programming pearls

By Jon Bentley

WRITING CORRECT PROGRAMS

In the late 1960s people were talking about the promise of programs that verify the correctness of other programs. Unfortunately, it is now the middle of the 1980s, and, with precious few exceptions, there is still little more than talk about automated verification systems. Despite unrealized expectations, however, the research on program verification has given us something far more valuable than a black box that gobbles programs and flashes "good" or "bad"—we now have a fundamental understanding of computer programming.

The purpose of this column is to show how that fundamental understanding can help programmers write correct programs. But before we get to the subject itself, we must keep it in perspective. Coding skill is just one small part of writing correct programs. The majority of the task is the subject of the three previous columns: problem definition, algorithm design, and data structure selection. If you perform those tasks well, then writing correct code is usually easy.

The Challenge of Binary Search

Even with the best of designs, every now and then a programmer has to write subtle code. This column is about one problem that requires particularly careful code: binary search. After defining the problem and sketching an algorithm to solve it, we'll use principles of program verification in several stages as we develop the program.

The problem is to determine whether the sorted array $X[1\,..\,N]$ contains the element T. Precisely, we know that $N \geq 0$ and that $X[1] \leq X[2] \leq \cdots \leq X[N]$. The types of T and the elements of X are the same; the pseudocode should work equally well for integers, reals or strings. The answer is stored in the integer P (for position); when P is zero T is not in $X[1\,..\,N]$, otherwise $1 \leq P \leq N$ and $T = X[P]$.

Binary search solves the problem by keeping track of a range within the array in which T must be if it is anywhere in the array. Initially, the range is the entire array. The range is diminished by comparing its middle element to T and discarding half the range. This process continues until T is discovered in the array or until the range in which it must lie is known to be empty. The process makes roughly $\log_2 N$ comparisons.

Most programmers think that with the above description in hand, writing the code is easy; they're wrong. The only way you'll believe this is by putting down this column right now, and writing the code yourself. Try it.

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) to convert the above description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take 30 minutes to examine their code, which the programmers did with test cases. In many different classes and with over a hundred programmers, the results varied little: 90 percent of the programmers found bugs in their code (and I wasn't always convinced of the correctness of the code in which no bugs were found).

I found this amazing: only about 10 percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult. In the history in Section 6.2.1 of his *Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.

Writing The Program

The key idea of binary search is that we always know that if T is anywhere in $X[1\,..\,N]$, then it must be in a certain range of X. We'll use the shorthand *MustBe*(*range*) to mean that if T is anywhere in the array, then it must be in *range*. With this notation, it's easy to convert the above description of binary search into a program sketch.

```
initialize range to designate X[1..N]
loop
    { invariant: MustBe(range) }
    if range is empty,
        return that T is nowhere in the
        array
    compute M, the middle of the range
    use M as a probe to shrink the range
        if T is found during the
        shrinking process, return its
        position
endloop
```

The crucial part of this program is the *loop invariant*, which is enclosed in {}'s. This is an *assertion* about the program state that is invariantly true at the beginning and end of each iteration of the loop (hence its name); it formalizes the intuitive notion we had above.

We'll now refine the program, making sure that all our actions respect the invariant. The first issue we must face is the representation of *range*: we'll use two indices L and U (for "lower" and "upper") to represent the range $L\,..\,U$. (There are other possible representations for a range, such as its begin-

ning position and its length.) The next step is the initialization; what values should L and U have so that *MustBe*(L, U) is true? The obvious choice is 1 and N: *MustBe*(1, N) says that if T is anywhere in X, then it is in $X[1 \ldots N]$, which is precisely what we know at the beginning of the program. Thus initialization consists of the assignments `L:=1` and `U:=N`.

The next tasks are to check for an empty range and to compute the new midpoint, M. The range $L \ldots U$ is empty if $L > U$, in which case we store the special value 0 in P and terminate the loop, which gives

```
if L>U then
    P:=0; break
```

The break statement terminates the loop denoted by the loop-endloop delimeters. This statement computes M, the midpoint of the range:

```
M := (L + U) div 2
```

The *div* operator implements integer division, so 6 *div* 2 is 3, as is 7 *div* 2. The program is now

```
L:=1; U:=N
loop
    { invariant:  MustBe(L,U) }
    if L>U then
        P:=0; break
    M := (L+U) div 2
    use M as a probe to shrink the range
        L..U.  If T is found during the
        shrinking process, note its
        position and break
  endloop
```

Refining the last three lines in the loop body will involve comparing T and $X[M]$ and taking appropriate action to maintain the invariant. Thus the code will have the form

```
case
    X[M] < T:  Action a
    X[M] = T:  Action b
    X[M] > T:  Action c
```

Action *b* is easy: we know that T is in position M, so we set P to M and break the loop. Because the other two cases are symmetric, we'll focus on the first and trust that the last will follow by symmetry (this is part of the reason we'll verify the code precisely in the next section).

If $X[M] < T$, then we know that $X[1] \le X[2] \le \cdots \le X[M] < T$, so T can't be anywhere in $X[1 \ldots M]$. Combining this with the knowledge that T must be in $X[L \ldots U]$, we know that if it is anywhere, then it must be in $X[M + 1 \ldots U]$, which we write as *MustBe*($M + 1$, U). Given this, how can we reestablish the invariant *MustBe*(L, U)? The answer is obvious: set L to $M + 1$. Putting these cases together with the previous pseudocode gives

```
L:=1; U:=N
loop
    { MustBe(L,U) }
    if L>U then
        P:=0; break
    M := (L+U) div 2
    case
        X[M] < T:  L:=M+1
        X[M] = T:  P:=M; break
        X[M] > T:  U:=M-1
  endloop
```

It's a short program: ten lines of code and one invariant assertion. The basic techniques of program verification—stating the invariant precisely and keeping eye towards maintaining the invariant as we wrote each line of code—helped us greatly as we converted the algorithm sketch into pseudocode. This development gives us some confidence that the program is correct, but we are by no means certain of its correctness. Spend a few minutes convincing yourself that the code is correct before reading on.

Understanding the Program

When I face a subtle problem, I try to derive a program at about the level of detail we just saw. I then use verification methods to increase my confidence that it is correct. In this section we'll study such an argument for the above binary search at a picky level of detail—in practice I'd go through a much less formal analysis. The program in Figure 1 is (too) heavily annotated with assertions that formalize the intuitive notions we used as we developed the code.

While the development of the code was top-down (starting with the general idea and refining it to individual lines of code), this analysis of correctness will be bottom-up: we'll start with the individual lines of code, and show how they work together to solve the problem.

Warning—Boring Material Ahead
Skip to Next Section When Drowsiness Strikes

It's easy to verify lines 1 through 3. The assertion in line 1 is true by the definition of *MustBe* (if T is anywhere in the array, then it must be in $X[1 \ldots N]$). The assignment in line 2 therefore gives the assertion in line 3.

We come now to the heart of the program: the loop in lines 4 through 28. There will be three parts to our argument for its correctness, each of which is closely related to the loop invariant:

Initialization. The loop invariant is true when execution of the loop begins.
Preservation. If the invariant holds at the beginning of an iteration and the loop body is executed again, then the invariant will still be true.
Termination. Upon termination of the loop, the desired result will hold (in this case, the desired result is that P has the correct value). Showing this will use the facts established by the invariant.

Initialization is easy: the assertion in line 3 is the same as that in line 5. To establish the other two properties, we will reason from line 5 through to line 27. When we discuss lines 9 and 21 (the break statements) we will establish termination properties, and if we make it all the way to line 27, we will have established preservation, because line 27 is the same as line 5.

If the test in line 6 is successful, we know the assertion of line 7: if T is anywhere in the array then it must be between L and U, and the success of the test means that $L > U$. Reasoning from that gives the assertion in line 8: T is nowhere in the array. Thus we correctly terminate the loop in line 9 after setting P to zero.

On the other hand, if the test in line 6 is unsuccessful, we come to line 10. The invariant still holds (we've done nothing to change it), and because the test failed we also know that $L \le U$. Line 11 sets M to the average of L and U, truncated down to the nearest integer. Because the average is always between the two values, we have the assertion of line 12.

This brings us to the case statement in lines 12 through 27; the analysis of the statement will involve considering each of

```
 1. { MustBe(1,N) }
 2. L := 1; U := N
 3. { MustBe(L,U) }
 4. loop
 5.     { MustBe(L,U) }
 6.     if L>U then
 7.         { L>U and MustBe(L,U) }
 8.         { T is nowhere in the array }
 9.         P := 0; break
10.     { MustBe(L,U) and L<=U }
11.     M := (L+U) div 2
12.     { MustBe(L,U) and L<=M<=U }
13.     case
14.       X[M] < T:
15.         { MustBe(L,U) and CantBe(1,M) }
16.         { MustBe(M+1,U) }
17.         L := M+1
18.         { MustBe(L,U) }
19.       X[M] = T:
20.         { X[M] = T }
21.         P := M; break
22.       X[M] > T:
23.         { MustBe(L,U) and CantBe(M,N) }
24.         { MustBe(L,M-1) }
25.         U := M-1
26.         { MustBe(L, U)}
27.     { MustBe(L,U) }
28.   endloop
```

FIGURE 1. A binary search program annotated with assertions.

its three possible choices. The easiest choice to analyze is the second alternative, in line 19. In that case we know the assertion in line 20, so we are correct in setting P to M and terminating the loop. This is the second of two places where the loop is terminated, and both end it correctly, so we have established the termination correctness of the loop.

We come now to the two symmetric branches of the case statement; because we concentrated on the first branch as we developed the code, we'll turn our attention now to lines 22 through 26. Consider the assertion in line 23; the first clause is the invariant, which the program has not altered. The second clause is true because $T < X[M] \leq X[M+1] \leq \cdots \leq X[N]$, so we know that T can't be anywhere in the array above position $M-1$; this is expressed in the assertion with the shorthand *CantBe*(M, N). But logic tells us that if T must be between L and U and can't be at or above M, then it must be between L and $M-1$ (if it is anywhere in X); hence line 24. Execution of line 25 with line 24 true leaves line 26 true—that is the definition of assignment. Thus this choice of the case statement re-establishes the invariant in line 27.

The argument for lines 14 through 18 has exactly the same form. Line 15 follows from line 14 and the sortedness of the array; it in turn implies line 16. The assignment in line 17 re-establishes the invariant in line 18. We've thus analyzed all three choices of the case statement. One correctly terminates the loop, and the other two maintain the invariant.

This completes one part of the analysis of the code: we've shown that if the loop terminates, then it does so with the correct value in P. It may still, however, have a bug: it might never halt. Indeed, this was the most common error in the programs written by the professional programmers.

Our halting proof will use a different aspect of the range $L \ldots U$. That range is initially a certain finite size (N), and lines 6 through 9 ensure that the loop terminates when the range contains less than one element. Thus to prove termination we show that the range shrinks during each iteration of the loop. But that is easy by combining lines 12, 17 and 25. Line 12 tells us that M is indeed within the current range. If lines 14 through 18 are executed, then line 17 ensures that the bottom of the range will be increased by at least one. Likewise, if lines 22 through 26 are executed, then line 25 will decrease the top of the range by at least one. In both cases in which the loop continues, the range is decreased by at least one; the program must therefore halt.

Implementing the Program

So far we've worked with the program in a high-level pseudo-language; our willingness to invent a new control structure allowed us to ignore the details of any particular implementation language and to focus on the heart of the problem. Eventually, however, we have to write the program in a real language. Just so you don't think that I chose the language to make the task easy, I implemented binary search in BASIC. Although that language is fine for some tasks, its paucity of control structures and its global (and typically short) variable names are substantial barriers to building real programs.

Even with these problems, it was easy to translate the above pseudocode into the subroutine in a BASIC dialect shown in Figure 2. Because I translated this program from the carefully verified pseudocode, I had good reason to believe that it is correct. Before I would use it in an application,

```
1000 '
1010 ' BINARY SEARCH FOR T IN X(1..N)
1020 '   PRE:  X(1..N) IS SORTED IN NONDECREASING ORDER
1030 '   POST: P=0  =>  T IS NOT IN X(1..N)
1040 '         P>0  =>  P<N+1 AND X(P)=T
1045 '   SIDE EFFECTS: L, U AND M ARE ALTERED
1050 '
1060 L=1: U=N
1070 ' MAIN LOOP
1080     ' INVARIANT: IF T IS ANYWHERE IN THE ARRAY,
1090     '            THEN IT MUST BE BETWEEN L AND U
1100     IF L>U THEN P=0: RETURN
1110     M=CINT((L+U)/2)
1120     IF X(M)<T THEN L=M+1: GOTO 1070
1130     IF X(M)>T THEN U=M-1: GOTO 1070
1140     '  X(M)=T
1150                     P=M:   RETURN
```

FIGURE 2. Binary search in a BASIC dialect.

however, I would test it on sample data. I therefore wrote a test program in about 25 lines of BASIC. After declaring an $(N + 2)$-element array (indexed from 0 to $N + 1$), the program initialized $X[I]$ to I. It then performed N searches for the elements 1 through N and checked that each returned $P = I$. After that, it performed $N + 1$ unsuccessful searches for $0.5, 1.5, \ldots, N + 0.5$ and checked that each returned $P = 0$. Finally, it searched for 0 and $N + 1$ and checked that they returned $P = 0$. When I ran these tests for values of N from 0 to 10 (inclusive), the first version of the program passed them all.

These tests poke around most of the program. They test every possible position for successful and unsuccessful searches, and the case that an element is in the array but outside the search bounds. Testing N from 0 to 10 covers the empty array, common sizes for bugs (one, two and three), several powers of two, and many numbers one away from a power of two. That testing would have been dreadfully boring (and therefore probably erroneous) by hand, but it used an insignificant amount of computer time.

Many factors contribute to my opinion that the BASIC program is correct: I used sound principles to derive the pseudocode; I used analytic techniques to "verify" its correctness; and then I let a computer do what it's good at and bombard the program with test cases.

Principles of Program Verification

This exercise displays many strengths of program verification: the problem is important and requires careful code, the development of the program is guided by verification ideas, and the analysis of correctness employs general tools. The primary weakness of this exercise is its level of detail; in practice we could argue at a more informal level. Fortunately, the details illustrate a number of general principles, including the following:

Assertions. The relations among input, program variables, and output describe the "size" of a program; assertions allow a programmer to enunciate those relations precisely. Their integral role throughout a program's life is discussed in the next section.

Sequential Control Structures. The simplest structure to control a program is of the form "do this statement then that statement." We understand such structures by placing assertions between them and analyzing each step of the program's progress individually.

Selection Control Structures. These structures include `if` and `case` statements of various forms; during execution, one of many choices is selected. We show the correctness of such a structure by considering each of the several choices individually. The fact that a certain choice is selected allows us to make an assertion in the proof; if we execute the statement following `if I > J`, for instance, then we can use the fact that `I > J` to derive the next relevant assertion.

Iteration Control Structures. Arguing the correctness of loops has three phases: initialization, preservation, and termination. We first argue that the loop invariant is established by initialization, and then show that each iteration maintains its truth. These two steps show by mathematical induction that the invariant is true before and after each iteration of the loop. The third step is to argue that whenever execution of the loop terminates, the desired result is true. These together establish that if the loop ever halts, then it does so correctly; we must prove termination by other means (the halting proof of binary search used a typical argument).

Subroutines. To verify a subroutine, we first state its purpose by two assertions. Its *precondition* is the state that must be true before it is called, and its *postcondition* is what the routine will guarantee on termination (see the BASIC binary search for examples). These conditions are more a contract than a statement of fact: they say that if the routine is called with the preconditions satisfied, then the routine will assume the burden of establishing the postcondition. After I prove once that the body of the routine satisfies the conditions, I can use the stated relations between the pre- and post-conditions without ever again considering its implementation.

The Role of Program Verification

In teaching verification techniques to professionals, I've observed that when one programmer tries to convince another that a piece of code is correct, the primary tool is the test case: execute the program by hand on a certain input. That's a powerful tool: it's good for detecting bugs, easy to use, and well understood. It is clear, however, that programmers have a deeper understanding of programs—if they didn't, they could never write them in the first place. One of the major benefits of program verification is that it gives programmers a language in which they can express that understanding.

The language is first used as code is developed. The programmer should be able to explain every line of code as it is written. The important explanations end up in the program text as assertions; deciding which assertions to include is an art that comes only with practice. Some languages provide an `assert` statement that allows the programmer to write the assertions as logical expressions that are tested at run time; if a false assertion is encountered, then it is reported and the run is terminated (most systems allow assertion checking to be turned off if it is too costly in run time).

The language of verification is also used often after the code has been written. During debugging, violations of the `assert` statements lead us to bugs, and examining the form of a violation shows us how to remove the bug without introducing another. The verification techniques formalize code walkthrough procedures. Assertions are crucial during maintenance of a program; when you pick up code that you've never seen before, and no one else has looked at for years, assertions about the program state can give invaluable insight.

I mentioned before that these techniques are only a small part of writing correct programs; keeping the code simple is usually the key to correctness. On the other hand, several professional programmers familiar with these techniques have related to me an experience that is too common in my own programming: when they construct a program, the "hard" parts work the first time, while the bugs are in the "easy" parts. When they came to a hard part, they hunkered down and successfully used powerful formal techniques. In the easy parts, though, they returned to their old ways of programming, with the old results. I wouldn't have believed this phenomenon until it happened to me; it's good motivation to use the techniques frequently.

Problems

1. As laborious as our proof of binary search was, it is still unfinished by some standards. How would you prove that the program is free of runtime errors (such as division by zero, word overflow, or array indices out of bounds)? If you have a background in discrete mathematics, can you formalize the proof in a logical system?
2. If the original binary search was too easy for you, try the variant that returns in P the first occurrence of T in the

array X (if there are multiple occurrences of T, our original algorithm returns an arbitrary one). Your code should make a logarithmic number of comparisons of array elements; it is possible to do the job in $\log_2 N$ such comparisons. [Hint: work from a precise invariant.]

3. Try the verification techniques on programs of your own (try the most subtle code you've written recently). Specify the assertions precisely, and then argue the transitions from assertion to assertion with a colleague.
4. David Gries calls this the "Coffee Can Problem" in his *Science of Programming*. You are initially given a large pile of "extra" black beans and a coffee can that contains some black beans and some white beans. You then repeat the following process until there is a single bean left in the can.
   Randomly select two beans. If they are the same color, throw them both out and insert an extra black bean. If they are different colors, return the white bean to the can and throw out the black.
   What can you say about the color of the final remaining bean as a function of the numbers of black and white beans originally in the can? [Hint: look for an invariant preserved by the process, and then relate the initial condition of the can to its terminal condition.]
5. A colleague faced the following problem in a program to draw lines on a bitmapped display. An array of N pairs of reals (a_i, b_i) defined the N lines $y_i = a_i x + b_i$. He knew that the lines were ordered in the x-range $[0, 1]$ in the sense that $y_i < y_{i+1}$ for all values of i between 1 and $N - 1$ and all values of x in $[0, 1]$ (thus the lines could be viewed as crooked rungs on a ladder). Given a point (x, y), where $0 \le x \le 1$, he wanted to determine the two lines that bracket the point. How could he solve the problem quickly?
6. [*Practice for the next column*]. We saw in September 1983 that binary search is fundamentally faster than sequential search: to search an N-element table, it makes roughly $\log_2 N$ comparisons while sequential search makes roughly $N/2$. While that difference is often enough for a particular program, in a few cases binary search must be made faster yet. Although you can't reduce the logarithmic number of comparisons made by the algorithm, can you rewrite binary search to lead to faster code? For definiteness, assume that you want to search a sorted table of $N = 1000$ integers.

Solutions to Old Problems

1. Each entry in a tax table contains three values: the lower bound for this bracket, the base tax, and the rate at which income over the lower bound is taxed. Including a final "sentinel" entry in the table with an "infinite" lower bound will make the program easier to write.
2. Use two arrays to represent the coefficients of the recurrence and the k previous values; the program consists of a loop within a loop.
3. Each line of a block letter can be represented by a sequence of integers telling the number of "black" spaces and then the number of "white" spaces; many lines will have the form white-black-white, though some letters will have more alterations (consider "W"). It might be profitable to encode also the number of times a particular line is to be repeated.
4. Only one routine need be written from scratch; the other two can use that as a subroutine. The routine for computing the number of days between two pairs of dates computes the number of each day in its respective year, subtracts the earlier from the later (perhaps borrowing from the year), and then adds 365 times the difference in years plus one for each leap year. The routine to compute the day of the week for a given day computes the number of days between the given day and a known Sunday, and then uses modular arithmetic to convert that to a day of the week. To prepare a calendar for a month in a given year we need to know how many days there are in the month and the day of the week on which the 1st falls.
5. Because the comparisons take place from the right to the left of the word, it will probably pay to store the words in reverse (right-to-left) order. Possible representations of a sequence of suffixes include a two-dimensional array of characters (which is usually wasteful), a single array of characters with the suffixes separated by a break character, and such a character array augmented with an array of pointers, one to each word.

Industrial-Strength Program Verification

The verification techniques in this column can have an immediate impact on any programmer: carefully specify the input and output conditions of every module you write, and then use the informal tools to develop the code and "verify" its correctness. Remember that verification is only one of many activities to ensure that you deliver correct, robust code. If you read a book like the one described in the section on further reading, it's bound to increase the quality of the code you write.

Harlan Mills describes the impact that verification methodologies have had on the Federal Systems Division of IBM in a special issue of the *IBM Systems Journal* devoted to software development (Volume 19, Number 4, 1980). Verification is a substantial part of a course required of all programmers in the division; the course is based on the book *Structured Programming* by Linger, Mills and Witt (published in 1979 by Addison-Wesley). In his article, Mills describes how methodologies based on verification have played an important role in the division's timely delivery of quality software. The projects are substantial: one project he describes delivered three million words of code and data (developed with 200 staff-years) on time and under budget. For more details on this effort and others within IBM, see that issue of the *Systems Journal*.

Although they are not yet ready to be used in a production environment, I believe that program verification systems may soon assist the development of certain kinds of software. Excellent research in this area is under way at a number of research centers, including Cornell University, the University of Southern California's Information Sciences Institute, Stanford University, and the University of Texas at Austin. The Gypsy system developed at Austin by a team led by Don Good is typical of this research.

Gypsy is a methodology for specifying, implementing, and proving the correctness of programs. At its heart is the Gypsy Verification Environment, which provides a set of tools for applying the methodology to the construction of programs. The programmer writes the specifications and the code itself; the system keeps track of the various aspects of the software (specifications, code and proof) and helps out in proving most theorems. Gypsy has been used to develop two substantial programs: a "message flow modulator" that filters out illegal messages in the flow from one machine to another (556 executable lines of code) and an interface to a computer network (4211 lines of code that are executed in parallel on two computers). Both programs have been extensively tested, and the process found no bugs.

Those facts must be understood in context. First, only the smaller program was proved "totally correct"; the verification of the larger showed only that it had certain properties (such as never passing along an inappropriate message). That program might still fail in some other way, but the proof shows that certain mistakes won't be made. The second piece of bad news is the cost: the productivity was only a few lines of code per programmer per day (two on the small program, four on the large program). Further research should increase the productivity, but even this high a cost may be acceptable in high-security and life-critical applications. I'm optimistic about the promise of program verification for such applications; to decide for yourself, try the paper "The Proof of a Distributed System in Gypsy" by Don Good (Institute for Computer Science Technical Report 30, University of Texas at Austin, September 1982). It is among the best that verification currently has to offer, and it's an excellent piece of engineering.

Further Reading

The notion of developing a program hand-in-hand with its proof of correctness was championed by E. W. Dijkstra in the early 1970s. David Gries's *Science of Programming* (Springer-Verlag, 1981) is an excellent introduction to the field. It starts with a tutorial on logic, goes on to a formal view of program verification and development, and finally discusses programming in common languages. In this column I've tried to sketch the potential benefits of verification; the only way that most programmers will be able to use verification effectively is to study a book like this.

Which Typewriter Do You Choose?

(The following essay is reprinted from pp. 170–171 of Gries's *Science of Programming* with the kind permission of Springer-Verlag.)

Back in 1867, the typewriter was introduced into the United States. By 1873, the current arrangement of the keys on the typewriter, called the QWERTY keyboard (after the first six letters of the upper key row), was implemented, never to be changed again. At that time typing speed was not important—most people used two fingers anyway. Moreover, the typewriters often jammed, and the most-used letters were arbitrarily distributed in order to reduce speed so jamming wouldn't occur so easily.

Today, millions of excellent, speedy touch-typists use the inefficient QWERTY keyboard, because that is the only one made. Every so often, a new arrangement is designed and tested. The tests show that a good typist can learn the new arrangement in a month or so, and thereafter will type much faster with much less energy and strain. Yet the new keyboard never catches on. Why? Too much is invested in hardware and training. Because of the high cost of changeover, because of inertia, QWERTY remains supreme.

Let's face it: the average programmer is a QWERTY programmer. He is stuck with old notations, like FORTRAN and COBOL. More importantly he has been thinking with two fingers, using the same mental tools that were used at the beginnings of computer science, in the 1940s and 1950s. True, "structured programming" has helped, but even that, by itself, is not enough. To put it simply, the mental tools available to programmers have been inadequate.

The work on developing proof and program hand-in-hand is beginning to show fruit, and it may lead to a more efficient arrangement of the programmer's keyboard. Luckily, the hardware need not change. Mental tools and attitudes are far more important in programming than the notation in which the final program is expressed. For example, one can use the principles and strategies espoused in this book even if the final program has to be in FORTRAN: one programs *into* a language, not *in* it. To be sure, considerably more than one month of education and training will be necessary to wean yourself away from QWERTY programming, for old habits are changed very slowly. Nevertheless, I think it is worthwhile.

Let us now turn to the elucidation of principles and strategies that may help give the QWERTY programmer a new keyboard.