1 THE BEGINNING: PSEUDO-CODE INTERPRETERS

1.1 HISTORY AND MOTIVATION

Why Study a Primitive Language?

In this chapter we investigate a very primitive programming language, so primitive in fact that you would never want to program in it. Furthermore, it's not even a "real" language, that is, it was made up specifically for this chapter. What is the purpose? Just as an architecture course might begin with an analysis of the Parthenon, or an automotive design course with the Model T, so we may benefit by beginning the study of language design in a context in which the issues are salient. Having honed our skills here, we will be better able to apply them to more modern, sophisticated, and complex languages.

Programming Is Difficult

Almost as soon as the first computers were built, it became obvious that programming was very difficult; this fact has not changed. Indeed, the tasks we have attempted to accomplish with computers have grown rapidly in ambitiousness and size. Much of the difficulty of programming stems from *complexity*, the necessity of dealing with many different details at one time. Programs may contain hundreds of thousands, or even millions, of lines of code. Considering these lines (together with the operators and operands that make them up) as parts that must be assembled individually to make a program work correctly leads to the conclusion that programs are some of the most complicated objects ever constructed. One of the primary tasks of programming languages is the conquest of this complexity.

Programming Early Computers Was Especially Difficult

Although the problems addressed on early computers were smaller than many of those now addressed, programming was still very difficult. Part of the reason was that early computers had very little storage; a few thousand words were considered a large memory. Thus, com-

pact code was a necessity. Also, by modern standards the early computers were very slow, so it was important that programs be coded very efficiently. Finally, early computers were more complicated to program than the ones with which we are now familiar.

As an example of these complications, some of the drum computers (which stored both the data and the program on a rotating magnetic drum) had a *four-address* instruction code, which means that each instruction contained the address of the next instruction to execute. This permitted a process called *optimal coding*, which means that when programmers coded, for example, an ADD instruction, they would determine how far the drum had rotated while that ADD instruction was being executed. They would then use that drum location for the next instruction after the ADD, placing its address in the ADD instruction. In this way the next instruction was always under the drum head when it was ready to be executed, thus saving wasted drum revolutions and greatly increasing the speed of the program (Figure 1.1). Aside from the difficulty of doing the calculations, there were always complications. For example, the optimal location for the next instruction might already be occupied, in which case the next available location in that track had to be used. Figure 1.2 shows a small part of a program written in the mid-1950s for the IBM 650. Notice that each instruction contains in its rightmost field (INST) the address of the next instruction (LOC).[1]

Needless to say, programming these machines was a tedious and error-prone process. Much of it was done without the aid of any software tools, including assemblers. There were other complications in programming these machines. For example, since the bits of an instruction often directly controlled the opening and closing of gates in the central processor, the codes used for various operations appeared to the programmer to follow no simple rule. This apparent irregularity made them very difficult to remember.

Many Program Design Notations Were Developed

The complexity of programming led to the development of *program design notations*, the precursors of programming languages. One of the earliest of these was von Neumann and Goldstine's *flow diagrams*, which developed into the flowcharts that are still often used during program design. Throughout the world, many different notations and languages were developed to try to conquer the complexity of programming. Some of these helped the programmer to design the memory layout and control flow of the program without being concerned with details (such as optimal coding). Others provided mnemonics for the ma-

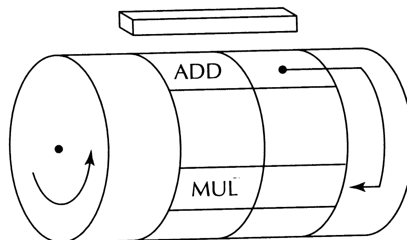


Figure 1.1 Optimal Coding

[1] The instructions are shown in the usual order of execution; on the drum they would be placed in the address given in LOC.

LOC	OP	DATA	INST	COMMENTS
1107	46	1112	1061	Shall the loop box be used?
1061	30	0003	1019	
1019	20	1023	1026	Store C.
1026	60	8003	1033	
1033	30	0003	1041	
1041	20	1045	1048	Store B.
1048	60	8003	1105	
1105	30	0003	1063	
1063	44	1067	1076	Is an 02-operation called for?
1067	10	1020	8003	
8003	69	8002	1061	Go to an 01-subroutine.

Figure 1.2 Part of an IBM 650 Program

chine operations, much like an assembly language. Konrad Zuse, in Germany, developed a sophisticated programming notation that included a data structure definition facility. He even outlined methods for compiling this notation in 1945! By and large, however, these languages and notations were intended for use by *people* during the design process, not for direct processing by the *computer*. The actual coding process was still done in numeric codes or with the aid of simple "assembly programs" (assemblers).

Floating Point and Indexing Were Simulated

The earliest computers did not have built-in floating-point operations; these did not appear until the IBM 704 in 1953. On the other hand, the primary application of many of these machines was in scientific and numerical computations, which require numbers of a wide range of magnitudes. This necessitated *manual scaling*, a technique in which numbers were multiplied by scale factors in order to keep them within the range of the integer arithmetic facilities of the computer. This was a very complicated process, which required a detailed analysis of the algorithm. The difficulty of manual scaling led to the development of floating-point subroutines, that is, of subroutines for performing basic floating-point operations (addition, subtraction, multiplication, division, square root, etc.). Although these often slowed down a program by at least an order of magnitude, they so simplified the programming of numerical problems that they were widely used.

Another facility missing from many early computers was *indexing*, the ability to add a variable index quantity to a fixed address in order to address an element of an array. You are probably aware from your own programming experience that the array is one of the most common data structures; it was even more common in the scientific and numerical problems that dominated the use of early computers. One of the important ideas of von Neumann, and one of the distinguishing characteristics of a *von Neumann machine* (which includes most computers), is that the program and data are stored in the same memory. Therefore, it is possible for a program to modify itself or another program as though it were data. Since most early computers did not have index registers or indexed addressing modes as do modern computers, it was necessary to accomplish indexing through *address modification*. That is, the

program would add the index value to the address part of a data accessing instruction. Needless to say, this was an error-prone process. It also consumed much of the scarce memory with this address modification code. For this reason, it was also common to use subroutines to perform indexing. You can probably imagine that since floating-point operations and indexing account for much of what is done in numerical algorithms, most of the actual execution time was spent inside the floating-point and indexing subroutines. This justified the use of pseudo-code interpreters, which we will discuss next.

Pseudo-Code Interpreters Were Invented

It was quickly recognized that consistent use of the floating-point and indexing subroutines simplified the programming process; it allowed one to program as though these facilities were provided by the hardware of the computer. This led to the idea of a *pseudo-code*, that is, an instruction code that is different from that provided by the machine—and presumably better.[2] Since the program was going to spend most of its time in the floating-point and indexing subroutines anyway, why not simplify programming by providing an entire new instruction code that was easier to use than the machine's own? This idea was first described in the famous Appendix D of the first programming book, *The Preparation of Programs for an Electronic Digital Computer*, written by Wilkes, Wheeler, and Gill in 1951. This appendix described the design of a simple pseudo-code and the design of an "interpretive subroutine" for executing that pseudo-code—what we now call an *interpreter.* There is some indication that the authors did not realize the full significance of what they were describing, otherwise they wouldn't have buried it in Appendix D. They present interpreters primarily as a means of saving memory since the pseudo-code is more compact than the machine's real instruction code. Other programmers soon grasped the importance of this idea and many pseudo-code interpreters were born. Later in this chapter, we will investigate the design and implementation of one of these.

The significance of these pseudo-code interpreters is that they implemented a *virtual computer* with its own set of data types (e.g., floating point) and operations (e.g., indexing) in terms of the *real computer* with its own data types and operations. One advantage of the virtual computer over the real computer was that it was *higher level*; that is, it provided facilities more suitable to the applications and it eliminated many details from programming. This is an example of the Automation Principle.

The Automation Principle

Automate mechanical, tedious, or error-prone activities.

The virtual computer was also more *regular*, that is, simpler to understand through the absence of special cases, which is summarized in the Regularity Principle.

---

[2] The term *pseudo-code* is often now used for informal program design notations, which are not intended to be executable by a computer (hence, *pseudo*). Here however *pseudo-code* is used in its original sense: a primitive, interpreted programming language.

The Regularity Principle

Regular rules, without exceptions, are easier to learn, use, describe, and implement.

We will see that all programming languages can be viewed as defining a virtual computer that is intended to be better in some respects than the real computer. Although the interpreter usually ran at least an order of magnitude slower than the real computer, most of this time was spent in the floating-point and indexing routines that were considered necessary anyway. Thus, pseudo-code interpreters were a valuable programming aid that imposed little additional cost.

Compiling Routines Were Also Used

At this same time, another approach was being used for implementing pseudo-codes—*compiling routines.* Grace Murray Hopper and others began to use programs to extract subroutines from libraries and combine them (a process called compiling) under the direction of a pseudo-code. Since this process was done once, at compilation time, it did not involve the overhead that resulted from interpretation. Therefore, the compiled program could run considerably faster than an interpreted program. Perhaps because this approach did not encourage programmers to look at a pseudo-code as a virtual computer, it did not produce pseudo-codes that were as regular. For this reason we will concentrate on interpreters in this chapter, and return to compiling routines, which are now called *compilers*, when we discuss FORTRAN.

1.2 DESIGN OF A PSEUDO-CODE

Basic Capabilities Must Be Decided

In this section we design a pseudo-code. It is similar to real pseudo-codes, the languages L_1 and L_2 designed by Bell Labs for the IBM 650 in 1955 and 1956. In Section 1.3 we discuss the design of an interpreter for this pseudo-code. Working through this example will illustrate many of the steps and decisions in the design of a programming language.

For the sake of the example, we will assume that we are designing this pseudo-code for a computer with 2000 words of 10-digit memory; this was the capacity of the 650 and is a reasonable assumption for machines of that vintage. Of course, we will want our virtual computer to provide the facilities found in any computer, such as arithmetic, control of execution flow, and input-output, but in a more regular fashion than real computers. So let's begin by making a list of some of the functions our pseudo-code should accommodate:

- Floating-point arithmetic ($+$, $-$, $\times$, $\div$, $\sqrt{}$)
- Floating-point comparisons ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$)
- Indexing
- Transfer of control
- Input-output

What syntax should be used for this pseudo-code? Since many early computers did not have facilities for alphabetic input-output, we will have to use a numeric code for the statements of the language. Furthermore, since the most common input devices were card readers, we will adopt the convention of writing one number (representing an operation) on each card. Next, if we suppose that each instruction will be represented by one word, we will have a sign and 10 digits with which to represent each instruction. How large will the addresses be? Two digits are clearly insufficient; they would only allow 100 locations to be addressed. Four digits are too much since they would permit addressing 10,000 locations and there are only 2000 in the machine. So three digits seems the right choice; it permits addressing 1000 locations, which is adequate (at least it was considered adequate at that time), and leaves the other 1000 locations for the interpreter and program. If the first 1000 locations are used for data, then three digits will not allow addressing out of the data area. This is an example of *security*, since in this case we have made it possible for the user to commit a particular class of errors: overwriting the program or the interpreter. This illustrates another important principle of programming language design.

The Impossible Error Principle

Making errors impossible to commit is preferable to detecting them after their commission.

Instructions Have Three Operands

Let's consider the form of the arithmetic operations since they will be the most common. The instructions could have two, three, or four addresses:

- Two addresses: $x + y \rightarrow x$
- Three addresses: $x + y \rightarrow z$
- Four addresses: $x + y \times z \rightarrow w$

Each of these addresses must be represented in an instruction. Therefore, four-address instructions require 12 digits for the operand addresses, too much to fit in a 10-digit word. Two-address instructions will work since they only require six digits for the operand addresses, although this leaves a sign and four digits for the operation, which is excessive. This would allow 20,000 operations, and we only have 13 in our list. Three-address instructions will consume nine digits for the operands, leaving a sign and a digit for the operations, which will allow the encoding of 20 operations and is adequate for our purposes. This leads to the following format for our arithmetic instructions:

op opn1 opn2 dest

where *op* is the operation, *opn1* and *opn2* are the operands (x and y), and *dest* is the destination (z). For instance, if +1 means addition, then

```
+1 010 150 200
```

would add the contents of location 010 to the contents of location 150 and store the result in location 200 (we have added blanks to the code to make it more readable).

Orthogonal Design Increases Regularity

Numbers are difficult to remember, so we should design our pseudo-code so that the operations are as easy as possible to recall. To put it another way, there isn't much point to our pseudo-code if it isn't simpler and more regular than the real code. Since some of the arithmetic operations come in pairs (e.g., + and −, × and ÷), we can use the sign to distinguish these pairs. A preliminary encoding of operations is

	+	−
1	+	−
2	×	÷
3	square	square root

Notice that we have added the square function; it is useful, and since we already had its inverse, the square root, it is *symmetric* to include square.[3]

This is an example of *orthogonal* language design; that is, there are two orthogonal, or independent, mechanisms: (1) the digit (1, 2, 3), which selects the class of operation (additive, multiplicative, quadratic), and (2) the sign, which selects the direct operation or its inverse. Here we have applied the Orthogonality Principle.

> **The Orthogonality Principle**
>
> Independent functions should be controlled by independent mechanisms.

This principle is a corollary of the Regularity Principle.

Why does orthogonality simplify a language? If we assigned an arbitrary number to each pseudo-code operation, it would be necessary for the programmer to remember 20 independent facts for the 20 pseudo-code operations. Instead, we reflect in the coding the distinction between the direct and inverse operations. Therefore, structure in the operations is reflected in structure in the coding. The result is that the programmer only has to remember 12 independent facts: the plus and minus signs associated with the direct and inverse forms of the operations and the group that is associated with each of the digits. Another way to express this is the well-known architectural principle: *form follows function.*

Orthogonal means right angled. What do right angles have to do with language design? If we have two independently meaningful axes, one with *m* positions and another with *n* po-

---

[3] You may have noticed that square and square root are different from the other operations in that they are *unary*, that is, they only have one operand. The extra operand position in the instruction will be unused.

sitions, then we can describe mn different possibilities even though we only have to memorize $m + n$ independent facts:

m
⋮
2
1

mn

1 2 3 ⋯ n

As m and n increase, mn grows much faster than $m + n$. Thus, orthogonal design becomes more important as more possibilities must be described. When there are many possibilities, it may be advantageous to have more than two orthogonal axes.

Even if no exceptions are necessary, orthogonality can be carried too far. For example, we know that only 5 bits are needed to represent 20 possibilities; thus 20 instructions could be expressed by 5 orthogonal binary choices. Although in this scheme only 10 facts need to be remembered, it is most likely inferior, because the axes probably have no simple meaning to the programmer. (Try it! Can you assign a simple meaning to each axis? You might succeed.) In general, the independent functions controlled by the axes should be at least comprehensible, and preferably obvious, to the user. An orthogonal decomposition should help the user to form a simple, accurate *cognitive model* of the operation of the system.

Design Principles Must Be Applied Flexibly

We have seen already four "principles"—Automation, Regularity, Impossible Error, and Orthogonality—and you may wonder what to make of them. The central organizing role they have in this book was inspired by Strunk and White's classic *Elements of Style* (Macmillan, 1959), wherein the principles of good writing are expressed in a number of terse rules. My intention has been to identify the most important principles of good programming language design and to express them as directly as possible. They may strike you as dogmatic and uncompromising, but that is not the way they are intended. Like any other system of rules, they make sense only if they are applied in a flexible fashion; it would be cumbersome, tedious, and ultimately impossible to state for each principle all its exceptions, conditions, and limitations. Further, as you will see later, some of the principles contradict one another, and so—as in all design—a balance must be struck between conflicting goals. There are no rules for this balancing task, but with experience you will acquire a good eye, which will help you to achieve a harmonious design. One purpose of this book is to help you start to acquire this experience.

Orthogonality May Be Inappropriate

Too much orthogonality can harm a language since the language may become cluttered with facilities that have been included for symmetry but are of little use. That is, some of the mn possibilities may be useless or difficult to implement. Some of them may even be illegal; in this case, the programmer must remember them as exceptions (thus violating the Regularity

Principle). If e is the number of exceptions, then orthogonalization is advantageous only if $m + n + e < mm - e$.

Exercise 1-1: Explain in detail the justification for the formula $m + n + e < mn - e$.

Exercise 1-2: Code an operation to add the contents of location 125 to the contents of 206 and store the result in 803.

Exercise 1-3: Code an operation to divide the contents of location 401 by the contents of location 623 and store the quotient in location 107.

Exercise 1-4: Let the contents of locations 402 and 761 be x and y. Code instructions to compute $(x + y)^2$ into location 100. Assume that the first 10 locations of data memory are available for temporary storage.

Exercise 1-5: How can the +3 and −3 operations be altered to be more regular (i.e., more like the other operations), while still accomplishing the square and square-root functions?

Comparisons Alter Control Flow

We have said that we want our virtual computer to be regular so that it will be easier to use than the real computer. Achieving regularity will be easier if we use the same format for our other operations that we've used for the arithmetic operations. Let's see how this applies to the comparison operations. In some sense *equal* is the inverse of *not equal*, and *greater than or equal* is the inverse of *less than*, so we can use signs to distinguish between each operation and its inverse. We extend the operation table as follows:

	+	−
1	+	−
2	×	÷
3	square	square root
4	if = goto	if ≠ goto
5	if ≥ goto	if < goto

For example, the instruction

```
+4 200 201 035
```

means: If the contents of (data) location 200 equal the contents of (data) location 201, then goto the instruction in (program) location 035. Notice that it is not necessary to include the *greater* and *less than or equal* comparisons since they can be coded by reversing their operands (e.g., $x > y$ is coded as $y < x$, operation −5). We have also omitted positive, negative, and zero tests since these can be coded using the comparisons. For instance, if we adopt the convention that location 000 always contains the number zero, then we can jump to location 100 if location 702 is negative by

```
-5 702 000 100
```

Of course, any location that we know to be zero could be used, but it is always valuable to adopt standard *coding conventions.*

- ***Exercise 1-6:*** Code an instruction to jump to program location 103 if the value of data location 732 is greater than or equal to that of location 500.
- ***Exercise 1-7:*** Suppose data location 000 contains zero. Code an instruction to jump to program location 803 if data location 465 is zero.
- ***Exercise 1-8:*** Code instructions to compute the absolute value of the contents of location 231 and store the result in location 505. Assume that the instructions you write will go into program location 102 and the succeeding program locations. State any other assumptions that you make.
- ***Exercise 1-9:*** This pseudo-code does not include an unconditional jump. How could you do an unconditional jump using the facilities provided?

Moving

What other operations do we need for programming? Certainly, one of the most common operations is simply to move the contents of one location to another without doing any operation. Strictly speaking, it is not necessary to have a separate operation for this; it could be accomplished by adding zero. For example,

```
+1 150 000 280
```

effectively moves the contents of location 150 to location 280. Since a premise of this design is that floating-point arithmetic is quite slow, you can see that this would be a very inefficient way to move values between locations. Therefore, we will use the +0 operation to move one location to another.

Why did we choose +0 rather than +6? By picking +0 for the move, the series of codes 0, 1, 2, 3 stand for an easy-to-remember series of operations of increasing complexity:

move, add, multiply, square

This application of the Regularity Principle will make the codes easier to remember.

- ***Exercise 1-10:*** Code an instruction to move the contents of data location 100 into data location 101.
- ***Exercise 1-11**[4]**:*** What should be the function of the −0 operation? Our symmetric design leads us to expect it to be related to a simple move. It should be a useful operation that is not easily or efficiently accomplished with the other operations.

---

[4] Exercises marked by an asterisk require more thought and time than those not so marked. Exercises with two asterisks are major projects.

Indexing and Loops

One of the justifications for our pseudo-code was that it provided built-in indexing, so we will turn to the design of this facility next. To perform indexing we will need the address of the array and the address of the index variable, thus consuming two of the three address fields in the instruction. Therefore, the only operations we can perform directly on array elements are to move them to or from other locations. We can use the codes +6 and −6 to move from or to an array: $x_i \rightarrow z$ and $x \rightarrow y_i$. The formats of these operations are

+6 *xxx iii zzz*

−6 *xxx yyy iii*

For example, if there is a 100-element array beginning at location 250 in data memory, and location 050 contains 17, then

```
+6  250  050  803
```

will move the contents of location 267 (= 250 + 17) to location 803 (Figure 1.3). Similarly,

```
-6  722  250  050
```

will move the contents of location 722 to location 267.

Of course, one of the main reasons for using arrays is that we can write a loop to perform the same operation on each element of the array. To do this requires us to be able to initialize, increment, and test index variables. We may expect that we can use the arithmetic and comparison facilities already defined in our pseudo-code for this. But this is not so because these are floating-point operations, and indices are represented by integers. Even if this were not so, it would be useful to abstract out the code common to all loops. By building this into a pseudo-code operation, we eliminate another source of error. This is an example of the Automation Principle and its corollary, the Abstraction Principle.

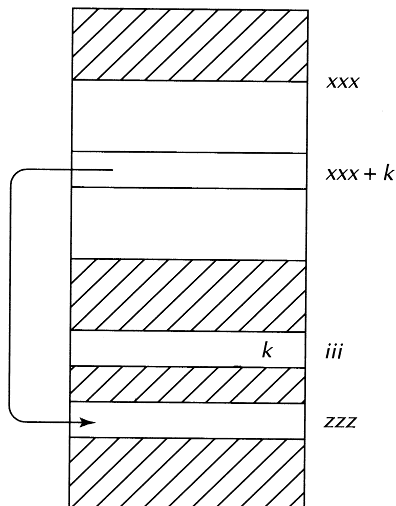


Figure 1.3 Indexing: +6 *xxx iii zzz*

> **The Abstraction Principle**
>
> Avoid requiring something to be stated more than once; factor out the recurring pattern.

Since we can use the move instruction (+0) to initialize indices, the new operation (+7) will only have to increment and test indices. To perform this operation, we need to know the location of the index, the location of the upper bound for the loop, and the location where the loop begins. The following format is analogous to the format of the comparisons:

+7 *iii nnn ddd*

Here *iii* is the address of the index, *nnn* is the address of the upper bound, and *ddd* is the location of the beginning of the loop. The operation increments location *iii* and loops to instruction *ddd* if the result is less than the contents of *nnn*. What is the meaning of the −7 operation? There are several possibilities—for instance, a *decrement* and test operation—so we will leave it undefined for the time being.

- ***Exercise 1-12:*** Suppose that there is an array stored in data memory beginning at location 401. Code an instruction that moves the contents of 207 into the array element indexed by location 950.

- ***Exercise 1-13:*** Suppose that an array begins at location 100 in data memory. Code instructions that add to location 020 the array element indexed by location 010.

- ***Exercise 1-14:*** Code an instruction that increments location 010, and loops to code location 005 if the contents of 010 are less than the contents of 030.

- ***Exercise 1-15:*** Suppose that an array begins at location 100 in data memory, and that location 030 contains the number of elements in the array. Code instructions that sum the elements of the array into location 005. State any additional assumptions that you make.

Input-Output

The only functions in our list that we have not yet addressed are the input and output operations. A program is not usually useful if it can't read data or print a result. Therefore, we will use the +8 operation to read a card containing one 10-digit number into a specified memory location and the −8 operation to print the contents of a memory location. (In a real pseudo-code, a punch operation would be more common than a print operation since this would allow the output of one program to be used as the input to another.) The complete list of operations is summarized in Figure 1.4. Notice that we have added a stop instruction to terminate program execution.

- ***Exercise 1-16:*** Code an instruction to read a number into location 044.

- ***Exercise 1-17:*** Suppose that an array begins at location 650 in data memory, and that location 907 contains the number of elements in the array. Code instructions to print out all the elements of the array. State any assumptions you make.

s	f	xxx	yyy	ddd

s = sign, f = function, xxx = operand1,
yyy = operand2, ddd = destination

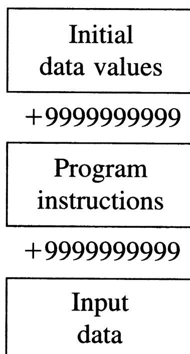
f \ s	+	−
0	move	(exercise)
1	+	−
2	×	÷
3	square	square root
4	=	≠
5	≥	<
6	$x(y) \rightarrow z$	$x \rightarrow y(z)$
7	incr. and test	(unused)
8	read	print
9	stop	(unused)

Figure 1.4 Pseudo-Code Operations

■ ***Exercise 1-18:*** Suppose an array begins at location 100 in data memory. Code instructions to read numbers into consecutive array elements until a card containing +9 999 999 999 is read. State any assumptions you make.

Program Structure

We now know how to write individual instructions, but we have not designed a means of constructing the program as a whole. For example, how do we arrange to get the program loaded into memory? How do we initialize locations in the data memory? How do we provide input data for the program? The simplest solution to this problem is to have the interpreter read initialization cards and store their content in consecutive memory locations. Thus, the structure of a program is



We have used a card containing the "flag value" +9999999999 to separate the initial values from the program and the program from the input data. The loader reads in the initial data values and stores them in consecutive locations (starting with 000) in the data memory. Then the loader reads in the program instructions and stores them in consecutive locations (starting with 000) in the program memory. The loader does *not* read the input data; this is read by the user's program whenever it executes a +8 instruction.

Therefore, the general structure of a program is (1) declarations, (2) executable statements, and (3) input data. This is not unlike the structure of a Pascal or FORTRAN program.

EXAMPLE: Mean Absolute Value of an Array As an example of the use of this pseudo-code, we show a program to compute the mean of the absolute values of an array. That is, if A is the array and it has n elements, we compute

$$\frac{1}{n}\sum_{i=1}^{n} |A_i|$$

The first problem is to determine the variables that will be needed and to lay out the data memory. We have used location 000 for a constant zero. The array to be averaged occupies

```
+0 000 000 000   (loc  0)       constant 0
+0 000 000 000   (loc  1)       index, i
+0 000 000 000   (loc  2)       sum of array
+0 000 000 000   (loc  3)       average of array
+0 000 000 000   (loc  4)       number of elements in array
+0 000 000 000   (loc  5)       temporary location
+0 000 000 000   (loc  6-999)   the array
+9 999 999 999                  end of initial data
+8 000 000 004   (loc  0)       read number of elements
+8 000 000 005   (loc  1)       read data into temp
+5 005 000 004   (loc  2)       if positive, skip
-1 000 005 005   (loc  3)       else negate
-6 005 006 001   (loc  4)       move temp into array sub i
+7 001 004 001   (loc  5)       incr. i, test with n, loop to loc. 1
+0 000 000 001   (loc  6)       reinitialize i to zero
+6 006 001 005   (loc  7)       add array sub i
+1 005 002 002   (loc  8)         to sum
+7 001 004 007   (loc  9)       incr. i, test with n, loop to loc. 7
-2 002 004 003   (loc 10)       sum / number of elements → avg.
-8 003 000 000   (loc 11)       print average
+9 000 000 000   (loc 12)       stop
+9 999 999 999                    end of program
+0 000 000 010                    number of input values
   input data
```

Figure 1.5 Pseudo-Code Program Example

locations 006 through the end of the data memory. The complete program appears in Figure 1.5; annotations on the right explain the steps. (This is not the best program for this task; we have written it to illustrate the use of this language.)

- ***Exercise 1-19:*** Write a simpler and more efficient pseudo-code program to accomplish this task.
- ***Exercise 1-20:*** Write a complete pseudo-code program to read in data cards (until a +9 999 999 999 flag card), add up the numbers on the cards, and print out the sum.
- ***Exercise 1-21:*** Write a complete pseudo-code program to print out the squares of the numbers from 1 to 100.
- ***Exercise 1-22:*** Write a complete pseudo-code program to print out the first 100 Fibonacci numbers.
- ***Exercise 1-23:*** Write a pseudo-code program to read in the coefficients of a quadratic equation and print both roots (if they exist). In solving this exercise, you will probably find that it is valuable to make a *variable map* that shows the location in which various variables are stored. It will also be useful to use symbolic labels until enough of the program is written to determine the actual location of the instructions.

1.3 IMPLEMENTATION

Automatic Execution is Patterned after Manual Execution

In this section we will see how to construct an interpreter for our pseudo-code. This will be an example of an *iterative interpreter*, one of the two important kinds (the other is a *recursive interpreter*, which is discussed in Chapter 11). How do we go about designing an interpreter? We can frequently get the insight necessary to design an interpreter by investigating how we would execute the language by hand. If we are to execute pseudo-code programs by hand, we will need some way to record the *state* of the computation, that is, the contents of the data memory. We will also need a listing of the program memory with the instruction at each location, together with a record of our place in the program (the latter is also part of the state). The major data structures required by our interpreter are those shown in Figure 1.6. Notice that we are using two arrays (each indexed from 0 to 999) to represent the areas of memory used for data and program storage. The data and program arrays are called `Data` and `Program` and the *instruction pointer* (which records our location in the program) is called `IP`. For example, '`Program[IP]`'[5] represents the instruction in the Program array designated by `IP`. We may find that we need some other minor data structures as we continue with the design.

---

[5] When necessary for clarity, programming language text is surrounded by single quotation marks (' '). The text being discussed is exactly that between the quotes (i.e., we don't include punctuation within the quotes). Double quotes (" ") will be used for all other purposes, such as direct quotations and titles. No quotation marks are used around displayed program text.

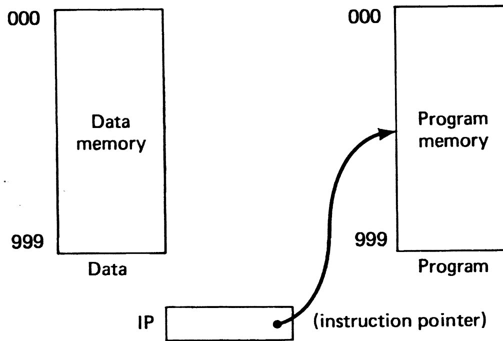


Figure 1.6 Interpreter Data Structures

The Read-Execute Cycle Is the Heart of an Iterative Interpreter

We can now consider how a program is actually interpreted. Roughly, what we will do is read the next instruction to be executed (as indicated by the instruction pointer), determine the operation encoded by the instruction, and then perform that operation. When execution of the operation is completed, we will begin this process again with the next instruction to be executed. This process is called the *read-execute cycle*, and can be summarized as follows:

1. Read the next instruction.
2. Decode the instruction.
3. Execute the operation.
4. Continue from step 1 with the next instruction.

Have you noticed that we have omitted a small but crucial detail? When is the instruction pointer advanced? The natural place to do this would seem to be step 4, since this is just prior to reading the next instruction. While this works fine as long as the program continues to execute sequentially, it will be difficult to handle jumps since they must alter the instruction pointer (in step 3). A better solution, and the one that is adopted in most computers (both real and virtual), is to advance the instruction pointer at the end of step 1. Typical code for step 1 is

```
instruction := Program[IP];
IP := IP+1;
```

The `IP` either is ready for the next cycle if sequential execution is to continue, or it can be altered in step 3 in the case of a jump.

Notice that we have written the code for step 1 in a Pascal-like descriptive notation (a *program design language*). Why would we want to write a pseudo-code interpreter if we have Pascal available for programming? We wouldn't. If we wanted to be realistic, we would write the pseudo-code interpreter in machine language. The result would look like Figure 1.2, which is actually a small part of a pseudo-code interpreter. This would be carrying things too far; our goal is to understand iterative interpreters, not to relive the 1950s. Therefore, we will use a more convenient, Pascal-like notation for describing the interpreter. This will al-

low us to see the algorithm without getting bogged down in the details of machine-language programming.

Instructions Are Decoded by Extracting Their Parts

We will discuss each of the other steps in the read-execute cycle. Since our pseudo-code has been designed with a regular structure, decoding is simple; we simply extract the sign, operation code, and three address fields. For example, the destination address could be extracted by

```
dest := abs (instruction) mod 1000
```

(where '*x* `mod` *y*' gives the remainder of dividing *x* by *y*). We will assume that the names of these extracted parts are `sign`, `op`, `opnd1`, `opnd2`, and `dest`.

■ ***Exercise 1-24:*** Write the code to extract the other fields in an instruction.

The next step in instruction decoding is to determine what kind of operation has to be performed. This is specified by a combination of the `sign` and the `op` fields. We can break down the execution into cases, depending on the value of these fields. The operation to be performed by each case is just read from Figure 1.4. The result is shown in Figure 1.7.

■ ***Exercise 1-25:*** Fill in the rest of the second case-statement in Figure 1.7.

■ ***Exercise 1-26*:*** Estimate the overhead of this pseudo-code interpreter. That is, estimate the number of memory references made in the read-execute cycle beyond those ac-

```
if sign is '+' then
  do case op of:
    0:  move;
    1:  add;
    2:  multiply;
    3:  square;
    4:  test equality;
    5:  test greater or equal;
    6:  fetch from array;
    7:  increment and test;
    8:  read;
    9:  stop.
if sign is '-' then
  do case op of
    0:  do operation from exercise;
    1:  subtract;
    .
    .
    .
```

Figure 1.7 Instruction Decoding

tually required for computing the result (e.g., the floating-point operation). What percentage of the execution time will be overhead if the average software-implemented floating-point operation requires 100 memory references? What percentage is overhead if the floating-point operations are implemented in the hardware and take just three memory references?

Computational Instructions

Most of the computational instructions are simple to interpret. For example, to interpret a multiplication, the two operands (`Data[opnd1]` and `Data[opnd2]`) must be fetched, multiplied by the floating-point multiplication routine, and stored at the destination location (`Data[dest]`). We can express this as

```
Multiply:
   Data[dest] := floating product (Data[opnd1], Data[opnd2]).
```

The other computational instructions are analogous.

■ ***Exercise 1-27:*** Write in a program design language the implementation of the other computational instructions of the pseudo-code (all the codes except $\pm$ 4 through 7).

Control-Flow Instructions

The control-flow instructions are implemented in an analogous manner; the only difference is that the `IP` must be altered if the test is satisfied. For example,

```
Test equality:
   if floating equality (Data[opnd1], Data[opnd2]) then
   IP := dest.
```

■ ***Exercise 1-28:*** Write in a program design language the implementation of the comparison operations of the interpreter.

■ ***Exercise 1-29:*** We now have a complete design for the "main loop" of a pseudo-code interpreter. In order to have a complete interpreter, it is necessary to write a *loader* that will read in the initialization and program cards and load them into the `Data` and `Program` arrays. Design this part of the interpreter and write it in a program design language.

■ ***Exercise 1-30*:*** Translate the entire interpreter into your favorite programming language and test it on the example program in Figure 1.5. You do not have to implement your own floating-point arithmetic; just use the floating-point operations provided in your chosen programming language.

Interpreters Simplify Debugging

Next, we will investigate some improvements that can be made to this interpreter, which will highlight some of the ways programming languages simplify programming. In the beginning of this chapter, we said that one of the motivations for pseudo-codes was the difficulty of

programming; you probably know from your own experience that much of this is a result of the difficulty of debugging. Since debugging can often be expedited by a better understanding of what the program is doing, programmers have often resorted to "playing computer," that is, to interpreting their programs by hand to see what they actually do as opposed to what they expect them to do. Clearly, this is a process that can be profitably automated. What we would like is the ability to get a *trace* of the execution of the program, that is, a record of the instructions it has executed. This can be done by adding code to step 1, Read Next Instruction, to print out the location and code for the current instruction:

```
Read Next Instruction:
   instruction := Program [IP];
   if trace is enabled then
      print IP, instruction;
   IP := IP + 1.
```

A trace of the program in Figure 1.5 would begin

```
000 +8000000004
001 +8000000005
002 +5005000004
004 -6005006001
005 +7001004001
001 +8000000005
        ⋮
```

■ ***Exercise 1-31:*** Show the next 10 steps in the trace of the program in Figure 1.5. State your assumptions about the input numbers.

■ ***Exercise 1-32:*** The above modification prints out the instruction as a 10-digit number. It would be preferable to print it out in interpreted form, that is, with its fields separated. For example, the trace may begin

```
LOC  OP   OPND1  OPND2  DEST
000  +8    000    000    004
001  +8    000    000    005
002  +5    005    000    004
004  -6    005    006    001
005  +7    001    004    001
001  +8    000    000    005
            ⋮
```

(Of course, this assumes the availability of a printer that can print letters.) Alter the interpreter to produce an interpreted trace.

■ ***Exercise 1-33*:*** The trace can be made even more valuable by printing out the operation name in English, the values of the source operands, and the value to be placed in the destination operand. Alter the interpreter to do this. Note, however, that not all fields are used in all of the instructions.

■ ***Exercise 1-34*:*** For a large program, the trace could be very long, even though the programmer was interested in only a very small region of the program. Design an interpreter operation (−9 perhaps) that will allow the programmer to enable and disable tracing at different points in the program. (Keep in mind that it is not convenient to insert or delete instructions in these pseudo-code programs because of their absolute instruction addresses.)

■ ***Exercise 1-35*:*** Another useful debugging tool is breakpoints. This feature allows the programmer to specify certain instruction addresses as *breakpoint addresses*; whenever execution reaches one of these addresses, interpretation of the program stops until the programmer restarts it. This allows the programmer to investigate the state of the data memory at selected points during execution. Design a breakpoint facility for the pseudo-code interpreter.

■ ***Exercise 1-36*:*** Design a *data trap* facility. This is like a breakpoint except that interpretation is interrupted whenever specified locations in the *data* memory are referenced.

Statement Labels Simplify Coding

We will now consider an aid to the coding of a program. One of the major goals of programming languages is the elimination of the tedious, error-prone tasks in programming (the Automation Principle). One of these tasks results from the use of absolute locations in pseudo-code instructions. Consider what would happen if we wanted to insert a new instruction (e.g., a trace instruction) after the instruction in location 000 in Figure 1.5. This would shift down all of the remaining instructions and require us to correct the destination addresses in locations 003 and 007. We can see that maintenance would be almost impossible for a large program, since we would have to find all the addresses that could be altered by a change.

One solution adopted by several early pseudo-codes was the provision of *symbolic labels* for statements. Let's see how this would work. When we describe an algorithm in English, such as the read-execute cycle described earlier, we often number the steps so that they can be referred to from other steps, for example, "Continue from step 1." This is an example of the Labeling Principle.

> **The Labeling Principle**
>
> Do not require the user to know the absolute position of an item in a list. Instead, associate labels with any position that must be referenced elsewhere.

We can modify the pseudo-code to do this by introducing a *label definition operator.* The instruction

```
-7 0LL 000 000
```

defines the statement number, or *label, LL.* (We will allow labels only in the range 00–99 so that we can use a 100-element label table.) Notice that this is not an *executable statement*; it merely marks the place in the program to be labeled *LL.* We call such statements *declarations* and say that they *bind* a symbolic label to an absolute location. We will also alter

the jump instructions to refer to symbolic labels in their destination field rather than *absolute* labels. Thus, the format of the equality test is

```
+4 xxx yyy 0LL
```

In the following illustration, the executable part of our example program has been rewritten making use of labels.

```
+8 000 000 004       read number of elements
-7 020 000 000     20:
+8 000 000 005       read into temp
+5 005 000 040       if positive, skip to label 40
-1 000 005 005       negate temp
-7 040 000 000     40:
-6 005 006 001       move temp to array sub i
+7 001 004 020       incr. i, test with n, loop to label 20
+0 000 000 001       reinitialize i to zero
-7 050 000 000     50:
+6 006 001 005       add array sub i
+1 005 002 002           to sum
+7 001 004 050       incr.i, test with n, loop to label 50
       etc.
```

How can we interpret symbolic labels? Again, we can begin by observing how people do it. If we were interpreting the above program and came to a jump to location 50, we would very likely find its location by looking through the program until we found a −7 instruction with a 050 in the destination field. This is, in fact, the way some interpreters work, such as those found in some programmable hand-held calculators. We can see, though, that if the program were very large, we would be spending a lot of time scanning the program to find labels. We would probably save ourselves this trouble by making a *label table* that listed the labels and their absolute locations, for example,

Label	Location
20	001
40	005
50	009

This table could be constructed exactly the way we do it by hand: The first time we search for a label, we put it in the table so that we will have the absolute location for later uses of the label. It would be better, however, to build the label table as the program is read into Program memory. This simplifies the interpretation of jumps since we know all labels are defined before execution begins. More important, it allows us to increase *security* by ensuring that all the labels that are referenced are defined once and only once. This is in accord with the Security Principle (p. 29).

This checking can be done by initializing the label table to some value that we will interpret to mean "undefined," say −1. During loading, whenever we encounter a −7 in-

struction defining a label, before we enter its absolute location into the label table, we will ensure that it has not already been defined by seeing if its entry is negative. Conversely, whenever we encounter an instruction *referencing* a label (e.g., +7), we will check to see if it has been defined, as indicated by a nonnegative value. If it has been defined, then all is well; if it hasn't, then we will store the value −2 indicating a label that has been referenced but not defined. If the label is later defined, this −2 will be changed to a positive value reflecting the absolute location of the label. At the end of loading, a final scan of the label table for any remaining −2 values will enable us to report the undefined labels. The label table we have described is a rudimentary form of a *symbol table*; this data structure is used in all programming language implementations for keeping track of labels, variables, and other symbolically named objects. Symbol tables will be discussed periodically throughout this book. The use of *symbols*, in which a sign refers to something other than its literal meaning, is a fundamental idea in computing. For example, in this case the symbol 20 refers to program location 001, not to its literal meaning, program location 020.

■ ***Exercise 1-37*:*** Modify your pseudo-code interpreter to use symbolic statement labels of the type we have described. Test it on the modified Mean Absolute Value program (p. 27).

■ ***Exercise 1-38*:*** We have only allowed statement labels in the range 0–99 so that only a 100-element label table will be required. Even so, for small programs many of the entries will be unused. Design a scheme for storing the absolute locations (and "undefined" codes) for labels in the range 0–999.

■ ***Exercise 1-39*:*** Label declarations provide new opportunities for debugging aids. For example, the interpreter can print a message every time the program jumps to a label or the interpreter can pause for programmer interaction whenever a label is encountered. Design and implement one or more of these debugging facilities for your interpreter.

Variables Can Be Processed Like Labels

Since we have eliminated the error-prone use of absolute statement labels, we will probably want to know if we can also eliminate absolute data addresses. The answer is "yes"; we can do this by constructing a *symbol table* that holds the absolute location of every variable. We can then use fixed *symbolic* labels (still in the form of three-digit numbers) in the pseudo-code instructions. In the initial-data section of the program, pairs of cards could be used to declare simple variables and arrays. Thus,

```
+0  sss  nnn  000
±d  ddd  ddd  ddd
```

will declare a storage area with the symbolic name *sss*, *nnn* locations long, initialized to all *±d ddd ddd ddd*. For instance,

```
+0 666 150 000
+3 141 592 654
```

could be used to declare a 150-element array to be identified by the label 666 and initialized to all +3141592654. Two simple variables, labeled 111 and 222, could be declared and initialized to zero by

```
+0 111 001 000
+0 000 000 000
+0 222 001 000
+0 000 000 000
```

We know they are simple variables because the amount of memory allocated to each is one word.

For each declaration the loader keeps track of the next available memory location and *binds* the symbolic variable number to that location. Therefore, we say that the *binding time* of this declaration is load time. We will see in later chapters that other binding times are possible. Also, notice that the loader has taken over another job for the programmer: *storage allocation.*

■ ***Exercise 1-40:*** What principle does the loader illustrate?

After the above declarations, we could use 111 to index 666 and store the result in 222 by

```
+6 666 111 222
```

This is analogous to the Pascal statement

```
V222 := V666 [V111]
```

where the variable names V222, V666, and V111 correspond to the symbolic storage labels 222, 666, and 111.

Clearly, these symbolic data names can be implemented in exactly the same way we implemented symbolic statement labels. We can also perform the same checking for undefined names, as well as additional checking, such as for out-of-bounds array references. That is, the interpreter can record in the symbol table the size of the array and then on each array reference instruction (±6) ensure that the index is less than this bound. Since this checking prevents a violation of the program's intended structure, it is in accord with the Security Principle first proposed by C. A. R. Hoare:

The Security Principle

No program that violates the definition of the language, or its own intended structure, should escape detection.

■ ***Exercise 1-41:*** Rewrite the Mean program using the variable declarations we have described.