtained, it will be added to the running total.
2. Read in a value from a card.
3. Test the value for end data indicator. If the value is greater than $9.0(10^8)$ it indicates that the previous value read in was the last value to be used in computing the average, and the next step is step 7. If the value is less than or equal to $9.0(10^8)$, proceed to step 4.
4. Add the input value to the running total.
5. Increase the running count by 1.
6. Read in the next value and repeat the processing steps by returning to step 2.
7. Divide the running total by the running count to compute the average.
8. Print this average with some explanatory text.
9. Indicate that there are no more instructions to be executed.

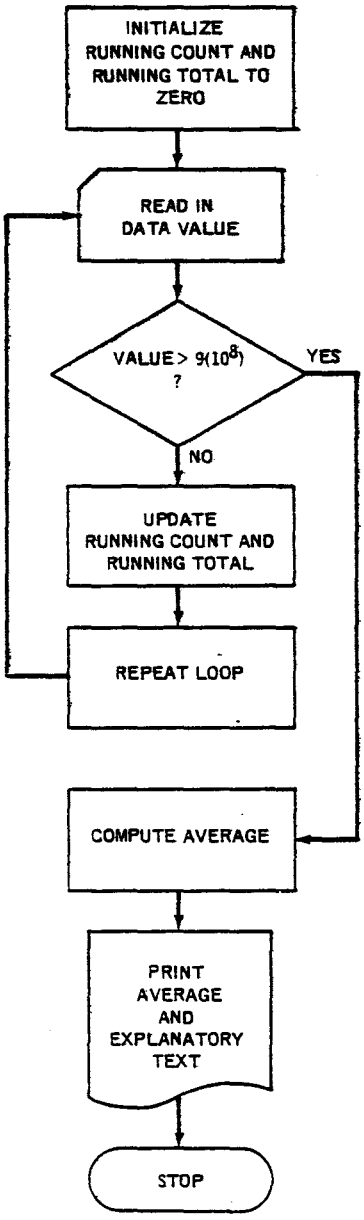


Figure 1-6. Sample Problem Flowchart

Figure 1–7 is the FORTRAN program written from the procedure outline in Figure 1–6. It is not the only program that could have been written for the problem, nor is it the shortest in terms of lines required. It does introduce FORTRAN nomenclature and basic concepts – the most important of which is the programmed *loop.* Such a loop is shown in the flowchart portion of Figure 1–6. The purpose of a loop is to repeat a series of operations without respecifying the steps for each repetition. In conjunction with every loop, there must be some test that will make it possible to leave the loop; otherwise, it will be performed without end, being limited only by the time allotted to the program. In the sample program, detection of the end data card provides the exit from the loop.

"C" FOR COMMENT | STATEMENT NUMBER 5 | 6 | FORTRAN STATEMENT → 7 10 20 30 40 50

```
C COMPUTE AVERAGE OF NUMBERS TO BE READ FROM DATA CARDS
C
C INITIALIZE RUNNING COUNT AND RUNNING TOTAL
      DATA KOUNT, TOTAL/0,0.0/
C OBTAIN VALUE
   30 READ(1,10) VALUE
   10 FORMAT (G10.0)
C TEST FOR END DATA
      IF(VALUE .GT. 9.0E8) GO TO 20
C UPDATE RUNNING TOTAL AND RUNNING COUNT
      TOTAL = TOTAL + VALUE
      KOUNT = KOUNT + 1
C REPEAT LOOP
      GO TO 30
C COMPUTE AVERAGE AFTER LAST VALUE
   20 XKOUNT = KOUNT
      XMEAN = TOTAL / XKOUNT
C PRINT AVERAGE AND TEXT
      WRITE(3,40) XMEAN
   40 FORMAT(17H AVERAGE VALUE = ,F10.5)
C INDICATE END OF EXECUTION
      STOP
C INDICATE NO MORE LINES
      END
```

Figure 1–7. Sample Program

Line 1 of the program is a *comment line*, indicated by the character C in column 1. This line (including the C) is printed when the program is compiled, but does not affect execution of the program. Comment lines provide documentation for the programmer.

Line 2 is also a comment line with all blank characters producing a blank line (except that the C is printed).

Each of the remaining comment lines applies to the statement(s) immediately following it. Again, comment lines are not required, but are included to aid the reader in following the program.

Line 4 is a DATA or *initialization statement* that sets an initial value of zero for KOUNT (symbolic name for the running count) and TOTAL (symbolic name for the running total). The DATA statement does this at compilation time rather than execution time, in order to reduce execution time for the program. KOUNT will be a fixed-point value; TOTAL, a floating-point value.

Line 6 is a READ statement that instructs the computer to read a punched card for the number to be represented by VALUE.

Line 7 is a FORMAT statement that indicates the required number is found in the first ten character positions of the punched card.

Line 9 is a logical IF statement. It tests the number that is read in for VALUE; if the number is greater than 9.0 (10^8), program control is transferred to the statement with a *statement label* of 20 (in columns 1 through 5). If the number is less than or equal to 9.0 (10^8), the next executable statement which follows in physical order is to be executed. The logical IF provides the exit from the loop.

Line 11 is an *arithmetic assignment statement* that updates the running total. It obtains the current value of TOTAL, adds to it the number just read for VALUE, and assigns this sum as the new value for TOTAL.

Line 12 is also an arithmetic assignment statement that increases the value of KOUNT by 1 each time a number for VALUE is read.

Line 14 is a GO TO statement that completes the loop. When executed, it transfers control to the statement with statement label 30, so that the next card can be read.

Line 16 is executed only after the end data card, which contains a value greater than 9.0 (10^8), has been read. Its effect is to convert KOUNT to floating-point form to be compatible with TOTAL so that one can be divided by another (line 17).

Line 17 is an arithmetic assignment statement that specifies the division necessary to calculate the average, XMEAN.

Line 19 is a WRITE statement which specifies that the value of XMEAN is to be printed.

Line 20 contains a FORMAT statement which specifies that the characters AVERAGE VALUE = are to be printed, followed by 10 print positions for the value of XMEAN, which is to be printed with five digits to the right of the decimal point.

Line 22 is a *program control statement*, the STOP statement, that terminates execution of the program.

Line 24 is an *end line*, terminating compilation of the program unit.

Note that every statement (columns 7 through 72) except the assignment statements starts with a *keyword*. (Comment lines are not considered statements.) The keyword is an English word that describes the purpose of the statement. Every statement in FORTRAN, except statement functions and certain assignment statements, begins with a keyword. Keywords are not reserved words in FORTRAN; they may be used anywhere in the program as symbolic names.

Figure 1–8 is the complete deck of punched cards for the program in Figure 1–7. This deck consists of the FORTRAN program cards followed by data cards and an end data card. It should be clearly understood that punched cards are not the only means of primary input. Punched cards are shown here because of their widespread use and because they are easily used for illustrative purposes in examples. At least one system control card is required preceding the program, another preceding the data, and possibly an end of file card after the last data card. Requirements of these control cards depend upon the computer being used, and they are not part of the standard FORTRAN language.

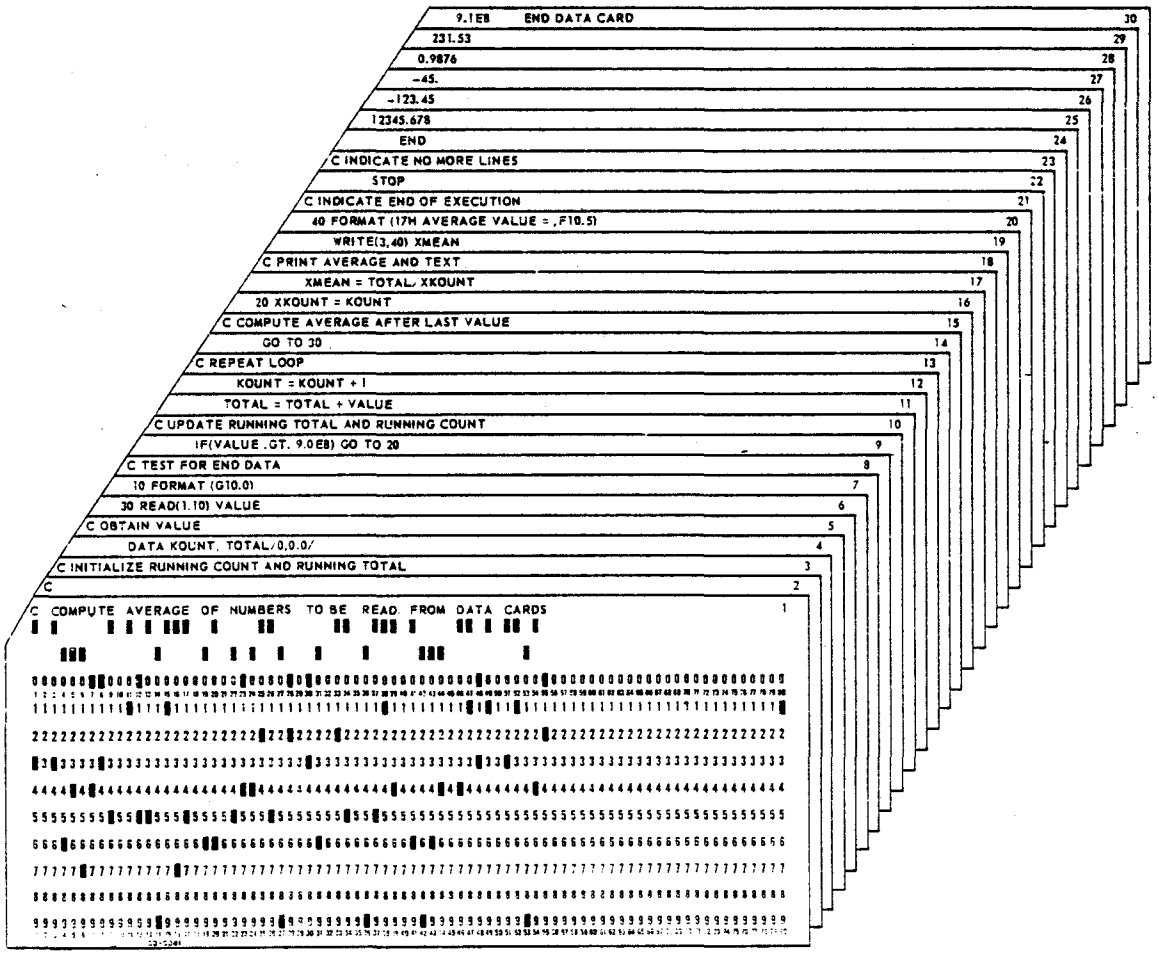


Figure 1–8. Sample FORTRAN Problem Deck and Data

2. WRITING A FORTRAN PROGRAM

2.1. GENERAL

This section discusses the organization of FORTRAN programs and the rules for writing a source program, with particular attention to constants, variables, and arrays.

2.2. ORGANIZATION

Organization of a FORTRAN program necessitates discussion in two areas: the concept of subprograms and the organization of program units.

2.2.1. FORTRAN Program

A FORTRAN program is made up of one, and only one, *main program* and as many subprograms as required. The main program contains the steps required to solve a given problem, the subprograms are subordinate program units used by the main program. Both are referred to as *program units*.

The various types of program units (see Figure 2–1) are as follows:

- A *main program* is a series of comments and statements which does not contain a FUNCTION, SUBROUTINE, or BLOCK DATA statement and is terminated by an end line.
- A *function subprogram* is a series of comments and statements starting with a FUNCTION statement and terminated by an end line.
- A *subroutine subprogram* is a series of comments and statements starting with a SUBROUTINE statement and terminated by an end line.
- A *specification subprogram* is a series of comments and specification statements starting with a BLOCK DATA statement and terminated by an end line.

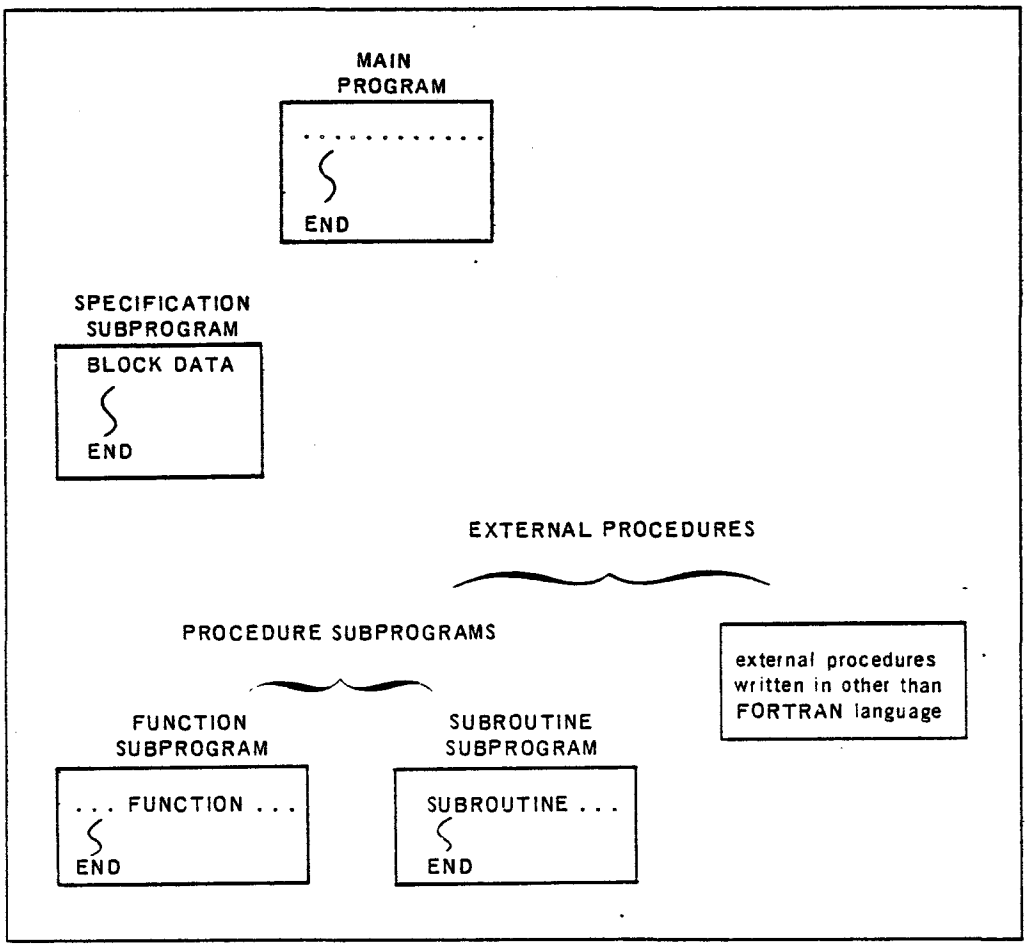


Figure 2–1. Program Units of FORTRAN Program

Each program unit is independently compiled and then linked together by the operating system to form an executable program starting with the main program unit. Program units may be written in languages other than FORTRAN but must conform to the rules for FORTRAN subprograms. Such program units and *procedure subprograms* (function and subroutine subprograms) are termed *external procedures.*

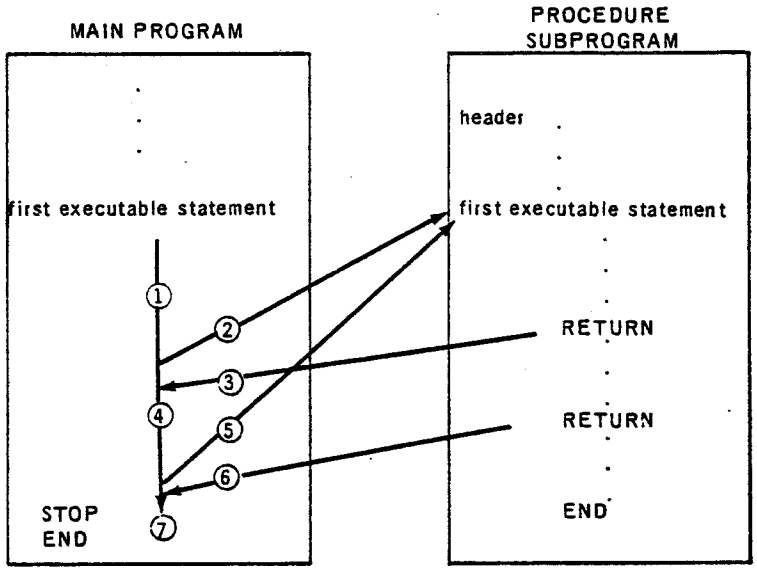
Execution of a program always starts with the first executable statement of the main program. In the first example shown in Figure 2–2, the main program proceeds until it encounters a reference (call) to the external procedure. The external procedure assumes control until it encounters a RETURN statement, which sends control back to the calling program unit (in this case, the main program). The main program then continues processing until another reference transfers control to the external procedure. The external procedure assumes processing until it encounters a RETURN statement (not necessarily the same as the first RETURN statement) and transfers control back to the main program. The main program then resumes processing until it encounters the STOP statement, which transfers job control to the operating system.

The second example in Figure 2–2 shows how a procedure subprogram can call upon another procedure subprogram during execution.

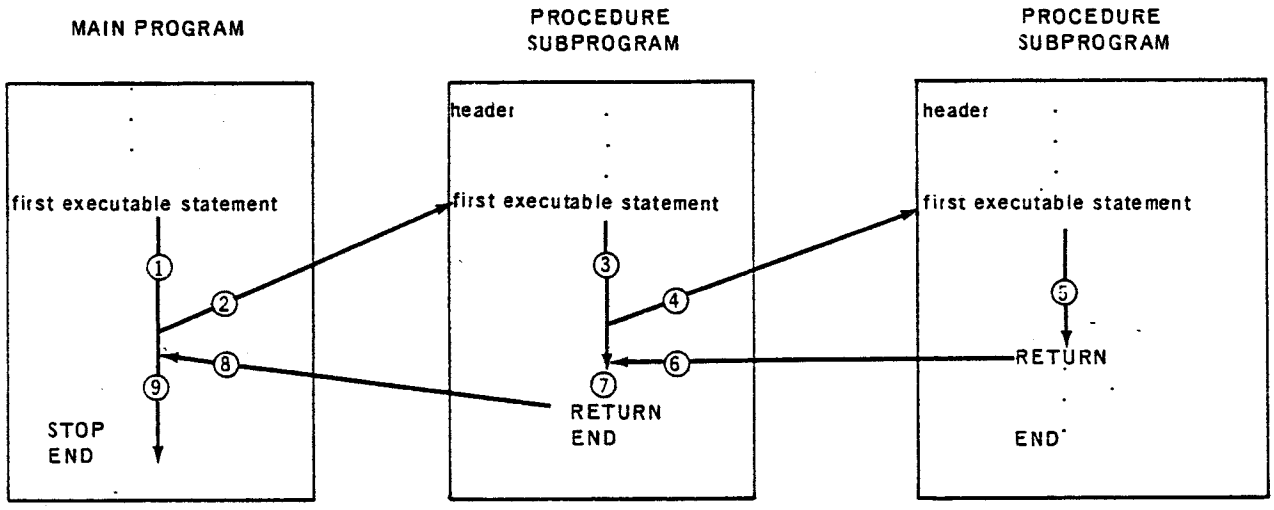
A specification subprogram (with a BLOCK DATA header) consists entirely of nonexecutable statements and therefore never assumes control during execution.

Subprograms are useful because they eliminate repetitive coding of procedures used many times in a program. In addition, a library of mathematical external procedures, called basic external functions, is present which contains debugged procedures for computation of mathematical functions such as square root, sine, etc. The main program of a large FORTRAN program can be coded as a logical skeleton consisting primarily of references to subprograms; these subprograms can be independently coded and compiled concurrently with the main program.

EXAMPLE 1



EXAMPLE 2



NOTE: A header is either a FUNCTION or a SUBROUTINE statement.

Figure 2–2. Control Path During Execution

2.2.2. Program Unit Organization

A program unit consists of comments, statements, and one end line. A FORTRAN statement falls into one of two categories: an *executable statement* or *nonexecutable statement*. An executable statement specifies an action; a nonexecutable statement describes the characteristics and arrangement of data, editing information, statement function definitions, and classification of program units. Nonexecutable statements are generally intended as instructions to the compiler; no executable machine language instructions are generated. Executable statements result in executable machine language instructions, effective at execution time. As an example, lines 6 and 7 of Figure 1–7 are as follows:

```
      30  READ(1,10)VALUE
      10  FORMAT(G10.0)
```

The READ statement is an executable statement specifying an action; the FORMAT statement is a nonexecutable statement specifying the characteristics and arrangement of data. Table 2–1 is a precise guide for determining whether a specific FORTRAN statement is executable or nonexecutable. The order (sequence) of statements within each program unit is shown in Table 2–2.

STATEMENT	GENERAL CATEGORY
EXECUTABLE STATEMENTS	
arithmetic assignment statement logical assignment statement GO TO assignment statement	assignment statements
GO TO statements IF statements CALL statements CONTINUE statement RETURN statement STOP statement PAUSE statement DO statement	control statements
READ statement WRITE statement REWIND statement BACKSPACE statement ENDFILE statement	I/O statements

Table 2–1. FORTRAN Statements (Part 1 of 2)

STATEMENT	GENERAL CATEGORY
NONEXECUTABLE STATEMENTS	
DIMENSION statement COMMON statement EQUIVALENCE statement EXTERNAL statement type-statements: INTEGER statement REAL statement DOUBLE PRECISION statement COMPLEX statement LOGICAL statement	specification statements
DATA statement	data initialization statement
FORMAT statement	format statement
statement function definition	function defining statement
FUNCTION statement SUBROUTINE statement BLOCK DATA statement	subprogram statements

NOTE: The end line is nonexecutable and is not considered a statement.

Table 2–1. FORTRAN Statements (Part 2 of 2)

PROGRAM UNIT	ORDER OF STATEMENTS
MAIN PROGRAM	(1) specification statements and FORMAT statements, in any combination (2) statement function definitions, DATA statements, and FORMAT statements, in any combination (3) executable statements, FORMAT statements, and DATA statements, in any combination (4) end line Minimum requirements: an executable statement and one end line.
PROCEDURE SUBPROGRAM	(1) FUNCTION or SUBROUTINE statement (2) specification statements and FORMAT statements, in any combination (3) statement function definitions, DATA statements, and FORMAT statements, in any combination (4) executable statements, FORMAT statements, and DATA statements, in any combination (5) end line Minimum requirements: for function subprogram – one FUNCTION statement, an executable statement, a RETURN statement, and an end line; for subroutine subprograms – one SUBROUTINE statement, a RETURN statement, and an end line.
SPECIFICATION SUBPROGRAM	(1) BLOCK DATA statement (2) specification statements (except EXTERNAL), in any combination (3) DATA statements (4) end line Minimum requirements: one BLOCK DATA statement, a COMMON statement, a DATA statement, and one end line.

Table 2–2. Ordering of FORTRAN Statements

2.3. CHARACTER SET

The FORTRAN character set (see Table 2–3) consists of all the characters required for writing FORTRAN statements.

ALPHANUMERIC { LETTERS	A through Z (only upper case letters)
ALPHANUMERIC { DIGITS	0 through 9
SPECIAL CHARACTERS	blank, represented by b in text and coding form
	= equals
	+ plus
	– minus
	* asterisk
	/ slash
	(left parenthesis
	) right parenthesis
	, comma
	. decimal point
	$ currency symbol

Table 2–3. FORTRAN Character Set

In addition to this set, each processor has its own set of additional characters. The FORTRAN character set and any additional characters in the processor set are referred to collectively as the *processor character set.*

2.4. FORTRAN PROGRAMMING FORM

A typical FORTRAN programming form (for 80-column punched card input) is shown in Figure 2–3. A FORTRAN line uses only columns 1 through 72; the information in columns 73 through 80, shown only in the program listing, may be (depending upon the processor) used for documentation. (In Figure 1–7 these columns are used to serially identify the lines of the program.) Columns 1 through 5 are used for statement labels; column 6 is used to indicate the continuation of a statement; columns 7 through 72 contain the statement (or its continuation) or an end line.

In writing a FORTRAN program, blank characters may be used freely to improve readability with these general exceptions: comment lines, the end line, and Hollerith data. Except where noted, blank characters have no meaning and are ignored. Exceptions to the general rule are explained in detail in the applicable paragraphs of the manual.

2.4.1. Comment Line

A comment line is printed only when a program unit is being compiled and is a convenience for the programmer. Only characters from the FORTRAN character set may be used. A comment line is indicated by the character C in column 1 (as shown in Figure 1–7). Any blank character in a comment line will be printed as such. A comment line may not be used between the initial line of a statement and its continuation lines, and must always be followed by an initial line, another comment line, or an end line. No executable code is generated for a comment line.

2.4.2. End Line

One end line must appear as the last line of each program unit to be compiled. Its purpose is to indicate the end of compilation for a particular program unit. Columns 1 through 6 must be blank; columns 7 through 72 must contain the characters END. The characters may be interspersed with, preceded by, and followed by blank characters. Thus, starting from column 7,

```
END
E   N   D
  EN    D
```

are all valid end lines.

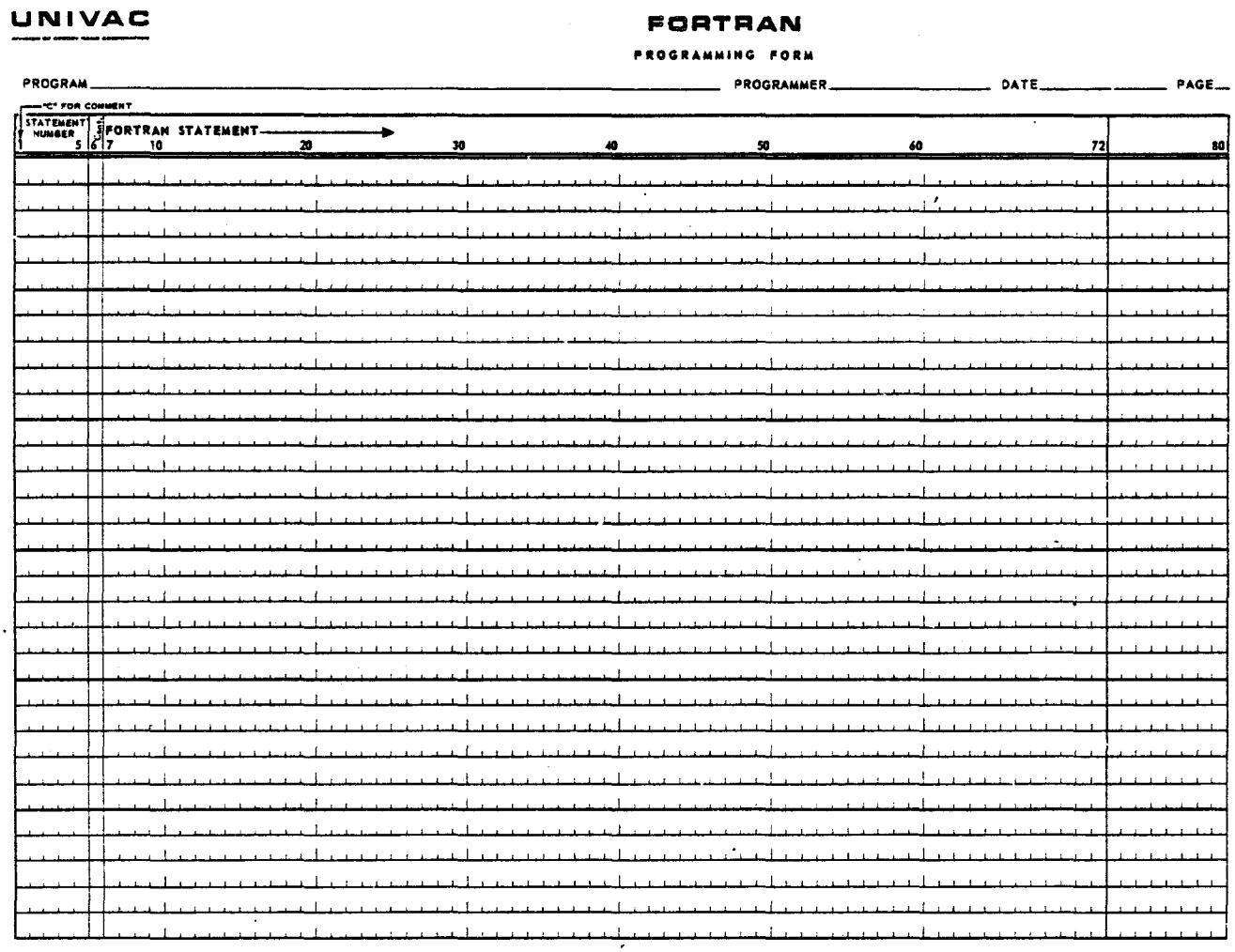
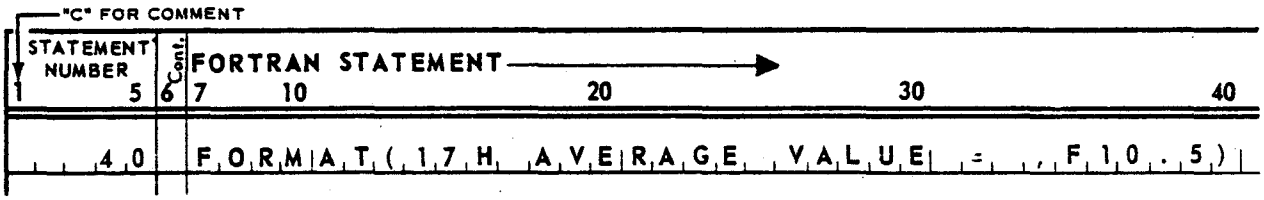


Figure 2-3. FORTRAN Programming Form

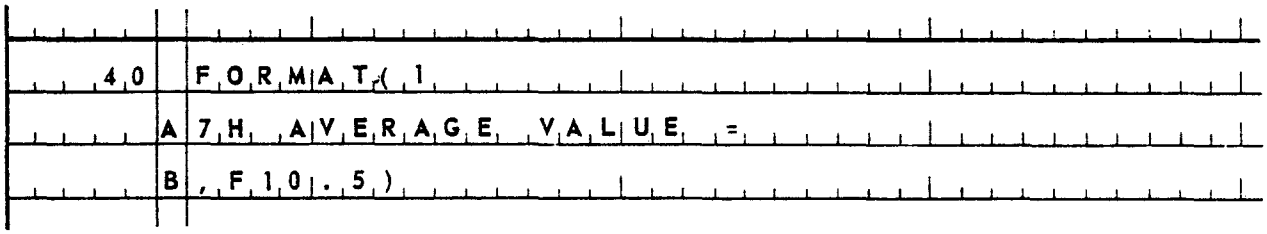
2.4.3. Statements

A statement consists of an *initial line* and, if required, up to 19 *continuation lines* which follow in sequence. Each line of the statement is written in columns 7 through 72. An initial line must have either the digit 0 or a blank character in column 6 and must not have the character C in column 1. Each continuation line must have a character other than the digit 0 or blank character in column 6, and must not have the character C in column 1. Columns 1 through 5 are available to the programmer for documentation (except for the character C in column 1). Each continuation line must be immediately preceded by an initial line or another continuation line.

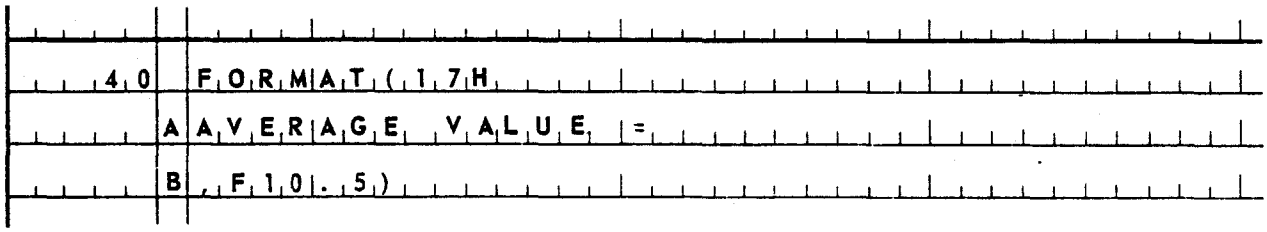
For example, line 20 in Figure 1–7 was written:



It could be written as:



However, it can *not* be written as



because the 17H of the FORMAT statement means that the 17 characters immediately following the H are Hollerith data. As was pointed out in 2.4, blank characters are significant in Hollerith data.

Frequently, the character in column 6 of a continuation line indicates the sequence of continuation lines (it is hoped that what is described as a common practice in this manual is not deemed mandatory by the programmer). For example, the first continuation line might contain the digit 1 in column 6; the second, the digit 2; and so on. Also, since the first five columns of continuation lines are available for documentation (using any characters from the FORTRAN set), these are sometimes used to contain the statement label of the initial line.

2.4.4. Statement Labels

A statement label is an unsigned integer (1 through 99999) that identifies a FORTRAN statement and is written in columns 1 through 5. Only the digits 0–9 and blank characters may be used in a statement label. The same statement label cannot be used more than once in a program unit; the same statement label can appear in more than one program unit. The value of a statement label does not affect the order in which statements are executed.

A statement label is meaningful only when used with the initial line of a statement; it is ignored when used with continuation lines. It enables other statements of the program unit to reference it; therefore, it is superfluous when used with a statement that is not referenced by another statement. Some processors will print out a warning diagnostic message that indicates an unreferenced statement label. The possibility of the programmer having forgotten to insert the referencing statement is strongly indicated.

A maximum of five digits may be used in a statement label. All blanks and leading zeros are neglected.

For example, in the sequence

"C" FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–)
157		F = (A + B – C) / D
		GO TO 157

the statement label of the arithmetic assignment statement could have been written:

157		F = (A + B – C) / D

or

```
      0 1  5 7  F  =  ( A  +  B  -  C )  /  D
```

or

```
      1 5  7    F  =  ( A  +  B  -  C )  /  D
```

but not as

```
      1 5 7  0  F  =  ( A  +  B  -  C )  /  D
```

because control is being transferred to statement label 157, not 1570.

2.5. FORTRAN DATA

Data is information manipulated by a program. An item of data is classified as *arithmetic, logical,* or *Hollerith.* Arithmetic data is used in computations restricted to numbers. Logical data is used to indicate whether a specific condition is *true* or *false.* Hollerith data is information to be used *literally;* it may contain any and all characters of the computer set of characters and is generally used for printing messages, titles, and headings.

2.5.1. Data Types

The various forms in which data may appear are known as *types.* The term *type,* when applied to data in FORTRAN, has a special meaning. An item of data, if arithmetic, must be *integer* type, *real* type, *double precision* type, or *complex* type. If not arithmetic, it must be *logical* type or *Hollerith* type. The different data types are discussed in the remainder of this section.

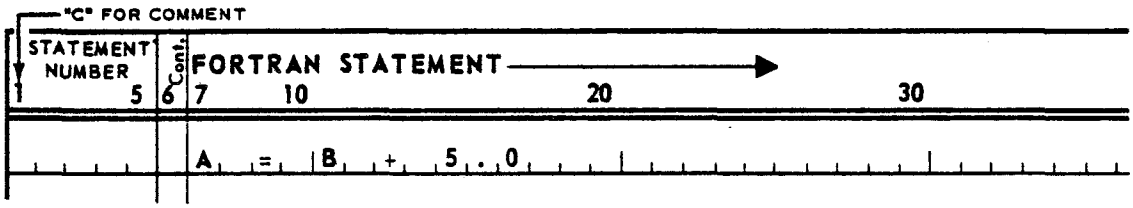
One important attribute of a data type is the number of storage units required for internal representation of a datum. This attribute is of special importance in the use of COMMON and EQUIVALENCE statements, and is listed in Table 2–4 for reference purposes.

DATA TYPE	STORAGE UNITS* REQUIRED
integer	1
real	1
double precision	2
complex	2
logical	1

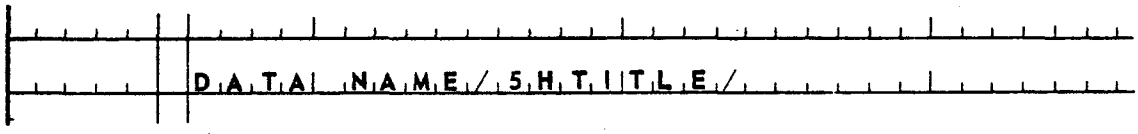
*The term "storage unit" is defined differently for different computers. In one computer, it may mean a word; in another computer, it may mean five consecutive bytes, and so on.

Table 2–4. Memory Requirements for Data Types

An item of data can be expressed as a constant or can be represented by a symbolic name. For example, in



A and B are symbolic names used as variables and 5.0 is a constant. In the DATA statement



NAME is a variable; 5HTITLE is a Hollerith constant representing the Hollerith datum TITLE.

2.5.1.1. Integer Type

An *integer type* datum is always the exact representation of an integer (a string digits without any decimal point). It may assume positive, negative, or zero values (the value zero is considered neither positive nor negative). Each computer places a limit on the number of digits (or the maximum absolute value) that may be contained in an integer type datum. An overflow condition occurs when this is exceeded. The term "exact representation" is used because an integer type datum is represented internally as a fixed-point number.

2.5.1.2. Real Type

A *real type* datum is the processor approximation to the value of a real number (as opposed to a complex number) which may or may not have a decimal point. It is limited to a specific number of significant digits and a specific range of values, both of which differ for different processors. The real type value may be positive, negative, or zero (the value zero is regarded as neither positive nor negative). When the range is exceeded for a value, an overflow condition occurs; when the value is less than the lower limit of the range and is not zero, an underflow condition exists. A real type value is an approximation because it is represented, internally, in floating-point form.

2.5.1.3. Double Precision Type

A *double precision type* datum, like a real type datum, is also the computer approximation of a real number, but the degree of approximation is greater than that of a real type datum because it can contain more significant digits. The number of significant digits is not necessarily double that of a real type value; it will be greater than that of a real type value, and this number must be specified for each processor. In addition, its range of values may be the same as or greater than, rarely less than, that of a real type value. This range must also be specified for each processor. A double precision type datum may be positive, negative, or zero. Here, as for all arithmetic values, the value zero is treated as neither positive nor negative.

2.5.1.4. Complex Type

A *complex type* datum is the processor approximation to the value of a complex number. It is represented as a pair of real type values; the first of the pair represents the real part of the complex value, the second of the pair represents the imaginary part of the complex value. Each part, therefore, has the same degree of approximation as a real type datum.

2.5.1.5. Logical Type

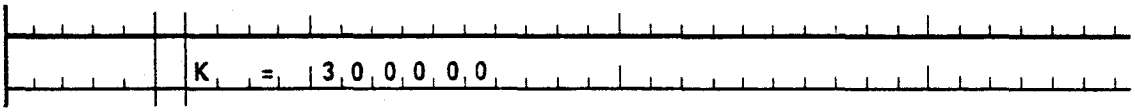
A *logical type* datum is an item that may assume only the values *true* or *false*. There is no standard internal representation for these values; internal representation depends upon the processor.

2.5.1.6. Hollerith Type

A *Hollerith type* datum is a string of characters taken from the processor set (which includes the FORTRAN set) of characters. The blank character is a significant character in Hollerith type data.

2.6. CONSTANTS

A *constant* is a value (arithmetic, logical, or Hollerith) that is defined by its appearance in a DATA or an executable statement. Once defined, it must not be redefined during execution of the program. For example, in the arithmetic assignment statement



the integer 300000 is an arithmetic constant.

Throughout this manual the term "constant" alone, when applied to an arithmetic constant, means a constant without a sign. An integer, real, or double precision constant is said to be *signed* when it is written immediately after a plus or minus sign. An *optionally signed constant* may have the plus sign omitted; the minus sign cannot be omitted for a negative value.

In the following discussions it is arbitrarily assumed that an integer type datum can consist of up to nine digits and a sign; a real type datum, up to eight significant digits; and a double precision type datum, up to 15 significant digits. It is also assumed that one storage unit can contain five Hollerith characters. These assumptions are made for use in examples. These specifications differ for each processor and the programmer should consult the reference manual for the particular processor being used.

2.6.1. Integer Constant

An *integer constant* is a nonempty string of decimal digits. No decimal point, comma, alphabetic, or special character (except blank characters) may appear anywhere in the string. Blank characters and leading zeros are ignored except where noted. The value 0 is a valid integer constant.

Examples:

0	valid integer constant
0000	valid integer constant
123000123	valid integer constant
123,456	invalid integer constant because of comma
.0	invalid integer constant because of decimal point
0.	also invalid integer constant because of decimal point
~~bbbbb~~	invalid integer constant because the string must not be empty

2.6.2. Real Constant

A *real constant* may be written in any one of the following three ways:

(1) As a *basic real constant*. A basic real constant consists of an integer part, a decimal point, and a decimal fraction part, in that order. Both the integer and the decimal fraction part are strings of decimal digits and (possibly) blank characters. Either part may be empty (consisting of only blank characters), but not both. Except for this restriction, any blank characters in the constant are ignored by the processor.

Examples:

.0	valid real constant
0.	valid real constant
0.0	valid real constant
123.45678	valid real constant
000123.45678	valid real constant. Leading zeros are ignored.
123.456789	valid real constant. The presupposed limit of eight significant digits is exceeded, but the processor can approximate the written constant by an appropriate real type value.
~~bbb.bbb~~	invalid real constant because integer part and decimal fraction part are empty

(2) As a basic real constant followed by a decimal exponent. A decimal exponent is the letter E followed by an optionally signed decimal integer. This decimal integer has a range unique to each processor, usually two or three digits, with leading zeros ignored. The decimal exponent has the effect of multiplying the preceding constant by a power of 10, as specified by the optionally signed decimal integer.

Examples:

.0E0	valid real constant
1.23E2	valid real constant. It could also have been written as a basic real constant, 123.0
123456.01E−2	valid real constant. It could also have been written as a basic real constant, 1234.5601
123456.01E+05	valid real constant. It could also have been written as the basic real constant 12345601000.0. The zeros trailing the rightmost 1 are not handled as significant digits, but as positional digits.

There is no assurance that equivalent mathematical versions of a value will yield identical processor values; these will be equivalently approximated by the processor.

(3) As an integer constant followed by a decimal exponent.

Example:

12345601E−4	valid real constant. It is equivalent to the real constant 123456.01E−2 or to the real constant 1234.5601.

Figure 2−4 shows different ways of stipulating the values $0.1234 \cdot 10^{-4}$ and $-0.1234 \cdot 10^{4}$ as real constants in FORTRAN statements of a program.

ALGEBRAIC VALUE	FORTRAN EQUIVALENT
$0.1234 \cdot 10^{-4}$	0.1234E-4
	+.1234E-4
	1234E-08
	0.00001234
$-0.1234 \cdot 10^{4}$	-0.1234E4
	-0.1234E+4
	-.1234E+04
	-1234E0
	-1234.

Figure 2−4. Real Constants in FORTRAN Statements

2.6.3. Double Precision Constant

A *double precision constant* may be written in either of the following ways:

(1) As a real part (of exactly the same form as a basic real constant) followed by a double precision exponent. A double precision exponent is similar to a decimal exponent, except that the letter D is used instead of the letter E.

Examples:

12345678.9012345D0	valid double precision constant
12345678.9012345D–5	valid double precision constant
123.0D3	valid double precision constant

(2) An integer part followed by a double precision exponent.

Example:

123456789012345D–5	valid double precision constant

2.6.4. Complex Constant

A *complex constant* is the processor approximation of a complex number. It is written as a pair of optionally signed real constants enclosed in parentheses. The first of the pair is the real part. This is followed by a comma and then the second of the pair, which is the imaginary part.

Examples:

(1.2 , 3.4)	valid complex constant. It represents 1.2 + 3.4*i*.
(0.0 , 3.4)	valid complex constant. It represents the imaginary number 3.4*i*.
(–34E–2, –4.5E3)	valid complex constant. It represents –0.34–4500*i*.
(–3400, –4.5E3)	invalid complex constant. *Both* constants of the pair must be real type constants.

2.6.5. Logical Constant

A *logical constant* is written as either .TRUE. or .FALSE. The periods are required, as shown. It represents a condition as either *true* or *false*, respectively. An analogy is a switch which is either on or off, with .TRUE. corresponding to on, and .FALSE. corresponding to off; then, in a program, the programmer can test the setting of the switch and condition the sequence of execution accordingly. In the following sequence, a logical IF statement is used for testing.

```
      LOGICAL EVEN
      EVEN = .TRUE.
      IF (EVEN) GO TO 100
   10
  100
```

The first statement is a type-statement which informs the compiler that the variable, EVEN identifies a logical value, that is, either true or false. The second statement is a logical assignment statement that sets EVEN to true. The third statement is a logical IF statement that means: IF EVEN is true go to statement 100; otherwise, go to the next statement (statement 10).

2.6.6. Hollerith Constant

A *Hollerith constant* is written as the letter H preceded by an integer constant which specifies the number of characters following the H that are part of the Hollerith constant. A Hollerith constant may appear only in a DATA statement and in a CALL statement.

Examples:

5HTITLE	valid Hollerith constant
6HIS NOT	valid Hollerith constant. The space character must be counted.
5HA=$5b	valid Hollerith constant. All characters from the processor character set are permissible (see 2.3).

NOTES:

(1) Although Hollerith constants can appear only in a DATA or CALL statement, Hollerith data may appear in input/output data and in the FORMAT statement.

(2) The internal representation of a Hollerith value may be different from the internal representation of an arithmetic value. For example, 1H1 need not have the same internal representation as the arithmetic value 1, and, if they are compared, they will generally be treated as unequal. The FORTRAN language does not define any correspondence between Hollerith values and arithmetic or logical values.

2.7. SYMBOLIC NAMES

A symbolic name consists of one to six alphanumeric characters (any blank characters are ignored, other symbols are prohibited), the first of which must be a letter. A symbolic name is followed by a subscript only when used as an array element reference. In a program unit, a symbolic name may identify an item in one (and usually only one) of the classes (array and array elements are considered the same class) shown in Table 2–5. The remainder of this section discusses the use of symbolic names for variables, arrays, and array elements. The programmer is entirely free in the choice of words for symbolic names. Keywords such as FORMAT, GO TO, READ, etc., are not considered reserved words; these may be used as symbolic names, or the first letters of a symbolic name anywhere in the program.

A SYMBOLIC NAME THAT IDENTIFIES A(N)	IS CALLED A(N)	TYPE ASSOCIATION?*	TEXT REFERENCE
variable	variable	yes	see 2.7.3
array	array name	yes	see 2.7.4
array element	array element reference	yes	see 2.7.4
statement function	statement function name	yes	see 8.2
intrinsic function	intrinsic function name	yes	see 8.3
subroutine	subroutine name	no	see 8.6
external function	external function name	yes	see 8.5
external procedure name alone as an actual argument	external procedure name	not necessarily	see 8.7
block	block name	no	see 8.8

*Type association means that the data type is either implied in the name or explicitly specified (see 2.7.2).

Table 2-5. Uses of Symbolic Names

2.7.1. Uniqueness of Symbolic Names

A symbolic name cannot be used in a program unit for more than one purpose except as noted in the following rules.

A block name can be used as an array name, variable, or name of a statement function in the same program unit.

In a function subprogram, the function name that follows FUNCTION in the header of the subprogram must also appear as a variable in the same program unit.

Once a symbolic name is used as an external function name, subroutine name, procedure name, or block name in any program unit, it may not be used anywhere in the program for the same purpose except as originally used.

A symbolic name that appears in an EXTERNAL statement of a program unit and is used only as an actual argument within that program unit must be the name of an external function or subroutine in the same program. However, that same name may be used in any other program unit for a valid purpose.

A symbolic name used to identify an intrinsic function in one program unit may be used for any purpose in another program unit.

2.7.2. Typing of Symbolic Names

It was indicated in Table 2–5 that certain uses of symbolic names have data type association. This means that the compiler must be informed of the data type for the symbolic name, that is, integer type, real type, double precision type, complex type, or logical type. There is no mechanism in FORTRAN for specifying Hollerith type; data of this type, other than Hollerith constants, are identified under the guise of a real, integer, or logical type variable or array element, as will be shown in one of the following paragraphs.

The data type of a symbolic name can be explicitly declared or, if not explicitly declared, implied by the first letter of the symbolic name. A constant automatically defines its own type by the form of its appearance.

2.7.2.1. Explicit Type Declaration

Explicit type declaration means that the symbolic name is declared in a type-statement. The one exception for explicit type declaration is the function name that appears in the FUNCTION statement of a subprogram. In the function subprogram, this symbolic name can only be explicitly typed in the FUNCTION statement.

"C" FOR COMMENT

STATEMENT NUMBER 1–5	Cont. 6	FORTRAN STATEMENT → 7–10–20–30
		REAL IOTA, LAMBDA
		DOUBLE PRECISION BETA, GAMMA
		COMPLEX ALPHA, A24
		INTEGER SIGMA
		LOGICAL XXX, YYY, KKK

This group of five statements consists of type-statements. The first statement specifies that IOTA and LAMBDA represent real type data, the second, that BETA and GAMMA represent double precision data; and so on.

2.7.2.2. Implied Type Declaration

Implied type declaration means that the data type associated with a symbolic name is implied by the first letter of its name, unless the data type of the symbolic name is declared explicitly. If the first letter of the symbolic name is any of the six letters I, J, K, L, M, or N, the compiler will assume integer data type for that name; if the first letter is any of the other alphabetic characters the compiler will assume real data type for that name.

This rule for implied data type association clearly indicates that symbolic names representing double precision, complex, or logical data must always be typed explicitly. An exception to this rule is that references to intrinsic or basic external functions (see Section 8) do not require explicit type declaration in the referencing program unit. The function names involved already have the data type of the function names known to the processor.

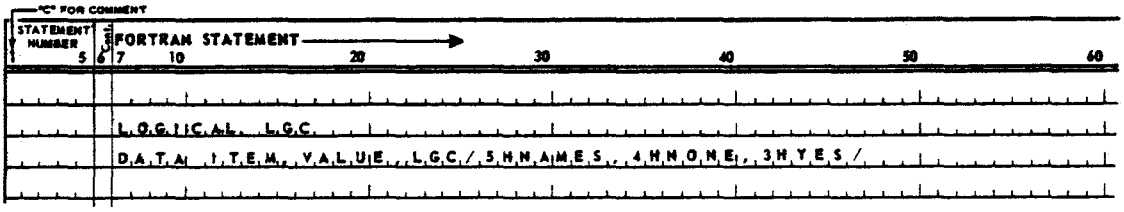
Examples:

```
      DOUBLE PRECISION V2, V4, V6
      LOGICAL TEST
      COMPLEX NMBR5, NMBR6
      .
      .
      KADD = NMBR2 + NMBR4
      .
      .
      IF(TEST) VALUE=VALUE + 1.0
      V2 = V4 - V6
      NMBR6 = (12.34,45.67) + NMBR5
```

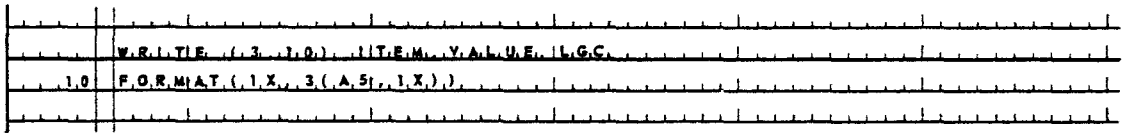
KADD is the sum of two integer values (all implied type declarations). In the next statement, if the logical value of TEST (explicitly declared) is true, VALUE is increased by one (implied real type). In the statement that follows, the double precision V2 is formed by the subtraction of V6 from V4 (all explicitly declared as double precision). In the last statement, the complex value NMBR6 is formed by adding 12.34 to the real part of NMBR5 and 45.67 to the imaginary part of the complex type NMBR5 (explicitly declared).

2.7.2.3. Hollerith Values

It has been stated that there is no mechanism in FORTRAN for specifying Hollerith type, as such, and that it must be guised as a real, integer, or logical type. The following example shows how this can be done.



The symbolic names LGC, ITEM, and VALUE each can represent five Hollerith characters (from the original assumption in 2.6). The variable ITEM contains the characters NAMES; the variable VALUE, the characters NONE followed by a blank character; the variable LGC, the characters YES followed by two blank characters. It is the programmer's responsibility to remember that LGC, ITEM, and VALUE contain Hollerith data; there is no way for the processor to "remember" this. In FORTRAN, there is no provision for moving Hollerith data from one internal location to another, or otherwise processing the data (as in comparisons). The Hollerith data from the previous example can be used in the following sequence:



This will cause a line to be printed as:

bNAMESbNONEbbYESbbb

The A5 in the FORMAT statement informs the processor that the contents of ITEM, VALUE, and LGC are to be treated as Hollerith data.

If the variables ITEM and VALUE are used in arithmetic statements, or the variable LGC in a logical expression, the results are unpredictable in standard FORTRAN and will vary, depending upon processor implementation.

2.7.3. Variables

A variable is a symbolic name that identifies a single value. The rules for declaring the data type of a variable have already been discussed. Once the value for a variable has been defined, it may be redefined as many times as required by the program. It may be defined by a DATA statement (Figure 1–7, line 4), by an input/output statement (Figure 1–7, line 6), by an assignment statement (Figure 1–7, line 17), by its use as an argument in a subprogram reference (discussed in Section 8), by its use as a DO statement or DO-implied list control variable, or by association in a COMMON or EQUIVALENCE statement. A variable cannot be used unless its value has been defined at least once. For instance, in Figure 1–7, if the original value of KOUNT had not been defined as zero, its use as a running count would have been meaningless. The same is true for the variable TOTAL used as a running total. Failure to define a variable used for such purposes is quite common and leads to unpredictable results.

2.7.4. Arrays

An *array* is an ordered set of values, each of which is called an *array element*. The entire set of values is identified by a symbolic name called the *array name*. All elements of the array must have the same data type, which is determined by the data type of the array name.

The use of array elements in FORTRAN corresponds to the use of subscripted variables in ordinary algebra. For instance, the algebraic expression $a - b - c$ uses simple variables which correspond to the use of variables as defined in FORTRAN. However, the algebraic expression $a_1 - a_2 - a_3$ uses subscripted variables with one subscript; the algebraic expression $b_{1,1} - b_{1,2} - b_{1,3}$ uses what is called in FORTRAN two-dimensional array elements; the expression $c_{1,1,1} - c_{1,1,2} - c_{1,1,3}$ uses three-dimensional array elements. The first expression is written in FORTRAN as A(1) – A(2) – A(3), for the one-dimensional array named A; the second is written as B(1,1) – B(1,2) – B(1,3), for the two-dimensional array named B; and the third is written as C(1,1,1) – C(1,1,2) – C(1,1,3), for the three-dimensional array named C. An array, therefore, is nothing more than an ordered list, any element of which can be identified by the array name followed by the appropriate subscript.

If it is assumed that the maximum value of each dimension is 3, then the order of the elements in array A is:

A(1)
A(2)
A(3)

The order of elements in array B is:

B(1,1)
B(2,1)
B(3,1)
B(1,2)
B(2,2)
B(3,2)
B(1,3)
B(2,3)
B(3,3)

The order of elements in array C is:

C(1,1,1)
C(2,1,1)
C(3,1,1)
C(1,2,1)
C(2,2,1)
.
.
.
C(3,3,1)
C(1,1,2)
C(2,1,2)
.
.
.
C(3,3,2)
C(1,1,3)
C(2,1,3)
C(3,1,3)
C(1,2,3)
C(2,2,3)
C(3,2,3)
C(1,3,3)
C(2,3,3)
C(3,3,3)

Note that the leftmost integer of the subscript varies most rapidly in the order of progression.

The total number of storage units required for an array is the sum of the storage units required for its elements. If the data type of array name A is integer, real, or logical, array A will require 3 storage units (see Table 2–4); if array name A is double precision or complex, it will require 6 storage units. Under the same conditions, array B will require 3 · 3 or 2(3 · 3) storage units; array C will require 3 · 3 · 3 or 2(3 · 3 · 3) storage units.

In a simple situation, visualize a book with the title NAME consisting of only columns of statistics on each page. Let the leftmost subscript represent the row; the next subscript, the column; and the last subscript, the page. Then, to obtain a single statistic, reference the desired statistic by title, row, column, and page. An example is NAME (3,4,5), referring to the third row of the fourth column of the fifth page of the book called NAME. This type of reference is called an array element reference (sometimes called a subscripted variable).

2.7.4.1. Array Declaration

Before an array or any of its elements can be referenced in an executable statement of a program unit, the array must be *declared*. This means that the name, data type, and maximum number of array elements required must be stipulated.

An array can be declared in a DIMENSION statement, COMMON statement, or type-statement. The form of the declarator is

v (i)

where: v is a symbolic name called the array name, and *i*, the declarator subscript, is composed of 1, 2, or 3 expressions, each of which may be either an integer constant or an integer type variable, except that a COMMON statement does not permit variables.

The parentheses are required as shown. Each expression is separated from the next by a comma. If any expression is an integer type variable, the array is known as an *adjustable array*.

The data type of the array name is governed by the rules for implied and explicit typing of symbolic names.

Examples:

```
      DOUBLE PRECISION VCTR(3,4,5),MTRX(2,3),STATX(15),MANY
      DIMENSION LIST2(4,5,6)
      LOGICAL LIST2
      COMMON MANY(2,3)
C     LIST2 COULD HAVE BEEN DECLARED BY
      LOGICAL LIST2(4,5,6)
C
C     THE FOLLOWING IS AN ADJUSTABLE ARRAY DECLARATION
      DIMENSION INTGRS(K2,4,K6)
C     NOTE IMPLIED TYPING
```

The first statement shows array declaration with a type-statement. The arrays VCTR, MTRX, and STATX are double precision arrays: VCTR is a three-dimensional array requiring 120 storage units; MTRX is a two-dimensional array requiring 12 storage units; STATX is a one-dimensional array requiring 30 storage units.

The next group of three lines shows two different ways of declaring an array. The first is array declaration with a DIMENSION statement and a type-statement; the second shows array declaration in a COMMON statement of the double precision array MANY.

The last group of lines shows array declaration for a three-dimensional adjustable array of integer type, INTGRS. Such a declaration (with integer type names as subscripts) can appear only in a procedure subprogram. The values for K2 and K6 will be furnished at execution time when the subprogram is referenced.

In a program unit, any appearance of an array name must be followed immediately by a subscript except, possibly, in any of the following cases:

- in the I/O list of a READ or WRITE statement;
- in the list of dummy arguments of a FUNCTION or SUBROUTINE statement;
- in the list of actual arguments supplied in the reference to an external procedure;
- in a COMMON statement;
- in a type-statement;
- in a formatted READ or WRITE statement, to designate the format specification.

2.7.4.2. Array Element Reference

An array element is one of the items in the set that constitutes the array. An array element is identified by an array element reference of the form:

array name (se)

or

array name (se1,se2)

or

array name (se1,se2,se3)

where *array name* must be the same as that in the array declaration and each *se* is a *subscript expression*. Parentheses and commas are required as shown.

A *subscript expression* is any of the following forms. In these forms, the single asterisk is the FORTRAN symbol for multiplication.

c

v

v + k

v – k

*c * v*

*c * v + k*

*c * v – k*

Each c and *k* is an unsigned integer constant; each *v* is an integer variable name.

Examples:

13

NMBR1

NMBR1 + 13

NMBR1 – 13

2 * NMBR1 + 13

2 * NMBR1 – 13

Note that the order of the different elements must follow the order described in the previous paragraph. As an example: 2 * NMBR1 + 13 is a valid subscript expression; NMBR1 * 2 + 13 is not a valid subscript expression.

Rules:

(1) An array element may not be referenced unless the array has been declared previously.

(2) Except in an EQUIVALENCE statement (where only constants are permitted), any and all forms of subscript expressions can be used.

(3) Except in an EQUIVALENCE statement, the number of subscript expressions must correspond to the number of dimensions in the array declarator.

(4) The value of a subscript expression must be greater than zero.

Examples:

(1) The following sequence shows how the elements of one array, LIST1, can be added to the corresponding elements of a second array, LIST2, to compute the values of the array elements in a third array, LIST3.

```
                "C" FOR COMMENT
STATEMENT NUMBER  Cont.  FORTRAN STATEMENT ------>
1         5       6     7   10        20        30        40
      DIMENSION LIST1(10),LIST2(10),LIST3(10)
C
C
C
      K = 1
   10 LIST3(K) = LIST1(K) + LIST2(K)
      K = K + 1
      IF (K .LT. 11) GO TO 10
C
C
C
```

(2) The following sequence shows how only the even-numbered elements of LIST1 can be added to the corresponding even-numbered elements of LIST2 to form another array, LIST4.

```
      DIMENSION LIST1(10),LIST2(10),LIST4(5)
C
C
C
      K = 1
   10 LIST4(K) = LIST1(2 * K) + LIST2(2 * K)
      K = K + 1
      IF (K .LT. 6) GO TO 10
C
C
C
```

(3) The following sequence shows how, by means of a *nested* loop, the elements of a two-dimensional array can be printed out, one element per line. The operator .LT. in the logical IF statements means "less than."

"C" FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–72)
		DIMENSION NARRAY(3,4)
		.
		.
		.
		K = 1
30		M = 1
10		WRITE(3,50) NARRAY(M,K)
50		FORMAT(15X,I20)
		M = M + 1
		IF(M .LT. 4) GO TO 10
20		K = K + 1
		IF(K .LT. 5) GO TO 30
		.
		.
		.

The elements will be printed in the following order:

NARRAY (1,1)
NARRAY (2,1)
NARRAY (3,1)
NARRAY (1,2)
NARRAY (2,2)
NARRAY (3,2)
NARRAY (1,3)
NARRAY (2,3)
NARRAY (3,3)
NARRAY (1,4)
NARRAY (2,4)
NARRAY (3,4)

2.7.4.3. Location of Elements Within Array

Table 2–6 indicates how to find the relative location of an array element within an array.

IF THE ARRAY IS DECLARED AS	THEN THE ARRAY ELEMENT REFERENCE	REFERS TO THE Nth ELEMENT WHERE N IS
ARRAY (A)	ARRAY (a)	a
ARRAY (A,B)	ARRAY (a,b)	$a + A \cdot (b - 1)$
ARRAY (A,B,C)	ARRAY (a,b,c)	$a + A \cdot (b - 1) + A \cdot B \cdot (c - 1)$

NOTE: A,B,C are the integer values of the array declarator; a,b,c are the integer values of the subscript expressions in the array element reference.

Table 2–6. Array Element Location in Array

Examples:

(1) If an array is declared as LIST1 (15), the array element reference LIST1 (9) refers to the ninth element in the array LIST1.

(2) If an array is declared as NARRAY (3,4), the array element reference NARRAY (2,3) refers to the Nth element where N is 2 + 3(3–1) or the eighth element. (Check with example (3) in 2.7.4.2.)

(3) If an array is declared as INPUT (3,4,5), the array element reference INPUT (3,4,5) refers to the 60th element, since it is the last array element. By checking Table 2–6, it is found that N = 3 + 3(3) + (3) (4) (4) or 60.

(4) If an array is declared as ARRAY (2,3,2), the array element reference ARRAY (1,3,2) refers to the 11th element. However, the array element reference ARRAY (3,2,2) also refers to the 11th element. Using the expression in Table 2–6 for a three-dimensional array, the relative location within the array for the element is evaluated as: 3+2(1)+6(1) or 11.

3. FORTRAN EXPRESSIONS

3.1. GENERAL

An expression is a group of one or more elements and operators which, at each execution, is evaluated as a single value. FORTRAN expressions are: arithmetic expressions, relational expressions, and logical expressions.

Evaluation of expressions is governed by the priority of operators in the expression. Parentheses can be used to force the order of evaluation, regardless of the operators; innermost groups within parentheses are evaluated first.

3.2. ARITHMETIC EXPRESSIONS

An *arithmetic expression* is a group of one or more arithmetic elements and operators which, at execution time, is evaluated as a single arithmetic value.

The rules for the formation of arithmetic expressions are much the same as those used in algebra, except that operations are restricted to exponentiation, multiplication, division, addition, and subtraction operations, each indicated by the proper FORTRAN operator. With these fundamental operations, more elaborate mathematical functions can be built up, such as trigonometric functions and definite integration. Some of these functions are supplied as *basic external functions* (see Section 8); others may be written by the programmer.

3.2.1. Arithmetic Operators

Table 3–1 is a list of the FORTRAN arithmetic operators and their meanings Associated with each operator is a priority number that determines its order of evaluation within an arithmetic expression.

OPERATOR	FUNCTION	PRIORITY	EXAMPLE	
			EXPRESSION	WRITTEN AS
**	exponentiation	1	n^k	N**K
*	multiplication	2	a·b	A*B
/	division	2	$\frac{a}{b}$	A/B
–	unary minus (zero minus)	3	–a	–A
+	unary plus (zero plus)	3	+a	+A
+	addition	4	a + b	A + B
–	subtraction	4	a – b	A – B

Table 3–1. Arithmetic Operators

3.2.2. Formation of Arithmetic Expressions

An arithmetic expression is formed in much the same fashion as in algebra (except for the FORTRAN operators and format requirements for array element references and function references) using (as primary operands) constants, variables, array element references, and function references. Blank characters anywhere in the expression have no significance and are ignored during execution; they may be used freely by the programmer. Parentheses also may be used to indicate grouping of operands and operators. No two arithmetic operators may appear in succession (two asterisks, even if separated by blank characters, are interpreted as the exponentiation operator). The unary plus, which is redundant, and unary minus operators must be preceded by a left parenthesis except when either one is the first (leftmost) nonblank character of the arithmetic expression.

A more rigorous description of an arithmetic expression with formal FORTRAN nomenclature is shown in Figure 3–1 and accompanying text. Note that this figure implies the order of evaluation: primary, factor, term, signed term, simple arithmetic expression, arithmetic expression.

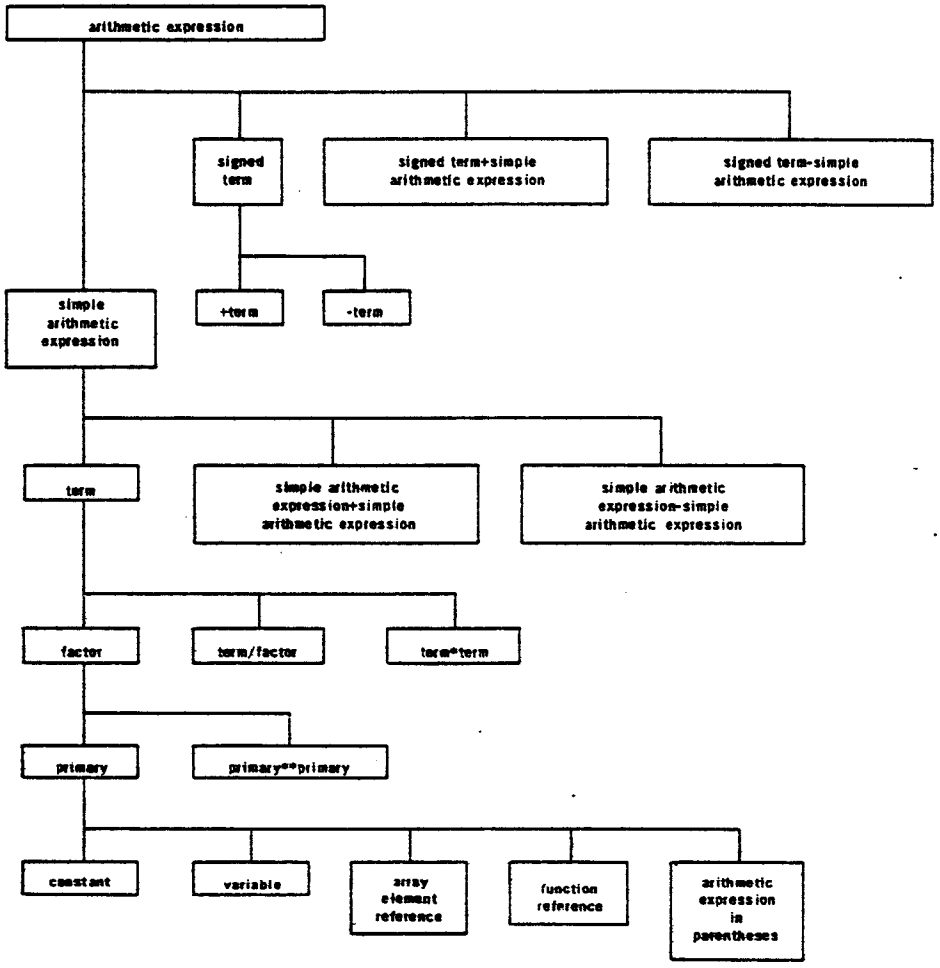


Figure 3–1. Structure of Arithmetic Expression

An *arithmetic primary* is an arithmetic constant, an arithmetic variable, an arithmetic array element reference, an arithmetic function reference, or an arithmetic expression in parentheses.

Examples:

1 2 3.4 4.5E-2 5.6D3 (54E-2, 3.0)	constants
ALPHA BETA LIST1 NAME	variables
SUM (IOTA, 2*NMBR-1, 5)	array element reference
SQRT (A**2 + B**2)	function reference

An *arithmetic factor* is an arithmetic primary or a construction of the form: *primary**primary*.

Examples:

1 4.5E-2 K2**2 K3**NAME (3E-2,0.04) * CMPLX

An *arithmetic term* is an arithmetic factor or a construction of one of the forms: *term/factor* or *term*term*.

Examples:

1 4.5D-2 (3E-2, 0.04) K3**NAME SQRT (4.5),

K3**NAME/34 K3**NAME/34/LIST,

5*6 3.2*SQRT(4.5) K3**NAME/34*2 ALPHA*BETA/GAMMA

A *simple arithmetic expression* is a term or two simple arithmetic expressions separated by a + or a –.

Examples:

1 4.5D2 32.*SQRT(4.5)

3E-2-ALPHA CMPLX + (3.2, 4E-1)

3E-2-ALPHA+BETA

A *signed term* is an arithmetic term preceded by - or +.

Examples:

+2.4 -2.4D2 +SQRT(A-B) -(2.2,3.3) -ARRAY(1,2,3)

An arithmetic expression is a simple arithmetic expression, a signed term, or a signed term followed by a + or - immediately followed by a simple arithmetic expression.

Example:

(-B - SQRT(B**2 - 4.0 * A * C)) / (2.0 * A) which is one of the roots of a quadratic equation

3.2.3. Type Rules for Arithmetic Expressions

The data type of an arithmetic expression involving the exponentiation depends upon the data type of its operands, as shown in Table 3–2.

*primary1**primary2*

PRIMARY 1	PRIMARY 2			
	INTEGER	REAL	D.P.	COMPLEX
INTEGER	integer	u	u	u
REAL	real	real	d.p.	u
D.P.	d.p.	d.p.	d.p.	u
COMPLEX	complex	u	u	u

NOTES: D.P. means double precision.
The letter u means that the result is not defined in standard FORTRAN and it depends upon processor implementation of exponentiation.
If *primary 1* has a negative value, *primary 2* must not be real or double precision.

Table 3–2. Type Rules for Exponentiation

The data type of an arithmetic expression involving an arithmetic operator other than the exponentiation operator or the unary operators depends upon the data type of its operands, as shown in Table 3–3. The data type of a unary operation is the same as its operand.

primary1 op primary2

PRIMARY 1	PRIMARY2			
	INTEGER	REAL	D.P.	COMPLEX
INTEGER	integer	u	u	u
REAL	u	real	d.p.	complex
D.P.	u	d.p.	d.p.	u
COMPLEX	u	complex	u	complex

NOTES: *op* is a nonunary operator: +, –, *, /.
D.P. means double precision.
The letter u means that the result is not defined in standard FORTRAN and depends upon processor implementation of the operation.

Table 3–3. Type Rules for Conventional Arithmetic

Of special importance in Table 3–3 is integer division. Only the integer portion of the quotient is retained; the remainder is dropped without roundoff. Thus, the arithmetic expressions 0/4, 1/4, 2/4, and 3/4 are evaluated as the integer 0.

3.2.4. Evaluation of Arithmetic Expressions

Rules:

(1) If the value of an arithmetic expression is not arithmetically defined, it cannot be evaluated. For example, the following arithmetic expressions must not be used: 0**0, X/0.0, and 0**(-3).

(2) In general, arithmetic expressions are evaluated from left to right governed by the priority of the operator (shown in Table 3–1). In the course of this evaluation, expressions in parentheses are evaluated before proceeding to the next evaluation, with innermost parenthetical expressions evaluated first.

There is a permissible exception. If mathematical use of operators is associative, commutative, or both, the order of evaluation may be changed internally to take advantage of these qualities, provided that integrity of parenthesized expressions is not violated. (An operation is associative if A *op* B *op* C can be evaluated as A *op* (B *op* C) or A *op* B) *op* C with no algebraic change in the results; an operation is commutative if A *op* B can be evaluated as B *op* A with no algebraic change in the result.) The only associative FORTRAN arithmetic operators are + and *; the only commutative FORTRAN arithmetic operators are + and *. Thus, in some processors the expression A + B + C can sometimes be evaluated as A + (B + C) and, at other times (in the same processor) as (A + B) + C; the expression A*B can sometimes be evaluated as if written B*A. The associative and commutative laws do not apply to evaluation of integer terms containing division; evaluation of such terms proceeds from left to right. For instance: K*M/N is evaluated as (K*M)/N.

(3) The evaluation of any function references in the expression must not alter the value of any other element within the expression or statement that contains the function reference (see Section 8).

Examples:

(1) Evaluate (-B-SQRT(B**2.0-4.0*A*C))/(2.0*A) where A, B, and C have the values of 1.0, -3.0, and -10.0 respectively. The order of evaluation is:

(-B-SQRT(B**2.0-4.0*A*C))/(2.0*A)
(a) (-B-SQRT(9.0-4.0*A*C))/(2.0*A)
(b) (-B-SQRT(9.0-4.0*C))/(2.0*A)
(c) (-B-SQRT(9.0-(-40.0)))/(2.0*A)
(d) (-B-SQRT(49.0))/(2.0*A)
(e) (-(-3.0)-SQRT(49,0))/(2,0*A)
(f) (3.0-SQRT(49.0))/(2.0*A)
(g) (3.0-7.0)/(2.0*A)
(h) (-4.0)/(2.0*A)
(i) -4.0/(2.0*A)
(j) -4.0/(2.0)
(k) -4.0/2.0
(l) -2.0

Note that in these steps, the successive evaluations are shown in terms of exact values. In actual practice, these values will be approximated. The degree of approximation will depend upon the processor implementation of real type arithmetic.

(2) Evaluate the expression (N/2)*2-N.

All possible results are shown in the following truth table. (Remember that, in integer division, the remainder is truncated.)

N		RESULT
POSITIVE/NEGATIVE	ODD/EVEN	
positive	odd	-1
-positive	even	0
zero		0
negative	odd	1
negative	even	0

Note that this expression can be used for odd/even testing of an integer value.

(3) Intrinsic functions (see 8.3) or assignment statements can be used to get around some of the restrictions imposed by the mixed type requirements of Table 3–3. It must be understood that some precision may be lost in this process and that it cannot always be done, since the range of a real or double precision value is always greater than the range of an integer value. (The restrictions on the use of an assignment statement for data type conversion are described in 4.2.) The following is an example of the need for such conversion in the calculation of the volume of a room where the length is originally given as an integer.

```
C  NEXT  LINE USES INTRINSIC FUNCTION FLOAT
      VOLUME=FLOAT(LENGTH) *WIDTH*HEIGHT
C  NEXT  LINE SHOWS USE OF ASSIGNMENT STATEMENT
      XLNGTH=LENGTH
      VOLUME=XLNGTH*WIDTH*HEIGHT
```

3.3. RELATIONAL EXPRESSIONS

A relational expression defines a relation between two arithmetic expressions. At execution time this relation is evaluated as either true or false.

3.3.1. Relational Operators

A relational expression is made up of two arithmetic expressions separated by one of the *relational operators* shown in Table 3–4. Blank characters may be used freely to improve readability.

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

NOTE: The periods, as shown, are necessary parts of relational operators.

Table 3–4. Relational Operators

Examples:

If K0, K1, K2, KK1, and KK2 represent the values 0, 1, 2, -1, and -2, respectively, then the expression:

K1.EQ.1 is true.
K2.GT.KK2 is true.
KK1.NE.K0 is true.
KK1.LE.KK2 is false.
K1**K1.NE.K2**K0 is false.

3.3.2. Type Rules for Relational Expressions

Only the combinations indicated in Table 3–5 are permitted for the relational expression:

expl op exp2

where: *op* is any relational operator, and *expl* and *exp2* are arithmetic expressions.

expl op exp2

expl	exp2			
	INTEGER	REAL	D.P.	COMPLEX
INTEGER	yes	no	no	no
REAL	no	yes	yes	no
D.P.	no	yes	yes	no
COMPLEX	no	no	no	no

NOTE: D.P. means double precision.

Table 3–5. Type Rules for Relational Expressions

Where a real arithmetic expression appears with a double precision arithmetic expression, the relation is evaluated as if written as (*expl* - *exp2*) *op* 0D0.

3.3.3. Applications of Relational Expressions

(1) In this sequence a set of data cards is read, each of which contains an integer value right-justified in the first 10 columns of the card. The program is to find the greatest value and print it out. It is assumed that none of these values will be -999999999, so that a card containing -999999999 indicates the end of the set containing at least one integer.

```
      READ(1,10) NMBR
   10 FORMAT(I10)
   20 READ(1,10) INFO
      IF (INFO .EQ. -999999999) GO TO 30
      IF (INFO .GT. NMBR) NMBR = INFO
      GO TO 20
   30 WRITE(3,40) NMBR
   40 FORMAT(I10)
      STOP
      END
```

Note that the contents of the first data card must be read into NMBR to create a basis for comparison. Then the value of the next card is read into INFO. The fourth line is a logical IF statement with the relational expression INFO.EQ. -999999999. If this relation is true, the GO TO 30 statement causes a jump out of the loop and a printout of the contents of NMBR. If this relational expression is false, GO TO 30 is disregarded and the next line, which is another logical IF statement, is executed. This statement tests the value of INFO. If the value of INFO is greater than the value in NMBR, the value of INFO is transferred to NMBR. Then control returns to 20 and a new value for INFO is read. Cards are read until a card containing -999999999 is reached. Note that it is necessary to test for -999999999 *before* the attempt to update NMBR; otherwise, the NMBR printed out will be -999999999.

(2) This sequence compares two complex numbers, KMPLX2 and KMPLX4. If their real portions are equal *and* the imaginary part of KMPLX2 is greater than the imaginary part of KMPLX4, control is passed to one set of statements; otherwise, control is passed to another set of statements (in this case, beginning with the statement label 50).

This sequence introduces two intrinsic functions, REAL and AIMAG. The intrinsic function reference REAL (*x*) obtains the real portion of the complex variable *x*; the intrinsic function reference AIMAG (x) obtains the imaginary portion of the complex variable *x*.

```
      COMPLEX KMPLX2, KMPLX4
      .
      .
      .
      IF (REAL(KMPLX2) .NE. REAL(KMPLX4)) GO TO 50
      IF (AIMAG(KMPLX2) .LE. AIMAG(KMPLX4)) GO TO 50
      .
      .
   50 .
```

Note the numerous parentheses used in the logical IF statements. The format of a logical IF requires one set of parentheses around the relational expression; the format of a function reference requires a set of parentheses around the list of arguments. In writing such statements it is not unusual to occasionally misplace a parenthesis. It is a good idea to check such statements and ensure that the number of left parentheses is the same as the number of right parentheses.

3.4. LOGICAL EXPRESSIONS

A *logical expression* is a group of one or more logical elements and operators which, at execution time, is evaluated as either true or false.

3.4.1. Logical Operators

The three *logical operators* are shown in Table 3–6.

OPERATOR	MEANING	PRIORITY
.NOT.	logical negation	1
.AND.	logical conjunction	2
.OR.	logical disjunction	3

NOTE: Parentheses can force the order of evaluation.

Table 3–6. Logical Operators

The meanings of the logical operators are more precisely illustrated by Table 3–7 where the format of a logical expression is *le1.op.le2* for the .AND. and .OR. operators, and .NOT. *le3* for the .NOT. operator. Each *le* is a logical element.

IF *le1* IS	AND *le2* IS	THEN *le1.AND.le2* IS	THEN *le1.OR.le2* IS
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

IF *le3* IS	THEN .NOT. *le3* IS
true	false
false	true

Table 3–7. Truth Table for Logical Operators

3.4.2. Formation of Logical Expressions

A logical expression is formed with logical elements and logical operators. A description of logical elements and their combination to form a logical expression is shown in Figure 3–2, and the text which follows the figure. Figure 3–2 also implies the order of evaluation: primary, factor, term, logical expression.

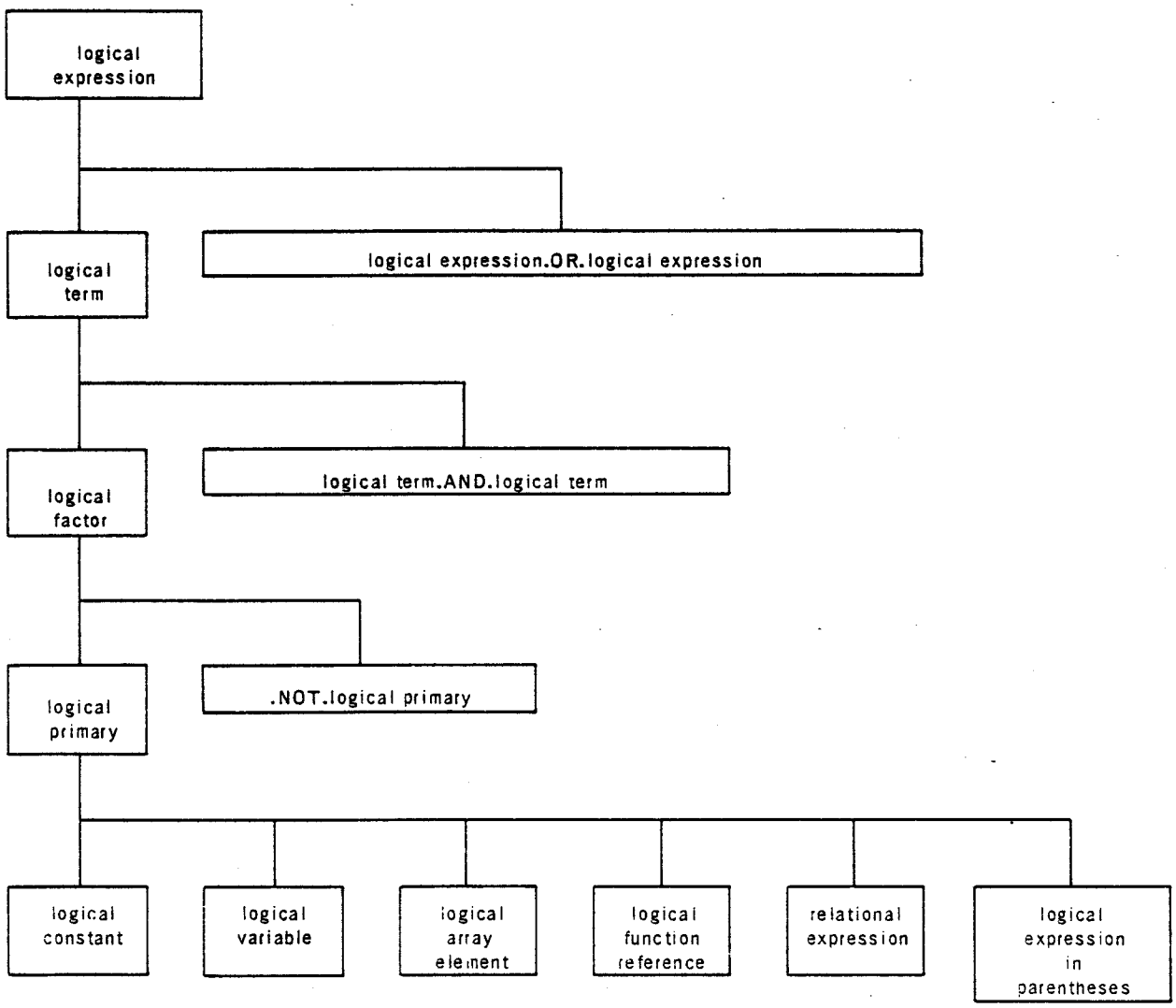


Figure 3–2. Structure of Logical Expression

A *logical primary* is a logical constant, a logical variable, a logical array element, a logical function reference, a relational expression, or a logical expression in parentheses.

Examples:

.FALSE.	logical constant
ALPHA	logical variable
SWITCH (K2, K4, K6)	logical array element reference
XXX(2)	logical function reference
A.LE.0.034	relational expression

A *logical factor* is a logical primary or a construction of the form: .NOT. logical primary.

Examples:

.TRUE.	logical primary
.NOT..TRUE.	evaluated as false
A.NE.B	logical primary
.NOT.A.EQ.B	equivalent to A.NE.B

A *logical term* is a logical factor or a construction of the form: logical term .AND. logical term.

Examples:

ALPHA	logical factor
.TRUE..AND..FALSE	always evaluated as false
.TRUE..AND.ALPHA	evaluated as true only if ALPHA is true
A.GT.B.AND.A.GT.C.AND.D	evaluated as true only if A is greater than B and greater than C, and D is true

A *logical expression* is a logical term or a construction of the form: logical expression .OR. logical expression.

Examples:

.TRUE..OR..FALSE.	always evaluated as true
.FALSE..OR.ALPHA	evaluated as true only if ALPHA is true
A.GT.B.OR.C	evaluated as true if C is true and/or A is greater than B
A.GT.(B.OR.C)	an invalid expression because both operands of a relational expression must be arithmetic expressions
A.GT.B.OR.A.GT.C	evaluated as true if A is greater than C and/or A is greater than B
A.AND..NOT.B.OR..NOT.A.AND.B	called the *exclusive OR* function; evaluated as true only if either A or B is true, but not both
(A.OR.B).AND..NOT.(A.AND.B)	another way of writing the exclusive OR function

3.4.3. Evaluation of Logical Expressions

The order of evaluation of logical expressions is determined by the priority of the logical operator(s), as shown in Table 3–6, and the order of evaluation implied by Figure 3–2, without violating the integrity of elements in parentheses. When parentheses are present, innermost parenthetical expressions are evaluated first. Thus, parentheses can be introduced to force the order of evaluation. When two elements are combined by an operator (or an element follows .NOT.), the element(s) must be evaluated before the logical relation can be evaluated.

Examples:

.NOT.A.AND..NOT.B	evaluated as true only if both A and B are false
.NOT.(A.AND.B)	evaluated as true if A and/or B is false
A.AND.B.OR.C	evaluated as true only if both A and B are true and/or C is true
A.AND.(B.OR.C)	evaluated as true only if A is true and either, or both, B or C is true

3.4.4. Applications of Logical Expressions

An application of logical expressions is shown in the following example.

In this program, an indicator called IND is set to −1 if integers K2 and K3 are both negative, to +1 if both are not negative, or to 0 if one is negative and the other is not.

```
      IND = 0
      IF (K2.GE.0.AND.K3.GE.0) IND = +1
      IF (K2.LT.0.AND.K3.LT.0) IND = -1
```

4. ASSIGNMENT STATEMENTS

4.1. GENERAL

Execution of an assignment statement causes assignment of a value to a variable or array element. This new value becomes its current value until the variable or array element is redefined.

There are three assignment statements in FORTRAN: the arithmetic assignment statement, the logical assignment statement, and the GO TO assignment statement. Because the GO TO assignment statement can only be used with an assigned GO TO statement (which is a control statement), it is discussed in Section 5 in conjunction with the assigned GO TO statement.

4.2. ARITHMETIC ASSIGNMENT STATEMENT

Function:

The arithmetic assignment statement evaluates an arithmetic expression and assigns this value to an arithmetic variable or an arithmetic array element.

$v = e$

where: v is an arithmetic variable or arithmetic array element;

e is any arithmetic expression.

Operation:

An arithmetic assignment statement is performed in up to three steps:

(1) The arithmetic expression e is evaluated to yield a single numerical value.

(2) The data type of this single numerical value is converted to the data type of v.

(3) This converted single value replaces the contents of v.

With arithmetic assignment statements such as K = 3 or S = 3.0, no evaluation or conversion is required. Since the statement can be performed in three distinct steps, it is possible for the same variable or array element to appear in both *v* and *e*. For example, the sequence

```
      K = 2
      K = K + 2
C     THE CURRENT VALUE OF K IS NOW 4
      ARRAY(K+2) = K+3
```

assigns the value of 7.0 (its real type approximation) to the array element ARRAY(6).

The arithmetic expression is evaluated and its value assigned to the variable or array element in accordance with the rules in Table 4–1.

v \ *e*	INTEGER	REAL	DOUBLE PRECISION	COMPLEX
INTEGER	(1)	(2)	(2)	(x)
REAL	(3)	(1)	(5)	(x)
DOUBLE PRECISION	(4)	(5)	(1)	(x)
COMPLEX	(x)	(x)	(x)	(1)

NOTES:

(1) Assign *e* to *v* without change.

(2) Truncate any fractional part of *e* and assign result to *v* as an integer.

(3) Transform *e* to real type value and assign this value to *v*.

(4) Transform *e* to double precision value and assign this value to *v*.

(5) Evaluate *e* by rules of 3.2.4 (or any more meaningful rules), transform to type of *v*, and assign to *v*.

(x) Prohibited combination.

Table 4–1. Type Conversion by Arithmetic Assignment Statement

Rules:

(1) Some arithmetic assignment statements should be avoided to save execution time or avoid possible inaccuracies. For example, the statements A = 1 and I = 10.0 should be replaced, if possible, by the statements A = 1.0 and I = 10, respectively.

(2) Some arithmetic operations can be avoided by using the appropriate form of constant, thus saving execution time. For example, the statement X = 2 E04 is preferred to X = 2.0 * 10.0 ** 4.0. The two values may not be identical due to the approximations involved in the computer representation of real type data.

(3) References to intrinsic functions are available (see Section 8) for handling complex type data and converting the real and imaginary parts as required.

Examples:

(1) This sequence shows how to compute and store the fractional remainder when the integer K2 is divided by the integer K3.

```
      DOUBLE PRECISION K22, K33, REM, QUO
C
C
      K22 = K2
      K33 = K3
      QUO = K2/K3
      REM = K22/K33 - QUO
```

(2) This operation can be source coded more conveniently with the basic external function DMOD (see Table 8–3) as follows:

```
      DOUBLE PRECISION K22, K33, REM
C
C
      K22 = K2
      K33 = K3
      REM = DMOD(K22, K33)
```

(3) The next example introduces the intrinsic function FLOAT and shows how it can eliminate an arithmetic assignment statement used for type conversion.

```
      XLENGTH = LENGTH
      VOLUME = XLENGTH * WIDTH * HEIGHT
C     LINE USES FLOAT FUNCTION
      VOLUME = FLOAT(LENGTH) * WIDTH * HEIGHT
```

4.3. LOGICAL ASSIGNMENT STATEMENT

Function:

This statement evaluates a logical expression and assigns this value (either true or false) to a logical variable or logical array element.

$v = e$

where: v is a logical variable or logical array element.

e is any logical expression.

Rule:

Execution of this statement consists of two parts: the evaluation of the logical expression as either true or false, and the assignment of this logical value to the logical variable or array element.

Examples:

(1) The following sequence evaluates x as true if K is an integer less than or equal to 3.

```
      LOGICAL X
      .
      .
      N=3
      X=K.LE.N
```

(2) With each execution of the following loop, SWITCH will alternate in value from true to false and can be used as an odd/even counter within the loop.

```
      LOGICAL SWITCH
      SWITCH = .FALSE.
   10 SWITCH = .NOT. SWITCH
      .
      .
      GO TO 10
```

5. CONTROL STATEMENTS

5.1. GENERAL

Control statements modify the normal sequence of execution. Some of these statements specify unconditional modification of the normal sequence; others contain a test that determines whether or not the sequence of execution shall be changed.

Execution of a program or procedure starts with the first executable statement and continues sequentially until a control statement is encountered.

The control statements of FORTRAN are:

- GO TO statement
- IF statement
- DO statement
- CONTINUE statement
- STOP statement
- PAUSE statement
- CALL statement
- RETURN statement

The CALL and RETURN statements are not described in this section because they are associated with external procedure subprograms. These statements are described in Section 8. The GO TO assignment statement, because it is associated with the assigned GO TO statement, is described in this section.

5.2. GO TO STATEMENTS

The GO TO statements are:

- unconditional GO TO statement
- computed GO TO statement
- assigned GO TO statement

5.2.1. Unconditional GO TO Statement

Function:

To transfer control, unconditionally, to a statement specified by statement label.

GO TO *sl*

where: *sl* is the statement label of an executable statement within the same program unit.

Rules:

(1) *sl* must be the statement label of an executable statement.

(2) Any executable statement immediately following the unconditional GO TO statement in the program unit must have a statement label, otherwise it can never be executed.

Example:

After the following sequence is executed, the variable K will have a value of 8.

```
      K=4
      GO TO 20
   30 K=7
   20 K=K+4
```

5.2.2. Computed GO TO Statement

Function:

To transfer control to one of several listed statement labels, as determined by a previously defined integer value.

GO TO $(sl_1,sl_2,\ldots,sl_n)$, i

where: each *sl* is a statement label of an executable statement in the same program unit, separated from the next *sl* in the list by a comma.

i is an integer variable representing a value, such that $1 \leq i \leq n$.

Rules:

(1) If the value of i is 1, the statement with sl_1 will be executed; if the value is 2, the statement with sl_2 will be executed; and so on.

(2) The value of i must be defined before execution of the statement.

(3) The parentheses around the list of statement labels and the comma before the integer variable are required as shown in the format.

(4) There is no restriction on other uses for the integer variable i.

(5) There is no standard FORTRAN restriction on the maximum value of i or n, but it is possible that a particular processor may specify a maximun value.

Examples:

(1) After execution of this sequence, control is transferred to the statement with statement label 35.

```
      .
      .
      K2=2
      GO TO (10,20,30), K2
      .
      .
   10 .
      .
      .
   20 ITEM=K2+1
      GO TO (15,25,35), ITEM
```

(2) Another application of the computed GO TO statement shows how it can be used to determine the course of processing in the main program. A data card having a decimal digit punched in the first character position is read in. If this digit is 1 through 4, it indicates a processing sequence. Any other digit (or the blank character) is treated as an end of file indication.

```
      INTEGER CHOICE
    1 READ(1,10) CHOICE
   10 FORMAT (I1)
      IF (CHOICE.EQ.0 .OR. CHOICE.GT.4) STOP
      GO TO (100,200,300,400),CHOICE
  100 .....
      .
      .
      GO TO 1
  200 .....
      .
      .
      GO TO 1
  300 .....
      .
      .
      GO TO 1
  400 .....
      .
      .
      GO TO 1
      END
```

(3) The following example shows how the computed GO TO statement can be used to create a closed internal block (a group of statements that cannot be entered by normal sequential execution, but must be called for execution by control statements).

```
      K = 0
      GO TO 10
   20 K = K + 1
      .
      .                                  CLOSED
      .                                  INTERNAL
      GO TO (100,200,300,400), K         BLOCK
   10 . . . . .
      .
      .
      GO TO 20
  100 . . . . .
      .
      .
      GO TO 20
  200 . . . . .
      .
      .
      GO TO 20
  300 . . . . .
      .
      .
      GO TO 20
  400 . . . . .
```

5.2.3. Assigned GO TO Statement

Function:

To transfer control to one of several listed statement labels, as determined by an integer value previously defined by a GO TO assignment statement.

GO TO $i,(sl_1, sl_2, \ldots)$

where: each *sl* is a statement label of an executable statement in the same program unit, separated from the next *sl* in the list by a comma.

i is an integer variable, followed by a comma, previously assigned a value by a GO TO assignment statement. This value is equal to an *sl* in the list.

Rules:

(1) Prior to execution of the assigned GO TO statement, the integer variable *i* must have been assigned a value by a GO TO assignment statement.

(2) Although standard FORTRAN places no restriction on the number of statement labels in the statement (except that there must be at least one), a particular processor may specify a limit.

(3) Standard FORTRAN specifies that the value assigned to *i* must be an *sl* in the list. If the value of *i* is not an *sl* in the list, some processors will treat this condition as if the value of *i* were in the list. The manual for the processor will specify how this condition is handled.

Example:

Logically, the assigned GO TO statement can be used whenever a computed GO TO statement is used (see 5.2.2). The format requirements differ and the assigned GO TO statement requires at least one previous ASSIGN statement (GO TO assignment statement). Figure 5–1 shows how the assigned GO TO statement can create a multi-legged GO TO after a series of statements shared by different parts of the same program unit.

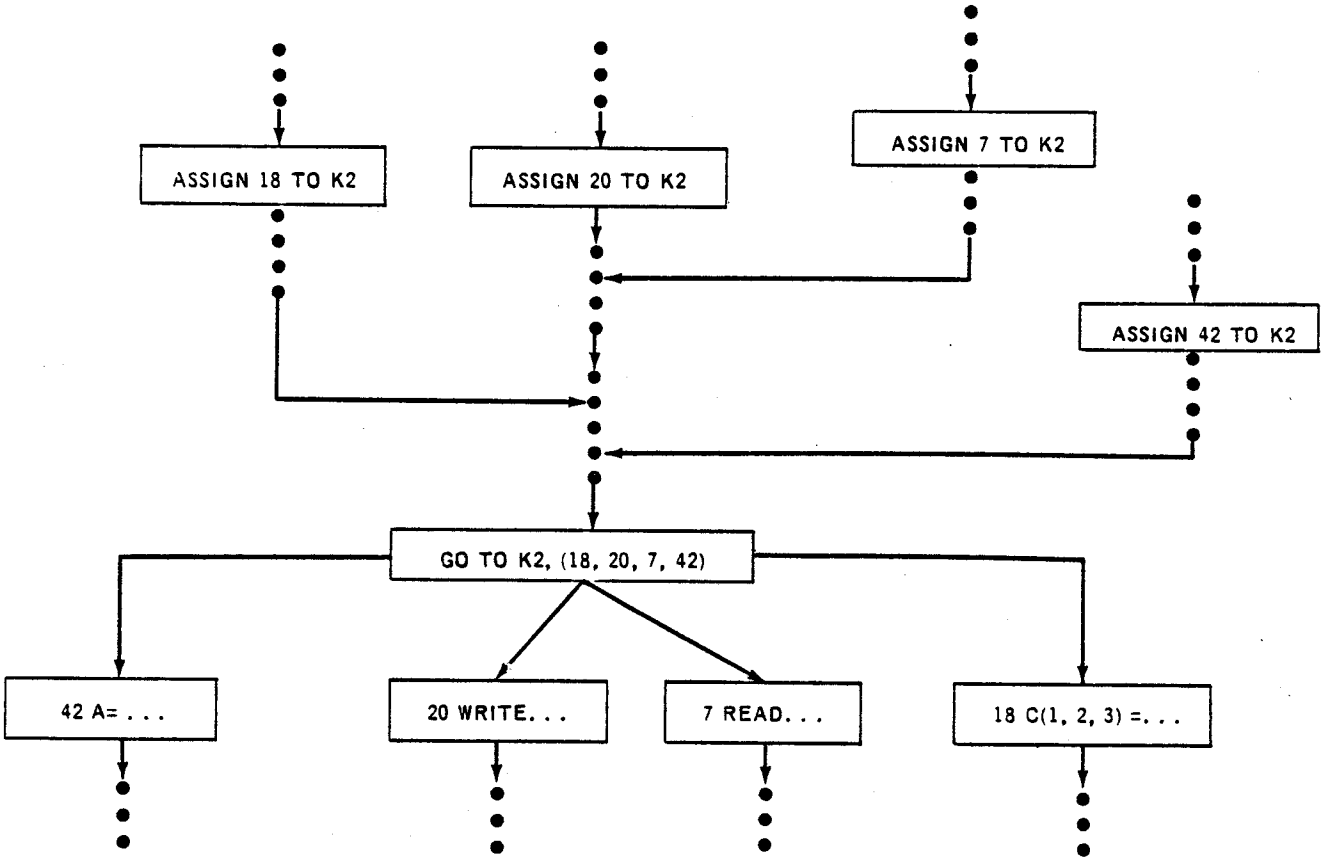


Figure 5–1. Use of Assigned GO TO Statement

5.2.3.1. GO TO Assignment Statement

Function:

To assign a statement label to the integer variable that is used in an assigned GO TO statement.

ASSIGN *k* **TO** *i*

where: *k* is the statement label of an executable statement in the same program unit.

i is an integer variable used in an assigned GO TO statement of the same program unit.

Rules:

(1) The ASSIGN statement must be executed before the assigned GO TO statement to which it applies.

(2) The statement ASSIGN 10 TO J does not have the same meaning as the arithmetic statement J = 10. In particular, the sequence

```
      ASSIGN 10 TO J
      K = J - 1
```

will not produce a meaningful result. If J is to be used as an arithmetic variable after its use in an assigned GO TO statement, it must be defined as such at some point after the assigned GO TO statement. The following sequence illustrates proper use of the ASSIGN statement:

```
      ASSIGN 10 TO J
            .
            .
            .
    1 GO TO J, (10, 20, 30)
            .
            .
            .
   10 J = 15
      K = J - 1
```

Control is transferred to statement 10 after execution of statement J. The variable K is assigned a value of 14.

Example:

See Figure 5–1.

5.3. IF STATEMENTS

The IF statements are the decision-making elements of FORTRAN. The test specified in the IF statement may modify the normal sequence of execution. FORTRAN provides two IF statements: the arithmetic IF statement and the logical IF statement.

5.3.1. Arithmetic IF Statement

Function:

To act as a three-way branch, as determined by evaluation of an arithmetic expression.

IF *(exp)* sl_1, sl_2, sl_3

where: *exp* is any arithmetic expression except complex type.

each *sl* is a statement label of an executable statement in the same program unit.

Rules:

(1) If *exp* is negative, control is transferred to sl_1; if zero, to sl_2; if positive, to sl_3.

(2) The list must contain three statement labels; however, any two, or all three, may be the same. If all three are the same, the statement is, in effect, an unconditional GO TO.

(3) An arithmetic IF statement must not branch to itself. For example, the following statement is illegal:

```
   20 IF (K - J) 10, 20, 30
```

(4) The first executable statement following an arithmetic IF statement must have a statement label or it can never be executed.

(5) If the arithmetic expression is real or double precision type and contains truncations and roundoffs, caution must be exercised, especially when the expression is tested for the zero condition.

Examples:

(1) If, in solving the quadratic equation $ax^2 + bx + c$, where all three coefficients are integer type numbers, the discriminant $b^2 - 4ac$ is negative, the result is two complex roots which are conjugates of each other; if zero, two equal real roots; if positive, two unequal real roots. The following sequence shows how an arithmetic IF statement can be used to select one of three root evaluation procedures based on the evaluation of the discriminant.

```
      NDISC = NB**2 - 4 * NA * NC
      IF(NDISC) 10,20,30
   10     procedure for two complex roots
          .
          .
          .
   20     procedure for two equal real roots
          .
          .
          .
   30     procedure for two unequal real roots
          .
          .
          .
```

(2) Now consider another procedure for the problem in (1), but with the coefficients real type numbers. This sequence introduces the basic function reference ABS(x), which returns the absolute value of a real type argument, x.

```
      DISC=B**2-4.0*A*C
      IF(ABS(DISC)-1E-10) 40,50,50
   40 DISC=0.0
   50 IF(DISC) 10,20,30
   10     two complex roots
          .
          .
          .
   20     two equal real roots
          .
          .
          .
   30     two unequal real roots
          .
          .
          .
```

In this program, all absolute values of the discriminant less than 10^{-10} are treated as zero. This handles possible errors occurring in the value of the discriminant due to computer representation of real type values, truncation, and roundoff. The criteria for these limits vary with the nature of the problem and the manner in which a particular computer treats real type values and computations.

(3) Paragraph 3.2.4, example (2), contains an arithmetic expression that could be used for testing odd/even integers. The following example shows how this expression might be used.

```
      IF((IN/2)*2-N) GO TO 100,200,100
100   routine for odd integers
      .
      .
      .
200   routine for even integers
      .
      .
      .
```

5.3.2. Logical IF Statement

Function:

To determine whether or not a single executable statement, written as part of the logical IF statement, shall be executed.

IF *(e) s*

where: *e* is any logical expression.

s is any executable statement except a DO statement or another logical IF statement.

Operation:

Execution of the IF statement proceeds in two parts: evaluation of the logical expression and (possibly) execution of the statement. If the logical expression, *e*, is evaluated as true, the statement, *s*, is executed; if false, the statement *s* is ignored and control is passed in normal sequence to the next executable statement.

Rules:

(1) A logical IF statement cannot refer to itself. For example, the following statement is illegal.

```
10  IF(A.AND.B) GO TO 10
```

(2) Because execution of the statement proceeds in two parts, the same variable may appear in both the logical expression *e* and the statement *s*. For example:

```
    IF(A.GT.B.OR.A.LT.C) A=B+C
```

(3) Caution must be exercised if the logical expression involves the comparison of real or double precision arithmetic expressions.

Examples:

(1) The following program determines how many months it takes for a deposit to double its original value for three different compound interest rates. The rate is the interest computed at the end of each three-month period, starting from the month of deposit. The three interest rates are 1%, 1.25%, and 1.5%. The months required for the three different rates are stored in the array MONTHS.

```
    DIMENSION MONTHS(3)
    N=1
    RATE=0.01
15  DPST=1.0
    R=RATE+1.0
    K=0
10  K=K+1
    DPST=DPST*R
    IF(DPST.LT.2.0) GO TO 10
    MONTHS(N)=3*K
    GO TO (20,30,40),N
20  RATE=0.0125
    N=2
    GO TO 15
30  RATE=0.015
    N=3
    GO TO 15
40  STOP
    END
```

(2) This program finds the values of y for $y=3x^2+2x+5$ for values of x between 0.1 and 0.2 at intervals of 0.001, that is, x = 0.100, 0.101, 0.102, . . ., 0.199, 0.200. This gives 101 values of y, stored in array Y.

```
      DIMENSION Y(101)
      N=0
   10 N=N+1
      K=99+N
      X=FLOAT(K)/1E3
      Y(N)=3.0*X**2+2.0*X+5.0
      IF(N.LT.101) GO TO 10
      STOP
      END
```

Because the integer type variable N is used for the counter and in the logical IF statement, the result will be exactly 101 values. Because a division operation is used to compute X each time around the loop, each sample of X will be as close to the desired value as the processor approximation to a real number permits. This avoids any pitfalls that might arise if X had been incremented each time around the loop by a real type value, by avoiding possible approximations due to successive truncations and roundoffs.

5.4. DO STATEMENT

Function:

To initiate and control repeated execution of a set of executable statements.

DO *n* $i=m_1,m_2,m_3$

or

DO $i=m_1,m_2$

where: *n* is the label of an executable statement called the terminal statement, which follows (not necessarily immediately) the DO statement.

i is an integer variable called the control variable.

m_1 is an integer constant or an integer variable called the initial parameter.

m_2 is an integer constant or an integer variable called the terminal parameter.

m_3 is an integer constant or an integer variable called the incrementation parameter. In the second form, its value is implicitly 1.

Operation:

For successive repetition of the same group of executable statements, the DO statement eliminates separate statements that set a variable to a starting value, increments this value after the group has been executed, and tests the new value to determine whether the group shall be executed again, as was done in the sample program of 1.6. The operation of a DO statement is shown in the following examples.

WITHOUT DO STATEMENT	WITH DO STATEMENT
$i=m_1$ *label first statement of set* . . . . *n terminal statement of set* $i=i+m_3$ IF (i.LE.m_2) GO TO *label*	DO *n* $i=m_1,m_2,m_3$ DO range { *first statement of set* . . *n terminal statement of set*

The steps in the execution of a DO statement are:

(1) The control variable i is assigned a value represented by m_1. This value must be less than or equal to the value represented by m_2.

(2) The range of the DO is executed. If this range contains a reference to a function or subroutine, the function or subroutine is considered part of that range when it is executed.

(3) After execution of the terminal statement, the control variable is incremented by the value of m_3.

(4) This new value of the control variable is tested. If it is less than or equal to the value of m_2, program control is transferred back to the first statement of the DO range, with the new value for the control variable; if it is greater than m_2, the DO is satisfied, and the control variable becomes undefined.

(5) If this DO range is nested within another DO range, and both have the same terminal statement, when the inner DO is satisfied, the control variable of the next outer DO is incremented and tested, and its DO range (which includes the inner DO) will be repeated until satisfied. This will continue for all nested DO statements sharing the same terminal statement until the outermost DO is satisfied. If there is no nesting of DO's with the same terminal statement, after a DO statement is satisfied, its control variable becomes undefined and program control is passed to the first executable statement after the terminal statement.

Rules:

(1) The terminal statement must be in the same program unit as the DO statement. It must not be a GO TO, arithmetic IF, RETURN, STOP, PAUSE, or a logical IF containing any of these forms. However, if the logic of a DO range indicates that such a statement is a terminal statement, such a statement can be followed by a CONTINUE statement (which has no logical function); the CONTINUE statement is then labeled and used as the terminal statement of the DO range. If the terminal statement is a CALL statement, the subroutine will be executed; after the RETURN of the subroutine is executed, the control variable is tested to determine whether the DO range shall be executed again.

(2) At execution time the parameters of the DO statement must be defined as values greater than zero.

(3) Because the control variable is tested at the end of the DO range execution, a DO statement will always be executed at least once when encountered.

(4) No statement in a DO range may redefine the control variable or any parameter of the DO statement; however, the control variable may be referenced in the DO range, as in:

DO *n* $i=m_1,m_2,m_3$

.
.
.

K=*i*+INT

.
.
.

n terminal statement

(5) If a control statement causes an exit from a DO range before the DO is satisfied, the control variable remains defined until redefined.

(6) DO statements can be nested in outer DO statements with this restriction: the range of each nested DO must be completely contained in the range of its next outer DO and may share the same terminal statement.

VALID NEST	INVALID NEST
DO 10 *i*= . . .	(The range of the second DO is not contained within the range of the first DO.)
DO 20 *i*= . . .	DO 10 *i*= . . .
DO 20 *i*= . . .	DO 20 *i*= . . .
20 executable statement	10 executable statement
10 executable statement	20 executable statement

A special type of nest is a *completely nested nest.* This is a nest of DO statements which satisfies both of the following conditions:

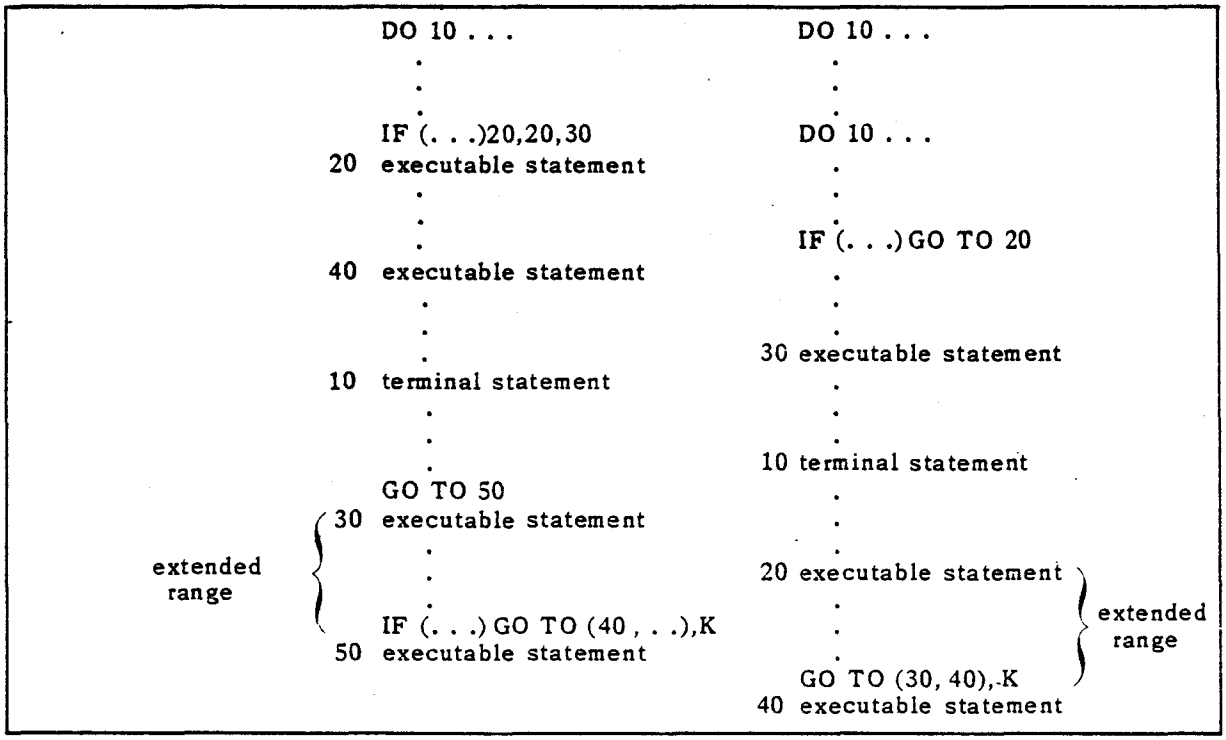
- The outermost DO statement of the nest is not contained in the range of another DO statement.
- The first occurring terminal statement within the nest physically follows (not necessarily immediately) the last DO statement within the nest.

COMPLETELY NESTED NEST	VALID NEST BUT NOT COMPLETELY NESTED NEST
	(The first occurring terminal statement does not follow the last DO.)
DO 10 . . .	DO 10 . . .
DO 20 . . .	DO 20 . . .
DO 20 . . .	20 executable statement
	DO 30 . . .
20 executable statement	30 executable statement
10 executable statement	10 executable statement

A DO range can have an *extended range* if it is a nonnested DO or the innermost nest of a completely nested nest and both of the following conditions are true:

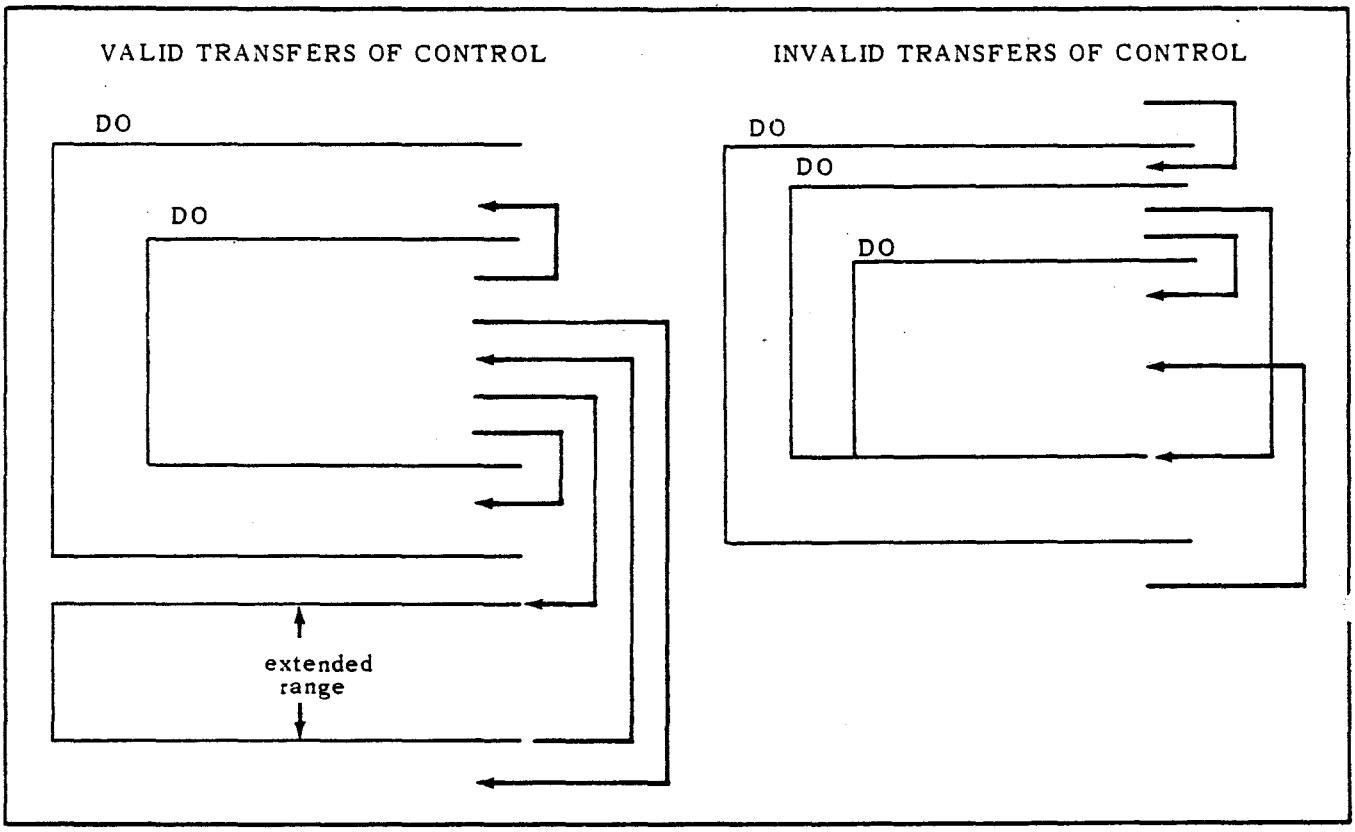
- The range contains a GO TO, or arithmetic IF, or a logical IF containing either of these two statements, that can pass control out of the range to another set of executable statements.
- This set of executable statements contains a control statement that could return control back to the DO range.

Examples of extended ranges are:



The extended range of a DO must not contain, in the same program unit, a DO statement that has an extended range.

(8) A GO TO, arithmetic IF, or a logical IF with either of these two forms cannot pass control into the range of a DO unless that GO TO or arithmetic IF is being executed as part of the extended range of that DO. If more than one DO statement share the same terminal statement, only the *innermost* DO range may contain a GO TO or arithmetic IF that transfers control directly to the terminal statement. The following shows examples of legal and illegal transfers of control:



Examples:

(1) The following sequence shows how to set all array elements of a one-dimensional array to zero.

```
      DIMENSION TABLE(100)
      .
      .
      DO 10 K=1,100
10    TABLE(K)=0.0
```

(2) The following sequence shows how to set all array elements of a three-dimensional array to zero.

```
      INTEGER SET(3,4,5)
      .
      .
      .
      DO 10 J = 1, 5
      DO 10 K = 1, 4
      DO 10 L = 1, 3
   10 SET(L, K, J) = 0
```

(3) The following example is an illustration of processing by DO statements.

```
      DIMENSION INTGRS(15)
      .
      .
      DO 10 K = 1, 15
   10 INTGRS(K) = K * K
      DO 20 K = 3, 15, 2
   20 INTGRS(K) = INTGRS(K - 2)
      .
      .
```

After the first DO (with terminal statement 10) has been satisfied, the array elements of INTGRS will contain, in succession, square numbers, 1, 4, 9, ..., 225. After the second DO has been satisfied, all odd-numbered array elements will contain the value 1; the even-numbered array elements are unchanged.

(4) The following internal numerical sort program contains an extended range. Originally the array elements of array NMBRS contain a series of integers in random sequence. The program is to re-arrange the integers from low to high, with the lowest in the first array element.

The sorting method used is known as a "bubble" sort. The first array element is compared with the second array element; if the first is greater than the second, the two are interchanged. Then the second array element is compared with the third array element, and so on, until the next-to-last array element is compared with the last array element. This is the initial pass. The result is that the greatest number has "bubbled" through to the last position. A record is kept of the last positions interchanged. This record determines if another sorting pass is required and how many array elements must be compared in that next sort.

```
      DIMENSION NMBRS(100)
C
C
      KOUNT = 1
      J = 99
40    DO 20 M = 1, J
      IF(NMBRS(M).GT.NMBRS(M+1)) GO TO 30
20    CONTINUE
      J = KOUNT - 1
      IF(J.EQ.0) STOP
      GO TO 40
30    KOUNT = M
      NSAVE = NMBRS(M)
      NMBRS(M) = NMBRS(M+1)
      NMBRS(M+1) = NSAVE
      GO TO 20
      END
```

Note the following:

- The terminal statement cannot be a logical IF containing a GO TO; therefore, a CONTINUE statement is added and used as the terminal statement. The CONTINUE statement (also known as a "no-op") does not perform any logical operation and can be used to satisfy format rules of FORTRAN.
- The extended range is considered part of the DO range being executed; therefore, parameters of the DO range can be referenced in the extended range.

(5) A DO statement can be used to advantage in cases where its use is not readily apparent. One such case is the evaluation of polynomials which have the form:

$$a_1x^n + a_2x^{n-1} + \ldots + a_nx + a_{n+1}$$

where each *a* and the value of *x* are known as execution time.

For polynomials where *n* is no greater than 3 or 4, an arithmetic assignment statement can be used, but as the value of *n* increases, execution time can be reduced with a DO statement.

Consider $a_1x^3 + a_2x^2 + a_3x + a_4$. By successive factoring of *x*, this can be represented as $x(x(x(a_1)+a_2)+a_3)+a_4$ which consists of three "multiply by *x* and add a constant" operations. With the DO statement, this can be evaluated by:

```
      Y = A(1)
      DO 10 I = 1, 3
   10 Y = X * Y + A(I+1)
```

For the general case, where *n* has the value N:

```
      Y = A(1)
      IF (N.EQ.0) GO TO 20
      DO 10 I = 1, N
   10 Y = X * Y + A(I+1)
   20 ....
```

5.5. CONTINUE STATEMENT

Function:

To act as a dummy executable statement. It is used primarily as the terminal statement of a DO range. It may be used wherever a dummy executable statement is required.

CONTINUE

Rules:

(1) The CONTINUE statement does not perform any logical operation.

(2) The sequence of statement execution is not changed by the CONTINUE statement.

Example:

The following sequence changes all negative values in array MAP to their corresponding positive values. Because the last logical operation is an arithmetic IF statement, a CONTINUE statement is added and made the terminal statement of the DO range.

```
      DIMENSION MAP(100)
      .
      .
      .
      DO 10 K=1,100
      IF(MAP(K)) 20,10,10
   20 MAP(K) = -MAP(K)
   10 CONTINUE
```

5.6. PROGRAM CONTROL STATEMENTS

A program control statement either temporarily halts execution of a program (the PAUSE statement) or terminates execution of a program (the STOP statement).

5.6.1. PAUSE Statement

Function:

To temporarily halt execution of a program.

PAUSE

or

PAUSE *n*

where: *n* is a string of one to five octal digits (the digits 0 through 7).

Rules:

(1) A PAUSE of either form temporarily halts execution of the program. The method of resuming execution differs with each processor and is described in the programming manual for that processor.

(2) The decision of resuming execution is not under program control, but is usually made by the operator under instructions from the programmer.

(3) With the second form of the PAUSE statement, the digit string is displayed or accessible by other means. Use of these digits depends upon the particular processor used.

(4) If execution is resumed without changing the state of the program, program control is passed to the next executable statement in normal sequence.

Examples:

The following sequence checks a list of account numbers in array LIST. If any of these numbers is not greater than zero, control is passed to an error routine and execution of the program is temporarily halted.

```
      DIMENSION LIST(20,30,40)
        .
        .
        .
      DO 10 M=1,40
      DO 10 L=1,30
      DO 10 K=1,20
      IF(LIST(K,L,M).LE.0) GO TO 20
   10 CONTINUE
   20   error routine
        .
        .
        .
      PAUSE
      GO TO 10
```

5.6.2. STOP Statement

Function:

To terminate execution of the program.

STOP

or

STOP *n*

where: *n* is a string of one to five octal digits (the digits 0 through 7).

Rules:

(1) There must be at least one STOP statement in a program to terminate execution of the program.

(2) Action that follows execution of the STOP statement depends upon the particular processor used.

(3) The digits of the second form of STOP statement are not necessarily accessible, depending upon the particular processor used.

Example:

The following program has read in a list of telephone numbers into array NUMBER. Each telephone number contains seven decimal digits (no area code). To ensure that all values have been read in correctly, each number is checked to see that it is greater than zero and not greater than 9999999. If any such error is detected, control is passed to an error routine (which may print a message) and execution of the program is terminated. The second STOP statement is assumed to be the STOP statement encountered during normal execution and is included to show that a program may have more than one STOP statement.

```
      DIMENSION NUMBER(1000)
      DO 10 K=1,1000
      IF(NUMBER(K).GT.0.AND.NUMBER(K).LE.9999999) GO TO 10
      error routine
      STOP
10    CONTINUE
      STOP
      END
```

6. INPUT/OUTPUT AND FORMAT STATEMENTS

6.1. GENERAL

Input statements fetch data from input and auxiliary storage devices to be used in the program. The input statement is the READ statement. Output statements store results obtained in the program on auxiliary storage devices or display the results on output devices. The output statement is the WRITE statement.

The sample program deck of cards of Figure 1–8 illustrates use of the READ and WRITE statements. The same program can be used without change for different data decks. All that need be changed in a particular application is the deck of data cards.

FORTRAN also gives the programmer some control over external devices with the BACKSPACE and REWIND statements for positioning such devices as magnetic tape, disc, and drum units. The ENDFILE statement can be used to demarcate files.

These statements apply to the transfer of *sequential files* to and from the processor. A *file* is the entire set of data on the I/O device designated in an I/O statement. A file may be subdivided into records. In an 80-column card reader, each record is an 80-column punched card; in a printer, each line is a record; in a magnetic tape unit, each record may have a different size (in characters) up to a maximum which is specified for each computer. Tape records are usually separated by a gap which contains no data. Each record may be subdivided into *fields*, the size of which is determined by the programmer.

The term *sequential file* is used as opposed to a random access or direct access file. For example, with standard I/O statements it is not possible to read the fifth record of a file directly; it is necessary to indicate that the preceding four records are to be passed over. Once the fifth record is read, it may be impossible (as in the case of a card reader) to go back and read the third record; however, this can be done where the BACKSPACE or REWIND statement is effective (as on magnetic tape).

READ and WRITE statements may refer to a FORMAT statement which describes the characteristics of the data being transferred. Such statements are called *formatted statements*. FORTRAN also provides for the transfer of information from one medium to another without change or conversion; these I/O statements are called *unformatted statements*. In addition, a READ or WRITE statement usually contains a list that identifies the items being transferred.

The remainder of this section describes:

- elements of I/O statements
- FORMAT statement
- formatted READ and WRITE statements
- unformatted READ and WRITE statements
- auxiliary I/O statements

6.2. ELEMENTS OF READ AND WRITE STATEMENTS

Each READ or WRITE statement may reference a FORMAT statement or specification and a logical unit, and may contain an I/O list.

6.2.1. Logical Unit Number

The logical unit number is an unsigned integer that designates the I/O device containing the file being referenced. A file may be transferred from one medium to another. If it becomes necessary to access the same file later on, it will have a different logical unit number than the one originally used. In previous examples, the integers 1 and 3 were used to denote a punched card reader and a printer, but these numbers were only for use in examples. There is no standard convention for assigning numbers to logical units. This information must be obtained from programming manuals for a particular processor.

6.2.2. Input/Output List

The purpose of an *input/output list* is to identify transferred items so that they can be referenced in the program. A transfer initiated by a READ or WRITE statement is not complete unless all items in the input/output have been transferred. It is convenient to define an input/output list in terms of a *simple list* and a *DO-implied list.*

A *simple list* is a variable, array element, array name, or two simple lists separated by a comma. For example,

V2,ARRAY,MATRIX(5)

is a simple list. Previous examples were restricted to simple lists in READ and WRITE statements. When an array name appears in a simple list, it refers to all elements of that array in the order described in 2.7.4.

An *I/O list* is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by a comma. When there is no I/O list in a READ or WRITE statement, the I/O list is said to be empty. For example,

V2,(ARRAY,MATRIX(5),(NAME))

is an I/O list. No I/O list may contain a constant except in a subscript expression or as a parameter of a DO-implied list.

A *DO-implied list* is a list followed by a comma and a *DO-implied specification,* all enclosed in parentheses. A DO-implied specification has the format:

$$i = m_1,m_2,m_3$$

or

$$i = m_1,m_2$$

The italicized parameters and the control variable are the same as those for the DO statement (see 5.4). For example, the DO-implied list (ARRAY(K), K=3,5) refers to the array elements ARRAY(3), ARRAY(4), and ARRAY(5), in that order. This illustrates one of the advantages of a DO-implied list. It enables selected array elements to be referenced without the use of a DO statement. The DO-implied list (MATRIX(J), ARRAY(J), J=1,3) refers to the array elements MATRIX(1), ARRAY(1), MATRIX(2), ARRAY(2), MATRIX(3), and ARRAY(3), in that order.

Examples of DO-implied lists:

(1) ((ARRAY(J,K), J=1,5,2), K=3,4)

refers to ARRAY(1,3), ARRAY(3,3), ARRAY(5,3), ARRAY(1,4), ARRAY(3,4), and ARRAY(5,4), in that order.

(2) (ARRAY(J,K), J=1,3)

refers to ARRAY(1,K), ARRAY(2,K), and ARRAY(3,K), in that order, where K was defined previous to execution of the READ or WRITE statement containing the DO-implied list.

(3) (((ARRAY(J,K,M), J=1,J2), K=1,K2), M=1,M2)

refers to the array elements of ARRAY in their natural order if J2, K2, and M2 were the declared dimensions.

(4) An example of an I/O list containing a DO-implied list is:

A,B, (C,ARRAY(K), K=3,5)

refers to A,B,C,ARRAY(3),C,ARRAY(4),C, and ARRAY(5), in that order.

(5) The elements of an array can be referenced in any order. For example,

((ARRAY(J,K), K=1,L), J=1,M)

interchanges the order of subscripts of ARRAY.

(6) An example of a DO-implied list within a DO-implied list is:

((A(I,J), I=1,10,2), B(J,3), J=1,K)

The order of reference is clearer if shown as follows:

```
DO   J=1,K
 DO  I=1,10,2
  A(I,J)
 B(J,3)
```

Thus, the order of reference is:

A(1,1),A(3,1),A(5,1),A(7,1),A(9,1),B(1,3),
A(1,2),A(3,2),A(5,2),A(7,2),A(9,2),B(2,3),
.
.
.
A(1,K),A(3,K),A(5,K),A(7,K),A(9,K),B(K,3)

6.3. FORMAT STATEMENT

Function:

To provide conversion and/or editing information between the internal representation and the external character strings in conjunction with a formatted READ and/or WRITE statement.

Format:

$(q_1 t_1 z_1 t_2 z_2 \cdots t_{n-1} z_{n-1} t_n q_2)$

where: each q is a series of one or more slashes (/) to act as a record demarcator and may be omitted.
each t is a field descriptor or a group of field descriptors.
each z is a field separator.
The part $(q_1 t_1 z_1 \cdots q_2)$ is called the *format specification.*

Rules:

(1) A FORMAT statement must have a label. It can be referenced by one or more formatted READ and/or WRITE statements.

(2) If a formatted READ or WRITE statement references a format specification by array name, only the format specification (including the enclosing parentheses) must be the first item in the array. Any information in the array after the rightmost parenthesis is ignored. For example,

```
      READ (1,10) K2, K3, K4, K5, K6, K7
   10 FORMAT (315, 318)
```

indicates that K2, K3, and K4 are to be interpreted as integers occupying five character positions, and that K5, K6, and K7 are to be interpreted as integers occupying eight character positions. The READ statement could also have been written as:

```
      DIMENSION INT(2)
      DATA INT(1)/5H(315,/,INT(2)/4H318)/
      .
      .
      .
      READ (1,INT) K2, K3, K4, K5, K6, K7
```

where the DATA statement initializes the contents of array INT to the required format specification, using Hollerith constants.

The only restriction on the use of a format specification in an array is that the format specification cannot contain a Hollerith field descriptor of the form *n*H.

(3) Blank characters may be used freely in the FORMAT statement or a format specification, except with a Hollerith field descriptor.

(4) If there is an I/O list in the formatted READ or WRITE statement, the format specification must contain at least one field descriptor other than *n*H or *n*X.

(5) Format control (control by a FORMAT statement or specification) is initiated when execution of a formatted READ or WRITE is started. (Further details on format control are furnished in 6.6.)

(6) The first character of a record to be printed is not printed; it is used as a form control character as shown in Table 6–1.

CHARACTER	VERTICAL SPACING BEFORE PRINTING
BLANK	ONE LINE
0	TWO LINES
1	TO FIRST LINE OF NEXT FORM
+	NO ADVANCE

Table 6–1. Form Control Characters

There are many ways of assuring that the required form control character is the first character. The safest way is to use 1H␣ or 1X for the blank character and 1H0, 1H1, 1H+, respectively, for the form control characters in Table 6–1. However the sequence

```
      K = 1
      WRITE(3,20) K
   20 FORMAT(12,...)
```

will have the same effect as if the FORMAT statement had been written

```
   20 FORMAT(1X,11,...)
```

In both cases, the printer will advance to the start of the next line and print the digit 1 in the first print position of the line.

However, if the FORMAT statement had been written :

```
      20   FORMAT(1H1,.....)
```

the printer would have advanced to the first line of the next form before printing the remaining items of the I/O list.

(7) A format specification without any field descriptors is valid. For example, the format specification (/////) is valid. It causes the printer to advance five lines.

6.3.1. Record Demarcator

The term *record demarcator* refers to the one or more consecutive slashes (/) that appear anywhere in a format specification. If there are *n* slashes at the beginning or at the end of the specification, *n* records will be skipped; if there are *n* slashes anywhere else, processing of the current record is terminated, and *n*−1 records are skipped.

For example, in reading punched cards, the sequence

```
      READ(1,10),K2,K3
   10 FORMAT(15/15)
```

will obtain the value of K2 from the current card and the value of K3 from the next card. The sequence

```
      READ(1,10),K2,K3
   10 FORMAT(//15//15//)
```

causes skipping of two punched cards before obtaining the value of K2 from the third card; skipping the fourth card; obtaining the value of K3 from the fifth card; then skipping two cards for processing of the next READ statement (if any) for the same file.

6.3.2. Field Separators

A *field separator* is either a comma or a series of one or more consecutive slashes. It is used to separate field descriptors in the list of format specifications. The slash(es) also acts as a record demarcator and ends processing of the current record.

For example, in printing information, the sequence

```
      WRITE(3,10),K2,K3,K4
   10 FORMAT(1X,15,15/1X,15)
```

will cause printing of the values K2 and K3 on one line, and the value of K4 on the next line.

6.3.3. Field Descriptors

Function:

To indicate how items from/to an input/output device shall be represented internally/externally and provide editing information in the form of spacing, form control (printer only), and Hollerith data.

*x*P*r*F*w*.*d*
*x*P*r*E*w*.*d*
*x*P*r*G*w*.*d*
*x*P*r*D*w*.*d*
*r*I*w*
*r*L*w*
*r*A*w*
*n*H$h_1h_2 \ldots h_n$
*n*X

where: the letters F, E, G, D, I, L, A, and H indicate the conversion and editing and are called *conversion codes*.

w and *n* are unsigned integers greater than zero indicating the number of character positions in a field of the external medium.

d is an unsigned integer constant indicating the number of digits in the fractional part of the external character string (except for the G conversion code).

r is an optional nonzero unsigned integer, the *repeat count*, indicating how many times to repeat a basic field descriptor that follows it.

*x*P (optional) indicates scaling, *x* being an unsigned integer representing the scale factor.

h represents a character from the processor character set.

Rules:

(1) The field width must be specified for all descriptors.

(2) For descriptors of the form *w.d*, *d* must be specified even if it is 0, and *w* must be greater than or equal to *d*.

(3) All output is right-justified in the output field width specified by *w*, preceded by leading blank characters (*w* permitting).

(4) The number of characters produced on output cannot exceed the field width, *w*.

6.3.3.1. Blank Field Descriptor

The field descriptor for blank characters is *n*X. On input, *n* characters of the input record are skipped; on output *n* blank characters are inserted in the output record, except for the carriage control character.

6.3.3.2. Numeric Data

There are five conversion codes for handling numeric data:

- the *r*I*w* code for handling integer type data
- the *x*P*r*F*w.d* code
- the *x*P*r*E*w.d* code
- the *x*P*r*G*w.d* code

  } for real type and complex type data

- the *x*P*r*D*w.d* code for handling double precision type data

On all numeric conversions, a blank character in the specified field is treated as a 0. A blank field is treated as the integer zero. Depending upon the processor and the field width specified, positive values may be preceded by a + or no character position; a negative value requires a character position in the field width for the minus sign. Leading blank or zero characters are not significant.

6.3.3.2.1. Integer Type Conversion

The basic field descriptor I*w* indicates that the external field occupies *w* positions as an optionally signed integer and is represented internally as an integer type item.

On input, the external field may contain only a sign followed by digits, with blank characters anywhere in the string. No other characters are permissible. If the sign is plus, it may be omitted; if the sign is minus, it is required. On output, the external field consists of blank characters (if necessary) and a minus sign or an optional plus sign followed by the magnitude of the internal integer type value.

Examples:

(1) An input card contains

Position 1 8

b̶b̶b̶1̶b̶2

and the following statements are used:

```
      READ(1,10) NAME2,NAME4
10    FORMAT(I5,I3)
```

The first five columns will be read to obtain the value for NAME2, which is 1. Then the next three columns will be read to obtain the value for NAME4, which is 2.

If the following statements are used instead:

```
      READ(1,10) NAME2,NAME4,NAME6
10    FORMAT (I4,I2,I3)
```

the first four columns will define the value of NAME2 as 0; the next two columns will define the value of NAME6, as 10; the next three columns will define the value of NAME6 as 20. Note that trailing blank characters are significant and treated as zeros.

If the following statements are used instead:

```
      READ(1,10) NAME7
10    FORMAT(I9)
```

the first nine columns will define the value of NAME7 as 10020.

(2) A punched card contains

```
Position  1        1         2         3         4         8
          0        0         0         0         0         0
          ␣␣␣␣1234␣␣␣ -1234␣-␣␣ 1234␣␣␣ -123␣␣␣+␣1␣2␣34␣␣␣ ... ␣
```

and the following statements are used:

```
      DIMENSION IARAY(2)
      .
      .
      READ (1,10) K1,IARAY,K2,K3,K4
10    FORMAT (3X,I5,3X,I6,I6,I5,I10,I5)
```

The first three columns are skipped;
the next five columns read in the value for K1, which is 1234;
the next three columns are skipped;
the next six columns define the value of IARAY(1), which is -12340;
the next eight columns define the value of IARAY(2), which is -12340;
the next five columns define the value of K2, which is -123;
the next 10 columns define the value of K3, which is 102034;
the next five columns define the value of K4, which is 0.
The list is now satisfied and any remaining items on the card are ignored.

(3) After the following sequence is executed,

```
      NAME2 = +10
      NAME4 = -10
      WRITE(3,1)NAME2,NAME4
    1 FORMAT(3X,I3)
```

the printed line is:

10bbb-10 starting in print position 1.

If the output list items in the WRITE statement had been

```
      WRITE(3,1)NAME4,NAME2
```

an error condition would result. The first print character is a minus character, which is not a legitimate form control character. If this minus sign is interpreted as a form control character (in some processor implementations), there is not enough field width provided for printing -10. In this case the FORMAT statement requires a change such as:

```
    1 FORMAT(3X,I4)
```

to ensure that NAME4 will be printed correctly.

6.3.3.2.2. Input of Real Type Data

For input numerical data to be represented internally as real type data, three conversion codes are available: the F, E, and G codes. Operation of these three basic field descriptors is identical for input data (their operation differs on output data).

The *basic form* of the external input field consists of an optional sign (optionally preceded by blank characters) followed by a string of digits and blank characters with one optional decimal point anywhere in the string. This basic form may be followed by an exponent, having any of the following forms:

- a signed integer constant
- the letter E followed by an optionally signed integer constant
- the letter D followed by an optionally signed integer constant

A decimal point in the basic form overrides the decimal point specified by the *d* designator.

Example:

A punched card contains

	1	2	3	4	5	6
Position 1	0	0	0	0	0	0
	3456789	345.6789	3456789+3	-67.89E-3	-6789D10	-67.89D-2

and the following statements are used:

```
      READ(1,10) A1,A2,A3,A4,A5,A6
   10 FORMAT(F10.0,E10.0,F10.3,G10.3,E10.2,2X,G8.1)
```

A1 is defined as 3456789.0;
A2 as 345.6789;
A3 as 3456789.0;
A4 as -0.06789;
A5 as -678900000000.0;
A6 as 0.6789 (note that the minus sign is lost).

6.3.3.2.3. Output of Real Type Data

On output of numerical data that is internally represented as real type data, three conversion codes are available: the F, E, and G codes. The output form is different for the F and E codes; the G code results in form similar to either the F or E code.

On output, the F*w.d* results in an output field consisting of blanks (if necessary), a minus sign or optional plus sign (depending upon the processor), followed by a string of digits containing a decimal point and a fractional part rounded to *d* fractional digits.

On output, the E*w.d* results in an output field of the form:

$$t0.x_1 \ldots x_d E \pm y_1 y_2$$

or

$$t0.x_1 \ldots x_d \pm y_1 y_2 y_3$$

where: the choice of form depends upon the processor.

$x_1 \ldots x_d$ are the *d* most significant rounded digits of the value to be transmitted.

y represents a digit of the decimal exponent.

the plus sign following the letter E may be represented by a blank character (depending upon the processor);

t is either no character position or a minus sign; the 0 may be replaced by no character position (depending upon the processor).

On output with the G*w.d* basic field descriptor, the form of the external field depends upon the absolute value of the internal real type value. If N is the absolute magnitude, the effect conversion is as follows:

MAGNITUDE OF N	EFFECTIVE CONVERSION
$0.1 \leq N < 1$	F(*w*-4) .*d*,4X
$1 \leq N < 10$	F(*w*-4) .(*d*-1) , 4X
.	.
.	.
.	.
$10^{d-2} \leq N < 10^{d-1}$	F(*w*-4) . 1, 4X
$10^{d-1} \leq N \leq 10^{d}$	F(*w*-4) . 0, 4X
otherwise	E*w.d*

For effective F conversion, the absolute value of the real type item in storage must be equal to or greater than 0.1 and equal to or less than 10^d.

Rule:

In general, on output with the E*w.d*, and possibly the G*w.d*, basic field descriptors, *w* should provide:

- four positions for the decimal exponent;
- one position for the decimal point;
- one position for the 0;
- one position for the sign.

Therefore, this general rule pertains to these basic field descriptors: *w* must be at least 7 greater than *d* on output.

Examples:

(1) The successive elements of array table are

.001234
.001234
−12.345
−12.345

and the following statements are used for printing:

```
      WRITE(3,10)TABLE
10    FORMAT(1X,F15.7,E15.7,F15.2,E15.7)
```

The printed line will be (subject to processor options):

```
Position  1               3               4               6
          5               0               5               0
          ␣␣␣␣␣␣␣.0012340␣␣0.1234000E-02␣␣␣␣␣␣␣␣␣␣-12.35␣-0.1234500E+02
```

(2) Examples for the basic field descriptor G*w.d* on output for different values of the transmitted item are:

ITEM VALUE	CODE DESCRIPTOR	EFFECTIVE DESCRIPTOR	PRINTED ITEM
−.0123456	G10.3	E10.3	-0.123E-01
-23.456789	G11.2	F7.0,4X	␣␣␣-23.␣␣␣␣
-23.456789	G11.3	F7.1,4X	␣␣-23.5␣␣␣␣
-23.456789	G11.5	F7.3,4X	-23.457␣␣␣␣
−123.456780	G11.2	E11.2	␣␣-0.12E+03

6.3.3.2.4. Double Precision Type Conversion

On input, the basic field descriptor D*w.d* is used for items to be represented internally as double precision type data; on output, it is used for items that are represented internally as double precision type data.

The form of the external input field is the same as that for real type conversion (see 6.3.3.2.2). The form of the external output field is the same as that for the E*w.d* field descriptor (see 6.3.3.2.3) except that the letter D may replace the letter E. As with real type numbers on input, a decimal point in the input value overrides the *d* specification.

The advantage of the D descriptor is that it can store and output more significant digits than the E descriptor can.

Rule:

The internal field must be explicitly declared as double precision type.

Example:

The following short program illustrates use of the D conversion code:

```
      DOUBLE PRECISION X,X2
      READ(1,10) X
   10 FORMAT(D15.5)
      X2 = 2.0 * X
      WRITE(3,20) X2
   20 FORMAT(D17.5)
      STOP
      END
```

6.3.3.2.5. Complex Type Conversion

A complex data type item is represented internally as two consecutive real type items. Therefore, for each of the pair (the first is the real part and the second is the imaginary part), a real type conversion code is required.

Requirements for the external input field are the same as for real type conversion. Each successive pair of items is interpreted as a complex type item.

The external output field requires a separate real type conversion code for each part of the complex value.

Rule:

The internal field requires explicit type declaration.

Examples:

(1) Input

A punched card contains

Position	1	2	3	4	5
1	0	0	0	0	0
	-123.56b	-123-3	45.6-02	123.-02	0.0123bb

and the following statements are used:

```
      COMPLEX CPX1,CPX2
      .
      .
      .
      READ (1,10) CPX1,A,CPX2
10    FORMAT(F10.0,F10.1,F10.1,E10.0,G10.0)
```

$-123.56 - 0.0123i$ is read into CPX1;

0.456 into A;

$1.23 + 0.0123i$ into CPX2.

Since the input list is satisfied, the last G10.0 of the FORMAT statement is disregarded.

(2) Output

After execution of the following program

```
      COMPLEX CPX2, CPX4
      CPX2 = (-123.56, -345E-3)
      CPX4 = (789.56, -890.1)
      A = -.456
      WRITE (3,5) CPX2, A, CPX4
    5 FORMAT (E13.5,2X,E12.5,5X,F5.3/F8.2,2X,F6.1)
      STOP
      END
```

The first print line will be:
−0.12356E+03~~bb~~−0.34500E+03~~bbbbb~~−.456

The second print line will be:
~~b~~789.56~~bb~~−890.1

Note that the first character of each print line (a blank character) is interpreted as the form control character and is not printed.

6.3.3.3. Logical Type Conversion

On input, the basic field descriptor L*w* is used for items to be represented internally as logical type data; on output, for items that are represented internally as logical type data.

Rules:

(1) The internal field must be explicitly declared as logical type.

(2) On input, the external field consists of optional blank characters, followed by either a T for true, or F for false, followed by optional characters which are ignored.

(3) On output, the external field consists of *w*- 1 blank characters followed by a T or an F.

Example:

An input card contains

```
                     1          1
Position   1         0          9
                  TON    FOFF
```

and the program contains the sequence:

```
      LOGICAL SWTCH2,SWTCH4
      READ(1,5) SWTCH2,SWTCH4
    5 FORMAT(L10,L9)
      WRITE(3,5) SWTCH2,SWTCH4
```

The printed line will be:

bbbbbbbTbbbbbbbbF

6.3.3.4. Hollerith Field Descriptors

Hollerith information may be transmitted by the field descriptors *n*H and A*w*, as follows:

(1) On input, the *n*H reads Hollerith data into the *n* characters following the *n*H field descriptor in the format specification. On output, it writes the last previously defined *n* characters of the format specification following the *n*H field descriptor.

(2) The A*w* field descriptor reads or writes *w* Hollerith characters into or from an element of the I/O list.

Rules:

(1) The *n*H field descriptor must not be used in a format specification if a READ or WRITE statement references the format specification by array name.

(2) For the A*w* field descriptor, let *g* be the maximum number of characters that can be represented in a single storage unit (see 2.5.1). If *w* is greater than or equal to *g* on input, the rightmost *g* characters will be accepted and any remaining leftmost characters will be lost. If *w* is less than *g* on input, the *w* characters will be accepted left-justified internally with remaining storage positions filled with blank characters. If *w* is greater than *g* on output, the external field will contain the *g* characters right-justified with the remainder of the field filled with blank characters. If *w* is less than or equal to *g*, the external field will contain only the leftmost *w* characters from the internal representation.

Examples:

For explanatory purposes, it has been assumed that no more than five characters can be stored internally in an integer, real, or logical data type item. The number of characters that can be stored in a single storage unit varies with processor implementation; there is not standard capacity.

(1) In this program, the *n*H field descriptor causes printing of specified characters.

```
      K=1
      J=2
      MSUM=J+K
      WRITE(3,10) MSUM
   10 FORMAT(8H TOTAL =,I2)
      STOP
      END
```

The printed line will be:
TOTAL = 3

(2) Consider the following short program which reads in one data card containing:

```
Position  1        1         2         3         4         5
          0        0         0         0         0
          CURRENT  1.2       56.78     2.3       34.56
```

```
      DIMENSION RSTR(10),VOLTS(10),AMPS(10)
      READ(1,10) (RSTR(K),VOLTS(K),K=1,2)
   10 FORMAT(10H CURRENT ,F10.0,F10.0,F10.0,F10.0)
      DO 20 K=1,2
   20 AMPS(K)=VOLTS(K)/RSTR(K)
      WRITE(3,10) (AMPS(J),J=1,2)
      STOP
      END
```

The printed output is:

CURRENT␣␣␣␣␣␣47.32␣␣␣␣15.03

The READ and WRITE statements are processed as:

(a) After the card was read in, the FORMAT statement became, in effect, 10 FORMAT(10H␣CURRENT␣␣,F10.0,F10.0,F10.0,F10.0). On printing, the first ␣ was interpreted as a carriage control character and was not printed.

(b) It was not necessary to define all elements of the arrays since these were not used.

(c) Although the WRITE statement required only two I/O list items, its FORMAT statement contained four field descriptors. As soon as an I/O list is satisfied (all items are accounted for), format control is terminated.

The next example shows how the coding of repeated field descriptors can be simplified.

(3) The program of the previous example is simplified and made more I/O oriented by:

(a) reading in the format specification;

(b) reading in the number of array elements to be processed in the DO-implied lists;

(c) simplifying coding of repeated field descriptors;

(d) reading in the numbers for input/output devices.

The first input record from the card reader is organized as follows:

(a) In columns 1 through 8, (8F10.0) is used as the format specification for input data for arrays RSTR and VOLTS. This specification is named INFORM.

(b) In columns 11 through 31 (A1,15X,A5,A2/(1X,F10.0)) is used as the output format specification, OUTFRM.

(c) In columns 39,40, a two-digit unsigned integer denotes NIN, the input device for data for arrays RSTR and VOLTS.

(d) In columns 44,45, a two-digit unsigned integer denotes NOUT, the output device for the computed data in array AMPS.

(e) In columns 49,50 a two-digit unsigned integer (not greater than 10) indicates NMBR, the number of array elements required in each of the arrays.

(f) In columns 51 through 57, CURRENT denotes output heading.

(Statements are continued for illustrative purposes only.)

```
1         DIMENSION RSTR(10),,VOLTS(10),,AMPS(10),,
2        1 INFORM(2),,OUTFRM(5),,TITLE(2)
3         DATA CNTRL/1H0/
4         READ(1,,10),,INFORM,,OUTFRM,,NIN,,NOUT,
5        1 NMBR,,TITLE
6      10 FORMAT(2A5,,5A5,,3(3X,I2)),,A5,,A2)
7         READ(NIN,,INFORM),,(RSTR(K)),,VOLTS(K)
8        1 K=1,,NMBR)
9         DO 20 M=1,,NMBR
10     20 AMPS(M)=VOLTS(M)/RSTR(M)
11        WRITE(NOUT,,OUTFRM),,CNTRL,,TITLE,,
12       1 (AMPS(J)),,J=1,,NMBR)
13        STOP
14        END
```

Up to three of the remaining input records can contain, as before, the input values for voltages and resistors (RSTR and VOLTS) in fields of 10 positions per value. The input device for these records is designated by the programmer.

The output contains a header and the calculated output values (AMPS), one value to a record. The output device for these records is also designated by the programmer.

The repeat specifications and the use of seemingly redundant parentheses within a format specification are described in 6.3.3.5 and 6.6, but their particular applications in this program are described in the following paragraphs.

In lines 4 and 5, the I/O list requires12 items: 2 for INFORM, 5 for OUTFRM, 1 each for NIN, NOUT, and NMBR, and 2 for TITLE.

In line 6, the two items required for INFORM are covered by 2A5, which could also be written as A5,A5; and the five items required for OUTFRM are covered by 5A5. However, a *group repeat designator* is used in this line. This indicates successive repetition of the group within the parentheses. The group repeat designator 3(3X,I2) could also be written as 3X,I2,3X,I2,3X,I2.

Line 7 refers to the format specification in INFORM, which is (8F10.0). The 8 means that there can be 8 fields to a record with the basic descriptor F10.0. However, as many as 20 values may be required: 10 for RSTR and 10 for VOLTS. If the DO-list indicates that more than 8 fields are required, the first 8 fields will be read in, a new record automatically started, and the same specification (8F10.0) used, and so on, until all values required for the I/O list are read in.

The WRITE statement refers to the format specification in OUTFRM, which is (A1, 15X,A5,A2/(1X,F10.0)). The A1 refers to the carriage control character called CNTRL, that was initialized to 1H0. It is not permitted to use an *n*H field descriptor directly in a format specification that is stored in an array; therefore, this indirect method must be used. (An alternative method would be to read it in.) The slash in this format specification indicates the end of a record. The field descriptors that follow apply to the next record which permits only one F10.0 field descriptor to a record, applying to the output results in AMPS. If AMPS contains more than one value (indicated by NMBR), it will automatically start a new record and use the 1X,F10.0 for each new value and record. If the 1X,F10.0 was not enclosed within its own parentheses, format control would have been started at A1, instead of after the slash each time a new record was required.

6.3.3.5. Repeat Specifications

Repetition of a field descriptor (except *n*H and *n*X) in a format specification is accomplished by using the repeat count *r* (see 6.3.3) immediately before the basic field descriptor. For example:

5L3 is the same as L3,L3,L3,L3,L3.

Repetition of a group of one or more field descriptors and/or field separators is indicated by enclosing the group in parentheses and optionally preceding the group with an integer constant. If no group repeat count is indicated, its value is implicitly 1. This form, with or without a group repeat count, is called a *basic group*. For example:

2(2X,2I5,F10.0) is the same as 2X,I5,I5,F10.0,2X,I5,I5,F10.0.

2(//) is the same as ////.

2(4HHALT) is the same as 4HHALT,4HHALT.

A further grouping may be formed by enclosing field descriptors, field separators, and/or basic groups within parentheses. Again, a group repeat count may be either specified or implicitly 1. The parentheses around this further grouping are called second level parentheses. Parentheses are permitted in a FORMAT statement only to a second level; therefore, this further grouping may not be contained in another grouping by parentheses.

The first left parenthesis and the last right parenthesis required of a format specification are not considered group delimiters.

Example:

```
      READ(IN,5) (M(K),K=1,2),N2,(A(J),B(J),C(J),J=1,2),N
    5 FORMAT(3I2,2(3X,2(2F6.3,2X,F6.4)),4X,I2)
```

The elements of the I/O list are read in as:

M(1)	with specification I2
M(2)	with specification I2
N2	with specification I2
skip three character positions	
A(1)	with specification F6.3
B(1)	with specification F6.3
skip two character positions	
C(1)	with specification F6.4
A(2)	with specification F6.3
B(2)	with specification F6.3
skip two character positions	
C(2)	with specification F6.4
skip four character positions	
N	with specification I2

6.3.3.6. Scale Factor

Input and output using the F, E, G, and D conversion codes can be scaled up or down by a power of 10 if the conversion code has the form xP immediately preceding a repeat specification (if any), where x is an integer constant that may be preceded by a minus sign. The effect of *x*, the *scale factor*, is to multiply the corresponding I/O list item by a power of 10.

Rules:

(1) When format control is initiated (see 6.6), a scale factor of zero is established. Once a scale factor has been established, it applies to all subsequently interpreted F, E, G, and D field descriptors until another scale factor is encountered and then established.

(2) The effect of the scale factor is temporarily suspended for input with F, E, G, and D conversions that contain an exponent in the external field.

(3) For input with F, E, G, and D conversion and no exponent in the external field, the internally represented value is the external value divided by 10^x.

(4) For output with F conversion, the external value is the internal value multiplied by 10^x.

(5) For output with E or D conversion, the basic constant part of the output is multiplied by 10^x and the exponent is decreased by *x*.

(6) If the effective use of E conversion is required for output with G conversion (see 6.3.3.2.3), the scale factor has the same effect as for E conversion. If effective use of the F conversion code is required, the scale factor has no effect.

Example:

Two identical punched cards have all values right-justified in their fields of 10 positions each, as follows:

columns 1 through 10:	100.21
columns 11 through 20:	100.21
columns 21 through 30:	100.21D-3
columns 31 through 40:	100.21
columns 41 through 50:	567890
columns 51 through 60:	12345E-2
columns 61 through 70:	1234.5
columns 71 through 80:	654321.321

with the following statements:

```
      DIMENSION A(7),B(7)
      DOUBLE PRECISION A8,B8
      .
      .
      .
      READ(1,10) A,A8,B,B8
   10 FORMAT(F10.0,2P2E10.0,2G10.0,0P2F10.5,2PD10.4)
```

A(1) is read in as 100.21 because the scale factor is established as 0 (unless stated otherwise) at the start.
A(2) is read in as 1.0021 because the scale factor of 2 is effective.
A(3) is read in as 0.10021 because the scale factor of 2 is temporarily suspended by the exponent in the external field.
A(4) is read in as 1.0021 because the scale factor of 2 is still established and effective.
A(5) is read in as 5678.9 because the scale factor of 2 is still established and effective.
A(6) is read in as 0.0012345 with the scale factor now established as 0.
A(7) is read in as 1234.5 because the scale factor is still established as 0.
A8 is read in as 6543.21321 because a scale factor of 2 is now established and effective.
Because a new record start is required by the list, the next card is read, but there has been no termination of format control.
B(1) is read in as 1.0021 because the scale factor of 2 is still established and effective.
B(2), . . . ,B(7), and B8 are read in with the same values as the corresponding A's.

6.4. FORMATTED READ STATEMENT

Function:

To initiate input of data from a specified input device and to scan and interpret this data in accordance with a format specification.

READ *(u,f) I/O list*

or

READ *(u,f)*

where: *u* is either an integer constant or an integer variable that identifies an input unit.
f is either the statement label of a FORMAT statement or the name of an array that contains the format specification.
I/O list is described in 6.2.2.

Rules:

(1) The number of records to be read depends upon the I/O list and the format specification.

(2) Format control is started and terminated in accordance with the rules of 6.6.

(3) There are no standard conventions for assignment of integers to input devices; the integer for a particular input device depends upon the computer being used.

Examples:

(1)

```
      READ (10) A,(ARRAY(K), K=1,7)
   10 FORMAT (8F10.0)
```

Eight values are read in: one value for A, and seven values for the elements of ARRAY.

(2)

```
      READ(INPUT,10) A,(ARRAY(K), K=1,7),
     1 B,(BRRAY(K), K=1,7)
   10 FORMAT(8F10.0)
```

Regardless of the input device represented by INPUT, one value is read in for A, then seven values for ARRAY. A new record is automatically started, one value is read in for B, followed by seven values for BRRAY.

(3)

```
      READ (INPUT,10)
   10 FORMAT (10HHEADING###)
```

The first 10 characters of the record from INPUT replace HEADING### in the format specification; the same FORMAT statement can then possibly be used in a WRITE statement to supply the header.

(4)

```
      DIMENSION ARRAY(1)
      DATA ARRAY(1)/4H(/,/)/
      .
      .
      .
      READ(IN,ARRAY)
```

The format specification for the READ statement is obtained from the array named ARRAY. The effect of the READ statement is to advance the input device by two records.

6.5. FORMATTED WRITE STATEMENT

Function:

To indicate the output of data to a specified output device in accordance with a format specification.

WRITE *(u,f) I/O list*

or

WRITE *(u,f)*

where: *u* is either an integer constant or an integer variable that identifies an input unit.
f is either the statement label of a FORMAT statement or the name of an array that contains the format specification.
I/O list is described in 6.2.2.

Rules:

(1) The number of records to be transmitted depends upon the I/O list and the format specification.

(2) Format control is started and terminated in accordance with the rules of 6.6.

(3) The first character of a print record is not printed; it is interpreted as a form control character (see Table 6–1). In standard FORTRAN the printed line may start in position 1; however, many printers indicate the presence of the control character by printing a blank character, so that printing may start only with position 2 of a print line. This characteristic should be checked for a particular processor.

(4) There are no standard conventions for assignment of integers to particular output devices; this depends upon the processor being used. (In examples, 3 is used to indicate a printer.)

Examples:

(1)

```
      WRITE(3,10)
   10 FORMAT(1H1,15X,15HNAME OF PROBLEM)
```

The printer starts at the first line of the next form, spaces 15 positions, and prints NAME OF PROBLEM.

(2)

```
      NOUT=6
      WRITE(NOUT,10) A,B,C
   10 FORMAT(1H1,15X,15HNAME OF PROBLEM/16X,3F10.4)
```

The output device, if magnetic tape (as an example), will contain the character 1, followed by 15 blank characters, followed by the characters NAME OF PROBLEM, all in the first record. The second record will contain 16 blank characters followed by the values of A, B, and C, each in accordance with the F10.4 conversion code. These records may be sent to a printer later on.

6.6. FORMAT CONTROL

The following rules describe the relation between format control and the I/O list (if any) of the formatted READ and WRITE statements.

(1) Format control is initiated with the start of each execution of a formatted READ or WRITE statement.

(2) When the format control encounters an I, F, E, G, D, A, or L basic descriptor in a format specification, it determines if there is a corresponding element in the I/O list. If there is no such element, format control is terminated; if there is a corresponding element, it is converted and transmitted, and format control proceeds.

(3) If format control proceeds to the outermost right parenthesis of the format specification, it determines whether another I/O list item is to be transmitted. If not, format control is terminated. If another list item is to be transmitted, format control starts a new record and control is transferred to that group repeat specification (which may be an implicit 1) terminated by the last preceding right parenthesis or, if none exists, to the first left parenthesis of the format specification. This action does not affect the scale factor that has been established.

For example, if there are list elements to be transmitted, a new record is started and format control continues at

$(\ldots x(\ldots)\ldots \underset{\downarrow}{x}(\ldots x(\ldots)\ldots)\ldots)$

where: x is a group repeat specification (which may be an implicit 1).
If the format specification does not contain any inner group parentheses, format control continues at

$(\underset{\downarrow}{\ldots})$

(4) There is no corresponding I/O list element for an X or H basic descriptor. An I/O list requires at least one field descriptor other than nX or nH.

Examples:

(1)

```
      |READ(INPUT,10)(ARAY2(K),K=1,8)
   10 |FORMAT(8F10.5)
```

The eight values of ARAY2 are obtained from one record.

(2)

```
      READ(INPUT,10)(ARAY2(K),K=1,8)
   10 FORMAT(F10.5)
```

The eight values of ARAY2 are obtained from eight successive records.

(3)

```
      READ(INPUT,10)(ARAY2(K),K=1,8)
   10 FORMAT(2F10.5/3F10.5)
```

The first two values for ARAY2 are obtained from the first record; the next three values from the second record; the next two values from the third record; and the last value from the fourth record.

(4)

```
      READ(INPUT,10)(ARAY2(K),K=1,8)
   10 FORMAT(2F10.5/(3F10.5))
```

The first two values for ARAY2 are obtained from the first record; the next three values from the second record; and the last three values from the third record.

6.7. UNFORMATTED WRITE AND READ STATEMENTS

Function:

To transmit the exact binary configuration of I/O list elements to and from an external device.

WRITE *(u) I/O list*

and

READ *(u) I/O list*

or

READ *(u)*

where: *u* is an unsigned integer constant or an integer variable designating an input or output device.

I/O list is described in 6.2.2.

Rules:

(1) An unformatted WRITE statement transmits the exact binary internal representation to an external device. An I/O list is required with an unformatted WRITE.

(2) An unformatted READ (of either form) can only be performed for records created by an unformatted WRITE.

(3) The unformatted READ with I/O list transmits items until the list is satisfied, provided that the record contains at least as many items as required by the I/O list.

(4) An unformatted READ without I/O list can be used for positioning the file to the next logical record.

(5) A formatted READ should not be used for a record created by an unformatted WRITE.

Example:

Execution time of unformatted READ or WRITE statements is much faster than the corresponding formatted statements. They are generally used to write out large lists on temporary files and read them back when required by the program so that main memory is free for other processing. For example,

```
      DIMENSION ARRAY(1000)
      .
      .
      .
      WRITE(MTAPE) ARRAY
      .
      .
      .
      READ(MTAPE) ARRAY
      .
      .
      .
```

where: MTAPE represents a temporary file on magnetic tape, drum, or disc. The binary contents (1's and 0's) are written and then read back just as they appeared in the 1000 storage units (see Table 2–4) reserved for ARRAY.

6.8. AUXILIARY INPUT/OUTPUT STATEMENTS

The auxiliary input/output statements are:

- REWIND statement
- BACKSPACE statement
- ENDFILE statement

6.8.1. REWIND Statement

Function:

To cause an input/output device to be positioned at its initial point.

REWIND *u*

where: *u* is either an integer constant or an integer variable identifying an input/output device.

Rule:

The REWIND statement is applicable to such I/O devices as magnetic tape, disc, and drum units. Its implementation on devices such as card readers and printers depends upon the computer being used.

Example:

```
      10  FORMAT(.....)
          .
          .
          .
          READ(MTAPE,10).....
          .
          .
          .
          READ(MTAPE,10).....
          .
          .
          .
          REWIND MTAPE
          .
          .
          .
          READ(MTAPE,10).....
```

The first READ references the first record on MTAPE, the second READ references the second record, and the third READ references the first record.

6.8.2. BACKSPACE Statement

Function:

To backspace one record on a specified input/output device.

BACKSPACE *u*

where: *u* is either an integer constant or an integer variable identifying an input/output device.

Rules:

(1) The BACKSPACE statement is applicable to such I/O devices as magnetic tape, disc, and drum units. Its implementation on devices such as card readers and printers depends upon the computer being used.

(2) If the unit identified by *u* is already at its initial point, the BACKSPACE statement has no effect.

Example:

```
      10   FORMAT(.....)
           .
           .
           .
           READ(MTAPE,10).....
           .
           .
           .
           READ(MTAPE,10).....
           .
           .
           .
           BACKSPACE MTAPE
           .
           .
           .
           READ(MTAPE,10).....
```

The first READ references the first record, the second READ references the second record, and the third READ references the second record on MTAPE.

6.8.3. ENDFILE Statement

Function:

To record an endfile record on a specified input/output device.

ENDFILE *u*

where: *u* is either an integer constant or an integer variable identifying an input/output device.

Rules:

(1) Execution of this statement causes creation of a unique endfile record. The form of this record depends upon the processor being used.

(2) When such an endfile record is encountered during execution of a READ statement, the action taken depends upon the processor being used.

Example:

```
10  FORMAT(.....)
      .
      .
      .
    WRITE(MTAPE,10)
      .
      .
      .
    ENDFILE MTAPE
```

7. SPECIFICATION STATEMENTS

7.1. GENERAL

Specification statements are nonexecutable and must precede all other statements in a program unit except the FUNCTION, SUBROUTINE, BLOCK DATA (all in subprogram units), and FORMAT statements (see Table 2–2).

The specification statements are:

- type-statements
- DIMENSION statement
- EQUIVALENCE statement
- COMMON statement
- EXTERNAL statement

This section describes all but the COMMON and EXTERNAL statements. Because of their association with procedure subprograms, these statements are described in Section 8.

7.2. TYPE-STATEMENTS

Function:

To explicitly declare the data type of a symbolic name and/or declare an array.

INTEGER $v_1, v_2, \ldots$

and

REAL $v_1, v_2, \ldots$

and

DOUBLE PRECISION $v_1, v_2, \ldots$

and

COMPLEX $v_1, v_2, \ldots$

and

LOGICAL $v_1, v_2, \ldots$

where: each v is a variable, an array name, a function name, or an array declarator, separated from the next v by a comma.

Rules:

(1) Explicit type declaration of a symbolic name applies to all appearances of that symbolic name in the same program unit.

(2) Any symbolic name of data type double precision, complex, or logical must be typed explicitly, since these are not governed by the rules for implied typing.

(3) Explicit type declaration of an array refers to each of its array elements. In a main program unit, the array declarator may contain only integer constants in the subscript. In a subprogram unit, the array declarator may contain integer constants and/or integer variables.

(4) Type-statements of the forms shown may not be used for explicit type declaration of a function name in a function subprogram if that name appears in the FUNCTION statement. The FUNCTION statement provides for explicit type declaration of the function name of the function subprogram.

Examples:

(1)

```
      INTEGER MATRIX(2,3,4),XQRT
```

This statement types the array MATRIX and the variable XQRT as integer type. (This is redundant, since MATRIX is implied integer type.) It also declares the dimensions of array MATRIX. Such a statement can appear in any program unit.

(2)

```
      INTEGER MATRIX(2,N,4),XQRT
```

This statement can only appear in a procedure subprogram because it contains an adjustable dimension. The function reference or subroutine call that must precede such a statement defines the value for N at execution time.

7.3. DIMENSION STATEMENT

Function:

To declare one or more array(s).

DIMENSION $v_1, v_2, \ldots$

where: each v is an array declarator separated from the next by a comma.

Rules:

(1) If any of the arrays is adjustable, it can be declared only in a procedure subprogram.

(2) An array may be declared in a DIMENSION statement and explicitly typed in a type-statement, although a type-statement can accomplish both these functions.

Examples:

(1) The statements

```
      DIMENSION ARRAY1(3,4,5),ARRAY2(5,6)
      INTEGER ARRAY1
```

can also be written as:

```
      DIMENSION ARRAY2(5,6)
      INTEGER ARRAY1(3,4,5)
```

(2) The statement

```
      DIMENSION ARRAY1(J,K,L),ARRAY2(1,2,3)
```

can only appear in a procedure subprogram because of the adjustable array.

7.4. EQUIVALENCE STATEMENT

Function:

To permit sharing of the same storage space by two or more entities of the same program unit.

EQUIVALENCE (k_1), (k_2), ...

where: each k, enclosed in parentheses and separated from the next k by a comma, is a list having the form:

$$a_1, a_2, \ldots, a_m$$

where: each a is either a variable or an array element (not a dummy argument) with only constants as subscript expressions;
m is equal to or greater than 2.

Operation:

The following series of statements causes sharing of storage units:

```
      DIMENSION V2(1,2), K(3,2)
      COMPLEX V1(1,2,3), C
      DOUBLE PRECISION D(2,2)
      LOGICAL L
      EQUIVALENCE (V1(1), V2(1)), (D(2), K(1,2), L(4), C)
```

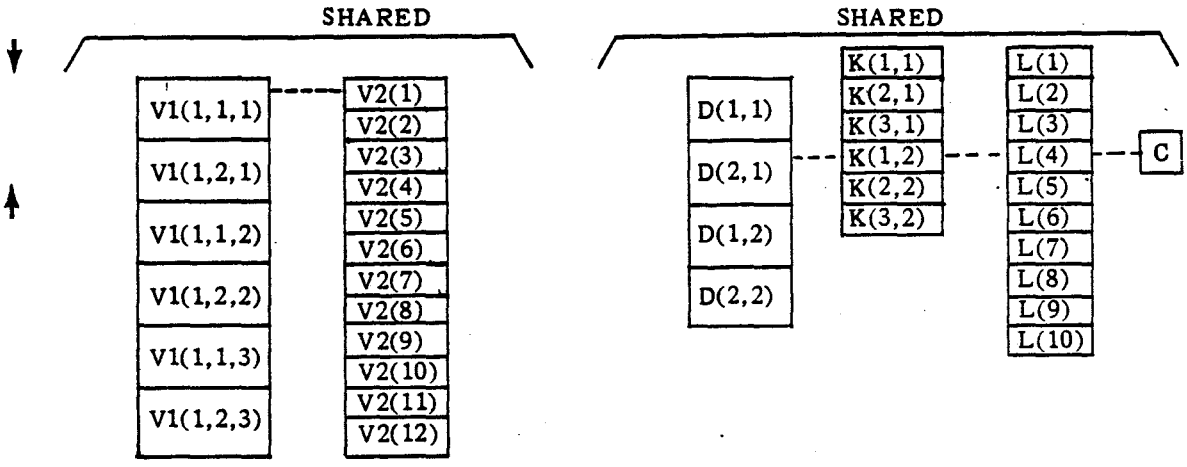


Figure 7–1. Effect of EQUIVALENCE Statement

From Figure 7–1, it can be seen that:

- Sharing is accomplished on the basis of storage units. For example, V2(4) shares the same storage space as the second storage unit of V1(1,2,1). (See Table 2–4 for storage unit attribute of the different data types.)
- An array can be referred to as a one-dimensional array in the EQUIVALENCE statement regardless of the number of dimensions in the array declarator.

Rules:

(1) Each entity in a given list, *k*, is assigned the same storage or part of the same storage. The sequence of items in a list is unimportant.

(2) The number of subscript expressions for an array element must be either the same number as in the array declarator or it must be 1. However, the number of storage units reserved by the array declarator must not be exceeded by the array element reference.

(3) If a two-storage entity is equivalenced to a one-storage entity, the one-storage entity will share storage with the first storage unit of the two-storage entity.

(4) If one array element is equivalenced to an element of another array, both arrays are equivalenced. (See 2.7.4 for the order of the array elements.)

(5) A dummy argument of a subprogram must not appear in an EQUIVALENCE statement.

(6) When one entity of a list, *k*, has its value defined, all its associated entities of the same list are defined. For example, when D(2,1) in Figure 7–1 is assigned a value, the contents of K(1,2), K(2,2), L(4), L(5), and C are defined.

(7) The programmer must avoid contradictions when referring to the same array more than once in an EQUIVALENCE statement. For example, the following statement

```
      EQUIVALENCE (A(3),C(2)),(A(2),D(2)),(C(2),D(1))
```

causes equivalence of A(3) and D(1), which is a logical contradiction.

(8) Special considerations apply when a COMMON statement is involved:

- When two entities are equivalenced, *both* entities must not appear in COMMON statements of the same program unit.
- Although it is always possible to equivalence an array past its end in an EQUIVALENCE statement (as was done in Figure 7–1), it is not always possible to extend an array ahead of its beginning (as was done to array DP in Figure 7–1) if one of the arrays is listed in a COMMON statement.

Examples:

(1) The primary intent of the EQUIVALENCE statement is conservation of storage. For example:

```
      DIMENSION INTGR(5000), RL(5000)
      DOUBLE PRECISION DP(5000)
      EQUIVALENCE (DP(1),INTGR(1)),(DP(2501),RL(1))
                  .
                  .
      DO 10 K=1,5000
   10 INTGR(K) = .....
                  .
                  .
      DO 20 K=1,5000
   20 RL(K) = .....
                  .
                  .
      WRITE(MPRINT,30) INTGR,RL
   30 FORMAT (.....)
                  .
                  .
      DO 40 K=1,5000
   40 DP(K) = .....
                  .
                  .
```

The EQUIVALENCE statement cuts down storage requirements by 1000 storage units.

(2) The EQUIVALENCE statement can force the storage of *non-COMMON* variables or arrays. In this example, the variables R, T, B, and A are stored consecutively in memory:

```
      EQUIVALENCE (R,V(1)),(T,V(2)),(B,V(3)),(A,V(4))
```

When the statement

```
      READ(MCARD,F) V(K)
```

is encountered, a value for R, T, B, or A is read in depending on whether the value of K is 1, 2, 3, or 4, respectively.

(3) An EQUIVALENCE statement can make allowances for errors in spelling that occur in a program unit. For instance, if an array was declared as CHIEF(3,4,5) and the name CHEIF was occasionally used to refer to the array, it is only necessary to write

```
      EQUIVALENCE (CHIEF(1),CHEIF(1))
      DIMENSION CHEIF(3,4,5)
```

rather than correct every statement containing CHEIF.

8. PROCEDURES AND PROCEDURE SUBPROGRAMS

8.1. GENERAL

Up to now, programs that contained only one program unit, the main program, have been considered, with little discussion of procedures. This section describes procedures that can be defined in a program unit and expands the concept of procedure subprograms, showing how the programmer can use these and, where they are not supplied with the compiler, create his own external procedure subprograms.

The four categories of procedures and procedure subprograms are:

- statement functions
- intrinsic functions
- external functions
- external subroutines

A procedure may be a single executable statement or it may be a fixed series of statements. The operation(s) to be performed by a given procedure is always the same. Each time a procedure is invoked, the values to be used in it can be defined by the programmer if required and the procedure can return values. The advantage of using procedures is that the series of operations for a given procedure need be defined only once, eliminating the necessity of redefining the procedure each time it is invoked.

There are two methods for passing values to and from procedures: (1) by use of arguments, (2) by use of the COMMON statement. Either, or both, may be used, except that a function reference requires at least one argument.

A function procedure is invoked by a function reference of the form:

$$\textit{function name}\ (a_1, a_2, \ldots, a_n)$$

where *function name* is the symbolic name that identifies the function procedure, and each *a* is an *actual argument.*

A function reference appears as a primary either in an arithmetic or in a logical expression. For each actual argument in the function reference there must be a *dummy argument* in the definition of the function procedure that will be replaced by the value of its actual argument during execution. Some function procedures, namely intrinsic and *basic* external functions, are furnished with the processor and need not be defined by the programmer. Statement functions and function subprograms are written by the programmer. An example of a basic function reference is:

```
      THETA = 1.5
      C = A * B * COS(THETA)
```

The function reference is COS(THETA), and THETA is the actual argument that must be defined before the function reference is encountered. During execution of the arithmetic assignment statement, THETA is evaluated and a value for COS(THETA) is then returned to the arithmetic expression so that it can be computed and a value can be assigned to C. A function reference must have at least one actual argument, but may have many. For example, a function subprogram is called AREA and computes the area of a triangle, given the three sides, and is referenced in an arithmetic expression as Y = AREA(X,Y,Z). This subprogram can be expanded to also return a value for the perimeter and reference it by Y = AREA(X,Y,Z,PER), where the value of PER is used in any statement that follows the function reference. A function reference cannot redefine any other values in the same expression containing the function reference.

A subroutine call invokes a procedure external to the calling program unit, but it does not necessarily return any values to the calling program unit and it need not have any actual arguments. After the task of the subroutine (defined by the programmer) is completed, control is returned to the next executable statement of the calling program unit. For example, if the same list of records is read in many times in a program, the procedure can be defined once in an external subroutine subprogram and be called upon many times in the program by CALL READIN(ARRAY), where ARRAY is the actual argument.

8.1.1. Statement Functions and Intrinsic Functions

Statement functions and intrinsic functions are function procedures requiring few *computer* (machine coded) instructions for execution. Therefore, when the program is submitted in machine code form for execution, it is feasible to insert these computer instructions *inline* for each reference. For example, if the intrinsic function ABS is referenced many times in a program, the instructions for this procedure will be repeated inline in the machine coded executable program as shown in Figure 8–1.

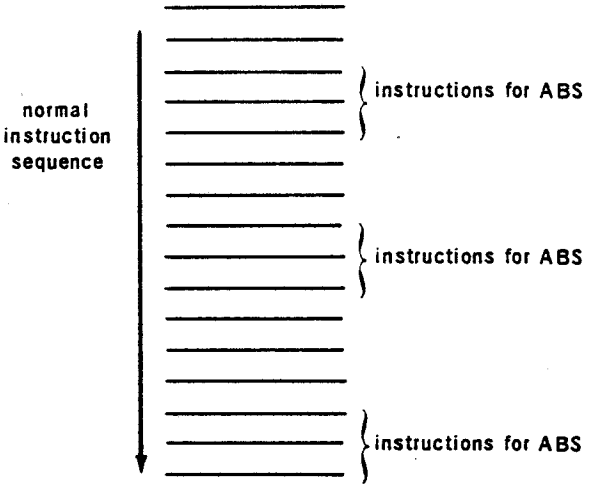


Figure 8–1. Inline Coding of Statement Functions and Intrinsic Functions

8.1.2. External Procedure Subprograms

External procedure subprograms (function and subroutine subprograms) are function procedures that generally require more than one FORTRAN statement for definition and many computer instructions for execution. Therefore, the set of instructions is recorded only once in the machine coded program and placed *out-of-line* so that it can be entered only by a machine coded control instruction. After execution, control is returned to the referencing program by another machine coded control instruction. For instance, if the cosine function COS is referenced many times in a program, the machine coded program will appear as in Figure 8–2.

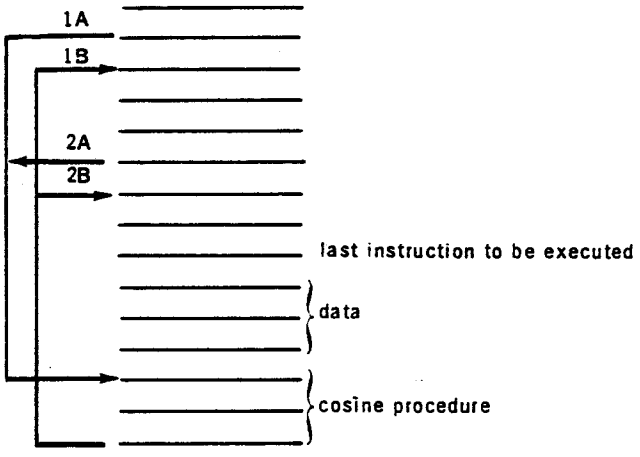


Figure 8–2. Out-of-Line Coding of External Procedure Subprograms

8.1.3. Communication Between Program Units

Values may be transmitted to a procedure or a procedure subprogram through the actual arguments of the reference or call and returned through a function name and actual arguments. The COMMON statement is another method of transmitting information between different program units. The EXTERNAL statement identifies a symbolic name used in the list of actual arguments as the name of an external subprogram.

8.1.4. Valid Forms of Arguments

Table 8–1 lists, for reference purposes, the valid forms of procedure arguments. These are explained, in more detail, in the applicable paragraphs.

PROCEDURE	FORM OF ARGUMENT(S)	
	ACTUAL	DUMMY
statement function	arithmetic expression, logical expression	variable
basic external function	arithmetic expression of required type	–
function subprogram (1)	arithmetic expression, logical expression, array name	variable, array name, name of external procedure
subroutine subprogram (1)	arithmetic expression, logical expression, array name, name of external procedure, Hollerith constant	variable, array name, name of external procedure
intrinsic function	arithmetic expression of required type	–

(1) If an actual argument corresponds to a dummy argument that is defined or redefined in the subprogram, the actual argument must be a variable, array element, or array name.

Table 8–1. Forms of Argument

8.2. STATEMENT FUNCTION

There are two types of statement functions: arithmetic statement function and logical statement function.

8.2.1. Arithmetic Statement Function

Function:

To define an arithmetic procedure with one FORTRAN statement. This procedure may be used in its program unit as many times as required.

function name(a_1,a_2, ...,a_n) = *limited arithmetic expression*

where: *function name* is a symbolic name identifying the procedure.

the *a*'s comprise the list of one or more *dummy arguments*, a list of variables enclosed in parentheses and separated by commas.

limited arithmetic expression is similar to an arithmetic expression except that it may not contain an array element as a primary, and it may not reference, in any way, the statement function of which it is a part.

the maximum value of *n* is defined for each processor; there is no standard value.

Rules:

(1) An arithmetic statement function reference is a primary in an arithmetic expression. The actual arguments are arithmetic expressions and must correspond in number, order, and type with the dummy arguments.

For example, the statement function

```
      AVRGE(A,B,C) = (A + B + C) / 3.0
```

can be used with the following statement:

```
      Z = Y - AVRGE(R,S,T)
```

The value for the actual argument R is substituted for its dummy argument A, S for B, and T for C. A value is returned to the statement function reference so that the arithmetic expression Y – AVRGE(R,S,T) can be evaluated and assigned to Z. The statement function reference

```
      Z2 = Y2 - AVRGE(R, S, N)
```

is incorrect because the data type of N is inconsistent with the data type of C. The statement function reference

```
      Z3 = Y3 - AVRGE(3, S2)
```

is incorrect because there are only two actual arguments, and three are required by the statement function definition.

(2) All arithmetic statement functions must appear between the specification statements (if any) and the first executable statement of the program unit (see Table 2–2). The limi ted arithmetic expression may use references to *previously defined* statement functions as primaries.

(3) The function name may not be used as a variable within the same program unit; it is used as part of a statement function reference.

(4) Each dummy argument in the list must appear at least once in the limited arithmetic expression.

(5) The arithmetic statement function definition is a nonexecutable statement and cannot have a statement label.

(6) The function name is governed by the rules for implied and explicit typing of symbolic names; it need not have the same data type as its arguments. Evaluation of the limited arithmetic expression is subject to the rules for mixed type evaluation of arithmetic expressions, and assignment of a value to the function reference is subject to the rules of Table 4–1.

(7) Variable names used as dummy arguments are purely local to the arithmetic statement definition. They may be used elsewhere in the same program unit for any purpose.

Examples:

(1)

```
      DIMENSION ARRAY(3)
      SUMSQ(A,B,C) = A**2 + B**2 + C**2
      .
      .
      .
      DO 10 N = 1,3
   10 ARRAY(N) = N
      Z = SUMSQ(ARRAY(1), ARRAY(2), ARRAY(3))
```

A value of 14.0 is assigned to Z. Note that the mixed data types in the limited arithmetic expression conform to the rules listed in Tables 3–1 and 3–2.

(2) The same statement function can be defined with only variables in its limited arithmetic expression, but these variables must be defined prior to the reference. For example:

```
      DIMENSION ARRAY(3)
      SUMSQ(A,B,C) = A**K1 + B**K2 + C**K3
      .
      .
      .
      K1 = 2
      K2 = 2
      K3 = 2
      DO 10 N = 1,3
   10 ARRAY(N) = N
      Z = SUMSQ(ARRAY(1), ARRAY(2), ARRAY(3))
```

(3) A statement function definition may contain references to *previously defined* statement functions. For example:

```
      SUM(X,Y) = X + Y
      SUMSQ(A,B) = A**2 + B**2 + SUM(A,B)**2
      .
      .
      .
      Z = 3.0
      C = SUMSQ(Z + 2.0, 4.0) + Z
```

The variable C will be evaluated as the real type approximation of $5.0^2 + 4.0^2 + 9.0^2 + 3.0$ or 125.0.

8.2.2. Logical Statement Function

Function:

To define a logical procedure with one FORTRAN statement that may be referenced in its program unit as many times as required.

function name(a_1,a_2, ...,a_n) = *limited logical expression*

where: *function name* is a symbolic name, explicitly typed as logical, that identifies the procedure.

the *a*'s comprise the list of one or more dummy arguments, a list of variables enclosed in parentheses and separated by commas.

limited logical expression is similar to a logical expression except that it may not contain an array element as a primary, and it may not contain a reference, directly or indirectly, to the statement function of which it is a part.

the maximum value of *n* is defined for each processor; there is no standard value.

Rules:

(1) A logical statement function reference is a primary in a logical expression. The actual arguments may be arithmetic or logical expressions and must correspond in number, order, and type with their corresponding dummy arguments.

(2) All logical statement functions must appear between the specification statements and the first executable statement of a program unit.

(3) The function name may not be used as a variable within the same program unit; it may only appear as part of a statement function reference.

(4) Each dummy argument in the list must appear at least once in the limited logical expression.

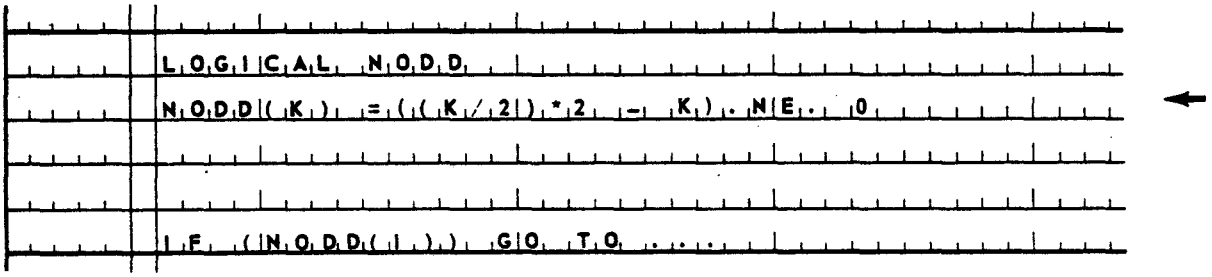
(5) The logical statement function definition is a nonexecutable statement and cannot have a statement label.

(6) Evaluation of the limited logical expression is subject to the rules for evaluation of a logical expression (see 3.4), and it assigns a value of either true or false to the statement function reference.

(7) Variable names used as dummy arguments are purely local to the statement function and may be used elsewhere in the same program unit for any purpose.

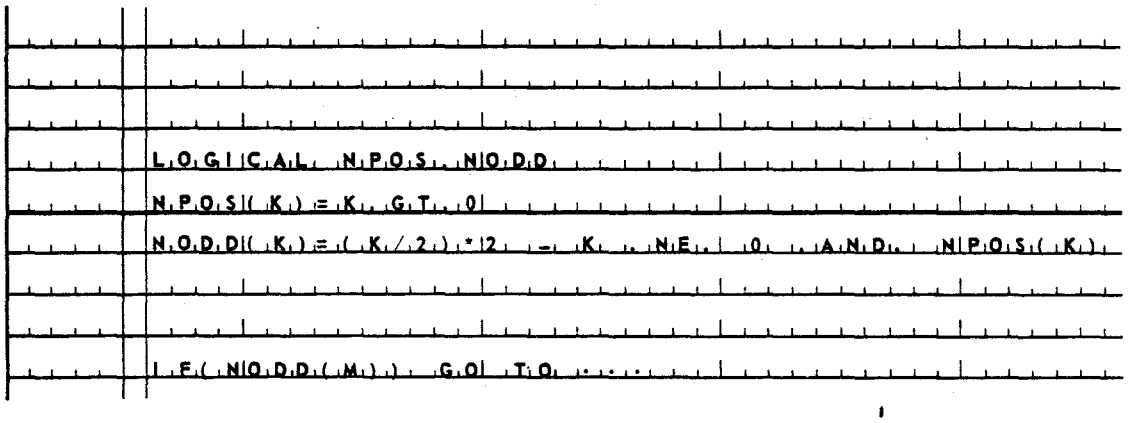
Examples:

(1)



In the logical IF, if I is an odd integer, the GO TO will be executed (see 3.2.4, example 2).

(2)



In the logical IF, if M is a positive odd integer, the GO TO will be executed.

8.3. INTRINSIC FUNCTIONS

Intrinsic functions (built-in functions) are provided with the processor and are not written or modified by the programmer. A list of standard FORTRAN intrinsic functions is presented in Table 8–2.

INTRINSIC FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	SYMBOLIC NAME	TYPE OF ARGUMENT	TYPE OF FUNCTION
absolute value	$\|a\|$	1	ABS IABS DABS	real integer double	real integer double
truncation	sign of a times largest integer $\leq \|a\|$	1	AINT INT IDINT	real real double	real integer integer
remaindering*	$a_1 \pmod{a_2}$	2	AMOD MOD	real integer	real integer
choosing largest value	max $(a_1, a_2, \ldots)$	≥ 2	AMAX0 AMAX1 MAX0 MAX1 DMAX1	integer real integer real double	real real integer integer double
choosing smallest value	min $(a_1, a_2, \ldots)$	≥ 2	AMIN0 AMIN1 MIN0 MIN1 DMIN1	integer real integer real double	real real integer integer double
float	conversion from integer to real	1	FLOAT	integer	real
fix	conversion from real to integer	1	IFIX	real	integer
transfer of sign	sign of a_2 times $\|a_1\|$	2	SIGN ISIGN DSIGN	real integer double	real integer double
positive difference	$a_1 - \min(a_1, a_2)$	2	DIM IDIM	real integer	real integer
obtain most significant part of double precision argument		1	SNGL	double	real
obtain real part of complex argument		1	REAL	complex	real
obtain imaginary part of complex argument		1	AIMAG	complex	real
express single precision argument in double precision form		1	DBLE	real	double
express two real arguments in complex form	$a_1 + a_2\sqrt{-1}$	2	CMPLX	real	complex
obtain conjugate of a complex argument		1	CONJG	complex	complex

*The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [x]a_2$, where $[x]$ is the greatest integer whose magnitude does not exceed the magnitude of a_1/a_2 and whose sign is the same as a_1/a_2 (see example 3).

Table 8-2. Intrinsic Functions

Rules:

(1) An intrinsic function is referenced as a primary in an arithmetic or relational expression, by name and list of actual arguments. The actual arguments must agree in type, number, and order with the specifications of Table 8–2 and may be any expression of the specified type. Note, however, that the number of arguments for the MAX and MIN intrinsic functions is variable.

(2) The intrinsic functions AMOD, MOD, SIGN, ISIGN, and DSIGN are not defined when the value of the second argument is zero.

(3) It is not necessary to declare the type of an intrinsic function in the program unit that contains a reference to an intrinsic function. The data type is already known to the processor.

(4) For a valid intrinsic function reference in a program unit, the symbolic name:

   (a) must appear as specified in Table 8–2, followed by the list of actual arguments, in parentheses, also specified in Table 8–2;

   (b) must not appear in an EXTERNAL statement or be used as a variable or array name in a program unit where it appears as a reference;

   (c) must not appear in any type declaration different from the implied type declaration of Table 8–2.

(5) If a particular intrinsic function is not referenced in a program unit, its symbolic name may be used for any valid purpose in that program unit.

Examples:

(1) This series of statements reads in the six complex type elements of array VCTR, then prints the conjugate of each array element, using intrinsic functions CMPLX, REAL, and AIMAG.

```
      COMPLEX VCTR(6)
      READ(1,10) VCTR
   10 FORMAT ( . . . . )
      DO 20 K = 1, 6
   20 VCTR(K) = CMPLX(REAL(VCTR(K)), -AIMAG(VCTR(K)))
      WRITE(3,30) VCTR
   30 FORMAT ( . . . . )
```

(2) This example uses the MAX0 intrinsic function.

```
      I=20
      J=30
      K=40
      .
      .
      IF(MAX0(I,J,K)-40),100,200,300
```

In this case, the maximum value is 40, causing the arithmetic IF to jump to the statement labeled 200.

(3) The remaindering function (also called the modulo or residue function) is useful in modulo-arithmetic. This intrinsic function divides one argument by the other and retains only the remainder prefixed by the sign of the quotient. For example, the reference MOD (9,8) returns a value of 1, and the reference AMOD (1.22, 1.1) returns a value of 0.12. In the statement

```
      IF(MOD(J,K)),10,20,10
```

a jump is made to the statement labeled 20 only if J is zero or an exact multiple of K (K can be multiplied by an optionally signed integer to obtain J).

(4) The IFIX and FLOAT can be used to calculate the values of arithmetic expressions where certain combinations of data types are not defined in standard FORTRAN (see Table 3–3). For example:

```
      K=M-IFIX(X*FLOAT(K))
```

The FLOAT function uses the real type form of K for the multiplication. The IFIX converts the real type result of the multiplication to integer type form so the subtraction can be performed.

8.4. RETURN STATEMENT

Function:

To return program control from a function or subroutine subprogram to the program unit that referenced or called the subprogram.

RETURN

Rules:

(1) There must be at least one RETURN statement in every function and subroutine subprogram.

(2) When executed, the RETURN statement terminates further execution of the subprogram that contains it.

Examples:

See 8.5.2.2 and 8.6.3.

8.5. EXTERNAL FUNCTIONS

External functions are of two types: external subprograms written by the programmer in FORTRAN, and *basic functions*. Basic functions are external function procedures supplied with the processor and stored in auxiliary storage in non-FORTRAN representation. These basic functions do not require compilation and can usually also be utilized by non-FORTRAN users of the processor.

8.5.1. Basic External Functions

Table 8–3 lists standard basic external functions.

BASIC EXTERNAL FUNCTION	DEFINITION	NUMBER of ARGUMENTS**	SYMBOLIC NAME	TYPE OF ARGUMENT	TYPE OF FUNCTION
exponential	e^a	1	EXP	real	real
		1	DEXP	double	double
		1	CEXP	complex	complex
natural logarithm	$\log_e(a)$	1	ALOG	real	real
		1	DLOG	double	double
		1	CLOG	complex	complex
common logarithm	$\log_{10}(a)$	1	ALOG10	real	real
		1	DLOG10	double	double
trigonometric sine	$\sin(a)$	1	SIN	real	real
		1	DSIN	double	double
		1	CSIN	complex	complex
trigonometric cosine	$\cos(a)$	1	COS	real	real
		1	DCOS	double	double
		1	CCOS	complex	complex
hyperbolic tangent	$\tanh(a)$	1	TANH	real	real
square root	$(a)^{1/2}$	1	SQRT	real	real
		1	DSQRT	double	double
		1	CSQRT	complex	complex
arctangent	$\arctan(a)$	1	ATAN	real	real
		1	DATAN	double	double
	$\arctan(a_1/a_2)$	2	ATAN2	real	real
		2	DATAN2	double	double
remaindering*	$a_1 \pmod{a_2}$	2	DMOD	double	double
modulus	$\sqrt{(\text{real part})^2 + (\text{imaginary part})^2}$	1	CABS	complex	real

*The function DMOD (a_1, a_2) is defined as $a_1 - [x]a_2$ where $[x]$ is the largest integer whose magnitude does not exceed the magnitude of a_1/a_2 and whose sign is the same as the sign of a_1/a_2.

**All angles are expressed in radians.

Table 8–3. Basic External Functions

Rules:

(1) A basic external function is referenced as a primary in an arithmetic expression.

(2) The form of the reference is:

function name (arg_1)

or

function name (arg_1, arg_2)

where: *function name* corresponds to one of the function names listed in Table 8–3.

each *a* is a dummy argument, the list is enclosed in parentheses, and the *a*'s are separated by commas.

(3) It is not necessary to explicitly type a basic function name in the referencing program unit.

Examples:

(1) The following sequence finds the sine of 30 degrees (which is 0.5):

```
      DOUBLE PRECISION PI,Y
      DATA PI/3.14159265358979D0/
      Y = DSIN(PI/6.0)
      WRITE (UPRINT,10) Y
   10 FORMAT(1X,F10.8)
```

(2) This example shows a basic external function reference containing another function reference in the list of actual arguments:

```
      Z = SQRT(AMAX1(A,B,C) + AMAX1(D,E,F))
```

(3) This example shows a basic external function reference in a logical expression:

```
      IF(AMIN1(A,B,C).LT.-0.5 .AND. SQRT(D).GT.0.5) GO TO 30
```

8.5.2. Function Subprograms

Function subprograms permit the programmer to create external arithmetic and logical procedures and refer to these many times in the body of the program using function references. The first statement of the external function subprogram definition must be a FUNCTION statement, and the last line must be an END line; between these there must be at least one RETURN statement and a definition of the function name. Each reference to a subprogram in an arithmetic or logical expression will have a value returned to it at the point of reference. Additional values may be returned for use in statements that follow the reference.

8.5.2.1. FUNCTION Statement

Function:

To identify an external function subprogram.

t **FUNCTION** *function name* $(a_1, a_2, \ldots, a_n)$

where: *t* is explicit type declaration of the value to be returned in the function reference and may be omitted, in which case the implied type of *function name* determines the data type of the value.
function name is a symbolic name identifying the function subprogram; each *a* is a dummy argument; the list is enclosed in parentheses, and the *a*'s are separated by commas.
n must be at least 1, and its maximum value depends upon the processor being used.

Rules:

(1) If *t*, the explicit type declaration, is present, it must be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL.

(2) Each dummy argument must be an external procedure name (see EXTERNAL statement), a variable, or an array name.

(3) The function name used in the FUNCTION statement must not appear in any other nonexecutable statement of the function subprogram.

(4) The symbolic names of the dummy arguments must not appear in an EQUIVALENCE, COMMON, or DATA statement of the function subprogram.

For examples, see 8.5.2.2.

8.5.2.2. Function Subprogram Definition

Function:

To define the procedure that computes a result(s) returned to the referencing program unit.

```
FUNCTION statement
        .
        .
        .
RETURN
        .
        .
        .
END
```

Rules:

(1) All rules applicable to the FUNCTION statement are applicable to the external function definition.

(2) The function name of the subprogram must appear as a variable at least once in the subprogram. During every execution of the subprogram, this variable must be defined before it may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in this subprogram is the value of the function and is the value returned to the function reference.

(3) The subprogram may define and redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

(4) The function subprogram may contain any statements except BLOCK DATA, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the subprogram being defined.

(5) The function subprogram must contain at least one RETURN statement.

(6) If a function reference causes a dummy argument in the referenced function to become associated with another dummy argument in the same function or with an entity in common (see COMMON statement), a definition of either within the function is prohibited. An example of such a function reference is:
Y = ADD(A,A)

(7) The following rules apply to arguments involving arrays or array elements:

(a) If an actual argument is an array element, its dummy argument must be either a variable or an array name.

(b) If an actual argument is an array name, its dummy argument must be an array name and that array must be declared in the subprogram with a size (in elements) that does not exceed the actual argument array.

(c) If the actual argument is the xth element of an array containing z elements and the dummy argument is an array name, that array must be declared in the subprogram with a size that does not exceed $z - x + 1$ elements.

(d) A dummy array declarator may use one, two, or three subscript expressions, regardless of how the actual array was declared. Each subscript expression of the dummy array declarator may be either an integer constant or an integer variable. If any subscript expression of the dummy array declarator is an integer variable, that array is called an *adjustable array*.

An adjustable array declarator must have each of its integer variable subscript expressions listed as dummy arguments and each must be defined by its actual argument. These variables must not be redefined in the subprogram.

Examples:

(1) This subprogram calculates the sum of the squares of all positive odd integers from 1 through the given positive integer K:

```
      INTEGER FUNCTION SM(K)
      SM = 0
      KSWTCH = -1
      DO 10 N=1,K
      KSWTCH = -KSWTCH
   10 IF(KSWTCH .EQ. 1) SM = SM + N**2
      RETURN
      END
```

Thus, if this external function subprogram is referenced by

```
      INTEGER SMQDSQ
      .
      .
      N3 = 5
      M = 5 + SM(N3)
```

the actual argument N3, which has a value of 5, will be substituted for its dummy argument K in the subprogram. The value of SM is $1^2 + 3^2 + 5^2 = 35$. When the RETURN statement is reached, this value is substituted for the function reference, so that the value assigned to M is 40.

(2) This function subprogram, which has more than one RETURN statement, calculates the absolute sum of all array element values in a 100-element array:

```
      FUNCTION SMARAY (ZRAY)
      DIMENSION ZRAY(100)
      SMARAY=0.0
      DO 20 K=1,100
   20 SMARAY=SMARAY+ZRAY(K)
      IF (SMARAY.GE.0.0) RETURN
      SMARAY= -SMARAY
      RETURN
      END
```

The referencing program unit contains:

```
      DIMENSION ARRAY(100), BRAY(100)
        .
        .
      SUM = SMARAY(ARRAY) + SMARAY(BRAY)
```

(3) This function subprogram also returns a value through one of its dummy arguments. Given the three sides of a triangle, it calculates the area and returns a value for the perimeter.

```
      FUNCTION AREA(A, B, C, PER)
      PER = A + B + C
      S = PER/2.0
      AREA = SQRT(S*(S-A)*(S-B)*(S-C))
      RETURN
      END
```

The referencing program contains:

```
      TRNGL = AREA(X, Y, Z, PERIM)
      .
      .
      .
      BNDS = PERIM * PERIM
```

Note that, in the referencing program, one value for AREA will be returned in the function reference. Another value will be returned as the value of PERIM. This value must not be used in the same statement as the function reference, but can be used in statements that are executed after the function reference. If a value is to be returned to an actual argument in the function reference, that actual argument must not be a constant; this would be an attempt to redefine a constant.

(4) The following subprogram illustrates the use of an adjustable array. The subprogram calculates the sum of the elements in an array, but each time the subprogram is referenced, the dummy array may have different dimensions.

```
      FUNCTION SIGMA(ARRAY,N)
      DIMENSION ARRAY(N)
      SIGMA = 0.0
      DO 10 K=1,N
   10 SIGMA = SIGMA + ARRAY(K)
      RETURN
      END
```

The referencing program unit may contain:

```
      DIMENSION SET1(4,5,6), SET2(1,1), SET3(8,1,0)
      AVRG1 = SIGMA(SET1,4*5*6) / (4.0*5.0*6.0)
      AVRG2 = SIGMA(SET2,1*1*1) / 1.1.0
      AVRG3 = SIGMA(SET3,8*1*0) / (8.0*1.0)
      AVRGE = SIGMA(SET1(3,5,6),2) / 2.0
```

Note that this last reference uses an array element as the actual argument. This array element is the 119*th* element (see Table 2–6) in an array that can contain a maximum of 120 elements. Hence, the function reference is obtaining the sum of the last two array elements of array SET1.

(5) This example uses the Newton-Raphson method to find the cube root of a number, *y*, as follows:

(a) Make an initial guess at the cube root of *y*. (In this program, *y* is divided by 3 if its absolute value is greater than or equal to 1; otherwise, it is multiplied by 3.) This value is called x_1.

(b) Substitute x_1 in the expression

$$\frac{1}{3}(2x_1 + y/x_1^2)$$

to arrive at a new value, x_2.

(c) Compare x_2 with x_1. If the comparison satisfies a criterion set up by the programmer, x_2 is the cube root of *y*; if not, x_2 becomes x_1, and this new x_1 is substituted in the above expression to arrive at a new value for x_2.

Steps (b) and (c) are repeated until the comparison of x_2 with x_1 is satisfied.

```
      FUNCTION CBROOT(A)
      IF(ABS(A).GE.1.0) X=A/3.0
      IF(ABS(A).LT.1.0) X=A*3.0
   10 CBROOT=(2.0*X+(A/(X*X)))/3.0
      IF(CBROOT.EQ.X) RETURN
      X=CBROOT
      GO TO 10
      END
```

A reference to this subprogram could be:

```
      Y = CBROOT(A+B+C)+D
```

Using this subprogram, the following is a list of successive CBROOT's calculated in finding the cube root of 1.0, 0.9, 0.1, and -0.027. The solution is repeated several times because, for this particular processor, the arithmetic operations are performed internally with greater precision than was required for the output.

cube root of 1.000	cube root of .100
3.2222223	.5703704
2.1802527	.4827094
1.5236256	.4648626
1.1593397	.4641599
1.0208964	.4641589
1.0004248	.4641589
1.0000002	
1.0000000	cube root of -.027
	-1.4257421
cube root of .900	-.9549222
1.8411523	-.6464846
1.3159345	-.4525238
1.0505315	-.3456326
.9721879	-.3057596
.9655354	-.3001078
.9654894	-.3000000
.9654894	-.3000000
.9654894	-.3000000

Such procedures can be refined by more accurate first estimates, the use of double precision, and different exits from the loop. For example, the precision of the result could be limited to a satisfactory value with the statement:

```
      IF(ABS(CBROOT-X).LT.DELTA) RETURN
```

where: DELTA is an additional argument.

8.5.2.3. References to Function Subprograms

Function:

To obtain a single value for use in an expression, by reference to a function subprogram.

name(a_1,a_2, ...,a_n)

where: *name* is the programmer-written symbolic name of the function.
each *a* is an actual argument; the list is enclosed in parentheses, and the *a*'s are separated by commas.
n has a minimum value of 1.

Rules:

(1) Each actual argument may be a logical or arithmetic expression, an array name, or the name of an external procedure (see EXTERNAL statement).

(2) If a value is to be returned through an actual argument, that actual argument need not have its value defined prior to the reference, and it must not be a constant.

(3) The actual arguments must agree in order, number, and type with the dummy arguments of the subprogram.

(4) If the actual argument is an array element, its dummy argument may be either an array name or a variable; if the actual argument is an array name, its dummy argument must be an array name.

(5) If an actual argument is the name of an external procedure, its dummy argument must be used in the subprogram as the name of a function or subroutine external to the procedure.

(6) The operation of a RETURN statement in the function subprogram must intervene between each reference to the same function subprogram.

Examples:

See FUNCTION, EXTERNAL, and COMMON statements.

8.6. SUBROUTINE SUBPROGRAMS

A subroutine subprogram is an external subprogram defined by FORTRAN statements, starting with the SUBROUTINE statement, and is invoked by a CALL statement.

8.6.1. CALL Statement

Function:

To invoke a subroutine subprogram.

CALL *name*

or

CALL *name* $(a_1, a_2, \ldots, a_n)$

where: *name* is the symbolic name of the subroutine subprogram.
each *a* is an actual argument; the list is enclosed in parentheses, and the *a*'s are separated by commas.

Rules:

(1) Each actual argument may be an expression (logical or arithmetic), an array name, a Hollerith constant, or the name of an external procedure (see EXTERNAL statement).

(2) The actual arguments must agree in order, number, and (except for a Hollerith constant) type with the corresponding dummy arguments of the subroutine.

(3) If an actual argument corresponds to a dummy argument that is defined or redefined in the subroutine, the actual argument must be a variable, array element, or array name.

(4) If an actual argument is an external function name or subroutine name, the corresponding dummy argument must be used as an external function name or subroutine name, respectively.

(5) If an actual argument is an array element, its corresponding dummy argument must be either a variable or array name; if an actual argument is an array name, its corresponding dummy argument must be an array name.

(6) If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in common (see COMMON statement), a definition of either entity is prohibited in the subroutine. An example of such a reference is: CALL ADD(A, A).

(7) Between any two successive calls on the same subroutine there must be the operation of a RETURN statement in the subroutine.

8.6.2. SUBROUTINE Statement

Function:

To identify a subroutine subprogram.

SUBROUTINE *name*

or

SUBROUTINE *name* $(a_1, a_2, \ldots, a_n)$

where: *name* is a symbolic name that identifies the subprogram.
each *a* is a dummy argument; the list is enclosed in parentheses, and the *a*'s are separated by commas.

Rule:

Each dummy argument is a variable, an array name, or an external procedure name (see EXTERNAL statement).

8.6.3. Subroutine Definition

Function:

To completely define the step-by-step procedure for a subroutine.

```
SUBROUTINE statement
      .
      .
      .
RETURN
      .
      .
      .
END
```

Rules:

(1) The first statement in the subroutine must be the SUBROUTINE statement, and the last line must be the END line. The subroutine definition must contain at least one RETURN statement.

(2) The symbolic name of the subroutine must not appear in any statement of the subroutine except the first.

(3) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement of the subroutine.

(4) The subroutine may contain any statements except BLOCK DATA, FUNCTION, another SUBROUTINE statement, or any statement that directly or indirectly calls this subroutine.

(5) The subroutine may define or redefine one or more of its arguments so as to return results through its arguments. If the subroutine returns a result through a dummy argument, its actual argument must not be a constant.

(6) Adjustable arrays may be used in the subroutine, in which case any adjustable dimensions may be passed as actual and dummy arguments for use in the array declarator of the subroutine.

(7) The RETURN statement returns program control to the next executable statement following the CALL statement in the calling program unit.

Examples:

(1) This subprogram clears the elements of any array to 0.0.

```
      SUBROUTINE CLEAR(ARRAY,N)
      DIMENSION ARRAY(N)
      DO 10 K=1,N
   10 ARRAY(K) = 0.0
      RETURN
      END
```

A calling program unit contains:

```
      DIMENSION ARAY(4,5,6), BRAY(5,6), CRAY(1,1),
     .
     .
     .
      CALL CLEAR(ARAY,4*5*6)
     .
     .
     .
      CALL CLEAR(BRAY,5*6)
     .
     .
     .
      CALL CLEAR(CRAY,1,1)
     .
     .
     .
      CALL CLEAR(ARAY(1,1,1),100)
```

The first three subroutine calls will clear all of the referenced arrays; the last subroutine call will clear only the first 100 elements of the array ARAY.

(2) The following subroutine subprogram, UPDATE, computes the current value of a savings bank account given a starting value, V, the number of interest periods, M, and the interest rate per interest period, R.

```
      SUBROUTINE UPDATE(M,V,R)
      IF(M.EQ.0) RETURN
      R2=1.0+R
      DO 10 K=1,M
   10 V=V*R2
      RETURN
      END
```

This subroutine subprogram is invoked in the following program. The program computes the current value of an account based on the assumption that the interest is computed and accrued at the end of each three-month period from the month of deposit.

The first card of the data deck is organized as follows:

columns 1 and 2	– the current month (1–12), right-justified
column 3	– blank character
columns 4 through 7	– current year

Each succeeding data card is organized as follows:

column 1	– blank character
columns 2 through 7	– account number, right-justified
column 8	– blank character
columns 9 through 48	– name of depositor, left-justified
column 49	– blank character
columns 50, 51	– month of last update or (if no previous update) month of deposit, right-justified
column 52	– hyphen (-) or minus character
columns 53 through 56	– year of last update or (if never updated) year of deposit
column 57	– $ (dollar sign)
columns 58 through 66	– *dddddd..dd*, the last updated value or original deposit, where each *d* is a decimal digit
columns 67, 68, 69	– blank characters
column 70	– months (0, 1, 2, or 3) from last update or original deposit to end of next interest period. (If original deposit and no update, it must be 3.)

The last card of the data deck contains all blank characters, except for column 70, which contains a digit greater than 3.

The subroutine is invoked by the following program which prints the updated record and punches an updated card record concurrently (assuming output device 5 is a card punch).

```
      DIMENSION NAME(8)
      INTEGER CMONTH,CYEAR,YEAR,RMTHS
      READ(1,10) CMONTH,CYEAR,RATE
   10 FORMAT(1X,I4,F8.4)
   50 READ(1,20,0) NAKINT,NAME,MONTH,YEAR,VALUE,RMNTHS
   20 FORMAT(1X,8A5,I3,I5,1X,F9.2,I4)
      IF(RMNTHS.GT.12) STOP
      MTIME=12.*(CYEAR-YEAR)+(CMONTH-MONTH)+(3-RMNTHS)
      M=MTIME/3
      RMNTHS=3-MOD(MTIME,3)
      CALL UPDATE(M,VALUE,RATE)
      NOUT=3
   40 WRITE(NOUT,30)NAKINT,NAME,CMONTH,CYEAR,VALUE,RMNTHS
   30 FORMAT(1X,8A5,I3,1H-,I4,1H$,F9.2,I4)
      IF(NOUT.EQ.5) GO TO 50
      NOUT=5
      GO TO 40
      END
```

(3) The following simple example shows how a Hollerith constant might be used in a CALL statement. The printed line will print LOW, AVRGE, or HIGH, depending upon the income read from an input card.

```
      DIMENSION NAME(8)
      READ(1,10) NAME,INCOME
   10 FORMAT(8A5,10X,I10)
      IF(INCOME.LE.10000) 20,20,30
   20 CALL PRINT(NAME,5HLOW  )
      STOP
   30 IF(INCOME.EQ.20000) 40,40,50
   40 CALL PRINT(NAME,5HAVRGE)
      STOP
   50 CALL PRINT(NAME,5HHIGH )
      STOP
      END
      SUBROUTINE PRINT(NAME,CLASS)
      DIMENSION NAME(8)
      WRITE(3,10) NAME,CLASS
   10 FORMAT(1X,8A5,5X,19HCLASS OF INCOME IS ,A5)
      RETURN
      END
```

8.7. EXTERNAL STATEMENT

Function:

To indicate that a symbolic name (with no arguments) in a list of actual arguments is the name of a subroutine or external function.

EXTERNAL $v_1, v_2, \ldots, v_n$

where: each v is the symbolic name of an external procedure.

Rules:

(1) If an external procedure name is used as an actual argument in a function reference or subroutine call, the name must appear in an EXTERNAL statement of the program unit that contains the function reference or subroutine call.

(2) If a symbolic name appears in an EXTERNAL statement of a program unit, it can appear in a type-statement of the same program unit only if that name is also being used as the function name in a function reference within the same program unit.

(3) If an actual argument is the name of an external procedure, its corresponding dummy argument must be used as the name of an external procedure.

(4) If a symbolic name is being used as the name of an intrinsic function or a statement function within a program unit, it must not appear in an EXTERNAL statement of the same program unit.

(5) The EXTERNAL statement is a specification statement; its position in a program unit is shown in Table 2–2.

Examples:

(1) The following subroutine can be used to print a table of either sine values or cosine values for tenths of a degree, from 0.0 degrees to 90.0 degrees.

```
      SUBROUTINE PRINT(A)
      DOUBLE PRECISION A,ONEDGR,XK,TABLE2(901),TABLE4(901),
     ONEDGR = 3.14159265358979D0/180.0
      DO 10 K=1,901
      XK = K-1
      TABLE2(K) = XK/10.0
   10 TABLE4(K) = A(TABLE2(K)*ONEDGR)
      WRITE(3,20)
   20 FORMAT(1H1,20X,6HTABLES)
      WRITE(3,30) (TABLE2(K),TABLE4(K),K=1,901)
   30 FORMAT(1X,F4.1,1X,D14.7)
      RETURN
      END
```

The calling program contains:

```
      EXTERNAL DSIN,DCOS
      CALL PRINT(DSIN)
      CALL PRINT(DCOS)
      END
```

The first call causes printing of double precision sine values; the second, double precision cosine values.

(2) In the following program unit:

```
      EXTERNAL ABC
      .
      .
      Z = X - SQRT(ABC(D))
```

The use of the EXTERNAL statement is not warranted. It is not the function name ABC that is being passed as an argument, but the value of the function reference ABC(D).

8.8. COMMON STATEMENT

Function:

To permit communication between program units without the use of arguments.

COMMON $/x_1/a_1/x_2/a_2/ \ldots /x_n/a_n$

where: each *x* is either a symbolic name, called a *block name*, or empty (no character or one or more blank characters).
each *a* is a nonempty list of variables, array names, and/or array declarators, with each item in the list separated from the next by a comma.
If x_1 is empty, the first pair of slashes before a_1 is optional.

Operation:

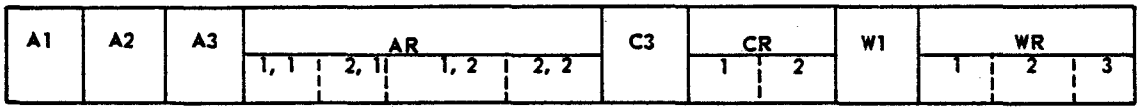
The COMMON statements of a program unit cause the processor to set aside locations for the listed variables and arrays in an area of main storage called the common area. These locations can be shared by the different program units that make up a program. This common area is composed of *blank common* and *labeled common blocks*. If an *x* in the COMMON statement is empty, its associated variables and arrays are stored in the blank common block. If an *x* is a block name, its associated variables and arrays are stored in that block of the common area identified by the block name. The total size (in storage units) of blank common and each labeled common block is determined by COMMON and EQUIVALENCE statements. The order of the locations in blank common and in each labeled common block is determined by the order of their appearance in COMMON statements.

In the following sequence:

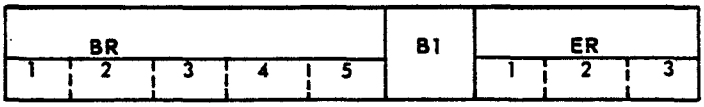
```
      COMMON A1,A2,A3,AR(2,2)/LBL1/BR(5),B1//C3,CR(2)
     1/LBL2/DR(2,2,2,2),D1/LBLL1/ER(3)
      COMMON //W1,WR(3)/LBL2/XR,X2
      DIMENSION XR(3)
```

the common area will consist of:

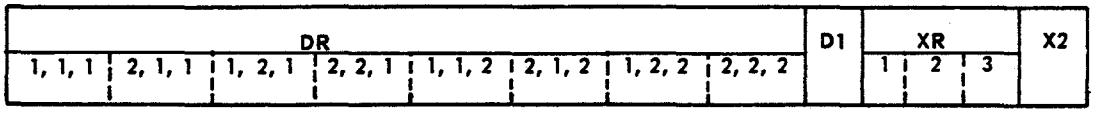
BLANK COMMON



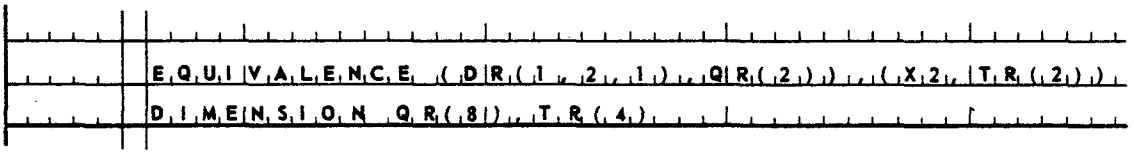
BLOCK LBL1



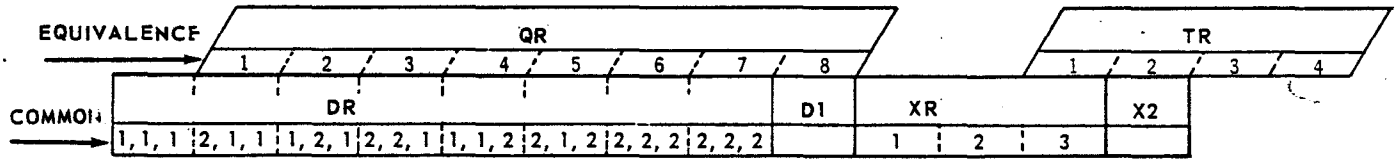
BLOCK LBL2



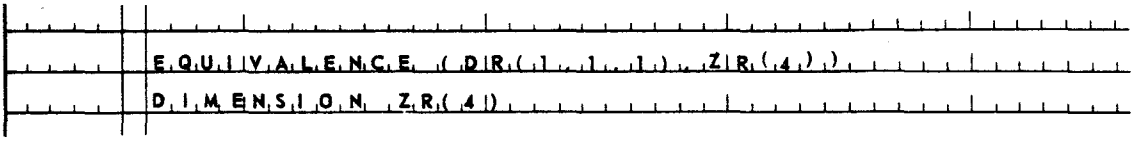
An EQUIVALENCE statement can lengthen a block (blank or labeled); the only lengthening permitted is that which extends a block past the last assignment for that block made directly by a COMMON statement. For example, if the following sequence is added to the previous sequence,



the labeled common block LBL2 is lengthened by two units.



However, in the same program unit, the statements



are illegal, since this is an attempt to extend a common block ahead of its first assignment for that block, DR.

In each program unit, the size (in storage units) of a labeled block with the same block name must be the same, if it is present in a COMMON statement. The size of blank common need not be the same in all storage units. The total size of blank common is equal to the largest number of storage units assigned to blank common by a program unit of the program. However, association of a common variable or array is by storage unit in a labeled block of the same name or in blank common. The *n*th storage of blank common in one storage unit is shared with the *n*th storage unit of blank common in any other program unit. The *n*th storage unit of a labeled common block with a given name in one program unit is shared with the *n*th storage unit of a labeled common block bearing the same block name in any other program unit.

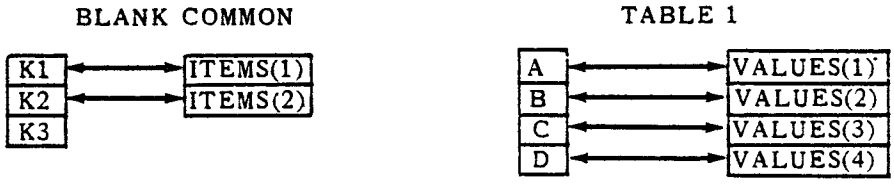
For example, if one program unit contains

```
      COMMON /TABLE1/A,B,C,D//K1,K2,K3
```

and a second program unit contains

```
      COMMON ITEMS(2)/TABLE1/VALUES(4)
```

the correspondence would be:



This means that the variable A of the first program unit is the same value as the array element VALUES(1) in the second program unit. Note that the second program unit cannot access K3 of the first program unit because of the difference in blank common size. However, KE of the first program unit could correspond to a blank common item in still another program unit of the same program.

Since correspondence is by storage unit:

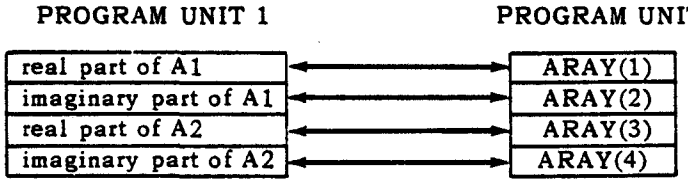
If program unit 1 contains

```
      COMMON A1, A2
      COMPLEX A1, A2
```

and program unit 2 contains

```
      COMMON ARAY(4)
```

the following correspondence occurs:



Rules:

(1) A COMMON statement is a specification statement; its order within a program unit is shown in Table 2–2.

(2) An array declarator in a COMMON statement must not contain dummy arguments.

(3) A block name is not related to any variable or array in the same or any other program unit; it has no data type or value associated with it. Once a symbolic name is used as a block name, no other name can be used to identify the same block throughout the entire program. However, the same symbolic name can be used as an array name, variable, or statement function name in the same program unit or any other program unit of the program.

(4) The same block name can occur more than once in a COMMON statement or in more than one COMMON statement of a program unit. All entities of the same block (blank or labeled) are stored consecutively, in the order of their appearance.

(5) An EQUIVALENCE statement may extend a block only past its end, not ahead of its beginning.

(6) The size, in storage units (see Table 2–4), of a common block for a program unit is the sum of the storage required for the elements introduced through COMMON and EQUIVALENCE statements. The sizes of labeled common blocks with the same block name must be the same in all program units of a program; the sizes of blank common in all the program units need not be the same.

(7) It is incorrect to equivalence two entities of a COMMON statement to each other, either directly or indirectly.

(8) Any program unit requiring access to a common block must have a COMMON statement.

(9) In a subprogram, a symbolic name appearing in a COMMON statement may not identify an adjustable array.

(10) Correspondence between different entities of the same common block in different program units is determined by order and by storage unit(s).

(11) In any procedure subprogram, symbolic names of dummy arguments must not appear in a COMMON statement.

(12) An item in blank common must not appear in a DATA statement; an item in labeled common may appear in a DATA statement.

Examples:

(1) A subroutine subprogram contains:

```
      SUBROUTINE CLARAY
      COMMON /ARRAYS/ SET(100)
      DO 10 K=1,100
   10 SET(K) = 0.0
      RETURN
      END
```

Program unit 2 contains:

```
      .
      .
      .
      COMMON /ARRAYS/ TABLE1(2,5),TABLE2(3,4,5),TABLE3(3,0)
      .
      .
      .
      CALL CLARAY
      .
      .
      .
      DO 10 K=1,10
   10 TABLE1(K) = 0.0
```

The CALL CLARAY statement in program unit 2 causes all the arrays in the labeled common block ARRAYS to have their elements cleared to 0.0. The DO range in program unit 2 causes clearing of all elements in the array TABLE1.

(2) If program unit 1 contains

```
      COMMON A,B(5),C,MTRX(3,4,5)
      LOGICAL A
      DOUBLE PRECISION D,B
      COMPLEX C
```

and program unit 2 will only access MTRX(3,4,5), it must provide space in its COMMON statement for those items in blank common that precede MTRX, even though they will not be used in program unit 2. For example:

```
      COMMON DUMMY(113),MARRAY(3,4,5)
```

9. INITIALIZATION

9.1. GENERAL

This section describes the DATA statement (data initialization statement) and the block data subprogram (specification subprogram).

9.2. DATA STATEMENT

Function:

To initialize the values of specified variables and/or array elements at compilation time.

DATA $k_1/d_1/,k_2/d_2/, \ldots,k_n/d_n/$

where: each k is a list of variables and/or array elements separated by commas. each d is a list of constants and optionally signed constants separated by commas, any of which may be preceded by j*, where j is an integer constant indicating repetition of a constant j times.

Rules:

(1) Dummy arguments may not appear in a DATA statement.

(2) Each subscript expression in an array element reference must be an integer constant.

(3) A Hollerith constant may be used in a list d.

(4) There must be a one-to-one correspondence between the list-specified items and the constants.

(5) An initially defined variable or array element must not be in blank common. An item in a labeled common block can only be initialized if the DATA statement is in a block data subprogram.

(6) No item can be initialized more than once in the entire program.

(7) Array names may not appear in DATA statements. To initialize an entire array, each array element must be listed separately.

(8) A program unit may contain as many DATA statements as required.

Example:

```
DIMENSION M(3)
DATA A,B/2*0.0/,M(2),M(3),M(1)/2*0,1H /
```

The values will be initialized as follows:

A to 0.0
B to 0.0
MATRIX(1) to all blank characters
MATRIX(2) to 0
MATRIX(3) to 0

9.3. BLOCK DATA SUBPROGRAM

Function:

To initialize values of labeled common blocks, at compilation time.

```
BLOCK DATA
    .
    .
    .
data initialization and specification
statements (except EXTERNAL)
    .
    .
    .
END
```

Rules:

(1) The first statement of a block data subprogram must be the BLOCK DATA statement.

(2) The body of the block data subprogram consists of one or more DATA statements, one or more COMMON statements, and all other required specification statements except the EXTERNAL statement. These specification statements are type-statements, DIMENSION, and EQUIVALENCE statements.

(3) All specification statements must precede the DATA statement(s).

(4) The last line of the subprogram must be an end line.

(5) The block data subprogram, although independently compiled, is a *specification subprogram* and should not be confused with an external procedure.

(6) There may be as many block data subprograms as required.

(7) If any entity of a common block is initialized in a block data subprogram, all items of that block must have their required specification statements, even though some of these items do not appear in a DATA statement.

(8) Initial values may be entered into more than one common block in a single block data subprogram.

(9) A block data subprogram cannot be referenced in a program.

(10) No executable instructions are generated by the processor for a block data subprogram. Therefore, execution time of a program is decreased and less storage space is required for execution.

Example:

```
BLOCK DATA
INTEGER B2, B4
DOUBLE PRECISION E
LOGICAL LR
COMMON /BLK1/ LR(2) /BLK2/ B2, B4(2), E
DATA LR(1), LR(2) /2*.TRUE./, B2/-1/, E/2.718281828D0/
END
```

9.3.1. BLOCK DATA Statement

Function:

To identify a block data (specification) subprogram.

BLOCK DATA

Rule:

The BLOCK DATA statement must (and can only) appear as the first line of a specification subprogram.

APPENDIX A. DIFFERENCES BETWEEN ANSI FORTRAN AND ANSI BASIC FORTRAN

ANSI (American National Standards Institute, Inc.) Basic FORTRAN lacks some of the features found in ANSI FORTRAN. The following is a list of the features which Basic FORTRAN lacks, and the sections of this manual affected by the differences:

(1) There is no DATA statement or specification (block data) subprogram (2.2.1, 2.2.2, Section 9).

(2) There is no explicit type declaration of any kind; there are no type-statements (2.2.2, 2.7.2.1, 7.2).

(3) There is no EXTERNAL statement (2.2.2, 7.1, 8.7).

(4) There are no ASSIGN or assigned GO TO statements (2.2.2, 4.1, 5.2.3, 5.2.3.1).

(5) There is no logical assignment statement (2.2.2, 4.1, 4.3).

(6) All DIMENSION statements must precede all COMMON statements; all COMMON statements must precede all EQUIVALENCE statements (2.2.2).

(7) There is no $ character in the Basic FORTRAN character set (2.3).

(8) There is a maximum of five continuation lines instead of 19 (2.4.3).

(9) There is a maximum of four digits in a statement label instead of five (2.4.4).

(10) A real constant may not be written as an integer constant followed by a decimal exponent (2.6.2).

(11) Hollerith data is permitted only in a FORMAT statement; there are no Hollerith constants and no A field descriptor (2.5, 6.3.3, 6.3.3.4).

(12) There is a maximum of five characters in a symbolic name instead of six (2.7).

(13) There are no double precision, complex, or logical data types (2.7.2).

(14) There is no provision for type declaration in a FUNCTION statement (2.7.2.1, 7.2, 8.5.2.1).

(15) Arrays are limited to two dimensions instead of three (2.7.4).

(16) There are no adjustable arrays and no array declaration in COMMON statement (2.7.4.1, 8.8).

(17) There are no relational or logical expressions (Section 3).

(18) There are no double precision or complex type arithmetic expressions (3.2.3).

(19) There is no logical IF statement (5.3).

(20) There is no provision for extended range in a DO loop (5.4).

(21) There is a maximum of four instead of five octal digits in a PAUSE statement (5.6.1).

(22) There is no form control character for formatted output records (6.3).

(23) There are no D, G, or L field descriptors (6.3.3).

(24) There is no provision in FORMAT statement for scale factor (6.3.3), data exponent on input for F field descriptor (6.3.3.2.2), or second level of parentheses (6.3.3.5, 6.6).

(25) In numeric fields, blanks are permitted only to the left of the first nonblank character and between the sign of the field and the next nonblank character (6.3.3.2).

(26) In formatted READ and WRITE statements, *f* must be the statement label of the FORMAT statement; no array name is permitted (6.4, 6.5).

(27) There are no logical statement functions (8.2).

(28) Basic FORTRAN provides only the following intrinsic functions: ABS, IABS, FLOAT, IFIX, SIGN, and ISIGN (8.3).

(29) Basic FORTRAN provides only the following basic external functions: EXP, ALOG, SIN, COS, TANH, SQRT, and ATAN (8.5.1).

(30) Function subprograms may not define or redefine any of their arguments or alter any entity in common or entity associated with common by an EQUIVALENCE statement (8.5.2, 8.8).

(31) There is no provision for labeled common blocks (8.8).

INDEX

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

_____

(Document Title)

_____ _____ _____

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

Cut along line.

From:

_____

(Name of User)

_____

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

CUT

FOLD