■ ***Exercise 1-42*:*** Modify the loader to build a symbol table for the variables and to initialize the `Data` array. Modify the interpreter to use these symbolic variable numbers. Include the error-checking facilities described above.

■ ***Exercise 1-43*:*** Propose a debugging aid based on symbolic variable numbers and describe its implementation in detail.

The Ideas Presented Above Are Easily Extended to a Symbolic Pseudo-Code

The provision of symbolic numbers for variables and statement labels has gone a long way toward making our pseudo-code easier to use. It is still necessary for users to remember the relationship between their variables and the numeric tags they invent. This is an error-prone process since the programmer has to remember whether `-2` or `-3` is divide, whether `111` is the index or the temporary, and so on. The programmer will probably keep lists of the correspondence between these codes and the abstractions they represent, such as the list of operation codes in Figure 1.4. Therefore, we can eliminate this source of errors by maintaining this correspondence for the user. This was done in many of the early pseudo-codes when input-output equipment that could handle alphabetic characters became available.

■ ***Exercise 1-44:*** What principle is illustrated by making the computer keep track of the correspondence described above?

How will we go about designing a symbolic pseudo-code? First, let's consider the *syntax* of (way to write) the variables. Currently, the interpreter looks up a three-digit symbolic variable number in the symbol table in order to find the absolute location of that variable in the `Data` array. If we replace this three-digit number with a three-character alphanumeric name, then we will be able to use the same lookup process while allowing the programmer to pick more mnemonic variable names. The programmer will be able to use a name like `AVG` instead of an absolute location (`003`) or an arbitrary numeric tag (`123`). The same can be done for the operation codes, using mnemonic words like `ADD` and `READ` instead of codes like `+1` and `+8`. The loader will have to look these up in a symbol table and replace them by their codes. Therefore, a typical statement in this symbolic pseudo-code would look like

```
ADD  TMP  SUM  SUM
```

As has been said, the primary input medium for early computers was punched cards. Since there was a long tradition (dating from the use of office punched card equipment in the first half of the twentieth century) of assigning particular fixed columns to the *fields* of data records, the same kind of *fixed format* convention was adopted for the pseudo-codes. If the operation names are limited to four characters and the variable names and statement labels to three characters, then we can use a format such as the following:

Columns 1–4:	operation	10–12:	operand 2
6–8:	operand 1	14–16:	destination

Only uppercase letters will be used since these were all that were available on key punches at that time. In Figure 1.8 the Mean program is shown translated into this symbolic pseudo-code. We have not included a list of all the mnemonics since they should be clear from context.[6]

We can see that the general structure of a program is

```
declarations
END
statements
END
```

This format, declarations followed by statements, has been preserved in most programming languages. For instance, in the language Ada it takes the form

```
declare
  declarations
begin
  statements
end;
```

Also, variable declarations have the syntax (form)

```
VAR variable-name type
initial-value
```

where 'type' means the number of locations the variable occupies. This format is also preserved in many modern languages. In Ada we write

```
variable-name: type := initial-value;
```

although the idea of a type in Ada (and most modern languages) involves much more than just the amount of storage to be allocated. There is one more thing to notice about the syntax of this pseudo-code: The operation comes first in the statements:

```
operation  operand1  operand2  destination
```

This is called a *prefix* format (pre = before), and is still used for statements in most programming languages, for example, in FORTRAN

```
DO 20 I=1, 100
PRINT 30, AVG
```

There is no reason why we had to pick a prefix form (there are others such as postfix and infix), although it does agree with English grammar in putting the verb first in an imperative sentence.

---

[6] You may wonder why programs are shown sometimes in all uppercase letters, sometimes in lowercase letters, sometimes in mixed cases, or in boldface, etc. The reason is that each language community has its own typographical conventions, which they have evolved and are part of the overall character of the language. Therefore we try to follow those conventions in our examples.

```
OPER    OP1   OP2   DST   COMMENTS
VAR     ZRO   1           CONSTANT ZERO
+0000000000
VAR     I     1           INDEX
+0000000000
VAR     SUM   1           SUM OF ARRAY
+0000000000
VAR     AVG   1           AVERAGE OF ARRAY
+0000000000
VAR     N     1           NUMBER OF ELEMENTS IN ARRAY
+0000000000
VAR     TMP   1           TEMPORARY LOCATION
+0000000000
VAR     DTA   990         THE DATA ARRAY
+0000000000
END
READ    N                 READ NUMBER OF ELEMENTS
LABL    20
READ    TMP               READ INTO TEMP
GE      TMP   ZRO   40    IF POSITIVE, SKIP TO 40
SUB     ZRO   TMP   TMP   NEGATE TEMP
LABL    40
PUTA    TMP   DTA   I     MOVE TEMP INTO THE I-TH ELEMENT
LOOP    I     N     20    LOOP FOR ALL ARRAY ELEMENTS
MOVE    ZRO         I     REINITIALIZE INDEX TO ZERO
LABL    50
GETA    DTA   I     TMP   ADD I-TH ELEMENT
ADD     TMP   SUM   SUM     TO SUM
LOOP    I     N     50    LOOP FOR ALL ARRAY ELEMENTS
DIV     SUM   N     AVG   COMPUTE AVERAGE
PRINT   AVG               AND PRINT IT
STOP
END
```

Figure 1.8 Mean Absolute Value in Symbolic Pseudo-Code

To implement the symbolic pseudo-code, all that is required is that as the loader reads in each instruction, it looks up the operation and the operands in the symbol table and replaces them with the proper codes. The encoded form of the instruction is then stored in the `Program` array. Thus, we can see that the loader is performing a *translation* function since it is translating the *source form* of the program (the symbolic pseudo-code) into an *intermediate form* (the numeric pseudo-code) that is more suitable for the interpreter. This two-stage process, translation followed by interpretation, is very common and will be discussed at length in the following chapters. In fact, the translator, with its name lookup and storage allocation functions, is a rudimentary form of a *compiler*. The function of a compiler is to translate a

program in some source language into a form that is more convenient for execution. This form is often machine language, which can be directly executed, but it may also be an intermediate language suitable for interpretation.

- ***Exercise 1-45*:*** Modify your interpreter to implement this symbolic pseudo-code and test it on the Mean Absolute Value program. Translate your quadratic roots program into this pseudo-code and execute it with this interpreter.
- ***Exercise 1-46*:*** Describe how you would make the pseudo-code *free format*, that is, independent of the columns in which the fields appear (of course, they must be in the correct order). How would you implement this?

1.4 PHENOMENOLOGY OF PROGRAMMING LANGUAGES

Obviously programming languages, even simple ones such as our pseudo-code, are tools, and so it will be worthwhile to investigate them from this perspective. Fortunately, the *phenomenology* of tools has been explored in some detail, and in this section I will be using the results of the investigations of Don Ihde.[7]

Tools Are Ampliative and Reductive

To better understand the phenomenology of programming languages, we may begin with a simpler tool. Ihde contrasts the experience of using your hands to pick fruit with that of using a stick to knock the fruit down. On the one hand, the stick is *ampliative*: it extends your reach to otherwise inaccessible fruit. On the other, it is *reductive*: your experience of the fruit is mediated by the stick, for you do not have the direct experience of grasping the fruit and tugging it off the branch. You cannot feel if the fruit is ripe before you pick it.

"Technological utopians" tend to focus on the ampliative aspect—the increased reach and power—and to ignore the reductive aspect, whereas "technological dystopians" tend to focus on the reductive aspect—the loss of direct, sensual experience—and to diminish the practical advantages of the tool. But, "both are reduced focuses upon different dimensions of the human technological experience." Therefore, we should acknowledge the essential ambivalence of our experience of the tool: positive in some respects, negative in others. As Ihde says, "all technology is nonneutral."

These observations apply directly to programming languages. In the earliest days, computers were programmed directly with patch-cords (an experience that is occasionally praised in words appropriate to picking your own fruit!). Programming in machine language is nearly as direct, and some early programmers even criticized decimal numbers for distancing programmers from the machine too much. Pseudo-codes are even more distancing, amplifying programmers' ability to write correct code, but reducing their contact with and control over

[7] See his *Consequences of Phenomenology* (State University of New York Press, 1986), pp. 104–136. In *phenomenology* one investigates the invariant structure of some phenomenon, that is, of some aspect of concrete human experience of the world, by systematic variation of that experience. Tools are the phenomena of interest here.

the machine. Early debates about the usefulness of pseudo-codes reflected ambivalence about them as tools.

Fascination and Fear Are Common Reactions to New Tools

When first introduced, programming languages elicited the two typical responses to a new technology: *fascination* and *fear.* Utopians tend to become fascinated with the ampliative aspects of new tools, so they embrace the new technology and are eager to use it and to promote it (even where its use is inappropriate); they are also inclined to extrapolation: extending the technology toward further amplification. (It is worth recalling that the root meaning of "fascinate" is "to enchant or bewitch.") Dystopians, in contrast, fear the reductive aspects of the tool (so higher-level languages are feared for their inefficiency), or sometimes the ampliative aspects, which may seem dangerous. The new tool may elicit ambivalent feelings of power or of helplessness. Ideally, greater familiarity with a technology allows us to grow beyond these reactions, for the benefits and limitations of a technology are seldom revealed in the fascination–fear stage of its acquaintance.

With Mastery, Objectification Becomes Embodiment

A tool replaces immediate (direct) experience with mediated (indirect) experience. Yet, when a good tool is mastered, its mediation becomes *transparent.* Consider again the stick. If it is a good tool (sufficiently stiff, not too heavy, etc.) and if you know how to use it, then it functions as an extension of your arm, allowing you both to feel the fruit and to act on it. In this way the tool becomes partially *embodied.* On the other hand, if the stick is unsuitable or you are unskilled in its use, then you experience it as an *object* separate from your body; you relate *to* it rather than *through* it. With mastery a good tool becomes transparent: it is not invisible, for we still experience its ampliative and reductive aspects, but we are able to look through it rather than at it.

Programming languages exhibit a similar variation between "familiar embodiment" and "alienated otherness." When you first encounter a new programming language, it is experienced as an object: something to be studied and learned about. As you acquire skill with the language, it becomes transparent so that you can program the machine through the language and concentrate on the project rather than the tool. With mastery, objectification yields to (partial) embodiment.

This is part of the reason that a full evaluation of a programming language requires considerable experience in its use. When the language is first encountered, one is apt to fall into the limited perspectives of fascination and fear. But even with increased familiarity, there is still a tendency to treat the languages as an object, until mastery is achieved, and the language's benefits and limitations can be viewed in a context of transparent use.

Programming Languages Influence Focus and Action

Tools influence the style of a project. For example, Ihde contrasts three writing technologies: a dip pen, an electric typewriter, and a word processor. In the case of a dip pen the speed of writing is so much slower than the speed of thought that a sentence can be crafted

word by word as it is written; this could tend to a style of *belle lettres* or to calligraphy. With an electric typewriter the speed of writing is closer to the speed of thought, so this tool inclines toward (but does not dictate) a more informal style. However, revisions require retyping, so there is a tendency to revise works as wholes. With a word processor, in contrast, text can be revised and rearranged in small units, so there is a greater tendency to salvage bits of text. There is a tendency toward a different style ("Germanic tomes," Ihde suggests).

In general, a tool influences *focus* and *action*. It influences focus by making some aspects of the situation salient and by hiding others; it influences actions by making some easy and others awkward. Like other tools, programming languages influence the focus and actions of programmers and therefore their programming style.

A programming language inclines programmers toward a style; it creates a tendency, which the majority of programmers will follow. However, I must emphasize that it does not dictate a style; individual programmers may choose to work against the language's inclination. Thus, for example, we sometimes observe a programmer "writing FORTRAN in LISP," that is, writing FORTRAN-style code in the LISP language. Nevertheless, we must consider carefully the stylistic inclinations of a programming language. Does it encourage the focus and actions that we want to encourage?

Summary

We summarize what we can conclude about programming languages from the phenomenology of tools. Programming languages transform the situations encountered in programming projects. They are nonneutral and have ampliative and reductive aspects, both of which should be kept in mind. Further, to assess the benefits and limitations of a programming language properly, it is necessary to advance beyond the fascination–fear stage. When a well-designed language is mastered, it becomes a transparent extension of the programmer rather than an obtrusive object. Finally, by influencing the focus and actions of programmers, a language inclines its users toward a particular style, but it does not force it on them.

- ***Exercise 1-47*:*** Identify the ampliative and reductive aspects of several common tools and technologies and discuss the conditions for transparency and embodiment. For example, consider eye glasses, automobiles, telephones, recorded music, or the Internet.
- ***Exercise 1-48*:*** Select an ampliative aspect of some programming language and describe the result of an extrapolation toward greater amplification. What is the correlative reduction? Discuss whether this extrapolation would be desirable.
- ***Exercise 1-49*:*** Amplificatory extrapolations often reflect our "imaginations and desires" for our projects. What do you think are the typical "imaginations and desires" of programmers? What sorts of "trajectories of extrapolation" might they lead to?
- ***Exercise 1-50*:*** Analyze in detail the effect of our pseudo-code (either the numerical or symbolic version) on the focus and actions of its users. Compare its effect on a 1950s programmer and on a contemporary programmer.
- ***Exercise 1-51*:*** Consider your favorite programming language. What focus and actions does it encourage? What focus and actions does it discourage? Give evidence in both cases.

■ ***Exercise 1-51*:*** Programming languages (and other technologies) are *culturally embedded*, which means that our reactions to them are influenced by our personal and collective backgrounds. Further, their stylistic inclinations may vary from user to user. Select a programming language with which you are familiar and discuss how it is experienced by different groups of programmers (scientific, systems, commercial, amateur, novice, etc.).

1.5 EVALUATION AND EPILOG

Pseudo-Code Interpreters Simplified Programming

We have seen that pseudo-codes simplified programming in many ways. Most important, they provided a *virtual computer* that was more *regular* and *higher level* than the real computers that were available at first. Also, they decreased the chances of error while taking over from the programmer many of the tedious and error-prone aspects of coding. Pseudo-codes increased *security* by allowing error checking, for example, for undeclared variables and out-of-bounds array references. Finally, they simplified debugging by providing facilities such as execution traces. We will see in later chapters that all of these remain important advantages of newer programming languages.

Floating-Point Hardware Made Interpreters Unattractive

Decoding pseudo-code instructions adds a great deal of overhead to program execution. In the beginning of this chapter, we pointed out that most of this overhead was swamped by the time necessary to simulate floating-point arithmetic. That is, since programs were doing mostly floating-point arithmetic, which was slow, they were spending most of their time in the floating-point subroutines. The little additional time they spent in the interpreter was well worth the advantages of the pseudo-code. This changed when floating-point hardware was introduced on the IBM 704 in 1953. Experience with floating-point arithmetic and indexing facilities in the pseudo-codes led IBM and the other manufacturers to include these in the newer computers. Since programs were no longer spending most of their time in floating-point subroutines, the factor of 10 (or more) slower execution of interpreters became intolerable. Since at this time computer time was still more expensive than programmer time, interpreters became unpopular because the total cost of running a machine-language program was less than that of a pseudo-code program.

Pseudo-codes are still used for special purposes such as intermediate languages. For example, Pascal is often translated into a pseudo-code called P-code. The P-code program is then either translated into machine language or interpreted. Programmers no longer write directly in pseudo-codes, except when programming some hand-held calculators.

Libraries Led to the Idea of "Compiling Routines"

An alternative to the use of interpreters was the "compilation" of programs from libraries of subroutines. The idea was that a programmer would write pseudo-code instructions that

would, at load time, call for subroutines to be copied from a library and assembled into a program. Since the translation and decoding were done once, at compilation time, compiled programs ran more quickly than interpreted programs. This was so because an interpreter, for example, must decode the instructions in a loop every time through the loop.

However, since the subroutines assembled by a compiler could not be made to fit together perfectly in all combinations, there was an *interface overhead* that made compiled programs less efficient than hand-coded ones. The result was that programmers considered these "automatic coding" techniques inherently inefficient and only suitable for short programs that would be run only a few times. Thus, the prevailing attitude in the early to mid-1950s was that important programming had to be done in assembly language. Although, as we will see in the next chapter, FORTRAN proved the viability of "automatic coding," this attitude was to continue for many years.

EXERCISES*

1. Compare and contrast the numeric pseudo-code interpreter, the symbolic pseudo-code interpreter, and an assembler.

2. Study the manual of an assembly language and critique that language with respect to the language design principles you have learned. Pay particular attention to the regularity and orthogonality of the language.

3. Pick some programmable calculator and evaluate its instruction set as a pseudo-code.

4. Make the following specification more precise, that is, make reasonable assumptions and justify them: *Free format* pseudo-code instructions allow the operator and operands of instructions to be separated by any number of blanks, and allow any number of instructions to be put on one line.

5. Alter the symbolic pseudo-code loader to accept the free format instructions specified in the previous exercise.

6. Suppose we wanted to add the three trigonometric functions (sin, cos, tan) and their inverses to our pseudo-code interpreter. Design this extension to the language. (Note that this extension will increase the number of operators to more than 20.)

7. As language evolve, they often must be extended. Discuss how to design a pseudo-code to accommodate the later addition of new operations. Discuss a policy for limiting extensions to those that are necessary.

8. In this chapter we designed a pseudo-code for numerical and scientific applications. Design a pseudo-code for commercial (business data-processing) applications. Discuss your rationale for including or omitting various features.

9. Implement the pseudo-code designed in the previous exercise.

10. Pick an application area that interests you (e.g., stock portfolio management, expenses, dates/appointments, checkbook management, grading). Design a pseudo-code appropriate to a hand-held computer that would be helpful in this application area. You will be graded on your adherence to the principles you've learned and on the wisdom of engineering trade-offs.

11. Implement the pseudo-code designed in the previous exercise.