CHAPTER 2

FUNDAMENTALS OF PROGRAMMING LANGUAGES

This chapter reviews basic concepts of programming. It introduces important terms dealing with how computers work, how programs are structured, and how data is organized and manipulated. It also surveys various types of programming environments, discussing their relative advantages and disadvantages. In particular, it summarizes the dialect of FORTRAN '77 used to encode the various programs and utilities described in this book.

2.1 THE PROGRAMMING ENVIRONMENT

Figure 2-1: Flow of information in a comprehensive programming environment - Upper case designations indicate precompiled utilities; lower case designations indicate user programs after various

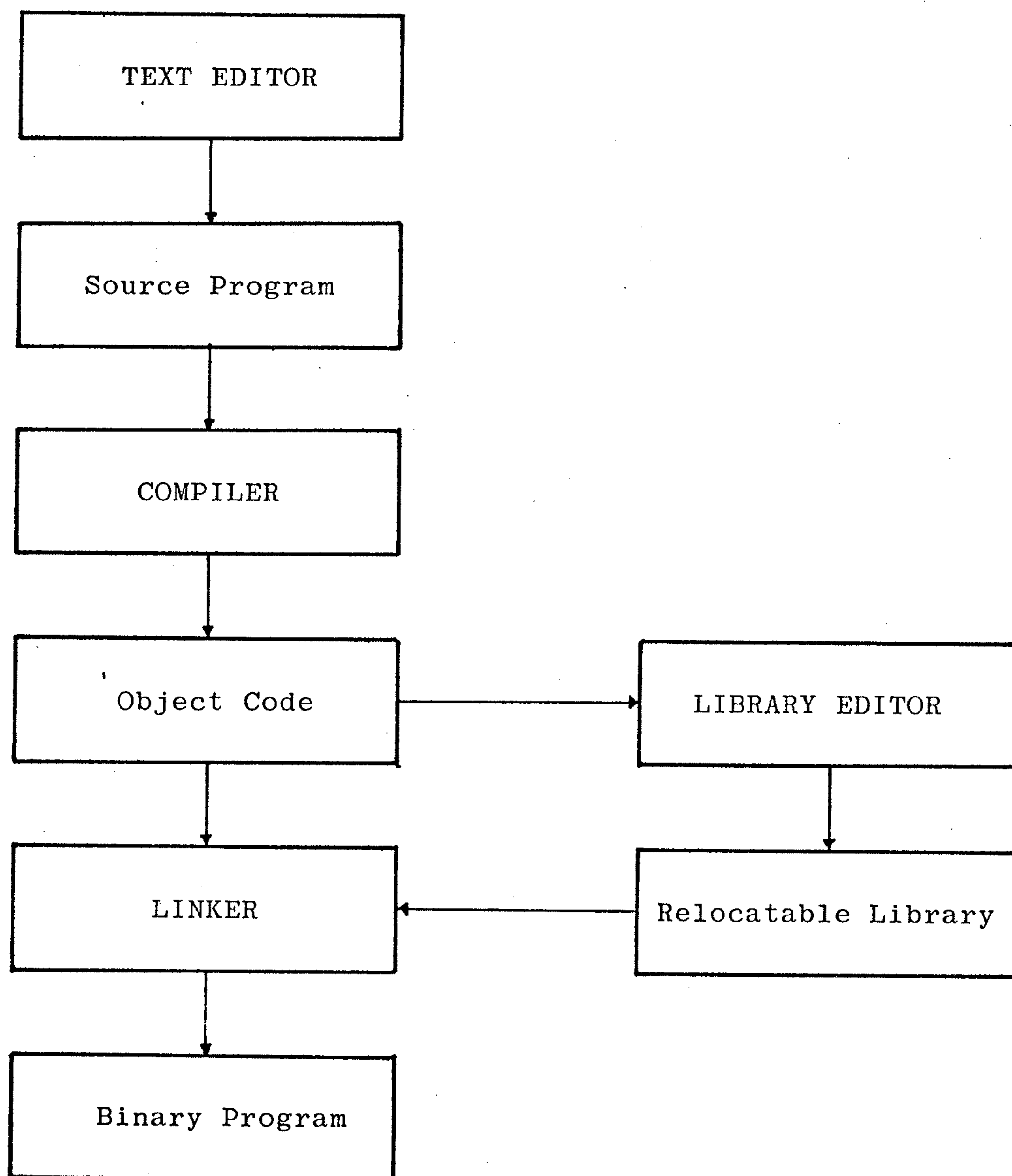


Fig. 2-1

stages of processing.

Figure 2-1 charts the steps involved in transforming a user program from a file of statements encoded in a language understandable to a programmer into a file of binary instructions executable by a computer. A complete programming environment includes at least four basic utilities which for many computing systems are supplied by the manufacturer: a text editor, a compiler (or assembler), a library editor, and a linker. The heart of any programming environment is the compiler, which accepts a file of statements written in a language such as FORTRAN and translates these statements into a new file containing binary instructions. The file accepted by the compiler is conventionally referred to as the source file, while the file of binary instructions is called the object file. The text editor enables the programmer to create and modify source files interactively at a terminal and to store these source files on a "mass storage" device such as a magnetic disk. Programs may be conceived for a specific purpose or may alternately be generalized subroutines useful under a wide variety of circumstances; in the latter case, it is often convenient to avoid recompiling a subroutine for each new application by storing the object code in a relocatable library. The library editor enables a programmer to insert

new object files into such a library and to delete old subroutines which no longer serve a purpose.

Typically, the compiler itself will delegate many generic procedures such as input/output activities and mathematical functions to precompiled library entries. To run a program, it is necessary both to convert the source file into object form and to 'borrow' from one or more relocatable libraries any subroutines which are not explicitly encoded in the source file. The <u>linker</u> satisfies these 'dangling' subroutine calls by joining object file with library entries into a binary program. This binary program -- itself be stored as a file on a mass storage device -- may then be loaded into the computer's random-access memory as direct instructions to the processor.

2.2 ASSEMBLERS

The simplest languages are <u>assembly</u> codes, which first appeared in the early 1950's. Assembly codes are so rudimentary that many programmers prefer not to dignify them as "languages". The program which translates an assembly code into direct binary instructions is called an <u>assembler</u>. Assemblers are

machine-dependent. The codes include mnemonics to represent each operation a machine is capable of performing, and symbols to represent locations in memory. Symbols for locations containing instructions to the computer are called labels, while symbols for locations containing data are called variables or, when appropriate, constants.

The following program illustrates the mnemonics and symbols used for an assembler. It compares the contents of locations X and Y and stores the smaller value in location Z. Notice the difference in function between labels and symbols. Labels appear as operands to branching instructions such as BR (branch unconditionally) and BGT (branch if greater than). Normally, the computer executes instructions sequentially as the eye moves down the page; a branching instruction causes the computer to transfer control from one part of a program to another. By contrast, symbols appear as operands to instructions such as CMP (compare two numbers), MOV (transfer a value from one location to another), addition, subtraction, multiplication, and so on.

-- Programming example 2-1 --

Assemblers are ideal for programs such as operating systems, which are heavily used and which must run very quickly. Since the code directly reflects the architecture of a particular

Ex 2-1

```
Label      Operation   Operands   Comment
           CMP         X,Y        ;Compare contents of X to contents of Y
           BGT         LABEL1     ;Branch to LABEL1 if positive difference
           MOV         Y,Z        ;Set contents of Z to contents of Y
           BR          LABEL2     ;Branch unconditionally to LABEL2
LABEL1:    MOV         X,Z        ;Set contents of Z to contents of X
LABEL2:    NOP                    ;No operation
```

Ex 2-2

```
MOV     X,Y     ;Move contents of X to Y
MUL     S,Y     ;Multiply Y by S
ADD     B,Y     ;Add B to product in Y
```

Ex 2-3

```
Y = S * X + B
```

machine, it allows programmers to optimize the size and speed of a program. However, the sheer detail of assembly-code programs often makes them a nightmare to create or revise.

Since very early on, assemblers have included features allowing programmers to define macros. A macro consists of several instructions in a predefined sequence which the programmer may invoke using a single "pseudo-instruction".

2.3 COMPILERS

The earliest compiler was FORTRAN (Backus, et al, 1957), which stands for "FORmula TRANslation". FORTRAN was first introduced in 1956. Where assemblers require a programmer to code his program entirely in simple instructions, compilers eliminate much drudgery by allowing a programer to use expressions. The following example illustrates that it requires only one FORTRAN expression to encode a formula which would have required three statements in assembly language:

-- Programming example 2-2 --

Another difference between assemblers and compilers is that

compilers are (properly) *machine independent*. The operations represented in compiled languages are generic: "assign a value to a variable", "jump to a label"; the compiler itself decides which specific instructions are most appropriate to each operation. Obviously, compilers represent an extreme increase in convenience to the programmer. In payment there is a noticeable decrease in efficiency; even with modern "optimizing" compilers, compiled programs tend run at about half the speed and require about one and a half times the memory used by assembled programs.

2.3.1 Data Structures

Compilers allow programmers to designate both *types* of variables and levels of *precision*. The precision indicates how much memory is allocated to a variable. The smallest unit of memory is the *bit*, or binary digit. Precisions are usually expressed in *bytes*, which consist of eight bits on most computers, six bits on a few older designs. There are four fundamental types of variable:

1. *Integers* are numbers without fractional parts.
   Integers are the numbers which computers are most adept

at manipulating. Each integer is stored as a one-bit sign and a magnitude whose maximum size depends on the precision of the variable.

2. Real or (properly) floating-point numbers have fractional parts, though fractional parts of zero are possible. Real numbers are stored as a sign, an exponent, and a mantissa. The mantissa gives the "most significant" digits, while the exponent places the decimal point. Usually the range of the exponent stays fixed, so that increased precision indicates a longer mantissa.

3. Characters include all of the symbols on a terminal keyboard, plus a number of special codes ("control-C", for example). Characters are usually stored as simple magnitudes in single bytes.

4. Logical or Boolean variables can take on two values: true (-1) or false (0). Expressions involving logical variables typically appear in statements affecting transfer of control. Logical variables are typically stored with single-byte precision, but their only significant content is their sign.

Variables are often declared and used singly; however, they are almost as often organized into constructs called both information structures and data structures. The simplest data structure is the array. Arrays consist of many variables, individually designated as elements, which are stored consecutively in memory. The number of elements in an array constitutes its dimension. In order to access a particular element, the programmer must specify both the array name and an index. The array name is a symbol representing the location in memory containing the first element of the array, and the index allows the program to compute an "offset" relative to this location. Arrays of characters are often referred to as strings.

All languages from assemblers on include features for defining one-dimensional arrays of variables. Newer languages attempt an additional increase in sophistication by including facilities for advanced data structures such as multi-dimensional arrays, linked lists, "files", and trees. The advanced data structures cited in this paragraph will be introduced as necessary in this book. Such facilities accomplish implicitly what programmers working in languages such as FORTRAN have had to code explicitly. In payment, a programmer must put up with a further reduction in computational efficiency. Sometimes an implicit data structure will in fact be limiting; for example, those languages which some computer scientists incorrectly call

"naturally recursive" due to their implementation around an implicit "stack" (included are ALGOL, PASCAL, and LISP) preclude varieties of recursive programming such as constrained search (Chapter 14), which uses a "queue".

2.3.2 Flow of Control

In order to transfer control from one part of an early FORTRAN program to another, the programmer defined labels similar in function to those provided by assemblers. The assembly-code program for setting the Z to the smaller of X and Y translates into FORTRAN as follows:

-- Programming example 2-3 --

Block structures enable a programmer to specify flow of control without defining labels. They contribute little to the efficiency of a program, but do allow for code which is much more readable and amenable to revision than label-oriented code such as the above excerpt. Using a language with block-structures such as FORTRAN '77, a programmer might set Z to the smaller of X and Y as follows:

Ex 2-3

```
      if (X.gt.Y) go to 10
      Z = X
      go to 20
   10 Z = Y
   20 continue
```

Ex 2-4

```
      if (X.gt.Y) then
        Z = Y
      else
        Z = X
      end if
```

Ex 2-5

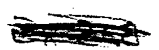
```
      Y = 1.0
   10 if (Y.ge.X) go to 20
      Y = Y + Y
      go to 10
   20 continue
```

Ex 2-6

```
    Y = 1.0
    do
      if (Y.ge.X) exit
      Y = Y + Y
    repeat
```

Ex 2-7

```
    do 43 IDX=1,LIMIT
      if (X.le.VALUE(IDX)) go to 44
 43 continue
 44 if (IDX.gt.LIMIT) stop 'Array overrun.'
```



```
    do (IDX=1,LIMIT)
      if (X.le.VALUE(IDX)) exit
    repeat
    if (IDX.gt.LIMIT) stop 'Array overrun.'
```

-- Programming example 2-4 --

Loops are fundamental constructs for performing repeated computations. Using older FORTRAN's, programmers need to specify labels for loops. For example, the following routine compares increasing powers of two with the variable X until it finds a power which just exceeds X:

-- Programming example 2-5 --

The dialect of FORTRAN '77 used in this book provides a block-structured construct which allows a programmer to dispense with labels:

-- Programming example 2-6 --

Many loops depend on a special index. For example, the following excerpt from a program locates the first element of array VALUE which is smaller than the variable X. It is presented in two versions, the first using the labeled DO construct standard to FORTRAN IV and FORTRAN '77, the second using a block-structured DO:

-- Programming example 2-7 --

Each time either version of the program reaches the bottom of the loop, (the CONTINUE statement in the first version and the REPEAT statement in the second) it increments the variable IDX by 1 and compares the result to LIMIT. As long as IDX does not exceed LIMIT, the program keeps looping. Notice, however, that if the program does manage to leave the loop through the bottom, the value of IDX upon leaving the loop becomes LIMIT+1. This fact allows the program to insure that it has indeed located a legitimate element of array VALUE by testing that IDX does not exceed LIMIT (note 1).

The first language to incorporate block structures, ALGOL, was first introduced in 1958. ALGOL stands for "ALGOrithmic Language". Though ALGOL is no longer widely used, it has found popular successors in PASCAL (Jensen and Wirth, 1978), a pedagogic language introduced in 1970, and C (Kerninghan and Ritchie, 1978), an efficient and highly practical language introduced in 1975. The attractions of block structures led in 1977 to establishment of a new standard of FORTRAN incorporating them to a limited extent (Wagener, 1980), as well as several more complete variants (for example, the dialect used here).

2.3.3 Subroutines

One technique of programming which is greatly facilitated by compilers is the use of subroutines (also called subprograms or procedures). Subroutines are useful when a program repeats similar tasks in multiple contexts. They allow a program to transfer control to an independent block of code (the subroutine); after performing its appointed task, the subroutine then directs the main program to resume wherever it left off. We say that the main program calls the subroutine, while the subroutine returns to the main program. The advantages of subroutines are that they conserve memory by eliminating redundant code (unlike the macros of assembly codes) and that they help resolve complicated routines into more easily understood modules. The latter immensely increases convenience in "debugging" (note 2).

Such modular programming is especially convenient in languages such as FORTRAN and the descendents of ALGOL, which allow programmers to distinguish between local and global (common) variables. In FORTRAN any variable which is neither explicitly passed in a subroutine call nor explicitly designated as common to both calling routine and subroutine will be local: operations performed on such a variable by the subroutine will have no effect upon similarly named variables in

the calling program, and vice versa.

Since tasks are likely to vary from situation to situation, it is usually necessary to have the main program pass parameters (also called arguments or dummy variables; the last term should be avoided) which affect how the subroutine works but which may change from one call to the next. Since subroutines must be capable of processing not only simple variables, but often entire arrays, it is conventional for the main program to pass the location of each parameter to the subroutine and to let the subroutine interpret these locations as it sees fit. This convention allows subroutines to actively manipulate the values stored in these locations. For example, after the following call to subroutine NULL from program MAIN:

-- Programming example 2-8 --

will set location 4 of ARRAY to zero. The size of array ARG declared in subroutine NULL is meaningless because the array has already been dimensioned in the main program; the DIMENSION statement serves simply to inform the compiler that ARG itself is an array.

FORTRAN seems unique among compiled languages in its implementation of common memory declarations. Blocks of common memory afford an additional means of passing information between

Ex 2-8

```
program MAIN
dimension ARRAY(5)
data ARRAY/1.0,1.0,1.0,1.0,1.0/
...
call NULL(ARRAY(3))
...
stop
end

subroutine NULL(ARG)
dimension ARG(1)
ARG(2) = 0.0
return
end
```

a main program and its various subroutines. Languages descending from ALGOL compensate by allowing for subroutines which can allocate memory _dynamically_ as temporary workspace to be retrieved for other purposes when the subroutine completes its tasks; in FORTRAN subroutines, such workspace must be allocated explicitly by the main program and passed in one or more arguments.

Compilers often include a special kind of subroutine called a _function_. Calls to functions are encoded directly into expressions involving the four fundamental types of data described previously. Each call returns a single item of data which assumes the role of a normal variable or array reference in an expression. Also related to subroutines are _coroutines_. Coroutines are used in real-time programming to implement tasks such as input and output of data which may performed "in parallel" (simultaneously) with the calling program.

2.4 INTERPRETERS

An alternative to both the assembler and the compiler is the _interpreter_. Interpreters are designed to "interact" with a programmer so that he may make small changes in his program and

test them instantly, without having to invoke a compiler or assembler. Interpreters reside in memory with a program and compile it line-by-line as the program executes. This means that interpreted programs are the slowest programs around (much too slow for many of the applications described in the latter part of this book). They must also yield large amounts of memory to the resident interpreter. The most widely known interpreters are BASIC and LISP.

BASIC was introduced by John Kemeny and Thomas Kurtz in 1963. Though intended for pedagogic use (the name is an acronym for "Beginner's All-purpose Symbolic Instruction Code"), BASIC has proliferated among users of personal computers. What convenience BASIC gains through its interpretive implementation is more than offset by the tedium of its format: each BASIC statement has its own numeric label, and each label must occur in order. This feature renders it difficult to insert new blocks of code. BASIC provides only the most rudimentary facility for subroutines, with no isolation of symbols.

LISP (McCarthy, et al, 1965), introduced in 1965, stands for "LISt Processor". It includes implicit facilities for certain kinds of linked lists and for stack-oriented recursion. In strong contrast to BASIC, flow of control in LISP is acheived exclusively through block structures. This feature makes LISP programs much more malleable than BASIC programs. LISP is highly

regarded in artificial intelligence circles; however, at least one respected author -- Donald Knuth -- treats LISP with disdain (Fundamental Algorithms, 1973, p. 229).

2.5 PSEUDO-COMPILERS

An intermediate step between the compiler and the interpreter is the pseudo-compiler. Unlike true compilers, which translate programs directly into machine instructions, pseudo-compilers translate programs into "object code". The translation parses expressions into simple instructions, but a the computer still requires a resident interpreter to make sense of the result. SNOBOL (Griswold, et al, 1968), introduced in 1962 but now largely out of favor, is pseudo-compiled. Pseudo-compilers are easier to implement than true compilers; many of the BASIC and PASCAL "compilers" which have been developed for personal computers are actually pseudo-compilers.

Since the pseudo-compiler eliminates many of the functions exercised by a full-fledged interpreter, pseudo-compiled programs are much faster than interpreted ones. Since the interpeter is smaller, they yield less memory. However, pseudo-compiled programs still take six or more times as long as assembled ones

and still take up much more space. Pseudo-compilers are no more interactive than true compilers.

2.6 NOTES

1. The DO-REPEAT-EXIT construct is not standard to FORTRAN '77 compilers; however, it appears in this book since it makes programs much more readable. A program for converting DO-REPEAT-EXIT constructs into conventional FORTRAN '77 appears in Wagener, 1980, page 346.

2. The term "bug" originated with the Mark I computer, a programmable, electro-mechanical calculating machine developed by Howard Aiken at Harvard University and used by the U.S. Navy to compute ordinance tables. One day in August, 1945, a two-inch moth became entangled in the machinery and caused the Mark I to malfunction. According to Commodore Grace Hopper, then a lieutenennt in Aiken's staff: "From then on, when anything went wrong with a computer, we said it had bugs in it." (Time Magazine, April 16, 1984).